Compiladores

Compiladores

Regina Fedozzi

Dados Internacionais de Catalogação na Publicação (CIP)

Fedozzi, Regina
F294c Compiladores / Regina Fedozzi. – Londrina : Editora e Distribuidora Educacional S.A., 2018.
264 p.

ISBN 978-85-522-1099-3

1. Analises: Léxica, Sintática e Semântica. 2. Código de máquina. 3. Otimização de códigos. I. Fedozzi, Regina. II. Título.

CDD 005.4

Thamiris Mantovani CRB-8/9491

2018
Editora e Distribuidora Educacional S.A.
Avenida Paris, 675 – Parque Residencial João Piza
CEP: 86041-100 — Londrina — PR
e-mail: editora.educacional@kroton.com.br
Homepage: http://www.kroton.com.br/

Sumário

Palavras do autor

A comunicação é um fator chave em qualquer área, portanto, ser assertivo, claro e preciso é crucial nos dias de hoje nesse âmbito, mesmo para computadores. A ponte para melhorar a comunicação entre humanos, que precisavam criar soluções para serem computadas pelas máquinas, foi estabelecer um tradutor, cuja função seria analisar a solução escrita pelo homem, traduzindo-a em uma linguagem mais fácil e que ao mesmo tempo fosse inteligível para o computador. Surgiu, assim, um programa cuja função era traduzir as instruções escritas na linguagem de programação para a linguagem do computador. Um dos objetivos deste curso é entender o funcionamento desse tradutor e seu processo evolutivo, que o tornou complexo, eficiente e, às vezes, transparente aos programadores, confundindo as regras da linguagem com o próprio tradutor, que é denominado compilador, foco desta disciplina.

Neste curso, estudaremos todas as fases envolvidas na construção de um compilador e a sua importância para o projeto, e mostraremos quanto é determinante uma especificação da linguagem formal detalhada e correta para construir um compilador rápido e eficiente. Esse tema é bastante extenso e vai além de um curso de graduação, por isso nossa opção será convidar você a desenvolver a fase de análise do compilador como prática neste curso. Essa é uma área de espectro abrangente, em que você poderá associar os conceitos estudados ao seu dia a dia profissional, além da área cientifica.

As nossas unidades estão organizadas de forma a permitir sua evolução, começando, na Unidade 1, com conceitos de linguagens de programação. Veremos os princípios básicos de todas as linguagens e seus paradigmas, o que é fundamental para você entender e ser capaz de especificar uma linguagem de programação quando for convidado a desenvolvê-la na próxima seção, em que estudaremos os formalismos para a especificação de uma linguagem. Ainda na Unidade 1, estudaremos os tipos de tradutores e a estrutura geral de um compilador. Essa organização facilitará o entendimento das funções e o desenvolvimento do compilador.

Na Unidade 2, vamos avançar e colocar a "mão na massa". Os alicerces teóricos foram estudados na primeira unidade e, na

segunda, estudaremos como especificar e implementar a fase de análise léxica usando geradores, e nos aprofundaremos na análise sintática, estudando árvores de derivações, problemas de ambiguidade e estratégias de recuperação de erros no processo de análise sintática.

Finalmente passaremos para a análise da síntese, na Unidade 3. Apresentaremos a importância da tabela de símbolos, a tradução dirigida pela sintaxe, técnica utilizada para incluir o contexto dos componentes analisados no processo de análise sintática, e a análise semântica. Concluiremos implementando um analisador sintático.

Na Unidade 4, desenvolveremos o tópico sobre a geração do código intermediário, sua otimização e a geração do código alvo, objeto final do compilador. Para encerrar, trataremos de temas relacionados à atualidade para o desenvolvimento de compiladores, o gerenciamento de equipe para o desenvolvimento de compiladores, os tipos de linguagens novas e seus tradutores, encerrando com uma biblioteca de expressões regulares muito utilizada, chamada REGEX.

Convidamos você a estudar, conhecer e mergulhar nesse fascinante mundo dos compiladores, que permitirá uma melhor compreensão das linguagens e do processo de programação dos computadores.

Estrutura e funcionamento de um compilador

Convite ao estudo

Esta primeira unidade propõe introduzir uma visão geral do estudo de compiladores e apresentar as opções de ambientes para seu desenvolvimento, com o objetivo de capacitar o leitor a planejar cada etapa do desenvolvimento.

Você buscou uma nova colocação no mercado e foi aceito em uma *start-up* inovadora para trabalhar no projeto de criação de uma nova linguagem de programação e no respectivo compilador. O projeto é denominado ***NEW Language New Compiler***.

Essa fase inicial de planejamento será o momento de definição das etapas do desenvolvimento do compilador, justificando as fundamentações técnicas que deverão ser utilizadas na definição da nova linguagem e estipulando as opções recomendadas para o desenvolvimento.

A conclusão da primeira fase do seu trabalho na empresa será a elaboração de uma apresentação oral e de um documento escrito expondo como será o desenvolvimento do projeto. O relatório deve servir como documento norteador para os passos do desenvolvimento.

Pronto para o desafio? Como você se sente tendo a oportunidade de ser um idealizador de uma nova linguagem? É o momento de inovar e usar toda a sua criatividade e a competência técnica necessária, que você irá adquirir estudando esta unidade.

Para ajudá-lo nesse novo desafio, na Seção 1, iremos estudar os princípios das linguagens de programação, com

ênfase nas novas tendências que surgem no mercado e suas implicações para construção dos compiladores. Na Seção 2, iremos nos familiarizar com os formalismos das linguagens, estudar a estrutura das linguagens formais, os tipos de linguagens e como especificar as gramáticas estudando a notação *Extended Backus-Naur Form* (EBNF). Na Seção 3, vamos entender a estrutura de um compilador e conhecer as ferramentas e técnicas de desenvolvimento necessárias para o planejamento da construção de um compilador. Pronto para começar?

Seção 1.1

Estrutura de um compilador

Diálogo aberto

Caro aluno, você sabia que as linguagens de programação, diferente das linguagens naturais, têm por objetivo apenas a comunicação de comandos computacionais? Há cinco princípios básicos das linguagens de programação, fundamentais para o projeto de uma linguagem de programação e a consequente aplicação das técnicas formais que torna possível o desenvolvimento do compilador: sintaxe, nomes, tipos, semântica e funções.

O estudo deles é indispensável porque, mesmo que você seja programador em algumas linguagens, um estudante de tecnologia deve ser capacitado para criar novas linguagens futuramente. Veja o caso de sucesso do brasileiro Jose Valim, criador da linguagem de programação ELIXIR, respeitada pela comunidade global e com mais 116 milhões de downloads (RODRIGUES, 2018).

Lembremos que você está nesse caminho em virtude da convocação para trabalhar no projeto *NEW Language New Compiler* e, nessa primeira fase, o objetivo é entender e transmitir aos desenvolvedores o conhecimento dos princípios das linguagens de programação e os diversos tipos de paradigmas.

Imaginemos que a direção da empresa já o alertou sobre um projeto fracassado de implementação, quando o profissional contratado, especialista na linguagem python, não obteve a motivação necessária dos funcionários que trabalhavam no desenvolvimento porque eles pouco conheciam a linguagem e nunca tinham trabalhado com paradigmas funcionais. Você não pode incorrer no mesmo erro, não é mesmo? Portanto, conhecer os princípios e os paradigmas das linguagens é imprescindível para escolher a solução mais adequadas.

Como essa é a primeira etapa do projeto em que você irá trabalhar, a empresa convocou uma reunião com toda a equipe para apresentá-lo. Nela você deverá mostrar que eles irão participar do desenvolvimento de uma nova linguagem de programação,

além de expor os princípios das linguagens, os diversos paradigmas e como é importante adquirir esses conhecimentos para encontrar soluções técnicas adequadas, que facilitarão o desenvolvimento, dispendendo menos esforço e obtendo produtos mais eficientes.

Aproveite esta seção para montar sua apresentação, pois iremos estudar os princípios das linguagens de programação para a construção de um compilador, a importância de compreender as linguagens mais profundamente e, sobretudo, as novas tendências que surgem no mercado e suas implicações para construção dos compiladores.

Pronto para o desafio?

Não pode faltar

Primeiramente, vamos estudar as linguagens de programação com relação à sua evolução, indo das mais simples do ponto de vista da máquina, até as mais complexas, ou seja, as chamadas linguagens de alto nível, essas mais simples para nós humanos!

Nessa linha, temos que as primeiras linguagens de programação foram as linguagens de máquina e a linguagem *assembly*, muito próxima a linguagem de máquina e simples para a máquina, mas difícil para os humanos. Portanto, devido à necessidade de melhorar a comunicação programador-máquina, muitas linguagens foram criadas desde a década de 1940.

Assimile

As **linguagens de alto nível** são linguagens com sintaxe mais fácil de ser compreendida pelo homem e independentes do hardware (máquina). Um programa escrito em uma linguagem de alto nível é denominado código-fonte.

A tradução da linguagem de alto nível para a linguagem de máquina, normalmente, é feita em mais de um passo e o programa que faz essa tradução é denominado **compilador**.

A computação avançou praticamente em todas as áreas: humanas, exatas e biológicas. Com isso, muitas linguagens foram criadas para facilitar o desenvolvimento de soluções específicas de maneira mais rápida. As primeiras linguagens de alto nível foram

as voltadas para as áreas científicas, como o FORTRAN, que surgiu em 1954 e, apesar de pouco utilizada atualmente, em 2008 teve uma nova versão lançada. Outra linguagem científica importante, desenvolvida na década de 1960, foi o ALGOL. Na área comercial, a linguagem mais utilizada inicialmente foi o COBOL, que surgiu em 1958, com a última versão datando de 2002 (TUCKER, 2009; SEBESTA, 2011).

Atualmente, ainda temos muitos sistemas de informações desenvolvidos em COBOL. Entretanto, com o advento da computação voltada para a Web, muitas soluções de aplicações migraram para essa plataforma e linguagens como Perl, PHP, Visual Basic, Java, Python, ou seja, tem suporte a este ambiente por serem orientadas a eventos.

Na área de inteligência artificial (IA), podemos citar a LISP, uma linguagem inovadora para a sua época (1960), que aplica o conceito do cálculo Lambda, sendo a primeira linguagem de paradigma funcional. A LISP evoluiu para a *Common* LISP. Na linha de linguagens para IA, na Europa, no início da década de 70, surgiu a Prolog, baseada na lógica de predicado de primeira ordem (TUCKER, 2009).

Reflita

Tantas linguagens de programação!

Qual a sua análise sobre esta proliferação de linguagens?

Ao reler o histórico apresentado, é possível agrupa-las não apenas em função da área para a qual foi escrita? Será que há outra(s) característica(s) em comum entre algumas dessas linguagens?

Segundo Tucker e Noonam (2009), os princípios de um projeto de linguagem de programação dividem-se em três (3) categorias: sintaxe, nomes e tipos, e semântica. Nesse estudo, vamos considerar nomes e tipos separados para uma melhor compreensão.

A **sintaxe** da linguagem descreve se o programa foi escrito de forma estruturalmente correta, quais palavras, símbolos que podem ser usados e como organizá-los. As regras da sintaxe são especificadas pelo formalismo das gramáticas livres de contexto.

As **regras para definições de nomes** na linguagem compõem

um conjunto de normas a serem seguidas, por exemplo: um nome de variável em C pode começar com traço baixo (*underline*) ou uma letra, os demais símbolos podem ser _ (*underline*), letra ou número.

Os **tipos** existentes em uma linguagem serão os indicativos para o programador implementar as estruturas para armazenar dados e executar cálculos. Há tipos simples e outros mais complexos, como listas, arvores, funções e classes.

A **semântica** em um programa é o efeito que aquele comando tem sobre os valores envolvidos. Por exemplo:

```
a = b ;  // para as variáveis  a   e   b   a semântica deve analisar se os tipos
         // das variáveis a   e   b   são compatíveis com as regras da LP
calcule ( n1, n2 ) ;  /*  é uma chamada para a  função calcule
                          a semântica, aqui, deve analisar se método calcule
                          aceita dois parâmetros, se o tipo dos parâmetros
                          e o tipo do valor retornado são compatíveis.
                      */
```

Exemplificando

Regras de nomes em C:

Variável: começa com _ e letra, os símbolos seguintes podem ser _, letra ou número.

a, **a1** e **endereco_residencial** são exemplos de variáveis válidas.

1a e endereco residencial são exemplos não válidos, pois o primeiro começa com número e o segundo tem espaço.

A linguagem **C** é *case-sensitive*, isto é, a variável saldo é diferente da variável **SALDO**.

Já o **Visual Basic** (VB) não é *case-sensitive*.

Em programa escrito em VB a variável **saldo** e **SALDO**, são as mesmas entidades (iguais).

No breve histórico das linguagens de programação apresentado, observa-se que há um padrão entre algumas linguagens para a resolução de problemas que chamamos de paradigmas de programação. São quatro os paradigmas de programação clássicos: imperativo, orientada a objeto, funcional e lógico. Segundo Tucker e Noonam (2009), eles evoluíram ao longo das últimas três décadas.

Paradigma imperativo: as linguagens imperativas têm as variáveis e os programas armazenados juntos na memória, além

dos comandos e das atribuições, que são usados para cálculos, entradas e saídas. A estrutura e os recursos da linguagem permitem uma transcrição quase direta da solução algorítmica. Exemplos de linguagens de programação imperativas: FORTRAN, COBOL, C, Basic, ALGOL, Pascal, PHP e Java.

Paradigma orientado a objeto: a base para entender o paradigma orientado a objeto está na abstração dos dados e tipos. Pensar em uma solução orientada a objeto é modelar o mundo como peças (objetos) de forma abstrata, sem atribuir valores, e somente depois desenvolver (implementar) a solução, relacionando os diversos objetos e comportamentos. Exemplo de linguagens POO: Smalltalk, C++, C#, Java e Python.

Paradigma funcional: o princípio do padrão funcional nasceu quando A. Church demonstrou com o $cálculo - \lambda$ que sempre existe uma função se o problema for computável. A primeira linguagem funcional, a LISP, surgiu em 1958. Hoje, a demanda das linguagens funcionais está em franco crescimento em virtude das aplicações na área de inteligência artificial, pois facilita a programação para sistemas, baseando-se em regras e processamento de linguagem natural. A linguagem ELIXIR é uma linguagem funcional nova, que em apenas cinco anos de existência obteve a aceitação extraordinária no mercado. A HASKELL é totalmente funcional. Ela surgiu de 1990, sendo recomendada para quem quer começar a desenvolver usando esse paradigma (SÁ, 2006).

Pesquise mais

$Cálculo - \lambda$ (cálculo lambda) - apresentado à comunidade científica na década 1930 por Alonzo Church, trata do estudo das funções recursivas computáveis, sendo fundamental na formalização da teoria das linguagens de programação. Aprofunde mais seu conhecimento sobre o cálculo-λ com a leitura do artigo a seguir:

SILVEIRA, P. **Começando com o cálculo lambda e a programação funcional**. 18 abr. 2011. [S.p; s.l]. Disponível em: <http://blog.caelum.com.br/comecando-com-o-calculo-lambda-e-a-programacao-funcional-de-verdade/>. Acesso em: 8 maio 2018.

Paradigma lógico: a chave para você raciocinar de acordo com o paradigma lógico é desenvolver a solução declarando qual resultado o programa deve alcançar, em vez de como o programa deve alcançar tal resultado. A linguagem Prolog é o melhor exemplo para esse paradigma.

Exemplificando

Vejamos um exemplo em Java, que é uma linguagem multiparadigmas, para percorrer uma lista de dados.

No exemplo, da linha 7 à 9, é usada uma instrução imperativa. Na linha 11 utilizamos o paradigma funcional, e nas linhas 13 a 16 a orientação a objeto com a abstração de dados.

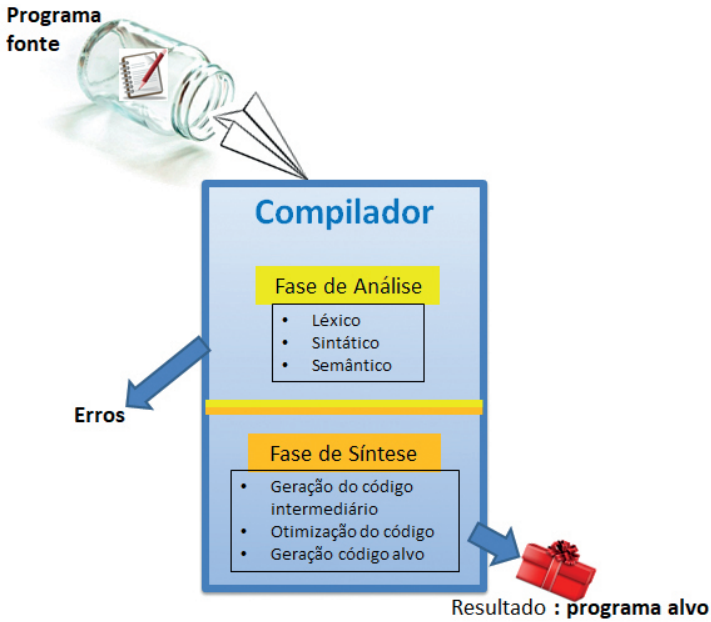
```
01 import java.util.*;
02 public class Exemplo01 {
03     public static void main(String[] args){
04     //declaração dos dados ( lista )
05      List<Integer> dados = Arrays.asList(0,1,2,3,4,5,6,7,8,9,10);
06      //imperativa
07      for( int num : dados ) {
08           System.out.println(num);
09      }
10     // funcional
11       dados.stream().forEach((num) -> {System.out.println(num);});
12     //orientação a objeto
13       for (Iterator<Integer> it = dados.iterator(); it.hasNext();) {
14                                        int num = it.next();
15                                        System.out.println(num);
16        } //fim for(...)
17    }  // fim main()
18  } // fim class Exemplo01
```

Vamos estudar, agora, os fundamentos dos compiladores. Vejamos uma definição formal: "Posto de forma simples, um compilador é um programa que lê um programa escrito numa linguagem – linguagem fonte – e o traduz num programa equivalente numa outra linguagem – a linguagem alvo." (AHO, 2007, p. 1)

De acordo com a definição de Aho (2007) citada, o compilador lê um arquivo para gerar um novo arquivo em uma linguagem diferente da que foi lida. Portanto, há um processo para essa conversão do arquivo lido, que chamamos de código-fonte, para a saída traduzida, que chamados de código-alvo. Esse processo é feito em fases, pelo

compilador, as quais estudaremos detalhadamente durante este curso. Veja-as na Figura 1.1.

Figura 1.1 | Processo de compilação



Fonte: elaborada pelo autor.

A evolução das linguagens de programação está associada aos tipos de tradutores, que, segundo Simão & Prince (2001), podem ser classificados em:

- **Montadores** – programas que traduzem um código fonte escrito em linguagem básica (*assembly*) em código de máquina, também denominado Assembler. Essa tradução normalmente é feita em um ou dois passos.
- **Compiladores** – traduzem o código fonte, programa escrito em uma linguagem de alto nível, em código alvo. Após o processo de compilação, o código alvo não precisa mais passar pelo processo de compilação para ser novamente executado. É um programa complexo e requer várias fases.
- **Interpretadores** – fazem o mesmo processo do compilador, mas cada comando (linha de programa) analisado executa

todas as fases de tradução até alcançar o código alvo e já o executa. As linguagens interpretadas exigem que os programas sejam submetidos ao interpretador todas as vezes que necessitarem ser executadas.

- **Compiladores híbridos** – seguem todos os passos de um compilador, mas não geram o código executável, e sim um código intermediário que será executado por uma máquina virtual. A grande vantagem desse tipo de compilador é a portabilidade.

O foco do nosso estudo são os compiladores, portanto vamos nos aprofundar no estudo das fases do compilador. Quando pensamos nas funções que um compilador deverá executar, há duas divisões bem claras: a fase da análise e a fase de síntese.

A **fase da análise** está diretamente associada à verificação de se o programa foi escrito corretamente, isto é, de acordo com as regras da linguagem. A análise está subdividida em 3 etapas, segundo Aho (2007):

- **Léxica**: em que se verifica se os nomes das entidades estão corretos.
- **Sintática**: analisa-se se os comandos estão corretos. Chama a verificação da frase. Aqui, não basta escrever as palavras corretamente, importando também a ordem em que elas aparecem.
- **Semântica**: nesse ponto, verifica-se o contexto. No caso das linguagens de programação, o compilador deverá analisar se os valores envolvidos nos comandos estão compatíveis.

A fase de análise, portanto, trata da corretude do programa quanto à sua gramática.

A **fase de síntese** visa gerar o código alvo a partir dos resultados obtidos pelas fases intermediárias, que são:

- **Gerenciamento da tabela de símbolos**: as diversas etapas do compilador alimentam e consultam essa tabela para coletar informações (nomes, tipos, atributos).
- **Geração do código intermediário**: a conversão para o código alvo é feita em etapas, e esta é uma conversão intermediária, em que as instruções serão representadas em não mais que três endereços.

Exemplo: r = a + b * raiz(16)
s1 = raiz(16)
s2 = b * s1
s3 = a + s2
r = s3

- **Otimização do código**: realiza transformações no código com o objetivo de melhorar o tempo de execução, o consumo de memória ou o tamanho do código.
- **Geração do código**: essa é a fase final, que gera o código alvo.

Nesta seção estudamos a evolução das linguagens de programação, os princípios comuns a todas as linguagens, os principais paradigmas de programação e como esse conhecimento é fundamental para compreendermos as estruturas das linguagens. Isso dará base para, na próxima seção, compreendermos e especificarmos a gramática das linguagens e aprofundarmos os conhecimentos sobre os tipos de tradutores das linguagens, como os compiladores estão estruturados e, sobretudo, entendermos a organização das fases do processo de compilação.

Sem medo de errar

Lembremos que você foi convocado para trabalhar no projeto ***NEW Language New Compiler*** e, na primeira etapa do projeto, a empresa convocou uma reunião com toda a equipe para apresentá-lo, em que você deverá mostrar que eles irão participar do desenvolvimento de uma nova linguagem de programação. Nessa oportunidade, você deveria expor os princípios das linguagens, os diversos paradigmas e quanto é importante adquirir esses conhecimentos para encontrar soluções técnicas adequadas, que facilitarão o desenvolvimento, dispendendo menos esforço e obtendo produtos mais eficientes.

Com o estudo desta seção, foi possível aprender quais são os princípios das linguagens de programação, os diversos tipos de paradigmas, os tipos de tradutores e as fases do processo de compilação.

Que tal mostrar as estruturas das linguagens? Durante o estudo dessa seção foi possível compreender que todas as linguagens possuem os mesmos princípios e, para criar uma linguagem, é

necessário definir regras de sintaxe, nomes, tipos e semânticas para a especificação da mesma, afinal, esses são os princípios de qualquer linguagem de programação.

Lembre-se de que, realizando uma apresentação clara, com fundamentos técnicos, você conquistará a confiança e motivação da sua equipe. Então, que tal mostrar as vantagens e desvantagens dos estilos de programação? Para tanto, você poderá mostrar que não há só um estilo de linguagem de programação, apresentando os diversos paradigmas de programação e associando-os à linguagem de programação que o representa, por exemplo, se deseja criar uma linguagem para IA, recomenda-se usar linguagens funcionais, como Haskell, ELIXIR, LISP.

O caminho para desenvolver o desafio proposto finalmente foi aberto. Desenvolva uma apresentação, com no máximo 10 slides, expondo por que é necessário conhecer os princípios das linguagens de programação para a construção de um compilador. Não deixe de apresentar quais são esses fundamentos, suas vantagens e desvantagens, concluindo o trabalho mostrando as tendências dos paradigmas das linguagens de programação atual.

É fundamental deixar claras as duas subdivisões do compilador: análise e síntese. Neste primeiro momento, foque nos tópicos estudados nessa seção sobre os princípios das linguagens de programação.

O que não pode faltar: os quatros princípios envolvidos na definição de uma linguagem e a quais fases do compilador estão associados. Relacione também as linguagens de programação existentes e a quais paradigmas estão vinculadas. Mostre que você está alinhado com as novas tendências do mercado, indicando o futuro das linguagens.

Na sua conclusão, recomendamos mostrar a importância de se conhecer os princípios e paradigmas das linguagens para uma boa especificação. Como fazer isso? Lembre-se: você começou com os quatro princípios importantes para definição de uma linguagem e observou que a fase de análise está subdivida em etapas diretamente ligadas aos princípios das linguagens. Ficou mais fácil chegar à conclusão, agora?

Bom trabalho!

Avançando na prática

Analise os tipos de erros de compilação

Descrição da situação-problema

Uma equipe de programadores pretende desenvolver um minicompilador Java, porém não consegue separar os tipos de erros de compilação do programa. Para os programadores dessa equipe, todo erro é, apenas, um erro de compilação.

O processo de análise, na compilação, apesar de integrado, possui três subfases: a léxica, a sintática e a semântica. A primeira fase é a léxica, que analisa se as palavras estão corretas. Caso haja erro de grafia, temos um erro léxico. Caso contrário, passamos para a análise sintática, que corresponde à verificação da estrutura da frase. Caso exista erro na organização da frase, temos um erro sintático. A última analise a ser realizada será analise semântica, quando se analisa se os tipos envolvidos na frase são compatíveis.

Vejamos: diante do recebimento do código a seguir descrito, ajude a equipe de programadores a identificar os tipos de erros de acordo com cada subfase da análise do processo de compilação.

Programa com erros:

```
01 import java.math.*;
02 public class Exemplo02 {
03    public static void main(String args[]){
04        int a, x=0;
05        double v = 123456;
06        byte b = 128;
07        char x-y;
08        if a == b {
09              System.out.println("igual);
10        }
11       for (int i = 9; 10>i ; i--){
12           double raiz = Math.sqrt(i);
13          System.out.println(raiz);
14        }
15        double r = a / x;
16    }
17 }
```

Resolução da situação-problema

Pois bem. Como vimos nesta seção, a análise léxica verifica a grafia dos nomes. Assim, no programa apresentado, na linha '7', temos um erro léxico: 'x-y', correspondente a um nome de variável não válido porque o símbolo '-' não é permitido para nome de variável.

A outra fase da análise é a sintaxe, em que verifica-se se a estrutura dos comandos está correta e, no caso em análise, constatamos que, na linha '8', a estrutura sintática do comando '*if*' está incompleta, faltando os parêntesis, sendo o comando correto '*if* (a==b) {'

Observa-se na linha 9 outro erro de sintaxe, pois falta fechar as aspas: System.out.println("igual");

Na linha '6' o erro semântico, pois o tipo *byte* suporta somente de 0 a 127, enquanto o valor atribuído é superior. Na linha 15 encontramos mais um erro de semântica, pois a variável '**a**' não foi inicializada. Se a tivéssemos inicializado e os erros apontados corrigidos, o programa seria executado, entretanto, surgiriam outros erros. Analise com cuidado a linha '15' e a linha '4'. Veja que 'x' vale zero (atribuição da linha 4) enquanto, na linha 15, temos a divisão 'a / x'. Uma vez que a divisão por zero não existe, haverá nessa linha um erro de execução. É um erro, mas não de compilação. O analisador sintático não aponta erros de execução nem de lógica. Veja como isso é verdade: na linha 11 não há erro sintático, mas esse comando resulta em um loop infinito, um erro do tipo lógico.

Faça valer a pena

1. Entre as linguagens de programação, observa-se a existência de diversos padrões para a resolução de problemas que chamamos de paradigmas de programação. Encontramos, entre elas, quatro paradigmas clássicos: imperativo, orientado a objeto, funcional e lógico.
Esses paradigmas permitem o desenvolvimento de soluções com menor esforço e são mais eficientes quando utilizados adequadamente na área de aplicação para o qual foram pensados. Por exemplo, a área de sistemas de informações adaptou-se bem às linguagens imperativas, e na área de inteligência artificial (IA) temos as linguagens funcionais e lógica, que são adequadas às necessidades dessas soluções. Veja a lista das linguagens e dos paradigmas de programação a seguir:

Linguagens
(A) COBOL
(B) HASKELL
(C) JAVA
(D) PROLOG

Paradigmas
(1) FUNCIONAL
(2) MULTIPARADIGMA
(3) LOGICO
(4) IMPERATIVA

Assinale a alternativa que corresponde ao relacionamento correto entre as linguagens e seu respectivo paradigma.
a) (A) → (1); (B) → (2); (C) → (3); (D) → (4).
b) (A) → (2); (B) → (1); (C) → (4); (D) → (3).
c) (A) → (4); (B) → (2); (C) → (1); (D) → (3).
d) (A) → (4); (B) → (1); (C) → (2); (D) → (3).
e) (A) → (2); (B) → (4); (C) → (1); (D) → (1).

2. Quando pensamos na estrutura de um compilador, verificamos duas funções bem distintas : a função de análise e a de síntese. Cada uma dessas funções é denominada de fase, subdividida e facilita a construção desse programa. Compreender a função de cada uma destas fases é importante para seu o funcionamento e desenvolvimento.
Assinale a alternativa que corresponde a uma sequência lógica das fases do processo de compilação.
a) Análise: semântica, sintática e léxica.
b) Síntese: geração de código → otimização do código.
c) Síntese → Análise.
d) Sintaxe, Semântica, Léxica.
e) Léxica, sintaxe, semântica, geração de código intermediário, otimização do código, geração do código.

3. A fase da análise está diretamente associada à verificação de se o programa foi escrito corretamente, isto é, de acordo com as regras da linguagem. A análise está subdividida em 3 etapas: léxica, sintaxe e semântica. Dessas fases, a mais complexa é a semântica.
A análise semântica é a mais complexa na fase de análise. Assinale a alternativa que corresponde à sua função.
a) A semântica analisa o significado do seu programa.
b) A semântica analisa se cada linha do programa foi escrita de acordo com as regras gramaticas correta da linguagem

c) A semântica analisa se os valores envolvidos nos comandos são compatíveis.
d) A semântica somente é analisada em tempo de execução, pois não é possível analisar o sentido do programa antes de serem atribuídos valores as variáveis.
e) A semântica analisa os erros de lógica de programação, por isso é tão complexa.

Seção 1.2

Fundamentos de linguagens formais

Diálogo aberto

Caro aluno, depois de ter estudado os princípios das linguagens, os fundamentos dos programas de tradução e os compiladores, chegamos ao momento de saber como especificar uma linguagem de programação e entender suas classes.

Você deu o primeiro passo e saiu-se muito bem na exposição do projeto ***NEW Language New Compiler*** para a equipe, obtendo sua confiança e motivando-a, mostrando as vantagens e desvantagens dos estilos de programação, entre outras informações. Nessa nova etapa, espera-se que você possa preparar a equipe para a fase seguinte do processo de criação de uma nova linguagem, na qual, além da apresentação, a direção propôs que você desenvolvesse um tutorial para especificar uma linguagem simples, mas criativa, como a linguagem LOGO, utilizada na área educacional para crianças brasileiras, pois é intenção da empresa atuar nesse ramo de negócio.

Nesse caminho, que tal conhecer um pouco mais da linguagem LOGO? Todas as linguagens têm elementos básicos, portanto, lembre-se de não os deixar de fora do seu exemplo. Qual seria a melhor técnica para especificar uma linguagem de programação? Não se esqueça de fazer a apresentação à equipe sobre a linguagem 'modelo' que será criada e de entregar um tutorial sobre como especificar a gramática para uma linguagem de programação.

Preparado para descobrir como fazer isso? Então, vamos ver os passos para essa descoberta, estudando os conceitos fundamentais das linguagens formais, e saber quais são suas classes, conhecendo a Hierarquia de Chomsky, para que, posteriormente, possamos conhecer como definir formalmente as gramáticas por meio de quádruplas e notação EBNF. Vamos enfrentar esse grande o desafio?

Não pode faltar

Inicialmente, é fundamental compreender como definir uma linguagem formal, estudando os elementos básicos, para que seja possível especificá-la.

Toda linguagem tem origem em um conjunto finito, não vazio de símbolos, denominado **alfabeto**. Quando fazemos a concatenação dos símbolos de um determinado alfabeto, definimos uma **palavra**, e um conjunto de palavras definidas com um alfabeto compõe uma **linguagem**.

Assimile

Alfabeto: são conjuntos finitos e não vazios, representados por letras MAIÚSCULAS. Exemplo: $\Sigma, \Delta, \Omega, A, B, C$

Palavra: é um elemento da linguagem, originado a partir da justaposição (concatenação) dos símbolos de um alfabeto.

Podemos fazer concatenações sucessivas de uma palavra.

Seja o alfabeto $\Sigma = \{0,1\}$ e $w = 101$, uma palavra desse alfabeto, teremos as seguintes concatenações sucessivas, possíveis:

$w^0 = \varepsilon$ → ε (épsilon representa a palavra vazia)

$w^1 = w = 101$

$w^2 = ww = 101101$

$w^3 = www = 101101101$

$w^4 = wwww = 101101101101$

...

$w^n = www...w = 101101101...101$ → Concatenação sucessiva de n vezes

Como você pode observar, uma palavra pode ser concatenada n vezes e, a cada passo, a palavra é repetida ao final da sequência.

Exemplificando

Dado o alfabeto $\Sigma = \{a,b\}$

aba é uma palavra do **alfabeto** Σ, pois é a concatenação dos símbolos 'a', 'b' e 'a', todos contidos em Σ.

É padrão usarmos uma letra minúscula para representar uma palavra, por exemplo, **w = aba**

Logo, se **w = aba**, então ***w*** é uma palavra do alfabeto Σ.

A representação de todas as palavras de um alfabeto Σ, segundo Menezes (2005), é Σ^*, em que:

Σ^* denota o conjunto de todas as palavras possíveis de Σ

Σ^+ denota $\Sigma^+ - \{\varepsilon\}$, em que Σ^+ representa o conjunto de todas as palavras possíveis de Σ, exceto a palavra vazia, representada por $\{\varepsilon\}$.

Assim, podemos concluir que, se **L** é uma linguagem de um alfabeto Σ, então $L \subseteq \Sigma^*$.

Reflita

Se um alfabeto não pode ser vazio e Σ^*, denota o conjunto de todas as palavras possíveis de Σ e se $L = \Sigma^*$.

Nesse caso, é verdade que $L \subseteq \Sigma^*$? E será L finito ou infinito?

Continuando a análise dos conceitos básicos de linguagem, verificamos que muitas possuem algumas regras específicas que as limitam, isto é, restringem algumas palavras, apesar das mesmas serem formadas somente por símbolos pertencentes ao seu alfabeto. O conjunto dessas regras é denominado gramática da linguagem. Veja a definição de gramática que encontramos em Menezes:

> **Uma gramática é basicamente, um conjunto finito de regras as quais, quando aplicada sucessivamente, geram palavras. O conjunto de todas as palavras geradas por uma gramática define a linguagem. (MENEZES, 2005, p. 85)**

De acordo com o exposto, podemos concluir que uma linguagem **L** qualquer é definida pela gramática de um determinado alfabeto Σ, sendo a gramática o conjunto das regras e o alfabeto um conjunto finito, não vazio.

Agora que já sabemos como uma linguagem é definida, e

antes de estudarmos os tipos de gramáticas, vamos compreender como as linguagens são hierarquizadas de acordo com sua complexidade. Essa hierarquização é conhecida como Hierarquia de Chomsky. Que tal conhecermos um pouco mais sobre os estudos de Noam Chomsky?

Noam Chomsky é um reconhecido filósofo, linguista, ativista político e considerado o precursor da linguística moderna. Suas pesquisas agregaram à ciência da computação um importante estudo sobre as linguagens, ao mostrar que, a partir de um conjunto limitado de regras, poderíamos criar um conjunto ilimitado de frases, ou seja, uma linguagem (CHOMSKY, 1980). Com essa teoria, chamada de Gramática Gerativa de N. Chomsky e fundamentada na lógica e no racionalismo, demonstrou que poderíamos ter linguagens, finitas ou infinitas, com apenas um conjunto limitado de regras, cuja definição possibilita criarmos algoritmos capazes de analisarem se as frases escritas estão corretas. E foi além, provando que a maioria das linguagens formais sempre possui uma mesma estrutura, conhecida como Hierarquia de Chomsky.

Assim, a Hierarquia de Chomsky mostra que as linguagens vão das mais simples às mais complexas. A linguagem mais simples é basicamente composta por palavras. Ao começarmos o aprendizado da escrita, um dos primeiros passos é o reconhecimento do alfabeto. Após essa fase, passamos para a grafia das palavras, que nada mais é do que a concatenação (justaposição) dos símbolos do alfabeto. Continuando o processo de escrita, analisam-se as palavras que estão de acordo com as regras definidas pela gramática, o que, nas línguas naturais, costumamos chamar de grafia correta. A esse nível da linguagem, Chomsky associou o nível 3 da hierarquização e o denominou Linguagem Regular.

Após a grafia correta das palavras, avançamos um pouco mais no domínio de uma linguagem, construindo frases em que a ordem das palavras deve ser analisada, análise essa chamada de análise sintática da frase. Noam foi capaz de demonstrar que isso é um padrão na maioria das linguagens formais e designou esse nível como 2 , chamado Linguagens Livre de Contexto.

Ao chegarmos ao nível 2 de uma linguagem formal, somos capazes de escrever uma 'redação' e, se associarmos essa analogia às linguagens de programação, poderíamos dizer que um programa

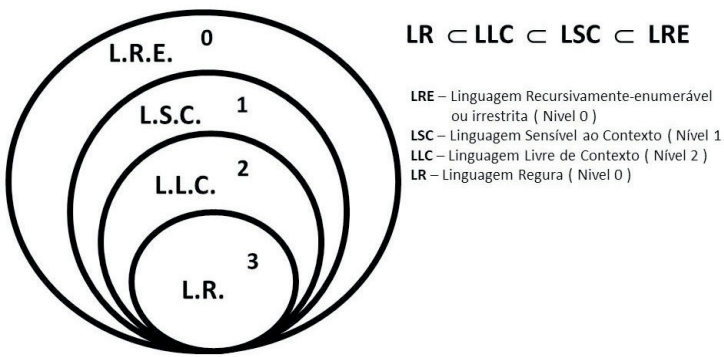
é uma 'redação'. Afinal, um programa de computador nada mais é que um conjunto de comandos (frases) sintaticamente corretos.

Assim, as linguagens mais simples são compostas apenas por palavras e possuem regras para a grafia, pertencendo à classe das linguagens regulares (nível 3). Já com a classe das linguagens livres de contexto, seremos capazes de verificar a sintaxe dos comandos (frases).

Portanto, o trabalho desenvolvido pela teoria gerativista, pela Hierarquia de Chomsky e pela tese de Turing-Church garantiu aos futuros projetistas de linguagens e compiladores a possibilidade de criar novas linguagens e desenvolver programas que analisem suas sintaxes, sabendo se estão escritos de acordo com o conjunto de regras gramaticais definido.

A maioria das linguagens de programação pode ser desenvolvida com as linguagens nível 2, não sendo necessário alçarmos voo para as de nível 1 e 0, respectivamente Linguagens Sensíveis ao Contexto (LSC) e as Linguagens Recursivamente Enumeráveis (LRE). A Hierarquia de Chomsky define as estruturas da linguagem como um processo crescente entre os tipos de linguagem, sendo: LR ⊂ LLC ⊂ LSC ⊂ LRE, veja a Figura 1.2.

Figura 1.2 | Gráfico representativo da Hierarquia de Chomsky



Fonte: elaborada pela autora.

As Linguagens Sensíveis ao Contexto (LSC), designadas como nível 1, geram especificações mais longas e complexas, o que dificulta sua utilização prática, isto é, sua implementação. Logo,

quando criamos especificações de gramáticas com o objetivo de construir compiladores, na prática, adotamos a LLC e utilizamos a análise dirigida por sintaxe para a implementação da verificação das dependências de contexto no processo de compilação. Como vimos na Seção 1.1, essa verificação consiste na análise semântica.

As linguagens nível 0 são as Linguagens Recursivamente Enumeráveis (LRE). Esse nível, o zero (0), abrange todos os tipos de linguagem e é conhecido como estrutura de frases ou irrestrita.

Assimile

A Hierarquia de Chomsky nos diz que:

$$\mathbf{LR} \subset \mathbf{LLC} \subset \mathbf{LSC} \subset \mathbf{LRE}$$

Isso significa que as LRs estão contidas nas LLCs, que, por sua vez, estão contidas nas LSCs e, consequentemente, todas essas linguagens pertencem ao conjunto da LREs.

É importante saber que para cada para tipo de linguagem há uma gramática correspondente que a gera. Veja o quadro a seguir:

Quadro 1.1 | Tipos de Linguagens e a gramática correspondente

TIPOS DE LINGUAGENS	GRAMÁTICA CORRESPONDENTE
(3) LR – regulares	Gramática Regular
(2) LLC – livre de contexto	Gramática Livre de Contexto
(1) LSC – Sensível ao contexto	Gramática Sensível ao Contexto
(0) LRE – Recursivamente Enumerável	Gramática Irrestrita

Fonte: elaborado pela autora.

Segundo Delamaro (2004), quase sempre as Linguagens de Programação (LPs) pertencem à classe das Linguagens Livres de Contexto (LLC), e uma LLC é uma quádrupla do tipo: G = (V, T, P, S), em que:

V: é o conjunto dos símbolos NÃO-TERMINAIS, e $V \neq \varnothing$.

T: é o alfabeto em que a linguagem é definida; podemos dizer que T também é o conjunto dos símbolos terminais.

P: é o conjunto de produções da forma $A \rightarrow \alpha$, em que $A \in V$ e $\alpha \in (T \cup V)^*$; lembre-se de que o asterisco indica todas as

combinações possíveis sobre $(T \cup V)$.

$A \rightarrow \alpha$, onde $A \in V$ e $\alpha \in (T \cup V)^*$.

Explicando: **A** é uma regra (produção) presente no conjunto **V** (conjunto das produções), gerando a sentença α, que **pertence a uma das possibilidades de combinações** de símbolos do alfabeto T, de acordo com as demais normas definidas na gramática (conjunto V), seja $(T \cup V)^*$.

S: é o símbolo NÃO-TERMINAL inicial da gramática. S ∈ V.

Exemplificando

Vamos tomar, por exemplo, a regra para nome de variáveis na linguagem C++:

O nome de uma variável em C++ pode começar com uma letra ou '_' (*underline*), e os demais símbolos podem ser letras, números ou '_'.

Vamos criar a gramática para esta linguagem de nome de variáveis em C++?

Seja L essa linguagem, diremos que L(G) é a linguagem L gerada pela gramática G. Vamos, portanto definir G.

G = (V, T, P, S), em que sabemos que:

T = {a,b,c,d,...z,0,1,2,3...,9,_} são os símbolos (alfabeto) que podem pertencer ao nome de uma variável.

S será nossa produção inicial.

Vamos ao segundo passo, que é o desenvolvimento da 'lógica' das produções. Assim temos:

P = { S → L | LA ; **(a)**

A → C | CA ; **(c)**

L → a | b | c |...| z | _ ; **(b)**

C → a | b | c |...| z | _ | 0 | 1 | 2 | ... | 9 } **(d)**

Agora, vamos entender a lógica apresentada no conjunto P exposto, que gera palavras válidas para representar nomes de variáveis:

a) L → a | b | c |...| z | _

'L' é uma produção que representa os símbolos terminais 'a' ou 'b' ou 'c', ou seja, letras ou o símbolo '_'(*underline*). O '|' (*pipe*) representa o 'ou lógico' em produções gramaticais.

b) $S \rightarrow L \mid LA$

'S' é a produção inicial da gramática, indicando que a palavra se inicia por símbolos contidos na produção 'L', pois, se o comprimento da palavra é um segue para a produção 'L' ou, caso seja maior que A, um irá para 'LA', sendo que 'L' analisa o primeiro símbolo e os demais serão analisados na produção 'A'.

c) $A \rightarrow C \mid CA$

A produção 'A' indica um processo recursivo para analisar os demais caracteres da palavra, como a seguir: caso o comprimento restante seja um, segue para a produção 'C' ou, caso seja maior que um, irá para 'CA', portanto, 'C' analisa o símbolo atual e os demais serão analisados novamente pela produção 'A'. O processo se repetirá até concluirmos a análise de todos os caracteres da palavra, um a um, pois a produção 'A' é novamente acionada.

d) $C \rightarrow a \mid b \mid c \mid ... \mid z \mid \_ \mid 0 \mid 1 \mid 2 \mid ... \mid 9$

A produção 'C' representará os símbolos permitidos para o segundo, e os demais caracteres, se houver, das palavras geradas pela nossa gramática, ou seja os nomes de variáveis.

Chegamos ao terceiro e último passo, está determinado o conjunto V = { S, A, L, C }, e concluída a gramática G.

Tendo em vista que essa forma de especificar a gramática não era muito legível e prática, dois pesquisadores, Jonh Backus e Peter Naur, no final dos anos 50, desenvolveram uma notação formal para LLC, denominada BNF (Backus-Naur Forms). Segundo Tucker (2009), a BNF foi adaptada da teoria de Chomsky, tornando-se um sinônimo para gramática livre de contexto, e foi generalizada para especificação das linguagens de programação. Basicamente, a notação BNF é uma metalinguagem, ou seja, uma linguagem para definir outra linguagem. Desse modo, a BNF pode definir formalmente uma gramática e tem os mesmos princípios encontrados na quádrupla da gramática de Chomsky, mas usando símbolos diferentes. Então, vejamos os meta-símbolos específicos utilizados na notação BNF, segundo Tucker (2009):

- Os não-terminais consistem em nomes escritos entre '<' e '>';
- ::= é usado para representar "definido como". Pode ser usado ':=';

- | é usado para indicar alternativa, é o "ou" lógico.

Que tal desenvolvermos, usando a notação BNF, o caso da gramática para nome de variáveis em C++ ? Vejamos como ficará:

Quadro 1.2 | Gramática para nome de variáveis em C++, usando notação BNF

Solução	Comentários
<*var*> ::= <*inicio*> | <*var*> <*seguinte*>	<var> analisa caractere a caractere da palavra escrita, da direita para a esquerda. Por exemplo, seja 'n1' o nome da variável, então a produção <*var*> indica:
	Passo 1: pega o símbolo mais à direita, '1', e envia para ser analisado na produção <*seguinte*>, e retorna 'n' para <*var*>.
	Passo 2: 'n' será analisado na produção <*inicio*>, pois o comprimento restante é igual a um.
<*inicio*> ::= _ | <*letra*>	<*inicio*> reconhece o símbolo '_'(*underline*) ou símbolos reconhecidos pela produção <*letra*>
<*seguinte*> ::= _ | <*letra*> | <*digito*>	<*seguinte*> reconhece os símbolos, quer sejam '_'(*underline*), quer sejam <letra> ou ainda <*digito*>
<*letra*> ::= a | b | c | ... | z	<*letra*> produção que representa símbolos terminais, no caso, as letras do alfabeto.
<*digito*> ::= 0 | 1 | 2 | 3 | ... | 9	<*digito*> produção que representa símbolos terminais, no caso, os números de 0 a 9.

Fonte: elaborado pela autora.

No ciclo de vida do padrão BNF ocorreram pequenas alterações e, segundo Tucker: "Diversas variações menores foram introduzidas, mas não afetaram o poder expressivo básico da BNF. Essas variações foram introduzidas principalmente para melhorar a clareza e a concisão das descrições de sintaxe" (TUCKER, 2009, p. 35).

Essas alterações foram compiladas na ISO/IEC 14977:1966 (SCOWEN, 1996), que normalizou uma nova notação, denominada Extended Backus-Naur Forms (EBNF), que é uma extensão da BNF com os seguintes meta-símbolos:

[ ] – Indica uma parte opcional, que pode repetir 0 ou 1 vez;

{ } – Indica uma parte que pode repetir 0 ou n vezes;

( ) – Indica precedências dentro da regra ;

‘ ’ – Indica um caractere a tratar como terminal.

Devido à EBNF ser mais clara e simples que a BNF, usaremos neste curso, preferencialmente, essa notação, lembrando que qualquer linguagem definida em EBNF pode ser convertida para BNF. Assim sendo, vamos reescrever o exemplo feito em BNF, do caso da gramática do nome de variáveis em C++ na notação EBNF:

Quadro 1.3 | Gramática para nome de variáveis em C++, usando notação EBNF

Solução EBNF	Comentários
<*var*> ::= (‘_’ | <*letra*>) { ‘_’ | <*letra*> | <digito> }	Palavra inicia por ‘_’ ou <*letra*>
	Os demais símbolos poderão ser ‘_’ ou <*letra*> ou <*digito*>
	Por estarem entre chaves ‘{ }’ podem repetir 0 ou ***n*** vezes.
<*letra*> ::= ‘a’ | ‘b’ | ‘c’ | ... | ‘z’	<*letra*> representa símbolos terminais, no caso as letras do alfabeto
<*digito*> ::= ‘0’ | ‘1’ | ‘2’ | ‘3’ | ... | ‘9’	<*digito*> produção que representa símbolos terminais, no caso os números de 0 a 9

Fonte: elaborado pela autora.

Nossos estudos nessa seção permitiram compreender como é possível especificar linguagens por meio da construção de um conjunto finito de regras, denominado gramática, e também que as linguagens estão estruturadas em classes, sempre em nível crescente, em que a mais complexa contém as mais simples. Após compreender como definir formalmente a gramática, avançamos estudando dois tipos de notações, a BNF e a EBNF, essa última mais clara e simples e, sobretudo, adequada para especificar linguagens de programação.

Agora que você já estudou a estrutura dos compiladores, conheceu os princípios das linguagens de programação e a forma como podemos especificar sua gramática, o próximo passo será conhecer as ferramentas e as etapas para planejar a construção de um compilador. Motivado para prosseguir?

Sem medo de errar

Você se saiu muito bem na primeira reunião. Sua apresentação foi um sucesso e você conquistou a confiança da equipe e conseguiu de motivá-la. Agora, eles esperam que você apresente a forma como poderão realizar a especificação da gramática para a linguagem LOGO, e a direção espera que o tutorial solicitado seja objetivo e funcional, para que todos os profissionais que participam do projeto *NEW Language New Compiler* sejam capazes de especificar futuras linguagens, pois a cada dia novas oportunidades de negócio estão surgindo e a empresa não deseja perdê-los.

Agora é o momento de elaborar o tutorial para especificar a gramática para a linguagem de programação LOGO. Assim, vamos seguir os seguintes passos:

Passo 1: estudar a nossa linguagem 'modelo', no caso, a linguagem LOGO, consultando o site a seguir: <https://sites.google.com/site/infoeducunirio/perspectiva-construtivista/linguagem-logo>. Acesso em: 14 maio 2018.

Passo 2: como iremos especificar uma linguagem de programação, a melhor técnica será a notação EBNF, como estudamos. Afinal, o objetivo da criação da EBNF era simplificar as especificações das linguagens de programação, como é o caso da linguagem LOGO.

Passo 3: A partir da apostila *Brincando com o LOGO*, presente no material sugerido no primeiro passo, podemos levantar os elementos básicos da gramática, tais como regra do nome das variáveis, tipos dos dados, instruções de controle, além da indicação de início e fim de programa. Sendo assim, encontramos as seguintes instruções:

1. parafrente n, pf n, paratrás n, pt n, paradireita n, pd n, paraesquerda n, pe n
2. tartarga, usenada, uselapis, useborracha, desapareçatat, apareçatat
3. mudeel[1 n]
4. mudecl[n n n]
5. mudecp n
6. pinte
7. repita n [<lista de comandos>]

8. aprenda <nome do programa> <lista de comandos> fim
9. atribua "<nomeVar> n

Ao analisar as instruções listadas, devemos buscar identificar os elementos básicos, tais como a declaração de variáveis (item 9) e as instruções de identificação de tipo de dado, que, nesse caso, não existem. Com relação à regra para nome de variável, sabemos que começa com uma letra e os demais caracteres, se houver, poderão ser letras ou números. E, para concluir, na identificação Dos elementos básicos, observamos que o programa começa e termina pela instrução indicada no item 8.

Antes de começar a especificação do próximo item, que tal apresentar um exemplo, associado ao levantamento realizado nesse passo? Vamos lá?

Quadro 1.4 | Exemplo de um programa em LOGO, que cria um triângulo

Programa em LOGO	Comentários
aprenda triangulo	**aprenda** <nome do programa> <lista de comandos> **fim**
	o valor do <nome do programa> é triangulo
	a <lista de comandos> são as 7 instruções entre aprenda e **fim**
parafrente 100	parafrente n, nesse caso o valor de n é 100
paradireita 120	paradireita n, nesse caso o valor de n é 120
parafrente 100	
paradireita 120	
parafrente 100	
mudecp 4	mudecp n, nesse caso o valor de n é 4
pinte	
fim	fim pertence a instrução aprenda

Fonte: elaborado pela autora.

Passo 4: é o momento de definir a gramática da linguagem LOGO. Conforme especificado no passo 2, usaremos a notação

EBNF. Vamos, então, criar as produções necessárias para gerar as instruções da linguagem listadas no passo 3:

<program> ::= 'aprenda' <nome> [':' <nome>] { <listaComandos> } 'fim'
<nome> ::= <letra> { <letra> | <digito> }
<letra> ::= 'a' | 'b' | 'c' | ... | 'z' | 'A' | 'B' | 'C' | ... | 'Z'
<digito> ::= '0' | '1' | '2' | '3' | ... | '9'
<listaComandos> ::= <atribua> | <cmd1> | <cmd2> | <cmd3> | <cmd4>
<atribua> ::= 'atribua" ' <nome> <numero>
<numero> ::= <digito> { <digito> }
<cmd1> ::= 'pinte' | 'tartarga' | 'usenada' | 'uselapis' | 'useborracha' | 'desapareçatat' | 'apareçatat'
<cmd2> ::= ('parafrente' | 'pf' | 'paratrás' | 'pt' | 'paradireita' | 'pd' | 'paraesquerda' | 'pe' | 'mudecp') (<numero> | ':'<nome>)
<cmd3> ::= 'mudeel [1 ' (<numero> | ':'<nome>) '] '
<cmd4> ::= 'mudecl [' (<numero> | ':'<nome>) (<numero> | ':'<nome>) (<numero> | ':'<nome>) '] '
<cmd4> ::= 'repita' (<numero> | ':'<nome>) { <listaComandos> } '] '

Feita a especificação da gramática, é o momento de elaborar a apresentação para a direção da empresa e os membros da equipe. Você pode usar o PowerPoint para elaborar alguns slides, mas pode também inovar, já que terá que criar um tutorial. Que tal usar o *Prezi*? Você pode saber mais sobre essa ferramenta no site https://prezi.com/pt/ (acesso em: 14 maio 2018).

No seu tutorial, mostre um exemplo de programa em LOGO, como apresentado no passo 3. Que tal construir um novo exemplo? Um quadrado, talvez? Em seguida, apresente as classes de linguagens e a qual classe o LOGO pertence, e, ainda, como uma gramática pode ser definida e as opções existentes para esta definição. Não deixe de mostrar os meta-símbolos da notação BNF e EBNF e, finalmente, sua aplicação, apresentando a especificação da linguagem LOGO usando a notação EBNF, como construído no passo 4.

Como você pôde perceber, todos os pontos estudados nessa seção foram necessários para construir sua especificação e apresentação: A hierarquia de Chomsky, a definição de gramática, as notações BNF e EBNF. Sem essas competências seu desafio não teria sido completado.

Avançando na prática

Padronizando e-mails

Descrição da situação-problema

A empresa em que você está trabalhando foi contratada para analisar o banco de dados do cliente, que apresenta duplicidades e dados inválidos ou fora das normas, o que provoca perdas financeiras e ações de marketing negativas. O objetivo deste trabalho é padronizar as informações presentes em alguns campos, tais como nome, razão social, endereço e e-mail.

Uma equipe será responsável pela verificação da estrutura da base e a importação dos dados para a análise de consistência dos registros. Essa equipe, sabendo do seu conhecimento sobre gramática, pediu que você criasse uma especificação para o reconhecimento de e-mails, assim eles poderão facilmente implementar o algoritmo para fazer a consistência dos dados. Como você faria essa especificação?

Resolução da situação-problema

Este é um problema clássico de linguagens do tipo 3, regular.

O primeiro passo para resolver este problema é estudar a estrutura dos endereços de e-mail.

A estrutura básica do e-mail é: nome-parte-local@dominio.tipo.pais.

O nome-parte-local pode conter:

– Letras maiúsculas e minúsculas do alfabeto inglês. Acentos nem sempre são reconhecidos por todos os servidores.

– Números de 0 a 9.

– Os seguintes caracteres: !, #, $, %, &, -, _, ~.

– O caractere '.' (ponto). Atenção: '.' não pode aparecer no início, nem duas vezes seguidas.

O domínio pode conter:

– Letras maiúsculas e minúsculas do alfabeto inglês.

– Números de 0 a 9.

– O caractere '-' (traço). Atenção: '-' não pode aparecer no início, nem o final.

O Tipo pode conter apenas letras, limitado a 3 caracteres.

Vamos limitar o tipo a 3 caracteres para a resolução do exercício.

O País pode conter apenas letras, limitado a 2 caracteres.

O nome-parte-local e o domínio são obrigatórios, já o tipo e o país são opcionais.

