Introdução aos Compiladores

Ed. v1.0

Sumário

1 Introdução **1**

1.1 Linguagens 1
1.2 O que é um Compilador? 2
1.3 Processadores de Programas: Compiladores, Interpretadores e Máquinas Virtuais . . 2
1.4 Organização de um Compilador 4
1.4.1 Análise 5
1.4.2 Síntese 6
1.5 Por que estudar os compiladores? 7
1.6 Aplicações da Tecnologia de Compiladores 8
1.7 Exemplos 8
1.8 Conclusão 9

2 Análise Léxica **10**

2.1 O Funcionamento da Análise Léxica 10
2.1.1 Implementação Manual de um Analisador Léxico 12
2.2 Linguagens Regulares e Expressões Regulares 17
2.2.1 Expressões Regulares 18
2.2.1.1 Expressões básicas 18
2.2.1.2 Caracteres especiais + e ? 19
2.2.1.3 Classes de caracteres, intervalos e negação 19
2.2.1.4 Metacaracteres e sequências de escape 20
2.2.1.5 Outras características 20
2.2.1.6 Alguns exemplos 20
2.3 Geradores de Analisadores Léxicos 21
2.4 Uso do flex 21
2.4.1 Formato da entrada 22
2.4.2 Uma especificação simples do flex 22
2.4.3 Analisador léxico para expressões usando flex 25

2.4.4 Lendo um arquivo de entrada 30
2.5 Análise Léxica de uma Linguagem de Programação 30
2.5.1 A Linguagem Mini C 30
2.5.2 O analisador léxico para a linguagem Mini C 31
2.6 Conclusão 36

3 Análise Sintática **37**
3.1 Estrutura sintática 37
3.1.1 Árvores de expressão 38
3.2 Relação com o Analisador Léxico 40
3.3 Gramáticas Livres de Contexto 41
3.3.1 Exemplo: Palíndromos 42
3.3.2 Derivação 43
3.3.3 Exemplo: Expressões Aritméticas 44
3.3.4 Árvores de Derivação 45
3.3.5 Ambiguidade 46
3.3.6 Exemplo: Linguagem de programação simples 48
3.4 Geradores de Analisadores Sintáticos 49

4 Análise Semântica **50**

Prefácio

texto

Público álvo

estudantes

Método de Elaboração

Financiamento da capes.

Contribuição

Erros e etc.

Capítulo 1

Introdução

OBJETIVOS DO CAPÍTULO

Ao final deste capítulo você deverá ser capaz de:

- Entender a função e a estrutura geral de um compilador
- Diferenciar interpretadores de compiladores
- Compreender os motivos por que se estuda os compiladores

Este capítulo é uma introdução aos compiladores: o que são, para que servem, e como são organizados. Também discutimos para quê se aprende sobre compiladores e onde esse conhecimento pode ser útil. Compiladores são, essencialmente, tradutores de linguagens de programação. Por isso, vamos começar a discussão falando sobre linguagens de programação em geral.

1.1 Linguagens

O que é uma linguagem? Deixamos para os linguistas e filósofos a definição geral do que vem a ser uma linguagem, nos seus vários sentidos. Aqui nos preocupamos apenas com as linguagens de programação, e daqui em diante quando se falar em *linguagem* será entendido que é uma linguagem de programação; quando for preciso tratar de outro tipo de linguagem, isso estará explícito no texto.

Um *programa* é uma seqüência de instruções que devem ser executadas por um computador. Em outras palavras, um programa especifica um algoritmo de maneira executável. Uma *linguagem de programação* é uma notação para escrever programas. Enquanto as linguagens naturais, como português, são notações para comunicação entre pessoas, as linguagens de programação existem, a princípio, para que o programador comunique ao computador as tarefas que devem ser realizadas. A linguagem deve ser, portanto, precisa; o computador não pode fazer julgamentos e resolver ambiguidades.

É importante notar também que um programa é freqüentemente um instrumento de comunicação entre programadores: é comum que um deles tenha que ler e entender programas escritos por outro. Alguns importantes cientistas da computação, aliás, defendem que a comunicação entre programadores é o objetivo primário de um programa, a sua execução sendo praticamente um "efeito colateral". Donald Knuth sugeriu que programação é a arte de dizer a outra pessoa o que se quer que o computador faça.

É impossível estudar compiladores sem estudar as linguagens de programação. Já o contrário é possível: podemos estudar linguagens sem conhecer nada sobre compiladores. Desta forma, as linguagens

se tornam simplesmente notações para descrever algoritmos — o que pode ser suficiente para algumas pessoas — mas em geral se quer executar esses algoritmos e não apenas descreve-los. Para isso é necessário ter ao menos um conhecimento mínimo sobre compiladores. Entender mais do que esse mínimo dá ao programador um maior poder e controle sobre questões de eficiência dos seus programas. Também é importante para aprender a usar melhor as linguagens de programação.

1.2 O que é um Compilador?

Como vimos, uma linguagem de programação é uma notação para escrever programas. Em geral, programas são escritos por pessoas para serem executados por computadores. Mas pessoas e computadores funcionam de forma diferente, o que leva à existência de linguagens de programação com diferentes *níveis*. Os processadores que executam os programas de computador normalmente executam instruções simples e elementares. As linguagens de *baixo nível* são aquelas mais próximas das linguagens dos processadores. Essas linguagens, entretanto, são consideradas difíceis de programar, devido à grande quantidade de detalhes que precisam ser especificados. Assim, algumas linguagens foram criadas para tornar mais fácil a tarefa de programação de computadores. Essas linguagens são chamadas de linguagens de *alto nível*.

Para executar programas escritos em uma linguagem de alto nível, entretanto, é preciso traduzir esses programas para uma linguagem de baixo nível que possa ser executada diretamente por alguma máquina. O programa que faz essa tradução é chamado de *compilador*.

Portanto, um compilador é um programa que traduz programas escritos em uma linguagem, chamada de *linguagem-fonte*, para outra linguagem, a *linguagem-destino*. Normalmente, a linguagem-fonte é uma de alto nível, e a linguagem de destino é uma linguagem de máquina de algum processador, ou algum outro tipo de linguagem de baixo nível que seja executada diretamente por uma plataforma existente. O diagrama na Figura 1.1 [2] resume essa estrutura básica.

Figura 1.1: Estrutura básica de um compilador.

1.3 Processadores de Programas: Compiladores, Interpretadores e Máquinas Virtuais

Mais uma vez, um compilador é um tradutor cujo objetivo principal é transformar um programa para uma forma diretamente executável. Esta não é a única maneira de executar programas em linguagens de alto-nível: uma alternativa é traduzir e executar ao mesmo tempo. É o que fazem os *interpretadores*. Um interpretador puro tem que analisar e traduzir o programa-fonte toda vez que ele precisa ser executado.

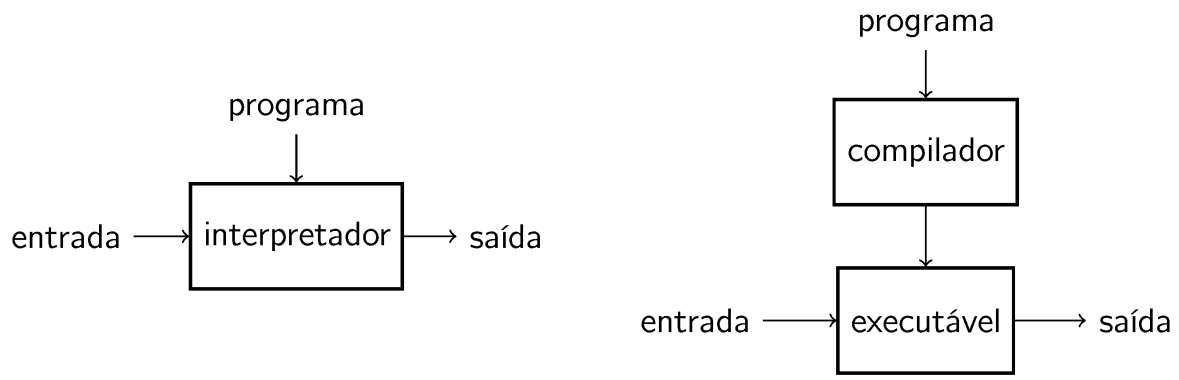

Figura 1.2: Fluxo de execução de um interpretador (à esquerda) e de um compilador (à direita).

Nos sistemas reais, modelos híbridos de interpretação e compilação são comuns. Por exemplo, o compilador Java `javac` não traduz os programas em linguagem Java para alguma linguagem de máquina de um processador, mas sim para a linguagem da máquina virtual Java (JVM), constituída de *bytecodes*. Uma implementação simples da JVM roda o programa compilado em *bytecodes* interpretando-o. Atualmente, a maioria das máquinas virtuais Java compilam o programa em *bytecode* para código nativo da máquina onde reside antes de executa-lo, para melhorar o desempenho. Isso é chamado de compilação *Just In Time*, ou JIT. Da mesma forma, os interpretadores reais não analisam e traduzem o programa inteiro em cada execução; os programas são normalmente transformados para alguma forma intermediária e parcialmente analisados para facilitar sua execução. Também é comum que mesmo linguagens compiladas para código nativo tenham um sistema de tempo de execução (*runtime*) que é acoplado aos programas traduzidos para código de máquina e que, como o nome esclarece, serve para dar suporte ao programa durante sua execução; desta forma, pode-se ter um pouco de interpretação envolvida. Com vista nestes fatos, é difícil dividir exatamente os compiladores dos interpretadores. Nesta disciplina consideramos principalmente os compiladores, mas muito do que é estudado serve também para interpretadores.

Aqui vale a pena considerar a relação entre o modelo semântico de uma linguagem de programação e uma máquina virtual. De fato, cada linguagem de programação pode ser vista como definindo uma máquina virtual que a executa. O modelo semântico da linguagem é o funcionamento desta máquina. Um interpretador puro para uma linguagem é uma máquina virtual para ela. Como no estudo da Organização de Computadores, é necessário organizar as máquinas em camadas. Por isso existe, em um nível mais baixo, a linguagem de máquina, que define o modelo de execução do *hardware* em si; logo acima temos o Sistema Operacional, que define uma linguagem com novas primitivas, conhecidas como *chamadas de sistema*. Acima do SO podemos ter um compilador de linguagem de alto nível que traduz diretamente para código nativo, como o compilador C `gcc`; ou podemos ter uma máquina virtual que executa diretamente uma linguagem em *bytecode*, como é o caso da máquina virtual Java. Acima da JVM temos o compilador `javac`, que traduz um programa em Java para sua versão em *bytecode*.

A definição do que é feito em *software* e o que é feito em *hardware* não é absoluta, sendo estabelecida por motivos de praticidade, desempenho e economia. Poderia se criar um processador que executasse diretamente a linguagem C, mas seu projeto seria complicadíssimo e seu custo muito alto.