Definidos os símbolos válidos, é possível conhecer o alfabeto e iniciarmos a construção da gramática, neste caso usaremos a notação EBNF para a especificação:

<email> ::= <partelocal>'@'<domínio>['.'<tipo>]['.'<pais>]
<partelocal> ::= <letra>{ <letra> | <digito> | <caracter> | '.'<letra> | '.' <digito> }
<domínio> ::= <letra>{<letra> | <digito> | '-'<letra> | '-'<digito>}
<pais> ::= <letra><letra>
<tipo> ::= <letra><letra><letra>
<letra> ::= 'a' | 'b' | 'c' | ... | 'z' | 'A' | 'B' | 'C' | ... | 'Z'
<digito> ::= '0' | '1' | '2' | '3' | ... | '9'
<caracter> ::= '!' | '#' | '$' | '%' | '&' | '-' | '_' | '~'
Está concluída a especificação.

Não pode faltar

1. O linguista Noam Chomsky, em 1958, publicou a obra *Estruturas Sintáticas*, em que demonstrou que tanto as linguagens formais como a maioria das naturais, possuem uma estrutura hierárquica padrão e estão organizadas em classes, sempre em nível crescente, e a mais complexa contém as mais simples. Devido à relevância desse trabalho para o estudo das linguagens, essa estrutura passou a ser conhecida como Hierarquia de Chomsky.
Marque a alternativa que apresenta as classes de linguagens da Hierarquia de Chomsky:
a) Irrestritas, Regulares, Irregulares e as Sensíveis.
b) Regulares, Irregulares, Livres e Sensíveis.
c) Recursivamente-Enumeráveis, Sensíveis ao Contexto, Regulares e Irregulares.
d) Regulares, Livres de Contexto, Recursivamente-Enumeráveis e Sensíveis ao Contexto.
e) Irregulares, Livres de Contexto, Regulares e Programáveis.

2. A etiqueta de identificação de produtos fabricados por uma empresa possui a seguinte estrutura padrão: xxx.nxnxnx-nome/aa, em que:
- **xxx** são três letras quaisquer do alfabeto inglês.
- **nxnxnx** é um número seguido de uma letra, e, nesse caso, pode ter no máximo 3 pares de nx e no mínimo 1 par.
- **nome** pode ter apenas letras, no mínimo uma, mas sem limite de tamanho.
- **aa** indica o ano, e os valores válidos devem estar entre 18 e 29.

Sabemos que é possível construir uma gramática que gere etiquetas escritas nesse formato.
Dada a produção <codigo> ::= <letra><letra><letra>'.'<parte2>'-'<nome>'/'<ano>, marque a alternativa que representa uma produção que pertence à gramática geradora de etiquetas, no formato xxx.nxnxnx-nome/aa:
a) <parte2> ::= { <digito><letra> }
b) <parte2> ::= <digito><letra> [<digito><letra>] [<digito><letra>]
c) <parte2> ::= [<digito><letra>] [<digito><letra>] [<digito><letra>]
d) <nome> ::= { <letra> }
e) <ano> ::= <digito><digito>

3. As linguagens de programação, na sua maioria, pertencem à classe das Linguagens Livres de Contexto (LLC) e são representadas por uma quádrupla do tipo: G = (V, T, P , S).
De acordo com a quádrupla G = (V, T, P, S) qual alternativa descreve corretamente o significado do elementos dessa quádrupla?
a) V representa o alfabeto da linguagem.
b) T representa a produção final, ou seja, aquela produção na qual a gramática termina.
c) P representa o conjunto dos símbolos das produções.
d) S representa a produção inicial, ou seja, aquela produção na qual a gramática inicia.
e) V representa o conjunto transição, que normalmente é representado por δ.

Seção 1.3

Planejamento da construção de um compilador e a seleção de ferramentas

Diálogo aberto

Prezado aluno, você está iniciando nesta seção o estudo que lhe permitirá planejar o projeto do compilador. Nas duas primeiras seções você teve contato com os conceitos fundamentais das linguagens e aprendeu que as linguagens possuem uma hierarquia, sendo possível agora criar conjuntos de regras finitas, chamados de gramáticas, que possibilitam construir algoritmos para analisar se os programas foram escritos de acordo com as regras definidas. Portanto, como as primeiras seções possibilitaram compreender como definir uma linguagem de programação e o compilador foi estruturado, agora chegou o momento de organizar e planejar o desenvolvimento do compilador.

Iniciaremos nosso estudo conhecendo as ferramentas existentes no mercado que auxiliam na construção de compiladores em cada fase do seu desenvolvimento e, em seguida, estudaremos como planejar as etapas de desenvolvimento e do design da linguagem fonte.

Em continuidade às atividades para as quais você foi contratado na *startup* de tecnologia, a direção necessita de uma proposta de cronograma e saber quais ferramentas serão necessárias para desenvolver o compilador da linguagem a ser criada pela equipe do projeto NEW Language New Compiler, a fim de analisar sua viabilidade financeira e, então, liberar os recursos.

Até aqui tudo correu muito bem nas suas duas apresentações. Os diretores perceberam sua capacidade para disseminar o conhecimento, o que te beneficiou, pois divulgar entre seus colaboradores aquilo que existe de mais moderno e relevante em tecnologias é estratégico para uma empresa orientada à inovação. Assim, eles desejam que você continue suas palestras destinadas à capacitação dos profissionais da empresa, portanto pediram que você apresente um workshop sobre as atuais ferramentas disponíveis

para agilizar a construção de compiladores e como elas poderão ajudar nos futuros projetos a serem desenvolvidos na empresa, além da entrega da proposta solicitada anteriormente.

Para atender às solicitações, nesta seção vamos conhecer quais são as ferramentas de apoio existentes no mercado, tais como JFLEX, CUP, JAVACC, Yacc & Lex, que poderão ajudá-lo no desenvolvimento e planejamento de um compilador. Também iremos analisar as opções de plataforma, tanto para desenvolvimento do compilador como para o código alvo a ser gerado. Essas competências permitirão a você planejar o desenvolvimento dos compiladores.

Assim, ao final desta seção você poderá apresentar seu workshop com um exemplo real de uma linguagem, a especificação da linguagem usando notação EBNF, bem como apresentar os critérios relevantes para a tomada de decisão sobre quais tecnologias são adequadas para serem adotadas no planejamento da implementação de um compilador específico e o cronograma para o desenvolvimento de cada fase. Portanto, agora só falta a última etapa para você concluir seu desafio, o estudo das ferramentas de apoio, dos tipos de implementações de compiladores e dos critérios relevantes para definições de plataformas. Está pronto?

Não pode faltar

Como foi visto anteriormente, um compilador está dividido em duas partes: a análise e a síntese. É bom lembrar que a fase de análise consiste basicamente em verificar se o programa escrito está correto com relação à gramática da linguagem, e isso significa que o algoritmo que fará esta análise deve implementar exatamente as regras da gramática. Logo, como é preciso implementar as especificações da gramática, uma notação direcionada para a construção de algoritmos e para linguagens de programação será bem-vinda, pois isso facilitará o trabalho de desenvolvimento. Você deve se recordar da notação *Extended Backus-Naur Form* (EBNF), estudada na seção anterior. Essa notação foi criada justamente com esse intuito.

Veja como o cerco vai se fechando: a EBNF é uma metalinguagem, ou seja, uma linguagem para criar outra linguagem, então, por

princípio, é uma linguagem. Assim, os cientistas da computação construíram aplicativos que interpretam a especificação EBNF de uma linguagem e automaticamente criam algumas classes (objetos) para reconhecer se a entrada é válida. Esses aplicativos agilizam e sistematizam algumas etapas do projeto do compilador, assim o desenvolvedor pode focar o seu trabalho na construção da gramática e na integração das fases do projeto.

De acordo com o exposto acima, não é preciso começar do zero para criar compiladores nos dias atuais, diferentemente do que acontecia nas décadas de 50 e 60. Portanto, será muito útil conhecer essas ferramentas para apoio no desenvolvimento de um compilador. Ainda assim, resta saber como surgiu o primeiro compilador. Ele foi escrito em código de máquina e era muito trabalhoso desenvolver, mas tinha como vantagem a rapidez e serviu como base para todos os outros, por meio do processo de *bootstraping*. Os compiladores construídos com esta técnica são conhecidos como autocompiláveis, pois em alguma fase da sua construção utilizam a mesma linguagem de programação na qual foram implementados.

Vamos entender como funciona o processo de *bootstraping*, em 3 passos:

1º passo: para a linguagem 'A', escrevemos o compilador na linguagem de máquina 'M'.

2º passo: agora, para a linguagem 'A', escrevemos um outro compilador na própria linguagem 'A'.

3º passo: a saída do 2º passo será entrada para o primeiro compilador, aquele construído no 1º passo.

Seguindo esses passos, tem-se o código de máquina a partir de um compilador feito na própria linguagem. Pelo processo de *bootstrapping*, a partir de dois compiladores é possível escrever um compilador em qualquer linguagem de programação. O importante a se considerar que quanto mais passos forem necessários para alcançar o código alvo, mais demorada será a compilação.

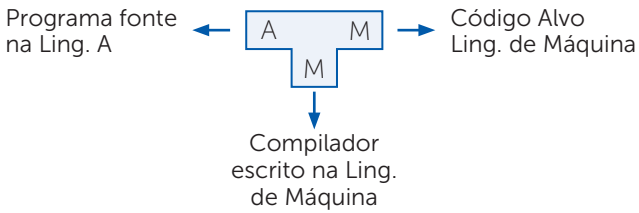
Exemplificando

Bootstrapping – processo utilizado pelos compiladores autocompiláveis, isto é, em algum momento é utilizada a própria linguagem de programação na qual o compilador foi implementado.

Vamos ver um exemplo com o uso de um diagrama em T.

O primeiro compilador para a linguagem 'A' foi escrito em código de máquina (Figura 1.3).

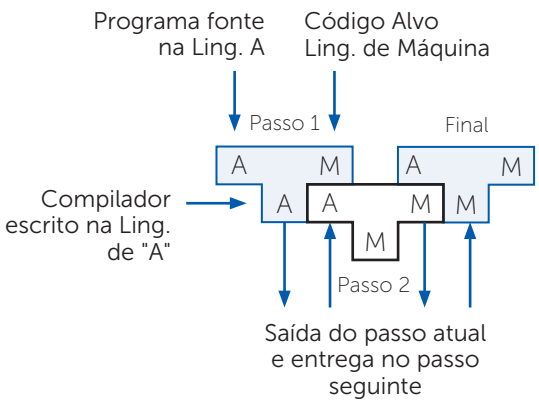
Figura 1.3 | Primeiro compilador para linguagem 'A'



Fonte: elaborada pelo autor.

Agora desejamos escrever um compilador para a linguagem 'A' na própria linguagem 'A'. Como já temos o compilador para 'A' na linguagem de máquina, ao aplicar o processo de bootstrapping, é possível obter o código alvo 'M'. Veja o passo a passo do processo no diagrama da Figura 1.4.

Figura 1.4 | Exemplo de bootstrapping



Fonte: elaborada pelo autor.

Logo, pelo processo de *bootstrapping*, a partir da entrada do programa fonte 'A' e do compilador escrito na própria linguagem 'A', obtém-se o código alvo 'M' com o auxílio do primeiro compilador criado para a linguagem 'A', desenvolvido em 'M'.

Além dos compiladores autocompiláveis, temos os que são *cross-compilers* (compiladores cruzados), que usam a técnica de *bootstrapping* e são escritos em um ambiente, mas rodam em outro.

Agora, vamos conhecer um pouco mais as ferramentas de apoio para o desenvolvimento dos compiladores. Basicamente, essas ferramentas estão divididas em: geradores de analisadores léxicos, geradores de analisadores sintáticos e geradores de código.

Os geradores de analisadores léxicos permitem a automatização do processo de criação de autômatos e o reconhecimento das sentenças regulares a partir da especificação na notação EBNF.

Assimile

Nesse universo dos compiladores, há muitos termos que precisam estar bem claros, por isso é bom saber:

Token: é o nome da produção da gramática.

Lexema: é o elemento do token. Se comparássemos a um programa, poderíamos dizer que token é o nome da variável e o lexema o conteúdo da variável.

Scanner: é o gerador de analisador léxico, por exemplo: LEX, JFLEX. O scanner lê a especificação em um padrão EBNF e gera um programa que analisa um arquivo fonte (programa) escrito de acordo com a especificação, por exemplo: o Lex gera em C e o JFLEX gera em Java.

Lexer: é o analisador léxico. O programa gerado pelo scanner.

Parser: é o gerador de analisador sintático, por exemplo: Yacc, CUP. O parser lê a especificação da GLC (gramática livre de contexto) no padrão EBNF, recebe os tokens analisados pelo lexer e gera um programa que analisa a sintaxe de um arquivo fonte (programa) escrito de acordo com a especificação GLC, por exemplo: o Yacc gera em C e o CUP gera em Java.

Parsing: é o analisador sintático. O programa gerado pelo parser.

Assembly: linguagem de baixo nível.

Assembler: é o compilador para a linguagem assembly. Faz a passagem do código fonte para o código alvo em uma passada, também chamado de montador.

O primeiro e mais conhecido gerador de analisadores léxicos é o **Lex**. Segundo Brown (2012), foi projetado por Mike Lesk e Eric Schmidt para trabalhar com o **Yacc**, um gerador de analisador sintático, e ambas ferramentas rodam na plataforma UNIX e geram código em C. Essas ferramentas trabalham juntas para construírem o analisador sintático (o *parser*).

Os geradores de analisadores léxicos, também conhecidos como *scanners,* leem a especificação em um padrão EBNF e geram um programa que analisa o arquivo fonte escrito de acordo com a especificação, por exemplo: o Lex gera em C e o JFLEX gera em Java.

Outro representante dos *scanners* é o **Flex**, cujo *scanner* tem a vantagem de gerar analisadores léxicos mais rápidos que o Lex e acompanha a sintaxe do Lex com pequenas variações. O código gerado também é em C.

Existem, também, geradores para Java. São o **JAVACC** e **JFLEX.** O **JAVACC** é uma ferramenta completa, com o *scanner* e o *parser*. Há, ainda, a dupla **JFLEX&CUP.** O JFLEX é o *scanner* e trabalha em conjunto com o CUP que é o *parser*. A dupla JFLEX & CUP é uma adaptação do Lex & Yacc para o Java, com pequenas variações.

Reflita

- Quais as vantagens de construirmos um compilador do zero?
- E no caso de usar ferramentas de apoio, haveria vantagens ou desvantagens?
- Se você desejasse construir um compilador usando Java, qual ferramenta usaria? Caso a escolha seja utilizar a linguagem Python, qual ferramenta de apoio poderia usar?

  Se lembrarmos que a técnica de bootstrapping permite criar compiladores em qualquer linguagem de programação, a escolha será limitada apenas pela plataforma que irá utilizar e pela linguagem utilizada pelas ferramentas de apoio, não é verdade?

As ferramentas, os *scanners* e os *parsers* compõem o *frontend* do compilador e estão associados à parte de análise. A parte da síntese, em que é gerado o código alvo, constitui o *backend*. Para essa fase do desenvolvimento do compilador, temos a ferramenta *Back End Generator* (BEG), um software proprietário. Podemos utilizar, ainda, simuladores para a geração do código para a máquina final, como é o caso da ENS2001, ou usar TASM e TLink, que são montadores

para a linguagem *assembly*. Para o Java, temos o **Jasmin,** que é um *assembler* (montador) para a *Java Virtual Machine* (JVM).

Exemplificando

Vamos conhecer um pouquinho do gerador **Lex** por meio de um exemplo simples para reconhecer algumas palavras. Tudo o que aparece entre /*...*/ são comentários para ajudar no entendimento do código:

```
%{
/* exemplo para reconhecer um conjunto de palavras
   Tudo que escrevemos entre %{    %} é colocado no início do fonte gerado na linguagem C.
*/
%}
%%    /* indica fim de uma seção */
[\t ]+    /* ignora os espaços em branco */
if |       /*  aqui temos uma lista  de palavras */
then |
else |
integer |
real |
go  { printf("%s: é Palavra reservada\n", yytext); }
[a-zA-Z]+  { printf("%s: NAO é Palavra reservada\n", yytext); }
.|\n   {ECHO;  }
%%
main(){
    yylex();
}
```

Ao testar o código acima para a entrada: **if ligado then 10**, a saída será:

if: é Palavra reservada.
ligado: NAO Palavra reservada.
then: é Palavra reservada.
10

Vamos, agora, entender o significado de cada seção do nosso exemplo:

%{   %} ⇒ os comandos escritos dentro dessa seção irão aparecer no início do código C, gerado pelo Lex.

%% ⇒ indicador de fim de uma seção.

+ ⇒ indica uma (1) ou mais ocorrências.

[\t]+ ⇒ reconhece os espaços em branco e os ignora.

A regra a seguir consiste em uma lista de palavras a serem reconhecidas

e, quando uma delas é encontrada, o comando indicado entre chave '{ }' é executado:

if |

then |

else |

integer |

real |

go { printf("%s: é Palavra reservada\n", yytext); }

[a-zA-Z]+ ⇒ reconhece palavras escritas em maiúsculas e/ou minúsculas, compostas apenas de letras e, reconhecendo, executa o comando { printf("%s: NAO é Palavra reservada\n", yytext); }.

.\n {ECHO; } ⇒ o '.' (ponto) aceita qualquer caractere de entrada, e {ECHO} gera uma rotina padrão em C para imprimir a entrada reconhecida e salta uma linha, efeito do '\n' após o ponto.

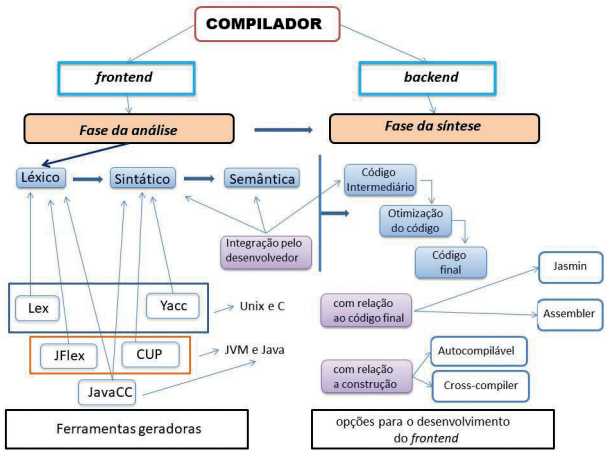
Após o encerramento da seção de regras (%%), colocamos o código em C que o Lex irá inserir no final do fonte que será gerado. No caso, foi colocada a função main().

O Lex cria a sub-rotina yylex(), por isso a chamamos em main().

Foram estudadas várias opções para construção de um compilador, seja para o *frontend*, seja para o *backend*. A escolha da ferramenta de apoio ao desenvolvimento implica em saber para qual ambiente se deseja usar o compilador e também em qual linguagem ele será escrito. A Figura 1.5 nos mostra um mapa com essas opções, permitindo uma visão geral dessas alternativas, apresentando um mapa conceitual do processo de construção do compilador. O *frontend* é a fase da análise, composta pelo léxico, sintático e semântico. Para a construção da análise léxica e sintática, caso seja utilizada a plataforma UNIX e a linguagem C para construção do compilador, poderão ser usadas as ferramentas LEX & YACC; caso sejam utilizados o ambiente JVM e a linguagem JAVA, pode-se utilizar as ferramentas JFLEX & CUP ou JAVACC. Para os analisadores gerados, quaisquer que tenham sido as ferramentas de apoio utilizadas, elas deverão ser integradas pelo desenvolvedor para conclusão da análise semântica e passagem para a fase de síntese. O *backend* do compilador consiste na fase de síntese, mas, para ela, não temos ferramentas de apoio. A fase de síntese está subdividida em código intermediário, otimização do código e

geração do código final. No planejamento do compilador, deve-se decidir se a implementação será autocompilável (uso da técnica de *bootstrapping*) ou por *cross-compillers*. Para geração do código de máquina será necessário usar algum montador (*Assembler*) ou, caso seja um compilador híbrido, usar uma máquina virtual, por exemplo Jasmin para o JVM.

Figura 1.5 | Mapa conceitual do processo de construção do compilador



Fonte: elaborada pelo autor.

Definido o processo para construção de um compilador, é importante sabermos o que esperar de um bom compilador. Segundo Grune (2012), um bom compilador: gera o código alvo correto; faz a análise do código fonte de acordo com a especificação da linguagem; é capaz de lidar com programas de qualquer tamanho e de usar algoritmos adequados para a otimização dos códigos e do gerenciamento da memória; e, finalmente, é facilmente portável para diferentes plataformas.

Após conhecer as etapas envolvidas no processo de construção do compilador e as qualidades a que ele deve atender, será necessário direcionarmos nossa visão para a meta a ser atingida por ele. Nesse ponto do planejamento, algumas perguntas deverão ser respondidas antes de ser iniciada a construção do compilador. Veja os itens que deverão ser analisados pelo projetista:

1. Finalidade do compilador a ser criado - É importante analisar se o objetivo será a geração do código de máquina ou apenas interpretação e análise de uma entrada, como é o caso da análise de dados, que, no meio profissional, é conhecido como "higienização", bem como sua importância para uso científico.
2. Eficiência – Nesse aspecto, há dois pontos de vista:

(a) Com relação ao processo de compilação, se necessitarmos rapidez na compilação, será importante analisar o número de passagens que serão feitas. Vale lembrar que a implementação por interpretação também resulta em baixa eficiência nesses casos.

(b) Com relação à eficiência do código alvo resultante, o cuidado que deverá ser tomado nesse caso está relacionado aos algoritmos de otimização do código e ao gerenciamento da memória. Apesar de o tamanho dos executáveis não ser mais um grande problema, isso tem grande relevância caso os usuários dos compiladores forem gerar aplicações para softwares embarcados, caso em que o tamanho do código gerado é limitado.

3. Plataforma – É preciso considerar para qual plataforma o código alvo irá rodar, qual será a plataforma do desenvolvimento e a portabilidade tanto do código alvo como do compilador. Vamos analisar cada item:

(a) Plataforma para o código alvo: Se o compilador e o código alvo usarem a mesma plataforma, devemos fazer a implementação *cross-compiler*, caso contrário, outras opções podem ser usadas, tal como a autocompilável ou a autoresidente;

(b) Plataforma do desenvolvimento: dependendo da plataforma do desenvolvimento, teremos opções especificas com relação às ferramentas de apoio, e, se a linguagem do desenvolvimento não tiver ferramenta que agilize sua construção, ele terá que ser feito manualmente e/ou teremos que usar mais passos, com a aplicação da técnica de *bootstapping*. Caso você opte por utilizar vários passos, não se esqueça de considerar a finalidade e a eficiência desejadas, de acordo com os itens 1 e 2 já expostos, pois, caso tenha muitos passos,

haverá uma queda na performance em relação ao processo de compilação.

(c) Portabilidade – Atualmente, a portabilidade, tanto do código alvo como do próprio compilador, deve ser levada em consideração, afinal, os clientes querem ter liberdade para usar os aplicativos em diversas plataformas e não ter limitações para uso, e o fabricante, por sua vez, não quer limitar seu mercado de venda criando aplicações apenas para uma plataforma. Saber a finalidade do código alvo ajuda na escolha da necessidade de versões para múltiplas plataformas, ou seja, de usar o desenvolvimento *cross-compiler*. Se a plataforma da construção e do uso do compilador for a mesma, devemos realizar o desenvolvimento *cross-compiler*. Se a questão da eficiência não exigiu a construção manual, um projeto cuidadoso e sistematizado, com o uso de ferramentas de apoio, ajudará muito em futuras adaptações para outras plataformas. Essa é uma das grandes vantagens do uso de ferramentas: a sistematização e a consequente documentação, além da agilização do processo de construção do compilador.

4. Linguagem do desenvolvimento – A escolha da linguagem na qual será construído o compilador abrirá ou não a possibilidade do uso de ferramentas de apoio, além de definir o grau de eficiência do processo de compilação. Tanto o uso de linguagem interpretada como uma compilação de múltiplos passos geram lentidão no processo. Ainda deve-se avaliar se o compilador será desenvolvido em uma plataforma e usado em outra, pois, nesse caso, será necessária uma implementação cruzada. Seja qual for a opção de linguagem, é fundamental a sua proficiência e, no caso de uso de ferramenta de apoio, o domínio dela.

Portanto, esses quatro itens (finalidade; eficiência do compilador e do código alvo; plataforma do compilador, do desenvolvimento e do código alvo; linguagem do desenvolvimento) permitem ao projetista fazer a melhor escolha.

Talvez você esteja se perguntando se o tamanho do compilador a ser criado faz diferença, e a resposta é sim, mas, atualmente, isso não é um fator crítico, pois os hardwares atuais possuem tanto

memória quanto velocidade suficientes para esse tipo de aplicação. Portanto, ambos deixaram de ser um fator crucial na maioria dos casos, como era no passado para os primeiros compiladores.

Pesquise mais

Desenvolver um compilador é construir um programa complexo, portanto, estude mais sobre:

a) Linguagem de programação C, C++ e Java.
b) Notação EBNF, gramática regular e linguagens livres de contexto.
c) Geradores de analisadores léxicos e sintáticos.

Isso o motivará e ajudará no seu aproveitamento dos estudos, trazendo um diferencial para a sua formação profissional, já que este é um tema de grande complexidade e com um mercado promissor, com poucos especialistas com visão prática.

Para ajudá-lo a ampliar sua visão sobre o assunto e reforçar alguns pontos, consulte os links a seguir:

1) THE CATALOG of Compiler Construction Tools. [S.l.; s.d.]. Disponível em: <http://catalog.compilertools.net/>. Acesso em: 4 jun. 2018.
   Este site apresenta um catálogo com opções de ferramentas para o frontend e para o backend, e o direciona a os sites oficiais de cada uma. Vale a pena conferir.
2) Para saber mais sobre o JFLEX, o link a seguir mostra um passo a passo do seu uso, sem muita teoria e com exemplos.
   CONSTRUINDO o primeiro analisador léxico com JFlex. [S.l.; s.d.]. Disponível em: <https://johnidm.gitbooks.io/compiladores-para-humanos/content/part2/building-the-first-lexical-analyzer-with-JFlex.html>. Acesso em: 4 jun. 2018.
3) Este você irá gostar: a ferramenta GALS é um gerador léxico e sintático para uso didático. Nela você poderá experimentar a notação EBNF. No link a seguir você encontrará um tutorial e poderá baixar a ferramenta.
   GERADOR de Analisadores Léxicos e Sintáticos. [S.l.; s.d.]. Disponível em: <http://gals.sourceforge.net/>. Acesso em: 4 jun. 2018.
4) Se o objetivo é prática, o vídeo a seguir é uma apresentação de um compilador para a linguagem SQL. A apresentação é de um projeto completo, mas recomendamos os 10 minutos iniciais e os 5 minutos finais.

PROJETO Compilador com Jflex e Java Cup. 10 dez. 2013. Disponível em: <https://www.youtube.com/watch?time_continue=92&v=GY0y288vRX4>. Acesso em: 4 jun. 2018.

5) Faça uma revisão do Java. Recomendamos o tutorial a seguir.
ORACLE. **Trail**: Learning the Java Language. [S.d.]. Disponível em: <https://docs.oracle.com/javase/tutorial/java/index.html>. Acesso em: 4 jun. 2018.

Avançamos mais um pouco no estudo dos compiladores, adquirindo conhecimento sobre a classificação da implementação, que pode ser: autocompilável, *cross-compilers* ou autoresidente. Também ficamos conhecendo as diversas ferramentas que apoiam o desenvolvimento dos compiladores, como visto na Figura 1.5, que mostra as diversas relações entre os processos envolvidos na construção, seja com relação à linguagem de desenvolvimento, seja pelas ferramentas ou pelas plataformas. Completamos a seção analisando os cuidados que o projetista deverá ter antes de iniciar a construção do compilador e quais são os fatores cruciais para atingir seus objetivos com boa eficiência e eficácia.

Compreendida a estrutura do compilador e da linguagem, bem como as necessidades para avaliação das escolhas a serem tomadas para o início da sua implementação, agora é o momento de programarmos a fase inicial do *frontend*: o analisador léxico. Preparado para começar a criar seu primeiro analisador léxico? A próxima unidade será exatamente sobre isso.

Sem medo de errar

Caríssimo aluno, os fatores envolvidos no projeto e na construção de um compilador foram estudados e agora será o momento de colocá-los em prática. A direção da *start-up*, fez duas solicitações: (a) uma proposta de cronograma, especificando as ferramentas que serão necessárias, para que analisem a viabilidade financeira do projeto, bem como (b) a continuidade das palestras destinadas à capacitação dos seus profissionais, com um workshop sobre ferramentas que agilizam a construção dos compiladores. Preparado e organizado para isso?

Você precisará (a) entregar o cronograma para implementação do compilador, elencar as ferramentas que serão utilizadas e (b) realizar o workshop sobre as ferramentas existentes para construção de um compilador, para capacitar os profissionais da empresa.

Sugerimos começar pelo planejamento do workshop, assim o cronograma poderá ser criado como a consequência do caso de uso utilizado no workshop. Use um exemplo completo de como construir um compilador, assim você irá concluir essa fase conforme solicitado no início da sua contratação na empresa. Como o objetivo é um workshop, que tal a linguagem LOGO, para mostrar como planejar o *frontend* e o *backend* de um compilador para o LOGO? Afinal, você já a especificou na unidade anterior.

Primeiramente, devemos começar pela análise dos quatro fatores estudados, que qualquer projetista deve considerar antes de tomar qualquer decisão, levantando os questionamentos, para somente depois de discutir com os participantes apresentar conjuntamente uma solução. Esse é o espirito do workshop. Vamos, então, elencar estes fatores:

1. Finalidade – neste caso será uma linguagem para fins educacionais.

   Será criado apenas o *frontend* ou *frontend* e *backend*?

2. Eficiência

   2.1 A compilação precisa ser rápida?

   2.1 O executável gerado precisa ser rápido?

3. Plataforma

   3.1 O programa resultante da compilação, o executável, será rodado na mesma máquina?

   3.2 O compilador será desenvolvido na mesma plataforma para a qual irá gerar os executáveis?

4. Linguagem

   4.1 Em qual linguagem se pretender escrever o compilador?

   4.2 O uso de ferramentas para agilizar o processo seria possível?

Responder a essas perguntas ajudará qualquer projetista a tomar decisões assertivas, já que todos os fatores importantes estão elencados. Você já conseguiu respondê-las? Vamos em frente? Que tal criar uma tabela?

Tabela 1.1 | Exemplo de tabela

FATORES A SEREM ANALISADOS	COM RELAÇÃO AO	O QUE SE ESPERA	
EFICIÊNCIA	COMPILADOR	**RAPIDEZ**	*não é relevante*
	EXECUTÁVEL	**RAPIDEZ**	*não é relevante*
PLATAFORMA	COMPILADOR	*única*	***MULTIPLATAFORMA***
	EXECUTÁVEL	*única*	***MULTIPLATAFORMA***
LINGUAGEM	DESENVOLVIMENTO	*não é relevante*	**ESPECÍFICA**
	FERRAMENTAS	JAVA	C++

Fonte: elaborada pelo autor.

De acordo com a Tabela 1.1, quando a resposta for 'não relevante' ou 'única', temos opções de escolha, mas nos outros casos será necessária coerência nas decisões, pois as dependências limitarão as escolhas. Isso é o que podemos chamar de visão sistêmica, que somente é alcançada com planejamento.

Observe que o item finalidade não consta na Tabela 1.1, pois, nesse quesito, a amplitude dos fatores a serem considerados não é fechada. Então vamos primeiramente definir a finalidade para depois iniciar o preenchimento, pois isso ajudará a responder adequadamente as demais perguntas.

Nosso compilador tem fins educacionais, e a probabilidade de escolas diferentes usarem ambientes distintos será grande. Por exemplo, uma usa Windows, outra Linux. Ainda com relação à finalidade, devemos responder se o compilador a ser criado será completo ou não. Deverá ser completo, pois será necessário um produto para ser comercializado pela empresa. Logo, conhecidas as respostas para a finalidade do produto, vamos responder as demais perguntas e marcá-las na tabela construída para esse caso. Veja o resultado na Tabela 1.2.

Tabela 1.2 | Exemplo de tabela

FATORES A SEREM ANALISADOS	COM RELAÇÃO AO	O QUÊ SE ESPERA	
EFICIÊNCIA	COMPILADOR		*não é relevante*
	EXECUTÁVEL		*não é relevante*
PLATAFORMA	COMPILADOR		***MULTIPLATAFORMA***
	EXECUTÁVEL		***MULTIPLATAFORMA***
LINGUAGEM	DESENVOLVIMENTO	*não é relevante*	
	FERRAMENTAS	JAVA	

Fonte: elaborada pelo autor.

De acordo com a Tabela 1.2, o ponto crítico nesse projeto é a necessidade de portabilidade, tanto do compilador como do executável gerado por ele. Logo, a implementação deverá usar a técnica de *cross-compilers* devido à probabilidade do executável rodar em uma plataforma diferente do compilador. E, com relação à linguagem para escrever o compilador, a escolha foi o Java, pois permite o uso de ferramentas de apoio que agilizaram o desenvolvimento, tal como o JFLEX & CUP para o *frontend* e o JASMIN para o *backend*, esse, sendo um *assembler* para a JVM, garante a possibilidade de executáveis multiplataforma.

Muito bem, seu workshop está planejado e o projeto praticamente pronto, pois, pelo exposto, podemos concluir que as ferramentas necessárias serão:

(a) a JDK, para o Java - linguagem na qual o compilador será construído.

(b) o JFLEX&CUP ou JAVACC para o *frontend*.

(c) o JASMIN para o *backend*.

(d) uma IDE para desenvolvimento, que poderá ser o Eclipse ou o Netbeans, já que ambas são compatíveis com o Java e com os geradores a serem utilizados. A boa notícia para o setor financeiro é que todas essas ferramentas não têm custo.

Resta elaborar o cronograma. Sugerimos que você o ajuste ao seu plano de estudo dessa disciplina, afinal iremos construir um

compilador durante o curso: planejamento, na Unidade 1; analisador léxico, na Unidade 2; analisador sintático e semântico, na Unidade 3; e, finalmente, o *backend*, na Unidade 4.

Durante a resolução da situação problema, os conhecimentos adquiridos nesta seção foram relevantes para permitir as tomadas de decisão com relação às técnicas que deverão ser utilizadas na criação do produto, o compilador, seja na análise dos fatores relevantes para que se obtenha um bom compilador, seja nas escolhas das ferramentas. Essas competências o prepararam para desenvolver um trabalho com qualidade.

Avançando na prática

Soluções integradas e o uso de compiladores

Descrição da situação-problema

Uma determinada companhia adquiriu um sistema ERP, que possibilita a extração de dados e até mesmo a programação de novos recursos pelos seus funcionários diretamente no sistema, entretanto, eles estão com dificuldades com a linguagem e alguns funcionários apresentaram uma proposta de como gostariam de escrever as instruções para o sistema. Considerando que esse recurso é muito importante para a gestão estratégica do negócio, eles procuraram a empresa de TI em que você trabalha para encontrar uma solução.

A sua empresa convocou uma reunião com o seu time de desenvolvimento e pediu para vocês analisarem a possibilidade da criação de algum aplicativo integrado com o ERP utilizado pelo cliente e que atenda às suas necessidades. É possível atender à solicitação da empresa e criar um aplicativo integrado com o ERP utilizado pelo cliente? Se não for possível, qual a justificativa?-

Resolução da situação-problema

Seus olhos brilharam quando ouviu o que estava sendo solicitado. Finalmente você iria aplicar os conceitos de linguagens formais e

de compiladores na prática. Você pensou consigo mesmo: valeu a pena o estudo de toda aquela teoria complexa, afinal, poucos estão enxergando que o caso apresentado refere-se à criação de um compilador que deverá analisar a linguagem proposta pelo cliente e traduzir para a linguagem do ERP. Sendo assim, você pediu a palavra e explicou isso, inclusive indicando exemplos encontrados em outros ERPs bem conhecidos no mercado, como a linguagem advPL, presente nos sistemas da empresa TOTVS, ou a ABAP, no sistema SAP R/3.

Para saber mais, sugerimos consultar os sites a seguir:

- http://tdn.totvs.com/display/tec/AdvPL
- https://www.cbsi.net.br/2015/06/cinco-apostilas-sobre-linguagem-de-programacao-abap.html

Como vimos, existe uma solução para este problema. Portanto, vamos planejar os passos necessários para o desenvolvimento:

1º. Definir com o cliente a linguagem de programação desejada e validá-la.

2º. Escrever a especificação da linguagem nova.

3º. Considerar os 4 fatores fundamentais para planejar a implementação de um compilador, sendo importante primeiro definir a finalidade e depois elaborar a tabela para tomada de decisão, a qual analisa os demais fatores, assim será possível definir:

a) Finalidade: criar um *frontend* e um *backend,* que deverá gerar um código de alto nível, ou seja, a linguagem do ERP. E, no caso dessa situação especifica, o time de TI deverá se familiarizar com essa a linguagem do ERP utilizado pelo cliente;
b) Construção da tabela:

Tabela 1.3 | Exemplo de tabela

FATORES A SEREM ANALISADOS	COM RELAÇÃO AO	O QUE SE ESPERA	
EFICIÊNCIA	COMPILADOR		*não é relevante*
	EXECUTÁVEL		*não é relevante*
PLATAFORMA	COMPILADOR	única	
	EXECUTÁVEL	única	

FATORES A SEREM ANALISADOS	COM RELAÇÃO AO	O QUE SE ESPERA	
LINGUAGEM	DESENVOLVIMENTO	*não é relevante*	
	FERRAMENTAS	JAVA	

Fonte: elaborada pelo autor.

Pela tabela apresentada, optou-se pela utilização do JAVA com JLFEX & CUP para construção do *frontend* do compilador. O uso do JFLEX & CUP ajudará na documentação dos processos e agilizará o desenvolvimento. Com relação ao *backend*, como a saída será a linguagem do ERP, essa de alto nível, logo não precisaremos do JASMIN. Entretanto, a transição para a linguagem alvo será feita pelo time de desenvolvimento (geração do código alvo).

Como você pode perceber, a tabela criada não apresentou qualquer restrição ou situação limitante, tanto com relação à plataforma como à velocidade ou linguagem. Portanto, será um compilador autoresidente, e o uso do Java permitirá a portabilidade, caso o ERP rode em diversas plataformas.

4º. Apresentar da proposta.

Vale a pena estudar compiladores, não é mesmo? Esta tecnologia está embutida em muitas situações profissionais. Basta você se lembrar de associar as teorias estudadas à prática, como fizemos nesse caso.

Faça valer a pena

1. *Compiler Compilers* é um termo em inglês utilizado para referenciarmos de forma genérica as ferramentas que auxiliam o desenvolvimento de compiladores, que são os geradores. Sendo a construção de compiladores um processo complexo, há diferentes geradores para cada fase do compilador.

Assinale a alternativa que correspondente à relação entre o aplicativo (ferramenta) e sua função:

a) LEX; gerador de código alvo.

b) JFLEX; gerador de analisadores Java.

c) JAVACC; gerador de analisadores semânticos para o Java.

d) Yacc; gerador de analisador sintático.

e) JASMIN; ferramenta completa para produção de compiladores.

2. Com relação à finalidade desejada para um compilador, a implementação será classificada em: autoresidente, autocompilável ou cruzada (*cross-compilers*). Compiladores que exigem rapidez da compilação e do executável, normalmente, são autorresidentes, já os cruzados (*cross-compilers*) e autocompiláveis exigem várias passagens.

Sobre a implementação de compiladores autocompiláveis, é correto afirmar:

a) Tem sido a opção mais utilizada na construção dos compiladores, pois faz todos os processos automaticamente.
b) Permite que você faça um compilador na própria linguagem em que foi implementado.
c) Utilizamos implementação autocompilável quando é necessário que o código alvo rode em plataformas diferentes da do compilador.
d) Utilizamos implementação autocompilável quando é necessária a portabilidade do compilador, do código fonte e do executável.
e) As implementações autocompilável e *cross-compiler* diferem apenas quanto ao código alvo.

3. O compilador é um aplicativo cujo objetivo é a tradução de uma linguagem para outra, e, sendo um produto, deverá atender adequadamente o seu propósito. Entretanto, um produto de qualidade é aquele que atinge o objetivo com diferenciais e excelência. Para isso, cuidados são necessários, desde seu projeto até a conclusão da sua construção.

De acordo a análise apresentada, o que devemos esperar para um compilador ser considerado bom? Assinale a alternativa correta:

a) (1) Gerar o código alvo correto; (2) lidar com programas fontes de qualquer tamanho; (3) ser facilmente portável; (4) ser pequeno; (5) ser rápido.
b) (1) Gerar o código alvo correto; (2) fazer adequadamente a análise sintática e semântica, e gerar mensagens de erro com clareza; (3) ser rápido; (4) lidar com programas fontes de qualquer tamanho; (5) ser fácil.
c) (1) Gerar o código alvo correto; (2) fazer adequadamente a análise sintática e semântica, e gerar mensagens de erro com clareza; (3) lidar com programas fontes de qualquer tamanho; (4) ser rápido; (5) ser pequeno.

d) (1) Gerar o código alvo correto; (2) fazer adequadamente a análise sintática e semântica, e não gerar mensagens de erro; (3) lidar com programas fontes de qualquer tamanho; (4) usar algoritmos adequados para a otimização do código e gerenciamento da memória; (5) ser facilmente portável.

e) (1) gerar o código alvo correto; (2) fazer adequadamente a análise sintática e semântica, e gerar mensagens de erro com clareza; (3) lidar com programas fontes de qualquer tamanho; (4) usar algoritmos adequados para a otimização do código e gerenciamento da memória;(5) ser facilmente portável.

Referências

AHO, A. V.; SETHI R.; ULLMAN J. D. **Compiladores**: princípios, técnicas e ferramentas. 2. ed. São Paulo: Pearson, 2007.

BROWN, D.; LEVINE, J.; MASON, T. **Lex & Yacc**. 2. ed. Sebastopol – CA: O'Reilly Media, 2012.

CHOMSKY, N. **Estruturas Sintacticas.** Lisboa: Edições 70, 1980. (Colecções Signos; 28). Tradução: Madalena Cruz Ferreira.

DELAMARO, M. E. **Como construir um compilador utilizando ferramentas Java**. 1. ed. São Paulo: Novatec, 2004.

FORENZANO, C. **Cinco apostilas sobre linguagem de programação ABAP gratuitas para download**. CBSI, 2015. Disponível em: <https://www.cbsi.net.br/2015/06/cincoapostilas-sobre-linguagem-de-programacao-abap.html>. Acesso em: 21 jun. 2018.

GRUNE, D. et al. **Modern Compiler Design**. 2. ed. New York: Springer, 2012.

MENEZES, P. B. **Linguagens Formais e Autômatos**. 5. ed. Porto Alegre: Sagra Luzzatto, 2005.

RODRIGUES, L. Brasileiro cria linguagem de programação e conquista o Vale do Silício. **Correio Braziliense**, 07 mar. 2018. Disponível em: <http://www.correiobraziliense.com.br/app/noticia/economia/2018/03/07/internas_economia,664353/brasileiro-cria-linguagem-de-programacao-e-conquista-o-vale-do-silicio.shtml>. Acesso em: 21 jun. 2018.

SÁ, C. C.; SILVA, M. F. **Haskell - Uma Abordagem Prática.** 1. ed. São Paulo: Novatec, 2006.

SCOWEN, R. S. **Information technology — Syntactic metalanguage — Extended BNF**. Middlesex: ISO/IEC 14997, 1996.

SEBESTA, R. W. **Conceitos de Linguagens de Programação**. Tradução técnica: Eduardo Kessler Piveta. 9. ed. São Paulo: Bookman, 2011.

SIMÃO, T.; PRINCE, A. **Implementação de Linguagens de Programação:** compiladores. Porto Alegre: UFRGS: Sagra Luzzato, 2001. (Série Livros Didáticos – nº 9).

TOTVS. **A linguagem AdvPL**. Disponível em: <http://tdn.totvs.com/display/tec/AdvPL>. Acesso em: 21 jun. 2018.

TUCKER, A.; NOONAN, R. **Linguagens de programação:** princípios e paradigmas. 2. ed. Porto Alegre: McGraw-Hill, 2009.

TURING, A. M. **On Computable Numbers, with an Application to the Entscheidungsproblem**. Proceedings of the London Mathematical Society, série 2, 42(1):230–265, 1936. Disponível em: <https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/plms/s2-42.1.230>. Acesso em: 21 jun. 2018.

Especificação da análise léxica e técnicas de implementação

Convite ao estudo

O objetivo desta unidade é capacitá-lo a construir analisadores léxicos e avançar na compreensão sobre como realizar a especificação da análise sintática, incluindo, também, o estudo das estratégias possíveis e mais comuns adotadas na implementação de um compilador relacionadas ao tratamento de erros de sintaxe.

A maioria das linguagens de programação pertencem à classe das linguagens livres de contexto (LSC), mas os elementos da análise léxica estão contidos em uma classe mais simples, a das linguagens regulares (LR). Assim, a fase da análise léxica é a mais simples para implementação, mas, por outro lado, é nessa fase que trabalhamos com as expressões regulares (ER), que são aplicadas em vários segmentos, por isso é muito importante conhecê-las.

Ao final desta unidade você será capaz de especificar a gramática completa de uma linguagem de programação e de construir seu analisador léxico, além de conhecer as principais estratégias que podem ser adotadas na implementação de um compilador relacionadas ao tratamento de erros de sintaxe.

Você foi procurado por uma grande companhia da área de manufatura para construir uma linguagem simples que permita que os executivos extraiam dados existentes em seus bancos de dados, pois a linguagem disponibilizada atualmente pelo sistema de apoio à decisão é muito técnica e seus executivos, que não são programadores, estão com dificuldades em usá-la.

Ao final da unidade você entregará à empresa: (a) a especificação léxica e a sintática da linguagem de programação

solicitada; e (b) o analisador léxico da linguagem, com o código fonte e os testes de validação.

Entusiasmado para iniciar o desenvolvimento do analisador léxico? Para ajudá-lo a vencer mais um desafio, iremos estudar, na primeira seção, as funções do analisador léxico e o legado que ele entrega para o analisador sintático, as vantagens desse processo de construção do compilador, como se dá o processo de leitura dos dados (chamada bufferização de entrada) e o uso de diagramas que auxiliam na especificação dos *tokens*, para, na segunda seção, construirmos um analisador léxico (*lexer*) utilizando um gerador, tal como o JFLEX, e avançarmos para a próxima fase do compilador, o analisador sintático, na terceira seção, na qual, além de estudarmos as funções do analisador sintático, estudaremos as estratégias de tratamento e recuperação de erros. Vamos começar?

Seção 2.1

Análise léxica

Diálogo aberto

Caro aluno, nesta seção você irá iniciar o estudo da primeira fase do desenvolvimento de um compilador, a análise léxica. Para tanto, irá estudar a função do analisador léxico, as razões de se dividir a fase de sintaxe em léxica e sintática, a técnica de bufferização da entrada para análise léxica, além de aprender como especificar os *tokens* e de conhecer como se faz o reconhecimento dos *tokens*. Portanto, o foco desta seção são as linguagens regulares (LR), que estão contidas nas linguagens livres de contexto (LSC), como vimos na Unidade 1. A fase da análise léxica é a mais simples para implementação, por outro lado, é nessa fase que trabalhamos com as expressões regulares (ER), que são aplicadas em vários segmentos, por isso é muito importante conhecê-las.

Hoje em dia, a tecnologia da informação é amplamente utilizada na gestão estratégica das empresas e, consequentemente, profissionais de diversas áreas necessitam filtrar ou criar relatórios a partir de dados existentes nos sistemas de informações utilizados pelas empresas para auxiliá-los na tomada de decisões. Frente a essa necessidade, os grandes sistemas de apoio à decisão (SAD) incorporaram facilidades para que seus usuários, que não necessariamente são programadores, possam criar seus próprios cruzamentos de dados e obter dados estatísticos, relatórios ou planilhas a partir dos dados existentes em seus sistemas de informações ou *datawarehouse*, sem que, para isso, seja necessária a intervenção das equipes de desenvolvimento. Assim, o processo de análise dos dados pode ser mais rápido e dinâmico, dando aos administradores maior domínio sobre as informações, o que permite agilidade nas decisões e melhor visão estratégica do negócio.

Mas, nem tudo são "flores". Os grandes sistemas, como BAAN, SAP/R3 ou ERP-TOTVS, possuem suas próprias linguagens, o que facilita sua customização pelas empresas que os utilizam, criando novos relatórios e funções, extrapolando suas funcionalidades

padrões, mas, ainda assim, dependem de suas equipes de desenvolvimento, mesmo que internas. Sendo assim, você foi procurado por uma grande companhia da área de manufatura para construir uma linguagem simples, que permita a seus executivos extrair os dados existentes em seus bancos de dados.

A empresa que o procurou utiliza a linguagem *Structured Query Language* (SQL) padrão e deseja que você crie uma linguagem usando termos em português, para que sejam facilmente compreendidos pelos seus executivos, com uma sintaxe simples e palavras similares ao português.

Para atender à necessidade da empresa, será necessário criar a linguagem e o respectivo compilador. Nessa primeira etapa, a empresa solicitou que você faça a especificação dos elementos básicos da nova linguagem, ou seja, a especificação léxica.

Quando você aprendeu a linguagem SQL padrão, quais foram as suas dificuldades? Como você gostaria que o comando SELECT fosse? O que você mudaria para facilitar o entendimento e uso do SELECT e para usar termos em português? Talvez lembrar-se do seu próprio processo de aprendizagem do SQL o ajudará a criar uma linguagem simples para os executivos desta empresa.

Pronto para começar a criar a especificação dos elementos básicos de uma nova linguagem de programação de alto nível para ajudar a empresa que o procurou a aumentar sua produtividade?

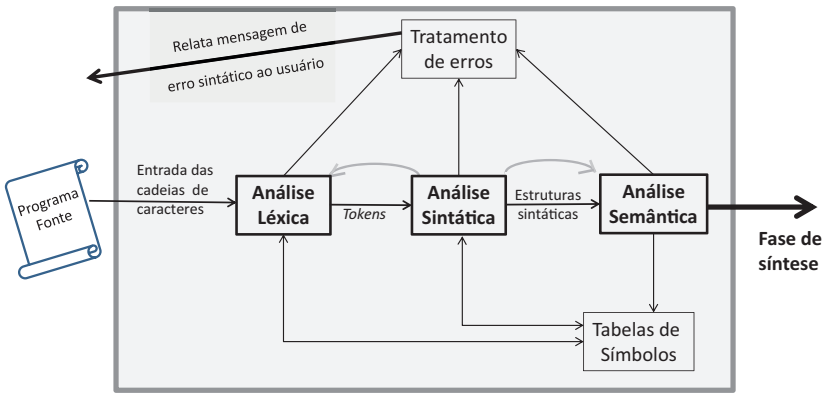
Vamos iniciar?

Não pode faltar

Como estudado na Unidade 1, o compilador divide-se em duas grandes fases, a de análise, também conhecida como *frontend*, e a de síntese, chamada de *backend*. Veja a Figura 2.1, que mostra todas as etapas da fase de análise. A análise léxica lê os caracteres de entrada, gera um fluxo de dados, os *tokens* e os disponibiliza para o analisador sintático. Por sua vez, o analisador sintático analisa o fluxo de dados gerado de acordo com a gramática livre de contexto (estruturas sintáticas) e, quando necessário, aciona a análise semântica para verificar se os elementos presentes na estrutura sintática são compatíveis. Se em cada uma dessas etapas não for encontrado erro de escrita no programa fonte, o fluxo de

dados será enviado para a fase seguinte, a de síntese, caso contrário, os erros de cada etapa da análise serão reportados ao usuário, o programador, e o processo de compilação será abortado. Durante essa fase, inicia-se a montagem da tabela de símbolos, que será utilizada durante o processo de compilação por todas as etapas.

Figura 2.1 | *Frontend*



Fonte: elaborada pela autora.

Nesta seção, iremos tratar especificamente da etapa inicial do *frontend*, a análise léxica, cuja função é ler os caracteres de entrada e, segundo Aho et al. (2007), produzir uma sequência de *tokens* que serão utilizados pela análise sintática.

Vale lembrar que, segundo Christensson (2009), em programação, um *token* é um elemento único de uma linguagem de programação e, como visto na unidade anterior, *tokens* são identificados por meio dos nomes dados às produções da gramática, ou seja, cada padrão reconhecido pela gramática.

A função exata do analisador léxico é reconhecer os *tokens* associados às expressões regulares, ou seja, o subconjunto da linguagem livre de contexto pertencente apenas às linguagens regulares, composto pelos elementos básicos, tais como identificadores, operadores, constantes, comentários, caracteres especiais e tipos compostos.

Na prática, é o analisador sintático (*parser*) que aciona o analisador léxico (*lexer*) para saber se a palavra lida na frase é válida para a linguagem, logo o analisador sintático depende da resposta do analisador léxico. Portanto, isolar o processo de reconhecimento de elementos básicos (*tokens*) facilita e agiliza muito processo de construção de um compilador, então construímos o analisador léxico antes do sintático.

Reflita

Porque separar o *lexer* (analisador léxico) do *parser* (analisador sintático)?

- *Lexer* classifica as palavras; é um processo simples.
- *Parser* gera derivações gramaticais; é um processo complexo e consequentemente lento.
- A existência de um *lexer* leva a um *parser* menor e mais rápido, além de agilizar a manutenção.

Uma técnica simples para construir um *lexer*, segundo Aho et al. (2007), é escrever um diagrama que ilustre a estrutura dos *tokens* da linguagem fonte e depois implementar um programa que o identifique. Podemos fazer essa implementação diretamente ou usar padrões para a especificação léxica e usar geradores de analisadores léxicos (*scanners*) tal como o Yacc ou o JFLEX.

Mas, o que motiva a adoção de padrões? Um dos motivos é que muitas áreas, além de compiladores, usam expressões regulares e já fazem uso destes padrões, tais como a linguagem de comandos *shell*, a função regexp() do mySQL, ou ainda, a função REGEX existente para várias linguagens. Além disso, a especificação das expressões regulares que representam os *tokens* da linguagem fonte é muito melhor documentada com o uso de padrões, pois deixam a gramática mais clara e precisa, facilitando futuras revisões e lançamentos de novas versões do compilador.

Nesta seção, vamos estudar a especificação do analisador léxico para que seja possível compreender integralmente seu processo de construção para, na próxima seção, praticarmos e estudarmos o uso da ferramenta JFLEX para implementar a especificação construída aqui.

Quando tratamos de análise léxica, devemos encontrar no programa fonte os padrões correspondentes ao par (tipo do *token*,

lexema). Para entendermos no que consiste esse padrão, vamos analisar o exemplo a seguir:

```
01    int x = 0  ;
02    x = 10 + 1b * x
```

As linhas 01 e 02 foram escritas na linguagem C e, de acordo com o código apresentado, identificamos os seguintes padrões para cada par (tipo do *token*, lexema): (<tipo de dado>, int), (<variável>, x), (<símbolo de atribuição>, =), (<constante numérica>, 0), (<terminador>, ;), (<constante numérica>, 10), (<operador aritmético>, +), (***Não reconhecido,*** 1b), (<operador aritmético>, *).

O analisador léxico irá ler carácter a carácter da entrada do programa fonte e reconhecer o padrão para cada termo identificado, assim, o termo, ou seja a palavra 'int', é reconhecida como um <tipo de dado>, e ´int' nesse caso é uma instância do <tipo de dado>, que denominamos lexema. Isto é, 'int' é o valor nessa situação que representa o <tipo de dado>. Assim, sucessivamente, *token* <variável> está associado ao lexema 'x', <constante numérica> está associada a 10 e *token* <símbolo de atribuição> está associado ao lexema '='.

Ainda podemos observar que o lexema '1b' não pode ser reconhecido por um *token*, portanto não é um padrão aceito pelo analisador léxico, assim o elemento '1b' será rejeitado e reportado à função de tratamento de erro, que deverá gerar uma mensagem adequada informando que o elemento foi rejeitado. Cada par (*token*, lexema) deve ser armazenado em uma tabela dinâmica de símbolos para futuras consultas pelas demais fases da compilação. Podemos utilizar tabelas *hash* para as pesquisas serem mais rápidas.

Pesquise mais

Tabela *hash* é uma estrutura de dados, que associa uma chave de pesquisa a valores. Assim, a partir de uma chave simples encontramos rapidamente o valor pesquisado. Para saber mais assista ao vídeo:

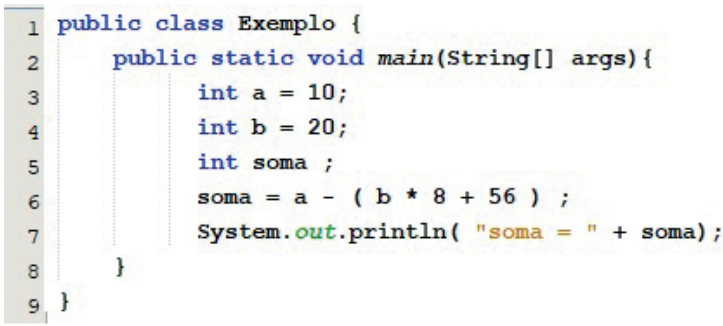
PAULO PARALUPPI. **Tabelas Hash** – Estrutura de Dados – Unicamp. 22 abr. 2011. [1m50s]. Disponível em: <https://www.youtube.com/watch?v=Non0I_OSt9o>. Acesso em: 28 jun. 2018.