1.4 Organização de um Compilador

Na Figura 1.1 [2] a estrutura básica de um compilador é apresentada de uma forma muito simplificada. Agora consideramos essa estrutura em maiores detalhes. Em décadas de desenvolvimento dos compiladores, estabeleceram-se algumas tradições na forma de estruturá-los. Uma dessas tradições é separar o compilador em duas partes principais: a primeira analisa o programa-fonte para verificar sua corretude e extrair as informações necessárias para a tradução; a segunda utiliza as informações coletadas para gerar, ou sintetizar, o programa na linguagem de destino. É o modelo de análise e síntese; a fase de análise também é chamada de vanguarda do compilador (*front-end*) e a de síntese é conhecida como retaguarda (*back-end*). Isso é mostrado na Figura 1.3 [4].

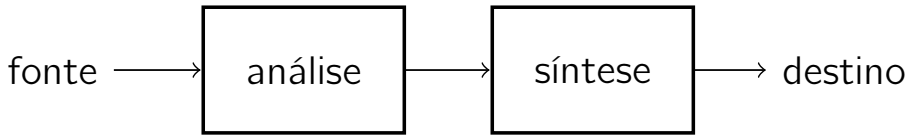

Figura 1.3: O modelo de análise e síntese.

Na figura, vê-se uma ligação entre as duas partes. O que é transmitido entre a análise e a síntese é uma forma chamada de representação intermediária do programa. É como se fosse uma linguagem "a meio caminho" entre as linguagens fonte e destino. A representação intermediária é a saída da fase de análise, e entrada da fase de síntese.

Há uma série de razões para dividir os compiladores desta forma. Uma delas é modularizar a construção dos compiladores: a interface entre análise e síntese fica bem determinada — é a representação intermediária. As duas partes ficam então menos acopladas e mais independentes, podendo ser trocadas sem afetar a outra parte, desde que a interface seja mantida. A construção modular reduz o custo de suportar várias linguagens fonte e várias linguagens destino: digamos que seja necessário compilar M linguagens diferentes para N arquiteturas; se for construído um compilador para cada combinação, serão necessários $M \times N$ compiladores no total. Caso a representação intermediária seja compartilhada, pode-se escrever apenas M módulos de análise e N módulos de síntese, para um esforço total de $M+N$ (ao invés de $M \times N$). Um exemplo dessa técnica é o GCC (GNU Compiler Collection), que usa as linguagens intermediárias RTL, GENERIC e GIMPLE, o que possibilita que os módulos de análise sejam escritos independente dos módulos de síntese; de fato, o GCC suporta várias linguagens (C, C++, Fortran, etc) e gera código para várias arquiteturas (Intel x86, Sparc, MIPS, etc).

O ideal deste modelo é que a representação intermediária fosse completamente independente tanto da linguagem fonte como da linguagem de destino. Neste caso seria possível ter uma representação intermediária universal, e todos os compiladores poderiam utiliza-la: para criar um compilador para uma nova linguagem seria necessário apenas escrever o módulo de análise; para suportar uma nova arquitetura bastaria escrever um módulo de síntese. Na prática, entretanto, isto não é possível. Para gerar código de destino com eficiência aceitável, a representação intermediária de um compilador vai depender tanto de características da linguagem fonte como da linguagem destino.

Agora consideramos em mais detalhe o que os módulos de análise e síntese devem fazer.

1.4.1 Análise

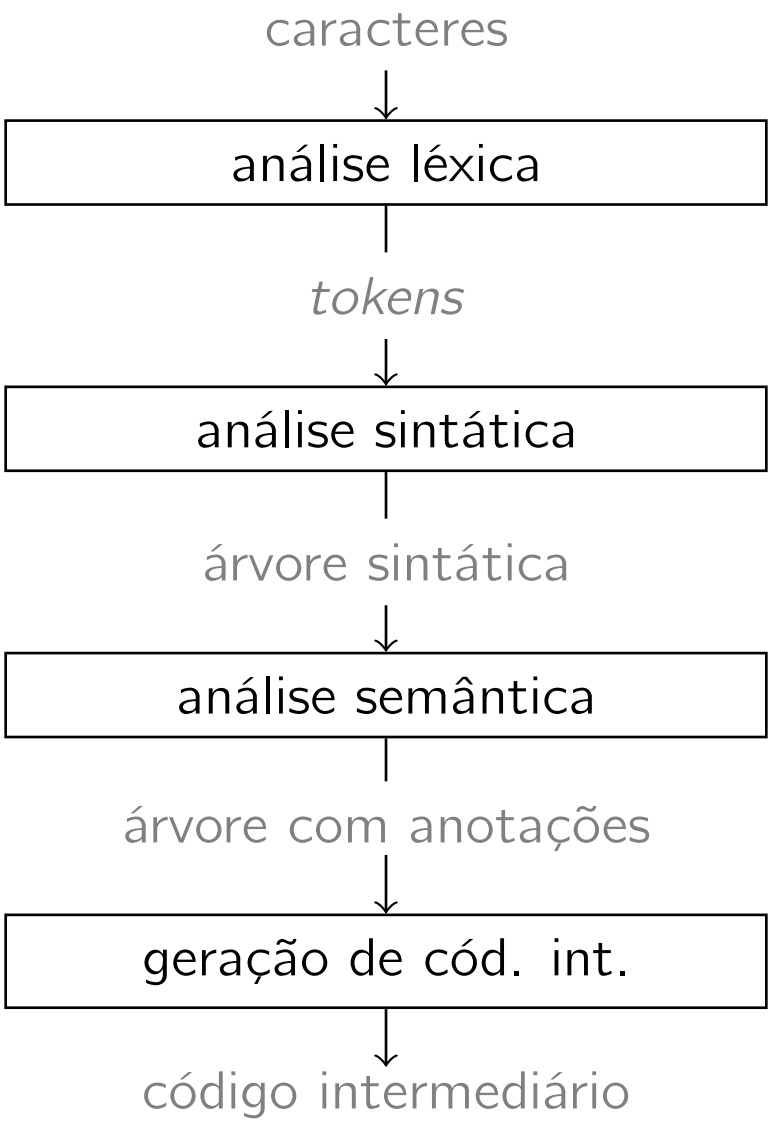

Figura 1.4: Estrutura do módulo de análise.

A Figura 1.4 [5] mostra a estrutura do módulo de análise. O programa-fonte é, inicialmente, um conjunto de caracteres; a tarefa da fase de análise léxica é agrupar esses caracteres em palavras significativas para a linguagem, ou *tokens*. Em seguida, a análise sintática deve, através do conjunto e ordem dos *tokens*, extrair a estrutura gramatical do programa, que é expressa em uma árvore sintática. A análise semântica, ou análise contextual, examina a árvore sintática para obter informações de contexto, adicionando anotações à árvore com estas informações. A fase final da análise é a transformação da árvore com anotações, resultado de todas as fases anteriores, no código intermediário necessário para a síntese.

1.4.2 Síntese

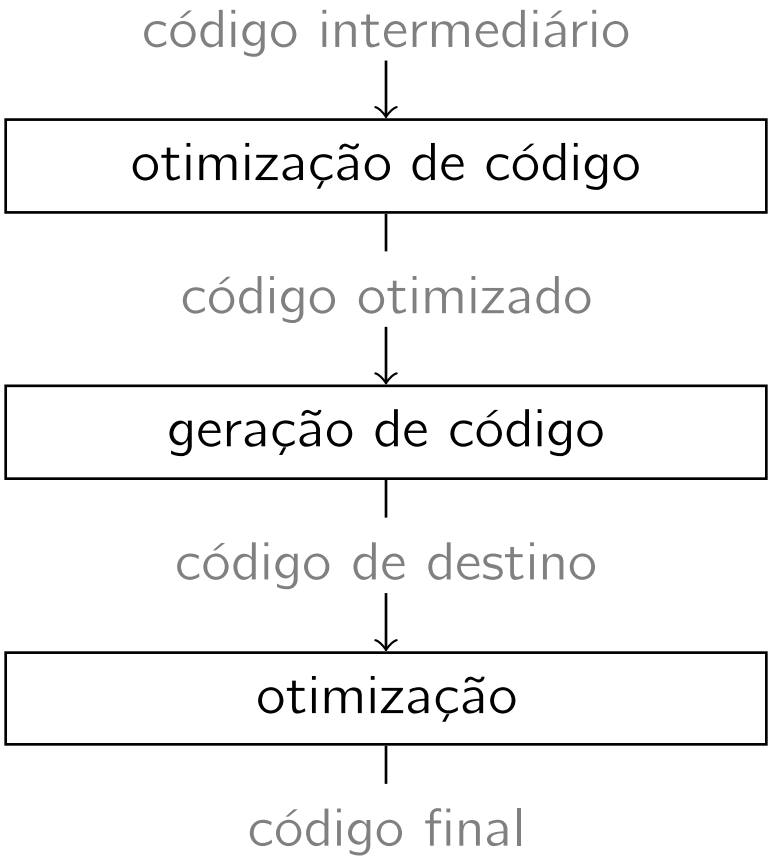

Figura 1.5: Estrutura do módulo de síntese.

O módulo de síntese é detalhado na Figura 1.5 [6]. Primeiro, o código intermediário recebido do módulo de análise é otimizado; o objetivo é tornar o código gerado mais eficiente no uso do tempo e/ou do espaço. Depois, o código intermediário otimizado é utilizado para gerar código na linguagem de destino, geralmente a linguagem de máquina de alguma arquitetura. O código de destino gerado ainda pode ser passado por mais uma fase de otimização, chegando enfim ao código final gerado pelo compilador. Dependendo da arquitetura, também pode ser preciso colocar o código final em um formato adequado para ser executado (não mostrado na figura).

Existem mais dois componentes dos compiladores que não são fases do módulo de análise nem do de síntese. Estes são a tabela de símbolos e o sistema de tempo de execução. A tabela de símbolos é usada por praticamente todas as fases do compilador; durante o processamento do programa fonte, muitos símbolos — nomes de variáveis, funções, classes, módulos e outras construções da linguagem — são definidos e referenciados. A tabela de símbolos guarda as informações sobre cada um deles (por exemplo, o tipo de um símbolo que é o nome de uma variável). O sistema de tempo de execução, como já mencionado, é composto por vários serviços que existem para suportar a execução dos programas gerados. Um exemplo de tarefa realizada pelo sistema de tempo de execução é o gerenciamento de memória, tanto da memória alocada na pilha quanto da memória alocada dinamicamente. Sempre que existe, em um programa em C, uma chamada a `malloc` ou `free`, o sistema de tempo de execução é invocado para administrar o *heap*. Na máquina virtual Java o sistema de tempo de execução inclui o coletor de lixo, que faz o gerenciamento automático da memória alocada dinamicamente.

1.5 Por que estudar os compiladores?

O estudo das linguagens de programação é uma das áreas principais da ciência da computação. Como os compiladores são, a rigor, implementações de linguagens de programação, sua importância fica automaticamente estabelecida. As situações encontradas por cientistas da computação requerem algum entendimento sobre a implementação das linguagens que ele usa, mesmo que ele nunca implemente um compilador em sua carreira.