Se deseja relembrar como usar tabelas hash, leia o capítulo abaixo da *Apostila Algoritmos e Estrutura de Dados com Java*:

TABELAS de Espelhamento. [S.l.; s.d.; s.p.]. In: **Apostila Algoritmos e Estruturas de Dados com Java**. Caelum. Disponível em: <https://www.caelum.com.br/apostila-java-estrutura-dados/tabelas-de-espalhamento/>. Acesso em: 28 jun. 2018.

Entretanto, antes do analisador iniciar o reconhecimento dos *tokens*, podemos eliminar os espaços em branco desnecessários, mas essa técnica requer a identificação de todos os símbolos terminais para não eliminar indevidamente um espaço em branco. Veja um exemplo disso nas Figuras 2.2 e 2.3

Figura 2.2 | Exemplo de uma classe em Java



Fonte: captura de tela da IDE Netbeans, elaborada pela autora.

O código fonte em Java representado na Figura 2.2 apresenta espaços em brancos desnecessários, e eliminar esses espaços pode facilitar o trabalho do analisador léxico para reconhecer os *tokens*, como mostrado na Figura 2.3. Assim o primeiro passo do analisador é eliminar os espaços em branco.

Figura 2.3 | Classe em Java sem espaços, escrita em uma mesma linha

```
1. public class Exemplo2{public static void
   main(String[] args){int a=10;int b=20;int
   soma;soma=a-(b*8+56);System.out.println("soma =
   "+soma);}}
```

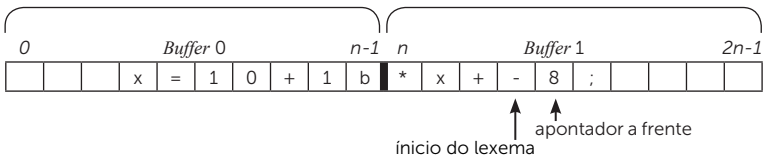
Fonte: captura de tela da IDE Netbeans, elaborada pela autora.

O segundo passo é o reconhecimento dos *tokens*. Nessa fase é realizada a leitura caractere a caractere, o que consome um tempo razoável, e há momentos em que o analisador léxico precisar verificar vários caracteres à frente para concluir o reconhecimento do *token*. Esse processo de armazenar dados e depois retroceder impõe uma sobrecarga no processamento, então a técnica de bufferização foi desenvolvida para obter uma melhor eficiência neste o processo.

Vejamos como se dá esse procedimento: uma estratégia é usar um *buffer* duplo, dividi-lo em duas partes e limitar o número de retrocessos. Seja a linha a ser analisada `x = 10 + 1b * x;` e o *buffer* representado na Figura 2.4., no qual podemos observar o uso de dois apontadores, um indicando o início do lexema e outro mais a frente. Nesse caso, o apontador à frente, ao avançar mais uma posição no *buffer* 1, encontrará o terminador ';' e o lexema '-8' será reconhecido como uma <constante numérica>, o qual encontra-se entre o apontador de início e o apontador à frente, o que torna desnecessário o retrocesso para a posição inicial para capturar o lexema.

Esse processo agiliza a varredura dos caracteres de entrada, mas, em alguns casos excepcionais, é necessário que a varredura avance. Entretanto, atualmente os geradores léxicos já implementam nas rotinas geradas por eles a técnica de bufferização, ou, no caso de criação direta, as linguagens atuais também possuem funções que implementam facilidade para esse processo.

Figura 2.4 | Exemplo de buffer de entrada



Fonte: elaborada pela autora.

Como visto, o primeiro passo foi preparar o arquivo de entrada a ser analisado, removendo espaços em branco, depois estudamos a técnica de bufferização para fazer a varredura da entrada, agora iremos estudar como desenvolver a especificação dos elementos léxicos de uma linguagem.

Os elementos léxicos (*tokens*) são basicamente os símbolos terminais das gramáticas e alguns elementos que pertencem às

linguagens regulares, e poderão ser formalizados pela notação EBNF, por expressões regulares ou por autômatos finitos determinísticos.

Exemplificando

Para você recordar um pouco de programação em JAVA, vamos mostrar uma classe que lê um arquivo e retira os espaços em branco. Assim você poderá ver o processo explicado na prática e aproveitar para relembrar um pouco da linguagem JAVA.

Figura 2.5 | Exemplo classe Leitura em Java - remove espaços em branco

```
package JFlex;
import java.io.*;     // importacao das classes do JAVA  para leitura
import java.util.*;
// classe para leitura
public class Leitura {
   // metodo LeFonte recebe como parametro o endereco e
   // o nome do arquivo a ser lido, ex.: C:\temp\programa.java
   public static ArrayList leFonte(String fonte){
      // arquivo é uma lista dinamica do tipo ArrayList
      ArrayList arquivo = new ArrayList();
      // try{...} catch --> caso o arquivo fonte nao seja encontrado
      // será emitida a mensagem de arquivo c:\temp\programa.java nao encontrado
      try{
          // cria uma instancia do objeto BufferedRead , para leitura
          // do arquivo fonte
         BufferedReader in = new BufferedReader(new FileReader(fonte));
         String linha;
         // le a primeira linha do arquivo fonte e armazena na variavel linha
         // e enquanto o arquivo de entrada nao teminar
         // repete o processo para as linhas subsequentes
         while( (linha = in.readLine()) != null){
             // remove todos os espacos conforme a expressao regular [ ]+
             // e substitui por apenas um espaco
            linha = linha.replaceAll("[ ]+", " ");
            // inclui no arquivo de saida a linha "limpa"
            // desde que nao seja vazia
            if (!linha.equals(""))
               arquivo.add(linha);
         } // fim do while (se arquivo foi totalmente lido segue para o return
           // caso contrario retorna ao while para le a proxima linha

         }catch (IOException e){
                System.out.println("ARQUIVO " + fonte + "nao encontrado");
              }
       // retorna o arquivo limpo em um objeto ArrayList ( lista dinamica )
      return arquivo;
   }
}
```

Fonte: captura de tela da IDE Netbeans, elaborada pela autora.

Que tal criar uma classe para testar o método leFonte apresentado?

A classe Teste, apresentada na Figura 2.6, lê e mostra o arquivo lido sem espaços.

Figura 2.6 | Classe Teste – chama classe Leitura, a qual elimina espaços em branco

```
1   package JFlex;
2   import java.io.IOException;
3   import java.util.ArrayList;
4   public class Teste {
5       public static void main(String[] args) throws IOException {
6           // ajuste esta linha para o arquivo texto que deseja ler
7           // substituindo "C:/temp/Exemplo.java"
8           // pelo seu arquivo e localização
9           String arq = "C:/temp/Exemplo.java";
10          ArrayList arqFonte = new ArrayList();
11          arqFonte = Leitura.leFonte(arq);   //chama o método de leitura
12          //imprime o arquivo gerado, sem espacos
13          for ( int i = 0; i < arqFonte.size(); i++){
14              System.out.println(arqFonte.get(i));
15          }
16      }
17  }
```

Fonte: captura de tela da IDE Netbeans, elaborada pela autora.

Agora, é com você, que tal testar o exemplo apresentado ?

Os elementos básicos de qualquer linguagem são os identificadores, as palavras-chaves, as constantes e os operadores. Vamos considerar a linguagem Pascal para identificarmos quais são esses elementos básicos nessa linguagem e usar a notação EBNF, estudada na unidade anterior, para especificá-los.

Um elemento básico são as letras. Assim, para identificarmos as letras criaremos a produção <letra> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

Como o Pascal não é *case-sensitive*, isto é, não diferencia maiúscula de minúscula, não é necessário uma produção para <letraMaiuscula> e outra para <letraMinuscula>, assim nossa produção <letra> ficará:

(1) <letra>::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v |w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | W | V | W | X | Y | Z

O próximo elemento serão os dígitos:

(2) <digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Assimile

É fundamental conhecer os padrões da notação EBNF, pois:

- São utilizados na especificação das linguagens de programação
- Foram criados para simplificar a especificação

Fixe as regras desse padrão. O EBNF:

[ ] - indica que os elementos entre [ ] podem aparecer 0 ou 1 vez

{ } - indica que os elementos entre [ ] podem aparecer 0 ou n vezes

| - indica alternância, é o OU lógico

< > - o nome da produção deve aparece entre < >

( ) - utilizado para alterar precedência

Lembre-se:

Token – é um padrão que ocorre na linguagem.

Lexema – é uma instancia do *token*.

Produção – é uma regra pertencente à gramática da linguagem.

Agora podemos definir a regra dos identificadores. Estes começam com uma letra e os demais símbolos podem ser letras ou dígitos:

(3) <identificador>::= <letra>{ [<letra> | <dígitos>] }

Vale lembrar, que { } indica repetições de 0 a n, e [ ] indica que pode aparecer 0 ou 1 vez, de acordo a notação EBNF. Mas, é importante saber que as ferramentas de apoio e as funções existentes em várias linguagens de programação usam expressões regulares e adotam o padrão EBNF com algumas variações, como pudemos observar no exemplo do método *leFonte()* da classe *Leitura*, em **linha = linha.replaceAll("[ ]+", " ")**, em que **[ ]+** indica uma repetição de 1 ou n vezes, e não como dita o padrão EBNF : **" "{[" "]}** . Veja que a notação utilizada para expressões regulares no JAVA é mais simples, pois o padrão EBNF para implementação na linguagem de programação

seria muito complexo, assim as funções e ferramentas utilizam pequenas variações no padrão EBNF. Por esse motivo iremos estudar a notação utilizada pelas ferramentas de apoio na próxima seção.

Dando continuidade à especificação dos elementos básicos de uma linguagem, no caso a linguagem Pascal, falta definirmos as palavras-chaves, as constantes e os operadores, pois já definimos os identificadores. Vamos, portanto, fazer estas especificações:

(4) <constante> ::= <const_numerica> | <const_literal>

Veja alguns exemplos de constante numérica: 10, 344, 2.22. Veja também outros exemplos de constante literal ou alfanumérica: "Maria", "RG 12.345.234-A".

As produções <const_numerica> e <const_literal> precisam ser criadas, e quanto mais simples for cada produção mais fácil será a implementação, pois veremos mais à frente que para cada produção léxica, sendo uma linguagem regular, há um autômato finito determinístico que a reconhece, ou podemos, ainda, representar cada produção léxica da gramática por um diagrama de sintaxe.

Vamos construir as produções auxiliares, para (4) <constante>:

(5) <inteiro> ::= <digito> { <digito> }. Um exemplo de número inteiro conforme especificado por <inteiro> é uma sequência de dígitos não vazia, por exemplo: 123, 3, 999

(6) <decimal> ::= <inteiro> '.' <inteiro>. Um exemplo de número conforme decimal, especificado por <decimal>, é definido por tipo <inteiro> à esquerda do ponto (.) e outro <inteiro> à direita do ponto (.), por exemplo: 12.23, 0.0, 45.0 .

(7) <const_numerica> ::= <inteiro> | <decimal>. Uma constante numérica poder ser do tipo <inteiro> ou <decimal>.

(8) <aspasDupla> ::= " . Essa produção, mesmo representando apenas um símbolo, é útil para a clareza na especificação da gramática, veja a produção (10).

(9) <caracterqq> ::= {[**<alfabeto>**]} . A produção <alfabeto>, neste caso, é uma produção com todos os símbolos possíveis do teclado (128 símbolos), e o tipo <caracterqq> pode ser uma cadeia qualquer de símbolos do <alfabeto>, por exemplo: Maria, 01/10/2010.

(10) <const_literal> ::= <aspasDupla> {[<caracterqq>]} <aspasDupla>. A produção <const_literal> representa uma cadeia

qualquer de símbolos entre aspas duplas, por exemplo: "Maria", "01/10/2010".

Agora, vamos às produções lexicais finais:

(11) <operadoresAritmeticos> ::= + | - | * | /

(12) <operadoresComparação> ::= <> | < | > | <= | >=

(13) <sombolosEspeciais> ::= **(|) | [|] | := | . | , | :**

(14) <palavrasChaves> ::= **div | or | and | not | if | then | else | of | while | do | begin | end | read | write | var | array | function | procedure | program | true | false | char | integer | boolean**

As palavras-chave de uma linguagem tem o mesmo critério, regra, dos identificadores, mas precisam ser reconhecidas e terem um tratamento específico na análise sintática e, portanto, não podem ser utilizadas como identificadores, chamadas também de palavras-reservadas. O projetista da linguagem é quem define as palavras-chave ao associá-las a instruções/comandos que existirão na linguagem que pretende criar.

As produções de (1) a (14) compõem a gramática do léxico para a linguagem Pascal, e observe que as produções (1), (2), (11), (12), (13) e (14), são exclusivamente de símbolos terminais. As demais produções são gramáticas regulares, portanto, pode-se construir autômatos finitos determinísticos para reconhecer essas produções.

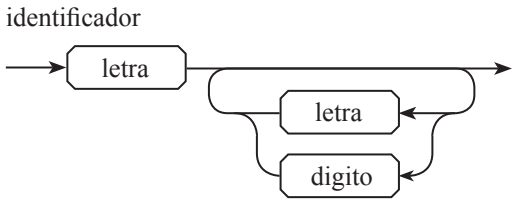
Na implementação do analisador léxico, os tipos de *token* são determinados pelas produções da gramática e o lexema o símbolo reconhecido como o *token*.

Vamos agora estudar como reconhecer um *token* do tipo <identificador>.

Já é conhecida sua produção, conforme especificado no item (3), portanto, <identificador> ::= <letra>{[<letra>|<dígitos]}.

Para ajudar nesse processo, podemos construir um diagrama que represente a produção (3), conforme representada na Figura 2.7.

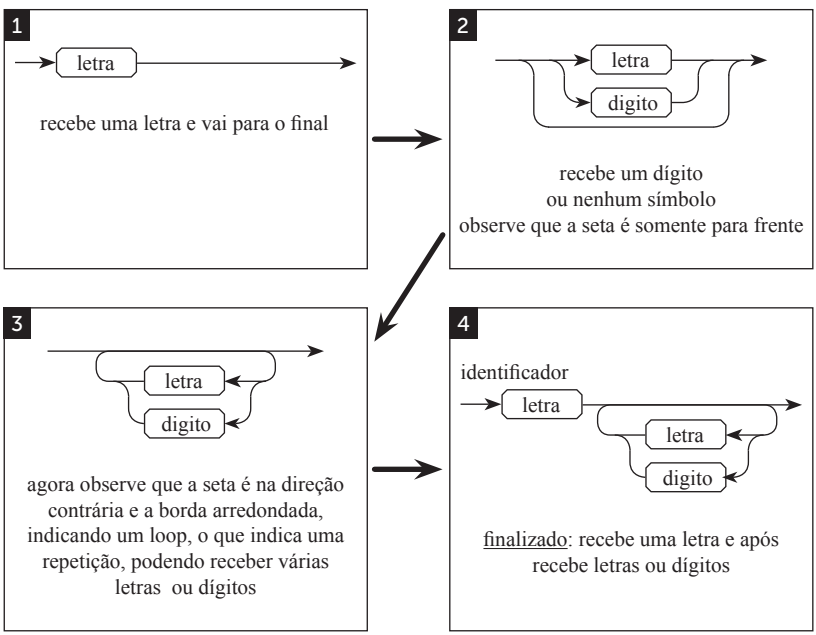
Figura 2.7 | Diagrama de sintaxe da produção (3) <identificador>



Fonte: elaborada pela autora.

O uso de diagramas de sintaxe permite uma representação gráfica da produção. A Figura 2.7 é um diagrama de sintaxe da produção (3) <identificador> ::= <letra>{[<letra>|<dígitos]}. Nele, o tipo <letra> é representado por -[letra]-, e a seta esquerda indica o início da produção, indicado pelas linhas que retornam para a esquerda, e a alternância pela apresentação em paralelo dos tipo <letra> e <digito>. A Figura 2.8 ilustra passo a passo esta explicação.

Figura 2.8 | Passo a Passo da construção do diagrama de sintaxe para a produção (3) <identificador>



Fonte: elaborada pela autora.

Os autômatos finitos determinísticos (AFD) são reconhecedores e facilmente implementados. O JFLEX gera métodos que fazem isso.

Um autômato finito determinístico, segundo Menezes (2005), é definido por uma 5-tupla ordenada $M = (\Sigma, Q, \delta, q_0, F)$, na qual:

Σ é um alfabeto.

Q é o conjunto dos estados possíveis.

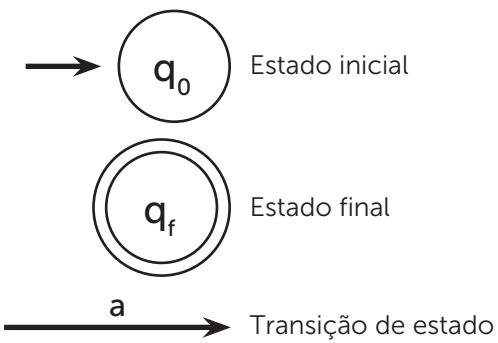
δ é a função transição.

q_0 indica o estado inicial.

F é o conjunto dos estados finais.

Um autômato finito (AF) pode ser representado na forma de diagrama, de acordo com as convenções representadas na Figura 2.9.

Figura 2.9 | Convenções utilizadas para representação de um AF



Fonte: elaborada pela autora.

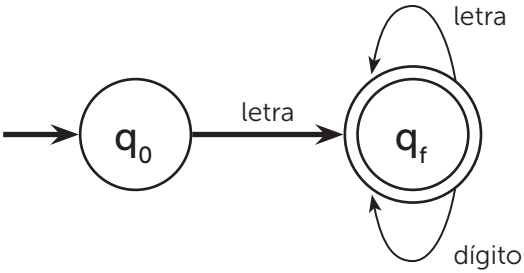
Para a produção (3) <identificador> ::= <letra>{[<letra>|<dígitos]}, o AFD é representado conforme a Figura 2.10. O estado inicial é q_0, ao receber uma <letra> ele muda do estado q_0 para o estado q_f. O estado é um estado final, mas pode receber, após a primeira <letra>, infinitos <letra> e/ou <digito>. Assim, podemos concluir que um <identificador> pode ser:

a) Gerado pela produção (3) <identificador> ::= <letra>{[<letra>|<dígitos]}, que é uma expressão regular.

b) Reconhecido pelo AFD, representado na Figura 2.10.

c) Ser representado pelo diagrama de sintaxe da Figura 2.7.

Figura 2.10 | AFD para <identificador>



Fonte: elaborada pela autora.

Pesquise mais

O aplicativo JFLAP é uma ferramenta experimental para criação de AF. Para saber mais sobre autômatos, você pode baixar este aplicativo em <http://www.jflap.org/>. Acesso em: 1 mai. 2018.

Pode também ler o tutorial sobre o JFLAP, disponível em <http://homepages.dcc.ufmg.br/~nvieira/cursos/tl/a05s2/relatorio_curso.doc>. Acesso em: 28 jun. 2018

Se você quiser construir diagramas de sintaxe usando, um aplicativo pode baixar o programa EBNF Visualizer no site <http://dotnet.jku.at/applications/Visualizer/#Setup>. Acesso em: 1 mai. 2018.

Nesta seção, pudemos trabalhar com a especificação dos elementos léxicos de uma linguagem de programação, praticar a notação EBNF, conhecer uma técnica de bufferização de entradas e aprender como reconhecer um *token*, além de estudar representações gráficas, como o diagrama de sintaxe e o uso de AFD para reconhecimento de *tokens*.

Pronto para colocar tudo isso em prática criando seu primeiro analisador léxico, na próxima seção?

Sem medo de errar

Vamos recordar que você foi procurado por uma grande companhia da área de manufatura para construir uma linguagem simples, para que seus executivos possam usá-la sem a necessidade de intervenção da equipe de desenvolvimento. Portanto, um dos requisitos da linguagem a ser criada por você é a simplicidade e, conforme solicitação da empresa, deverá ter termos similares ao português.

A linguagem hoje utilizada é a SQL padrão, e você deverá construir a nova linguagem, tendo em mente que o compilador final (*backend*) irá traduzir a linguagem que você criará para o SQL padrão. Sugestão: dê um nome para a nova linguagem. Que tal SQLbr. Gostou ?

O primeiro passo para especificar a nova linguagem, a finalidade, foi colocado pela empresa, agora é o momento de se reunir com o cliente e saber quais comandos da linguagem SQL serão usados.

Segundo a empresa, seus executivos precisam apenas realizar consultas nas diversas tabelas existentes no banco de dados, mas o comando SELECT não é muito claro, especialmente em cláusulas com termos como HAVING ou GROUP BY. Assim, a empresa forneceu uma lista dos elementos básicos que poderão ser usados no comando SELECT e sugestões de como ela gostaria que eles existissem na sua linguagem SQLbr :

a) Exemplos de comandos SELECT na linguagem padrão (Figura 2.11).

Figura 2.11 | Lista de comandos SQL padrão que serão utilizados

```
USE <nome da tabela>
SELECT * FROM cliente WHERE codigo = 1
SELECT nome FROM cliente
SELECT codigo, nome FROM cliente
SELECT nome FROM cliente WHERE (uf = 'MG' OR uf IS NULL) AND nome LIKE 'M%' ORDER BY nome DESC
SELECT A.codigo, A.descricao, B.descricao, B.qtde FROM produtos A
        INNER JOIN componentes B ON (A.codigo = B.codproduto)
SELECT codigo, nome FROM clientes UNION SELECT codigo, nome FROM funcionários
SELECT AVG(valor) FROM pedido
SELECT codigo, COUNT(*) FROM pedidos GROUPY BY código HAVING COUNT(*) >= 2
```

Fonte: elaborada pela autora.

b) Na Figura 2.12 há algumas sugestões dos termos desejados para nova linguagem, a SQLbr.

Figura 2.12 | Sugestões de comandos para a linguagem SQLbr

Comando SQL padrão	Sugestão para a linguagem SQLbr
USE	ABRIR_TABELA
SELECT	MOSTRAR
FROM	DE
WHERE	ONDE
ORDER BY	ORDENAR_POR
DESC	DECRESCENTE
ASC	CRESCENTE
INNER JOIN	EM_CONJUNTO_COM
ON	ATRAVES_DA_LIGACAO
UNION	UNIDA_COM
GROUPY BY	AGRUPAR POR
HAVING	FILTRO_DO_GRUPO
COUNT	CONTAR
AVG	MEDIA
MIN	VALOR_MINIMO
MAX	VALOR_MAXIMO
SUM	SOMATORIA
OR	OU
IS	EH
NULL	VAZIO
AND	.E.
LIKE	CONTENDO

Fonte: elaborada pela autora.

Como nessa etapa você irá construir apenas o analisador léxico, não há necessidade de se preocupar com a estrutura dos comandos, apresentados no item (a), mas apenas identificar os elementos básicos da linguagem, tais como as palavras-chave, os identificadores e os operadores lógicos, matemáticos e de comparação.

Nesse ponto, observe como na prática são fundamentais os conhecimentos dos princípios das linguagens, estudados na Unidade 1, e saber que todas as linguagens possuem determinados elementos básicos que fazem parte das linguagens regulares, que, por sua vez, são especificadas na parte léxica da gramática, como estudado nessa seção.

Agora, você pode criar as produções para esses elementos, utilizando a notação EBNF. Mãos à obra?

A especificação completa ficará conforme indicado na Figura 2.13.

Figura 2.13 | Especificação léxica completa para a linguagem proposta SQLbr

```
<letra> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q |
r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | K
| L | M | N | O | P | Q | R | S | T | W | V | W | X | Y | Z
<digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<underline> ::= _
<identificador> ::= <letra>{ <letra> | <digitos> | <underline> }
<constante> ::= <const_numerica> | <const_literal>
<inteiro> ::= <digito> { <digito> }
<decimal> ::= <inteiro> '.' <inteiro>
<const_numerica> ::= <inteiro> | <decimal>
<aspasDupla> ::= "
<caracterqq> ::= {[ <alfabeto> ]}
<const_literal> ::= <aspasDupla> { <caracterqq> } <aspasDupla>
<operadoresAritmeticos> ::= + | - | * | /
<operadoresComparação> ::= <> | < | > | <= | >=
<operadoresLogicos> ::== .E. | .OU.
<simbolosEspeciais> ::= ( | ) | [ | ] | . |, | ;
<palavrasChaves> ::= ABRIR_TABELA | MOSTRAR | DE | ONDE | ORDENAR_POR |
DECRESCENTE | CRESCENTE | EM_CONJUNTO_COM | ATRAVES_DA_LIGACAO | UNIDA_COM |
AGRUPAR POR | FILTRO_DO_GRUPO | CONTAR | MEDIA | VALOR_MINIMO |
VALOR_MAXIMO | SOMATORIA | EH | VAZIO | CONTENDO
```

Fonte: elaborada pela autora.

Trabalho realizado! Mas, que tal mostrar, usando diagrama de sintaxe, algumas produções especificadas ou construir AFDs? Assim você permitirá que a empresa perceba o quanto está capacitado para continuar a trabalhar nas próximas etapas do projeto do compilador, como a implementação do analisador léxico, ora especificado.

Avançando na prática

Reconhecedor de palavras em Grego

Descrição da situação-problema

Você e seus amigos resolveram ir para um campeonato de futebol na Grécia. Um coordenador do campeonato está com dificuldades, pois a rede hoteleira da província onde o campeonato será realizado não tem internet, e em função das pessoas locais só conhecerem o idioma grego, haverá um grande transtorno na comunicação.

Um dos seus amigos se propôs a elaborar um aplicativo para uso no celular sem necessidade de conexão, mas está com dificuldade para estruturar o algoritmo, pois acha o grego muito difícil. Assim, apresentou a você o problema e explicou que precisa saber como identificar palavras somente com letras do alfabeto grego, sem espaços, e validar entre a lista de palavras pré-definida pelo organizador do campeonato. Outra questão é que qualquer palavra começada por letras maiúsculas é válida, desde que contenha apenas letras em grego. Mas não é só isso, ainda será necessário reconhecer frases que serão compostas por um substantivo + verbo + (substantivo ou adjetivo).

Como você pode ajudá-lo a identificar as palavras em grego e ainda reconhecer se as frases estão escritas na estrutura definida? É possível fazer isto?

Resolução da situação-problema

Esse é um típico caso de especificação de um tradutor de palavras e frases simples, que são nada mais que linguagens regulares. Assim que você especificar a análise léxica da gramática, seu amigo poderá facilmente seguir cada produção definida e implementar os autômatos para a construção do algoritmo, viabilizando a criação do aplicativo que ele deseja. O primeiro passo é conhecer o alfabeto grego, mostrado na Figura 2.14, e o correspondente significado no alfabeto latino para referência.

Figura 2.14 | Alfabeto grego

MAIUSCULO	minuscula	equivalente ao alfabeto latino	Pronúncia
Α	α	a	alfa
Β	β	b	beta
Γ	γ	c	gama
Δ	δ	d	delta
Ε	ε	e	épsilon
Ζ	δ	z	dzeta
Η	η	ê	eta
Θ	θ	t	teta
Ι	ι	j	iota
Κ	k	k	capa
Λ	λ	L	lambda
Μ	μ	m	mü
Ν	ν	n	nü
Ξ	ξ	x	ksi
Ο	ο	o	ômicron
Π	π	p	pi
Ρ	ρ	r	rô
Σ	σ	s	sigma
Τ	τ	t	tau
Υ	υ	u	upsilon
Φ	φ	f	fi
Χ	χ	qu	qui
Ψ	ψ	ps	psi
Ω	ω	ô	ômega

Fonte: elaborada pela autora.

Pelo exposto, será necessária uma lista das letras maiúsculas do alfabeto grego e outra das minúsculas para reconhecer palavras, além da lista das palavras pré-definidas pelo organizados.

As palavras pré-definidas na lista são as palavras-chave, que identificaremos por <palavraValida>. Em função da estrutura substantivo + verbo + (substantivo ou adjetivo), devem ser construídas produções auxiliares separando as palavras-chave por tipo, tais como verbo, substantivo e adjetivo. Assim, a especificação final será:

<letraMaiuscula> ::= $\mathrm{A} \mid \mathrm{B} \mid \Gamma \mid \Delta \mid \mathrm{E} \mid \mathrm{Z} \mid \mathrm{H} \mid \ldots \mid \mathrm{X} \mid \Psi \mid \Omega$

<letraMinuscula> ::= $\alpha \mid \beta \mid \gamma \mid \delta \mid \varepsilon \mid \xi \mid ... \mid \chi \mid \psi \mid \omega$

<verboValido> ::= <lista de verbos validos> // lista fornecida pelo cliente

<substantivoValido> ::= <lista de substantivos validos> // lista fornecida pelo cliente

<adjetivoValido> ::= <lista de adjetivos validos> // lista fornecida pelo cliente

<palavraValida> ::= <verboValido> | <substantivoValido> | <adjetivoValido>

<nomeProprio> ::= <letraMaiuscula> { <letraMinuscula> }

<frase> ::= (<substantivoValido>|<nomeProprio>) <verboValido> (<substantivoValido> | <adjetivoValido> | <nomeProprio>)

<**inicio**> ::= <frase> | <palavraValida> | <nomeProprio>

Nesta situação problema, você foi levado a praticar todos os conceitos fundamentais de linguagens formais, demonstrando sua habilidade de associação da teoria com a prática, pois:

a) Como essa gramatica é integralmente regular, a especificação léxica é a própria gramática, assim pode indicar a produção <inicio> como a produção inicial da gramática, e isso demonstra que o conceito formal de definição foi assimilado desde a Unidade 1 e utilizado aqui.

b) Com a especificação léxica dessa gramática, você praticou a notação EBNF e identificou os elementos básicos existentes nesse caso, além de demonstrar flexibilidade e síntese para relacionar as situações e construir produções mais simples, elaborando uma gramática organizada, clara e modular, que facilitará a implementação pelo programador.

Para finalizar, vamos chamar a atenção para a produção <frase>, que exigiu um raciocínio analítico para observar o tipo de token <nomeProprio>, é também um <substantivoValido>, por isto pode estar também no início da frase ou no final. Isso demonstra que você está atento a detalhes, e que você também está desenvolvendo habilidades especificas que o diferenciará no mercado de trabalho.

Faça valer a pena

1. Considere que linguagem L possua apenas os seguintes tipos de *tokens*:
Números inteiros não sinalizados.
Números reais não sinalizados (números com casas decimais simples, usando ponto para indicar a parte decimal).
Operadores aritméticos: +, -, /, *.
Operador de atribuição: =.
Identificadores, onde são válidas apenas letras.
Qual alternativa apresenta produções escritas na notação EBNF e válidas para *tokens* pertencentes à linguagem L?

a) <letras> ::= <identificadores>
b) <operadoresAritmeticos> ::= <Identificador> + <identificador>
c) <atribuição> ::= A + B;
d) <id> ::= { [<letra>] }
e) <identificador> ::= { <letra> }

2. A função exata do analisador léxico é reconhecer os *tokens* associados às expressões regulares.
Assim, por meio das expressões regulares definidas na gramática, o analisador léxico poderá identificar o par (tipo do *token*, lexema).
Assinale a alternativa correta:

a. Lexema, morfema e grafema são elementos da análise sintática de uma linguagem de programação.
b) Lexema é um tipo de padrão e *token* é o nome dado ao lexema.
c) Tipo de *token* é um padrão pertencente à linguagem e o lexema é uma instância do *token*.
d) *Token* e lexema são instâncias do mesmo objeto.
e) *Token* é uma produção e lexema é a semântica.

3. Uma grande quantidade de tempo é consumida lendo o programa-fonte e separando em *tokens*. Assim, o uso da técnica de bufferização nesse processo acelera o desempenho do compilador. Uma estratégia da técnica de bufferização é usar um *buffer* duplo, dividi-lo em duas partes e limitar o número de retrocessos.
Sobre a técnica de bufferização, é correto afirmar:

a) Foi desenvolvida para obter uma melhor eficiência no processo de varredura caractere a caractere no processo de reconhecimento dos *tokens*.
b) A bufferização apenas usa funções para carregar vários caracteres para a memória antes de efetivamente os analisar.
c) Não é mais utilizada, por apresentar erros no reconhecimento dos *tokens*.
d) Consome mais memória, além de gerar uma busca circular quando apresenta alguma ambiguidade no reconhecimento do token.
e) Sem bufferização não é possível a construção de um analisador léxico.

Seção 2.2

Construção de um analisador léxico

Diálogo aberto

Caro aluno, nas seções anteriores você tomou conhecimento que construir um compilador é uma tarefa complexa e por isso ela está dividida em fases. A primeira fase trata da verificação da grafia do programa, por isso é comum dizer: o compilador da linguagem C analisa se o programa foi escrito corretamente. Já a segunda fase traduz para a linguagem alvo.

Assim, na seção anterior, você especificou a parte léxica da gramática da linguagem solicitada pela empresa que o procurou, pois esse é justamente o primeiro passo para viabilizar o objetivo final, que é construir o analisador léxico para esta nova linguagem, a qual deverá ser mais simples que o SQL padrão, além de usar termos similares ao português.

A companhia gostou muito da proposta apresentada para a nova linguagem e considera que ela atenderá suas necessidades, possibilitando a seus funcionários melhorar a produtividade, portanto, deseja que você inicie a implementação o mais rápido possível. Sendo assim, espera que, nesta etapa, faça a implementação da especificação léxica definida, ou seja, do analisador léxico da nova linguagem.

Para desenvolver o analisador léxico (*lexer*), a empresa deseja que você use a linguagem JAVA e o gerador de analisador léxico (*scanner*) JFLEX para agilizar o desenvolvimento, pois eles têm urgência para o uso do aplicativo.

Você deverá entregar o código fonte, o .jar, e os testes realizados para validação do gerador léxico da nova linguagem. Feliz em ter alcançado sucesso na sua primeira fase do desenvolvimento de um compilador e motivado para criar o *lexer* de acordo com sua especificação?

Para ajudá-lo nessa construção, iremos estudar nessa seção o gerador de analisador léxico JFLEX, como instalá-lo, quais as regras especificas da notação utilizada por esse *scanner,* a sua integração com a *Integrated Development Environment* (IDE) NetBeans e o JAVA.

Por fim, você iniciará a construção do compilador. E, após ter assimilado os conceitos teóricos de linguagens formais e a estrutura de um compilador, com certeza sentirá facilidade em desenvolver a primeira fase, o analisador léxico, provavelmente com muita vontade de utilizar o JFlex para agilizar esse processo. Mãos à obra!

Não pode faltar

Na Seção 1.3, estudamos os pontos importantes a serem considerados antes de iniciarmos o desenvolvimento de um compilador, e entre os fatores relevantes está o uso de ferramentas que auxiliam no desenvolvimento dos analisadores léxicos e sintáticos. Essa seção tratará exclusivamente dos geradores de analisadores léxicos, também conhecidos como *scanners*.

Um gerador de analisador léxico tem como entrada uma especificação com um conjunto de expressões regulares e ações correspondentes. Ele gera um programa (um *lexer*) que lê a entrada, a combina com as expressões regulares no arquivo de especificação e executa a ação correspondente, caso haja correspondência. Geralmente, os lexers são a etapa inicial em compiladores, palavras-chave, comentários, operadores, etc., e geram um fluxo de *tokens* de entrada para analisadores. Eles também podem ser usados para muitos outros propósitos.

Segundo Klein, um gerador de analisador léxico (*scanner*) tem como entrada uma especificação com um conjunto de expressões regulares. Ele gera um programa (*lexer*) que lê a entrada, o programa escrito na linguagem (o fonte) e retorna o padrão de cada tipo de *token* identificado, ou se não é válido, isto é, não é reconhecido pela gramática.

Como os *scanners* fazem isto? Como estudado nas Seções 1.2 e 2.1, as linguagens regulares podem ser especificadas por expressões regulares e há um autômato finito determinístico (AFD) que as representa, assim, o algoritmo implementado pelos *scanners* a partir das expressões regulares especificadas no arquivo de entrada implementa os AFDs para reconhecer as palavras, ou seja, os padrões dos *tokens* definidos na especificação.

Portanto, sendo o *lexer* a etapa inicial da implementação dos compiladores, terá como função reconhecer padrões básicos

de uma linguagem, por exemplo: palavras-chave, comentários, operadores, etc. Na seção anterior, tomou-se conhecimento que o analisador léxico gera um fluxo de tokens de entrada para os analisadores sintáticos, os *parsers*.

Os *lexers* também podem ser usados para muitas outras finalidades. Várias linguagens utilizam expressões regulares para separar, buscar ou realizar trocas de palavras em um texto, além de reconhecer padrões, tais como CPF, e-mails e URLs. Nesta seção, mostraremos exemplos para essas finalidades quando estudarmos as expressões regulares de acordo com o JFlex.

Na Seção 1.3, vimos que existem vários *scanners*, tal qual o Lex para o ambiente Unix e linguagem C, o JFLex para o ambiente Unix/Windows e linguagem Java e as ferramentas integradas JAVACC ou ANTLR, ambas para ambiente Windows e linguagem JAVA.

Optamos pelo aprofundamento no *scanner* JFlex por ser um dos analisadores que gera *lexers* mais rápidos e, também, pela notação/sintaxe de sua especificação ser uma variação próxima ao padrão EBNF e de outro *scanner*, o Lex, além de ser uma extensão ampliada do gerador JLex.

O JFlex foi projetado para trabalhar em conjunto com o gerador de analisador sintático (*parser*), o CUP, mas também pode ser usado com outro gerador de *parser*, o ANTLR.

Pesquise mais

Caso queira conhecer mais sobre outro *scanner*, recomendamos o JAVACC (Java *Compiler Compiler*). No site oficial dessa ferramenta, há uma vasta documentação.

Disponível em: <https://javacc.org/>. Acesso em: 10 maio 2018.

Há, ainda, o livro do professor Delamaro (2004), e o material disponibilizado em seu site:

DELAMARO, M. **Como Construir um Compilador Utilizando Ferramentas Java.** Novatec Editora: [S.l.]. 2004. Disponível em: <http://conteudo.icmc.usp.br/pessoas/delamaro/SlidesCompiladores/CompiladoresFinal.pdf>. Acesso em: 10 maio 2018.

Vale a pena conhecer.

Para implementar um analisador léxico, vamos precisar:

1. Do kit JDK 8.0 ou superior para o JAVA SE. Disponível em:<http://www.oracle.com/technetwork/java/javase/downloads/jdk10-downloads-4416644.html>. Acesso em: 8 maio 2018.
2. Da IDE NetBeans 8.1. Disponível em: <https://netbeans.org/community/releases/81/index_pt_BR.html>. Acesso em: 08 maio 2018.

Atenção: o kit JDK para o Java deve ser instalado antes da instalação do Netbeans.

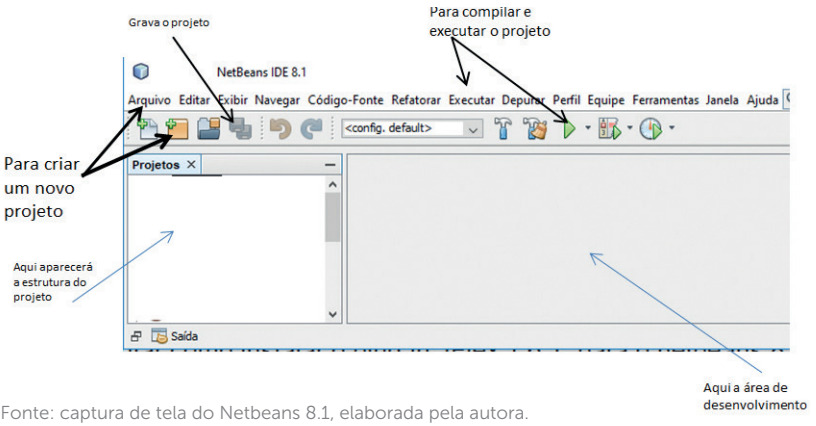
3. Do gerador JFlex 1.6.1. Disponível em: <http://jflex.de/download.html>. Acesso em: 8 maio 2018.

Depois de baixados os softwares indicados, eles devem ser instalados nesta ordem: primeiro o JDK, depois o Netbeans. Siga as instruções recomendadas no site de cada ferramenta baixada. Agora, vamos orientar como instalar o plug-in JFlex 1.6.1 para o Netbeans 8.1.

Você poderá visualizar o processo de instalação por meio de um tutorial, em vídeo, no link <https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/compiladores/u2/s2/U2S2_video_1.mp4> ou por meio do QR Code.

Após baixar o JFLEX 1.6.1.zip, faça a descompactação e abra o Netbeans. O ambiente Netbeans terá a aparência da Figura 2.15.

Figura 2.15 | Ambiente Netbeans 8.1



Fonte: captura de tela do Netbeans 8.1, elaborada pela autora.

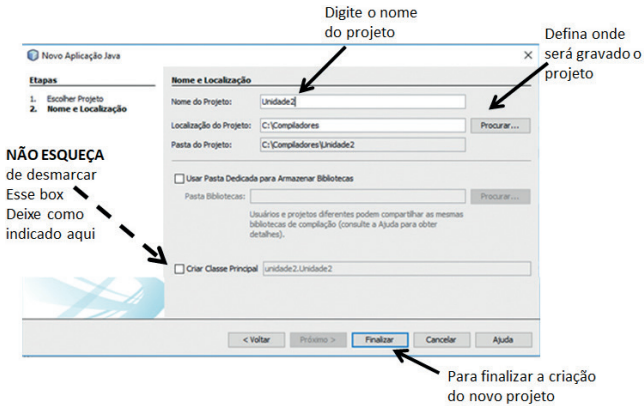
Vamos criar um novo projeto clicando em , ou CTRL+SHIFT+N.

Em Categorias, escolha JAVA Categorias: Java.

Em seguida, em Projetos, escolha Aplicação Java Projetos: Aplicação Java. Siga para o próximo passo, clicando em Próximo >.

Nessa nova tela, você indicará o nome do projeto, o qual iremos denominar **Unidade2**, e o local onde irá gravá-lo. Lembre-se de **desmarcar** o *box* Criar Classe Principal. Pronto, podemos finalizar. Veja a Figura 2.16, que mostra a tela com a finalização da criação do projeto.

Figura 2.16 | Finalização da criação de um novo projeto no Netbeans



Fonte: captura de tela do Netbeans 8.1, elaborada pela autora.

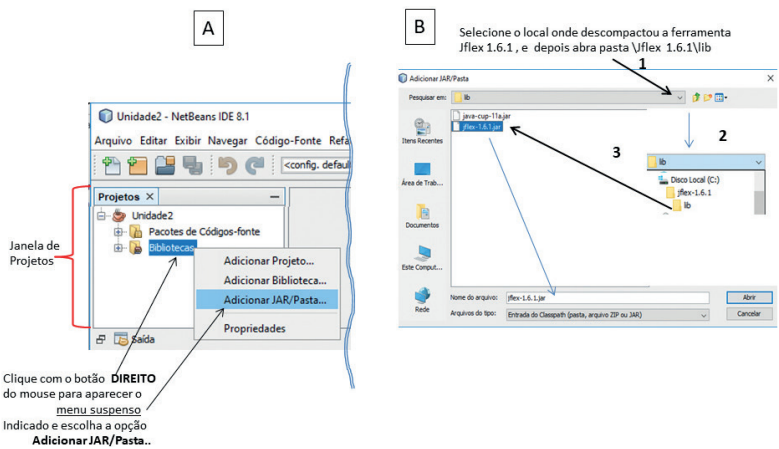
Pesquise mais

Para conhecer mais sobre a ferramenta IDE Netbeans, recomendamos consultar o tutorial abaixo:

NETBEANS. **Tutorial para Início Rápido do Java do NetBeans IDE**. [S.l.; s.d.]. Disponível em: <https://netbeans.org/kb/docs/java/quickstart_pt_BR.html>. Acesso em: 12 maio 2018.

O projeto criado aparecerá na interface da IDE Netbeans, na janela projeto, localizada à esquerda, como pode ser visto na Figura 2.17 (A). O projeto tem a estrutura de uma árvore hierárquica. Nesse ponto podemos adicionar o plug-in JFlex ao projeto, clicando com o botão direito do mouse na pasta **biblioteca**, e, em seguida, com botão esquerdo na opção **adiciona JAR/Pasta**, como indicado na Figura 2.17 (B). Você irá apontar para o arquivo 'jflex1.6.1.jar', o qual foi baixado e descompactado, conforme orientado anteriormente. Para concluir, certifique-se de que o 'jflex1.6.1.jar' aparece na caixa de texto **Nome do arquivo**, conforme indicado na Figura 2.17 (B), e clique no botão Abrir. O plug-in JFlex 1.6.1 foi adicionado ao seu projeto e poderá ser utilizado.

Figura 2.17 | Adicionando o JFlex a um projeto Java



Fonte: captura de tela do Netbeans, elaborada pela autora.

Assimile

Você irá desenvolver uma aplicação em uma linguagem orientada a objeto, o Java. Assim, é importante entender e incorporar os conceitos a seguir, que serão usados no desenvolvimento do compilador:

Plug-in – é um pequeno programa, que adiciona funcionalidades a programas maiores. Também pode ser chamado de extensão, pois amplia os recursos.

Conceitos de orientação a objeto

Pacote – é um conjunto de objetos agrupados por finalidade.

Classe – é o esquema do objeto.

Métodos – são as ações implementadas na classe.

Objeto – é a instância de uma classe, ou seja, o esquema tornou-se um objeto real.

Enum – é um tipo de dados em Java que permite trabalhar com conjuntos de constantes pré-definidas.

Veja o exemplo para relembrar e assimilar os conceitos, citados:

Figura 2.18 | Exemplo básico em Java

```
1   package Exemplo;   // pacote Exemplo
2   // Classe Assimile
3   public class Assimile {
4       private double raio;   // atributo
5       // construtor da classe
6       public Assimile(double raio){
7           this.raio = raio;
8       }
9       // metodo
10      public double area(){
11          return (3.14*raio*raio);
12      }
13      public static void main(String args[]){
14          //instanciando a classe
15          Assimile objeto = new Assimile(5);
16          System.out.println("Area = "+ objeto.area());
17          // exemplo com Enum
18          Token tipo =  Token.LITERAL;
19          switch (tipo) {
20              case LITERAL:
21              case ID:
22              }
23      }
24  }
25  ------------------------------------------
1   public enum Token {
2       ID, INT, DEC, LITERAL, ERROR;
3   }
```

Fonte: captura de tela do Netbeans 8.1, elaborada pela autora.

Vamos criar no nosso projeto Unidade2 um pacote que chamaremos de **Compilador**. Para isso, clique com o botão direito do mouse sobre o nome do projeto, no nosso caso Unidade2, depois leve o mouse até a opção Novo>, mova-o sobre a nova lista de opções que aparecerá à direita e escolha Pacote Java. Digite "Compilador" na caixa de texto do nome do pacote e, em seguida, clique em finalizar.

Agora vamos **criar um arquivo java vazio** para construirmos a especificação dos *tokens*, ou seja, a gramática do léxico. Clique com o botão direito do mouse no pacote Compilador 

, depois leve o mouse até a opção Novo>. Com o mouse sobre a opção Novo, mova-o sobre a nova lista de opções que surge a direita, e escolha arquivo vazio. Nomeie o arquivo como "**especificacao.flex**". Nesse arquivo iremos escrever a especificação da gramática do léxico, de acordo com as regras das expressões regulares do JFLEX.

A estrutura do JFlex para construir uma especificação é:

(1) *Aqui podem ser declarados os comandos que serão colocados no início da classe.*

%%

(2) *No bloco entre %% ..... %% são declaradas as customizações e as regras.*

Esse bloco é o mais importante. Iremos detalhar este item (2) mais à frente.

%%

(3) *Declara-se a regras sintáticas e os comandos Java associados.*

No ponto (1), insere-se a declaração do pacote em que a classe gerada pelo JFlex será colocada e as importações necessárias.

No ponto (2), pode-se:

(a) Inserir métodos que serão incluídos na classe que será criada pelo JFlex, e, para isso, é necessário que o código a ser inserido esteja dentro do bloco:

%{

... digite aqui o código que deseja inserir no analisador léxico

%}

(b) Há uma série de diretivas do JFlex, e as mais importantes são:

%class *nomedaclasse // define o nome do* ***lexer*** *que se deseja* criar

%type *tipo // define o tipo do objeto que o método yyLex() irá retornar*

// yyLex() é um método do lexer que retorna o tipo do token reconhecido

%line // habilita o método yyLine() para retornar a linha

%column // habilita o método yychar() para retornar a linha

%unicode // habita o padrão Unicode. É default

(c) Definir as expressões regulares

Na Figura 2.19, temos uma série de regras da notação reconhecida pelo Jflex para as expressões regulares.

Figura 2.19 | Regras para expressões regulares no JFlex

a | b - indica reconhece a ou b;
ab - indica concatenação de ab;
a* - indica o fecho de Kleene, ou seja, são reconhecidas palavras com 0 ou n concatenações de **a**. Por exemplo: ε , a, aa, aaa, aaaa, e assim sucessivamente;
a+ - indica a* - { ε }
a? - indica que reconhece **a** ou vazio, ou seja, **a** é opcional;
!a - é a negação, pode ser qualquer outro símbolo do alfabeto, exceto **a**
~a - considera todos os caracteres até a primeira ocorrência de '**a**', chamado de *upto*
a{n} - indica n ocorrências de 'a', por exemplo a{4} indica aaaa;
a{n,m} - o primeiro parâmetro indica o número de ocorrências obrigatório, e o segundo o número de ocorrências opcionais, por exemplo a{2,4} indica que a palavra reconhecida poderá ser aa, aaa ou aaaa.
(a) - representa **a**. Utilizamos os () para alterar precedência na expressão regular.
[aeiou] - indica que a palavra pode conter 'a' ou 'e' ou 'i' ou 'o' ou 'u', a que ocorrer primeiro.
[a-g] - indica um intervalo continuo, ou seja, reconhece uma letra no intervalo entre 'a' e 'g'.
. - o ponto(.), indica qualquer carácter, exceto o \n (salto de linha)

Fonte: elaborada pela autora.

Finalmente chegamos ao detalhamento do ponto (3) da estrutura do arquivo de especificação do JFlex, que corresponde às regras sintáticas e os comandos Java associados. Essa seção trata exclusivamente da analise léxica, portanto apresentaremos somente os comandos e regras associados aos *tokens*:

{nomeDaExpressaoRegular } {comandos Java que se deseja inserir no lexer gerado pelo JFlex}

Exemplificando

Os *tokens* básicos da maioria das linguagens de programação são: (1) regras de nomes de variáveis, (2) números, (3) comentários, (4) palavras-chave.

A especificação desses elementos, seguindo as regras do JFlex e considerando uma mini linguagem C como exemplo, nos dá a seguinte gramática do léxica:

1) A regra de nome de variável pode começar com "_" (*underline*) ou letra, e poderá ter infinitas letras, números ou *underline* após o primeiro carácter. Assim, a expressão ficará:

   nomeVariavel = [_a-zA-Z][_a-zA-z0-9]*

   [_a-zA-Z] - indica que o *token* do tipo nomeVariavel deve começar por "_" (underline), por uma letra minúscula entre a-z ou, ainda, por uma letra maiúscula entre A-Z.

   [_a-zA-z0-9]* - os símbolos entre [ ] são os aceitáveis e podem se repetir 0 ou n vezes. Isso é indicado pelo *, o fecho de Kleene.

2) Números podem ser inteiros ou decimais. A expressão será:

   inteiro = [0-9]+

   [0-9]+ indica que aceita dígitos (símbolos) de 0 até 9, e a palavra terá comprimento 1 ou n.

   decimal = [0-9]+["."]+[0-9]+

   Nesse exemplo, temos uma série de dígitos seguidos de um ponto (".") e, após o ponto, outra série de dígitos. Essa é a estrutura de um número decimal, por exemplo: 123.00, ou 478.3333

3) **O comentário em C tem a duas estruturas:**

   a) ***/ qualquer texto /*, ou**

   b) **// qualquer texto**

   Aplicando as regras do JFlex, teremos a seguinte expressão:

   blocoComentario = "/*" ~"*/"

   No qual "/*" indica que o *token* é iniciado com os símbolos /*, e ~"*/" indica que podem ocorrer quaisquer símbolos até que sejam encontrados os símbolos /*.

   Para reconhecer uma linha única de comentário, vamos criar duas

expressões, o que facilita a construção da produção final. Assim, teremos:

branco = [\t\n\r]+

linhaComentario = {branco}* "//" .*

Os metacaracteres \t, \n, \r, indicam, respectivamente: tabulação, salto de linha, e *carrige return* (volta o cursor ao inicio da linha). Assim, a produção [\t\n\r]+ reconhece linhas em branco e espaços em branco (observe que após o 'r' há um espaço).

{branco}* indica que pode haver espaços ou linhas antes dos símbolos "//", e **"//"** indica a obrigatoriedade destes símbolos no inicio deste tipo de *token* (comentário).

O **"."** **(ponto)** indica que aceita qualquer símbolo, e o asterisco indica que pode ocorrer repetição. Novamente aqui, o fecho de Kleene.

Atenção: O "." (ponto) não reconhece o símbolo \n (salto de linha).

4) **Para concluir nossa gramática, vamos considerar uma lista com apenas cinco palavras-chave do C: *if*, *class*, *int*, *while* e *do*.**

Observe que as palavras-chave são casos especiais de nomeDeVariáveis, pois constituem palavras compostas apenas por letras, mas devemos classificá-las como tipos diferentes do padrão nomeDeVariavel, assim, será necessário identificá-las na última seção do arquivo de especificação e antes da classificação do nomeDeVariável.

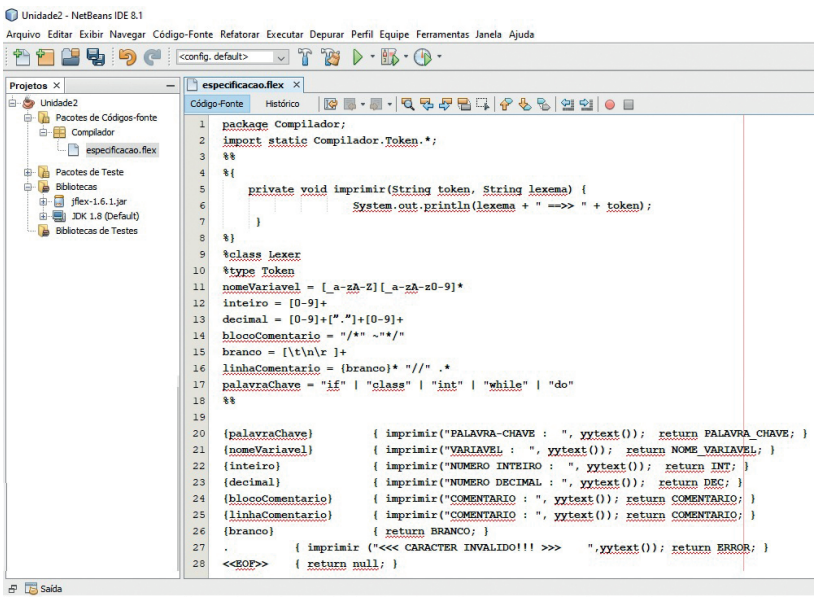
Após as definições acima, nosso arquivo de especificação ficará como apresentado na Figura 2.20.

Você, também, poderá acessar essa especificação, com todos os comentários apresentados, no link <https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/compiladores/u2/s2/U2S2_CODIGO_2_especificacao.pdf> ou por meio do QR Code.

Pronto, a especificação esta concluída!

Agora podemos retornar ao nosso projeto Unidade2, que construímos no NetBeans e digitarmos a especificação, conforme orientado no quadro '**exemplificando**', no arquivo **especificacao.flex** criado anteriormente. Veja a guia exemplificando, na Figura 2.20, que mostra o projeto com a especificação digitada. Você também pode acessar esse código por meio do QR Code ou do link indicados anteriormente.

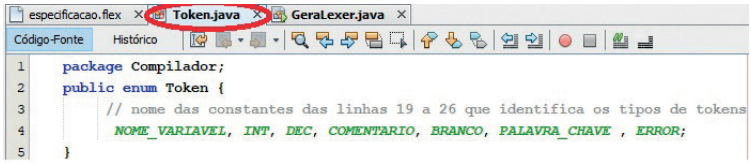
Figura 2.20 | Projeto do analisador léxico com a especificação no padrão JFlex para o mini-C exemplificado



Fonte: captura de tela do NetBeans 8.1, elaborada pela autora.

Para concluir e testar o analisador léxico, é preciso construir a classe *Token* do tipo eNum, com as constantes declaradas na especificação para cada *token*. Veja o nome das constantes nas linhas 20 a 27 da Figura 2.20. Assim, a classe *Token* será o indicado na Figura 2.21.

Figura 2.21 | Classe *Token*



Fonte: captura de tela do NetBeans 8.1, elaborada pela autora.