Mas há outros motivos que tornam este estudo importante e interessante. Aprender sobre compiladores é útil, pois os algoritmos e estruturas de dados utilizados são aplicáveis em vários outros contextos. Compreender como as linguagens são implementadas também confere ao programador um maior conhecimento sobre elas, e quais os custos envolvidos no uso de suas características. Isso permite tomar melhor decisões sobre que linguagem usar para um determinado problema; um profissional competente deve sempre, dentro das restrições apresentadas, escolher a melhor linguagem para cada problema.

O estudo também é interessante do ponto de vista teórico, pois os compiladores interagem com várias outras áreas centrais da computação, demonstrando um ótimo exemplo de sintonia entre teoria e prática:

- **Teoria da Computação** — Como o compilador é um programa, a teoria da computação nos permite prever que tipo de análises podem ser feitas, e quais são possíveis mas a um custo muito alto (problema NP)
- **Linguagens Formais e Autômatos** — Os formalismos empregados na análise sintática vêm do estudo dessa área
- **Arquitetura de Computadores** — É importante para entender as interações entre o código gerado e a máquina que executa o programa, e qual o impacto dessas interações na eficiência do programa gerado
- **Paradigmas de programação** — Permite entender os diferentes modelos semânticos utilizados nas linguagens que devem ser traduzidas

É natural esperar que poucos profissionais da área da computação precisem, algum dia, escrever um compilador para uma linguagem de propósito geral. Entretanto, uma tendência atual no desenvolvimento de software é usar Linguagens de Domínio Específico para dividir a solução de um problema em partes gerais e partes específicas. Como mencionado antes, o uso de LDEs traz vantagens expressivas na criação de soluções, diminuindo a distância entre a linguagem de programação utilizada e os conceitos do domínio do problema. Alguns profissionais e pesquisadores da área já propõem, hoje, um paradigma chamado de Programação Orientada às Linguagens (ou *Language-Oriented Programming*, em inglês), que consiste em sempre criar linguagens específicas para cada sistema desenvolvido. Isso enfatiza a necessidade de se educar sobre linguagens de programação e sua implementação.

Por fim, como ferramentas que traduzem de uma linguagem para outra, os compiladores também são mais abrangentes do que parecem a princípio. Um exemplo na área de banco de dados são os programas que compilam buscas a partir de uma especificação SQL: eles traduzem da linguagem de consulta para um conjunto de operações em arquivos que são as primitivas do banco de dados. Técnicas usadas pelos compiladores também são empregadas para resolver dependências entre equações em planilhas como Excel. Como estes, existem vários outros exemplos, em praticamente todas as áreas da computação, onde as técnicas estudadas nesta disciplina são utilizadas. Alguns exemplos são mostrados a seguir.

1.6 Aplicações da Tecnologia de Compiladores

As técnicas usadas na implementação dos compiladores encontram aplicação em muitos outros problemas que envolvem a análise de uma linguagem de entrada ou a tradução de informações de um formato para outro.

Algumas aplicações estão relacionadas a outras tarefas envolvendo linguagens de programação. Por exemplo, editores de código e IDEs para programação precisam analisar o código para sinalizar erros, sugerir melhorias e fornecer outras ferramentas para auxiliar na programação. Outro exemplo são as ferramentas de *análise estática*, que podem analisar o código-fonte de um programa para descobrir erros ou condições de falha, sugerir melhorias no código ou gerar testes automaticamente.

Outras aplicações que usam das mesmas técnicas dos compiladores são relacionadas à análise de alguma linguagem ou formato de dados de entrada. Um exemplo são aplicações que precisam obter informações que estão em alguma página na *Web*, no formato HTML. Essas aplicações podem usar um analisador sintático de HTML para mais facilmente obter as informações procuradas. De forma similar, algumas aplicações armazenam dados em algum formato derivado de XML, e usar um analisador sintático de XML pode ajudar bastante a acessar as informações dessas aplicações. Além de formatos padronizados como HTML e XML, muitas aplicações usam vários outros formatos proprietários. Saber usar as técnicas de análise sintática usadas em compiladores torna tarefas como essas muito mais simples.

Existem também classes de aplicações que precisam analisar textos escritos em alguma linguagem natural, como a língua portuguesa. Embora um texto em português seja bem mais difícil de analisar do que um código-fonte escrito em alguma linguagem de programação, as técnicas básicas e os conceitos envolvidos são similares. Muitas aplicações de análise de linguagen natural são usadas hoje em dia nas redes sociais. Um exemplo: comitês de campanha eleitoral de um candidato podem coletar o que as pessoas estão falando sobre o candidato nas redes sociais, e determinar automaticamente (sem que alguém precise ler todas as mensagens) se a maioria está falando bem ou mal dele. Com análises mais detalhadas, é possível tentar determinar que pontos positivos e negativos estão sendo comentados; essa é uma tarefa normalmente chamada de *análise de sentimento*. Aplicações de tradução automática de textos em uma língua para outra língua (como o serviço de tradução do Google) também usam algumas técnicas que são similares às utilizadas em compiladores.

1.7 Exemplos

Existem vários compiladores que são utilizados no dia-a-dia pelos programadores. Para as linguagens C e C++ a coleção de compiladores GCC (*GNU Compiler Collection*) é muito utilizada, sendo o compilador padrão em muitas IDEs de programação como o Dev-C++.

Na plataforma Windows, a ferramenta de programação nativa mais utilizada é a IDE Visual Studio, que inclui compiladores para várias linguagens: para C e C++ o compilador usado é o `cl.exe`, enquanto que para C# o compilador é o `csc.exe`.

A linguagem Java possui um sistema de compilação mais complexo, mas o compilador principal do JDK (*Java Development Kit*) é o `javac`, que compila código Java para *bytecodes* da Máquina Virtual Java. A Máquina Virtual Java traduz o código em *bytecodes* para código nativo no momento da interpretação, usando um compilador JIT (*Just In Time*). Outro compilador comumente usado por programadores Java é o compilador incremental incluído como parte da IDE Eclipse.

1.8 Conclusão

Este capítulo serviu como um primeiro contato com as ideias e técnicas envolvidas na implementação de compiladores. Vimos o que são linguagens de programação e o que é um compilador, além da estrutura geral de um compilador e como ela é dividida primariamente nas etapas de *análise* e *síntese*. Essas duas etapas são, por sua vez, divididas em sequências de fases que efetuam tarefas bem definidas. Os capítulos seguintes irão detalhar as técnicas necessárias em cada uma dessas fases.

Capítulo 2

Análise Léxica

OBJETIVOS DO CAPÍTULO

Ao final deste capítulo você deverá ser capaz de:

- Entender a função do analisador léxico dentro de um compilador
- Descrever a estrutura léxica de uma linguagem usando expressões regulares
- Criar o analisador léxico para uma linguagem usando um gerador de analisadores

A análise léxica é a primeira etapa do compilador, e recebe o arquivo de entrada criado pelo usuário. O arquivo de entrada é geralmente armazenado como uma sequência de caracteres individuais que podem ser lidos. A análise léxica tem como função agrupar os caracteres individuais em *tokens*, que são as menores unidades com significado no programa-fonte. Um *token* pode ser pensado como sendo similar a uma palavra.

Podemos fazer uma analogia do processo de análise do compilador com o ato de ler um texto. Na leitura, nós não lemos e decodificamos individualmente cada letra; nosso cérebro lê e processa um texto uma palavra por vez. Isso é comprovado pelo fato que conseguimos entender um texto mesmo que as palavras tenham erros de ortografia ou mesmo sejam escritas de maneira diferente.

A análise léxica faz com que as etapas seguintes do compilador possam trabalhar no nível das palavras, ao invés do nível dos caracteres individuais. Isso facilita bastante o trabalho das etapas posteriores. Na análise léxica também são realizadas algumas tarefas como remover comentários dos arquivos do programa-fonte e registrar em uma tabela os nomes de identificadores usados no programa. Os detalhes de como isso é feito são o tópico deste capítulo.

2.1 O Funcionamento da Análise Léxica

Como vimos, a análise léxica agrupa os caracteres do arquivo de entrada (que contém o programa-fonte) em *tokens*. Um *token* é similar a uma palavra do texto de entrada e é composto por duas partes principais:

1. um *tipo*;
2. um *valor* opcional.

O tipo indica que espécie de “palavra” o *token* representa: um número, um sinal de pontuação, um identificador (nome de variável ou função), etc. O valor é usado em alguns tipos de *tokens* para armazenar alguma informação adicional necessária. Outras informações podem ser associadas a cada *token*, dependendo das necessidades do compilador. Um exemplo comum é a posição no arquivo de entrada (linha e coluna) onde o *token* começa, o que ajuda no tratamento de erros.

Um exemplo vai deixar essas ideias mais claras. Vamos usar uma linguagem simples para expressões aritméticas com operandos constantes, uma linguagem "de calculadora". Na linguagem são permitidos números inteiros (positivos ou negativos), parênteses, e as quatro operações aritméticas básicas, representadas pelos caracteres usuais:

- soma `+`
- subtração `-`
- multiplicação `*`
- divisão `/`

Os tipos de *tokens* são: número, operador e pontuação (para representar os parênteses). Todos os três tipos precisam armazenar informação no campo de valor do *token*. Por exemplo, um *token* do tipo número diz apenas que um número foi encontrado, e o valor do número é guardado no campo de valor do *token*.

Um exemplo de programa nessa linguagem é o seguinte:

Exemplo de expressão aritmética

```
42 + (675 * 31) - 20925
```

Neste exemplo, todos os *tokens* de tipo número são formados por mais de um caractere, o maior tendo cinco caracteres (`20925`). O analisador léxico gera para esse exemplo a seguinte sequência de *tokens*:

Tabela 2.1: Sequência de *tokens* para o exemplo

Lexema	**Tipo**	**Valor**
42	Número	42
+	Operador	SOMA
(	Pontuação	PARESQ
675	Número	675
*	Operador	MULT
31	Número	31
)	Pontuação	PARDIR
-	Operador	SUB
20925	Número	20925

Um *lexema* é a sequência de caracteres que dá origem a um *token*. No exemplo atual, o lexema `20925` gera um *token* de tipo número e valor 20925. Note que o lexema é um conjunto de caracteres, a *string* `"20925"`, enquanto que o valor do token é o valor numérico 20925.

Os valores dos *tokens* de tipo operador representam que operador gerou o *token*, e os valores do tipo pontuação funcionam da mesma forma. Os valores são escritos em letra ção do analisador léxico esses valores são representados por constantes numéricas.

Para tornar esse exemplo mais concreto, vamos examinar a estrutura da implementação do analisador léxico para essa linguagem simples.

2.1.1 Implementação Manual de um Analisador Léxico

Para uma linguagem simples como a linguagem de expressões aritmética do exemplo, escrever um programa que faz a análise léxica não apresenta grande dificuldade. Nesta seção vamos examinar as partes mais importantes do analisador léxico para essa linguagem, pois vários elementos serão similares para linguagens mais complexas.

O código fonte completo do analisador léxico para a linguagem de expressões pode ser encontrado no seguinte arquivo:

Código fonte code/cap2/exp_lexer.c

Aqui vamos analisar as principais partes deste programa. Começamos com a definição da estrutura que vai guardar os *tokens*:

Definição de estrutura para tokens

```
typedef struct
{
  int tipo;
  int valor;
} Token;
```

Como vimos, um *token* tem dois campos: o tipo do *token* e um valor associado. Ambos os campos são inteiros, então definimos algumas constantes para representar os valores possíveis desses campos. As primeiras constantes especificam o tipo de *token*:

Constantes que representam o tipo do token

```
#define TOK_NUM         0
#define TOK_OP          1
#define TOK_PONT        2
```

Com relação ao valor, para números o valor do *token* é apenas o valor do número encontrado. Para operadores e pontuação, por outro lado, precisamos apenas de alguns valores para representar os quatro operadores e dois caracteres de pontuação:

Constantes para operadores e pontuação

```
#define SOMA            0
#define SUB             1
#define MULT            2
#define DIV             3

#define PARESQ          0
#define PARDIR          1
```

O código do analisador léxico usa algumas variáveis globais, para facilitar o entendimento. O programa funciona recebendo o programa de entrada como uma *string* (normalmente um compilador

recebe o programa de entrada em um arquivo). As informações guardadas em variáveis globais são a *string* contendo o código do programa de entrada, o tamanho dessa *string* e a posição atual da análise dentro da *string*:

Variáveis globais para guardar o estado da análise

```
// string que contem o codigo que esta em analise
char *codigo;

// tamanho da string com o codigo
int tamanho;

// guarda posicao atual no codigo
int pos;
```

A análise é iniciada ao chamar a função `inicia_analise`, que estabelece o valor inicial das variáveis globais:

Função para inicializar a análise léxica

```
void inicializa_analise(char *prog)
{
  codigo = prog;
  tamanho = strlen(codigo);
  pos = 0;
}
```

A função `inicia_analise` recebe uma *string* contendo o código do programa de entrada como parâmetro (`prog`), e armazena um ponteiro para essa *string* na variável global `codigo`; a função também estabelece o valor da variável global `tamanho` e inicializa a posição atual na análise com valor zero.

A análise léxica em si funciona de maneira incremental: ao invés de analisar todo o código de entrada de uma vez e retornar todo o fluxo de *tokens*, a função de análise retorna um *token* de cada vez. Por isso, o nome da função que realiza a análise é `proximo_token`, e ela retorna o próximo *token* na sequência a cada vez que é chamada.

Vamos analisar a função `proximo_token` por partes. Começando pelas variáveis locais usadas pela função:

Função que realiza a análise léxica

```
Token *proximo_token(Token *tok)
{
  char c;
  char valor[200];    // string para obter valor de um numero
  int  vpos = 0;      // posicao na string de valor
```

Como indicado nos comentários, a *string* `valor` é usada para determinar o valor de um *token* de tipo número. Isso é necessário porque a função de análise lê um caractere do número de cada vez; a variável `vpos` é usada para guardar a posição atual na string `valor`. A variável `c`, de tipo caractere, guarda o caractere atualmente sendo lido do código de entrada.

Na maioria das linguagens de programação, os espaços em branco não são significativos para o programa, e portanto o programador pode usar quantidades variáveis de espaço entre os *tokens* do programa. Por isso, a primeira tarefa do analisador é pular todo o espaço em branco que for necessário

para chegar até o primeiro caractere do próximo *token*. Isso é feito pela seguinte parte da função `proximo_token`:

Código para pular o espaço em branco antes do próximo token

```
c = le_caractere();
while (isspace(c)) {
  c = le_caractere();
}
```

A função `le_caractere` é uma função auxiliar que obtém o próximo caractere do programa de entrada, atualizando a posição atual dentro da *string* que contém o programa (para detalhes, veja o código-fonte completo do analisador). O código acima lê caracteres da entrada enquanto eles os caracteres lidos forem de espaço (espaço em branco, tabulação e caracteres de nova linha), usando a função `isspace` da biblioteca padrão da linguagem C. Ao final desse *loop*, a variável `c` vai conter o primeiro caractere do próximo *token*.

A função de análise deve então usar esse caractere para determinar que tipo de *token* está sendo lido, e continuar de acordo com o tipo. Nessa linguagem é possível determinar o tipo do *token* olhando apenas para o seu primeiro caractere, mas em linguagens mais complexas isso geralmente não é possível.

Se o primeiro caractere do *token* for um dígito, a análise determina que o próximo *token* é um número. O processo a seguir é ler os próximos caracteres enquanto forem dígitos, armazenando cada dígito lido na *string* auxiliar `valor`. Ao final, o valor do *token* é obtido através da conversão da string `valor` para um número inteiro, usando a função `atoi()`:

Leitura de um token de tipo número

```
if (isdigit(c)) {
  tok->tipo = TOK_NUM;
  valor[vpos++] = c;
  c = le_caractere();
  while (isdigit(c)) {
    valor[vpos++] = c;
    c = le_caractere();
  }
  // retorna o primeiro caractere que nao eh um digito
  // para ser lido como parte do proximo token
  pos--;
  // termina a string de valor com um caractere 0
  valor[vpos] = '\0';
  // converte string de valor para numero
  tok->valor = atoi(valor);
}
```

Se o primeiro caractere não for um dígito, a análise testa se é um caractere de operador e, se for, apenas determina qual constante deve ser usada como valor do *token*:

Leitura de um token operador

```
else if (strchr(ops, c) != NULL) {
  tok->tipo  = TOK_OP;
  tok->valor = operador(c);
}
```

A condição no `if` acima é uma forma mais curta de verificar se o caractere é um dos operadores, ao invés de usar quatro comparações. A constante global `ops` é definida da seguinte forma:

Conjunto de operadores da linguagem

```
const char *ops = "+-*/";
```

E a função `strchr` da biblioteca padrão da linguagem C retorna um valor diferente de `NULL` se o caractere `c` faz parte da *string* `ops`. A função auxiliar `operador` apenas associa o caractere do operador com a constante numérica correspondente; por exemplo se `c` for o caractere `+`, a função `operador` retorna a constante `SOMA`. A definição da função `operador` pode ser vista no código-fonte completo do analisador.

A última possibilidade de *token* válido para essa linguagem ocorre se o primeiro caractere for um parêntese. Nesse caso o tipo do *token* é determinado como pontuação e o valor é a constante `PARESQ` se o caractere lido foi `(` e `PARDIR` se o caractere lido foi `)`. Se o caractere não foi nenhuma das possibilidades anteriores (dígito, operador ou parêntese) a análise retorna o valor `NULL` para indicar uma falha na análise. Isso pode ocorrer porque foi encontrado na entrada um caractere que não pertence à linguagem, ou porque a entrada chegou ao fim. Se não ocorreu uma falha de análise, a função deve retornar o *token* que foi obtido da análise. Isso nos leva ao final da função `proximo_token`:

Final da função de análise léxica

```
  else if (c == '(' || c == ')') {
    tok->tipo  = TOK_PONT;
    tok->valor = (c == '(' ? PARESQ : PARDIR);
  }
  else
    return NULL;

  return tok;
}
```

A função `proximo_token` completa, reunindo os trechos vistos de forma separada, pode ser vista a seguir:

Função completa que faz a análise léxica

```
Token *proximo_token(Token *tok)
{
  char c;
  char valor[200];    // string para obter valor de um numero
  int  vpos = 0;      // posicao na string de valor

  c = le_caractere();
  // pula todos os espacos em branco
  while (isspace(c)) {
    c = le_caractere();
  }

  if (isdigit(c)) {
    tok->tipo = TOK_NUM;
    valor[vpos++] = c;
    c = le_caractere();
    while (isdigit(c)) {
      valor[vpos++] = c;
```

```
      c = le_caractere();
    }
    // retorna o primeiro caractere que nao eh um digito
    // para ser lido como parte do proximo token
    pos--;
    // termina a string de valor com um caractere 0
    valor[vpos] = '\0';
    // converte string de valor para numero
    tok->valor = atoi(valor);
  }
  else if (strchr(ops, c) != NULL) {
    tok->tipo  = TOK_OP;
    tok->valor = operador(c);
  }
  else if (c == '(' || c == ')') {
    tok->tipo  = TOK_PONT;
    tok->valor = (c == '(' ? PARESQ : PARDIR);
  }
  else
    return NULL;

  return tok;
}
```

O código completo do analisador inclui algumas funções de impressão e uma função principal que lê o programa de entrada a partir do teclado e mostra a sequência de *tokens* obtida desta entrada. A função principal é:

Função principal do programa de análise léxica

```
int main(void)
{
  char  entrada[200];
  Token tok;

  printf("Analise Lexica para Expressoes\n");

  printf("Expressao: ");
  fgets(entrada, 200, stdin);

  inicializa_analise(entrada);

  printf("\n===== Analise =====\n");

  while (proximo_token(&tok) != NULL) {
    imprime_token(&tok);
  }

  printf("\n");

  return 0;
}
```

Executando esse programa para a expressão de exemplo que vimos anteriormente, obtemos a seguinte saida:

Saída para a expressão 42 + (675 * 31) - 20925

```
Analise Lexica para Expressoes
Expressao: 42 + (675 * 31) - 20925

===== Analise =====
Tipo: Numero     -- Valor: 42
Tipo: Operador   -- Valor: SOMA
Tipo: Pontuacao  -- Valor: PARESQ
Tipo: Numero     -- Valor: 675
Tipo: Operador   -- Valor: MULT
Tipo: Numero     -- Valor: 31
Tipo: Pontuacao  -- Valor: PARDIR
Tipo: Operador   -- Valor: SUB
Tipo: Numero     -- Valor: 20925
```

A saída está de acordo com o que esperamos da análise léxica dessa linguagem, como pode ser visto na Tabela Tabela 2.1 [11].

Vimos que para uma linguagem simples como a de expressões, é fácil criar diretamente o analisador léxico necessário. Entretanto, à medida que a estrutura da linguagem se torna mais complexa (como ocorre nas linguagens de programação real), a complexidade do analisador léxico vai crescendo e se torna difícil criar o analisador léxico sem ter alguma técnica sistemática para lidar com a complexidade.

As técnicas que usaremos para isso são relacionadas a uma classe de linguagens formais conhecida como *linguagens regulares*. Essas técnicas são fundamentadas em uma teoria bem desenvolvida, e contam com ferramentas que automatizam a maior parte do processo de análise léxica.

2.2 Linguagens Regulares e Expressões Regulares

As linguagens regulares são um tipo de linguagem formal que são frequentemente utilizadas para representar padrões simples de texto. Uma técnica de representação muito utilizada para as linguagens regulares são as chamadas *expressões regulares*. Bibliotecas que dão suporte ao uso de expressões regulares estão disponíveis na maioria das linguagens de programação e são muito usadas para busca em textos e para validação de entrada textual (para formulários de entrada de dados, por exemplo).