Para incluir essa classe no projeto, deve-se clicar com o botão direito do mouse sobre o pacote Compilador, mover o mouse para a opção Novo, selecionar Classe Java, identificar o nome da classe como *Token* e clicar em Finalizar. Em seguida, deve-se digitar o código da classe *Token*, conforme apresentado na Figura 2.21, na guia Token.java.

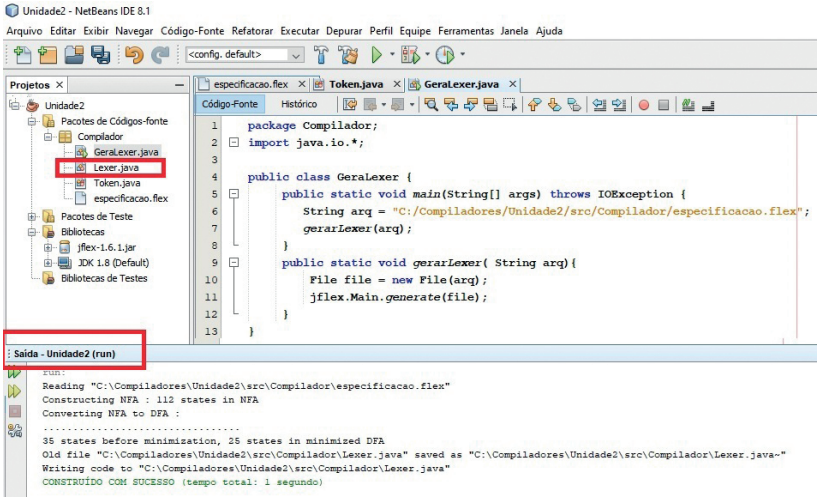
O próximo passo será criar uma classe para que o JFlex leia o arquivo **especificação.flex** e gere a classe que será o nosso *Lexer*.

Repita os mesmos passos indicados para criar a classe *Token*, e nomeie esta nova classe como **GeraLexer** e digite o código apresentado na Figura 2.22, que chama o JFlex para criar o analisador léxico.

Você também poderá visualizar o código proposto para a classe GeraLexer por meio do link <https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/compiladores/u2/s2/U2S2_CODIGO_3_GeraLexer.pdf> ou por meio do QR Code.

Pronto! Agora você pode executar o projeto pressionando a tecla <F6>, e o resultado será uma nova classe no seu projeto, denominada **Lexer**, e a tela de saída será a indicada na Figura 2.22.

Figura 2.22 | Classe GeraLexer



Fonte: captura de tela do NetBeans, elaborada pela autora.

Para testar o projeto, é necessário criar uma classe Java. Assim, iremos denominá-la TesteAnalisador.java e também devemos criar um arquivo texto com palavras para serem analisadas pelo *Lexer* construído. A Figura 2.23 apresenta a classe TesteAnalisador.

Figura 2.23 | Classe TesteAnalisador

```
package Compilador; // nome do nosso pacote
import java.io.*;  // iremos precisar desta classe para leitura do arquivo de teste

public class TesteAnalisador {
    /* o throws IOException é necessário para tratar erro de exceção,
       se o arquivo de entrada não for localizado
    */
    public static void main(String[] args) throws IOException {
        /* arq recebe Unidade:/caminho/nomedoarquivoDeEntrada
           substitua "C:/temp/MiniC.TXT"
                     pelo local e nome do seu arquivo de entrada
        */
        String arq = "C:/temp/MiniC.TXT";
        /* o Jflex pode ler String ou um fluxo de dados(um arquivo),
           nesse caso usamos o Reader
        */

        BufferedReader in = new BufferedReader(new FileReader(arq));
        /* Lexer é a classe criada pela especificação.flex, o analisador
                e analise é uma instancia da classe Lexer */

        Lexer analise = new Lexer(in);
        /* criamos um loop para ler todos os tokens identificados
           no arquivo de entrada */
        while (true){
            // o método yylex() do objeto analise, retorna uma constante

            Token token = analise.yylex();  /* token conterá um tipo de padrão
                                               reconhecido pelo lexer */
            if (token==null) break;      /* EOF
                                            se o token for null indica
                                            fim do arquivo de entrada
                                            e o processo é encerrado(break)
                                         */
        } //fim do loop
    }// fim método main
}//fim da classe
```

Fonte: elaborada pela autora.

Você, também, poderá visualizar o código proposto para a classe TesteAnalisador por meio do link <https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/compiladores/u2/s2/U2S2_CODIGO_4_TesteAnalisador.pdf> ou por meio do QR Code.

Depois de criar a classe TesteAnalisador, é o momento de criar o arquivo de entrada para testar a aplicação. Lembre-se de que o arquivo de teste é o programa o qual o compilador irá analisar. Nesta seção, foi implementado somente o analisador léxico, portanto é recomendável que você crie um arquivo de teste somente com palavras válidas para os tipos de *tokens* que são analisados pela especificação criada, além de alguns casos de *tokens* que não devem ser reconhecidos como válidos, como o exemplo de arquivo de teste mostrato na Figura 2.24.

Feito o arquivo de teste, podemos finalmente executar o projeto. Selecione a guia TesteAnalisador.java ou clique duas vezes no nome dessa classe no projeto e pressione <shift><F6>. A Figura 2.24 apresenta, à esquerda, o arquivo de teste (a entrada) e, à direita, a saída correspondente. A saída mostra o lexema e o tipo do *token*, à sua frente.

Figura 2.24 | Arquivo de entrada (teste) e o resultado da análise léxica

ARQUIVO DE TESTE	SAÍDA CORRESPONDENTE

soma
nota1
234teste
12.50
100
/* comentario com varias
linhas
final */
// comentario de uma linha
if
class
int while do
preco_de_venda
preco.maximo
	soma ==>> VARIAVEL :
	nota1 ==>> VARIAVEL :
	234 ==>> NUMERO INTEIRO :
	teste ==>> VARIAVEL :
	12.50 ==>> NUMERO DECIMAL :
	100 ==>> NUMERO INTEIRO :
	/* comentario com varias
	linhas
	final */ ==>> COMENTARIO :
	// comentario de uma linha ==>> COMENTARIO :
	if ==>> PALAVRA-CHAVE :
	class ==>> PALAVRA-CHAVE :
	int ==>> PALAVRA-CHAVE :
	while ==>> PALAVRA-CHAVE :
	do ==>> PALAVRA-CHAVE :
	preco_de_venda ==>> VARIAVEL :
	preco ==>> VARIAVEL :
	. ==>> <<< CARACTER INVALIDO!!! >>>
	maximo ==>> VARIAVEL :
	CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)

Fonte: elaborada pela autora.

Reflita

(1) Por que ao pressionar F6 o projeto apenas executa a classe GeraLexer?

Lembre-se que existem duas classes no projeto com o método *main*, logo apenas uma é defina para iniciar o projeto.

Você pode selecionar a classe pela qual deseja iniciar o projeto e pressionar <shift>+F6, e pode arterar sua opção por meio da propriedade do projeto da seguinte forma:

Clique com o botão direito do mouse no nome do projeto, selecione propriedades, depois selecione executar e em seguida altere a classe principal para a classe que desejar.

Qual método será mais produtivo, nesse caso?

(2) Na classe TesteAnalisador a linha String arq = "**C:/temp/MiniC.TXT**";

Isso indica o exatamente o quê?

Para testar seu projeto, qual o nome do seu arquivo de entrada?

Em qual pasta você gravou esse arquivo?

Você alterou a linha indicada de acordo com suas respostas para o nome do arquivo o qual criou e o local?

Você, também, poderá baixar o projeto Unidade2 completo por meio do link <https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/compiladores/u2/s2/US2S2_Projeto_Unidade2_5.zip> ou por meio do QR Code.

Nesta seção, você pôde praticar como construir um analisador léxico utilizando a ferramenta JFlex, bem como viu que as expressões e regras do JFlex são parecidas com os padrões da notação EBNF, com pequenas variações, e que o uso de uma ferramenta permite focar a construção do analisador na montagem da especificação da gramática. Além disso, viu que o fato da construção dos compiladores estar divida em fases permite testar cada uma delas antes de avançar para a próxima, assim é possível realizar testes para o *lexer*, mesmo não existindo o analisador sintático, o que garante qualidade na continuidade do projeto na fase seguinte, o analisador sintático.

Pronto para a próxima fase de análise sintática, que será o tema da próxima seção? Avante!

Sem medo de errar

A empresa que o procurou para o desenvolvimento de uma linguagem simples e com termos similares ao português gostou muito da gramática apresentada e agora espera ansiosamente pela implementação do analisador léxico, de acordo com a proposta entregue na primeira etapa.

Assim, o primeiro passo para começar o desenvolvimento do analisador léxico é, a partir da especificação feita na primeira etapa (Seção 2.1), adaptá-la para as regras do JFlex estudadas. Assim, temos que:

a) A gramática do léxico desenvolvido conforme o padrão EBNF foi:

```
1   <letra> ::= a | b | c | ... | Z
2   <digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
    8 | 9
3   <underline> ::= _
4   <identificador> ::= <letra>{ <letra> | <dígi-
    tos> | <underline> }
5   <constante> ::= <const_numerica> | <const_li-
    teral>
6   <inteiro> ::= <digito> { <digito> }
7   <decimal> ::= <inteiro> '.' <inteiro>
```

```
8   <const_numerica> ::= <inteiro> | <decimal>
9   <aspasDupla> ::=  "
10  <caracterqq> ::= {[ <alfabeto>  ]}
11  <const_literal> ::= <aspasDupla> {  <carac-
    terqq>  } <aspasDupla>
12  <operadoresAritmeticos> ::= + | - | * | /
13  <operadoresComparação> ::= <> | < | > | <= |
    >=
14  <operadoresLogicos> ::==  .E.  |  .OU.
15  <sombolosEspeciais> ::= (   |   )   |   [   |
    ]   |  .  |  ,  |  ;
16  <palavrasChaves> ::=  ABRIR_TABELA | MOSTRAR
    | DE | ONDE | ORDENAR_POR | DECRESCENTE |
    CRESCENTE | EM_CONJUNTO_COM | ATRAVES_DA_LI-
    GACAO |  UNIDA_COM | AGRUPAR POR |  FILTRO_
    DO_GRUPO | CONTAR | MEDIA | VALOR_MINIMO |
    VALOR_MAXIMO | SOMATORIA | EH | VAZIO | CON-
    TENDO
```

b) Agora, deve-se adaptar a gramática do item (a) às regras do JFlex, que ficará assim:

```
1   letra = [a-zA-Z]
2   digito = [0-9]
3   underline = "_"
4   identificador =  {letra} ({letra} | {digito} |
    {underline})*
5   constante = {const_numerica} | {const_lite-
    ral}
6   inteiro = {digito}+
7   decimal = {inteiro}".”{inteiro}
8   para aspas duplas usamos o metacaracter \”
9   e o <qqcaracter> no caso de literais basta
    negar o salto de linha \n, o retrocesso \n e
    a própria aspas dupla, assim expressão regu-
    lar ficará [^\”\n\r]
```

```
10  e devemos ter cuidado com o símbolo \, e para
    representa-lo, também usamos metacaracter \\,
    portanto \\\" estamos aceitando o símbolo \ e
    o símbolo "
11  const_literal =  \" ( \\\" |  [^\"\n\r] )* \"
12  operadoresAritmeticos = "+" | "+" | "-" | "/"
    | "*"
13  operadoresComparacao = "<>" | "<" | ">" |
    "<=" | ">="
14  operadoresLogicos ==  ".E."  |  ".OU."
15  simbolosEspeciais = "(" | ")" | "[" |  "]" |
    "." | "," | ";"
16  palavrasChaves =  "ABRIR_TABELA" | "MOSTRAR"
    | "DE" | "ONDE" | "ORDENAR_POR" | "DECRES-
    CENTE" | "CRESCENTE" | "EM_CONJUNTO_COM" |
    "ATRAVES_DA_LIGACAO" | "UNIDA_COM" | "AGRUPAR
    POR" | "FILTRO_DO_GRUPO" | "CONTAR" | "MEDIA"
    | "VALOR_MINIMO" | "VALOR_MAXIMO" | "SOMATO-
    RIA" | "EH" | "VAZIO" | "CONTENDO"
```

Você observou que há uma correspondência direta entre a notação EBNF e a do JFlex? E que poucos são os casos especiais? Observe que nas linhas 8, 9 e 10 há a indicação de algumas situações especiais. Portanto, quando se estuda a notação EBNF, a curva de aprendizagem das regras das diversas ferramentas existentes para criar analisadores é mais rápida.

O segundo passo para alcançar o objetivo proposto no desafio será construir o projeto na linguagem Java, usando a IDE NetBeans e o JFlex.

Os passos para criação do projeto serão os mesmos utilizados no projeto Unidade2, exposto nessa seção. Sugerimos denominar o **projeto como SQLbr**, o **pacote como Lexico**, e o **arquivo vazio** para especificação de **gramatica.flex**. Para as demais classes, recomendamos usar os mesmos nomes do projeto Unidade2, ou seja *Token*, GeraLexer e TesteAnalisador.

O arquivo gramatica.flex, final ficará, assim :

```
package Lexico;
import static Lexico.Token.*;
```

```
%%
%{
   private void imprimir(String token, String
lexema) {
          System.out.println(lexema + " ==>> " +
token);
   }
%}
%class Lexer
%type Token
```
// digite aqui as linhas 1 a 7, e 11 a 16 do passo 1 item (b)
```
branco = [\n|\t|\r| ]+
%%
{palavrasChaves} { imprimir("PALAVRA-CHAVE  ----->
", yytext());  return PL; }
{branco}         { return BRANCO; }
{identificador}  { imprimir("IDENTIFICADOR  ----->
", yytext());  return ID; }
{const_literal}  { imprimir("CONSTANTE      ----->
", yytext());  return CTE; }
{inteiro}        { imprimir("NUMERO INTEIRO ----->
", yytext());  return INT; }
{decimal}        { imprimir("NUMERO DECIMAL ----->
", yytext());  return DEC; }
{operadoresAritmeticos}  { imprimir("OPERADOR
ARITM.----->  ", yytext());
                           return OPARITM; }
{operadoresComparacao}   { imprimir("OPERADOR
COMP. ----->  ", yytext());
                           return OPCOMP; }
{operadoresLogicos}      { imprimir("OPERADOR
LOGICOS --->  ", yytext());
                           return OPLOG; }
{simbolosEspeciais}      { imprimir("SIMBOLOS
ESPEC.----->  ", yytext());
                           return SIMB_ESPEC; }
.          { imprimir ("<<< CARACTER INVALIDO!!!
>>>    ",yytext()); return ERROR; }
<<EOF>>    { return null; }
```

Para concluir o seu trabalho, prepare um arquivo de teste com tokens válidos e alguns não válidos, e entregue o seu projeto com toda a documentação elaborada: um projeto completo com o .jar e o arquivo de teste.

Você, também, poderá baixar o projeto SQLbr, completo por meio do link https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/compiladores/u2/s2/US2S2_Projeto_SQLbr_6.zip ou por meio do QR Code.

Ao concluir seu analisador léxico, você utilizou todos os conhecimentos adquiridos até o momento neste curso. Aprofundou-se no processo de definição de gramática e exerceu suas habilidades para propor soluções mais simples, que irão facilitar o trabalho de seus clientes/usuários. Pôde, ainda, aplicar conhecimentos de programação anteriormente adquiridos, usar expressões regulares, e desenvolver o domínio de uma ferramenta para geração de analisadores léxicos, como o Jflex, fixando as regras utilizadas pela ferramenta pela prática, e, ainda, comparar essas regras com as da notação EBNF. Finalmente, encerrou o desafio com os testes de validação, que permitiram garantir que o trabalho desenvolvido alcançou os objetivos estabelecidos.

Avançando na prática

Reconhecendo padrões de entrada

Descrição da situação-problema

Um jornal de grande circulação disponibilizou aos escritórios de advocacia e contabilidade um portal para digitarem editais e enviarem para publicação. Porém, estão chegando textos demais à redação do jornal que são inviáveis para a publicação por estarem fora do padrão para circulação, o que vem trazendo prejuízos a ambas as partes, clientes e jornal.

O portal já está em funcionamento, mas a equipe de desenvolvimento não consegue resolver o problema, pois o texto não tem um formado fixo, parametrizado, necessitando apenas conter ao menos uma ocorrência dos formatos a seguir:

Responsável: nome
Valor do edital: *R$ 9999,90*
Cidade, *dia de nomedomes de ano*

As ocorrências desses padrões podem estar em qualquer posição do texto, devendo apenas aparecer uma vez.

Como você pode ajudar a equipe de desenvolvimento?

Resolução da situação-problema

A aplicação disponibiliza uma área de texto livre para digitação e é necessário identificar se os elementos do padrão definido pelo jornal estão presentes. Para atender essa necessidade, bastar criarmos uma especificação no JFlex para que a equipe de TI, a partir do reconhecimento dos tokens, valide se o formato do texto digitado está correto antes de aceitar o envio. Assim, para ajudá-los deve-se construir as expressões regulares de acordo com as regras do Jflex para que eles possam usá-las:

responsavel = [“Responsavel: ”][A-Z]+

valor = ”Valor do edital: R$”+[0-9]+[“,”][0-0]+

mes = ”janeiro” | ”fevereiro” | ”marco” | ”abril” | ”maio” | ”junho” | ”julho” | ”agosto” | ”setembro” | ”outubro” | ”novembro” | ”dezembro”

data = ”Cidade, ”[0-3][0-9][“de”[{mes}]”de”[0-9]+

O analisador léxico gerado pelo JFlex retorna os tipos de tokens das expressões regulares, que poderão ser analisados pelo aplicativo criado pela equipe de TI do jornal. Esse é um dos recursos dos scanners que auxiliam em várias atividades de buscas em textos longos, afinal os scanners não são usados apenas na construção de compiladores, e essa é uma das importâncias do domínio destas ferramentas.

Sem medo de errar

1. Os geradores de analisadores léxicos e sintáticos facilitam a construção de compiladores e auxiliam na busca de textos longos.
Na literatura técnica, esses geradores são conhecidos por termos em inglês, tanto para os geradores como para o analisador gerado.
Com relação à nomenclatura dos geradores de analisadores, é correto afirmar:
a) Os *Scanners* geram *Parsers*
b) Os *Scanners* geram imagens e *Parsers* analisadores
c) Os *Scanners* geram *Lexers*
d) Os *Lexers* geram *Scanners*
e) *Parsers* e *Scanners* são analisadores sintáticos e léxicos, respectivamente.

2. Dadas as produções no padrão EBNF:
<d> = 0|1|2|3|4|5|6|7|8|9|0
<CPF> = <d><d><d>"."<d><d><d>"."<d><d><d>"-"<d><d>
A expressão regular equivalente à produção <CPF>, de acordo com as regras do JFLex é:
a) cpf = ddd.ddd.ddd-dd
b) cpf = {d}{d}{d}.{d}{d}{d}.{d}{d}{d}-{d}{d}
c) cpf = ([0-9]".")**{3}**"-"dd
d) cpf = [0-9].[0-9].[0-9]-[0-9][0-9]
e) cpf = {d}{3}"."{d}{3}"."{d}{3}"-"{d}{2}

3. Analise as afirmações a seguir, referentes às regras de notação reconhecidas pelo JFlex e utilizadas para as expressões regulares.
a | b indica alternância entre a ou b, e ab é uma concatenação.
a* indica o fecho de Kleene, e a+ indica infinitas repetições.
a{n,m}, em que ***m*** indica o limite máximo de ocorrências de a e ***n*** o limite mínimo de ocorrências de a.
Assinale a alternativa correta.
a) Apenas a alternativa I está correta.
b) Apenas as alternativas I e II estão corretas.
c) Apenas as alternativas I e III estão corretas.
d) Todas as alternativas estão corretas.
e) Todas as alternativas estão erradas.

Seção 2.3

Análise sintática

Diálogo aberto

Prezado aluno, no desenvolvimento desta disciplina, você pôde experimentar a importância da especificação das expressões regulares para o reconhecimento dos elementos básicos de uma linguagem, ao construir um analisador léxico. Pôde também observar que para analisar se um programa foi escrito corretamente, não basta apenas reconhecer os elementos básicos, os *tokens*, mas também se a estrutura da frase está correta, isto é, se as palavras estão de acordo com as regras gramaticais.

Após ter atendido as solicitações iniciais da empresa que o procurou para construir uma linguagem alternativa à linguagem SQL padrão e desenvolvido o analisador léxico para a mesma, a empresa deseja que você conclua a gramática para a linguagem proposta, denominada SQLbr. Por que o trabalho desenvolvido apresentou uma ótima qualidade, eles desejam que a linguagem SQLbr tenha mais instruções do que a SQL padrão, além do comando SELECT inicialmente solicitado na primeira fase, tais como comandos de alterações de dados e alguns comandos da linguagem procedural do SQL. Assim, para atender às solicitações da empresa, o seu trabalho será concluir a especificação sintática da gramática da nova linguagem.

Pense: se uma etapa do trabalho já foi entregue, não seria prudente, antes de continuar, você se reunir com os executivos que irão usar a nova linguagem para saber se eles têm sugestões de melhorias e o que gostaram ou não da SQLbr?

E as novas implementações? Estão de acordo com os objetivos iniciais da linguagem solicitada, ou seja, simples e fáceis para não programadores?

Como será a futura troca de mensagens entre o analisador sintático e o usuário? Como o analisador sintático irá tratar os erros de sintaxes, quando ocorrerem? Afinal, um bom analisador deve

apontar mensagens claras para os erros contidos no programa fonte escritos pelos programadores, não é mesmo?

Para você responder a essas perguntas, iremos ajudá-lo, mostrando a função do analisador sintático, quais são as quatros estratégias que podem ser implementadas em um analisador sintático para o tratamento de erros e, por fim, como definir as regras da sintaxe de uma gramática completa.

Com certeza você deve estar ansioso por concluir mais esta etapa, especialmente ao perceber que um processo que inicialmente parecia complexo, com cada seção, ao compreender o passo a passo de um compilador, vai se tornando mais simples, não é mesmo?

Após concluir mais esta etapa, será o momento da entrega integral do trabalho solicitado. Para isso, ao final, disponibilize em uma mídia digital, que pode ser em um *pendrive* ou em uma área compartilhada na nuvem, os arquivos fonte do analisador léxico, da especificação completa da linguagem na notação EBNF (léxico e sintático) e do arquivo executável (.jar).

Não pode faltar

Nesta seção, iremos iniciar o estudo da análise sintática, a qual está no centro da fase de análise de um compilador, pois o analisador sintático interage com os analisadores léxico e semântico, acionando-os quando necessário. Assim, o analisador sintático, ao verificar uma "frase", aciona o analisador léxico, que retorna o tipo de *token*, ou uma lista de *tokens*, e, a partir do fluxo recebido, inicia a análise da produção.

Assimile

"Uma gramática de *Chomsky, Gramática Irrestrita ou simplesmente Gramática*, é uma quadrupla ordenada: G = (V, T, P, S)" (MENEZES, 2005, p.35).

Vale lembrar que **S** é a produção inicial da gramática.

P é o conjunto das regras de produções, ou simplesmente produções, que definem as condições de geração das palavras da linguagem.

E palavras, aqui, significam um conjunto de palavras com uma determinada estrutura (frase).

T é o alfabeto da linguagem e **V** é o conjunto dos símbolos não terminais, em outras palavras "***o nome das produções***".

Logo, as produções definem as regras da gramática.

Segundo Aho (2007), cada linguagem de programação (LP) possui regras que definem a sua estrutura sintática e, ainda, a sintaxe das construções das LPs pode ser descrita por linguagens livres de contexto (LLC) ou pela notação BNF (Backus-Naur Forms), como visto na Seção 1.2.

O analisador sintático, nesse contexto, tem como função verificar se o programa fonte está escrito de acordo com as regras definidas pela sua gramática, ou seja, as estruturas sintáticas. Para alcançar este objetivo, ele solicita ao analisador léxico a análise do fluxo dos *tokens* de entrada e, após receber o retorno do léxico, verifica se o fluxo, o conjunto de *tokens*, pode ser gerado pela gramática da linguagem, isto é, se a estrutura do comando está em conformidade a regra definida.

Assim, o agrupamento de *tokens* é a uma "frase gramatical" e o analisador sintático irá gerar uma representação hierárquica, **a saída**, que será uma árvore gramatical. As árvores gramaticais e derivações serão objeto de estudo mais detalhado na Seção 3.1.

Na prática, algumas tarefas a mais são realizadas pelo analisador sintático, além da geração da representação da árvore gramatical a partir do fluxo de *tokens* fornecido pelo léxico. Por exemplo, o analisador sintático pode realizar a verificação de tipos e outras análises semânticas, pois ele está no meio do processo, e a análise completa de um comando envolve as três etapas: léxica, sintática e semântica. Logo, o próprio analisador sintático aciona a tabela de símbolos para as consultas semânticas.

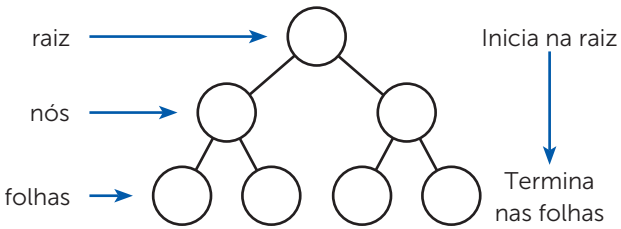
Os métodos comumente utilizados para a análise sintática são os descendentes (*top-down*) e os ascendentes (*bottom-up*). O método descendente faz a análise de cima para baixo, isto é, da raiz para as folhas. Já o método ascendente analisa de baixo para cima, ou seja, das folhas para a raiz. Esses métodos serão estudados na Seção 3.1.

Reflita

Se a análise descendente é de cima para baixo, por que dizemos da raiz para as folhas?

Quando falamos de árvore em ciências da computação, estamos normalmente nos referindo a uma estrutura de dados, e, segundo Forbellone e Eberspächer (2005), essa estrutura, a árvore, consiste em uma lista que começa pelo nó inicial, chamado de raiz, e possui um ou mais sucessores, denominamos de nós. Os nós que não possuem sucessores são chamados de folhas.

Figura 2.25 | Estrutura de uma árvore



Fonte: elaborada pela autora.

As estruturas de listas e árvores serão amplamente aplicadas na implementação dos algoritmos para análise sintática e na tabela de símbolos, utilizados na construção dos compiladores.

Para saber mais consulte o link a seguir, ou mesmo seus materiais sobre estrutura de dados:

UFES. **Estrutura de Dados** – Aula 15: Árvores. [S.l.], 17 maio 2017. Disponível em:<https://inf.ufes.br/~pdcosta/ensino/2012-1-estruturas-de-dados/slides/Aula15%20%28arvores%29.pdf>. Acesso em: 20 maio 2018.

Quem nunca teve um erro de compilação, que atire a primeira pedra. Na sua experiência como programador, a probabilidade de ter ocorrido algum erro durante a compilação de algum programa é provavelmente próxima a 100%. Assim, se erros de compilação ocorreram nas suas atividades de programação, então você recebeu mensagens de erros emitidas pelo compilador, e obviamente as leu, não é mesmo? Você as compreendeu? Elas eram claras e ajudaram a corrigir seus programas?

Frequentemente, os programadores escrevem programas incorretamente, assim, se o planejamento do tratamento de erros for pensado pelo projetista do compilador desde o início do projeto, pode-se melhorar o desempenho do compilador e a resposta aos erros. Isto é, mensagens melhores e uma recuperação mais eficiente.

Segundo Aho (2007) há três metas simples que o tratador de erros em um analisador sintático deve buscar: 1) relatar a presença de erros com clareza e precisão; 2) ser capaz de se recuperar do erro a fim de conseguir identificar os próximos, se houver; 3) não ser lento, no caso de programas corretos.

A segunda meta, ser capaz de se recuperar após a detecção de um erro e continuar a análise do programa, é um dos pontos mais difíceis para o projeto do compilador. Afinal, o que se deseja é identificar todos os erros a cada processo de compilação. Porém, isso não é possível, e assim buscaram-se estratégias para que identificar o máximo de erros a cada compilação. Há muitas estratégias, e nenhuma totalmente perfeita. Vamos conhecer algumas que são bastante utilizadas:

a) **Modalidade do desespero** – essa estratégia talvez seja a mais simples para ser implementada. Após encontrar um erro, o compilador descarta *token* a *token* até encontrar um *token* de sincronização, ponto em que ele reinicia a análise.

   A vantagem dessa estratégia é a simplicidade para a implementação, desde que, na especificação da gramática, haja *tokens* delimitadores, os chamados pontos de sincronização. Essa estratégia também é conhecida como recuperação de erro **sem** correção.

b) **Nível de frase** – também conhecida como recuperação de erro **com** correção, pois quando o analisador sintático se deparar com um erro, ele tenta realizar uma correção para poder continuar a análise do restante do programa.

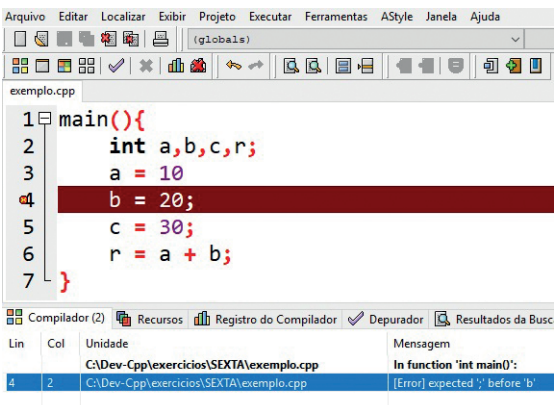
   Para utilizar essa estratégia, o projetista do compilador precisa construir produções na gramática que considerem os pontos passíveis de intervenção para correção, e também precisa tomar cuidado para que o processo de compilação não entre em um *loop* infinito. Porém, há a possibilidade do erro ter ocorrido antes da detecção do erro e, nesse caso, o erro não é identificado com a acuracidade esperada.

Além das estratégias de recuperação de erro apresentadas, podemos citar mais duas: c) produções de erros e d) correção global. Essas não são muito utilizadas, pois a estratégia de recuperação por produções de erro exige que o projetista incorpore na gramática da linguagem produções (regras) para tratamento de erros que os programadores comumente cometem, o que aumenta significativamente a gramática, o tempo de compilação e ainda poderão ocorrer erros não previstos. Já com relação à estratégia de correção global, que seria o método perfeito, pois incorpora as duas estratégias apresentadas (modalidade do desespero e nível de frase), nos deparamos com um algoritmo que é muito custoso, isto é, o tempo de processamento é lento, o que torna o processo de compilação vagaroso.

Exemplificando

Qual a estratégia de recuperação de erro utilizada pelo compilador C, no exemplo apresentado na Figura 2.26?

Figura 2.26 | Programa exemplo em C



Fonte: captura de tela do Dev-C++ 5.1, elaborada pela autora.

Nesse caso, foi utilizada a estratégia de recuperação sem correção (modalidade do desespero). Como pode ser visto na Figura 2.26, o erro ocorreu na linha 3, com a falta do ponto e vírgula, mas o compilador apontou o erro na linha 4, somente quando detectou o símbolo

terminal (;). Nesse caso, a mensagem de erro é clara:

[Error] *expected ';' before 'b'* , ou seja era esperado ';' antes de 'b'.

Contudo, se o programador tivesse escrito na linha 4 apenas '= 20;', a mensagem de erro seria [Error] *value required as left operand of assignment* (valor é exigido como operando à esquerda da atribuição), e nenhuma referência ao erro da linha 3 seria feita, pois as linhas 3 e 4 são vistas pelo compilador como uma única "frase". Portanto, a recuperação de erro sem correção é eficiente, rápida, mas não é perfeita, e, infelizmente, ainda não foi encontrada uma estratégia definitiva para isto.

Vamos, agora, avançar na gramática e compreender a diferença entre a gramática somente léxica e quando a produção está definindo uma estrutura sintática. Assim, vejamos o exemplo: **a = 10;**. Para construir a gramática para esse exemplo, considerando tratar-se da linguagem C, precisamos identificar uma variável, um número, o símbolo '=' e ';', assim, a gramática léxica será:

```
<variavel> ::= ( '_' | <letra> ) { '_' | <letra>
| <digito> }
<numero> ::= <digito> { <digito> [ '.' {<digito>}
] }
<igual> ::= '='
<ptv> ::= ';'
```

As definições de `<variavel>` e `<numero>` são expressões regulares e definem apenas tokens. Já `<igual>`, `<ptv>`, `<letra>` e `<digito>` reconhecem símbolos terminais.

Caso fosse escrito '10 a = ;' , o analisador léxico reconheceria os *tokens*, logo não haveria erro léxico, mas haveria erro sintático, pois a estrutura do comando de atribuição não está correta, já que deve começar com uma variável, seguida do símbolo '=' e de uma expressão. Assim, a produção para reconhecer o comando de atribuição deve ser:

```
<comm_atrib> ::= <variavel> <igual> <expressao>
<ptv>
<expressao> ::= <variavel> | <numero>
```

Veja que a produção <comm_atrib> é uma sequência de *tokens* em uma determinada ordem, e cada *token* foi anteriormente identificado pelo analisador léxico, razão porque afirmamos que o analisador sintático recebe um fluxo de *tokens* e analisa se ele está na ordem correta, conforme definido na produção <comm_atrib>.

Portanto a sintaxe da gramática determina a estrutura que se espera de cada comando, por exemplo:

Em C, o comando de tomada de decisão tem a seguinte estrutura:

```
if ( condição )  { lista de comandos }  [ else  {
lista de comandos }  ]
```

Já em Visual Basic, o comando de tomada de decisão tem a seguinte estrutura:

```
if condição then lista de comandos [ else lista de
comandos ] end if  |
if condição then lista de comandos [ elseif condição
then lista de comandos [ else lista de comandos ]] end
if
```

A gramática para os comandos de tomada de decisão deve reconhecer qual é o *token* inicial de cada estrutura, seja para o Visual Basic, seja para a linguagem C, e, a partir do estado inicial, analisar se a estrutura sintática segue as regras definidas na produção, espelhando, assim, a sintática da gramática. Veja como fica a gramática sintática para o comando de tomada de decisão no caso da linguagem C:

```
1. <comm_if> ::= 'if' '('<condicao>')'
'{'<comandos>'}' [ 'else' '{'<comandos>'}' ]
2. <comandos> ::= { <comm_if> | <comm1> | <comm2>
}
3. <termo> ::= ( <variavel> | <numero> )
4. <comparacao> ::= <termo> <operadorComparacao>
<termo>
5. <condicao> ::= <termo> | <comparacao> {[
<operadorLogico> <condicao> ]}
```

A produção (1) <comm_if>, é a regra do comando *if* em C, e as produções de 2, 4, e 5 são símbolos não terminais, ou seja, outras produções que através do processo de derivação de cada produção

nos levam a uma produção terminal, por exemplo: produção 5 nos leva a produção 3 ou 4; a produção 4, por sua vez, nos leva a 3, que contém apenas produções léxicas, e as produções léxicas nos levam a símbolos terminais.

Vale lembrar que as produções `<variavel>`, `<numero>`, `<operadorComparacao>`, `<operadorLogico>` são produções do léxico e foram objeto de estudo na Seção 2.1.

Observe que a produção `<comandos>`, além da regra para o comando ***if*** (`<comm_if>`), apresenta outros dois comandos, `<comm1>` e `<comm2>`. Eles foram colocados para indicar que, nesta produção, a `<comandos>`, deve-se especificar toda a lista de comandos aceitos. Como neste caso estamos tratando apenas do comando ***if***, eles não foram especificados.

Pesquise mais

O estudo de compiladores não apenas envolve a sua construção, mas permite ao programador ampliar sua visão sobre a estrutura das linguagens e compreender melhor seus manuais.

Pesquise mais sobre exemplos de estruturas sintáticas de comandos nos links indicados a seguir, e veja como ficou mais fácil de entendê-las após ter estudado análise sintática e o padrão EBNF.

CLÁUSULA SELECT (Transact-SQL). In: Microsoft Docs. [S.l.], 9 ago. 2017. Disponível em: <https://docs.microsoft.com/pt-br/sql/t-sql/queries/select-clause-transact-sql?view=sql-server-2017>. Acesso em: 27 maio 2018.

MICROSOFT. **Gramática de estrutura de frase –** Visual Studio. [S.l.; s.d.]. In: Microsoft Developer Network. Disponível em: <https://msdn.microsoft.com/pt-br/library/sxbyfc2c.aspx>. Acesso em: 27 maio 2018.

RIBEIRO, S. F. **Especificação sintática para a linguagem Pascal**. Petrolina, [s.d.]. Faculdade de Ciências Aplicadas e Sociais de Petrolina – FACAPE. Disponível em: <http://files.ccfacape.webnode.com/200000109-67fce68f84/projeto2_Pascal.pdf>. Acesso em: 27 maio 2018.

CASTILHO, M. et al. **Guia rápido de referencia da linguagem Pascal -** Versão *Free Pascal*. [S.l.], dez. 2009. v. 0.2. p. 12. Departamento de Informática. Universidade Federal do Paraná. Disponível em: <http://www.inf.ufpr.br/cursos/ci055/pascal.pdf>. Acesso em: 27 maio 2018.

Nesta seção, conhecemos a função do analisador sintático, e a importância do projetista do compilador planejar o tratamento de erros, afinal uma das metas do analisador é relatar a presença de erros de sintática de forma clara e precisa. Avançamos na compreensão da diferença entre as produções do léxico e do sintático, e no reconhecimento da sintaxe e da definição das produções da análise sintática. Agora é o momento de aprofundar os conhecimentos sobre análise sintática para integrar o analisador léxico e o sintático, além de saber como trabalhar com situações críticas, como ambiguidade e métodos para implementação dos analisadores sintáticos.

Como você já percebeu, a cada seção avançamos no processo de desvendar as funções e o processo para implementar um compilador, e tudo vai ficando mais simples e fácil de entender. Vamos continuar?

Sem medo de errar

Você se saiu muito bem na primeira etapa de desenvolvimento de uma linguagem mais simples para o padrão SQL para a empresa que o procurou, então eles decidiram solicitar que o projeto inicial fosse ampliando.

Estrategicamente, você percebeu que, em função das alterações solicitadas, o objetivo inicial proposto, ou seja, uma linguagem simples e de fácil uso para não programadores está tendo seu escopo alterado ao incluir-se comandos procedurais do SQL, assim, você solicitou: a) uma avaliação dos executivos que irão usar a linguagem e b) um parecer do departamento de TI.

Qual é a melhor maneira de fazer a avaliação com os usuários do léxico que você já entregou? um questionário, que pode ser interativo ou papel, ou ainda outra forma? O que você sugere?

E com a equipe técnica? Por que o uso de comandos procedurais não seria recomendado? Como eles veem a questão de segurança/integridade dos dados ao liberar esses comandos a usuários leigos?

Após ter feito seus levantamentos, você estará em condições de continuar e apresentar seu trabalho como solicitado inicialmente, apenas com a inclusão da cláusula de UPDATE para alteração de

dados, e, como o comando procedural envolve programação, a equipe de TI concordou com as razões dos seus questionamentos. Portanto, mais uma vez você foi capaz de mostrar sua habilidade em trabalhar com o projeto e direcionar corretamente o foco para o objetivo: uma linguagem simples para não programadores.

Já com relação aos usuários, valeu a pena consultá-los. Pequenas mudanças foram sugeridas, que não alteram o conceito do projeto inicial, mas melhoram a compreensão e facilidade de uso dos comandos, o que permitirá que você não apenas incorpore essas melhorias já nesta definição da sintaxe da gramática, mas também projetar as respostas que o analisador sintático dará para as mensagens de erros do usuário.

Feitas todas essas considerações, você já pode apresentar a sintaxe completa da nova linguagem, mas, que tal também mostrar conjuntamente a sintaxe do SQL padrão considerado com a sintaxe da SQLbr. Não seria uma boa ideia? Considerando isso, você elaborou a apresentação da sintaxe das duas linguagens:

Sintaxe do SQL padrão – limitado	Sintaxe do SQLbr - proposta
SELECT *lista de campos* **FROM** *lista de tabelas* [**WHERE** *condição*] [**ORDER BY** *lista de campos*]	**MOSTRAR** *lista de campos* (**DAS** | **DA**) *lista de tabelas* [(**CUJO** | **CUJA**) *condição*] [**ORDENAR_POR** {*campo* [**,**(**CRESCENTE** | **DECRESCENTE**)]}
USE *nome da tabela*	**ABRIR_TABELA** *nome da tabela*
UPDATE *nome da tabela* **SET** *lista de atribuições* [**WHERE** *condição*]	**ALTERAR_OS_DADOS** *lista de atribuições* **DA_TABELA** *nome da tabela* (**CUJO** | **CUJA**) *condiçao*

Após apresentar a sintaxe da SQLbr, você pode elaborar a gramática do comando, e já pensando na estratégia para recuperação de erro, que tal adotar a mais simples e a que também é utilizada pelo JFlex, a modalidade do desespero?

Se você utilizar a modalidade do desespero, será importante utilizar delimitadores para os comandos. Que tal o ponto e vírgula? Desta forma, a gramatica de acordo com a sintaxe proposta será:

Gramática do SQLbr

1. <mostrar> ::= <plMostrar> <campos> <plOrigem><tabelas> [clausulaOrdenacao] ';'
2. <campos> ::= [<nomeDaTabela> '.'] <nomeDocampo> { [',' [<nomeDaTabela> '.'] <nomeDocampo>}
3. <tabelas> ::= <nomeDaTabela> {[','<nomeDaTabela> }
4. <clausulaCondicao> ::= <plCujo> <condição>
5. <expressão> ::= <fator> <opComparacao> <fator> | <fator>
6. <fator> ::= <nomeDocampo> | <numero> | <const_literal>
7. <clausulaOrdenacao> ::= <plOrdem> <camposOrd>
8. <camposOrd> ::= [<nomeDaTabela> '.']<nomeDocampo> [<tipoOrdem>] { , [<nomeDaTabela> '.']<nomeDocampo> [<tipoOrdem>] }

9. <plMostrar> ::= **'MOSTRAR'**
10. <plOrigem> ::= **'DA'** | **'DAS'**
11. <plCujo> ::= **'CUJO'** | **'CUJA'**
12. <plOrdem> ::= **'ORDENAR_POR'**
13. <opComparacao> ::= **'<'** | **'>'** | **'<>'** | **'<='** | **'>='**

14. <nomeDaTabela> ::= <letra>{ [<letra> | <dígitos>] }
15. <nomeDocampo> ::= <letra>{ [<letra> | <dígitos>] }
16. <numero> ::= <digito> {<dígitos>] } { ['.' {<dígitos> }] }
17. <nomeVariavel> ::= <letra>{ [<letra> | <dígitos>] }
18. <const_literal> ::= ' " ' <qqsimbolo> ' " '

Nas linhas 1 a 8 temos a definição da sintática das regras da gramática, e nas linhas 9 a 18, as definições léxicas. Veja que, ao elaborar a sintática da gramática, as produções são fluxos de *tokens*, como estudamos nesta seção, e o ponto e vírgula foi adicionado, pois cabe ao projetista prever quais serão suas necessidades para posteriormente implementar os algoritmos, sejam ascendentes ou descentes, para que o compilador possa se recuperar após encontrar um erro e continuar o processo de análise até o final do programa.

Falta pouco para concluir a gramatica solicitada, não é mesmo? Resta somente especificar os comandos ABRIR_TABELA e ALTERAR_OS_DADOS, os quais são bem mais simples, então aproveite para treinar mais um pouco especificando-os.

Sua primeira linguagem foi concluída e você pôde, neste desafio, além de aplicar os conceitos de análise sintática, verificar como é importante desenvolver todo o projeto da gramática antes de iniciar o desenvolvimento, pois, na análise sintática surgem novas regras que exigem rever o léxico, e assim você irá observar na próxima etapa a análise semântica, que não foi considerada até o momento. Por outro lado, a grande vantagem da construção do compilador estar divida em fases é que seu aprendizado pode ser feito passo a passo. Você está gostando de dominar esta tecnologia?

Como diz o sábio chinês Lao Tse, "um caminho de mil quilômetros começa com o primeiro passo", e você, além de estar no caminho certo para dominar o processo de construção de compiladores, já avançou mais da metade dele. Parabéns!

Para encerrar o trabalho para o qual foi contratado, é o momento da entrega do projeto completo: o analisador léxico dos arquivos fontes, o .jar e a especificação no JFLEX, além da definição completa da gramática, com o léxico e o sintático.

Avançando na prática

Resolvendo problemas de verificação de entradas

Descrição da situação-problema

Uma distribuidora de produtos firmou um contrato com diversos sites de vendas de produtos customizados para fornecer

automaticamente os materiais que tais sites irão utilizar, por meio do envio do arquivo. No processo anterior, em que recebiam um arquivo parametrizado, tudo ia muito bem, mas com a nova tecnologia do uso de *bots* para os usuários identificarem as customizações que desejam no produto, o time do projeto proposto se deparou com uma mudança de paradigma na entrada de dados, o qual não estão conseguindo resolver, pois não existirá a parametrização campo a campo, mas sim frases como respostas dos consumidores.

Assim, a empresa de desenvolvimento pensou em contratar um consultor para ajudá-los nesse problema, pois, apesar de algumas soluções terem sido propostas, a que melhor se comportou teve, ainda, muita reclamação dos próprios usuários dos sites de venda, por que informa um erro de cada vez e os clientes precisam submeter os pedidos repetidas vezes, e o que esperam ao clicar em enviar é receber uma informação com todos os erros.

E você o quê acha? Há solução para isto?

Resolução da situação-problema

Sim, claro que há, e ela envolve a integração de várias áreas da ciência da computação. Primeiramente, podemos utilizar uma plataforma que integre linguagens funcionais ou base de conhecimento, como a Prolog, ao *front-end* atualmente utilizado, afinal os sistemas atuais estão evoluindo e os paradigmas estão mudando, logo, conhecer os diversos paradigmas das linguagens ajudará na melhor escolha para solucionar cada problema, como estudado na Seção 1.1. Em segundo lugar, como o usuário irá escrever uma frase, será importante adotar uma gramática flexível, ou seja, com diversas palavras-chave, para reconhecer cada solicitação. Para isso, basta que o *bot* faça pergunta a pergunta e para cada pergunta exista uma produção associada aos comandos (respostas) aceitos. Portanto, a sua proposta nada mais é que uma linguagem, a identificação dos *tokens* (o léxico) e as frases (o sintático). Lembrando que o léxico foi estudado nas Seções 2.1 e 2.2, e o sintático nesta seção.

Desta forma, como consultor neste assunto, sua proposta será:

a) Propor uma sintaxe para cada frase, no caso, uma resposta para cada pergunta.

b) Escrever a especificação EBNF para a linguagem.

c) Lembrar que as palavras chaves para cada site de venda serão diferentes e que o projetista do site deverá levar isso em consideração na especificação da linguagem e do site.

d) As especificações das estruturas de cada frase da linguagem devem incluir um delimitador, podendo implementar no analisador sintático um tratador de erros. Portanto, caso encontre um erro de escrita, ignore os *tokens* após o erro até encontrar o próximo delimitador e, a partir desse ponto, continuar a análise do texto.

Nessa solução, é possível perceber que, como consultor em um assunto, além de ter conhecimento técnico, é necessário o desenvolvimento de habilidades como as apresentadas aqui, nas quais você integrou várias competências técnicas, tais como paradigmas de linguagens de computação, linguagens formais, especificação de linguagens e estratégias de tratamentos de erros, reunindo essas competências e apresentando uma solução aplicando todos estes conhecimentos.

Para concluir, valerá enfatizar, como foi visto nesta unidade, que não há solução perfeita para a recuperação de erros, e a que foi apresentada é a mais fácil de ser implementada, pois evita loop infinito e consegue o melhor equilibro em relação à facilidade para implementação e rapidez de resposta, com a maior identificação de erros em cada processo de compilação. Outro ponto importante estudado nesta seção é a sintaxe dos comandos, que dá uma visão geral e passível de ser compreendida e ajuda aos usuários do site a escreverem suas solicitações, já a especificação do analisador sintático é voltada exclusivamente ao desenvolvedor.

Faça valer a pena

1. Ao especificamos as regras de uma linguagem, construímos sua gramática. Contudo, há regras de produções que são apenas descritas por símbolos terminais, outras descritas por expressões regulares e há, ainda, as que são expressas por um fluxo de ***tokens***.
Sabemos que as produções léxicas geram apenas símbolos terminais ou são expressas por expressões regulares, e que as sintáticas são expressas por fluxo de ***tokens***.

Assinale a alternativa que apresenta apenas produções sintáticas:
a) <digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
b) <variável> ::= <letra> { <letra> | <digito> }
c) <com_if> ::= 'if' <condição> 'then' <comando>
d) <palavraReservada> ::= 'if' | 'the' | 'begin' | 'end' | 'else'
e) <<numero> ::= <digito> {<digito>} ['.' {<digito>}]

2. O agrupamento de ***tokens*** é uma "frase gramatical", e o analisador sintático irá gerar uma representação hierárquica, a saída, que será a árvore gramatical. Para a análise da sintática, há dois métodos muito utilizados. Quais são eles?
Assinale a alternativa correta:
a) descendente (*bottom-up*) e recursivamente ascendente (*top-down*).
b) ascendente (*top-down*) e recursivamente descendente (*bottom-up*).
c) descendente (*top-down*) e ascendente (*bottom-up*).
d) recursivamente ascendente (top-down) e recursivamente descendente (*bottom-up*).
e) arvore binaria da raiz para as folhas e o inverso das folhas para a raiz.

3. O analisador sintático, entre suas funções, deve sinalizar os erros de sintática e de grafia que o programador comete no código fonte. Desta forma, o tratador de erros em um analisador sintático deve buscar relatar a presença de erros com clareza e precisão, ser capaz de se recuperar do erro a fim de conseguir identificar os próximos, se houver, e não ser lento no caso de programas corretos.
A recuperação do erro é fundamental para a análise do código-fonte até o final, após a detecção de um erro.
Assinale a alternativa que apresenta estratégias de recuperação de erros:
a) Modalidade do desespero, nível de frase, produções de erros e correção global.
b) Recuperação de erro com correção, correção da frase, evitar erros comuns e não produzir erros.
c) Modalidade de recuperação, nível de frase, recuperação global e recuperação de erro sem correção.
d) Recuperação de erro com correção, recuperação de erro sem correção e evitar erros comuns.
e) Modalidade do desespero, nível de frase, correções de erros e correção global

Referências

AHO, A. V.; SETHI, R.; ULLMAN, J.D. **Compiladores**: Princípios, técnicas e ferramentas. 2ª ed. São Paulo: Pearson, 2007.

CHRISTENSSON, P. **Token Definition**. 9 abr. 2009. Disponível em: <https://techterms.com/definition/token>. Acesso em: 18 maio 2018.

DELAMARO, M. E. **Como Construir um Compilador -** Utilizando Ferramentas Java. Editora Novatec, 2004.

FORBELLONE, A. L. V.; EBERSPÄCHER, H. F. **Lógica de Programação:** a construção de algoritmos e estruturas de dados. 3ª ed. São Paulo: Pearson Prentice Hall, 2005.

KLEIN, G.; ROWE, S.; DÉCAMPS R. **JFlex User's Manual** - The Fast Lexical Analyser Generator. 2015. v. 1.6.1. Disponível em: <http://jflex.de/manual.html>. Acesso em: 10 maio 2018.

MENEZES, P. B. **Linguagens Formais e Autômatos**. 5ª. ed. Porto Alegre: Sagra Luzzatto, 2005.

WELSH, J.; MCKEAG, R. M. **Structured system programming**. 1ª. Ed. Englewood Cliffs: Prentice Hall, 1980.

Tabela de símbolos, análise semântica e tradução dirigida por sintaxe

Convite ao estudo

Nesta unidade, iremos conhecer e nos aprofundar na análise sintática, e avançaremos para a última fase de análise, a análise semântica. Concluiremos a unidade com o estudo da tabela de símbolos, que está presente em todas as fases do compilador. Você saiu se muito bem nos estudos sobre compiladores e aproveitou para divulgar suas novas habilidades e competências, atualizando seu perfil no canal Conecta e no LinkedIn, além de divulgar no seu blog o analisador léxico construído na Unidade 2. Como resultado, recebeu um convite para participar de um processo seletivo em um instituto de pesquisas que possui um laboratório dedicado aos estudos sobre linguagens de programação. Eles irão contratar três profissionais para atuarem nas pesquisas e, futuramente, atuarem em um projeto de *Domain-Specific Language* (DSL). Você ficou muito feliz com o convite e aceitou prontamente, afinal, os primeiros frutos dos seus estudos sobre compiladores começavam a ser colhidos.

O responsável pelo laboratório informou que o processo de seleção será realizado em duas fases: a primeira será sobre o projeto conceitual da análise sintática e semântica, e apenas seis candidatos avançarão para a segunda fase, na qual os candidatos deverão demonstrar conhecimentos para trabalharem com as estruturas de dados que envolvem a implementação da tabela de símbolos na construção dos compiladores.

Na primeira fase do processo de seleção, você será convidado a elaborar uma apresentação que demonstre domínio da análise

sintática da gramática que será apresentada pela comissão julgadora, bem como do tratamento dos erros de semântica. A forma de apresentação será livre para que a comissão julgadora possa, além de analisar as competências, também avaliar a criatividade, clareza e inovação de cada participante. Na segunda fase, os candidatos selecionados, serão convidados a elaborarem a proposta para a implementação da tabela de símbolos. Vencerá a melhor proposta, e, neste caso, o candidato mais preparado será aquele com maior domínio na aplicação das técnicas mais eficientes. Pronto para fazer valer a pena a oportunidade alcançada?

Com sua motivação, empenho e os temas que serão estudados nesta unidade, você estará preparado para vencer essa seleção. Assim, na Seção 3.1 iremos estudar como fazer a análise sintática, os métodos existentes e quando usar um ou outro, e aproveitaremos para conhecer mais sobre o método *top-down*, as construções das árvores gramaticais e as derivações. Ainda na Seção 3.1, iremos analisar um problema comum das gramáticas livres de contexto, a ambiguidade, a qual uma boa gramática deve evitar. Já na Seção 3.2, vamos estudar a última fase da análise, a semântica, e a técnica de tradução dirigida pela sintaxe para implementá-la. Concluiremos a unidade, na Seção 3.3, estudando a importância da tabela de símbolos para o processo de construção de um compilador e as estruturas de dados envolvidas na implementação da mesma.

Com os estudos que serão desenvolvidos nesta unidade, você ficará confiante e pronto para vencer a seleção. Vamos iniciar a aprendizagem?

Seção 3.1

Tabela de símbolos, análise semântica e tradução dirigida por sintaxe

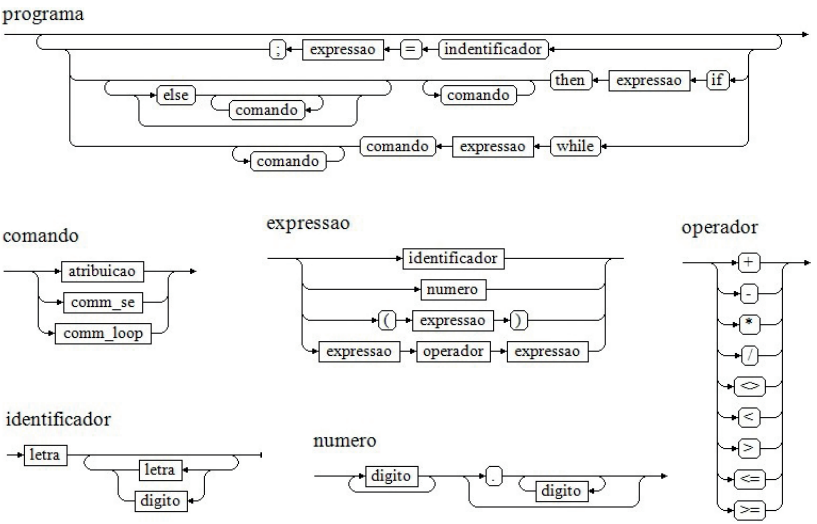
Diálogo aberto

Caro aluno, o laboratório de pesquisa do qual você participará de processo seletivo gostou muito do seu perfil e achou muito bons os trabalhos publicados em seu blog. Por isso, o incentivou a continuar a publicação dos artigos e, agora, espera que você seja tão criativo e inovador como já demostrou com as suas iniciativas no blog e nos canais corporativos nos quais apresentou seu currículo.

Ao ter aceitado o convite para participar do processo seletivo, você foi convocado para uma entrevista com o responsável. Durante a entrevista, foi entregue a sintaxe da linguagem para a qual você deverá elaborar a apresentação à comissão julgadora no prazo de 30 dias. A Figura 3.1 corresponde a sintaxe da linguagem.

O entrevistador explicou que você deverá analisar o diagrama de sintaxe e: a) avaliar o diagrama entregue e elaborar a definição EBNF correspondente; b) fazer a representação da análise sintática da definição EBNF criada; c) apresentar os resultados e justificativa dos métodos utilizados. Ele informou, ainda, que a comissão julgadora irá avaliar, além dos itens (a) e (b), a clareza da apresentação, a desenvoltura e o domínio do assunto. Com relação à forma da apresentação, você terá entre 30 (mínimo) e 45 minutos (máximo) para expor os resultados à comissão julgadora.