Uma outra técnica de representação usada para linguagens regulares são os *autômatos finitos*. Autômatos finitos e expressões regulares são equivalentes, ou seja, todo padrão que pode ser representado por uma técnica também pode ser representada pela outra. Os autômatos finitos podem ser utilizados para organizar os padrões léxicos de uma linguagem, facilitando a implementação direta de um analisador léxico para ela. Ou seja, com os autômatos finitos podemos criar analisadores léxicos para linguagens mais complexas, e de maneira mais sistemática e confiável do que vimos no exemplo da linguagem de expressões.

Para criar um analisador léxico dessa forma devemos definir os autômatos finitos que representam os padrões associados a cada tipo de *token*, depois combinar esses autômatos em um único autômato, e então implementar o autômato finito resultante como um programa. Mais detalhes sobre como fazer isso podem ser encontrados em outros livros sobre compiladores, por exemplo o famoso “livro

do dragão" (Compiladores: Princípios, Técnicas e Ferramentas, 2ª edição, de Aho et al., editora Pearson/Addison-Wesley).

Aqui vamos usar uma abordagem mais automatizada, criando analisadores léxicos a partir de ferramentas chamadas de *geradores de analisadores léxicos*. Esses geradores recebem como entrada uma especificação dos padrões que definem cada tipo de *token*, e criam na saída o código-fonte do analisador léxico. Criar analisadores usando um gerador é prático e temos um certo nível de garantia que o código gerado estará correto. Para usar um gerador, no entanto, é preciso saber como representar os padrões que definem tipos de *tokens* da linguagem como expressões regulares.

2.2.1 Expressões Regulares

As expressões regulares descrevem padrões simples de texto de forma compacta e sem ambiguidade. Por exemplo, o padrão que descreve todas as *strings* formadas com caracteres `a` e `b` que começam com `a` e terminam com `b` pode ser escrito como a expressão regular `a(a|b)*b` (a construção dessa expressão será explicada em breve).

Existem várias sintaxes e representações diferentes para expressões regulares, dependendo da linguagem ou biblioteca utilizada. Como vamos utilizar o gerador de analisadores flex, usaremos aqui a sintaxe usada nessa ferramenta.

2.2.1.1 Expressões básicas

Cada expressão regular (ER) é uma *string* que representa um conjunto de *strings*; também podemos dizer que uma ER representa um *padrão* que é satisfeito por um conjunto de *strings*.

A maioria dos caracteres representam eles mesmos em uma expressão regular. Por exemplo, o caractere `a` em uma ER representa o próprio caractere `a`. A ER `a` representa um padrão que poderia ser descrito em português como "o conjunto de *strings* que possuem um caractere `a`". Obviamente só existe uma *string* dessa forma: a *string* `"a"`. Colocando um padrão após o outro realiza a *concatenação* dos padrões. Começando com caracteres simples, se juntarmos um `a` e um `b` formamos a expressão `ab`, que representa a *string* que contém um `a` seguido por um `b`, ou seja, a *string* `"ab"`.

Mas o poder das Expressões Regulares vem de alguns caracteres que não representam eles mesmos; esses são *caracteres especiais*. Um caractere especial bastante usado é o `*`, que representa zero ou mais repetições de um padrão. Por exemplo, a expressão `a*` representa *strings* com zero ou mais caracteres `a`. A *string* vazia satisfaz esse padrão e corresponde a zero repetições; outras *strings* satisfeitas pelo padrão são `"a"`, `"aa"`, `"aaa"`, etc. O asterisco representa zero ou mais repetições do padrão que vem antes, não só de um caractere: a expressão `(ab)*` representa , `"ab"`, `"abab"`, `"abababab"`, etc. Mas pelas regras de precedência das expressões, `ab*` é o mesmo que `a(b*)`, que representa um `a` seguido por zero ou mais caracteres `b`, e não é igual a `(ab)*`.

Outro caractere especial importante é a barra vertical `|`, que representa opções nas partes de um padrão. Por exemplo `a|b` representa `a` ou `b`, ou seja, as *strings* `"a"` e `"b"`.

Isso nos leva ao exemplo apresentado antes: `a(a|b)*b` é uma expressão regular formada por três partes concatenadas em sequência: `a`, depois `(a|b)*` e por fim `b`. Isso significa que uma *string* que satisfaz essa expressão deve começar com um caractere `a`, seguido por caracteres que satisfazem o padrão `(a|b)*` e terminando com um caractere `b`. O padrão `(a|b)*` é satisfeito por zero ou mais repetições do padrão `(a|b)`, que por sua vez é um padrão que é satisfeito por caracteres `a` ou `b`. Ou seja, `(a|b)*` é um padrão que representa zero ou mais repetições de caracteres `a` ou `b`. Alguns exemplos de cadeias que são representadas pela expressão `a(a|b)*b`:

- `"ab"` (zero repetições do padrão interno `(a|b)*`)
- `"aab"`
- `"abb"`
- `"aabbbb"`
- `"abbaabab"`

2.2.1.2 Caracteres especiais + e ?

Ja vimos que o caractere especial `*` representa zero ou mais repetições de um padrão. O caractere especial + é similar, mas representa uma ou mais repetições; a única diferença é que o caractere + causa a obrigatoriedade de pelo menos uma repetição do padrão. A expressão `a+` representa as *strings* `"a"`, `"aa"`, `"aaa"`, etc., sem incluir a *string* vazia.

O caractere especial `?` representa partes opcionais em um padrão, ou seja, zero ou uma repetição de um determinado padrão. A expressão `b?a+` representa *strings* com uma ou mais repetições de `a`, podendo começar opcionalmente com um `b`.

2.2.1.3 Classes de caracteres, intervalos e negação

As classes de caracteres são uma notação adicional para representar opções de um caractere em um padrão. A classe `[abc]` representa apenas um caractere, que pode ser `a`, `b` ou `c`. Isso é o mesmo que a expressão `(a|b|c)`, e a notação de classes é apenas um atalho, principalmente quando existem várias opções.

A expressão `[0123456789]` representa um caractere que é um dígito numérico. Adicionando um caractere de repetição temos `[0123456789]+`, que representa *strings* contendo um ou mais dígitos. Essas são exatamente as *strings*, como `"145"` ou `"017"`, que representam constantes inteiras.

Quando uma classe inclui vários caracteres em uma sequência, como o exemplo anterior, podemos usar *intervalos* para tornar as expressões mais compactas. Por exemplo, a expressão `[0123456789]` pode ser igualmente representada pelo intervalo `[0-9]`. A expressão `[a-z]` representa uma letra minúscula. Podemos usar vários intervalos em uma classe. Por exemplo, `[A-Za-z]` representa uma letra maiúscula ou minúscula, e `[0-9A-Za-z]` representa um dígito ou letra. Note que cada classe ainda representa apenas um caractere; os intervalos apenas criam novas opções para esse caractere.

Algumas classes especiais podem ser usadas como abreviações. Por exemplo `[:alpha:]` representa um caractere alfabético (ou seja, é o mesmo que `[A-Za-z]`), e `[:alnum:]` representa um caractere alfabético ou um dígito. Outras classes especiais úteis são `[:space:]` para caracteres de espaço em branco, `[:upper:]` para caracteres maiúsculos e `[:lower:]` para caracteres minúsculos. Existe a classe especial `[:digit:]` para dígitos, mas em geral é mais compacto escrever `[0-9]`. É importante lembrar que essas classes especiais, assim como os intervalos, só podem ser usados dentro de classes de caracteres, ou seja, não é possível ter uma expressão que seja apenas `[:alpha:]`; é preciso colocar a classe especial `[:alpha:]` dentro de uma classe, resultando na expressão `[[:alpha:]]`, que representa um caractere que pode ser qualquer letra (maiúscula ou minúscula).

Uma outra notação útil com classes é a negação. Usar um caractere `^` no começo de uma classe representa "caracteres que não estão na classe". Por exemplo, `[^0-9]` representa um caractere que não é um dígito de `0` a `9`. A negação também pode ser usada com classes especiais: `[:^alnum:]` representa um caractere que não é uma letra ou dígito.

2.2.1.4 Metacaracteres e sequências de escape

Um outro tipo de caracteres especiais são os *metacaracteres*. Um metacaractere é um caractere especial que pode representar outros caracteres. O exemplo mais simples é o metacaractere `.`, que pode representar qualquer caractere. A expressão `a.*b` representa *strings* que começam com `a`, terminam com `b` e podem ter qualquer número de outros caracteres no meio, por exemplo `"a0x13b"`.

As sequências de escape são iguais as que existem na linguagem C: `\n` representa um caractere de nova linha, `\t` um caractere de tabulação, etc. A barra invertida também pode ser usada para desativar a interpretação especial de um caractere especial. Por exemplo, se quisermos um caractere + em uma expressão regular que representa o símbolo de soma, e não a repetição de uma ou mais vezes, devemos usar `\+`.

2.2.1.5 Outras características

Além das possibilidades de repetição que vimos até agora (zero ou mais vezes, uma ou mais vezes, zero ou uma vez), é possível na notação do flex ser mais específico no número de repetições. Se `r` é um padrão, `r{2}` representa exatamente duas repetições do padrão, `r{2,}` representa duas ou mais repetições, e `r{2, 5}` representa um número de repetições entre duas e cinco, inclusive. A expressão `(la){3}` representa três repetições de `la`, ou seja, a *string* `"lalala"`; e a expressão `(la){1,3}` representa as *strings* `"la"`, `"lala"` e `"lalala"`.

Não vamos tratar aqui de todos os detalhes das expressões regulares no flex, mas eles podem ser consultados no manual da ferramenta em http://flex.sourceforge.net/manual/Patterns.html

2.2.1.6 Alguns exemplos

Agora que introduzimos a maior parte das características das expressões regulares no flex, veremos alguns exemplos de padrões descritos usando essa notação. Depois veremos outros exemplos diretamente ligados à análise léxica de linguagens de programação.

- `[0-9]{3}\.[0-9]{3}\.[0-9]{3}\-[0-9]{2}` é um padrão que descreve os números de CPF: três dígitos (`[0-9]{3}`) seguidos por um ponto (`\.`), depois mais três dígitos e um ponto, depois mais três dígitos, um hífen (`\-`) e finalmente dois dígitos.

- `[0-9]{2}\/[0-9]{2}\/[0-9]{4}` é um padrão que descreve datas no formato DD/MM/AAAA com dois dígitos para dia e mês, e quatro dígitos para o ano. É preciso usar uma barra invertida `\` para incluir a barra `/` no padrão, caso contrário a barra seria interpretada como um caractere especial; no padrão, isso ocorre como `\/`. Esse padrão não verifica se a data é válida (uma *string* como `"33/55/2033"` satisfaz o padrão).

- `[A-Z]{3}-[0-9]{4}` descreve placas de carro no Brasil, começando com três letras maiúsculas (`[A-Z]{3}`) seguidas por um hífen e quatro dígitos.