Figura 3.1 | Diagrama de Sintaxe proposto no processo seletivo



Fonte: elaborada pela autora.

Parece que os organizadores do processo seletivo esperam um senso crítico sobre a sintaxe da linguagem entregue para avaliação. O que você achou do diagrama da Figura 3.1? Está correto? Pode ser melhorado? Puxando pela memória, você se recorda que a saída da análise sintática gera uma representação hierárquica, a árvore gramatical? E, como poderia ser feita esta representação?

Você se mostrou ágil ao atualizar seu currículo e expor o que aprendeu, utilizando muito bem a rede para isso, com o uso do blog. E agora, qual será a tecnologia inovadora que irá utilizar para expor seu trabalho à comissão julgadora?

Para que você possa atender às solicitações propostas e se sair muito bem na sua apresentação, iremos mostrar a você os métodos pelos quais podem ser estruturadas (montadas) as árvores gramaticais e as derivações, e iremos estudar como analisar uma gramática para verificar se a mesma apresenta alguma ambiguidade e como corrigi-la, caso seja ambígua. Assim, você estará preparado para responder os questionamentos levantados e elaborar uma apresentação exemplar. Pronto para ir em busca de uma locação no mercado?

Não pode faltar

Na Seção 2.3, estudamos as funções do analisador sintático, que verifica se a sequência de *tokens* recebidos está na ordem correta, mas alguns comandos possuem uma estrutura recursiva, e, nesses casos, a gramática para gerar essa estrutura não poderá ser regular, logo não será representada por uma expressão regular, mas sim por uma gramática livre de contexto (GLC), presente nas estruturas sintáticas.

Assimile

É importante lembrar:

A **sintaxe** de uma linguagem define as estruturas básicas da linguagem.

As **estruturas básicas**, por sua vez, são compostas pelas regras léxicas e sintáticas, que são definidas pela gramática.

As **regras léxicas** são definidas pelas gramáticas regulares (GR).

As **regras sintáticas** são definidas pelas gramáticas livres de contexto (GLC).

Segundo a hierarquia de Chomsky, $\mathbf{LR} \subset \mathbf{LLC}$, isto significa que toda linguagem regular (LR) é livre de contexto (LLC), mas nem toda LLC é LR. Logo, GR define LR, e GLC define LLC.

É comum usar simplesmente o termo gramática ao se referir às regras que definem uma linguagem de programação. Na gramática, símbolos terminais pertencem ao alfabeto da linguagem e símbolos não-terminais representam produções (regras). E é bom não esquecer:

- Uma linguagem é regular se existe uma expressão regular que a representa.

Já as linguagens exclusivamente livres de contexto, segundo Aho (2007), são mais úteis na descrição de **estruturas "aninhadas"**, ou seja, quando há recursividade e não podem ser representadas por expressões regulares.

As estruturas recursivas estão presentes nos comandos de decisão e de repetição, encontrados na maioria das linguagens. Vejamos o caso do comando de tomada de decisão:

`if` *`<condicional>`* **`then`** *`<bloco de comandos>`* **`else`** *`<bloco de comando>`*

Esse comando não pode ser representado por uma ER (expressão regular), mas sim por uma GLC. Isso pode ser observado claramente se o `<bloco de comandos>` for substituído por uma série de comandos ***if***, assim representamos:

comm_if **if** → *condicional* **then** *comm_if* **else** *comm_if*

Veja que, nesse caso, é muito fácil identificar a recursividade, pois a produção à esquerda aparece também à direita, gerando um processo recursivo.

Reflita

Analise o comando em C a seguir:

```
if (a>100) if (b==5) print ("1"); else printf("2");
```

Sintaticamente, está correto para a linguagem C? O ***else*** está associado ao primeiro ou ao segundo ***if***? Como ficaria a produção para este comando?

Verificar se o programa foi escrito de forma sintaticamente correta é construir um analisador léxico, segundo as regras léxicas, que lê o código fonte e, se forem gerados apenas *tokens* válidos, passa à etapa seguinte. Na etapa seguinte, construímos um analisador sintático que analisa se o código fonte está de acordo com as regras sintáticas e semântica, concluindo a segunda etapa do processo de análise. A pergunta é: como o analisador faz isso? Por meio da derivação da árvore sintática. Logo, construir um analisador sintático será desenvolver um algoritmo que nos permita construir uma árvore de derivação para qualquer sentença (estrutura de comando) que pertença à linguagem. Vamos estudar como fazer isto com um exemplo?

Nosso exemplo será:

Seja uma estrutura simples, que represente uma expressão matemática, do tipo `a+b*c`. O nosso alfabeto será $\Sigma =$ {a, b, c, +, *} e a gramática no padrão EBNF, a qual iremos denominar de **L**, será:

```
<expr> ::= <expr> <op> <expr> | <id>
<id> ::= a | b | c
<op> ::= + | *
```

Seja ω uma palavra (sentença) que pertença à linguagem **L** definida, então $\omega \in L$.

Seja $\omega_1 = a + b$ e desejamos verificar se $\omega_1 \in L$.

Uma forma de verificar isso é por meio do processo de derivação, que consiste em substituir a sentença, ou parte dela, no lado direito da produção da gramática repetidas vezes, até alcançar os símbolos terminais, e, segundo Aho (2007), podemos aplicar repetidamente as produções em qualquer ordem, a fim de obtermos uma sequência de substituições. Esse é o processo de derivação.

Usa-se o símbolo "$\Rightarrow$" para indicar um passo no processo de derivação, assim, segundo Aho (2007), podemos indicar de uma forma mais abstrata se a produção $A \rightarrow \gamma$, e sendo α e β cadeias quaisquer, dizemos que $\alpha A \beta \Rightarrow \alpha \gamma \beta$.

Para ajudar a compreensão do processo de derivação, vamos substituir por letra **MAIÚSCULA** cada **símbolo não-terminal** da gramática, e adotaremos para os **símbolos terminais**, aqueles que pertencem ao alfabeto, neste caso "Σ", a grafia em **negrito** para diferenciá-los dos demais. A seta "$\rightarrow$" indicará a definição da produção, equivalente à notação "::=" no padrão EBNF, e "$\Rightarrow$" indicará **uma** derivação. Assim, de acordo com essas convenções, antes de aplicarmos o processo de derivação à gramática no padrão EBNF, a qual estamos analisando, vamos reescrever a gramática equivalente de acordo com os padrões convencionados aqui, assim teremos:

Gramática padrão EBNF	Gramática equivalente
<*expr*> ::= <*expr*> <*op*> <*expr*>	E $\rightarrow$ E OP E | ID
<*id*> ::= **a** | **b** | c	ID $\rightarrow$ **a** | **b** | **c**
<*op*> ::= **+** | *	OP $\rightarrow$ **+** | *

A cadeia que desejamos derivar é $\omega_1 = a + b$, assim, vamos aplicar as derivações sucessivas até chegarmos aos símbolos terminais.

$\Rightarrow$ E $\rightarrow$ E OP E	1º passo deriva a regra mais à esquerda, pois a cadeia ω_1 tinha um **OP**
$\Rightarrow$ E $\rightarrow$ **ID** OP E	
$\Rightarrow$ E $\rightarrow$ **a** + E	2º passo derivou o **E** mais à esquerda
	3º passo derivou o **ID**. Chegamos ao símbolo terminal **a**
$\Rightarrow$ E $\rightarrow$ **a OP** E	4º passo derivou **OP**.
$\Rightarrow$ E $\rightarrow$ **a** + E	5º passo derivou o **ID**. Chegamos ao símbolo terminal **+**
$\Rightarrow$ E $\rightarrow$ **a** + **ID**	6º passo derivou o **E**
$\Rightarrow$ E $\rightarrow$ **a** + **b**	7º passo derivou **ID**. Chegamos ao símbolo terminal **b.**

Portanto, podemos afirmar que a sentença "a+b" é uma cadeia gerada pela linguagem L, pois, ao fazermos as derivações sucessivas, segundo a gramática **L**, chegamos a uma cadeia de símbolos terminais, logo se $\omega_1 = \mathbf{a} + \mathbf{b}$ então $\omega_1 \in \mathbf{L}$.

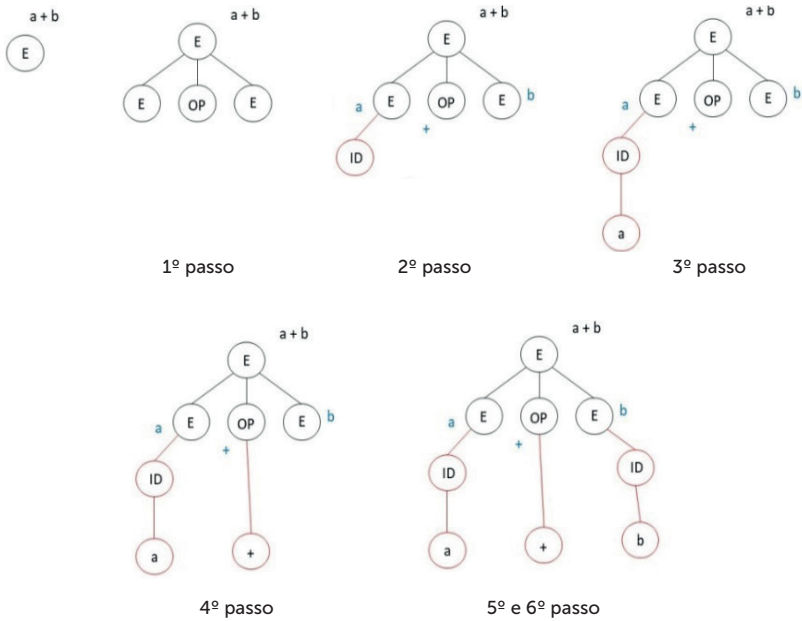
Reflita

Na derivação apresentada, no 2º passo, optamos por derivar mais à esquerda, e assim continuamos até o final. Poderíamos optar por outra ordem, a direta, por exemplo? Como ficaria a derivação, neste caso?

A derivação da árvore sintática, ou gramatical, segundo afirma Aho (2007) pode ser vista como a representação gráfica da derivação. Vimos na Seção 2.3 que os nós tem sucessores, logo são símbolos não terminais, e as folhas são os símbolos terminais, assim, ao construirmos a árvore de derivação de uma sentença de acordo com a gramática de uma linguagem, aplicarmos sucessivamente as regras e alcançarmos as folhas para todos os nós, então a sentença é gerada pela gramática.

Veja isso na construção da árvore de derivação para a sentença 'a + b', pertencente a gramática da linguagem **L**. A Figura 3.2 mostra passo a passo o processo de construção da árvore de derivação para essa sentença.

Figura 3.2 | Passo a passo da árvore de derivação para a sentença "a+b"



Fonte: elaborada pela autora.

Você poderá visualizar o processo de construção da árvore de derivação tambem em vídeo, no link https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/compiladores/u3/s1/VIDEO_1.mp4 ou do QR Code

Tanto a derivação como a árvore gramatical foram derivadas da esquerda para a direita (***L**eft-to-**R**ight)*, mas poderíamos ter feito isso da direita para esquerda (***R**ight-to-**L**eft*). No caso da sentença "a+b", tanto a derivação quanto a árvore seriam iguais se tivéssemos realizado a derivação mais à direita, mas nem sempre é assim. Se sentença a ser analisada for "a + b * c" para a gramática definida, as árvores de derivações serão diferentes se fizermos a derivação mais à esquerda ou mais à direita, e isso não é desejável para uma gramática, pois indica que a gramática é ambígua.

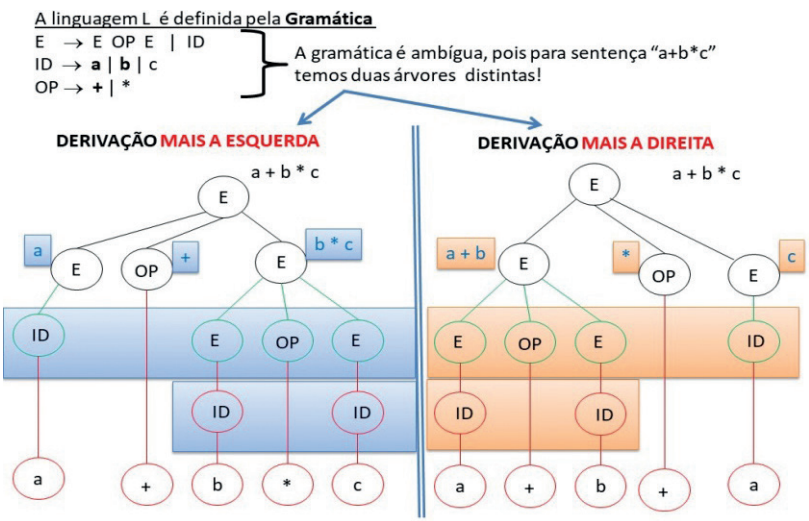
Assimile

"Uma gramática que produza mais de uma árvore gramatical para alguma sentença é dita ambígua" (AHO, 2007, p.77). Isso quer dizer que se uma mesma sentença tiver duas ou mais árvores gramaticais distintas, a gramática é ambígua.

A gramática que define a linguagem L é ambígua, pois a sentença "a + b * c" possui duas árvores gramaticais distintas, conforme pode ser visto na Figura 3.3.

Você poderá visualizar o processo de construção das árvores para a sentença "a+b*c" também em vídeo, no link https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/compiladores/u3/s1/VIDEO_2.mp4 ou do QR Code

Figura 3.3 | Exemplo de Gramática Ambígua



Fonte: elaborada pela autora.

Não é recomendável que a gramática definida para L seja, pois não trata o caso de precedência de operadores. No caso, a multiplicação (*) tem maior precedência do que adição (+). Para

eliminar a ambiguidade, devemos reescrever a gramática, tratando o problema de ambiguidade dos operadores, e, neste caso, construir mais produções para tratar o operador de adição e o de multiplicação distintamente, assim, a nova gramática sem ambiguidade será:

E → E **+** F | F

F → F ***** ID | ID

ID → **a** | **b** | **c**

Após conhecer e compreender a representação da análise sintática, as formas de derivação e o problema da ambiguidade de uma gramática, vamos estudar os tipos de algoritmos que podemos utilizar para construir o analisador sintático.

Como citado na Seção 2.3, há dois tipos de analisadores sintáticos: os **ascendentes**, que constroem a árvore gramatical das folhas para a raiz, chamados também de ***bottom-up***, ou seja, de baixo para cima. E os **descendentes**, que analisam a árvore gramatical da raiz para as folhas, isto é, de cima para baixo, também chamados de ***top-down***. As árvores gramaticais desenvolvidas nos exemplos anteriores foram todas descendentes, pois começamos pela produção inicial (nó inicial) e derivamos cada nó até chegarmos aos símbolos terminais (as folhas). Esse é o método mais intuitivo e usual e consiste em construir a árvore gramatical em pré-ordem (raiz-esquerda-direita), como você pôde acompanhar no processo de construção das árvores apresentadas.

Os analisadores descendentes utilizam um modelo de algoritmo preditivo recursivo, que pode ser com retrocesso ou sem retrocesso. Por que preditivo? É fácil entender. Para isso, vamos considerar a produção:

E → E **+** E | E ***** E | **a**

O algoritmo que irá analisar a sentença de entrada precisará avançar na leitura, além do primeiro *token*, para saber se escolhe a primeira opção, "E **+** E" , a segunda, "E ***** E", ou, ainda, a terceira, caso haja apenas o *token* '**a**', apesar de estar analisando apenas o primeiro *token*. Ou seja, um algoritmo preditivo é aquele que avança para ser capaz de tomar uma decisão agora, com base em uma informação que que será inserida no futuro, mas que se conhece previamente.

E o que significa com retrocesso ou sem? Um algoritmo de análise sintática **com retrocesso** faz a escolha da primeira opção

e avança na construção da árvore gramatical, armazenando os dados em uma pilha, até alcançar o final, mas se não reconhecer a sentença, retrocede, desempilhando os dados até o nó inicial e reinicia o processo. Esse algoritmo é ineficiente do ponto de vista do tempo gasto na análise (o vai e volta), do tratamento da recuperação de erros, e, também, para da análise semântica. Mas, e o algoritmo de análise sintática sem retrocesso, como funciona afinal?

Como o próprio nome diz, o objetivo desse algoritmo é não retroceder, isto é, não voltar. Para que seja possível utilizar um algoritmo de **análise sintática recursivo preditivo sem retrocesso** há uma condição: a gramática não pode ter recursão a esquerda. Assim, basta conhecer apenas o primeiro o *token* de entrada e a produção à frente para expandir a árvore. Esse tipo de analisador é capaz de derivar a maioria das linguagens de programação, por serem do tipo livre de contexto.

Exemplificando

Vamos ver na prática como eliminar a recursão à esquerda da gramática?

Seja gramática: $E \longrightarrow E + F \mid F$

$F \longrightarrow F * ID \mid ID$

$ID \longrightarrow \mathbf{a} \mid \mathbf{b}$

Vamos fatorar essa gramática para eliminar a recursividade à esquerda, assim:

$E \longrightarrow F\ E'$	Deslocamos a recursão para a direita
$E' \longrightarrow + F\ E' \mid \varepsilon$	Criamos uma nova produção (fatoração), assim o símbolo terminal fica à esquerda, seguido da produção, e a recursão à esquerda. O processo recursivo termina em vazio.
$F \longrightarrow ID\ F'$	O mesmo conceito aplicado para E é reproduzido para eliminar a recursividade em F
$F' \longrightarrow * ID\ F' \mid \varepsilon$	
$ID \longrightarrow \mathbf{a} \mid \mathbf{b} \mid \mathbf{c}$	

A nova gramática é equivalente, isto é, gera a mesma linguagem e não apresenta recursividade à esquerda.

Agora que você já aprendeu a fatorar, também estará alerta ao escrever sua gramática, e irá procurar escrevê-la sem recursão a esquerda, o que evitará o trabalho de fatoração.

Você poderá visualizar o processo detalhado de fatoração da gramática apresentado, também em vídeo, no link https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/compiladores/u3/s1/VIDEO_3.mp4 ou do QR Code

Pesquise mais

Sendo a análise sintática o centro do processo de análise, é importante procurar conhecer mais a respeito deste tema. Assim, indicamos alguns links para que possa aprofundar e reforçar pontos fundamentais estudados nesta seção:

1. Você pode observar que os algoritmos para análise sintática usam os conceitos de estrutura de dados, portanto, é fundamental ter domínio sobre este tema. Que tal revisar estes conceitos? O link abaixo mostra os conceitos básicos de estrutura de árvore:

   UNIVESP. **Estrutura de Dados** - Aula 15 - Árvores - Conceitos básicos. 29 ago. 2016. Disponível em: <https://www.youtube.com/watch?v=eiMMtyRBYCE>. Acesso em: 30 jul. 2018.

   Recomendamos assistir a partir de 5m22s minutos para entender pré-ordem, mas todo o vídeo é relevante, se você julgar necessário rever plenamente os conceitos básicos de árvore.

2. Esse outro vídeo é ótimo para mostrar o passo a passo do algoritmo de análise sintática top-down (descendente) preditivo.

   MARCELO ROMEU GONÇALVES. **Análise Descendente (top down)**. 2 dez. 2013. Disponível em: <https://www.youtube.com/watch?v=hLndvXdlc0M>. Acesso em: 30 jul. 2018.

   Recomendamos, ao assistir a esse vídeo, ativar as caixas de som.

3. E, para você ir além, deve e saber que os conceitos estudados em compiladores estão integrados a diversas áreas da computação e, hoje em dia estão fortemente utilizados nas novas tecnologias, tal como os modelos preditivos usados para "prever o futuro" e

"ensinar a aprender", além de serem aplicados a tecnologia em IA (inteligência artificial). Mas, não se engane as previsões em computação são feitas a partir de dados.

Conheça mais sobre este assunto, pois com certeza irá ajudá-lo a manter-se atualizado com as tendências do mercado e que prever o futuro na computação é uma fórmula matemática.

a) MICROSOFT. **O que é Power BI.** [S.l.; s.d.]. Disponível em: <https://powerbi.microsoft.com/pt-br/what-is-power-bi/>. Acesso em: 30 jul. 2018.

   DATA SCIENCE ACADEMY. **Cap09** - O Que é Um Modelo Preditivo - Parte 1. 3 ago. 2017. Disponível em: <https://www.youtube.com/watch?v=bnSIkNKvlD0>. Acesso em: 30 jul. 2018.

b) TAURION, C. **Saiba como usar algoritmos preditivos.** 9 fev. 2015. Disponível em: <https://imasters.com.br/devsecops/saiba-como-usar-algoritmos-preditivos>. Acesso em: 30 jul. 2018.

Nesta seção foi possível conhecer como funciona o algoritmo para análise sintática descendente preditiva recursiva sem retrocesso, também conhecida por análise *top-down*, entender a diferença entre análise com ou sem retrocesso, realizar a fatoração de uma gramática para eliminar a recursividade à esquerda e, ainda, a importância de se evitar construir gramáticas ambíguas. Para dominar todo o processo de análise de um compilador, falta apenas um passo: o estudo da análise semântica, que será o objeto da nossa próxima seção, e a tradução dirigida pela sintaxe, que utilizada para desenvolver a análise semântica. Vamos continuar?

Sem medo de errar

Realmente, esta seção era o que faltava após você ter sido convidado para participar de um processo seletivo em um instituto de pesquisas que possui um laboratório dedicado aos estudos sobre linguagens de programação.

Durante sua primeira entrevista, você recebeu a sintaxe da linguagem (Figura 3.1) e prontamente se questionou se a representação da sintaxe estava correta. Logo no início da análise,

pôde observar que havia alguma redundância. No caso, o diagrama da produção 'programa', apresenta três alternativas e pode ser substituído simplesmente pela produção 'comando'. Por outro lado, não existem as produções 'atribuição', 'comm_se' e 'comm_loop', mas foram incluídas na produção 'programa'.

Após identificar essas falhas, você pôde perceber como estava bem à vontade para fazer esta análise, afinal, foram temas já estudados nas seções 1.2 e 2.1, respectivamente. A questão é: como fazer a análise sintática dos comandos de acordo com a gramática que você precisa criar?

Depois dessa seção, tudo ficou mais claro. Agora, você sabe que é necessário tomar alguns cuidados para construir as gramáticas:

1. Analisar se a gramática definida é ambígua. Caso afirmativo, reescrevê-la, eliminando a ambiguidade;
2. Ao construir a gramática, não utilizar recursão a esquerda para que possa utilizar um analisador *top-down* sem retrocesso. E, caso haja recursão a esquerda, pode-se fatorar para eliminá-la.

Como seu objetivo será a apresentação para os examinadores, e como eles analisarão as justificativas, além da solução, que tal incluir nos slides os pontos levantados acima e um exemplo para mostrar um caso de ambiguidade, outro de recursividade a esquerda e, ainda, a respectiva fatoração? Para isso, sugerimos o comando de "atribuição" dessa gramática, pois apresenta a mesma estrutura do exemplo desenvolvido nessa seção. E, para atender as solicitações feitas:

A. Avaliar o diagrama entregue e elaborar a definição EBNF correspondente;

B. Fazer a representação da análise sintática da definição EBNF criada;

C. Apresentar os resultados e justificar os métodos utilizados.

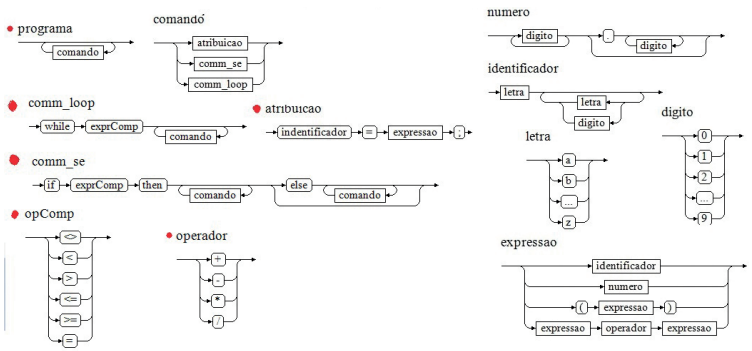
Que tal organizar a sua apresentação desta forma:

i) Abertura com uma apresentação pessoal, rápida, dinâmica e com os links dos trabalhos desenvolvidos em sua carreira universitária, não se esquecendo de agradecer a oportunidade de estar participando do processo seletivo;

ii) Em resposta ao item A, que tal mostrar que faltam alguns diagramas e que o diagrama "programa", apesar de correto, não acompanha a produção "comando", além de apresentar

a resposta, representada na Figura 3.4. À esquerda da imagem estão as correções e à direita os demais diagramas.

Figura 3.4 | Diagrama de sintaxe proposto



Fonte: elaborada pela autora.

iii) Ainda com relação ao item A, construa a gramática na notação EBNF, evitando ambiguidade e sem utilizar recursão a esquerda. Aproveite para chamar a atenção para esses fatos, pois, no futuro, o analisador sintático *top-down* sem retrocesso exigirá isso. As regras corretas para a gramática são:

Tabela 3.1 | Regras sintáticas e léxicas para a gramática

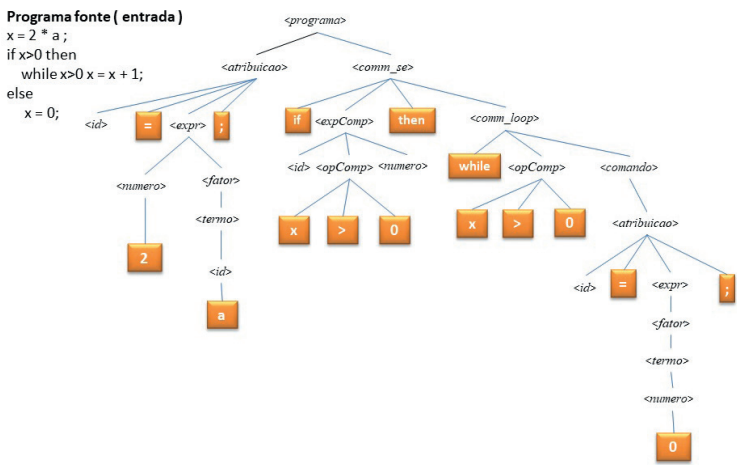
Regras Sintáticas	Alguns Comentários
<*programas*> ::= {<comando>}	AS preditivo
<*comando*> ::= <*atribuição*> | <*comm_se*> | <*comm_loop*>	
<*comm_loop*> ::= while <*expr*> <*comando*>	<*comm_sef*> construída sem recursão a esquerda. Assim, para a sentença:
<*comm_se*> ::= **if** <*exprComp*> **then** <*comando*> <*comm*>'	"if a>100 then if (b>5) x=1; else x=2;"
<*comm*>' ::= **else** <*comando*> | ε	Há somente uma árvore de derivação, logo a produção não gera ambiguidade.
<*exprComp*> ::= (<*id*> | <*numero*>) <*opComp*> (<*id*> | <*numero*>)	
<*atribuição*> ::= <*id*> = <*expr*>	
<*expr*> ::= <*fator*> <*expr1*>	A produção <*expr*> foi fatorada para eliminar a recursividade a esquerda

Regras Sintáticas	Alguns Comentários
<*expr1*> ::= + <*fator*><*expr1*> | - <*fator*><*expr1*> | ε <*fator*> ::= <*termo*><*fat1*> <*fat1*> ::= * <*termo*><*fat1*> | / <*termo*><*fat1*> | ε <*termo*> ::= <*numero*> | <*id*> | (<*expr*>)	
Regras léxicas	
<*numero*> ::= <*digito*> {<*digito*>} [. <*digito*> {<*digito*>}] <*id*> ::= <*letra*> {<*letra*> | <*digito*> } <*letra*> ::= a | b | c |...| z <*numero*> ::= 0 | 1 | 2 | 3 | ... | 9 <*opComp*> ::= <> | < | > | <= | >= | =	

Fonte: elaborada pela autora.

iv) Para a representação da análise sintática, desenvolva a árvore gramatical descendente para um exemplo de programa com todos os comandos da linguagem, conforme a Figura 3.5.

Figura 3.5 | Árvore gramatical descendente



Fonte: elaborada pela autora.

v) Finalmente, conclua a apresentação mostrando que a solução atende aos conceitos de análise sintática descendente recursiva preditiva sem retrocesso, e esta é uma solução de fácil implementação, permite uma recuperação de erros eficiente e pode ser usada para a maioria das linguagens de programação, desde que a gramática não apresente recursividade a esquerda.

O resultado da sua exposição será um sucesso, pois, ao aplicar os conceitos estudados para construir uma gramática sem recursividade a esquerda, não ambígua e pronta para ser verificada por um analisador *top-down* sem retrocesso, o resultado não poderá ser outro a não ser: "convocado para a segunda etapa do processo seletivo".

Esteja preparado para avançar à próxima etapa: a análise semântica. Assim, com certeza, você irá obter o melhor resultado.

Avançando na prática

Cuidados ao criar novas versões para linguagens já existentes

Descrição da situação-problema

Uma empresa desenvolveu uma linguagem simples de programação, praticamente uma DSL (*Domains-Specific Language*), com o objetivo específico de controlar as esteiras do depósito de produtos. A linguagem está em uso há dois anos e funcionando muito bem, mas precisaram implementar uma alteração em um comando e agora nada funciona. O analisador sintático está se perdendo e entrando em loop, e não encontram o erro. Na versão atual, o comando de atribuição era simples, do tipo:

armazem1 = valor; ou armazém1 = armazemN; ou ainda, armazen1 = x;

Agora, na nova versão, os armazéns podem assumir valores iguais aos demais e precisam habilitar isso em um só comando do tipo:

armazem1 = armazen2 = armazem3 = ... ;

A alteração feita na gramática foi a seguinte:

Gramática anterior	Alteração
<atrib> ::= <variável> = <expr> ;	<atrib> ::= <parteEsq> {<parteEsq> = } <expr> ; <parteEsq> ::= <variável> =

Qual será o problema que está ocorrendo? Será que o erro está na implementação ou na definição da gramática? Como solucionar?

Resolução da situação-problema

O problema apresentado é um típico comando já existente em linguagens como C e Java, portanto há solução para isto. O primeiro passo para solucionar o problema deve ser a análise da documentação do sistema. E, se tratando de uma linguagem, o primeiro documento a analisar deve ser a especificação da gramática.

Ao estudá-la e a documentação do analisador sintático (AST), verificou-se que a primeira versão utiliza analisador descendente sem retrocesso e a especificação estava coerente para esse tipo de analisador, pois não tinha recursão à esquerda, mas na alteração feita o comando de atribuição apresenta recursão à esquerda, logo, é necessário reescrever esse comando de acordo com os requisitos desse tipo de analisador, ou seja, sem recursão à esquerda, e, para isso, é preciso fatorar o comando <atrib> para eliminar a recursão à esquerda. Assim, a solução será:

Alteração	Correção
<atrib> ::= <parteEsq> {<parteEsq> = } <expr> ; <parteEsq> ::= <variável> =	<atrib> ::= <parteEsq> <expr> ; <parteEsq> ::= <variável> = {<parteEsq>} | ε

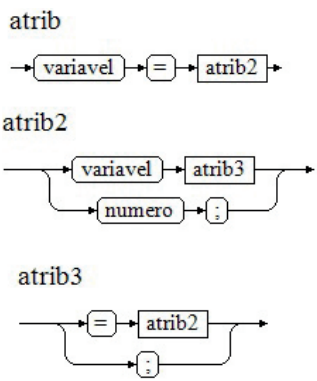
Mas, ainda assim, ao implementarem a correção, continuou ocorrendo erro para alguns casos. Observou-se, então, que apesar de não haver recursão à esquerda, as produções <expr> e <parteEsq> podem começar por <variável>, o que invalida o método descendente sem retrocesso, pois ambas produções teriam o mesmo ponto de início, gerando um indeterminismo.

Logo, é importante ser criterioso ao alterar uma gramática, e analisar o ponto de início para o reconhecimento de um comando. Na primeira versão, não havia um caso assim, mas a alteração gerou essa situação. Para solucionar o problema, pode ser feita uma

alteração no comando, que simplifica as alterações no AST e não limita a linguagem, pois a outra solução, com retrocesso, implica em alterações mais profundas no AST e a inclusão de vários comandos de marcações de início (*lookahead*) para tratar este caso.

As marcações caracterizam uma forma de reduzir o processo de "ida e volta" para AST com retrocesso, logo, o projeto teria suas características alteradas e outros erros poderiam surgir. Assim a solução encontrada foi:

Sintaxe:



Gramática EBNF:

<*atrib*> ::= <*variável*> "=" <*atrib2*>

<*atrib2*> ::= <*variável*> <*atrib3*> | <*numero*>

<*atrib3*> ::= "=" <*atrib2*>

O problema encontrado neste caso é típico de gramáticas que começam com símbolos não-terminais iguais em uma mesma produção. Veja o exemplo a seguir:

<*comm_se*> ::= **if** <*condição*> **then** <*comm_se*> **else** <*comm_se*> | **if** <*condição*> **then** <*comm_se*>

Neste caso, **if** <condição> **then** <comm_se> aparece no início das duas opções, e o analisador, para escolher a opção correta, precisará avançar cinco (5) *tokens* à frente. Para solucionar este caso, pode-se reescrever a gramática:

<*comm_se*> ::= **if** <*condição*> **then** <*comm_se*> <*c_else*>

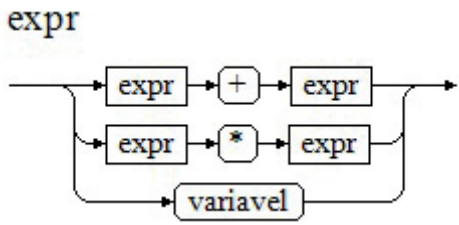
<*c_else*> ::= **else** <*comm_se*> | ε

Essa solução, além de não precisar usar o retrocesso, também elimina a ambiguidade da gramática.

Faça valer a pena

1. A gramática de um comando define as regras sintáticas para geração do mesmo, já o diagrama de sintaxe é a representação gráfica e facilita o entendimento da estrutura do comando.
Quando usamos a notação EBNF para definir um comando, é recomendável não utilizar recursão à esquerda, para facilitar a implementação do analisador sintático descendente sem retrocesso.
Analise o diagrama de sintaxe a seguir:

Figura | Diagrama de sintaxe



Fonte: elaborada pela autora.

Esse diagrama pode ser definido por uma gramática sem recursão à esquerda.

Qual das alternativas a seguir define a gramática sem recursão à esquerda para esse diagrama?

a) <exp> ::= <expr> "+" <expr> | <expr> "+" <expr> | <variável>

b) <exp> ::= <expr> { ("+" | "-") <expr> } | <variável>

c) <exp> ::= <variável> <expr1>
<exp1> ::= ("+" | "*") <variável> <expr1> | ε

d) <expr> :: = <variável> <expr1>
<exp1> ::= ("+" | "*") <variável> <expr1> | <expr2>
<exp2> ::= <variável> | ε

e) <exp> ::= <variável> <expr1>
<exp> ::= "+"<variável> <expr1>
<exp1> ::= "*"<variável> <expr2>
<exp2> ::= <variável>

2. Uma gramática é dita ambígua se existe mais de uma árvore de derivação para a mesma sentença. Contudo, não podemos provar que uma gramática não é ambígua, mas, para provar a ambiguidade, basta encontrar uma sentença gerada pela gramática que gere pelo menos duas árvores distintas. Dada a gramática:

S → **if** C **then** S **else** S | **if** C **then** S | **a**
C → **a=b;**

Em que os símbolos em negritos são símbolos terminais e os símbolos em maiúsculos são não terminais.

Com relação à gramática apresentada, é correto afirmar:

a) A gramática não é ambígua, pois essa é a estrutura padrão de um comando de tomada de decisão, presente na maioria das linguagens de programação.
b) É uma gramática ambígua, por ser não determinística ao utilizar o operador |, gerando a alternância na produção;
c) Nada podemos afirmar sobre a gramática, pois não é possível provar se uma gramática não é ambígua.
d) É uma gramática ambígua, pois a sentença "**if a=b then if a=b then a else a"** gera duas árvores de derivações distintas.
e) Não é uma gramática ambígua, pois a sentença "**if a=b then a else a**" é representada por apenas uma árvore de derivação.

3. Os analisadores sintáticos são classificados segundo a forma pela qual a árvore de derivação da sentença analisada é construída, podendo ser feita de duas formas, E a estrutura de árvore permite a implementação do processo recursivo presente nas estruturas gramaticais.

Sobre os tipos de analisadores sintáticos, a alternativa correta é:

a) Analisadores sintáticos ascendentes preditivos com retrocessos são fáceis de serem implementados, rápidos e o apresentam tratamento de erros eficiente.
b) Analisadores sintáticos descendentes preditivos recursivos com retrocessos são fáceis de serem implementados, rápidos e apresentam tratamento de erros eficiente.
c) Analisadores sintáticos descendentes preditivos recursivos sem

retrocessos são fáceis de serem implementados, rápidos, apresentam tratamento de erros eficiente e podem ser aplicados a qualquer gramática.

d) Analisadores sintáticos ascendentes preditivos recursivos sem retrocessos são fáceis de serem implementados, rápidos e apresentam tratamento de erros eficiente;

e) Analisadores sintáticos descendentes preditivos recursivos sem retrocessos são fáceis de serem implementados, rápidos, apresentam tratamento de erros eficiente e a gramática não pode ter recursão à esquerda.

Seção 3.2

Tradução dirigida pela sintaxe

Diálogo aberto

Caro aluno, depois de conhecer as técnicas para construção do analisador sintático, é chegado o momento de avançar mais um pouquinho na fase de análise de um compilador e estudar como desenvolver a análise semântica para as linguagens livres de contexto (LLC). A técnica utilizada para isto é a tradução dirigida pela sintaxe ao qual será nosso objeto do estudo nesta seção.

O processo seletivo do laboratório de pesquisas em linguagens de programação, do qual você aceitou participar, dividiu a metodologia de seleção em duas fases, sendo a primeira o projeto conceitual da análise sintática, na qual você se saiu muito bem e foi aprovado para continuar. Agora, você deverá incorporar os conceitos para análise semântica à proposta apresentada da análise sintática para concluir esta fase.

Na segunda fase, os candidatos deverão demonstrar conhecimentos para trabalharem com as estruturas de dados que envolvem a implementação da tabela de símbolos na construção dos compiladores, mas, antes disso, você precisa concluir a primeira fase. Assim, o foco será como analisar as dependências dos atributos envolvidos nos comandos, afinal, não é possível definir uma regra do tipo "se a variável não existe, o comando não é válido" ou "os tipos envolvidos no comando são compatíveis".

Você foi muito bem na primeira etapa da fase inicial, corrigindo os erros na gramática proposta e mostrando alternativas, sempre com uma fundamentação teórica. Agora você poderá, com as técnicas de tradução dirigida pela sintaxe que serão apresentadas, seja por esquema de tradução ou atributos, incorporar o tratamento da semântica na gramática proposta e atender aos quesitos que serão avaliados: a) realização dos ajustes necessários na gramática da linguagem, para incorporar as definições necessárias para o tratamento da semântica; b) apresentação de como pretende realizar o tratamento dos erros de semântica; e c) clareza da apresentação, tecnologia utilizada, desenvoltura e domínio do assunto continuam valendo nesta etapa,

tanto quanto à forma da apresentação, que continuará livre e a duração não deverá ultrapassar a 30 minutos e não inferior a 20.

Para montar sua apresentação, quais os passos devem ser elencados para prepará-la? Ainda, como você organizará as informações dentro do tempo estipulado?". Parece que você seguiu um bom caminho e já desenvolveu a parte mais difícil, agora somente terá que aplicar as técnicas que irá conhecer nesta seção para avançar no seu objetivo: ser aprovado para a segunda etapa do processo. Entusiasmado para conhecer as técnicas de tradução dirigida pela sintaxe?

Não pode faltar

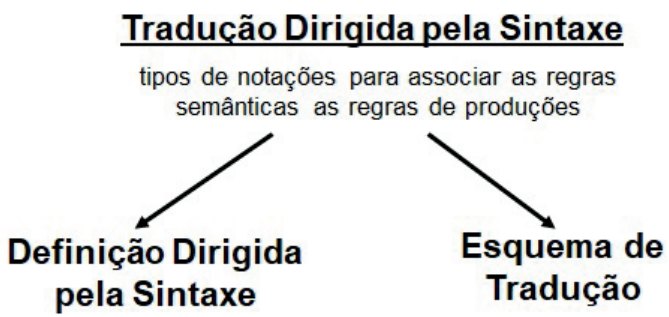
Após conhecer sobre a análise sintática e as técnicas para construir um analisador sintático. Também foi possível identificar que a maioria das linguagens de programação podem ser definidas por linguagens livre de contexto (LLC). Mas, para completar a fase de análise de uma gramática, é necessário verificar se o contexto da frase, no caso das linguagens de programação os comandos, estão corretos. A esse processo de análise do contexto da frase denominamos análise semântica.

Como estudamos, o coração da análise é o analisador sintático, que recebe dados do léxico e verifica se existe uma árvore de derivação para o fluxo de dados gerado pelo léxico, seguindo as regras da gramática da linguagem. Já os aspectos semânticos são tratados por sub-rotinas específicas acionadas pelo analisador sintático. Durante as etapas da análise léxica, sintática e semântica, é montada a tabela de símbolos, a qual é utilizada em todas as etapas do desenvolvimento do compilador, e será objeto de estudo especifico. Appel, dá uma definição da fase de análise semântica bem simples e objetiva:

> **A fase de análise semântica de um compilador conecta as definições das variáveis com o seu uso, verifica se cada expressão tem um tipo correto e traduz a sintaxe abstrata em uma representação mais simples e adequada para a geração do código de máquina (Appel, 2002, p.103, tradução nossa).**

Sendo, assim depois do analisador sintático ter verificado a estrutura do comando e se os tipos envolvidos na estrutura são compatíveis, isto é, feita a análise semântica, o passo seguinte será a geração de código. Feitas as definições conceituais iniciais, é o momento de conhecer como associar os aspectos semânticos à gramática. Segundo Aho (2007), há dois tipos de notações para isto: a) definições dirigidas pela sintaxe e b) esquemas de tradução. A Figura 3.6 mostra esquematicamente a tradução dirigida pela sintática.

Figura 3.6 | Notações para a tradução dirigida pela sintática



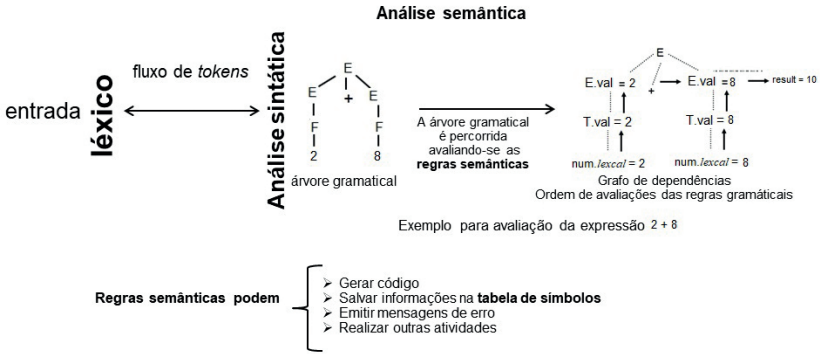
Fonte: elaborada pela autora.

Na **definição dirigida pela sintaxe**, as especificações são de alto nível e a cada *token* associamos um conjunto de atributos. Com base na árvore de derivação, esses atributos são:

- **Sintetizados**: quando é anotada a avaliação da regra semântica de cada nó, de baixo para cima, ou seja, da folha para a raiz.
- **Herdados**: quando o valor semântico do nó é definido pelo nó superior (pai) e o lateral (irmão). Por exemplo, no caso de uma variável aparecer do lado esquerdo e direito de uma atribuição.

Já nos **esquemas de tradução**, inserem-se as ações semânticas no lado direito das produções e é possível determinar a ordem em que as ações e as avaliações dos atributos irão ocorrer. Apresentamos um esquema do processo da tradução dirigida pela sintaxe na Figura 3.7. Vale lembrar que esse processo é igual para os dois tipos de notações expostos na Figura 3.6.

Figura 3.7 | Processo da tradução dirigida pela sintaxe



Fonte: elaborada pela autora.

Assimile

Lembrando: lexema é o valor representado pelo padrão do token.

Tomemos como exemplo "valor = 10". Na análise léxica desse comando, identificamos que:

- "valor" é o lexema e o *token* é ID
- "10" é o lexema e o *token* é NUMERO

O analisador léxico para integrar-se aos analisadores sintático e semântico deverá, além de identificar o tipo do token, **adicionar na tabela de símbolos (TS) os atributos do *token***, por exemplo: nome do token, o lexema e o valor.

No exemplo apresentado, uma ação semântica será addTS(ID,"valor",tipo.inteiro), que indica inclusão na tabela de símbolo do ID, o lexema e o valor, e a outra ação será addTS(NUMERO,10,tipo.inteiro).

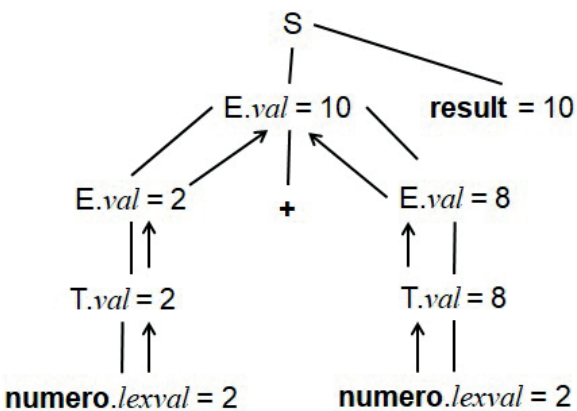
É função do analisador léxico fornecer o atributo sintetizado para os *tokens*, o qual será identificado pela notação ***lexval***, e, nesse exemplo, o símbolo terminal é o token NUMERO. ***lexval***=10.

A árvore gramatical representada na Figura 3.7 corresponde à sentença 2 + 8, e, à direita da Figura 3.7, temos uma árvore gramatical anotada por meio das regras semânticas. Para melhor compreensão da árvore anotada, vamos comparar as regras de produção e as regras semânticas para este caso:

Regras de produção	Regras semânticas	Comentários
S :=**result**	Resultado (E.*val*)	**result** é um terminal que representa o resultado do cálculo.
E := E + E | T	E.*val* = E.*val* + E.*val* E.*val* = T.*val*	Quando realizamos a soma de valores, há dependência entre os termos, assim, a definição dirigida por sintaxe associa um atributo a cada termo. Neste caso, ***val*** indica o tipo e está associado aos símbolos não terminais
T := **numero**	T = **numero**. *lexval*	**numero** é um símbolo terminal e está associado a um atributo sintetizado, pois, neste caso, não há regra semântica para o mesmo, já que o valor deverá ser fornecido pelo léxico.

A árvore gramatical anotada correspondente às produções e respectivas regras semânticas é apresentada na Figura 3.8.

Figura 3.8 | Árvore gramatical anotada



Fonte: elaborada pela autora.

Os atributos **numero**.*lexval* e T.*val* são atributos sintetizados, bem como o E.*val*. Nas regras de produção, o símbolo não-terminal '**E**' aparece à esquerda e à direita na produção, caso em que teremos um atributo herdado. Como vimos na seção anterior, é possível fatorar a gramática para evitar esse tipo de construção e, assim, trabalharmos

apenas com atributos sintetizados. Uma definição dirigida pela sintaxe somente com atributos sintetizados, segundo Aho (2007), é chamada de **definição S-atribuída**. Ainda sobre a Figura 3.8, as setas indicam o grafo de dependência para a árvore gramatical: observe que o nó E.*val*.=10 depende dos nós à direita e à esquerda.

Exemplificando

Dada a gramática:

<*decl*> ::= <*tipo*> <*listaVariaveis*>

<*tipo*> ::= **int** | **double**

<*listaVariaveis*> ::= **id ,** <*listaVariaveis*> | **id**

Vamos incluir as regras semânticas usando a definição dirigida pela sintaxe, em que cada símbolo tem um conjunto de atributos associados.

1. Vamos usar a notação formal para deixar a definição dos atributos mais evidente:

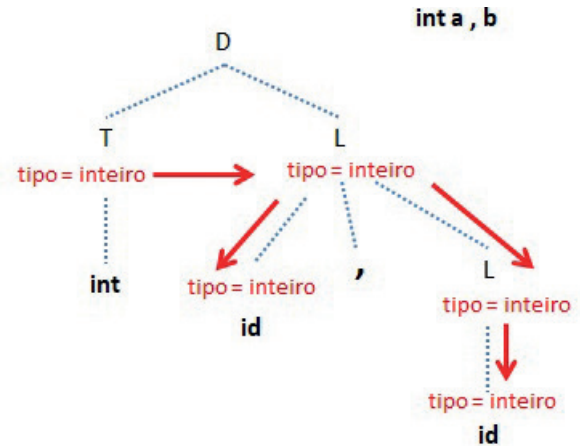
D := T L	D é o não terminal para <*decl*> T é o não terminal para <*tipo*> L é o não terminal para <*listaVariaveis*>
T := **int** | **double**	A produção T nos leva aos símbolos terminais **int** ou **double**
L := **id ,** L | **id**	id e , são símbolos terminais

2. Agora vamos inserir as regras de atributos. Para que você possa entender melhor, leia os comentários na sequência 1 e 2, assim será fácil observar que L tem uma dependência e D uma dependência herdada:

Regras de Produção	Regras Semânticas		Comentário
D := T L	L.tipo = T.tipo	1	Separamos a produção, pois **cada tipo de dado tem valor específico**, logo ações diferentes.
T := **int**	T.tipo = inteiro		
T := **double**	T.tipo = real		
L := **id**	id.tipo = L.tipo	2	O identificador (variável) deve ter o mesmo tipo da lista de variáveis(L)
L := **id ,** L	id.tipo = L.tipo		

3. Para concluirmos nosso exemplo, vamos construir a árvore de derivação anotada para a sentença **int a , b:**

Figura 3.9 | Árvore anotada com o grafo de dependência



Fonte: elaborada pela autora.

A Figura 3.9 mostra a árvore anotada para a sentença **int a, b**. As setas indicam as dependências e seguem a sequência 1> 2> 3, conforme a gramática de atributos definida no item 2.

De acordo com o apresentado, observamos que a tradução dirigida pela sintaxe é fortemente relacionada com o analisador sintático e as regras semânticas da linguagem, entretanto não são plenamente especificadas na gramática. Entre as notações disponíveis na tradução dirigida pela sintaxe, a melhor forma utilizada nas linguagens modernas para especificação formal das regras semânticas é a definição da gramática de atributos em conjunto com as regras e ações semânticas.

As ações semânticas muitas vezes são implementadas por meio de de sub-rotinas e, no processo de verificação da semântica, é necessário apontar os erros: a) de declaração de identificadores; b) compatibilidade de tipos; e c) compatibilidade entre parâmetros e argumentos. Vejamos o exemplo apresentado a seguir do trecho de um programa em C e o tratamento para cada erro semântico existente no código:

	Código em C	Identificação dos erros - mensagem
1	`void funcao1(int a, char`	
2	`b){`	
3	`// comandos`	
4	`}`	
5	`char funcao2(double c){`	
6	`// comandos`	(E1)-Há um erro semântico de
7	`return 10.5;`	incompatibilidade no retorno da
8	`}`	função – linhas 4 e 6
9	`int main(){`	
10	`int x, y;`	(E2)-Há um erro de
	`x = '1';`	incompatibilidade de tipo
		entres as linhas 9 e 10
11		
	`k = 10;`	(E3)-[Error] 'k' was not declared
		in this scope (erro de declaração
		da variável) – linha 11
12		
	`x = funcao2(y);`	(E4)-Há um erro de incompatibilidade
		entre as linhas 4 e 12
13		
	`funcao1(x);`	
14	`}`	(E5)- [Error] too few
		arguments to function 'void
		funcao1(int, char)
		[Note] declared here (linha 1)
		Erro de compatibilidade
		entre argumentos – linha 4
		e 13. Foram passados menos
		argumentos na chamada em
		relação a declaração)

Observe que os erros (E3) e (E5) são identificados pelo compilador C, **mas** os erros (E1), (E2) e (E4) não são identificados pelo analisador semântico do compilador, pois esse compilador não implementa a análise da equivalência de tipos. Isso é uma indicação clara de como

a definição das regras semânticas possuem complexidade e nem sempre todos os erros semânticos são implementados pelo projetista do compilador. A implicação disso é que o programador poderá ter dificuldade para encontrar erros em seus programas, pois, no caso apresentado, não ocorre erro de compilação nem erro de execução, mas ocorrerá um erro de saída não esperado, o chamado erro lógico, e erros lógicos são difíceis de serem corrigidos. Outros detalhes importantes com relação ao tratamento de erro são acuracidade e clareza das mensagens do tratamento dos erros. Se a linha 10 fosse escrita x = "1", o compilador C apontaria o seguinte erro:

```
[Error]invalid conversion from 'const char*' to 'int'
```

Isso suscita o questionamento: quando uso a aspa dupla, ocorre a análise de equivalência de tipo? A resposta é sim e não. Ao usar a aspa dupla, o compilador reconhece ser necessária a declaração da estrutura para representar um vetor de *char*, e a ação tratada pelo analisador semântico é "não existe este tipo". Se alterarmos o parâmetro da função **main** para:

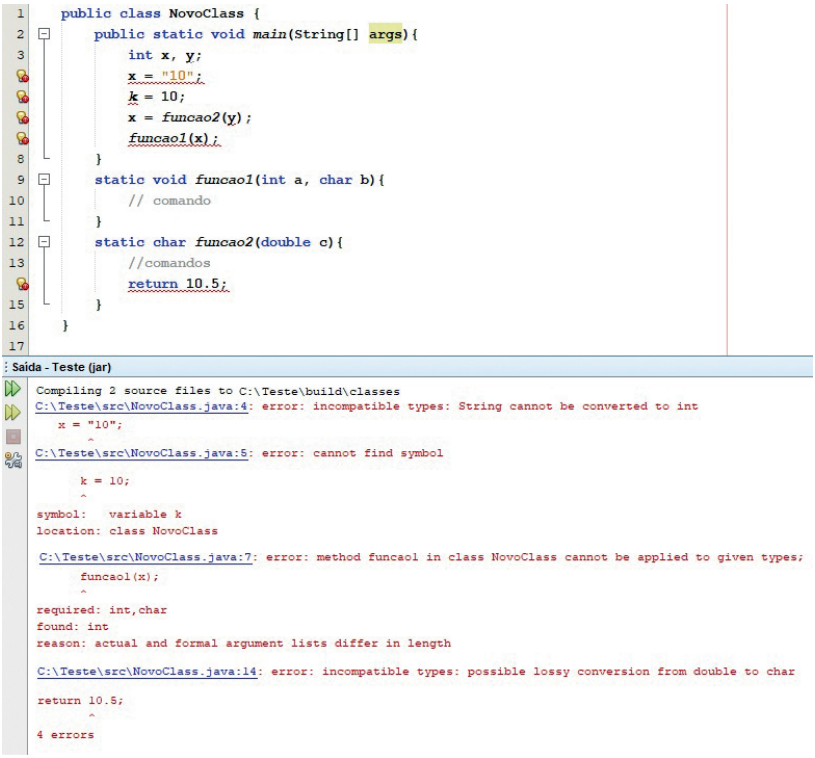
```
int main(char *argv[])
```

E mantivermos x = "1"; o compilador indicará apenas um alerta:

```
In function 'main':
[Warning] assignment makes integer from pointer without a cast
```

Tal alerta indica que, na linha 10, não é possível realizar a conversão de tipo, mas irá gerar o código executável, ou seja, a análise da equivalência de tipos é parcial. Para mostrar uma comparação com outra linguagem utilizando o mesmo exemplo, com as devidas alterações, já que as linguagens tem sintaxe diferente, na Figura 3.10 pode ser visto um exemplo em JAVA e as respectivas mensagens geradas pelo compilador, as quais são mais precisas, detalhadas e todos os erros semânticos são identificados.

Figura 3.10 | Exemplo de tratamento de erros de semânticos de um programa na linguagem Java



Fonte: captura de tela da IDE Netbeans, elaborada pela autora.

Reflita

A função do analisador sintático, na teoria, é informar se uma sentença pertence ou não a uma linguagem. Mas, na prática, ou seja, na hora da implementação, é isto que ocorre? Não.

Como foi visto, associamos aos símbolos da gramática um conjunto de atributos e, a cada produção, um conjunto de regras semânticas para analisar a compatibilidade de tipos, quantidades de parâmetros, escopo, etc. A técnica para especificar esta associação é a notação definição dirigida pela sintaxe.