- Os endereços de email seguem um conjunto de várias regras que estabelecem que caracteres podem ser usados (a maioria das regras pode ser encontrada nos RFCs 2821 e 2822). Um padrão simplificado para endereços de email pode ser o seguinte: `[[:alnum:]\._]+@[[:alnum:]]+\.[[:alnum:]]+` começando com um ou mais caracteres que podem ser letras, números, pontos e *underscore* (a parte `[[:alnum:]\._]+`), seguidos pela arroba, depois uma ou mais letras ou dígitos, seguindo por um ponto e mais um grupo de uma ou mais letras ou dígitos. Esse padrão descreve um endereço de email simples como

`nome@dominio.com`, mas não um endereço que tenha mais de um ponto após a arroba (por exemplo um email do Centro de Informática da UFPB, da forma `nome@ci.ufpb.br`).

Agora que sabemos como especificar padrões de texto usando expressões regulares no flex, podemos usá-lo para gerar analisadores léxicos.

2.3 Geradores de Analisadores Léxicos

Um gerador de analisadores léxicos é um programa que recebe como entrada a especificação léxica para uma linguagem, e que produz como saída um programa que faz a análise léxica para essa linguagem. Como vimos anteriormente, é possível escrever o código de um analisador léxico sem o uso de uma ferramenta, mas usar um gerador de analisadores é menos trabalhoso e tem maior garantia de gerar um analisador léxico correto. A Figura 2.1 [21] mostra um diagrama de blocos que descreve o uso de um gerador de analisadores (no caso o flex). O gerador recebe uma especificação da estrutura léxica na entrada, e gera um programa analisador (no caso do flex, um programa na linguagem C). Este programa, quando executado, recebe como entrada os caracteres do programa de entrada, e produz como saída a sequência de *tokens* correspondentes.

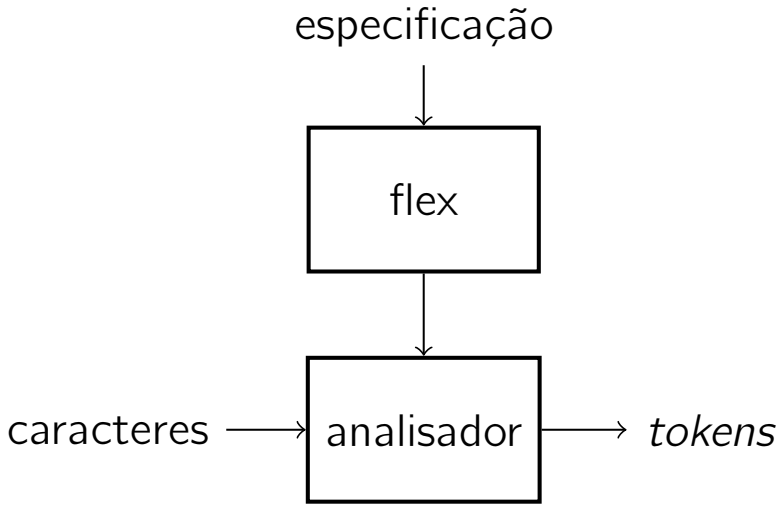

Figura 2.1: Uso de um gerador de analisadores léxicos (flex).

2.4 Uso do flex

No sistema Unix original foi criado um gerador de analisadores léxicos chamado lex, um dos primeiros geradores desse tipo. O projeto GNU criou o flex como uma versão do lex com licença de *software* livre. O flex, assim como o lex original, é um gerador de analisadores léxicos que gera analisadores na linguagem C, e possui versões compatíveis nos principais sistemas operacionais atuais.

Como mostrado na Figura 2.1 [21], o flex recebe como entrada um arquivo de especificação e produz, na saída, um programa na linguagem C que implementa o analisador léxico que segue a especificação dada. Para usar o flex primeiramente precisamos saber como escrever a especificação léxica da linguagem no formato esperado pela ferramenta.

Instalando o flex

Como é comum para a maioria das ferramentas, para usar o flex é preciso instalar o programa antes. Essa instalação geralmente é simples, ou mesmo desnecessária por já vir instalado, dependendo do sistema operacional utilizado:

- Em sistemas Mac OS X já vem uma versão do flex instalada. Embora não seja uma das versões mais recentes da ferramenta, isso não é um problema para o nosso uso.
- Em sistemas Linux o flex deve estar disponível como um pacote no sistema gerenciador de pacotes da distribuição. Por exemplo, no Ubuntu basta fazer `apt-get install flex` (possivelmente com o uso do `sudo`) para instalar.
- No Windows o mais adequado é instalar usando um instalador criado especificamente para esse sistema operacional, que pode ser encontrado no endereço http://gnuwin32.sourceforge.net/packages/flex.htm

2.4.1 Formato da entrada

A parte principal da especificação léxica no flex é um conjunto de regras. Cada regra é composta por duas partes: um padrão e uma ação; o padrão é uma expressão regular que descreve um determinado tipo de *tokens* da linguagem, e a ação determina o que fazer quando encontrar o padrão correspondente. Para um compilador, a maioria das ações vai simplesmente criar um *token* para o lexema encontrado, como veremos adiante.

O formato do arquivo de especificação do flex é dividido em três partes separadas por uma linha contendo os caracteres `%%`, da seguinte forma:

Formato de um arquivo de especificação do flex

```
definições
%%
regras
%%
código
```

A única parte obrigatória do arquivo são as regras. As definições permitem dar nomes a expressões regulares, o que é útil quando uma determinada expressão regular aparece como parte de vários padrões, ou como forma de documentação, para deixar mais claro o que significam as partes de um padrão complexo. Veremos exemplos de uso das definições mais adiante. No começo do arquivo tamém podem ser especificadas algumas opções que alteram o comportamento do analisador gerado.

A terceira parte do arquivo pode conter código em linguagem C que será adicionado, sem alterações, ao programa C gerado pelo flex. Como a saída do flex é um programa em linguagem C, isso permite que o criador do arquivo de especificação adicione funções ou variáveis ao analisador gerado. Geralmente a parte de código é útil para definir funções auxiliares que podem ser usadas pelas ações.

Já entendemos a maior parte do que é necessário para usar o flex, mas alguns detalhes só ficam claros com alguns exemplos. Vamos começar com um exemplo de especificação bastante simples.

2.4.2 Uma especificação simples do flex

Para exemplificar o uso do flex, vamos ver um exemplo de especificação simples e auto-contida que também serve como uma forma de testar os padrões do flex.

Código fonte code/cap2/simples.ll

Especificação simples para o flex

```
%option noyywrap                                    ❶

CPF [0-9]{3}\.[0-9]{3}\.[0-9]{3}-[0-9]{2}   ❷
EMAIL [[:alnum:]\._]+@[[:alnum:]]+\.[[:alnum:]]+

%%

{CPF}   { printf("CPF\n"); }                ❸
{EMAIL} { printf("EMAIL\n"); }
.       { printf("Caractere nao reconhecido\n"); }

%%

// funcao principal que chama o analisador  ❹
int main()
{
  yylex();
}
```

❶ Opção para não precisar criar uma função `yywrap`

❷ Definições

❸ Regras usando as definições

❹ Código: função `main` para o programa do analisador

A especificação contém as três partes: definições, regras e código. Antes das definições está especificada a opção `noyywrap`, que simplifica a especificação (sem essa opção seria necessário escrever uma função chamada `yywrap`). São definidas duas expressões, `CPF` e `EMAIL`.

Na parte de regras são descritas três, as duas primeiras determinam o que o programa deve fazer quando encontra um *token* que satisfaz os padrões `CPF` e `EMAIL`, e a terceira regra determina o que o programa deve fazer com *tokens* que não satisfazem nenhum dos dois.

A primeira regra é:

```
{CPF}   { printf("CPF\n"); }
```

Uma ação sempre tem duas partes, o padrão e a ação, separados por espaços em branco:

```
padrão ação
```

`CPF` entre chaves especifica que deve ser usada a definição com esse nome, ao invés de tratar os caracteres como representando eles mesmos (o padrão `CPF`, sem chaves, seria satisfeito apenas pela *string* `"CPF"`). A regra é um trecho de código C que será executado casa o padrão seja satisfeito pelo *token*. Nesse caso apenas é impressa a *string* `"CPF"`, para que o usuário veja o tipo de *token* que foi determinado.

O uso de definições é opcional. A seguinte regra tem exatamente o mesmo efeito que a regra anterior para números de CPF:

```
[0-9]{3}\.[0-9]{3}\.[0-9]{3}-[0-9]{2}    { printf("CPF\n"); }
```

Veja que nesse caso não é preciso usar chaves ao redor do padrão. O uso de uma definição torna a especificação muito mais legível, deixando claro para o leitor o que a expressão regular representa.

A regra para endereços de email segue os mesmos princípios. A terceira regra é:

```
.       { printf("Token nao reconhecido\n"); }
```

Como já vimos, o ponto é um metacaractere no flex que representa qualquer caracter. O padrão `.` significa "qualquer caractere", ou seja, esse padrão reconhece qualquer caractere não reconhecido pelos padrões anteriores. É importante entender como as regras do flex são processadas: o analisador testa a *string* atual com todos os padrões da especificação, procurando ver que padrões são satisfeitos pela *string*. Se mais de um padrão é satisfeito pel *string* ou parte dela, o analisador vai escolher o padrão que é satisfeito pelo maior número de caracteres.

Usando as regras do exemplo atual, digamos que a *string* atual seja um CPF. Essa *string* satisfaz o padrão para números de CPF na primeira regra do arquivo de especificação, mas o primeiro caractere da *string*, que é um número, também satisfaz a última regra (o padrão `.`), pois esse padrão satisfaz qualquer caractere. Entre as duas regras ativadas, a regra do CPF é casada com todos os caracteres da *string* atual, enquanto que a regra do ponto só é casada com o primeiro caractere da *string* atual. Portanto, a regra do CPF é casada com o maior número de caracteres, e essa regra é escolhida. Mas se a *string* atual for uma sequência de três dígitos como `123`, o único padrão que é satisfeito é o último, do ponto, que aceita qualquer caractere, e nesse caso o analisador imprime mensagens de erro (o padrão do ponto é satisfeito três vezes por essa *string*, já que o ponto representa apenas um caractere).

Como o flex casa a entrada com os padrões

O funcionamento geral do flex é determinado pelos padrões que estão presentes nas regras especificadas para o analisador. Para a sequência de caracteres da entrada, o analisador gerado pelo flex tenta casar os caracteres de entrada, ou uma parte inicial deles, com algum padrão nas regras.

Se apenas um padrão é casado com os caracteres iniciais da sequência, a regra de onde vem o padrão é *disparada*, ou seja, a ação da regra é executada pelo analisador. Se nenhum padrão for casado com os caracteres atuais, o analisador gerado executa uma regra padrão inserida pelo flex. A regra padrão simplesmente imprime na saída os caracteres não reconhecidos por nenhum padrão.