Nesta seção, estudamos como realizar a análise semântica, e pudemos conhecer as técnicas de definição dirigida pela sintaxe e o esquema de tradução, ambos utilizados na tradução dirigida pela sintaxe para implementação das ações semânticas no analisador sintático. Também conhecemos os tipos de erros que devem ser tratados pelo analisador semântico, e, devido à complexidade desta fase da análise, foi possível perceber que o método comumente utilizado para a descrição formal da tradução dirigida pela sintaxe privilegia o uso da gramática de atributos e ações, mas, ainda assim, cabe ao projetista desenvolver rotinas para implementar todas as regras semânticas da linguagem.

Pesquise mais

O tema análise semântica é a etapa de maior complexidade da fase de análise, por isso é recomendável rever os conceitos estudados nesta seção, assim, indicamos a seguir um vídeo para essa revisão e um material de consulta para ajudá-lo:

a) GEOVANE GRIESANG. **Compiladores** - Análise semântica - Parte 01/02. 18 nov. 2013. Disponível em: <https://www.youtube.com/watch?v=jMohFC244ic>. Acesso em: 31 jul. 2018.

b) SANTOS, R. R. **Aula 1802 –** Fases de um compilador. Jul. 2006. Disponível em: <http://www.facom.ufms.br/~ricardo/Courses/CompilerII-2009/Material/Ricardo/Traducao_Dirigida_Sintaxe.pdf>. Acesso em: 31 jul. 2018.

Para encerrar, recomendamos que você reveja o vídeo:

PROJETO compilador com Jflex e Java Cup. Disponível em:

<https://www.youtube.com/watch?time_continue=92&v=GY0y288vRX4>. Acesso em: 31 jul. 2018.

Mas, agora que você já passou por todas as fases da análise de um compilador, indicamos o intervalo entre 8m02s e 24m19s.

Na próxima seção, resgataremos o conceito de tabela de símbolos, que está presente em todas as fases do compilador, e incorporaremos ao analisador léxico atributos para alguns *tokens*, bem como incluiremos funções no analisador sintático para tratar regras semânticas, tudo isso na prática, para fecharmos

com chave de ouro o processo de análise, e assim fixarmos todos os conceitos estudados até este momento. Vamos em frente para construir mais um pedacinho do compilador na próxima seção?

Sem medo de errar

Depois de ter atendido às solicitações iniciais dos examinadores do processo seletivo para uma das vagas para desenvolvedor no laboratório de pesquisa, você conquistou a oportunidade de continuar concorrendo a uma das vagas disponíveis. Agora é o momento de resgatar o trabalho realizado na Seção 3.1, para dar continuidade ao desenvolvimento na etapa seguinte: a análise semântica.

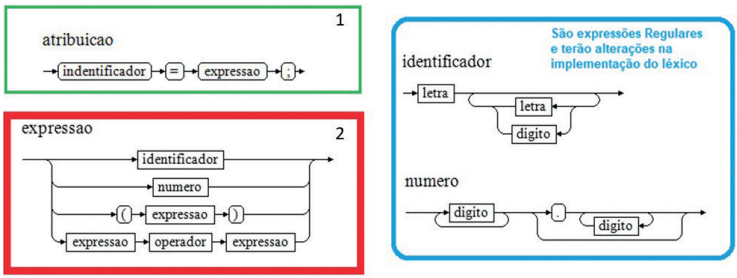
No início do processo seletivo, o entrevistador entregou o diagrama de sintaxe da linguagem e informou que você deveria analisá-lo e realizar as alterações necessárias para a definição de um bom projeto, o que você executou com mérito. Já nesta etapa, o que eles desejam é que conclua a etapa da análise sintática e: a) realize os ajustes necessários na gramática da linguagem para incorporar as definições para o tratamento da semântica; b) inclua na sua apresentação como pretende realizar o tratamento dos erros de semântica.

Lembrando que você terá apenas 30 minutos para a apresentação, e a forma da apresentação continuará livre. Para a exposição desta fase, os quesitos como clareza, uso de tecnologia, desenvoltura e domínio do assunto serão decisivos para obter a classificação para a segunda etapa. Como você já pôde perceber, projetos complexos requerem planejamento e organização, portanto, nada melhor que elencar cada passo para preparar a apresentação:

1. Com experiência adquirida na apresentação inicial e a percepção com relação aos quesitos que a comissão julgadora considera para diferenciar as habilidades entre os candidatos, sugerimos, primeiramente, o slide completo da correção apresentada na fase inicial e situar os ouvintes sobre o projeto, fornecendo assim uma visão geral do projeto e qual é o objetivo fundamental da apresentação.
2. A partir do diagrama de sintaxe corrigido na fase inicial, identifique os elementos que requerem atributos e regras

semânticas e apresente um slide mostrando isso, conforme a Figura 3.11:

Figura 3.11 | Elementos do diagrama de sintaxe relacionados com a análise semântica



Fonte: elaborado pela autora.

3. Depois de fazer a introdução do que será tratado especificamente nessa etapa, que tal indicar que será usada a tradução dirigida pela sintaxe para o tratamento dos erros semânticos? Para demonstrar pleno domínio sobre o assunto e os pontos mais críticos da análise semântica, é recomendável preparar alguns slides com um exemplo de programa.

   Um exemplo em JAVA seria bem interessante, o que você acha? Afinal, nesta seção, você tomou conhecimento que essa linguagem verifica a equivalência de tipos, tem um bom tratamento de erros semânticos, com mensagens claras e boa acuracidade. Você poderá usar o próprio o exemplo estudado, que permitirá bom domínio, isso lhe dará confiança na apresentação, pois, em um processo seletivo classificatório, há momentos para mostrar domínio e outros para inovar. Nos pontos em que são avaliados conhecimentos, é importante mostrar domínio.

   E, para mostrar seu senso crítico, não deixe de expor que a especificação da linguagem entregue para análise não faz declaração de tipo de variáveis e que isso implica em vinculação implícita, ou seja, somente poderá ser vinculado a um tipo quando ocorrer a atribuição de um valor a mesma. Aqui, também, você pode demonstrar conhecer que esse

tipo de declaração ocorre em interpretadores, por exemplo, linguagens como PHP.

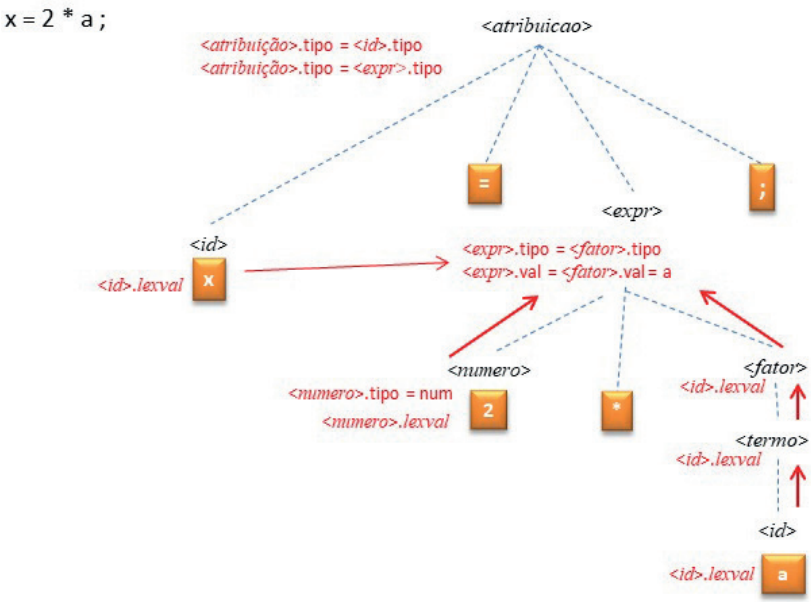
4. Depois de ter apresentado as considerações anteriores, mostre os comandos da gramática que precisarão das regras semânticas de atributos e ação.

Regras Sintáticas	Regras Semânticas
<*atribuição*> ::= <*id*> = <*expr*>	<*atribuição*>.tipo = <*id*>.tipo < *atribuição*>.val = <*expr*>.tipo ++ <*id*>. *lexval* || *"="* || <*expr*>.val
<*expr*> ::= <*fator*> <*expr1*>	<*expr1*>.tipo = <*fator*>.tipo <*expr1*>.val= <*fator*>.val
<*expr1*> ::= **+** <*fator*><*expr1*>	<*expr1*>.tipo = <*fator*>.tipo <*expr1*>.val = <*expr*>.val ++<*fator*>.val ++<*expr1*>.tipo || "=" || <*expr*>.tipo || "+" || <*fator*>.tipo
<*expr1*> ::= **-** <*fator*><*expr1*>	<*expr1*>.val = <*expr*>.val ++<*fator*>.val ++<*expr1*>.tipo || "=" || <*expr*>.tipo || "+" || <*fator*>.tipo
<*fator*> ::= <*termo*><*fat1*>	<*fator*>.tipo = <*termo*>.tipo <*fator*>.val = <*termo*>.val
<*fat1*> ::= ***** <*termo*><*fat1*>	<*fat1*>.tipo = <*termo*>.tipo <*fat1*>.val = <*expr*>.val ++<*termo*>.val ++<*fat1*>.tipo || "=" || <*expr*>.tipo || "*" || <*termo*>.tipo
<*fat1*> ::= **/** <*termo*><*fat1*>	<*fat1*>.val = <*expr*>.val ++<*termo*>.val ++<*fat1*>.tipo || "=" || <*expr*>.tipo || "/" || <*termo*>.tipo
<*termo*> ::= <*numero*>	<*termo*>.tipo = <*numero*>.tipo <*termo*>. *lexval* = <*numero*>. *lexval*
<*termo*> ::= <*id*>	<*termo*>.tipo = <*id*>.tipo <*termo*>. *lexval* = <*numero*>. *lexval*
<*termo*> ::= (<*expr*>)	<*termo*>.tipo = <*expr*>.tipo <*termo*>. *lexval* = <*expr*>. *Lexval*
Regras léxicas	
<*numero*> ::= <*digito*> {<*digito*>} [. <*digito*> {<*digito*>}]	<*numero*>.*lexval*
<*id*> ::= <*letra*> {<*letra*> | <*digito*> }	<*id*>.*lexval* = ***id***

5. Em seguida, defina as regras formais com a gramática de atributos. Para finalizar, dê o exemplo da árvore de derivação anotada, de acordo com as definições feitas. Por exemplo, para o comando x = 2 * a;.

Antes de apresentar a árvore, o que você acha de mostrar a árvore gramatical criada na primeira fase para análise de um programa completo. Isso mostrará que tem visão integrada do projeto. Não deixe de incluir as dependências.

Figura 3.12 | Árvore anotada para x = 2 * a; incluindo as dependências



Fonte: elaborada pela autora.

Seguindo o planejamento montado, você pode organizar sua apresentação com a certeza de que todos os quesitos solicitados foram atendidos, assim você fará uma ótima apresentação, com confiança e sabendo que o conhecimento adquirido nesta seção, em conjunto com as técnicas de tradução dirigida pela sintaxe, o domínio das notações definidas pela sintaxe e os esquemas de tradução foram fundamentais para elaborar a apresentação, bem como o domínio dos tipos de erros semânticos permitiram analisar o material entregue e identificar falhas e limitações.

Com todo esse talento e conhecimento, é bom já ir se preparando para dominar os princípios para implementação da tabela símbolos, tema da próxima seção, afinal, com a sua apresentação, você já está na próxima fase.

Avançando na prática

A gramática de atributos na prática

Descrição da situação-problema

Um grupo de alunos está interessado em conhecer e aplicar a gramática de atributos no projeto que estão desenvolvendo na pós-graduação, na área de engenharia elétrica. Como esse tema não está no currículo, resolveram procurar os alunos da graduação matriculados na disciplina de compiladores para que elaborassem um material explicativo para a linguagem que eles desenvolveram, mas falta a eles conhecimento técnico para a especificação das ações semânticas, e a tentativa de implementação de uma solução usando o JFLex não está dado certo. A linguagem para a qual eles precisam a especificação das ações semânticas é:

P := **programa** D **inicio** C **fim**.

D := **decl** id : T

T := **int** |**texto**

C := **id** = E | **se** L **entao** C | **leia(id)** | **escreva(id)**

E := **num** | **literal** | (E) | E + E

L := F OP F

F := **num** | **literal** | **id**

OP := **>** | **>** | **=** | **<>**

Ela é usada em um robô para cálculos. Ao incluírem a declaração de tipo, o analisador sintático não valida corretamente as dependências no comando. Você pode ajudá-los?

Resolução da situação-problema

Será uma ótima oportunidade para aplicar a tradução dirigida por sintática, elaborando a gramática de atributos para a linguagem

apresentada, além de estar mais próximo aos professores da pós e poder preparar o terreno para sua futura pós-graduação. Como os alunos já possuem uma noção, que tal começarmos por uma tabela, explicando os atributos que serão necessários para as definições das ações semânticas, antes de iniciar a resolução, assim será fácil entender a solução:

Produção	Atributo	Explicação
P	tc	*Sintetizado*. Booleano, É verdadeiro se não existem erros de tipos no programa.
D	tipo	*Sintetizado*. Conjunto de pares (identificador, tipo). Os tipos permitidos são: *inteiro*, *caractere*, *booleano* e *erro*.
T	tipo	*Sintetizado*. Um tipo, a ser usado em declarações (nunca será *erro*).
C	tc	*Sintetizado*. Booleano, que é verdadeiro se e somente se não existem erros de tipos no comando.
E	tipo	*Sintetizado*. O tipo da expressão (pode ser *erro*).
id	***lexval***	*Sintetizado*. O texto do identificador.
num	***lexval***	*Sintetizado*. O valor correspondente ao número.
literal	***lexval***	*Sintetizado*. Corresponde a cadeia de caracter.

A operação busca(**id.*lexval***) devolve o tipo associado ao identificador "**id**".

A produção L e E, quando E+E ou (E), terá atributos herdados, pois geram conjunto de pares.

De acordo com as explicações (detalhamento) apresentadas, a definição formal para as regras semânticas é a seguinte:

Produções	Regras semânticas
P := **programa** D **inicio** C fim.	P.tc = C.tc
D := **DECL** id : T	D.tipo = par(id.lexval, T.tipo)
T := **int**	T.tipo=inteiro
T := **texto**	T.tipo=caracter
C := **id =** E	Se C.busca(id.lexval) = E.tipo então C.tipo=E.tipo Senão **ERROR**

C ::= **se** L **entao** C	Se L.tipo = booleano então C0.tipo=C1.tipo
C ::= **leia**(id)	Se not C.busca(id.lexval) então **ERROR**
C ::= **escreva**(id)	Se not C.busca(id.lexval) então **ERROR**
E ::= **num**	E.tipo = inteiro
E ::= **literal**	E.tipo = caracter
E ::= **(**E**)**	E1 = E0.tipo; E2 = E0.tipo
E ::= E **+** E	E1 = E0.tipo; E2 = E0.tipo Se E1.tipo = inteiro e E2.tipo = inteiro então E0.tipo = inteiro Senao **ERROR**
L ::= F OP F	Se F1.tipo = inteiro e F2.tipo=inteiro então L.tipo=booleano Senao Se F1.tipo = caracter e F2.tipo=caracter então L.tipo=booleano Senao L.tipo = **ERROR**
F ::= **num**	F.tipo=inteiro
F ::= **literal**	F.tipo = caracter
F ::= **id**	Se C.busca(id.lexval) então F.tipo=id.tipo Senao **ERROR**
OP ::= **>**	
OP ::= **<**	
OP ::= **=**	

Faça valer a pena

1. Na tradução dirigida pela sintaxe, associamos atributos aos símbolos, sejam terminais ou não-terminais. Os atributos podem armazenar símbolos, que serão úteis no processo de reconhecimento das frases.
Sobre os atributos é correto afirmar:

a) Os atributos sintetizados são calculados em função dos nós superiores (pais).
b) Os atributos herdados, como o próprio nome diz, são calculados pelo

nó imediatamente superior (pai).

c) Uma definição dirigida pela sintaxe que utilizada exclusivamente atributos sintetizados é denominada definição S-atribuída.
d) Cada *token* sempre tem um atributo e é único.
e) A definição de atributos é feita somente na implementação do analisador sintático.

2. Dada a gramática:
$N := N\ D \mid D$
$D := 0 \mid 1$

Quais são as ações semânticas para esta gramática que gera números binários?

a) $N_1 := N_2\ D$ { $N_1.val = N_2.val$ + D.*val* };
$N := D$ { *N.val = D.val };*
$D := 0$ { D.*val* = 0 };
$D := 1$ { D.*Val* = 1 }
b) $N := N\ D \mid D$ { N.*val = N.val* D.*val* || D.*val* };
$D := 0 \mid 1$ { D.*val* = 0 || D.*Val* = 1 }
c) $N := N\ D \mid D$ { N.*val = N.val* D.*val* | D.*val* };
$D := 0 \mid 1$ { D.*val* = 0 | D.*val* = 1 }
d) $N := N\ D$ { N.*val = N.val +* D.*val* };
$D := 0$ { D.*val* = 0 };
$D := 1$ { D.*val* = 1 }
e) $N_1 := N_2\ D$ { $N_1.val = N_2.val$ * D.val };
$N := D$ { *N.val = D.val };*
$D := 0$ { D.val = 0 };
$D := 1$ { D.Val = 1 }

3. A tradução dirigida pela sintaxe está focada nas linguagens guiadas por gramáticas livres de contexto (LLC) e associa atributos a símbolos gramaticais definidos, e, a esses atributos, computamos os valores por meio de regras semânticas.
Como são formalmente feitas essas associações?

Assinale a alternativa correta:

a) Definindo atributos sintetizados e/ou herdados.
b) Infelizmente, não é possível fazer associações formais, pois não

podemos definir regra gramatical para situações como "Variável só pode ser usada se já existe".

c) Por meio de ações semânticas.
d) Pelo grafo de dependência.
e) Por meio das definições dirigidas pela sintaxe e esquemas de tradução.

Seção 3.3

Tabela de símbolos

Diálogo aberto

Caro aluno, lembrando que o desenvolvimento do compilador é dividido em duas grandes fases, a análise e a síntese, nesta seção iremos concluir a abordagem do tema sobre a primeira fase: a análise, também chamada de *front-end*.

Até aqui estudamos e conhecemos as diversas técnicas e ferramentas para a realização da análise léxica, sintática e semântica. Todas essas etapas usam a tabela de símbolos como auxiliar para desenvolver o *front-end* do compilador. Ela também é utilizada no *back-end* do compilador, a fase de síntese. Logo, devemos dar uma atenção especial à tabela de símbolo, pois é um elemento importante na estrutura de um compilador.

Portanto, o objetivo desta seção será conhecer como implementar a tabela de símbolos e integrá-la a todas essas etapas. Para isso, além dos conceitos teóricos sobre a tabela de símbolos, iremos desenvolver em Java um analisador completo, utilizando os geradores JFLEX e CUP, que possibilitarão a implementação da tabela de símbolos integrada com a análise léxica, sintática e semântica. Isso o ajudará no seu objetivo junto ao laboratório de pesquisa no qual você se candidatou para uma vaga e, também, em suas futuras atividades como desenvolvedor nesta área.

Dando continuidade ao processo seletivo para contratação dos novos desenvolvedores, o laboratório de tecnologia dedicado à pesquisa sobre linguagens de programação, na primeira fase, selecionou seis dos muitos candidatos participantes, e você conseguiu a melhor pontuação, avançando para a segunda fase. Agora, será o momento para repetir o seu excelente desempenho. A comissão julgadora propôs a cada candidato, nesta segunda e última fase, que elaborassem um exemplo de como implementar a tabela de símbolos. Vencerá a melhor proposta.

Assim como na primeira fase, alguns quesitos mínimos deverão ser atendidos e serão avaliados. Você deverá: a) elaborar uma explanação

sobre a importância da tabela de símbolos para o processo de construção de um compilador b) especificar quais estruturas de dados são usadas na implementação da tabela de símbolos; e c) apresentar um exemplo desenvolvido por você, principalmente.

A metodologia para a apresentação do trabalho será a mesma das etapas anteriores: a forma da apresentação é livre para cada participante e a duração não deve ultrapassar a 45 minutos ou ser inferior a 30. A clareza da apresentação, desenvoltura e domínio do assunto continuam valendo nesta etapa.

Qual será a melhor estrutura para implementar a tabela de símbolos: vetor ou lista encadeada? Qual técnica usar para realizar as pesquisar na tabela? E, ainda, o que o léxico insere na tabela se símbolos? E o sintático e o semântico? Inserem dados ou apenas consultam?

Para responder a essas perguntas e fundamentar os conceitos que serão utilizados para a implementação da tabela de símbolos, esta seção se inicia mostrando quais estruturas são utilizadas para criação da tabela de símbolos. Em seguida, revisa alguns conceitos de estruturas de dados utilizados pela tabela de símbolos e, para finalizar, mostra um exemplo de desenvolvimento da tabela de símbolos em Java, usando os geradores Jflex e CUP.

Esses conhecimentos permitirão que, além de apresentar uma proposta de implementação de tabela de símbolos conceitual, você construa um analisador sintático completo, com a análise léxica sintática e semântica, assim, poderá mostrar pleno domínio sobre os conceitos de análise e obter um diferencial entre os candidatos, o que o qualificará para a vaga. Pronto para iniciar esta próxima etapa?

Não pode faltar

Nesta seção, iremos aprofundar o conhecimento sobre o JFlex, e integrar o analisador léxico com o analisador sintático usando as ferramentas JFflex e CUP. O JFlex é o gerador de analisadores léxicos, já estudado na Seção 2.2, e, agora, iremos avançar, integrando o léxico com o sintático, utilizando a ferramenta CUP. Com o uso integrado do JFlex e do CUP, será possível implementar ações semânticas e a tabela de símbolos. Contudo, antes de usarmos as facilidades dos geradores JFlex e CUP, vamos conhecer a estrutura da tabela de símbolos (TS) .

A TS começa a ser preenchida no analisador léxico quando, ao reconhecer o *token*, o léxico faz a inserção do par (*token,* lexema). Outros atributos do *token* somente poderão ser identificados na análise sintática, quel irá consultar a TS para acrescentar atributos, alterar ou mesmo incluir novos elementos na TS.

Com relação à estrutura de dados da TS, utilizamos listas ligadas, pois, segundo Aho, Sethi e Ullman (2007), o uso de tabela *hash* será fundamental na construção do compilador, já que:

a) **O número de elementos a serem inseridos na tabela é grande**, por exemplo, um programa, não muito longo, tem milhares de *tokens*.

b) **Os elementos não possuem a mesma estrutura**. Por exemplo, há tipos terminais, como "int", que são compostos apenas pelo par (token, lexema), outros, como variável, que possuem mais atributos (ID, lexema, tipo, valor). Há, ainda, outros tipos de variáveis com mais atributos, como limite, no caso de vetores, ou quantidade de parâmetro, no caso de funções. Todas estas características são definidas nas ações semânticas.

c) **A quantidade de operações de busca, inclusão, alteração e exclusão** realizada na TS é enorme, e isso requer resposta rápida, caso contrário o processo de compilação se torna excessivamente lento.

Pesquise mais

Você pode perceber que o domínio sobre os conceitos de estrutura de dados é fundamental para a compreensão dos recursos necessários para implementar a tabela de símbolos. Portanto, pesquise mais sobre listas encadeadas, fila, árvore binária e tabela *hash*,.

CAELUM. Apostila do curso CS-14 – Algoritmos e Estruturas de Dados com Java. [S.l.; s.d.]. Disponível em: <https://www.caelum.com.br/apostila-java-estrutura-dados/>. Acesso em: 31 jul. 2018.

Para saber mais sobre tabela *hash* (tabela de espalhamento), consulte o capítulo 9, para pilha, consulte o capítulo 6. Caso você julgue necessário rever os conceitos elementares, poderá iniciar com o capitulo 5, sobre listas ligadas.

Como exposto, a TS é acessada em todas as etapas do processo de compilação, seu tamanho não é fixo, apresenta um grande número de elementos (quanto maior o programa, maior a tabela) e muitos acessos de buscas são realizados durante as análises léxica, sintática e semântica.

Portanto, implementar a TS usando uma tabela *hash* dará ao processo de compilação melhor eficiência, pois sua complexidade, no pior caso, é $\sigma(n)$, e na média, $\sigma(n/m)$, onde n é o número de entradas e m a quantidade de elementos da tabela, sendo assim um método de busca muito eficiente. Afinal, o princípio da tabela *hash* é a busca direta $\sigma(1)$, e, segundo Aho, Sethi e Ullman (2007), o tempo para ter acesso a uma entrada da tabela é essencialmente constante. Vale que citar Sedgewick (2011) dedica em seu livro *Algorithms* um capítulo inteiro à TS, sendo uma seção exclusiva sobre tabela *hash*, dada a sua importância e suas aplicações. Segundo Sedgewick (2011), podemos definir tabela de símbolos (TS) como uma estrutura de dados para o **par chave-valor** que suporta inserção e busca de valores, associado a uma dada chave. Para implementar a TS por meio de uma tabela *hash*, é preciso definir dois elementos: a função *hash* e o tratamento de colisão.

Segundo Sedgewick (2011), a função *hash* que transforma a chave de busca em um código que permite a busca direta, como o índice de um vetor, é um ideal longe de ser alcançado. Portanto, muitas vezes, a função *hash* gera um mesmo código para representar diferentes chaves, conflito em que se diz que ocorreu uma colisão, que precisará ser tratada, pois a função de *hash* deve sempre retornar um mesmo código *hash* para a mesma chave.

O estudo dos tipos de funções *hash* e os métodos para tratamento de colisões são objeto de estudo da disciplina de estrutura de dados, entretanto, ter em mente o conceito apresentado é fundamental para entender as estruturas disponíveis no JAVA para implementar tabelas *hash*, que iremos utilizar para desenvolver a TS na fase de análise do compilador.

Reflita

A função *hash* muitas vezes gera códigos iguais para uma mesma chave e, com isso, ocorrem conflitos. Por que tabelas *hash* são mais rápidas e tão úteis se é preciso tratar estes conflitos?

Responda rápido: o pronto-socorro (PS), está lotado, como organizar a sala de esperas e fornecer aos médicos onde se encontra o paciente que deve atender diretamente?

Que tal criar filas por especialidades e entregar a cada paciente uma ficha com perguntas específicas de cada especialidade. Desta forma, organiza-se melhor a fila e o gerenciamento do espaço, restando apenas associar um número para permitir ao médico localizar rapidamente o paciente e seus dados.

A tabela *hash* usa basicamente essa técnica de espalhamento dos dados, separando os pacientes em várias filas e cada objeto (paciente) será identificado por um número (chave) que nos leva exatamente à localização do paciente e a seus dados. A chave única é determinada por uma função *hash*.

A função *hash* permite o espalhamento dos dados, e quanto melhor a função, menos conflitos haverá. No exemplo do os, tratar o conflito é encontrar a vaga disponível no setor específico.

Em uma TS, durante o processo de compilação, qual "fila" será mais longa é aleatório e as subdivisões dependerão da complexidade de cada produção gramatical, sendo umas mais complexas, outras mais simples. Assim, o uso da estrutura *hash* torna ágil a recuperação dos dados para análise, e faz o espalhamento dos dados pelas estruturas de listas ligadas.

Encontramos no JAVA a API Collections java.util.concurrent, que disponibiliza a classe ConcurrentHasMap, e na java.util temos as classes HashMap e Hashtable. Essas três classes fornecem suporte para tabelas *hash*, ou seja, os dois elementos, função *hash* e tratamento de colisões, além dos métodos para inserção, busca e exclusão na tabela, estão presentes na estrutura de dados implementada por essas classes.

Qual das classes usar? O recomendável é ConcurrentHashMap, por se tratar de uma versão melhorada da Hashtable. A diferença essencial entre as classes Hashtable e HashMap é a sincronização. Hashtable e ConcurrentHashMap são sincronizadas, enquanto a HashMap não, o que implica que as duas primeiras suportam manipulação por várias *threads*, logo, devemos utilizá-las no caso de programação concorrente. Entretanto, iremos estudar e utilizar neste material a classe Hashtable, devido ao fato da ferramenta CUP utilizá-la.

Pesquise mais

Para saber mais sobre programação concorrente e a classe ConcurrentHashMap, consulte o artigo a seguir:

Ricardo, L. **Concorrência e objetos thread-safe**. 5 mar. 2014. Disponível em: <http://luizricardo.org/2014/03/concorrencia-e-objetos-thread-safe/>. Acesso em: 31 jul. 2018.

A seguir, apresentamos um exemplo para explicar os métodos suportados pela classe Hashtable e que interessam para a manipulação da TS, no nosso caso:

```
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Set;

public class HashTableExemplo {
   public static void main(String args[]) {
      Hashtable TS = new Hashtable(); //  TS é uma instancia da uma tabela hashtable do JAVA

      TS.put("1", "x");  // INSERE um novo elemento em TS.put(chave,valor)
      TS.put("2", "ra");

      TS.get("1");    // .get(chave) - método para busca direta na hashtable
     // exemplo do uso do get para mostrar  o valor a partir da chave
      System.out.println(TS.get("1").toString());

       // método .containsKey(chave) para verificar se a CHAVE existe
     //retorna true se encontrar ou false se não achar
       System.out.println("Encontrou a chave 1: "+ TS.containsKey("1"));

      // método .containsValue(valor) para o VALOR existe
```

```
        //retorna true se encontrar ou false se não
achar
        System.out.println("Busca pelo valor --> En-
controu o lexema 'ra' : "+
                           TS.containsValue("ra"));

      // metodo .isEmpty() - Verifica se a Hashtable
está vazia - retorna true ou false
       System.out.println("HashTable está vazia ? "+
TS.isEmpty());

      // metodo .size() - Retorna o tamanho da Hash-
Table
       System.out.println("Tamanho da HashTable: " +
TS.size());

     // é possivel pegar com um só comando todas as
chaves da HashTable
       // o objeto Set é uma lista ligada que não
aceita chave duplicada
     Set Chaves = TS.keySet();

      // ou pode criar um Enumeration e  atribuir
todas a chaves com o uso do metodo .key()
     Enumeration TabelaEnum = TS.keys();

     // para "esvaziar" a Hashtable
     TS.clear();
  }
}
```

Agora que já conhecemos qual estrutura é usada para implementar uma TS e como ela funciona, vamos avançar e conhecer a ferramenta CUP, para gerar analisadores sintáticos de forma integrada com o JFlex. Segundo Petter (2014), a sigla CUP significa ***C****onstructor of* ***U****seful* ***P****arsers*, a tradução literal é construtor de analisadores úteis, mas o termo *useful,* aqui, tem muito mais haver com a ideia de integração, afinal, o CUP integrado ao JFlex permitirá a implementação completa da fase de análise de um compilador e facilitará o desenvolvimento também da fase de síntese no futuro.

O CUP implementa a maioria das facilidades de outro gerador de analisadores sintáticos, o YACC, e com poucas variações na sua sintaxe, sendo que o YACC surgiu primeiro e gera o código na linguagem C e o CUP foi criado em JAVA e gera o código também em JAVA.

O código do analisador sintático gerado pelo CUP implementa um analisador LALR (**L**ook-**A**head **L**eft-to-**R**ight), ou seja, é um analisador ascendente (*bottom-up)*, analisando as produções de baixo para cima (das folhas para a raiz) com análise preditiva (análise de símbolos a frente). Como o JFlex, o CUP também possui algumas regras para a montagem do arquivo de especificação da gramática. A estrutura do arquivo de especificação está dividida em quadro partes, na ordem a seguir:

- Diretivas do CUP e código Java que se deseja incluir na classe que será gerada;
- Definição dos símbolos terminais e não-terminais;
- Definição das precedências;
- Definição das produções gramaticais.

Exemplificando

Vamos mostrar um arquivo de especificação para o CUP.

```
//(1)declaração dos código que deseja que o CUP insira
no analisador
package ExemploCUP;
import java_cup.runtime.*;
import java.util.*;
import java.io.*;

// parser code {: codigo q deseja que o CUP insira na
classe Parser  :}
parser code {:
           public void report_error(String message,
Object info)  {
           System.out.println("AVISO - " + message);
           System.out.println(info.toString());
        }
```

```
        public void report_fatal_error(String message,
Object info)   {
            System.out.println("ERRO - " + message);
            System.exit(-1);
         }
:};

//(2.1)definição dos símbolos terminais
terminal PROGRAMA, INICIO, FIM, TIPO_INTEIRO, TIPO_
CHAR;
terminal PTVG, SIMBOLO_ATRIB, CHAR, INTEIRO, VARIAVEL;

//(2.2)definição dos símbolos não-terminais
non terminal program, comandos, comando, decl_atrib,
expr;
non terminal decl_variavel, tipo_dado;

//(4.1)definição de qual será a produção inicial
start with program;

//(4.2)definição das produções
program  ::= PROGRAMA INICIO comandos FIM;
comandos ::= comando comandos | comando;
comando  ::= decl_atrib | decl_variavel;
decl_atrib ::= VARIAVEL SIMBOLO_ATRIB expr PTVG;
expr ::= VARIAVEL | INTEIRO | CHAR;
decl_variavel ::= tipo_dado VARIAVEL PTVG;
```

No exemplo, você pode observar três partes da estrutura do arquivo para especificação da gramática no CUP. Leia os comentários (1), (2.1), (2.2), (4.1) e (4.2) do exemplo apresentado.

Assista ao video com a demonstração desse exemplo do CUP integrado com o Jflex, no link https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/compiladores/u3/s3/VIDEO_1.mp4 ou por meio do QR Code.

Sendo o CUP um gerador de analisador sintático, ele facilita a construção do analisador, pois, ao criarmos o arquivo de especificação, a classe **Parser** da API do java_cup.runtime implementa a TS usando uma **HashTable,** assim o projetista do compilador precisará se preocupar apenas com a definição dos símbolos terminais, as produções, as ações semânticas e o tratamento para recuperação de erros.

E, para integrar o analisador léxico gerado pelo JFlex com o analisador sintático gerado pelo CUP, é necessário incluir uma diretiva no arquivo do JFlex para compatibilizar com CUP. Para isso, precisaremos incluir **%cup** na primeira seção do arquivo de especificação do JFlex e adicionar o nome dos *tokens* na TS criada pela ferramenta CUP, que, neste caso, são as classe Symbol e Sym.

A seguir, apresentamos um exemplo de especificação do léxico, de acordo com o JFlex e integrado ao CUP. Acompanhe atentamente os comentários.

```
//declaração dos código que deseja que o JFLEX
insira no analisador
package ExemploCUP;
import java_cup.runtime.*;  // este é a biblioteca
do CUP

%%
%cup              // O NOVO paramêtro para
compatibilização com o CUP
%public
%class Lexer
%line
%column
// O OBJETIVO AGORA É passar o token identificado
para o SINTATICO
// portanto iremos inserir o token na TS (tabela
de símbolos)
// Symbol --> é a classe que recebe os dados do
léxico
//           o exemplo a seguir tratamos dois
tipos de token
%{
```

```
  // inclusão de um token terminal sem qq atributo
além do nome
  private Symbol symbol(int type) { return new
Symbol(type, yyline, yycolumn); }

  // já neste caso incluímos o valor do token ,
parâmetro value
  private Symbol symbol(int type, Object value) {
             return new Symbol(type, yyline,
yycolumn, value);  }
%}
// declaração das expressões regulares - nada muda
DIGITO = [0-9]
LETRA = [a-zA-Z_]
LITERAL =  \"[^\"]*\"
INTEIRO = {DIGITO}+
VARIAVEL = {LETRA}+
IGNORE = [\n|\s|\t\r]
%%

// diretiva de estado <YYINITIAL>
<YYINITIAL> {
// símbolos terminais  { AÇÃO }
//                       INCLUI o token PROGRAMA
na TS
    "programa"      {return new Symbol(Sym.
PROGRAMA); }
//                              INCLUI o token
INICIO e assim sucessivamente nos demais comandos
    "inicio"        {return new Symbol(Sym.
INICIO); }
    "fim"           {return new Symbol(Sym.FIM); }
    "int"           {return new Symbol(Sym.TIPO_
INTEIRO); }
    "caracter"      {return new Symbol(Sym.TIPO_
CHAR); }
    ";"             {return new Symbol(Sym.PTVG);
}
    "<-"            {return new Symbol(Sym.
```

```
SIMBOLO_ATRIB); }
    {LITERAL}        {return new Symbol(Sym.CHAR);
}
    {INTEIRO}        {return new Symbol(Sym.
INTEIRO); }
    {VARIAVEL}       {return new Symbol(Sym.
VARIAVEL); }
    {IGNORE}         {}
}
<<EOF>> { return new Symbol( Sym.EOF ); }
[^] { throw new Error("caracter inválido:
"+yytext()+" na linha "+(yyline+1)+", coluna
"+(yycolumn+1) ); }
```

Assimile

Para desenvolver a aplicação do analisador sintático são necessários:

1. O kit JDK 8.0 ou superior para o JAVA SE, disponível no site <http://www.oracle.com/technetwork/java/javase/downloads/jdk10-downloads-4416644.html> (acesso em: 1 ago. 2018).
2. A IDE NetBeans 8.1, disponível no site <https://netbeans.org/community/releases/81/index_pt_BR.html> (acesso em: 1 ago. 2018).
3. O gerador JFlex 1.6.1, disponível no site <http://jflex.de/download.html> (acesso em: 1 ago. 2018).

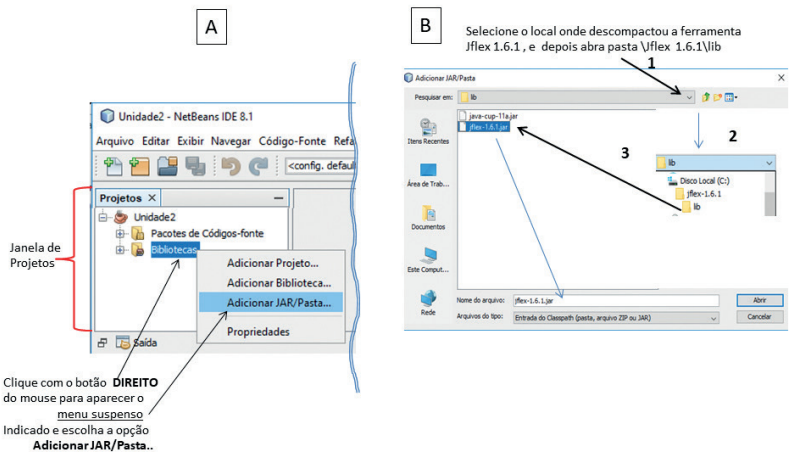
Você poderá visualizar o processo de instalação por meio de um tutorial, em vídeo, no link https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/compiladores/u2/s2/U2S2_video_1.mp4 ou por meio do QR Code

Você pode revisar a Seção 2.2 para maiores detalhes sobre o JFlex.

Portanto, para criar um analisador completo, você precisará adicionar *os plug-ins* JFlex 1.6.1 e java-cup-11a ao seu projeto. Assim, depois de ter baixado o JFLEX 1.6.1.zip, faça a descompactação e abra o Netbeans. Localize o seu projeto, adicione o *plug-in* JFlex ao projeto clicando com o botão direito do mouse na **pasta biblioteca**

e, em seguida, com botão esquerdo, na opção **adicionar JAR/Pasta.** Esses passos estão indicados na Figura 3.13 (B), que representa o momento em que você irá apontar para o arquivo 'jflex1.6.1.jar', que foi baixado e descompactado, conforme orientado anteriormente. Repita a operação, agora para adicionar o arquivo java-cup-11a.jar. Para concluir, certifique que o 'jflex1.6.1.jar' e o 'java-cup-11a.jar' aparecem na caixa de texto Nome do arquivo, conforme indicado na Figura 3.13 (B), e clique no botão Abrir. Os *plug-ins* serão adicionados ao seu projeto e poderão ser utilizados.

Figura 3.13 | Adicionando o JFlex a um projeto Java



Fonte: captura de tal da Netbeans 8.1, elaborada pela autora.

Para encerramos esta unidade, nada melhor que você experimentar o exemplo apresentado.

O projeto completo do exemplo encontra-se disponível para download no link https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/compiladores/u3/s3/codigo_1.zip ou por meio do QR Code

Recomendamos que faça o download do projeto, descompacte e abra no Netbeans 8.1. A estrutura do projeto, uma estrutura padrao de um analisador, é mostrada na Figura 3.14. Na parte inferior direita da imagem, pode ser vista a árvore do projeto no Netbeans. Antes de

testar o analisador, vamos comentar sobre a sintaxe da linguagem do exemplo apresentado no Quadro 3.1

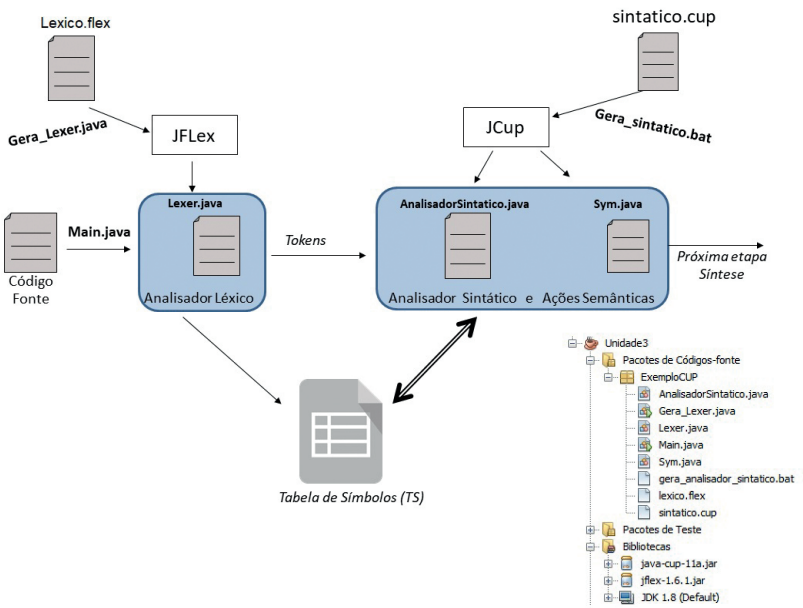
Quadro 3.1 | Exemplo comentado de código fonte

Exemplo do código fonte programa.prg	Comentários

programa
  inicio
  int VALOR;
  caracter NOME;
  VALOR = 10;
  NOME <- "Compiladores";
  fim
	Programa, inicio, int, caracter e fim são simbolos terminais
	; e <- tambem são simbolos terminais
	É uma declaracao de variavel
	
	Aqui há um erro de sintaxe o correto é <- e nao o igual '='
	É um comando de atribuição

Fonte: elaborado pela autora.

Figura 3.14 | Estrutura de um analisador sintático desenvolvido usando Jflex & Cup



Fonte: elaborada pela autora.

A classe Main desenvolvida para ler o arquivo fonte e fazer a análise é diferente da classe Main desenvolvida para o analisador léxico, feita anteriormente apenas pela linha:

AnalisadorSintatico p = new AnalisadorSintatico(new **Lexer(new FileReader(fileName)**));

Observe que a classe Lexer é parâmetro da classe AnalisadorSintatico, assim fica claro que o analisador sintático recebe o fluxo de token e os analisa de acordo com a gramática livre de contexto definida por meio do método parser. Portanto, com o uso das ferramentas Jflex e Cup foi possível desenvolver um analisador sintático completo, criando tabela de símbolo para inclusão e busca de dados, com estruturas *hash*.

Encerramos, assim, a fase de análise de um compilador, conhecendo a estrutura da tabela de símbolos, sua aplicação, uso e implementação, além de revisar conceitos de *hash*. Agora você está pronto para avançar para a segunda fase do processo de compilação, a SÍNTESE. Vamos conhecer como funciona esta etapa?

Reflita

Manter-se atualizado sobre novas tendências do mercado poderá ser útil a você, em especial sobre IoT (**I**nternet-**o**f-**T**hings), ou seja a Internet das Coisas, que é uma realidade que está ai para ficar, bem como as indústrias 4.0.

Um bom profissional é aquele que está preparado para unir competência (conhecimento técnico) e habilidades, isto é, aplicar os conhecimentos técnicos às situações do dia a dia da empresa.

Que tal conhecer um pouco mais sobre este assunto assistindo o vídeo:

MARCELO BARBOZA. **Como a indústria 4.0 vai mudar a automação?** 13 abr. 2018. Disponível em: <https://www.youtube.com/watch?time_continue=396&v=5hQUQtDKLSY>. Acesso em: 1 ago. 2018.

Sem medo de errar

Está chegando o momento da última etapa do processo seletivo para a contratação dos novos desenvolvedores em um laboratório de tecnologia dedicado à pesquisa sobre linguagens de programação. A comissão julgadora propôs a cada candidato nesta segunda e última fase que elaborasse um exemplo de como implementar a tabela de símbolos. Vencerá a melhor proposta.

Como na primeira fase, alguns quesitos mínimos deverão ser atendidos e serão avaliados. Você deverá: a) elaborar uma explanação sobre a importância da tabela de símbolos para o processo de construção de um compilador b) especificar quais estruturas de dados são usadas na implementação da tabela de símbolos; e c) apresentar um exemplo desenvolvido por você, principalmente.

Depois dos conhecimentos adquiridos ao longo desta seção, fica fácil responder qual a melhor estrutura para implementar a tabela de símbolo: tabelas *hash*. Como a comissão espera inovação, você pode inovar fazendo sua apresentação usando o PREZI, se desejar. Aqui vão algumas dicas sobre como fazer isso:

1. Como usar o PREZI: acessando o link https://prezi.com/business/workshop/ (acesso em: 1 ago. 2018), você encontrará vários treinamentos sobre como criar a sua apresentação com PREZI.
2. Pode, também, encontrar dicas importantes para criar uma apresentação interativa em https://prezi.com/the-science/?gclid=EAIaIQobChMIsO6Ntob_2wIViQ2RCh0ltQTZEAAYASACEgLNh_D_BwE (acesso em: 1 ago. 2018).

Lembre-se que não basta apresentar os conceitos de *hash*, sendo importante dar exemplos. Assim, depois da introdução do conceito de *hash* com uma bela apresentação, mostrando que você está atualizado e não usa apenas PowePoint, é o momento de apresentar um exemplo, mas, como o concurso busca que o candidato conheça TS e a área de trabalho é desenvolvimento de novas linguagens de programação, que tal elaborar seu exemplo com a aplicação do JFlex e Cup utilizando TS e semântica? Com isso, você mostrará que tem um bom domínio na área em que irá atuar, além do conhecimento básico que estão solicitando. Vamos lá:

1. Na primeira fase, a comissão solicitou a correção e análise de uma gramática que continha um comando de atribuição, assim, utilizar essa estrutura sintática é uma ótima ideia, pois permitirá você implantar um TS completa, inclusive com a parte da semântica, sem ultrapassar o tempo para a apresentação.
2. Você pode mostrar o diagrama de sintaxe e a gramática dessa produção, depois de ter mostrado os conceitos da tabela *hash* e informar que pretende fazer a implementação prática com esse exemplo.
3. Mostrar que fará o cálculo com os valores para a expressão, como uma calculadora para mostrar todas as funcionalidades;
4. Construir a especificação do CUP:

```
package Calculadora;
import java_cup.runtime.*;
parser code {:
      //codigo criado para reportar erro sintático
      public void report _ error(String message,
Object info)  {
           System.out.println("AVISO: - " +
message); }

      public void report _ fatal _ error(String
message, Object info)  {
           System.out.println("ERRO - " + message);
           System.exit(-1);      }
:}
terminal PTVG, MAIS, MENOS, VEZES, DIVIDE, ABRE _
PAREN, FECHA _ PAREN;
//aqui estamos informando que o terminal NUMERO
terá um valor do tipo inteiro
terminal Integer NUMERO;

non terminal            expr _ list;
non terminal Integer     expr;   // a produção expr
terá um valor do tipo inteiro

//por se tratar da analise de um calculo matemático,
foi definida as precedencias
```

```
// para as operacoes
precedence left MAIS, MENOS;
precedence left VEZES, DIVIDE;

start with expr_list;  // indica qual a produção inicial
                                        // acoes para mostrar o resultado do calculo
expr_list ::= expr_list expr:e PTVG        {: System.out.println(">> " + e); :}
            | expr:e PTVG                  {: System.out.println(">> " + e); :}
;
// aqui temos as acoes semantica  RESULT armazena um resultado
expr  ::= expr:e1 MAIS  expr:e2   {: RESULT = e1+e2;   :}
        | expr:e1 MENOS expr:e2   {: RESULT = e1-e2;   :}
        | expr:e1 VEZES expr:e2   {: RESULT = e1*e2;   :}
        | expr:e1 DIVIDE expr:e2  {: RESULT = e1/e2;   :}
        | ABRE_PAREN expr:e       {: RESULT = e;       :}
FECHA_PAREN
        | NUMERO:n                {: RESULT = n;       :}
;
```

5. Construir a especificação do JFlex:

```
package Calculadora;
import java_cup.runtime.*;
%%
%class LexerCalc
%cup
%line
%column
%{
  private Symbol symbol(int type) { return new Symbol(type, yyline, yycolumn); }
   private Symbol symbol(int type, Object value) { return new Symbol(type, yyline, yycolumn, value); }
%}
```

```
EspacoEmBranco = [\t\n\r ]+
numero         = [0-9]+
%%
<YYINITIAL> {
   {numero} { return symbol(Sym.NUMERO, new Inte-
ger(Integer.parseInt(yytext()))); }
    "+"             { return symbol(Sym.MAIS); }
    "-"             { return symbol(Sym.MENOS); }
    "*"             { return symbol(Sym.VEZES); }
    "/"             { return symbol(Sym.DIVIDE); }
    "("           { return symbol(Sym.ABRE_PAREN); }
     ")"              { return symbol(Sym.FECHA_PA-
REN); }
    ";"             { return symbol(Sym.PTVG); }
   {EspacoEmBranco} {}
}
<<EOF>>             { return symbol(Sym.EOF); }
//mensagem de erro caso ocorre erro léxico
[^] { throw new Error("caracter INVÁLIDO <" + yy-
text() + ">");}
```

6. Construir o projeto e testar.

   Exemplo para o arquivo de entrada Teste.calc

```
(2 * 1) * 9;
1 + 3;
10 / 3 ;
(20 + 30 ) - ( 5 * ( 10 + 2 ) );
( 1 + 3 + 5) * 8;
8+2;
```

7. Encerrar a apresentação, colocando-se à disposição para questionamentos.

Conclua essa etapa entregando os arquivos fontes do projeto e da apresentação, e assim, você estará orgulhoso de seu desenvolvimento e confiante de que uma das vagas será sua. Parabéns!

Avançando na prática

Conceitos de tabela de símbolos aplicada a outras áreas de conhecimentos, além de compiladores

Descrição da situação-problema

Uma empresa de logística passou a utilizar um robô para inserir fisicamente e buscar produtos no estoque, e sua equipe de TI se deparou com um problema decorrente da necessidade dos dados estarem embarcados no equipamento: incompatibilidade com o acesso ao BD utilizado e a memória do robô é insuficiente para carga do BD.

Eles até pensaram em uma solução, que seria apenas carregar o cadastro dos produtos que estão presentes no estoque para minimizar a carga, mas, mesmo assim, as buscas ficariam bastante lentas, pois cada produto possui campos diferentes, o que resultaria em muitas tabelas. Será que a empresa não conseguirá avançar na implantação do processo de inovação para a indústria 4.0?

Resolução da situação-problema

Claro que a empresa poderá avançar tecnologicamente para o processo 4.0. Como um profissional "antenado" da área de TI, você sempre está buscando se atualizar em relação às novas tecnologias e às mudanças de paradigmas no mercado de negócios, e você já tinha tomado conhecimento sobre a revolução industrial 4.0 e IoT, pois um bom profissional se mantém atualizado e sabe quando e como aplicar seus conhecimentos técnicos nas situações críticas do dia a dia da empresa.

Portanto, você já vinha vem se preparando para o processo de inovação. Seu conhecimento sobre indústria 4.0, tabela *hash* e sua aplicação no desenvolvimento de compiladores permitirá a você propor o uso dessa técnica para implantar uma TS para o dispositivo que a empresa adquiriu, o robô.

Assim, basta a equipe fazer a carga apenas do estoque atual no galpão, implementar as classe para cada tipo de produto e uma classe para a manutenção (inclusa, recuperação e exclusão). Para dar suporte a equipe de desenvolvimento, você preparou o exemplo a seguir:

```
public class TesteTS {
    public static void main(String[] args){
      // simulação(exemplo) de entrada de produto 3
tipos diferentes
         String[] ordem= {"api", "1515", "ref-EUA-
-ANVISA"};
        RacaoAnimal objA = new RacaoAnimal(122050,
50, 124512, ordem);

      DefensivoAgr objB = new DefensivoAgr(10,5,80);
      // cria a TS
        TS estoque = new TS();
      // insere os dados
        estoque.insere(1,objA);
        estoque.insere(2,objB);
        estoque.insere(3,1010);

     //verifica se chave existe (produto) - se existe
imprimi o valor
        if (estoque.busca(3))
           System.out.println(estoque.recupera(3).
toString());

      //recupera dados do produto a partir da chave
        DefensivoAgr objAux ;
        if (estoque.busca(2)){
             objAux = (DefensivoAgr) estoque.recu-
pera(2);
            System.out.println(objAux.toString());
        } else
          System.out.println("Chave do produto não
encontrado!");
       //recupera dados do produto a partir da chave
        DefensivoAgr obj1 = null;
        RacaoAnimal obj2= null;
        int obj3;
        int x=0;
        Object objX;
        int chave = 1;
        // usa conceito de poliformismo para apre-
sentar os dado
        if (estoque.busca(chave)){
            objX =  estoque.recupera(chave);
            if (objX instanceof DefensivoAgr){
                obj1 =  (DefensivoAgr) objX;
                x=1;
            }
```

```
            else if (objX instanceof RacaoAnimal){
                obj2 =  (RacaoAnimal) objX;
                x=2;
            } else {
                 obj3 =  (int) objX;
                 x=3;
            }
            switch (x) {
                case 1:
                       System.out.println(obj1.toS-
tring());
                        break;
                case 2:
                       System.out.println(obj2.toS-
tring());
                        break;
                case 3:
                        System.out.println(estoque.
recuperaValor(chave));
                        break;
            }

        } else
           System.out.println("Chave do produto não
encontrado!");
    }
}
```

O projeto completo do exemplo, incluindo todas as classes, encontra-se disponível para download no link https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/compiladores/u3/s3/codigo_2.zip ou por meio do QR Code

Pronto, problema resolvido com base em conhecimentos de estrutura de dados e vivencia com o desenvolvimento de compiladores.

Faça valer a pena

1. A análise sintática é o coração da fase de análise e, ao desenvolver o compilador, é nesta fase que integramos todas as três etapas: léxica, sintática e semântica. Com a árvore sintática "decorada", isto é, feita a análise semântica, a tradução do código está pronta para ser iniciada. Em todas essas etapas, a tabela de símbolos está presente.

Assinale a afirmação correta:

a) O número de elementos normalmente inseridos na tabela de símbolos é constante.
b) Os elementos inseridos na tabela de símbolos são do mesmo tipo, logo, um *array* atende muito bem a estrutura de uma tabela de símbolos.
c) A quantidade de operações de inclusão é sempre maior que a de busca na tabela de símbolos.
d) As operações de busca e inclusão na tabela de símbolo requerem rapidez, caso contrário, o processo de compilação será moroso.
e) Como o processo de compilação não é uma operação crítica, pois o aplicativo ainda não está em operação, rapidez nas buscas de dados nos processos internos não é fundamental.

2. O JFlex e o Cup são geradores de analisadores léxicos e sintáticos, respectivamente. Essas ferramentas geram o código em Java. As regras para a especificação da gramática livre de contexto no Cup estão divididas em 3 partes. Dado do trecho a seguir de um arquivo de especificação no CUP:

```
consulta ::= SELECT campos FROM tabelas PTVG
campos ::= campos campo | campo
campo ::= ID VG | ID
```

Para completamos a especificação, tendo por base o trecho apresentado, quais linhas estão corretas?

Assinale a alternativa correta:

a) terminais SELECT, FROM, PTVG, ID, VG
b) non terminais campos, tabelas, id;
c) terminal campos, tabelas, campo, tabela;
d) non terminal SELECT, FROM, PTVG, ID, VG
e) terminal campos, campo, tabelas;

3. O arquivo de especificação do Cup possui 4 divisões, que devem ser definidas nesta ordem:

a) Diretivas do Cup e código do usuário a ser incluído;
b) Definição dos símbolos terminais e não-terminais;
c) Definição das precedências;
d) Definição das regras de produção da gramática livre de contexto.

Com relação à função e sintaxe das diretivas de comando para as definições do Cup, é correto afirmar:

a) %cup é uma diretiva para compatibilizar o Cup com o JFlex e deve ser inserida na especificação do arquivo do Cup.
b) parser code {: :} é uma diretiva da especificação do Cup e sua função é permitir incluir o código desenvolvido pelo projetista no código gerado pelo Cup.
c) start with program é uma diretiva do Cup para iniciar a geração do analisador.
d) terminal integer producao1, producaoN; define símbolos terminais que possuem valores inteiros.
e) terminais produção1, producaon; define símbolos terminais

Referências

AHO, A. V., SETHI, R., ULLMAN, J.D.; **Compiladores**: Princípios, técnicas e ferramentas. 2. ed. São Paulo: Pearson, 2007.

APPEL, A. W. **Modern Compiler Implementation in Java.** 2. ed. Cambridge University Press, 2002.

PETTER, M.; HUDSON, S. **CUP User's Manual**. Jun. 2014. v. 0.11b. Modificado por Frank Flannery, C. Scott Ananian, Dan Wang e Michael Petter, com apoio de Andrew W. Appel. Disponível em: <http://www2.cs.tum.edu/projects/cup/docs.php>. Acesso em: 31 jul. 2018.

SEDGEWICK, R.; WAYNE, K. **Algorithms.** 4. ed. Princeton University: Addison-Wesley, 2011.

Geração de código intermediário, do código alvo e otimização

Convite ao estudo

Caro aluno, depois de saber como especificar uma linguagem e desenvolver um analisador sintático, é o momento de transformar o código escrito no código alvo, aquele que o computador realmente consegue executar. A geração desse código alvo, executável, é o objeto de estudo desta unidade.

No início da computação havia poucas linguagens e arquiteturas de computadores, mas hoje há uma grande diversidade de arquiteturas. Essa diversidade resultou na necessidade de conhecer não apenas como gerar o código alvo para um determinado tipo de arquitetura, mas, sobretudo, de estar conectado com a evolução das linguagens e máquinas. Para isso, é importante saber quais as técnicas para geração de um código intermediário, que não seja dependente de uma única arquitetura e também facilite a otimização do código. Assim será mais fácil gerar um código final que possa funcionar em várias arquiteturas.

Para essa etapa do estudo sobre a construção de compiladores, buscamos apresentar por que os conceitos utilizados no seu desenvolvimento são úteis em outras áreas, convidando-o a participar, hipoteticamente, de uma Maratona de Programação. Essa Maratona de Programação, promovida pela Sociedade Brasileira de Computação (SBC) e pela *Association for Computing Machinery* (ACM), existe desde 1996 e é um convite para que os estudantes da área de computação possam aplicar seus conhecimentos teóricos. Assim, a partir de agora, vamos supor que a disciplina de Compiladores foi escolhida para preparar uma equipe de

alunos para representar a instituição na maratona. Essa foi a disciplina eleita, pois a construção de um compilador requer a aplicação de vários fundamentos teóricos da computação.

Para essa preparação, imagine que seu professor tenha estruturado uma competição parecida com a maratona, entre os alunos de sua disciplina, dividindo-a em três fases. Na primeira fase, deseja-se que você seja capaz de apresentar uma proposta para geração de código intermediário e de demonstrar quais fundamentos teóricos foram necessários na construção da solução proposta. Já na segunda fase, espera-se que seja capaz de desenvolver uma pesquisa sobre as técnicas de otimização de código e relacionar quais teorias se aplicam em um cenário que traz inovações a cada dia. Na terceira e última fase, as equipes classificadas deverão propor uma linguagem de programação, planejar e gerenciar o projeto de desenvolvimento do compilador.

As fases serão classificatórias, sendo a primeira fase individual, avançando para a segunda fase apenas 65% dos inscritos. A terceira fase terá apenas os 12 primeiros colocados, e os candidatos classificados formarão três (3) equipes. A equipe finalista estará apta a se inscrever para participar da Maratona de Programação promovida pela SBC e pela ACM. No ano de 2018, a final foi programada para a cidade de Pequim, em 2017 foi nos Estados Unidos, e em 2016 na Tailândia. E você, já está pensando onde será a próxima final? Você poderá estar lá.

O estudo de compiladores o ajudará muito a formar as bases da preparação para essa competição, pois, como pôde perceber ao longo desta disciplina, o tema Compiladores permite a conexão de diversos conteúdos e sua aplicação na construção dos compiladores. Se na fase de análise, vista nas unidades anteriores, vários conceitos de linguagens formais, teoria da computação, estrutura de dados, linguagens de programação foram requisitados para desenvolver os analisadores léxicos e sintáticos, agora, na fase de síntese, objeto nesta unidade, você irá conhecer como desenvolver

o código intermediário, na Seção 1. Posteriormente, você irá saber como é possível melhorar este código, conhecendo as várias técnicas que podem ser aplicadas para a otimização do código e como se dá a geração do código, na Seção 2. Por fim, verá como planejar e gerenciar o desenvolvimento da construção de um compilador e o estudo da biblioteca REGEX, na Seção 3.

Preparado para mais um desafio? Participar desta preparação para a maratona de programação o ajudará, também, na sua formação como profissional que encontra soluções e sabe exatamente o que fazer, seja em situações críticas, seja sob pressão, seja quando solicitado a inovar. Portanto, reiteramos nesta unidade o convite para ir além da sala de aula e estar motivado a participar da Maratona Mundial de Programação. Já está preparando a mochila para participar da grande final? Vamos lá!

Seção 4.1

Geração de código intermediário

Diálogo aberto

Caro aluno, no passado existiam poucos tipos de linguagens, bem como de máquinas. Assim, muitas vezes a tradução do código fonte para o código de máquina era feito em um único passo, mas hoje não é bem assim. Atualmente, além das arquiteturas mais antigas, como CISC (*Complex Instruction Set Computer*) e RISC (*Reduced Instruction Set Computer*), e das novas tecnologias de processadores multicore, vetorial e VLIW (*Very Long Instruction Word*), a cada dia surgem novas tecnologias. Gerar o código alvo diretamente para cada tipo de arquitetura diferente seria muito complexo, e inviável.

A solução para esta situação foi gerar um código intermediário, mais próximo à instruções simples, e independente da máquina, e, posteriormente, e a partir desse código intermediário, realizar a geração do código final (alvo). O objetivo desta seção é estudarmos a geração do código intermediário.

A oportunidade em participar na preparação para a Maratona Mundial de Programação promovida pela Sociedade Brasileira de Computação (SBC) em conjunto com a *Association for Computing Machinery* (ACM), será uma chance ímpar. Os temas abordados na fase de síntese de um compilador irão requer vários conceitos de algoritmos, sistemas operacionais e arquitetura de computadores, ideais em sua preparação para a Maratona de Programação e para ajudá-lo a obter ótimos resultados.

Nessa primeira fase da preparação, será avaliada a sua habilidade para demonstrar, parafraseando Knuth (1989), quanto a prática inspira uma boa teoria e uma boa teoria inspira uma boa prática, pois deverá ser apresentada uma proposta para geração de código intermediário e um relatório comentando, detalhadamente, o código referente à solução que será apresentada, bem como os fundamentos teóricos que foram necessários na construção da solução proposta. Reflita sobre a resposta para a seguinte situação: é possível um compilador para várias linguagens que possa gerar executáveis para diversas plataformas?

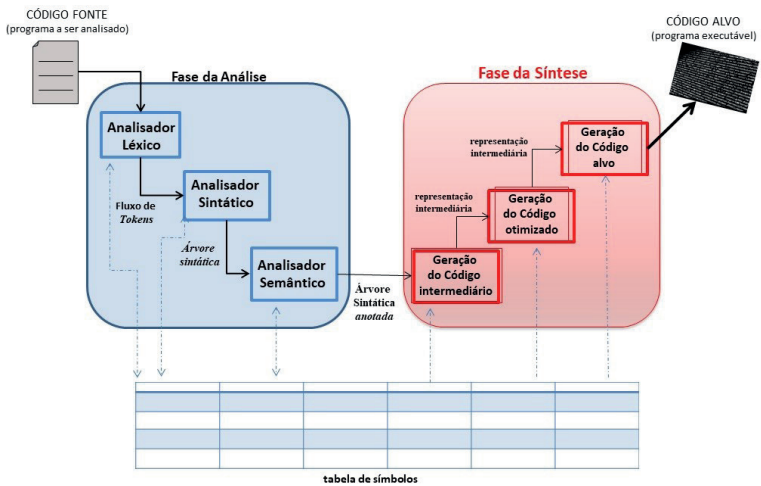
Para ajudá-lo neste desafio, primeiramente iremos estudar as vantagens de gerar um código intermediário, depois passaremos ao estudo das formas para geração do código intermediário a partir da árvore sintática e concluiremos a seção desenvolvendo uma solução prática com uma das técnicas apresentadas. Ávido para aprofundar seus conhecimentos e poder demonstrá-los na prática?

Não pode faltar

O desenvolvimento de um compilador está divido em duas fases: a análise e a síntese. Essas duas fases da estrutura de um compilador, e suas respectivas subdivisões, estão representadas na Figura 4.1.

A fase de análise, também denominada *front-end*, é responsável por verificar se o arquivo-fonte está escrito de acordo com as regras definidas pela gramática. Esta fase está divindade em três etapas: análise léxica, sintática e semântica, como representado na Figura 4.1. O *front-end* disponibiliza para a fase seguinte a árvore de análise sintática, além de alimentar e consultar informações na tabela de símbolo. A fase seguinte, a síntese, também, divide-se em três etapas: a geração do código intermediário, otimização do código e geração do código alvo. As subdivisões da fase de síntese, facilitam a implementação do compilador.

Figura 4.1 | As fases do compilador



Fonte: elaborada pela autora.

A tradução para um código intermediário, como veremos nesta seção, traz uma grande vantagem para o projetista do compilador, pois não haverá a necessidade de escrever um compilador que traduza diretamente a representação da árvore sintática anotada para o código de máquina, o que é um processo altamente complexo e, se assim o fizéssemos, seria necessário escrever um compilador para cada tipo de arquitetura. Entretanto, se subdividimos a fase de síntese, criando um código intermediário que não precisa estar atrelado a uma determinada arquitetura, quando for preciso gerar o código final para diversas plataformas, apenas a tradução do código intermediário precisará ser feita, o que é um processo muito mais simples.

Outra vantagem da geração do código intermediário está em permitir uma análise muito melhor da etapa mais complexa do desenvolvimento de um compilador: a otimização do código antes de efetivamente gerar o código final. Sem essa subdivisão a otimização do código ficaria muito difícil de ser desenvolvida

Reflita

Seja:

M a linguagem de máquina;

A uma representação intermediária;

L uma linguagem de alto nível qualquer;

C1 o compilador de A, escrito em M e gera código alvo M; e

C2 o compilador de L, escrito em A.

Como podemos gerar o código alvo M para o compilador C2?

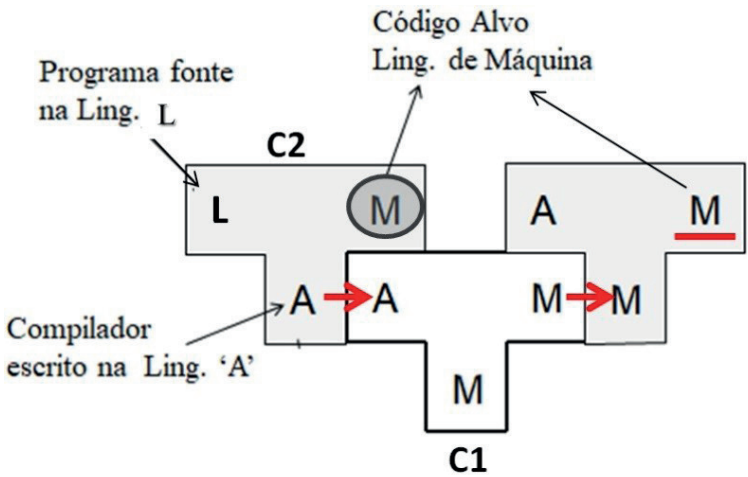
Você se lembra do processo ***Bootstrapping***?

Por meio do processo de *Bootstrapping*, é possível escrever um compilador em qualquer linguagem de programação a partir da existência de dois compiladores.

No caso acima, podemos escrever C2 para gerar o código alvo na representação intermediária A, que pode ser compilada por C1, assim, não será necessário reescrever um compilador para gerar código de máquina para a linguagem L, pois C1, ao compilar o código alvo

A gerado por C2, gera o código de máquina M. A este processo denominamos *Bootstrapping*, e está representado na Figura 4.2.

Figura 4.2 | Exemplo de bootstrapping



Fonte: elaborada pela autora.

Agora reflita: há alguma desvantagem neste processo?

A escolha do código intermediário é muito importante para que seu objetivo não seja perdido, portanto, essa representação, segundo Mogensen (2010), deve garantir:

a) Que seja fácil traduzir a linguagem fonte, a linguagem de alto nível, para a representação intermediária;

b) Que seja igualmente fácil traduzir a representação intermediária para o código de máquina;

c) Que a representação intermediária seja adequada para a otimização.

Os itens (a) e (b) podem ser difíceis de serem conciliados, e, em alguns casos, para facilitar a construção do compilador, podemos ter vários códigos intermediários. É o caso de compiladores com várias passagens, quando aplicamos o processo de *Bootstrapping* repetidas vezes.

Segundo Aho (2007), há três formas básicas de representação intermediária (RI): as árvores sintáticas abstratas, a notação pós-fixa e código de três endereços. No caso da RI por meio da árvore sintática, cada nó é representado por um registro, e cada registro contém um ponteiro para o registro filho. Por exemplo, considere a gramática e a árvore de derivação anotada apresentadas na Figura 4.3. O nó <atrib> aponta para uma folha à esquerda. Essa folha é representada na tabela de símbolo e pelo registro (id, x, inteiro), respectivamente tipo do token, lexema e tipo do dado representado. Já no lado esquerdo da árvore há um apontamento para outro nó, que, por sua vez, tem filhos (folhas), os dados do registro deste nó são (e.val, e1.ponteiro, e2.ponteiro, inteiro), respectivamente o valor (resultado), ponteiros para as folhas e tipo.

Outra forma possível para representação é cada nó ser um vetor de registros, que foi a que utilizamos com a tabela *hash* e os objetos (registros) para armazenar os atributos na análise semântica, implementada no analisador sintático desenvolvido.

Assimile

Atenção com as nomenclaturas e referências ao termo árvores em compiladores.

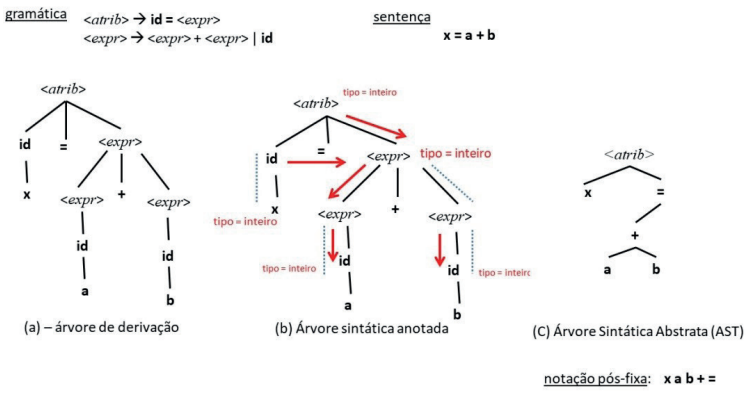
Tenha em mente os seguintes conceitos:

Árvore de derivação – é a representação gráfica da derivação de uma produção. Veja Figura 4.3 (a).

Árvores sintática anotada – é representação da árvore de derivação, com as ações semânticas, também chamada de decorada. Veja Figura 4.3 (b).

Árvore sintática abstrata – é a árvore armazenada em uma estrutura de dados de uma sentença analisada. Sua representação gráfica pode ser vista na Figura 4.3 (c). Também é chamada de AST (*Abstract Syntax Tree*).

Figura 4.3 | Representações de árvores



Fonte: elaborada pela autora.

A outra forma, a notação pós-fixa, é uma representação linear. No caso, a AST representada na Figura 4.3, se percorrida em pós-ordem, resulta em x a b + =, e, se armazenada em uma pilha, permite a execução das operações sem a necessidade de parênteses, quando for o caso, e sempre em pares. Isso facilitará a conversão de instruções complexas em instruções mais simples, objetivo da geração do código, seja o RI ou final.

Pesquise mais

Para a geração do código intermediário, tanto o uso da forma de árvore ou pós-fixa requer conhecimento sobre como percorrer uma árvore binária. Para lembrar como são os percursos em pré-ordem, in-ordem e pós-ordem, assista ao vídeo:

SIDCLEY SANTOS. **Estrutura de dados** – pré-ordem, pós-ordem e i. 14 out. 2014. Disponível em: <https://www.youtube.com/watch?v=FLp8qXM2010>. Acesso em: 30 ago. 2018.

E, para saber mais sobre a notação pós-fixa, você pode consultar os materiais a seguir:

MIRANDA, P. A. V. **Notação infixa para pós-fixa**. 2014. Disponível em: <http://www.vision.ime.usp.br/~pmiranda/mac122_2s14/aulas/aula13/aula13.html>. Acesso em: 30 ago. 2018.

NOTAÇÃO posfixa. [S.d.]. Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas. Disponível em: <http://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/node75.html>. Acesso em: 30 ago. 2018.

Vamos praticar usando a notação pós-fixa, para compreendermos seu funcionamento, que é muito importante, pois este tipo de notação é utilizado em calculadoras, como as HPs 11c e 12c, para cálculos financeiros e científicos respectivamente. Esta notação, também chamada de notação polonesa reversa, usa o conceito de pilha. O uso dessa técnica permitiu programação com uma capacidade de memória bastante reduzida, o que mostra que suas instruções eram simples, o que facilita a geração de uma RI.

No nosso exemplo, o compilador precisará simplificar a instrução A * (B + C) / D. A notação pós-fixa insere a operação depois das variáveis ou número. No caso da expressão infixa A * (B + C) / D, a notação pós-fixa para essa expressão será A B C + * D / . Parece muito complicado à primeira vista, mas para a máquina será fácil, pois os valores são armazenados em uma pilha e, ao encontrarem o operador, sempre dois valores serão desempilhados e o cálculo será feito. Isso simplifica as instruções e não é necessário o uso de parênteses. Veja na Figura 4.4 o esquema de funcionamento da pilha e acompanhe seu funcionamento pelo algoritmo a seguir.

Esse algoritmo lê a sentença de entrada A * (B + C) / D e escreve essa sentença na notação pós-fixa, a qual permite uma conversão mais fácil para o código alvo (executável). O algoritmo para conversão deverá:

1. Criar uma pilha vazia;
2. Ler os *tokens* da sentença de entrada, neste caso temos apenas variáveis, operadores e parênteses, e, a cada *token*,:
    a. Se encontrar um identificador, colocar diretamente na saída;
    b. Se encontrar um OPERADOR (+, -, * ou /):
        i. Enquanto a pilha não estiver vazia e no topo existir um operador com prioridade >= ao encontrado, desempilhar o operador e colocá-lo na saída;

ii. Em seguida, empilhar o operador encontrado na pilha;

c. Se encontrar um abre-parêntese '(', empilhar;

d. e encontrar um fecha-parêntese ')':

i. Enquanto não encontrar o abre-parêntese '(' na pilha, desempilhar o operador e colocá-lo na saída, exceto o abre-parêntese, que deverá ser apenas desempilhado;

3. Ao concluir a varredura da sentença, certificar-se de esvaziar a pilha, desempilhando os elementos diretamente na saída.

Figura 4.4 | Teste do algoritmo para a expressão A * (B + C) / D; conversão de uma notação infixa para pós-fixa

	ENTRADA	prioridade	PILHA	SAIDA	acoes do algoritmo com relacao a entrada
1	A		vazia	A	(2.a) escreve direto na saida
2	*	3	*	A	(2.b) pilha vazia empilha
3	(	1	(*	A	(2.c) sempre empilha
4	B			AB	(2.a) escreve direto na saida
5	+	2	+ (*	AB	(2.b) empilha, pois topo da pilha com prioridade menor
6	C			ABC	(2.a) escreve direto na saida
7	)		*	ABC+	(2.d) desempilha e escreve na saida até achar (
8	/	3	/	ABC+*	(2.b) prioridade igual. Desempilha * e empilha /
9	D			ABC+*D	(2.a) escreve direto na saida
10	vazio		vazia	ABC+*D/	(3) esvazia a pilha e escreve na saida

Fonte: elaborada pela autora.

Exemplificando

Vamos escrever o algoritmo para conversão da notação infixa em pós-fixa em Java, para praticar e fixar. Conceitos de orientação a objetos, fila, pilha e *hash* serão aplicados para o desenvolvimento da solução.

Lembrando que o nosso foco é a RI na forma pós-fixa, assim, neste ponto, os analisadores sintático e semântico já fizeram sua parte e a

sentença que iremos analisar está correta.

A estrutura da nossa classe **Converte** será:

```
import java.util.StringTokenizer;

import java.util.ArrayList;

import java.util.Scanner;


public class Converte {

    // metodo lexico - separa a entrada em tokens

     private static ArrayList<String> lexico(String
sentenca){  ...   }


     //metodo prioridade - define a prioridade dos
operadores

     // aqui é um exemplo de aplicacao de uma acao
semantica

     private static int prioridade(String operador)
{  ...  }


    //metodo posFixa - é nosso algoritmo de conversão

     public static String posFixa(String sentenca){
...  }


    //metodo para testar

    public static void main(String[] args){

        // leitura da expressao infixa a ser lida

        String exprIn;

        Scanner in = new Scanner(System.in);

        exprIn = in.nextLine();

        // converte a expressao
```

```
        System.out.println(Converte.posFixa(exprIn));
    }
}
```

Vamos implementar o algoritmo apresentado no método **posFixa**:

```
//metodo posFixa - é nosso algoritmo de conversão
public static String posFixa(String sentenca){
  // cria saida vazia
     String posFixa = "";

  // cria pilha vazia
     Pilha pilha = new Pilha();

  // chama metodo lexico - que separa a sentenca em tokens
  // IMPORTANTE ; somente é aceita entrada em espaco entre os tokens
  // exemplo: A + B é aceito, MAS A+B não é convertido
     ArrayList<String> infixa = lexico(sentenca);

//  n armazena o numero de tokens a ser analisado
     int n = infixa.size();

// faz a analise de todos os tokens e monta a saida na notacao posFixa
   for (int i = 0 ; i< n; i++){
        // analisa o token
        switch (infixa.get(i)){
          // 2.b - se é um OPERADOR
```

```
            case "+":
            case "-":
            case "*":
            case "/":
            /* 2.b.i. Enquanto a pilha não estiver
vazia e no topo
               existir um operador com prioridade <= ao
encontrado,
               desempilha o operador e coloca na saída;
            */
               while (prioridade(pilha.topoGet()) >=
                                        prioridade(infixa.
get(i))){
                    posFixa = posFixa + " " + pilha.des
empilha();
               }
              // empilha o operador atual para continuar
o processo
               pilha.empilha(infixa.get(i));
               break;

            case "(":
              //2.C - encontrou abre-parentese '(' ->
empilha o token
          //                                  encontrado
               pilha.empilha(infixa.get(i));
               break;
                    case ")":
            /* 2.d - encontrou o fecha-parentese '('
```

```
                é hora de desempilhar e adicionar o
operador a saida

              até encontrar o parentese correspondente
ao abre_parentese

           */

              while ( !pilha.topoGet().equals("(")){

                    posFixa = posFixa + " " + pilha.
desempilha();

              }

              // desempilha o abre_parentese

              pilha.desempilha();

              break;


           default:

                 //numero ou variavel - é adicionado
diretamente a saida

              posFixa = posFixa  + " " + infixa.get(i);


        } // fim do switch


        } // fim do for


        // 3. - esvazia a pilha e adiciona a saida

        while(!pilha.vazio()){

         posFixa = posFixa + " " + pilha.desempilha();

        }

        return posFixa;

    }
```

A implementação do algoritmo exemplificado requer o uso da estrutura de pilha, em que o primeiro elemento a entrar é o primeiro elemento a sair, FIFO (First-In-First-Out). Para uma boa organização do código, e seguindo o padrão de orientação a objeto, é recomendável criar uma classe para gerenciar a pilha com os métodos básicos: empilha (*push*) insere o elemento no topo da pilha, desempilha (*pop*) retorna o elemento do topo da pilha e o remove, topGet (*get*) retorna o elemento do topo da pilha e o método vazio (*empty*) retorna se a pilha está vazia ou não. Essa classe poderá ser consultada no código completo do projeto. Utilizamos a classe **ConcurrentHashMap**, por ser uma estrutura mais atual de *hash* disponibilizada pelo Java.

O projeto completo do exemplo, incluindo todos os métodos e a classe pilha, encontra-se disponível para download no link https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/compiladores/u4/s1/codigo_1.zip ou no QR Code.

Reflita

Na classe **Converte**, qual foi a vantagem de usarmos um método static? Por que os métodos **léxico** e **prioridade** são **private**, e o método **posFixa** é **public**?

Os conceitos de orientação a objeto, em especial os de linguagens de programação, como estudado em paradigmas de programação (Unidade 1), são fundamentais, não somente para implementar um compilador (programar), mas também para entender como funciona uma linguagem de programação com relação ao armazenamento dos dados e execução das instruções na hora de desenvolver a geração do código.

Dica: lembre-se da estrutura de uma classe Java.

*<visibilidade> <modificador><retorno> <**nome**> (<parâmetro>) { <bloco de instruções> }*

Para encerrar, vamos estudar a forma de três (3) endereços. A notação pós-fixa é útil para expressões numéricas, já a forma de três endereços permite que as instruções sejam convertidas

sempre em uma estrutura intermediária simples, a qual utiliza apenas três endereços. Assim, a otimização será mais fácil, bem como a geração do código final. Vamos fazer a conversão passo a passo, transformando expressões complexas em instruções sempre com três endereços:

Código fonte alto nível	Código em 3 endereços

`x = a * ( b + 40 )`	`t0 = int(40)`
	`t1 = b + t0`
	`t2 = a * t1`
	`x = t2`

`if ( x > y ) {`
`  y = y + 1 ;`
`}`
	`_R : if  (x>y) goto _R1`
	`t3 =  int(1)`
	`t4 = y + t3`
	`y = t4`
	`_R1:`

O *bytecode* do JVM (*Java Virtual Machine*) é um exemplo de código intermediário (RI), e talvez um dos melhores casos da vantagem do uso da RI.

Por exemplo, o programa : public class Teste {

public int i = 0;

}

Ao compilar **javac** Teste.java, será gerado o arquivo **Teste.class** em *bytecode*, o código intermediário, o qual será convertido pela JVM para ser executado. É possível ler o *bytecode* gerado usando a classe **javap**, que desmonta o código para a RI, assim, ao executarmos **javap Teste.class**, leremos o código intermediário:

```
Compiled from "Teste.java"
public class Teste {
  public int  i;
```

```
  public Teste();
    code:
       0: aload_0
       1: invokespecial #1;    // Method java/lang/
Object.”<init>”:V
       4: aload_0
       5: iconst_0
       6: putfield #2          // Field i:I
       9: return
}
```

Vale lembrar que o **.jar** é apenas um arquivo compactado dos arquivos **.class** que compõem o projeto. Assim, a cada passo dado no desenvolvimento do compilador, mais você vai descobrindo e conhecendo o funcionamento das linguagens em profundidade, assim, concluímos esta seção mostrando que gerar um código intermediário traz muitas vantagens, tanto para o projetista, quanto em relação à portabilidade do código alvo. Foi possível conhecer as três formas de geração código intermediário ou RI (representação intermediária), desenvolver um exemplo na prática, além de termos observado que, na implementação das diversas fases de um compilador, a teoria e a prática andam de mãos dadas. Na próxima seção, estaremos a um passo de alcançarmos o objetivo final do compilador, o código alvo.

Sem medo de errar

Nesta etapa do estudo sobre a construção de compiladores, foi necessária a utilização de vários conceitos, o que demonstrou a existência de uma interdisciplinaridade, levando-o a vivenciar como uma boa base teórica, conjuntamente com situações que exigem soluções complexas, permite encontrar as melhores respostas para resolver os problemas. Serão justamente esses tipos de problemas que serão propostos na Maratona de Programação promovida pela Sociedade Brasileira de Computação (SBC) e pela *Association for Computing Machinery* (ACM), da qual, hipoteticamente, nesta unidade, você pretende participar.

Para sua preparação, imagine que o professor tenha estruturado uma competição parecida com a maratona entre os alunos dessa disciplina, dividindo-a em três fases. Agora, na primeira fase, deseja-se que você apresente uma proposta para geração de código intermediário e mostre quais fundamentos teóricos foram necessários na construção da solução proposta. Sua habilidade para relacionar a teoria com a prática será requisitada para apresentar uma proposta de geração de código intermediário. A conclusão do trabalho será a apresentação de um relatório, comentando detalhadamente a solução apresentada, bem como os fundamentos teóricos que foram necessários na construção da solução proposta.

Frente às três (3) formas de geração de código intermediário estudadas e às diversas arquiteturas existentes nos dias atuais, o importante será encontrar uma solução que permita a melhor portabilidade possível, sem a necessidade de reescrever a conversão do código alvo. O exemplo do *bytecode* estudado e da JVM é um tipo de solução que permite portabilidade, e, se associado à técnica de *bootstrapping*, a construção da fase de síntese será simplificada. Boa ideia, não acha? Mas, se o desafio é inovar e descobrir soluções, pesquisar se faz necessário, então conheça os tópicos a seguir:

1. Começar por pesquisar como são os testes na maratona, visitando o site da mesma: http://maratona.ime.usp.br/ (acesso em: 14 jul. 2018);
2. Pesquisar a existência de outras máquinas virtuais de processo além da JVM. Sugerimos conhecer a LLVM, no site https://llvm.org/ (acesso em: 14 jul. 2018);
3. O tão falado projeto GNU ("GNU não é Linux"), de software aberto, e o GCC (GNU *Compiler Collection*):
    - https://www.gnu.org/ (acesso em: 14 jul. 2018);
    - https://gcc.gnu.org/ (acesso em: 14 jal. 2018);
4. E o artigo:
    - DOEDERLEIN, O. P. Bytecode: Escondendo e Revelando. **DevMedia**, 2009. Disponível em: <https://www.devmedia.com.br/bytecode-escondendo-e-revelando/12302>. Acesso em: 14 jul. 2018.

Depois de ler este artigo, você irá entender a importância de conhecer o código intermediário e quanto ele pode ajudar a entender seus programas Java em profundidade.

Depois de realizar as pesquisas indicadas, você já tem a resposta para o seu questionamento: sim, é possível um compilador para várias linguagens que gere executáveis para diversas plataformas. O LLVM é um exemplo, o GCC outro. A representação intermediária (RI), como o *bytecode*, permite à JVM interpretar esse código para diferentes plataformas. Fantástico, não é mesmo?

O processo da JVM com o compilador JIT (*Just-in-Time*) permite a geração do código nativo quando necessário, o que compensa a única desvantagem desse processo, uma compilação mais lenta do que se não existisse a RI. Pronto, as vantagens e a importância da geração do código intermediário estão fundamentadas e você já tem um bom material para escrever a justificativa. Agora é apresentar um exemplo. Sugerimos utilizar o exemplo da forma de notação pós-fixa do projeto desenvolvido. E, que tal melhorar e desenvolver o resultado, como uma calculadora, após a conversão da notação infixa para a pós-fixa?

Importante: tente você mesmo, pois, na maratona, você precisará apresentar resultados de desafios em tempo pré-determinado, mas, aqui, você pode dar uma olhadinha na solução disponibilizada para referência.

O projeto completo para conversão da notação infixa para pós-fixa, incluindo o cálculo da notação pós-fixa encontra-se disponível para download no link https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/compiladores/u4/s1/codigo_2.zip ou nd QR Code

E, que tal no seu relatório colocar um exemplo de RI do bytecode? Basta descompilar um dos arquivos (classes) .class do seu projeto com o javap. Envie seu relatório em .pdf e o seu projeto compactado em .zip para o e-mail do professor, no ambiente AVA.

Contente em ter vivenciado como a fase de síntese do compilador está associada aos sistemas operacionais, à arquitetura dos computadores, à programação e em conhecer a

importância da geração do código intermediário? Conhecer as formas existentes para simplificar o código fonte desmistificou esse processo e tornou fácil o entendimento, especialmente pela vivência disso. Preparado para avançar para a fase da otimização desta RI e finalmente alcançar o código alvo?

Avançando na prática

Aplicação de notação pós-fixa na integração da automação com a troca de dados – a revolução 4.0

Descrição da situação-problema

Uma empresa está inovando e integrando sua produção com equipamentos que farão a leitura dos produtos que saem do estoque e seguem diretamente pelas esteiras para embarque. Quando uma caixa é retirada da esteira, antes de alcançar o pallet, o equipamento identifica a ocorrência e a registra em seu sistema. Os equipamentos serão conectados diretamente ao ERP.

Esse equipamento, que foi desenvolvido pelo próprio setor de engenharia da empresa, é inovador, simples, e de baixo custo e vem ao encontro dos objetivos de automação e troca de informações com o ERP, a revolução 4.0. Porém, o equipamento tem pouca memória, e, caso o projeto tenha que ser modificado, haverá atraso na implantação, o que acarretará na terceirização sa produção deste dispositivo, cenário indesejado pela empresa, que pretende, futuramente, comercializar a solução.

O equipamento lê o QRcode das embalagens, e gera:

a) Uma cadeia de caracteres (*string*) com o código dos produtos. Cada produto é separado por um ponto-e-vírgula;

b) Outra cadeia de caracteres, com valor e quantidade separados por espaço. Porém, eles não conseguem encontrar um padrão para interpretar os dados para uso pelo ERP, que precisa manipulá-los. Veja, á esquerda, três exemplos de dados enviados pelo equipamento e, à direita, a saída que se deseja enviar ao ERP.

Dados1

```
C100;C201;C101;
4 5 + 6 -
```

Produto	preço	qt.
C100	4.00	1
C201	5.00	1
C101	6.00	-1

Dados2

```
C123;C701;C300;
3.5 3 x 8 + 9 +
```

Produto	preço	qt.
C123	3.50	3
C701	8.00	1
C300	9.00	1

Dados3

```
C150;C400;C101;
4 3 * 8 2 * + 6 2 * -
```

Produto	preço	qt.
C150	4.00	3
C400	8.00	2
C101	6.00	-2

Eles não sabem como criar uma solução simples para a montagem da integração do ERP, que precisa de uma tabela, como a mostrada do lado direito do exemplo, e do total de cada lote de dados enviados. Você pode contribuir com a equipe de desenvolvimento para encontrar uma solução para este problema?

Resolução da situação-problema

Cada vez mais situações como está irão surgir com o avanço da necessidade de comunicação entre máquinas e sistema integrados de gestão. Depois de analisar os exemplos apresentados, você quase acreditou se tratar de uma notação pós-fixa, mas, para ter certeza, solicitou mais lotes de dados gerados pelo equipamento e submeteu as cadeias do tipo (b) – preço, quantidade e operação – ao seu projeto desenvolvido na disciplina de compiladores, que simplificava cálculos usando a notação pós-fixa, e retornou aos engenheiros os resultados dos valores de cada lote, para que eles validassem se os resultados estavam corretos.

A boa é que eles estavam sim corretos, então era verdade o que você tinha ouvido em aula: uma boa teoria, quando bem aplicada, resulta em uma boa solução, mas isso só se comprova quando temos oportunidade vivenciar esta conexão. O equipamento, provavelmente, usa apenas pilhas de endereçamentos das entradas, simplificando a programação, e é provável que seja um tipo de código intermediário. Agora, basta você ler a entrada e montar a saída, no formato de tabela para o ERP, aplicando a técnica da notação pós-fixa, praticamente num processo reverso.

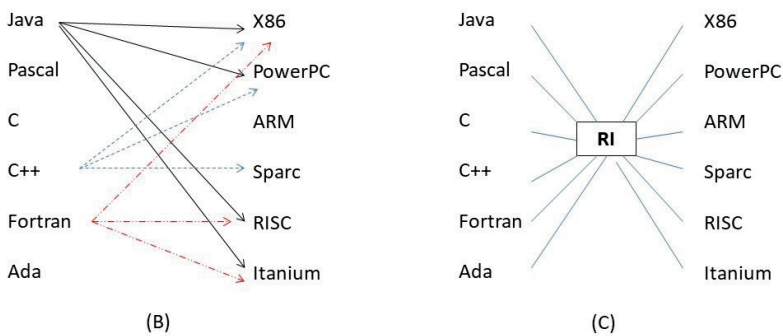
Assim você apresentou a solução em Java e a compartilhou com toda a equipe.

O projeto da solução completa encontra-se disponível para download no link https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/compiladores/u4/s1/codigo_3.zip ou no QR Code.

Faça valer a pena

1. Um compilador que lê o código fonte em uma linguagem e traduz diretamente para o código de máquina não tem portabilidade, já um compilador que pode traduzir um ou mais tipos de linguagens fontes para vários tipos de maquinas é dito portável.
Analise os gráficos a seguir:

Figura | Árvore de representação - linguagem fonte para linguagem de máquina



Fonte: adaptado de Appel (2002, p. 137).

Ao analisarmos a figura, podemos concluir que:

a) Sem a representação intermediária, serão necessários 10 compiladores para todas as linguagens e arquiteturas propostas na Figura (B).
b) A RI é algo que existe apenas como conceito, pois um compilador como o apresentado na Figura (C) não existe.
c) Com a representação intermediária, serão necessários 12 compiladores para todas as linguagens e arquiteturas propostas na Figura.
d) As traduções apresentadas na Figura (B) são os únicos compiladores possíveis de serem construídos sem a RI.

e) Sem a representação intermediária, somente o código intermediário poderá ser utilizado para construir os compiladores da Figura (B).

2. Há três formas para a representação intermediária: AST, notação pós-fixa e código de três endereços. A notação pós-fixa também é conhecida como notação polonesa reversa.
Dada a expressão infixa (3 * 7 + 2 * 15) + 81 – 4.

Qual a representação pós-fixa dessa expressão?

a) 3 7 * 2 15 * + 81 + 4 -
b) 3 7 * 2 + 15 * 81 4 +
c) (3 7 * 2 15 *) 81 4 - +
d) (3 7 * 2 15 *) 81 + 4 -
e) 4 81 - 15 * 2 + 7 3 *

3. A forma intermediária de representação com três endereços facilita a passagem para o código alvo e pode ser representada na notação EBNF. Dada a expressão em C:, area = ((base1 + base2) * altura) / 2, que calcula a área de um trapézio, como fica a representação dessa expressão com três endereços?

Assinale a alternativa correta:

a) t0 = base1 + base2; t1 = t0 * altura; area = t1 / 2;
b) t0 = base1 ; t1 = base2; t3 = altura; area = t0 * t1 / t3;
c) t0 = base1 ; t1 = base2; t3 = altura; area = (t0 * t1) / t3;
d) t0 = base1 + base2; t2 = altura / 2 ; t3 = área;
e) t0 = base1 ; t1 = t0 + base2; t3 = altura / 2 ; area = t3;

Seção 4.2

Geração de código e otimização de código

Diálogo aberto

Na seção anterior, você conheceu as formas de geração do código intermediário e a importância dessa representação intermediária (RI) na construção do back-end de um compilador. Agora, você estudará como é feita a geração do código alvo a partir da RI.

Como existem muitas arquiteturas, é inviável o estudo da geração de código para todas as arquiteturas existentes. Além disso, a cada momento, novos processadores surgem, assim, optamos neste curso pelo estudo da geração de um código alvo para uma máquina hipotética, o que nos permitirá apresentar os conceitos utilizados na tradução da RI para o código alvo, independentemente da necessidade de um profundo conhecimento da arquitetura de um processador, o que tornaria o curso limitado a ele e desviaria do foco principal, que é a geração do código alvo.

Entretanto, o convite que você aceitou para se preparar para a Maratona de Programação, promovida pela Sociedade Brasileira de Computação (SBC) conjuntamente com a *Association for Computing Machinery* (ACM), permitirá a você não só aplicar os conhecimentos que serão desenvolvidos nesta seção, como desenvolver pesquisas para as diversas arquiteturas de processadores, possibilitando um aprofundamento nas arquiteturas em que você tiver maior interesse. Lembramos que essa preparação irá definir quem serão os quatro alunos da instituição que serão inscritos na competição.

Você conseguiu a classificação para a segunda fase ao encontrar uma boa solução e apresentar uma fundamentação teórica consistente. Agora será avaliada a capacidade para pesquisar e expor as opções para otimização de código independente de máquina, frente aos avanços tecnológicos e a diversas opções de hardwares e sistemas operacionais existentes atualmente. Espera-se que você seja capaz de relacionar os avanços tecnológicos das máquinas e sistemas operacionais a suas implicações para a melhoria e otimização do código no desenvolvimento de um compilador.

Assim, o quesito fundamental para obter a classificação para a próxima fase será a apresentação de um trabalho escrito bem estruturado e claro, que relacione os aspectos envolvidos na otimização do código e o estado de arte atual das opções de hardware e sistemas operacionais. Preparado para mais um desafio? Durante a Maratona da Computação, temas são apresentados e respostas precisam ser dadas, por escrito, de forma objetiva e clara. Como você pensa em organizar uma dissertação precisa, com conteúdo que traga pontos importantes e de interesse para uma solução? Será que mostrar diferenças será importante? E, que tipo de diferença interessa saber no caso da geração de código?

Essa etapa da sua preparação para a Maratona de Programação será um convite a caminhar pelo campo do desenvolvimento de softwares de baixo nível, que. Provavelmente, será seu primeiro contato com este tipo de programação e, portanto, poderá ser o momento para desmistificar essa área e descobrir como será fácil depois desta seção. Pronto para conhecer mais sobre geração de código de máquina?

Não pode faltar

O estudo da geração do código intermediário permitiu compreender que, apesar de ser possível a tradução "à mão" da AST (Abstract Syntax Tree) para o código de máquina, esse é um trabalho muito complexo, que exige conhecimento profundo da arquitetura da máquina para a qual se deseja fazer a conversão, mas, sobretudo, é desaconselhável, já que a grande diversidade de plataformas implicaria que o compilador construído estaria limitado a uma única plataforma, portanto, resultaria em um aplicativo de alto custo, pois consumiria muitas horas de trabalho e não teria portabilidade. Logo, a conversão para a uma RI independente de plataforma liberou o projetista do compilador para desenvolver a geração do código alvo em três etapas, facilitando seu trabalho para chegar até o código de máquina.

A primeira etapa da geração do código, a representação intermediária, converte a AST em uma linguagem intermediária, com comandos mais simples que a linguagem fonte, o que a torna mais fácil de ser convertida para a linguagem de máquina, que é um dos objetivos desta seção. O outro tema que iremos tratar nesta seção, que anda de mãos dadas com a tradução para o código

alvo, é a geração de um código de máquina eficiente, que não seja redundante e utilize plenamente os recursos do processador. Este é o objetivo da otimização do código gerado.

Porém, não existe um método que garanta que o código alvo gerado será o mais eficiente, mas, na prática, como afirma Aho (2007), o problema de gerar um código ótimo não pode ser solucionado, contudo, é possível garantir, com a aplicação das técnicas de otimização, um código bom. Aqui, cabe uma desmistificação na construção de compiladores: a fase de otimização do código é muito mais complexa que a de geração do código, pois gerar (traduzir) é associar os comandos da linguagem de alto nível às instruções de baixo nível, percorrendo os nós da AST, enquanto otimizar requer a análise de várias propriedades, de acordo com o tipo de estrutura gerada. Vamos analisar quais são estas propriedades, segundo Aho (2007):

1) Uma transformação precisa preservar o significado dos programas – durante o processo de otimização, não podemos comprometer a execução do programa, gerando códigos que não existiam no programa fonte. Por exemplo, ao otimizar o código, é gerada uma passagem que pode em algum momento resultar em um erro que não existia no programa original, logo, essa otimização não deverá ser feita.
2) Uma transformação precisa acelerar o programa que será gerado, **na média**, e por um fator que possa ser mensurável – a eficiência média é o que importará nesse ponto da otimização, pois nem sempre o fato de aumentarmos o número de instruções no processo de conversão de uma instrução da linguagem de alto-nível para o código de máquina, ocupando mais memória, resulta em menor eficiência na média. Esse é um caso no qual se tenta alcançar uma eficiência média, mensurável pela taxa de crescimento (*Big-Theta*).
3) Uma transformação deve valer o esforço – isso significa que o tempo que o compilador irá gastar para gerar o código alvo tem que ser compensado na eficiência do programa gerado. Podemos dizer que não vale fazer a otimização se "a emenda for pior do que o soneto", isto é, se o resultado do esforço que será gasto na otimização for muito maior que o resultado que será obtido com o programa executável gerado.

Assimile

Para melhor compreensão desta seção de geração e otimização do código, como estamos tratando da tradução de uma linguagem de alto-nível para a linguagem de máquina, tenha em mente essas analogias para o nosso contexto:

Linguagem de alto nível = linguagem fonte = código fonte = é o programa escrito pelo programador em uma linguagem com instruções fáceis de serem compreendias pelos humanos, como linguagem Java, Pascal, Python, C, C#.

Linguagem intermediária = código intermediário = representação intermediária = RI = é um código mais simples para a máquina, gerado pelo compilador e independente do processador. O compilador faz isso depois da análise sintática e semântica, e utiliza a forma pós-fixa ou três endereços para conversão.

Linguagem de baixo-nível = código de máquina = código alvo = é o código que o computador consegue executar.

Complexidade de algoritmos – analisa o custo de um algoritmo, com relação ao tempo gasto. Pior caso (Big-Ô), o melhor caso (Big-Ômega) e a taxa de crescimento (Big-Theta).

Para saber mais sobre complexidade de algoritmo, e especificamente sobre notação assintótica, leia o texto a seguir:

CORMEN, T; BALKCOM, D. **Notação assintótica**. [S.d.]. Khan Academy. Disponível em: <https://pt.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/asymptotic-notation>. Acesso em: 3 set. 2018.

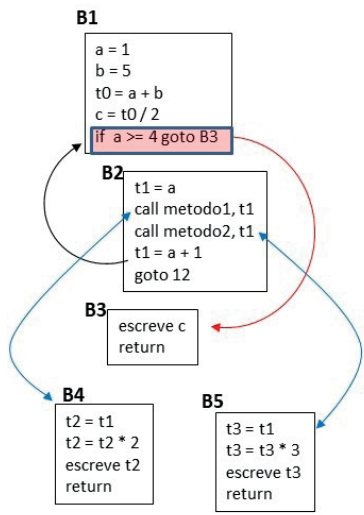
Vamos entender o processo de otimização e a geração do código por meio de um exemplo. Seja a classe Exemplo.java a seguir:

```
public class Exemplo {
    public static void main(String[] args){
        double a, b, c;
        a = 1;
        b = 5;
```

```
            c = ( a + b ) / 2;
            while (a < 4 ){
                Exemplo.metodo1(a);
                a++;
            }
        }
        public static void metodo1(double x){
            System.out.println(x*2);
            Exemplo.metodo2(x);
        }
        public static void metodo2(double x){
            System.out.println(x*3);
        }
}
```

Vamos converter o código-fonte para a RI com 3 endereços e separar o código em blocos básicos. A Figura 4.5 mostra esta conversão e os blocos B1 a B5.

Figura 4.5 | Grafos de Fluxo de Controle (CFG) e Blocos Básicos (BB)



Fonte: elaborada pela autora.

As representações em blocos básicos e grafos de fluxo, segundo Aho (2007), são úteis para a compreensão dos algoritmos de três endereços e os grafos ajudam a coletar informações a respeito do código intermediário durante o processo de conversão para o código de máquina. Assim, entender o que vem a ser um bloco básico (BB) é fundamental para a sistematização do processo de otimização. Segundo Aho (2007, p. 229), "um bloco básico é uma sequência de enunciados consecutivos, na qual o controle entra no início e o deixa no fim, sem uma parada ou possibilidade de ramificação, exceto ao final."

Logo, pela definição, em um algoritmo para particionar um programa em blocos básicos, devemos considerar que um bloco básico começa por um líder, o alvo de um desvio é um líder e a sequência que sucede um desvio é um líder. Assim, na Figura 4.5, o primeiro bloco (B1) tem a instrução líder **a=1** e termina ao encontrar o desvio **if a>= 4 goto B3**, o bloco 2(B2) começa imediatamente depois do desvio, sendo **t1=a** o líder deste bloco, e o mesmo termina ao encontrar o desvio **goto 12**.

Observe que a instrução de alto-nivel ***while*** foi convertida em um desvio condicional na conversão RI de três endereços. No exemplo que estamos analisando, é possível realizar otimização na conversão para três endereços no bloco B4, veja:

Bloco b4

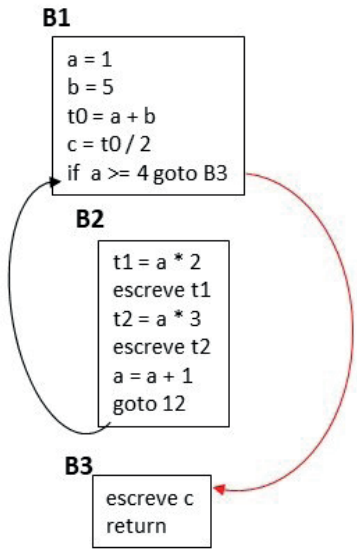
```
t2 = t1
t2 = t2 * 2
escreve t2
return
```

Bloco B4 - otimizado

```
t2 = t1 * 2
escreve t2
return
```

Entretanto, se o valor de t1 for alterado fora do método, poderemos ter problemas, portanto é importante, nas otimizações, lembrar-se das propriedades citadas anteriormente. Neste caso, poderíamos colocar em risco a propriedade relativa ao significado do programa fonte.

Figura 4.6 | Grafos de Fluxo de Controle Otimizado



Fonte: elaborada pela autora.

Na Figura 4.6 o grafo de fluxo de controle está otimizado e representa a otimização final da RI, e, para isso, foram analisados todos os BBs e transformações no código intermediário para melhorar a eficiência do código. Observe o código original, à direita, e as otimizações, à esquerda, a seguir:

TRECHO DO CÓDIGO-FONTE	COMENTÁRIOS COM RELAÇÃO

`while (a < 4 ){`
`Exemplo.metodo1(a);`
`a++;`
`}`
`}`
`public static void metodo1(double x){`
`System.out.println(x*2);`
`Exemplo.metodo2(x);`
`}`
	A Chamada ao **metodo1** está dentro de um loop
	
	O **metodo1** chama o **metodo2**
	
	Assim no processo de otimização os blocos repetitivos foram otimizados dentro do loop
	E ficou:
	`t1 = a * 2`
	`escreve t1` ações do metodo1

```
    public static void meto-
do2(double x){
        System.out.printl-
n(x*3);
    }
}
```

```
            t2 = t1 * 2
            escreve t2    ações
do metodo2

Portanto, as quatro linhas
consistem em um bloco básico e
com menos operações de memória
e desvios.
```

Talvez você esteja se perguntando se o otimizador não seria desnecessário caso o programador em Java já tivesse pensado nisto, e você está parcialmente certo, pois os compiladores avançaram nos processos de otimização e muitas melhorias de desempenho são realizadas durante esse processo, encobrindo falhas de estilo dos programadores. Porém, as otimizações vão além, explorando os recursos específicos da máquina alvo.

Entretanto, tenha em mente que programar é uma arte, como afirma Knuth (1974), e deve-se, portanto, desenvolvê-la com estilo, técnica, eficácia e, sobretudo, eficiência. Porém, para que seus aplicativos alcancem todas essas qualidades, usar os recursos de otimização no processo de compilação poderá fazer toda a diferença no resultado final, quando colocar seus aplicativos em produção.

Reflita

No exemplo apresentado, o diagrama de controle de fluxo (CFG) ajudou a identificar a estrutura de loop? O que se pode concluir sobre as estruturas de loop? São boas candidatas para melhorias?

Como foi possível observar no exemplo, a otimização tem uma fase de análise do RI, na qual usamos as estruturas de BB e CFG para análise, e outra fase de transformação, na qual é gerado um código mais otimizado. Agora, iremos partir para a geração do código alvo, ou simplesmente geração do código.

A geração do código deverá converter a RI em um código que possa ser executado na máquina de destino. As ações desta fase da compilação, conforme definidas por Cooper (2014), são:

1. Seleção de instruções – consiste em selecionar uma sequência de instruções da máquina-alvo que implemente as operações de RI;

2. Escalonamento das instruções – consiste em definir a ordem na qual as operações deverão ser executadas;
3. Alocação de registros – consiste em definir quais valores deverão residir nos registradores em cada ponto do programa.

Para cada ação, seleção de intruções, escalonamento das instruções e alocação de registros, o compilador deverá reescrever o código intermediário para refletir estas decisões. Agora, vamos selecionar as instruções da máquina alvo para efetuarmos a geração do código da classe Exemplo.java. Como nosso código fonte foi escrito em Java, a máquina alvo será a JVM, no caso, o conjunto de instruções de máquina é o *bytecode*.

Pesquise mais

Para ter acesso a todas as instruções da JVM, acesse o manual disponibilizado pela Oracle. O capitulo 6(pag.361) trata especificamente da especificação das instruções.

BUCKLEY, A. et al. **The Java® Virtual Machine Specification**. Oracle, 13 fev. 2015. Disponível em: <https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf>. Acesso em: 3 set. 2018.

A versão em html está disponível em <https://docs.oracle.com/javase/specs/jvms/se8/html/index.html>. Nela, será mais fácil utilizar o tradutor do browser, se precisar.

O número de instruções de uma máquina é grande, porém, a classe Exemplo.java é simples e o seu código RI corresponde a instruções básicas, que apresentamos a seguir:

RI – otimizado	*Bytecode*	Comentários
a =1	0: dconst_1 1: dstore_1 2: dload_1 3: ldc2_w #10	Carrega constante **1** na pilha Armazena na variável **a** Carrega variável **a** na pilha Carrega constante **4** na pilha
if a > = 4 goto 45	6: dcmpg 7: ifge 45	Compara variável **a** com **4** Se a >= 4 desvia para instrução 45
t1 = a	10: dload_1	Carrega variável **a** na pilha

2 t1 * 2	**11**: dstore_3 15: dload_3 16: ldc2_w #6 19: dmul	Armazena na variável **t1** (da pilha) Carrega variável **t1** na pilha Carrega constante **2** na pilha Multiplica **t1** * **2** (valores da pilha)
	12: getstatic #12 20: invokevirtual #13	System.out Método println – imprime valor da pilha
t2 = t1	23: dload_3 **24**: dstore 5 29: dload 5 31: ldc2_w #8 34: dmul	Carrega variável **t1** na pilha Armazena na variável **t2** (pilha) Carrega variável **t2** na pilha Carrega constante **3** na pilha Multiplica **t2** * **3** (valores da pilha)
	26: getstatic #12 35: invokevirtual #13	System.out Método println – imprime valor da pilha
a++	38: dload_1 39: dconst_1 40: dadd 41: dstore_1	Carrega variável a na pilha Carrega a constante **1** na pilha Adiciona **a + 1** (valores da pilha) Armazena na variável a (pilha)
	42: goto 2	Fim do loop – retorna a instrução 2
	45: return	Fim do programa

Exemplificando

Que tal fazer a análise do bytecode de uma classe desenvolvida por você no Java? Para a classe Exemplo.java que utilizamos, basta executar :

javap –v Exemplo.class

Assim, todo o *bytecode* será apresentado, inclusive com os endereços das constantes, mas o resultado não estará otimizado, como o apresentado. Para isso, pode ser utilizado o programa **ProGuard** (disponível em <https://www.guardsquare.com/pt-br/produtos/proguard>, acesso em 3 set. 2018), um analisador, otimizador, ofuscador e verificador de arquivos .class do Java. Esse programa, ao analisar o código, detecta e remove classes, campos, métodos e atributos não utilizados, ptimizando o *bytecode* dos métodos. Além disso, ele age para ofuscar o código. **Mas o que isso significa e por que fazê-lo?**

Ofuscar é encobrir, deixar confuso, e é esse mesmo o objetivo. Na etapa da ofuscação, são renomeadas as classes, os campos e os métodos restantes, usando nomes curtos, sem sentido, para dificultar a engenharia reversa. Assista ao video apresentado no quadro a seguir para praticar. Esse recurso será muito útil para os seus aplicativos em Java, além de ajudar a compreender melhor a geração de código.

Assista ao video com a demonstração desse exemplo de uso do java e do ProGuard, no link https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/compiladores/u4/s2/video_1.mp4 ou no QR Code.

É comum associarmos a geração do código-alvo apenas com a ação de seleção de instruções de máquina para a RI, porém, a fase da geração do código-alvo envolve mais tipos de ações. No exemplo, é possível verificar a ação de escalonamento de instruções, que ocorre nas instruções 12 e 20, que tratam da instância da classe System.out. A instrução 12 ocorreu antes das instruções 15 a 19, pois, antes de realizar as instruções para o cálculo que aparece como parâmetro para o método desta classe, a prioridade é instanciar primeiramente a classe, para que não ocorra erro, e somente depois se preocupar com os cálculos, no caso, o cálculo associado aos parâmetros do método invocado na linha 20. Veja:

`10: dload_1`
`11: dstore_3`
`12:` `getstatic #12`
	System.out

`15: dload_3`
`16: ldc2_w #6`
`19: dmul`
	15 a 19 faz a multiplicação

`20:` `invokevirtual #13`	Método println – imprime valor da pilha

No decorrer desta seção, você pôde ver que a geração do código tem por objetivo principal a transformação da representação intermediária (RI) para o código-alvo. Para isso, seleciona as instruções de máquina correspondentes à RI, define a ordem das instruções e a alocação dos registros. Porém, o compilador tem grande importância com relação à eficiência do código gerado por ele, que será executado na máquina alvo, assim, em várias etapas da conversão, desenvolve ações de análise da RI, com objetivo de otimização das instruções para obter um bom executável final.