Se mais de um padrão for satisfeito por uma sequência inicial dos caracteres atuais, o analisador escolhe o padrão que é casado com o maior número de caracteres e dispara a regra desse padrão. Se vários padrões casam com o mesmo número de caracteres da sequência atual, o analisador escolhe aquele que aparece primeiro no arquivo de especificação. Isso significa que a ordem das regras no arquivo de especificação é importante.

Se uma regra é disparada ao casar os caracteres atuais com algum padrão, os caracteres restantes que não foram casados com o padrão permanecem guardados para uma próxima vez que o analisador for chamado. Por exemplo, se a entrada é a *string* `123456` e o único padrão do analisador representa cadeias de três dígitos, a primeira chamada ao analisador vai casar os caracteres `123` e disparar a regra associada, deixando os caracteres `456` no analisador, para uma próxima chamada. Se o analisador for chamado novamente, a regra vai casar com os caracteres `456` e disparar novamente. Dessa forma o analisador pode atuar em um *token* de cada vez.

O código nesse caso define uma função `main`, para que o código C gerado pelo flex possa ser executado diretamente. A função `main` apenas chama a função `yylex()` que é a função principal do analisador léxico criado pelo flex. Todo arquivo C gerado pelo flex contém uma função `yylex`. O comportamento dessa função pode ser alterado de várias formas que veremos adiante, mas se chamada diretamente, da forma que fazemos nesse exemplo, ela funciona da seguinte forma: recebe *tokens* na entrada padrão (lendo do teclado) e executa as ações associadas para cada regra que é satisfeita; esse processo continua até que o fim de arquivo seja encontrado. Isso funciona bem como um forma de testar os padrões usados na especificação.

Com o arquivo de especificação acima, podemos gerar o código C correspondente chamando a ferramenta flex na linha de comando. Por convenção, os arquivos de especificação do flex têm extensão `.ll`. Depois de gerar um arquivo com código C, este pode ser compilado e executado diretamente (já que ele possui uma função `main`). Em um sistema Unix, a sequência de comandos é a seguinte:

Geração e compilação do arquivo C

```
Sandman:cap2 andrei$ flex -o simples.c simples.ll
Sandman:cap2 andrei$ gcc -o simples simples.c
```

Depois disso, o executável `simples` vai funcionar como descrito: esperando entrada pelo teclado e imprimindo os tipos de *tokens* reconhecidos:

Exemplo de uso do analisador

```
Sandman:cap2 andrei$ ./simples
111.222.333-99
CPF

nome@mail.com
EMAIL

123
Caractere nao reconhecido
Caractere nao reconhecido
Caractere nao reconhecido
```

Para terminar o teste, deve-se digitar o caractere de fim de arquivo (em sistemas Unix o fim de arquivo é entrado com Ctrl+D, enquanto em sistemas Windows deve-se usar Ctrl+Z).

Essa especificação pode ser usada para testar outros padrões. Ao mudar a especificação, deve-se gerar novamente o arquivo C usando o flex e compilar o arquivo C gerado.

Em um compilador geralmente queremos que a entrada seja lida de um arquivo, e precisamos gerar os *tokens* para cada lexema, não simplesmente imprimir o tipo de cada *token* encontrado. Veremos nos próximos exemplos como fazer isso.

2.4.3 Analisador léxico para expressões usando flex

Vimos anteriormente um analisador léxico para uma linguagem de expressões criado diretamente na linguagem C, sem uso de gerador. Nosso próximo exemplo é um analisador para a mesma linguagem, mas agora usando flex. Isso serve a dois propósitos: o primeiro é mostrar mais algumas características do uso do flex; o segundo é comparar o esforço necessário para criar um analisador com e sem usar um gerador como o flex.

O analisador é composto pelo arquivo de especificação, `exp.ll`, um arquivo C que chama o analisador léxico (`exp_flex.c`), e um arquivo de cabeçalho com definições (`exp_tokens.h`).

O arquivo de cabeçalho contém as definições de tipos e constantes, iguais ao analisador que foi mostrado anteriormente:

Código fonte code/cap2/exp_flex/exp_tokens.h

O conteúdo principal desse arquivo é:

Arquivo de cabeçalho para analisador léxico de expressões

```
// constantes booleanas
#define TRUE            1
#define FALSE           0

// constantes para tipo de token
#define TOK_NUM         0
#define TOK_OP          1
#define TOK_PONT        2
#define TOK_ERRO        3

// constantes para valores de operadores
#define SOMA            0
#define SUB             1
#define MULT            2
#define DIV             3

// constantes para valores de pontuacao (parenteses)
#define PARESQ          0
#define PARDIR          1

// estrutura que representa um token
typedef struct
{
  int tipo;
  int valor;
} Token;

// funcao para criar um token
extern Token *token();

// funcao principal do analisador lexico
extern Token *yylex();
```

Além das constantes já vistas para tipo e valor do *token*, temos um novo tipo de *token* declarada e protótipos para duas funções. O novo tipo é o `TOK_ERRO`, que sinaliza um erro na análise léxica. Esse tipo é usado quando o analisador recebe uma sequência de caracteres que não reconhece como *token* da linguagem. As duas funções declaradas são a função principal do analisador, e uma função para criar *tokens* que é usada pelo analisador. A função principal desse analisador retorna um ponteiro para uma estrutura `Token`, e por isso ela deve ser declarada de outra forma (a função `yylex` padrão retorna um inteiro, como vimos antes).

Em seguida vamos ver o arquivo de especificação para essa linguagem, que contém algumas novidades em relação ao que vimos antes:

Código fonte code/cap2/exp_flex/exp.ll

Arquivo de especificação do flex para a linguagem de expressões

```
%option noyywrap                                                    ❶
%option nodefault
%option outfile="lexer.c" header-file="lexer.h"

%top {                                                              ❷
#include "exp_tokens.h"
}

NUM [0-9]+

%%

[[:space:]] { }  /* ignora espacos */

{NUM}    { return token(TOK_NUM,  atoi(yytext)); }                  ❸
\+       { return token(TOK_OP,   SOMA);    }
-        { return token(TOK_OP,   SUB);     }
\*       { return token(TOK_OP,   MULT);    }
\/       { return token(TOK_OP,   DIV);     }
\(       { return token(TOK_PONT, PARESQ);  }
\)       { return token(TOK_PONT, PARDIR);  }

.        { return token(TOK_ERRO, 0);       }  /* erro p/ token  ↩
   desconhecido */

%%

// variavel global para um token                                    ❹
Token tok;

Token * token(int tipo, int valor)
{
  tok.tipo = tipo;
  tok.valor = valor;
  return &tok;
}
```

❶ Opções: sem regra *default*, nomes dos arquivos gerados

❷ Trecho de código para incluir no topo do arquivo gerado

❸ Uso do lexema casado pelo padrão: a variável `yytext`

❹ Código para incluir no final do arquivo gerado

O arquivo de especificação usa todas as três partes: definições (e opções), regras e código. Na parte de definição são incluídas as opções para não precisar da função `yywrap` e opções para selecionar o nome dos arquivos de saída. No exemplo anterior o nome do arquivo de saída foi selecionado na linha de comando; nesta especificação usamos opções para não só selecionar o nome do arquivo C

gerado, mas também garantir que será gerado um arquivo de cabeçalho com definições do analisador léxico (nesse caso, o arquivo `lexer.h`).

A opção `nodefault` deve ser usada para evitar que o flex inclua uma regra padrão (*default*) se nenhuma outra regra for satisfeita. A regra padrão do flex apenas mostra na saída os caracteres que não forem reconhecidos em alguma regra. Isso significa que erros léxicos no programa de entrada não serão reconhecidos, e que o analisador vai gerar saída desnecessária.

Apenas uma definição é criada, a definição `NUM` para constantes numéricas. A primeira regra ignora quaisquer caracteres de espaço (espaços, tabulações, caracteres de nova linha). Essa regra tem uma ação vazia, o que indica que os espaços devem ser apenas ignorados. As outras regras seguem a estrutura léxica da linguagem de expressões, que já vimos antes. O único cuidado adicional é que muitos dos caractere usados na linguagem são caracteres especiais no flex (`+`, `*`, barra e os parênteses) e portanto precisam ser incluídos no padrão com uma contrabarra antes. Cada ação apenas cria um *token* com o tipo e o valor adequados, usando a função `token` que veremos a seguir.

A regra que cria *tokens* de tipo número precisa obter o valor do número, e isso depende da sequência de caracteres que foi casada com o padrão. Ou seja, o analisador precisa ter acesso ao lexema que foi identificado para gerar o *token*. Analisadores gerados pelo flex incluem uma variável chamada `yytext` que guarda os caracteres casados pelo padrão da regra que foi disparada. Por isso, a regra para *tokens* de tipo número obtem o valor chamando a função `atoi` da linguagem C na cadeia `yytext`, como visto na regra:

```
{NUM}   { return token(TOK_NUM,  atoi(yytext)); }
```

De resto, a última regra usa o padrão com um ponto para capturar erros léxicos na entrada.

A seção de código inclui uma variável e uma função que serão incluídas no analisador gerado. A variável `tok` serve para guardar o *token* atual, e a função `token` serve para guardar os dados de um *token* e retorná-lo para a parte do programa que chama o analisador.

O arquivo C `exp_flex.c` contém o programa principal que recebe entrada do teclado e chama o analisador léxico. Esse arquivo contém funções `operador_str` e `imprime_token` que são idênticas às funções no arquivo `exp_lexer.c` do analisador léxico anterior. A função principal do programa em `exp_flex.c` é mostrada abaixo:

Código fonte code/cap2/exp_flex/exp_flex.c

Função principal do progra em exp_flex.c

```
int main(int argc, char **argv)
{
  char  entrada[200];
  Token *tok;

  printf("Analise Lexica para Expressoes\n");

  printf("Expressao: ");
  fgets(entrada, 200, stdin);

  inicializa_analise(entrada);

  printf("\n===== Analise =====\n");

  tok = proximo_token();
  while (tok != NULL) {
```

```
    imprime_token(tok);
    tok = proximo_token();
  }

  printf("\n");

  return 0;
}
```

A função principal é quase igual ao analisador criado diretamente, mas as funções `inicializa_analise` e `proximo_token` são diferentes. A função `proximo_token` apenas chama a função principal do analisador gerado pelo flex, `yylex`:

Função para obter o próximo token

```
Token *proximo_token()
{
  return yylex();
}
```

E a função `inicializa_analise` chama uma função do analisador gerado que configura a análise léxica para ler de uma *string* ao invés da entrada padrão. Para isso é preciso usar uma variável do tipo `YY_BUFFER_STATE`, que é um tipo declarado no analisador gerado.