Depois de conhecer todas as etapas da construção dos compiladores, você está pronto para desenvolver o seu próprio

compilador. O estudo de compiladores também possibilitou a revisão de seus conhecimentos sobre linguagens formais, algoritmos, paradigmas de linguagens de programação, projeto de software e arquitetura de computadores, afinal, todos eles foram requisitados durante o estudo até aqui. Agora é o momento de fazer uso desse conhecimento no seu dia a dia para nossa próxima seção.

Pesquise mais

Conheça mais sobre a JVM e, em especial, o JIT(*Just-in-Time*), que converte o *bytecode* Java para o código nativo é muito importante, com a leitura do artigo a seguir:

IBM. **O compilador JIT**. [S.d.]. Disponível em: <https://www.ibm.com/support/knowledgecenter/pt-br/SSYKE2_8.0.0/com.ibm.java.vm.80.doc/docs/jit_overview.html>. Acesso em: 3 set. 2018.

Nessa fase do *back-end* do compilador, conhecer/relembrar sobre arquitetura de computador é bastante recomendável. Para uma visão da arquitetura RISC e CISC, consulte o resumo a seguir:

MELO NETO, A. **Computadores com um conjunto reduzido de instruções.** [S.d.]. Disponível em: <https://www.ime.usp.br/~adao/COMPUTADORESRISC.pdf>. Acesso em: 3 set. 2018.

E, caso tenha interesse em "escovar-bits" e mergulhar um pouco mais na linguagem de máquina, você pode ler este outro resumo:

MELO NETO, A. **Conjunto de instruções de um processador (UCP).** [S.d]. Disponível em: <https://www.ime.usp.br/~adao/conjuntodeinstrucoes2018f.pdf>. Acesso em: 3 set. 2018.

Nessa área de geração e otimização de código, há muito para você avançar, especialmente em algoritmos de otimização, tema abordado com mais detalhes por Cooper (2014). Se desejar saber mais sobre os algoritmos de otimização, consulte os capítulos de 8 a 10, já se desejar saber sobre a área de geração de código, veja os capítulos de 11 a 13 do livro *Construindo Compiladores*, de Cooper (2014).

Não deixe de consultar o **apêndice A**, sobre a linguagem **ILOC**, explorada por vários autores e artigos sobre geração de código, que é uma RI para uma máquina RISC hipotética.

Caso queira se aventurar a conhecer um descompilador, explore o Java Decompiler (JD-Gui) (disponível para download em <http://jd.benow.ca/>, acesso em 3 set. 2018). Já se quiser conhecer como melhorar a otimização de código no Visual C++, leia o artigo abaixo:

BRAIS, H. O**timizações do compilador**: Como simplificar o código usando Otimização Guiada por Perfil nativa. Set. 2015. Disponível em: <https://msdn.microsoft.com/pt-br/magazine/mt422584.aspx>. Acesso em: 3 set. 2018.

Lembre-se: sempre que precisar, você pode ativar o tradutor automático do browser que você estiver utilizando. Google Chrome e FireFox têm esse recurso nativo.

Sem medo de errar

Agora é o momento de dar continuidade à sua preparação para a Maratona de Programação, promovida pela Sociedade Brasileira de Computação (SBC) conjuntamente com a *Association for Computing Machinery* (ACM), afinal, você deseja avançar para a próxima etapa e garantir sua vaga para participar da competição, pois apenas quatro alunos da instituição serão inscritos na competição. Nesta etapa, você terá a oportunidade de desenvolver pesquisas sobre arquitetura de computadores, possibilitando um aprofundamento na arquitetura em que você tiver maior interesse.

Nesta fase, será avaliada sua habilidade com pesquisa bibliográfica e sua capacidade para sintetizar, expor e organizar ideias de forma a apresentar as soluções mais relevantes. Por se tratar de um assunto complexo e extenso, o candidato que conseguir apresentar uma resposta que mostre foco em um ponto específico e uma contribuição para o entendimento do assunto terá suas chances de classificação potencializadas. Preparado para mais um desafio? Já pensou em como organizar o seu artigo?

O primeiro passo é fazer a escolha por um assunto. No nosso caso, há dois temas macros para você optar: otimização ou geração do código-alvo. Ambos os temas apresentam subtemas que podem ser explorados, logo, o segundo passo será definir qual elemento específico será abordado. Por exemplo, se você explorar a indicação feita do livro do Cooper (2014) e escolher o assunto geração do

código-alvo, há três subtemas que você poderá escolher: seleção do código, escalonamento de instruções ou alocação de registros.

Feita a escolha, leia sobre o tema especifico e foque na apresentação do tema baseada em um exemplo. Aí vai uma dica: se você precisa apresentar uma contribuição à comunidade, seja acadêmica ou sua equipe de trabalho, é necessário mostrar resultados que sejam úteis e possam ser facilmente aplicáveis. Que tal mostrar o uso das instruções ILOC? E comparar com o *bytecode*? É claro que você não irá fazer isso apresentando todas as instruções não é mesmo, sendo cinco ou seis mais do que suficientes para mostrar um exemplo, como o que você pôde acompanhar nesta seção. Você pode conseguir isso usando o exemplo desta seção ou até mesmo um mais reduzido.

Conclua o seu artigo com alguns comentários sobre as arquiteturas de computadores e a grande vantagem da RI para a construção dos compiladores, utilizando este modelo para a otimizações e geração de código. Como você já sabe, organização é fundamental, etnão que tal seguir este planejamento:

1º. Definir o tema, delimitar o foco e onde buscar informações para balizar o início das investigações.

2º. Desenvolver as pesquisas, sem deixar de definir o tempo exato para isto.

3º. Organizar um roteiro de produção escrita, que pode ser: introdução do tema e a motivação para a escolha do foco que será apresentado, em seguida, apresentar o tema específico tratado no estudo e, finalmente, apresentar o exemplo para mostrar a aplicação dos conceitos tratados.

4º. Mostrar a relação desse tema com a diversidade de arquiteturas e ambientes atuais, não se esquecendo de finalizar mostrando qual foi a contribuição de conhecer mais sobre o tema escolhido para a construção de compiladores ou para melhorias do código gerado.

5º. Não se esquecer das normas da redação: terceira pessoa, sem "achômetros", isto é, tudo que você for apresentar deve ter referências, e não deixe de apresenta-las no final do artigo. Siga um padrão de fonte, tamanho e espaçamento.

Entregue seu trabalho, enviando por e-mail o link onde o mesmo se encontra ao professor. Você pode disponibilizá-lo na nuvem ou elaborá-lo como um artigo, em seu blog ou na sua página do LinkedIn, afinal, este é um tema técnico, com poucos bons materiais em português, e capitalizar um artigo de qualidade feito por você, para o seu currículo, não é má ideia.

Avançando na prática

Como melhorar a segurança e simplificar o código utilizando a otimização do compilador.

Descrição da situação-problema

O departamento de desenvolvimento de uma empresa tem um sistema feito em Java que funciona satisfatoriamente bem, mas às vezes apresenta certa lentidão. Depois de várias reclamações, decidiram acompanhar alguns usuários durante a utilização, e os técnicos apontaram que isso acontecia quando uma determinada rotina fosse chamada.

A equipe de desenvolvimento trocou a rotina, pois a mesma usava recursividade. Porém, o programa cresceu muito em tamanho, utilizando muita memória, e, além disso, agora, todos os usuários observaram uma queda no desempenho. Já o caso crítico teve uma pequena melhora. A equipe não ficou satisfeita com a solução e pediu para você estudar se é possível alguma outra solução, pois sabiam de seu conhecimento sobre compiladores otimizantes. Será que é possível alguma otimização do executável?

Resolução da situação-problema

Você pôde observar, nos seus estudos sobre compiladores, que esse tema não serve apenas para construir este aplicativo, mas também permite conhecer mais a fundo como vários processos são realizados durante a compilação, e ter estudado a otimização e geração de código permitirá a você intervir em alguns parâmetros da otimização, tanto na JVM como no JIT, já que a aplicação foi desenvolvida em Java.

É claro que, polidamente, você irá recomendar que seja feita uma análise métrica da complexidade da classe em questão ou mesmo do sistema que apresenta um desempenho não muito bom, pois é claro que um projeto adequado e um padrão de boas práticas no desenvolvimento de um sistema resultam em um aplicativo também com qualidade. Porém, neste caso, foi solicitado a você atuar na outra ponta: o código alvo.

Mesmo em um aplicativo de qualidade, dependendo das características, se o compilador gerar um código de máquina ruim, uma parte do trabalho feito no desenvolvimento poderá ser perdido, daí a importância do domínio de todas as fases que envolvem um aplicativo. Assim, você deve solicitar para ver a rotina crítica. Caso observe que ocorra algum processo recursivo, como nova instancialização, por exemplo:

```
Cadastro objC = new Cadastro();
    ....................
    estoque.insere(n,objC);
```

Com o uso do ProGuard, como você utilizou nesta seção, poderá, talvez, obter uma melhora na otimização do *bytecode*. Assim, como primeiro passo, você deve fazer a otimização com o ProGuard, usando o projeto original com recursividade, para alcançar uma melhora. Aproveite para visualizar o código e mostrar para a equipe o resultado com o JD-Gui, que é um descompilador.

Ao verificar o tamanho dos .class resultantes, e depois de alguns testes, será possível comprovar melhora no desempenho depois deste passo da otimização. Você deve mostrar à equipe que é possível, também, ofuscar o código para evitar engenharia reversa, ou seja, obter uma segurança a mais durante este processo de otimização. Quando a primeira otimização der resultado, você poderá focar na otimização de uma peculiaridade deste projeto, o fato de ocorrerem várias instancializações.

Para um caso deste tipo, é recomendável ajustar alguns parâmetros para a JVM, já que ela é responsável pela alocação de memória. Nesse caso, a cada nova instância, uma nova locação é realizada. Apesar da JVM fazer isto automaticamente, neste caso é possível intervir para melhorar o desempenho e evitar paradas da JVM para redefinir o tamanho de memória alocado para isto,

provavelmente um dos motivos das paradas do sistema. Você pôde ter contato com este processo ao conhecer as otimizações específicas das arquiteturas RISC e CISC.

No caso do Java, é possível intervir no parâmetro de compilação do *heap*, e no caso específico apresentado, você solicitou a realização de um teste com o limite superior máximo para a situação crítica, e assim poderá definir o melhor valor para o ***heap***. Sendo esta a pior situação, você deve recomendar que seja definido o mesmo valor para o parâmetro ***perm***, assim, durante o uso da aplicação não irá ocorrer parada na JVM para redefinir novos parâmetros para alocação de memória.

Porém, você também deve deixar bem claro que essa solução implica que não haverá realocação de espaço (aumento). Avise que procedimentos para redefinição desses parâmetros deverão ser feitos, caso contrário o sistema irá parar.

Você percebeu como várias áreas do conhecimento da computação foram necessárias para encontrar uma solução adequada para este problema? Neste caso, para uma boa solução não bastou apenas uma otimização padrão, mas também conhecer algoritmos, complexidade de algoritmos, a JVM e gerenciamento de memória.

Faça valer a pena

1. "Um bloco básico é uma sequência de enunciados consecutivos, na qual o controle entra no início e o deixa no fim, sem uma parada ou possibilidade de ramificação, exceto ao final" (AHO, 2007, p. 229).
Para podermos identificar os blocos básicos em um código de três endereços, devemos primeiramente determinar o conjunto de líderes.

Analise as regras a seguir:

I. A primeira linha ou enunciado do código é um líder.
II. Qualquer linha ou enunciado que sucede um desvio condicional é um líder.
III. Qualquer linha ou enunciado que sucede um desvio incondicional é um líder.
IV. Qualquer linha ou enunciado que seja alvo de um desvio é um líder.

Com relação às regras acima, quais são necessárias para definirmos corretamente um líder de um bloco básico (BB)?

Assinale a alternativa verdadeira.

a) II e IV
b) II e III
c) I, II e IV
d) I, III e IV
e) I, II, III e IV

2. O *bytecode* do Java talvez seja, hoje em dia, a representação intermediária (RI) mais conhecida, como no passado o P-Code foi para o Pascal. O *bytecode* é a representação para a JVM, e entender como um compilador a utiliza é útil para o projetista do compilador, bem como para conhecer o próprio funcionamento da JVM. Dadas as instruções a seguir da JVM:

const_n	n pode ser um valor entre -1 a 5.
istore_n	Carrega a constante n na pilha.
iload_n	Armazena a valor carregado na variável.
iadd	Carrega o valor da variável na pilha.
return	Soma os elementos da pilha e carrega o resultado na pilha.
	Encerra.

E o trecho de código em Java: x = 1;
a = x + 2;
Qual a representação correta para a = x + 2; em bytecode?

Assinale a alternativa correta:

a) iconst_2iadd
istore_2

b) iconst_1
istore_1
iload_1
iadd

c) iconst_1
iconst_2
iadd

d) iconst_2
iadd_2

e) iconst_1
   iconst_2
   iadd
   iload_3

3. O objetivo final da geração do código é converter o código-fonte para o código-alvo. Na estrutura conceitual de um compilador, no entanto, o código-fonte é primeiramente convertido para uma RI, com o objetivo de facilitar a otimização e a geração do código-alvo. São várias as ações da geração do código-alvo, e não apenas encontrar a sequência de instruções para a máquina alvo a partir da RI.

Com relação às ações realizadas pela geração do código-alvo, é correto afirmar:

a) As ações da geração do código-alvo são: seleção de instruções e otimização do código.
b) As ações da geração do código-alvo consistem em descobrir qual é a arquitetura da máquina alvo para selecionar corretamente as instruções da máquina.
c) As ações da geração do código-alvo consistem em selecionar as instruções de máquina para a RI, efetuar a alocação de registros e definir a ordem das instruções.
d) A responsabilidade da etapa da geração do código consiste em converter o código intermediário para as instruções da JVM.
e) As ações da geração do código-alvo são: conversão da RI para o código-alvo e o escalonamento das instruções.

Seção 4.3

Especificação de uma proposta de linguagem inovadora

Diálogo aberto

Caro aluno, depois de termos conhecido a estrutura, as técnicas e as ferramentas para a construção de um compilador, esta seção fecha este processo convidando-o a planejar o desenvolvimento de um compilador e a gerenciar o projeto, instrumentalizando-o para escrever seu próprio compilador. Assim, você poderá aplicar esse conhecimento já na próxima etapa, em que irá participar no treinamento para a Maratona de Programação.

A Maratona de Programação, promovida pela Sociedade Brasileira de Computação (SBC) e a *Association for Computing Machinery* (ACM), vem ocorrendo há mais de 23 anos e já passou por vários países, como China, EUA, Chile, etc. Você apresentou um ótimo desempenho no treinamento até aqui, na busca por uma vaga na equipe que irá representar a instituição nesta maratona. Na primeira fase, foi avaliada a capacidade para encontrar uma solução para um problema, já na segunda foram observadas suas habilidades para pesquisa e síntese. Agora, espera-se que seja capaz de trabalhar em equipe para gerenciar, planejar e desenvolver um projeto de compilador, fechando, assim, o ciclo do treinamento e os requisitos para as rodadas classificatórias da maratona oficial.

Para esta fase, foram classificados 12 estudantes que serão divididos em três equipes, por sorteio. A cada equipe, será solicitado realizar as seguintes tarefas para a proposta da criação de uma nova linguagem:

a) Cada equipe deverá eleger um líder do projeto, um arquiteto do sistema, um integrador e um testador do sistema;
b) Escrever um relatorio, em estilo "white paper", para apresentar o manual de referência da linguagem no padrão EBNF;
c) Fazer uma apresentação de dez minutos da linguagem proposta;
d) Apresentar o projeto de desenvolvimento do compilador; e
e) Encerrar com a apresentação do cronograma para desenvolvimento e de sua forma de execução.

A equipe finalista estará apta a se inscrever para participar da Maratona de Programação promovida pela SBC e pela ACM. Mas, para desenvolver um compilador, muitas ações deverão ser feitas e planejadas. Como pretende organizar este trabalho? Trabalhar em equipe requer que tipo de habilidade? Quais são as características esperadas de cada membro na equipe? Como irão realizar estas escolhas? O tempo é curto e o trabalho pesado, então, que tipo de linguagem será adequada para este projeto?

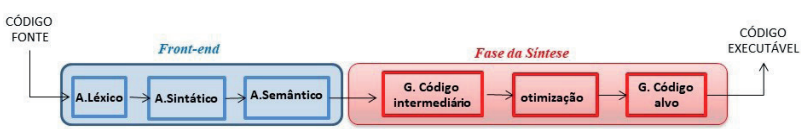
Para ajudá-lo nesta atividade, nesta seção será realizado o estudo de alguns compiladores, também será mostrado o estado de arte das atuais linguagens de programação e o que mudou nos compiladores para as novas necessidades e características das linguagens, convidando-o a olhar para o mercado e analisar as tendências. Além disso, não poderíamos deixar de fora o planejamento de um compilador e como gerenciar um projeto tão complexo.

Concluiremos a seção com um exemplo da biblioteca REGEX, muito útil, pois reduz muitas linhas de programação e é pouco usada por programadores que desconhecem expressões regulares. Porém, os profissionais com formação sólida e com conhecimento sobre linguagens formais e compiladores dominam esse instrumento e podem oferecer soluções diferenciadas. Parabéns, por ter escolhido este caminho.

Não pode faltar

Você conheceu a estrutura clássica de um compilador (Figura 4.7), as técnicas para implementação de cada etapa e a importância desta modularização. Nesta seção, será apresentada uma visão atualizada, as novas tendências, e iremos orientá-lo a explorar ao máximo os conceitos sobre compiladores no desenvolvimento de suas aplicações.

Figura 4.7 | Estrutura Clássica do Compilador



Fonte: elaborada pela autora.

Nos dias atuais, a estrutura conceitual de compiladores continua sendo a mesma, porém, como diversos outros sistemas complexos, não são mais monolíticos, e, além disso, a portabilidade tornou-se imprescindível frente à probabilidade do código gerado precisar ser executado em diversas plataformas.

A época em que a compilação demorava minutos passou. Hoje, o desenvolvedor quer uma IDE (*Integrated Development Environment*) que apresente rapidamente os erros do código fonte e facilite ao máximo a edição de seu código.

Reflita

Podemos notar que a maioria das IDEs, como o Netbeans, Eclipse, DevC++, entre outras, apresenta o seu código fonte colorido. Diferenciando uma palavra reservada de um identificador (variável). Você já pensou em qual técnica é utilizada para implementação deste recurso na IDE ou no editor?

Se a sua resposta foi a análise por um Lexer, parabéns! Os editores, depois de 1989, passaram a analisar os *tokens* e a associar elementos coloridos a cada tipo de *token*, o que tornou mais fáceis as edições do código. Isso graças à análise do código em tempo real e à divisão em etapas de um processo complexo, como é a compilação.

Atualmente, o uso de conceitos aplicados ao compilador tem sido muito mais recorrente para a solução de problemas quando comparado à antigamente, principalmente devido aos sistemas necessitarem de autonomia, flexibilidade e ao aumento da quantidade de aplicações em inteligência artificial (IA).

Este é o caso das *Domain Specific Languages* (DSLs). Por exemplo, a criação de DSLs requer conceitos aplicados à construção de compiladores, e elas são cada vez mais utilizadas. Segundo Kung (2008), elas aumentam a expressividade do código, isto é, tornam o código mais legível. Fowler (2012) as define como pequenas linguagens focadas em um problema específico. Desenvolver DSLs requer domínio sobre analisadores léxicos, sintáticos e semânticos, especialmente se forem externas.

Pesquise mais

Você sabia que CSS e T-SQL são DSLs? Você provavelmente já utilizou a ferramenta MySQL Workbench. Sabia que ela é uma DSL gráfica? Pesquise mais sobre DSLs, pois talvez você seja convidado a criar uma DSL muito antes do que imagina em sua carreira profissional.

1. Para saber o que é DSL consulte o artigo a seguir: ALMEIDA, R. **Criando uma DSL**. 8 ago. 2008. Disponível em: <https://manifestonaweb.wordpress.com/2008/08/08/criando-uma-dsl/>. Acesso em: 3 set. 2018.

2. Sobre *Language Workbench*, melhor ir à fonte e ler: FOWLER, M. **Language Workbench**. 9 set. 2009. Disponível em: <https://martinfowler.com/bliki/LanguageWorkbench.html>. Acesso em: 3 set. 2018. A partir desse artigo, você terá acesso a uma série de artigos sobre o tema, publicados por Fowler, que cunhou esse termo. Caso você não domine a língua inglesa, opte pela tradução automática do browser que você estiver usando.

3. Para conhecer uma DSL na prática leia as páginas 1, e 9 a 14 deste TCC:

XAVIER, Marcos A. C. Farias; **Definição e Especificação de uma DSL para implementação de variações em uma linha de produtos**. 2010, 81 f. Trabalho de conclusão de curso (graduação) – Centro de Informática da UFPE, Recife, 2010. Disponível em: <http://www.cin.ufpe.br/~tg/2010-1/macfx.pdf>. Acesso em: 3 set. 2018.

Como você pôde ver, compiladores estão presentes no dia a dia de quem decide ser um desenvolvedor de software avançado, nas IDEs, nas DSLs internas e externas, e também nas ferramentas para compilação de projetos, como Ant, o antigo Make, ou no MsBuild do .Net., todos exemplos de DSLs.

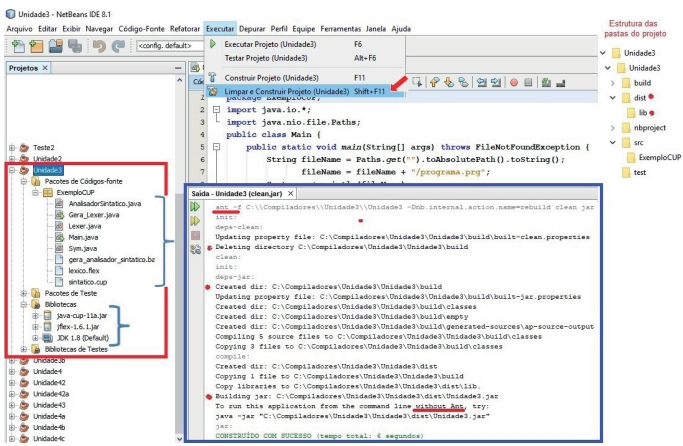
É provável que você já tenha usado **ferramentas de compilação de projeto**, mas talvez nunca tenha se dado conta disso. Este será seu primeiro desafio quando for implantar e, principalmente, distribuir seu projeto, ou seja, fazer a instalação do aplicativo nos clientes. Para isso, será necessário construir um projeto que contenha todas

as dependências necessárias para que o sistema funcione, isto é, a construção da compilação do projeto (build).

O Ant (*Another Neat Tool*), ou Apache Ant, é uma ferramenta para automatização do processo de compilação do projeto, e, como afirma Humble (2014), essa ferramenta já faz parte do desenvolvimento de software há muito tempo. Humble (2014) diz que essa ferramentas tem o objetivo de modelar as dependências do projeto, executando tarefas apenas uma vez. O **Ant** descreve o processo de construção do projeto (*build*) utilizando arquivo no padrão XML. Na Figura 4.8, à esquerda, você pode ver a estrutura do projeto e, à direita, as pastas do projeto. Na pasta **Unidade3**, você encontrará o arquivo **build.xml**, que descreve os comandos para construção do projeto que, por sua vez, aciona os arquivos da pasta **nproject**,, no caso, o arquivo **build-impl.xml**. O objetivo é incluir no projeto todas as dependências para que o sistema funcione na máquina cliente, e é isso que você precisará fazer quando iniciar seus testes de softwares.

Se isso não é muito realizado em um ambiente educacional, é por que, nesta fase, o foco é desenvolver a capacidade técnica para criar o software, e o ambiente é restritivo por questões de segurança. Porém, o estudo do processo não o é, e estudá-lo é importante. Por isso, agora, acreditamos que você irá olhar para os comandos que aparecem na saída quando construir o **.jar**. Veja a chamada do **Ant** na janela saída na Figura 4.8.

Figura 4.8 | Estrutura de um projeto Java no Netbeans



Fonte: captura de tela do Netbeans 8.1, elaborada autora.

Reflita

Você já percebeu que, quando constrói o .jar, as IDEs, seja o Eclipse ou o Netbeans, acionam o **Ant** e tudo é empacotado no .jar? Ótimo, não é mesmo? Mas já existe uma ferramenta que disponibiliza melhorias para a equipe de desenvolvimento, em especial para projeto mais complexos, o **Maven**, uma ferramenta de apoio para automatizar a compilação de projeto.

Para saber mais sobre o **Maven**, recomendamos a seguinte leitura:

OTTERO, R. Introdução ao Maven. **DevMedia**, 12 jul. 2012. Disponível em: <https://www.devmedia.com.br/introducao-ao-maven/25128>. Acesso em: 4 set. 2018.

Para finalizar nosso estudo de compiladores, vamos analisar as linguagens de programação mais populares atualmente. Vejamos o índice TIOBE (*The Importance Of Being Earnest*), que é um indicador da popularidade das linguagens de programação baseado nas pesquisas dos principais mecanismos de busca. A Tabela 4.1 apresenta esse índice de participação, e, como pode ser visto na coluna "% acumulado", cinco (5) linguagens representam praticamente 50% das linguagens mais populares em um ranking de 100 linguagens (TIOBE, 2018).

Tabela 4.1 | Índice TIOBE – julho de 2018

jul/18	jul/17	alteração	Linguagem	% participação	% acumulado
1	1		Java	16,14%	16,14%
2	2		C	14,66%	30,80%
3	3		C++	7,62%	38,42%
4	4		Python	6,36%	44,78%
5	7	^	Visual Basic .NET	4,25%	49,02%
6	5	v	C#	3,80%	52,82%
7	6	v	PHP	2,83%	55,65%
8	8		JavaScript	2,83%	58,48%
9	-	^^	SQL	2,33%	60,82%

10	18		Objective-C	1,45%	62,27%
11	12		Swift	1,41%	63,68%
12	13		Ruby	1,20%	64,88%
13	14		Assembly language	1,15%	66,04%
14	15		R	1,15%	67,19%
15	17		MATLAB	1,13%	68,32%
16	9		Delphi/Object Pascal	1,11%	69,43%
17	11		Perl	1,10%	70,53%
18	10		Go	0,97%	71,50%
19	16		Visual Basic	0,89%	72,38%
20	20		PL/SQL	0,70%	73,09%

Fonte: adaptada de TIOBE (2018).

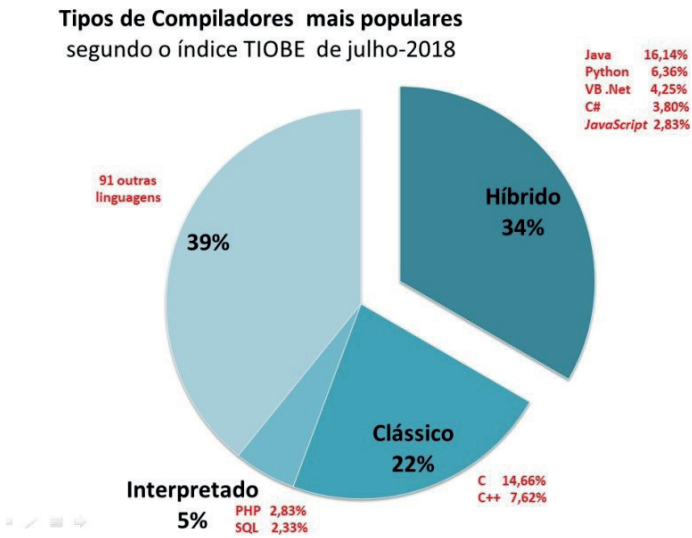
As nove (9) primeiras linguagens classificadas representam 60% das 100 linguagens classificadas pelo índice TIOBE (Tabela 4.1), sendo que a maioria dos compiladores dessas linguagens são híbridos:

- C e C++ são clássicos (fonte -> compilação -> executável);
- PHP e SQL são interpretados, mas vale lembrar que SQL é uma DSL, ou seja, uma linguagem para um fim específico (manipulação de dados);
- Java, Python/Psyco, VB.Net, C# são híbridos, isto é, compiladores que geram a RI e o *back-end* utiliza um compilador JIT;
- JavaScript, nos dias atuais, pode ser interpretado ou compilador. O **Rhino** é uma implementação de linguagem JavaScript, totalmente desenvolvido em Java, portanto um compilador híbrido. Há outros compiladores para JavaScript bastante usados, como o V8 e o Node.

Portanto, 33,37% das linguagens mais populares são compiladores híbridos, fazendo uso da RI e de JIT. Entre os clássicos monolíticos,

temos 22,28%, restando 5,17% de interpretados. Estes valores estão apresentados na Figura 4.9.

Figura 4.9 | % Participação dos tipos de compiladores mais populares



Fonte: adaptado de TIOBE (2018).

Agora que você já conhece o atual estado dos compiladores, vamos supor que você deseja escrever um compilador. Depois de ter estudado sobre o assunto, você sabe que se trata de um sistema complexo, com várias características que precisam ser consideradas na fase do projeto. Durante o desenvolvimento, será necessário um bom gerenciamento, e não termina por aí. Depois da liberação do produto, será necessário gerenciar as versões.

Atualmente, compiladores monolíticos não são muito interessantes para criar uma nova linguagem. A tendência hoje é utilizar compiladores híbridos, e é provável que, na sua carreira de desenvolvedor, você precise escrever uma DSL interna, que utilize a solução híbrida. As vantagens dessa solução são ser modular, que, segundo Aho (2007), pode auxiliar o projetista da linguagem nas mudanças ao longo do tempo de vida do compilador, permite focar no *front-end* e utilizar ***back-ends*** já existentes, possibilitando uma maior portabilidade do código-alvo, além de agilizar o desenvolvimento.

No gerenciamento do projeto do compilador, não poderá ficar de fora a análise das ferramentas de apoio para construção de compiladores, como os geradores de analisadores léxicos, geradores de analisadores sintáticos, ou mesmo os ***compiler-compilers.*** Também não devemos esquecer de escolher a ferramenta para automatizar o processo de compilação do projeto, uma vez que o compilador será composto, como um todo, de módulos, e a automatização da compilação irá ajudar esse processo, bem como o controle da versão do software.

Assimile

Compiler-Compilers – nome dado às ferramentas de apoio para geração de compiladores. Englobam em uma mesma ferramenta o gerador léxico e sintático. Exemplo de *compiler-compilers*: JavaCC, COCO/R, SableCC , ANTLR4.

Automatização do processo de compilação – tem por objetivo a modelagem das dependências do projeto, executando tarefas apenas uma vez. Existem diversas ferramentas para automatizar estas tarefas, tais como o Ant, Make, Maven, Grunt.js.

DSL interna – é uma linguagem limitada, focada em um determinado domínio e escrita na linguagem nativa, porém, em um estilo mais fácil e direcionado, mas ainda próxima a linguagem nativa. Funciona praticamente como uma extensão da linguagem "mãe", diferente da DSL externa, que é completamente separada e precisa ser compilada para linguagem "mãe".

Back-ends – tem como entrada a RI e como saída o código alvo (executável). Projetos como GNU-GCC e LLVM são capazes de analisar diversos tipos de linguagens de entradas e de gerar saída para diversas máquinas alvo, com excelente otimização do código. O uso de *back-ends* garante portabilidade e agilidade na construção do projeto.

Nosso objetivo nesta seção é levar você, aluno, a analisar a teoria e a prática atual. O artigo de Cox (2007) traz um alerta importante sobre como o uso de autômatos finitos na implementação de expressões regulares e o uso de retrocesso na implementação não são uma boa ideia para expressões regulares. Segundo o autor,

Thompson e Ritchie criaram o Unix e trouxeram expressões regulares com eles, implementando-o como um modelo computacional simples. As expressões regulares foram uma característica do Unix em ferramentas como ed, sed, grep, egrep, afirma Cox (2007), e foram exemplos brilhantes na ciência da computação de que uma boa teoria leva a bons programas.

O uso das expressões regulares se espalhou por todos os cantos e, hoje, a maioria das linguagens suportam expressões regulares, porém diz Cox(2007) se tornaram um exemplo brilhante de como ignorar uma boa teoria leva a programas ruins. Algumas soluções atuais são significativamente mais lentas do que as soluções originais. Assim, vale o alerta que usar expressões regulares é uma forma para simplificar a analise de uma *string*, porém analise se a performance da solução encontrada com o seu uso é eficiente.

Exemplificando

Expressões regulares (ER) não são usadas apenas em linguagens formais e compiladores. Elas se tornaram muito populares desde sua aplicação em comandos dos Unix. Vamos ver um exemplo da aplicação da API REGEX no Java para ER.

Vamos utilizar o JUnit para o teste da unidade, que é útil no gerenciamento do desenvolvimento de projetos modulares, como é o caso dos compiladores.

Suponha que você precise fazer a busca de uma palavra em um texto, mas existem muitas variações, e alguns programadores conhecem ER, mas outros não. O ideal seria construir uma classe em que fosse possível passar a lista de palavras a serem buscadas e o método retornasse se a palavra foi encontrada. Caso o programador conheça ER, poderá usá-las ao passar a lista de palavras.

Se a palavra a ser procurada for '**também**' e desejarmos buscar suas variações, será necessário passar a lista: tambem, também, também, tanbém, tambe, tb e tbm. Se o programador conhecer ER, poderá usar a lista com apenas duas variações: **ta[nm]b[eé](m?),** tb.

Relembrando os metacaracteres de ER:

ER utilizam metacaracteres, os mesmos já utilizados no JFlex.

- **[abc]** – caracteres aceitos. O exemplo [nm] indica que são aceitos n ou m, e, no caso de [eé], pode ser aceito e ou é.
- **(x?)** – o caractere x pode aparecer ou não. No nosso exemplo, (m?) indica que a letra m no final é opcional.
- **a | b** – o pipe (|) indica que a ou b são aceitos.
- . – o ponto indica qualquer caractere.
- – o asterisco indica repetição de 0 ou n ocorrências.

Um método simples em que o programador utilizaria a ER, seria:

```
public boolean buscaTexto(String texto){

   String er = ".*(ta[nm]b[eé](m?)|tb).*"; //o .* no
inicio e no fim

                              // faz a busca em qualquer
parte do texto

  return texto.matches(er);

}
```

Mas queremos algo mais prático e flexível, que possa ser utilizado por programadores que conheçam ou não os metacaracteres. Assim, podemos concatenar a lista recebida montando uma ER, facilitando o desenvolvimento pelo programador "como uma DSL interna" bem simples.

Assim nosso método **buscaTexto** ficará:

```
public boolean buscaTexto(String[] p, String texto){

// p é a lista de palavras aceitas, texto onde será
pesquisado

// inicia a montagem da ER : .* indica  aceita qq
string no inicio

//                            e a lista inicia, no caso
usa-se o (
```

```
 String er = ".*(" + p[0];   // concatena a 1a. palavra
da lista

                                // ou ER

// inicia o loop para montagem da ER

   for ( int i=1; i<p.length; i++ ){

       er += "|" + p[i];  // utiliza o pipe (|) para
a indicar a

                                // alternância entre as
palavras da lista

   }

   er += ").*";       // encerra fechado o parentese e
indicando que

                     // pode ocorrer ou símbolos após.

// buscar a ER no texto. Retorna true se encontrou,

//                        caso contrário retorna false

   return texto.matches(er);

}
```

Essa é uma solução bem mais flexível! Para registramos os testes, podemos usar o JUnit, bastando clicar com o botão direito do mouse sobre a classe que contém o método buscaTexto e selecionar Ferramentas/Criar e Atualizar testes.

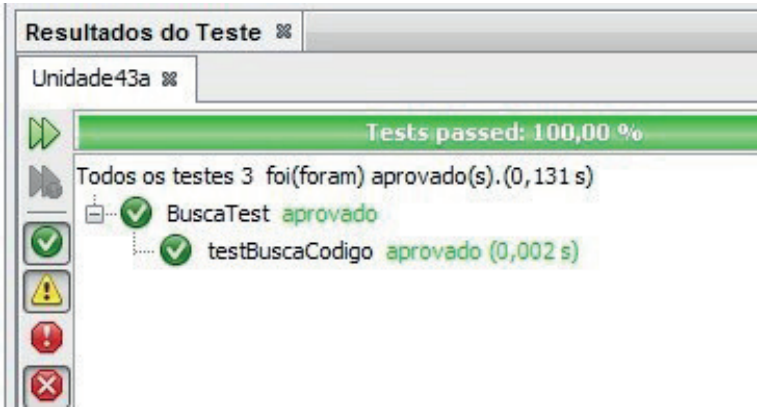
No pacote de Teste, será criada *NomeDaClasseTest*. Pronto, basta montar os seus testes, que ficaram registrados no projeto de compilação. Veja o exemplo a seguir:

```
// as linhas em destaque foram inseridas para o teste

// as demais linhas foram construída pela unidade de
teste

import org.junit.Before;
```

```
import org.junit.Test;

import static org.junit.Assert.*;


public class BuscaTest {

  //criamos um objeto da classe que será instanciada

   // no caso a classe Busca contem o nosso método
buscaTexto

  private Busca teste;

  public BuscaTest() {       }


  @Before

  public void setUp() {

    // aqui crio a instance da classe a ser testada

     teste = new Busca();

  }

  @Test

  public void testBuscaTexto() {

    // inicio dos testes

    // definição dos parametros

           String[]   lista   =   {"tambem","também",
"tanbem","tb","tbm"};

    String texto = "Em casa tambem existem muitos.";

    //uso do meto assert para teste

    // chamo o método com os parâmetros para teste

    // o resultado deverá ser true
```

```
    assertTrue(teste.buscaTexto(lista,texto));

       assertTrue(teste.buscaTexto(lista," tudo tb
passa"));

    String[] lista2 = {"ta[mn]b[eé](m?)","tb"};

   // neste caso o resultado do teste deverá ser false

     assertFalse(teste.buscaTexto(lista2,"mas tamben
deu errado"));

}
```

Figura 4.10 | Resultado do Teste



Fonte: captura de tela do Netbeans 8.1, elaborada pela autora.

Você também poderá fazer o download do projeto completo referente ao exemplo apresentado, por meio do link https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/compiladores/u4/s3/codigo_1.rar ou do QR Code.

Nesta seção, você pôde aprender quais são as linguagens mais populares e os tipos de compiladores dessas linguagens. Também viu que, em projetos complexos, o uso de automatização do processo de compilação é importante, bem como o acompanhamento das unidades de teste. Conhecendo mais sobre DSLs, é possível se

certificar de que os conceitos de compiladores serão aplicados a elas, que são linguagens de domínio específico, sabendo que muitos projetos de software requerem ou mantêm DSLs. Concluímos a seção com um exemplo de aplicação de expressões regulares.

Esperamos que o domínio sobre o tema compiladores seja um dos seus diferenciais como um profissional com formação sólida e com conhecimentos avançados sobre linguagens de programação, e permita a oferecer soluções diferenciadas no seu dia a dia.

Pesquise mais

Para acompanhar um projeto completo de um compilador, consulte o livro:

DEITEL, H. M.; DEITEL, P. J. **Java**: como programar. 6. ed. São Paulo: Pearson Prentice Hall, 2005.

Os capítulos 7 e 17 possuem uma seção especial, denominada "Construindo seu próprio computador", entre as páginas 251 a 255 (cap.7), e 636 a 644 (cap.17). Recomendamos a leitura e o desenvolvimento desse projeto, especialmente por este livro também ser uma boa referência sobre a linguagem Java.

O treinamento de que você está participando para conquistar uma vaga na equipe que irá representar a instituição na Maratona de Programação, promovida pela SBC e ACM, chegou à etapa final. Você passou pelas duas primeiras etapas e, para esta última fase, 12 competidores foram classificados, e você está entre eles.

Por sorteio, foram montadas três equipes. A cada equipe, foi solicitado apresentar uma proposta para criação de uma nova linguagem. A comissão julgadora definou alguns critérios:

a) As equipes deverão definir as funções de cada membro: o líder do projeto, o arquiteto do sistema, o integrador e o testador do sistema;

b) Apresentar um manual de referência da linguagem no padrão EBNF, de preferência on line, em estilo "white paper" do Java, conhecido por você;

c) Fazer uma apresentação à comissão julgadora da proposta, de no máximo 10 minutos;

d) Apresentar o projeto de desenvolvimento do compilador juntamente com o cronograma.

Como primeiro passo, é recomendável é reunir a equipe e organizar um brainstorm, ou seja, deixar as ideias virem para definirem o que desejam desenvolver. A contribuição de todos nesta fase é muito importante. Lembrem-se de que o tempo é curto, então pensem em uma linguagem simples, ou mesmo em uma extensão. Sugerimos uma DSL interna, escrita em Java, que é simples para desenhar, entrada em linguagem fonte próxima ao Java e saída em Java. Nesta solução, a parte de síntese já estará resolvida, e há várias opções que podem seguir esta idéia.

Após a definição do que será feito, antes de começarem a desenvolver as atividades, deve-se definir os papéis de cada membro da equipe. Vocês já se conheceram e tiveram a oportunidade mostrar alguns conhecimentos e características. Algumas dicas:

- Se durante o primeiro contato alguém conduzir os trabalhos com naturalidade, provavelmente essa pessoa pode ser o líder do projeto.
- Os membros mais técnicos poderão ser o arquiteto do projeto e o integrador.
- O responsável pelos testes do sistema poderá ser o membro que questiona e analisa cada passo e detalhe das funções, e também poderá apoiar o líder em algumas tarefas executivas, neste caso.
- Será importante que todos definam em comum acordo os papéis, e alinhados a cada característica da função e comprometimento com a equipe. Isso é fundamental para o sucesso do projeto. Não se esqueçam disto.

A solicitação do manual de referência será util para vocês organizarem a proposta e apresentá-la. É claro que o seu "white paper" não terá as proporções de uma linguagem do porte do Java, porém, há algumas caracteristicas que todas as linguagens possuem, então, que tal organizarem seu documento com estes tópicos:

- **Propósito** – iniciem o documento definindo o propósito da linguagem, o domínio que irá atender;
- **Ambiente** – definam em que ambiente irá trabalhar, do

programa fonte e da máquina alvo, bem como qual será o tipo do compilador;

- **Características principais** – definam o tipo do paradigma da linguagem e os tipos de dados. Se houver alguma estrutura específica que diferencie essa linguagem das demais, não deixe de citar;
- **Sintaxe** – apresentem um ou dois exemplos de como será a sintaxe do programa fonte da linguagem. Neste item, não use o formalismo, mas sim como será para os desenvolvedores a escrita dessa nova linguagem proposta;
- **Gramática** – nesta seção, desenvolvam a notação formal da gramática da proposta em EBNF. Sugerimos começar pela definição dos tipos simples (léxicos) e depois definir a sintaxe.
- **Comentários finais** – que tal concluírem comentando sobre como pretendem realizar o tratamento de erros e encerrem o documento comentando sobre as limitações da linguagem e se elas são vantagens ou desvantagens frente ao propósito apresentado no início.

Após desenvolverem esse documento, você e sua equipe estarão prontos para apresentá-lo. Lembrem-se, vocês terão apenas 10 minutos para a apresentação oral, logo, a sugestão é focar no propósito da linguagem e em suas vantagens, ou seja, mostre que a proposta da linguagem é boa. A especificação EBNF irá interessar apenas ao desenvolvedor. Em uma apresentação de uma nova linguagem, o exemplo do código fonte poderá ser muito mais relevante e interessante.

Depois de desenvolverem a apresentação, resta elaborar o último quesito solicitado pela comissão julgadora: o projeto para o desenvolvimento e o cronograma. Para esse item, vale lembrar que um projeto de compiladores tem alguns pontos que você não deve deixar de fora:

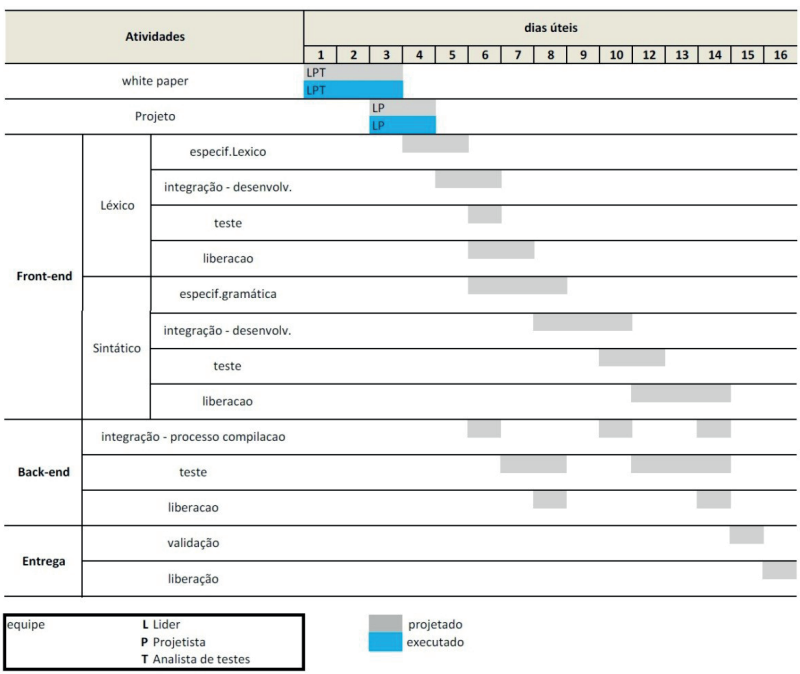
- **Levantamento e análise dos requisitos** – todo projeto de software começa assim. Com o documento escrito no item anterior, vocês já fizeram esse levantamento, portanto o projeto do compilador deverá atender aos requisitos apresentados no "*white paper*".
- **Definição da arquitetura do projeto** - definam os módulos

que serão criados, tais como os elementos que comporão o *front-end* e o do *back-end*, e também em quais plataformas o compilador e o código alvo irão funcionar. Essa etapa é o momento para definir se usará ferramentas para gerar o compilador e quais. Lembrem-se que há um membro da equipe responsável por esse papel de arquiteto do projeto.

- **Implementação** – é recomendável que toda a equipe participe na definição dos tempos para desenvolvimento de cada módulo definido no projeto. E, por se tratar de uma equipe pequena, é recomendável definir as atividades que cada um irá executar durante o desenvolvimento de cada módulo.
- **Testes** – é a fase da definição dos testes e documentação.
- **Validação e liberação da 1ª versão**.

Para montar o cronograma, vocês podem seguir a sugestão apresentada na Figura 4.11.

Figura 4.11 | Sugestão para montagem do cronograma



Fonte: elaborada pela autora.

Depois de você e sua equipe desenvolverem todos os pontos solicitados pela comissão julgadora, restará organizar cada item e entregar ao professor: o *white paper* da linguagem, a apresentação, o projeto do desenvolvimento e o cronograma. Então, aguarde que o resultado seja vencedor.

Avançando na prática

Como reconhecer a data do vencimento do código de barras do boleto

Descrição da situação-problema

Uma associação beneficente passou a ter em sua carteira um novo grupo de contribuintes, porém, eles enviam à entidade um arquivo texto apenas com o código de barras dos boletos. A equipe de voluntários montou um aplicativo para ler o arquivo e extrair o valor, contudo, o crédito somente é feito 3 dias após o vencimento do boleto. A equipe pesquisou no site da Federação Brasileira de Bancos (FEBRABAN) a estrutura do código:

Estrutura do código: bbbmm.nnnnn nnnnn.nnnnn nnnnn. nnnnn n ddddvvvvvvvvv

Onde: bbb – indica o número do banco.

m – indica a moeda, 9 para reais.

n – indica número de 0 a 9.

dddd – indica o número de dias corridos, de 7/10/1997 até o vencimento do titulo.

vvvvvvvvv - indica o valor do título.

Enquanto os códigos estão na estrutura correta, a solução dada consegue capturar o valor do título, porém, quando está fora do padrão, o valor apresentado pelo aplicativo é incorreto. Além disso, não estão sabendo encontrar a data do vencimento a partir da indicação dddd. Você começou um trabalho voluntário na instituição e, devido a seus conhecimentos em programação, perguntaram se conseguiria orientá-los.

Resolução da situação-problema

Poder ajudar a instituição a qual você decidiu oferecer seu trabalho voluntário, aplicando seus conhecimentos de TI, será realmente muito gratificante e ótimo para seu currículo. Uma ferramenta muito útil para validar estruturas de código são as expressões regulares, e você estudou isso em compiladores, portanto, agora é o momento de aplicar.

Com relação a ocorrerem erros na captura do valor, é uma consequência de se trabalhar com uma entrada errada, ou seja, primeiro a entrada precisa estar correta para depois ser feita a conversão. O mesmo conceito é aplicado na construção dos compiladores: primeiro se faz a análise sintática, para garantir que a entrada está correta, para depois fazer a transformação para o código alvo, pois não tem sentido transformar algo errado. Portanto, comecemos pela análise léxica da entrada, montando a ER:

Estrutura do código	ER	comentários
bbb	[0-9]{3}	[0-9] aceita dígitos de 0 a 9 {x} indica quantidade dígitos obrigatórios
mm indica 99	99	
.	[.]	o ponto sem os colchetes indicaria um metacaracter
nnnnn	[0-9]{5}	
	[ ]	Indica um espaço
nnnnn.nnnnnn nnnnn.nnnnnn	([0-9]{5}[.][0-9]{6}[ ]){2}	São **duas** sequências de: 5 dígitos ponto 6 dígitos espaço
n dddd nnnnnnnnnn	[0-9][ ] [0-9]{4} [0-9]{10}	

A expressão regular ficará: [0-9]{3}99[.][0-9]{5}[ ]([0-9]{5}[.][0-9]{6}[ ]){2}[0-9][ ][0-9]{14}. Antes de iniciar a implementação, vamos pensar em como resolver a questão da data do vencimento, já que a equipe, a partir da sequência válida, já sabe capturar o valor. Precisamos capturar o dddd. Depois de se certificar que o *token* padrão do código de barra do boleto está correto, podemos separar

cada elemento do código com segurança. Os separadores dos elementos são ponto e espaço. Pronto, é só aplicar o conceito de separação de *tokens* e capturar os 4 primeiros dígitos do último *token*. Para finalizar, basta adicionar dddd à data "01/10/1997" e teremos a data do vencimento.

Agora que tudo está planejado, é só desenvolver o código e, claro, construir um arquivo de teste, afinal, você não vai mais querer digitar 38 caracteres n vezes para testar todas as possibilidades, e ainda ficar sem a documentação, não é mesmo? Nossa classe para analisar se a estrutura do código está correta será:

```
import java.util.Date;
public class Boleto {
  public Date vencto(String codigoBoleto){
    String er="[0-9]{3}99[.][0-9]{5}[ ]([0-9]{5}
[.][0-9]{6}[ ]){2}[0-9][ ][0-9]{14}";
   Date d = new Date("1997/10/07");
   if (codigoBoleto.matches(er)){
      String[] token = codigoBoleto.split("[.]|[
]");
          String dias = token[token.length-1].
substring(0, 4);
       d.setDate(d.getDate() + Integer.valueOf(
dias));
   } else
       d = null;
   return d;
  }
}
```

Não deixe de criar a classe da unidade de Teste, ok?

Você também poderá fazer o download do projeto completo referente à solução do "avançando na prática", inclusive com os testes, por meio do link https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/compiladores/u4/s3/codigo_2.rar ou do QR Code.

Faça valer a pena

1. Fowler (2012) definiu a linguagem de domínio específico (DSL) como uma linguagem de programação para um determinado domínio, com uma expressividade limitada. Alguns dos objetivos principais das DSLs são simplificar um código complexo, facilitar a contextualização, melhorar a legibilidade do código e aumentar a produtividade no desenvolvimento. Dado o quadro a seguir:

Tipos de DSL	Características
A – DSL interna	**1** – É separada da linguagem principal **2** – Sintaxe semelhante a linguagem principal **3** – Sintaxe customizada
B – DSL externa	**4** – Pode ser embutida na linguagem principal **5** – Possui seu próprio ***parser*** **6** – Usa um subconjunto de recursos da linguagem principal **7** – Apresenta um estilo simplificado de utilização de um subconjunto de recursos da linguagem principal

Relacione os tipos de DSLs com suas respectivas características.

Assinale a alternativa correta:

a) **A** – 2, 6, e 7; **B** – 1, 3, 4 e 5.
b) **A** – 3, 4, 5 e 6; **B** – 1, 2, 3, 6 e 7.
c) **A** – 1, 2, 3, 4, 5, 6 e 7; B – 1, 2, 3, 5, 6, 7.
d) **A** – 1, 2, 3 e 6; **B** – 1, 3, 4, 5.
e) **A** – 2, 3, e 4; **B** – 1, 2, 5 e 6.

2.

> Expressões regulares são uma notação para descrever conjuntos de cadeias de caracteres. Quando uma cadeia de caracteres (string) específica está no conjunto descrito por uma expressão regular, costumamos dizer que a expressão regular corresponde à string.
> [...]
> Duas expressões regulares podem ser alternadas ou concatenadas para formar uma nova expressão regular: Se *e1* combina *s* e *e2* corresponde a *t*, então *e1* | *e2* corresponde *s* ou *t* e *e1e2* corresponde a *st* (COX, 2007, [s.p], tradução nossa).

Dado o método:

```
public boolean validaData(String texto){
    String er = "^[0-9]{2}-[0-9]{2}-[0-9]{4}$";
    return (texto.matches(er));
  }
```

Analise a REGEX ^[0-9]{2}-[0-9]{2}-[0-9]{4}$.

Quais *strings* são aceitas por esta expressão regular?

Assinale a alternativa correta.

a) 01-01-18
b) 12-25-2012
c) 8-12-2010
d) 03-13-17
e) ^03-12-2017$

3. "O Cadastro de Pessoas Físicas (CPF) é um banco de dados gerenciado pela Secretaria da Receita Federal do Brasil – RFB, que armazena informações cadastrais de contribuintes obrigados à inscrição no CPF, ou de cidadãos que se inscreveram voluntariamente."
Fonte: <https://idg.receita.fazenda.gov.br/orientacao/tributaria/cadastros/cadastro-de-pessoas-fisicas-cpf/assuntos-relacionados/perguntas-e-respostas#Resposta1>. Acesso em: 4 set. 2018.

O número do CPF é composto por 11 dígitos. Suponhamos que sejam aceitos dois (2) formatos para o CPF, nnn.nnn.nnn-nn ou nnnnnnnnnn, em que n é um digito entre 0 a 9.

Qual expressão regular, em Java, aceita/reconhece os dois formatos indicados?

Assinale a alternativa correta:

a) \\d{3}.\\d{3}.\\d{3}-\\d\\d
b) ([0-9].{3}){3}[0-9]{2}
c) \\d{3}.?\\d{3}.?\\d{3}-?\\d\\d
d) [0-9]{3}.?[0-9]{3}.?[0-9]{3}-?[0-9]{2}
e) [0-9]{3}.[0-9]{3}.[0-9]{3}-[0-9]{2} | [0-9]{11}

Referências

AHO, A. V.; SETHI R.; ULLMAN J. D. **Compiladores**: Princípios, técnicas e ferramentas. 2. Ed. São Paulo: Pearson, 2007.

AHO, A. V. Teaching the compilers course. **ACM SIGCSE Bulletin**, 40(4):6-8; nov. 2008. Disponível em: <https://www.researchgate.net/publication/220613337_Teaching_the_compilers_course>. Acesso em: 4 set. 2018.

APPEL, A. W. **Modern Compiler Implementation in Java**. 2. ed. Cambridge University Press, 2002.

COOPER, K. D.; TORCZON, L. **Construindo Compiladores**. Trad. Daniel Vieira. 2. ed. Rio de Janeiro: Elsevier, 2014.

COX, R. **A correspondência de expressões regulares pode ser simples e rápida**. Jan. 2007. Disponível em: <https://swtch.com/~rsc/regexp/regexp1.html>. Acesso em: 4 set. 2018.

FOWLER, M.; PARSONS, R. **DSL**: Linguagens Específicas de Domínio. Trad. Eduardo Kessler Piveta. Porto Alegre: Bookman, 2012.

HUMBLE, J.; FARLE, D. **Entrega Contínua**: Como entregar software de forma rápida e confiável. Trad. Marco A. V. Cunha, Ronaldo M. Ferraz. 1. ed. Porto Alegre: Bookman, 2014.

KNUTH, Donald. E. Computer Programming as an Art. **Comunications of ACM**, v. 17, n. 12, p.667-673. Nova Iorque: ACM New York, 1974.

KUNG, F. DSLs não são para gerentes. 30 dez. 2008. Disponível em: <http://blog.caelum.com.br/dsls-nao-sao-para-gerentes/>. Acesso em: 4 set. 2018.

MOGENSEN, T. A. **Basics of Compiler Design**. Ed. de aniversário. Copenhagen: University of Copenhagen, 2010.

TIOBE. **Programming Community Index Definition**. 2018. Disponível em: <https://www.tiobe.com/tiobe-index/programming-languages-definition/#instances>. Acesso em: 4 set. 2018.

ISBN 978-85-522-1099-3