Função que inicializa a análise léxica

```
YY_BUFFER_STATE buffer;

void inicializa_analise(char *str)
{
  buffer = yy_scan_string(str);
}
```

Para compilar o analisador são necessários dois passos, como antes: gerar o arquivo C do analisador usando o flex, e então compilar os arquivos C usando o compilador C. Para gerar o analisador usando o flex, é preciso usar uma opção para determinar uma declaração diferente para a função principal do analisador, `yylex`:

```
flex -DYY_DECL="Token * yylex()" exp.ll
```

Isso vai gerar os arquivos `lexer.c` e `lexer.h`. O passo seguinte é compilar o arquivo `lexer.c` juntamente com o arquivo principal `exp_flex.c`. Usando o `gcc` como compilador, a linha de comando seria:

```
gcc -o exp lexer.c exp_flex.c
```

Isso gera um executável de nome `exp`. Quando executado, o programa funciona praticamente da mesma forma que o analisador criado diretamente para a mesma linguagem de expressões; a única diferença é o tratamento de erros. O analisador que usa o flex sinaliza os erros obtidos e continua com a análise.

Esse exemplo demonstra quase tudo que precisamos para fazer a análise léxica de uma linguagem de programação. O único detalhe que falta é saber como ler a entrada de um arquivo ao invés de uma *string* ou da entrada padrão.

2.4.4 Lendo um arquivo de entrada

A função `yylex` gerada pelo flex normalmente lê sua entrada de um arquivo, a não ser que seja usada a função `yy_scan_string` como no exemplo anterior. O analisador gerado pelo flex contém uma variável chamada `yyin` que é um ponteiro para o arquivo de entrada usado pelo analisador. Normalmente essa variável é igual à variável global `stdin` da linguagem C, ou seja, a entrada padrão. Para fazer que o analisador leia de um arquivo ao invés da entrada padrão, é necessário mudar o valor dessa variável para o arquivo desejado.

O código necessário é basicamente o seguinte:

Exemplo de como fazer o analisador ler de um arquivo

```
int inicializa_analise(char *nome)
{
  FILE *f = fopen(nome, "r");

  if (f == NULL)
    return FALSE;

  // arquivo aberto com sucesso, direcionar analisador
  yyin = f;
}
```

Ou seja, deve-se abrir o arquivo desejado usando `fopen`, e depois atribuir a variável `yyin` do analisador para apontar para o mesmo arquivo aberto. A partir daí, chamadas à função `yylex` vão ler os caracteres usados no analisador do arquivo, ao invés da entrada padrão. É uma boa prática fechar o arquivo aberto durante a inicialização ao final da análise léxica. Vamos ver um exemplo de analisador que lê a entrada a partir de um arquivo a seguir.

2.5 Análise Léxica de uma Linguagem de Programação

Os exemplos vistos até agora foram preparação para sabermos como criar um analisador léxico para uma linguagem de programação. Vamos usar o flex como maneira mais fácil de criar um analisador do que escrever o código diretamente. Para especificar os padrões que definem os vários tipos de *tokens* no flex, precisamos usar expressões regulares. Já vimos como escrever um arquivo de especificação do flex para criar um analisador léxico, e como trabalhar com o código C gerado pelo flex. Nesta seção vamos juntar todas as peças e criar um analisador léxico para uma pequena linguagem de programação, uma versão simplificada da linguagem C.

2.5.1 A Linguagem Mini C

A linguagem que vamos usar como exemplo é uma simplificação da linguagem de programação C, chamada aqui de Mini C. A ideia é que todo programa Mini C seja um programa C válido, mas muitas características da linguagem C não estão disponíveis em Mini C.

Um programa Mini C é um conjunto de declarações de funções. Cada função é uma sequência de comandos. Alguns comandos podem conter expressões. Variáveis podem ser declaradas, apenas do tipo `int`. As estruturas de controle são apenas o condicional `if` e o laço `while`. As expressões que podem ser formadas na linguagem também são simplificadas.

Discutiremos mais sobre a sintaxe da linguagem Mini C no próximo capítulo. Aqui o que interessa é a estrutura léxica da linguagem, ou seja, que tipos de *token* ocorrem na linguagem e quais devem ser os valores associados a eles. Já que o analisador léxico é encarregado de retirar os comentários do código-fonte, também nesta fase precisamos definir a sintaxe para comentários. A linguagem C tem atualmente dois tipos de comentários: os comentários que podem se extender por múltiplas linhas, delimitados por `/*` e `*/`, e os comentários de linha única, que começam com `//` e vão até o final da linha. Vamos adotar esse último tipo de comentário na linguagem Mini C, pois ele é um pouco mais simples de tratar no analisador léxico.

Os tipos de *tokens* da linguagem Mini C são:

- identificadores (nomes de variáveis e funções)
- palavras-chave (`if`, `else`, `while`, `return` e `printf`)
- constantes numéricas
- *strings* (delimitadas por aspas duplas `"`)
- pontuação (parênteses, chaves, etc.)

Os identificadores devem ser iniciados por uma letra, seguida de zero ou mais letras ou dígitos. As constantes numéricas são formadas por um ou mais dígitos, e como pontuação incluímos as chaves `{` e `}`, os parênteses `(` e `)`, a vírgula e o ponto-e-vírgula. Nos operadores estão incluídos operadores aritméticos (as quatro operações básicas), comparações (a operação de menor e de igualdade), e dois operadores lógicos (E-lógico e negação).

Como a ideia é que os programas Mini C sejam compatíveis com compiladores C, todo programa Mini C deve poder ser compilado como se fosse um programa C. Os programas Mini C podem usar `printf` para imprimir na tela (em Mini C, `printf` é uma palavra-chave, não uma função como em C), e portanto para que isso não crie problemas com compiladores C, é preciso que os programas Mini C tenham a linha `#include <stdio.h>` no começo. Por isso, um *token* especial na linguagem Mini C é o chamado de *prólogo*, que é a *string* `#include <stdio.h>`.

Nos capítulos seguintes veremos mais detalhes sobre a linguagem Mini C, mas neste capítulo vamos apenas trabalhar a estrutura léxica da linguagem.

2.5.2 O analisador léxico para a linguagem Mini C

Aqui veremos um analisador léxico para a linguagem Mini C criado com o flex. Quase todas as características do flex usadas aqui já foram apresentadas antes, mas veremos algumas novidades. A maior novidade no analisador para a linguagem Mini C é a necessidade de usar tabelas para armazenar as *strings* e os nomes de identificadores que ocorrem no programa.

Constantes para os tipos e valores de *tokens* são definidos no arquivo de cabeçalho `minic_tokens.h`.

Código fonte code/cap2/minic/minic_tokens.h

A parte mais importante do conteúdo deste arquivo é mostrada abaixo:

Definição de constantes para o analisador léxico

```
// Tipos de token
#define TOK_PCHAVE                              1
#define TOK_ID                                  4
#define TOK_NUM                                 5
#define TOK_PONT                                6
#define TOK_OP                                  7
#define TOK_STRING                              8
#define TOK_PROLOGO                             9
#define TOK_ERRO                                100

// valores para palavra-chave
#define PC_IF                                   0
#define PC_ELSE                                 1
#define PC_WHILE                                2
#define PC_RETURN                               3
#define PC_PRINTF                               4

// valores para pontuacao
#define PARESQ                                  1
#define PARDIR                                  2
#define CHVESQ                                  3
#define CHVDIR                                  4
#define VIRG                                    5
#define PNTVIRG                                 6

// valores para operadores
#define SOMA                                    1
#define SUB                                     2
#define MULT                                    3
#define DIV                                     4
#define MENOR                                   5
#define IGUAL                                   6
#define AND                                     7
#define NOT                                     8
#define ATRIB                                   9

// tipos
typedef struct
{
  int tipo;
  int valor;
} Token;
```

Agora vamos analisar o arquivo de especificação do flex para a linguagem Mini C, uma seção de cada vez. O arquivo completo pode ser encontrado no endereço abaixo.

Código fonte code/cap2/minic/lex.ll

A seção inicial contém opções e definições, além de trechos de código que são adicionados no começo do arquivo do analisador gerado:

Opções e definições na especificação para o analisador Mini C

```
%option noyywrap
```

```
%option nodefault
%option outfile="lexer.c" header-file="lexer.h"

%top {
#include "minic_tokens.h"
#include "tabelas.h"

// prototipo da funcao token
Token *token(int, int);
}

NUM [0-9]+
ID [[:alpha:]]([[:alnum:]])*
STRING \"[^\"\n]*\"
```

As opções são as mesmas que já vimos no exemplo anterior. Temos um trecho de código incluído no começo do analisador (a parte começando com `%top`) que realiza a inclusão de dois arquivos de cabeçalho e define o protótipo da função `token` (definida na seção de código). Os cabeçalhos incluídos são `minic_tokens.h`, visto acima, e `tabelas.h`, que declara as funções para tabelas de *strings* e de símbolos: as funções são chamadas de `adiciona_string` para adicionar uma nova *string* na tabela, e `adiciona_simbolo` para um novo identificador na tabela de símbolos. Essas funções são definidas no arquivo `tabelas.c` e retornam o índice da *string* ou símbolo na tabela respectiva; esse índice pode ser usado como valor do *token*. O uso de tabelas de *strings* e de símbolos é importante por vários motivos, entre eles a eficiência do código do compilador. Em capítulos seguintes veremos como a tabela de símbolos é uma estrutura de importância central em um compilador.

A especificação tem três definições: `NUM` é o padrão para constantes numéricas inteiras, idêntica à que vimos no exemplo anterior; `ID` é o padrão que especifica os identificadores da linguagem e `STRING` é o padrão que determina o que é uma literal *string* em um programa Mini C: deve começar e terminar com aspas duplas, contendo no meio qualquer sequência de zero ou mais caracteres que não sejam aspas duplas nem caracteres de nova linha (`\n`).

As regras da especificação são mostradas abaixo:

Regras na especificação para o analisador da linguagem Mini C

```
[[:space:]]  {  }  /* ignora espacos em branco */
\/\/[^\n]*     {  }  /* elimina comentarios */

"#include <stdio.h>"  { return token(TOK_PROLOGO, 0); }

{STRING} { return token(TOK_STRING, adiciona_string(yytext)); }

if       { return token(TOK_PCHAVE, PC_IF); }
else     { return token(TOK_PCHAVE, PC_ELSE); }
while    { return token(TOK_PCHAVE, PC_WHILE); }
return   { return token(TOK_PCHAVE, PC_RETURN); }
printf   { return token(TOK_PCHAVE, PC_PRINTF); }

{NUM}    { return token(TOK_NUM, atoi(yytext)); }
{ID}     { return token(TOK_ID, adiciona_simbolo(yytext)); }

\+       { return token(TOK_OP, SOMA);  }
```

```
-         { return token(TOK_OP, SUB);    }
\*        { return token(TOK_OP, MULT);   }
\/        { return token(TOK_OP, DIV);    }
\<        { return token(TOK_OP, MENOR);  }
==        { return token(TOK_OP, IGUAL);  }
&&        { return token(TOK_OP, AND);    }
!         { return token(TOK_OP, NOT);    }
=         { return token(TOK_OP, ATRIB);  }

\(        { return token(TOK_PONT, PARESQ); }
\)        { return token(TOK_PONT, PARDIR); }
\{        { return token(TOK_PONT, CHVESQ); }
\}        { return token(TOK_PONT, CHVDIR); }
,         { return token(TOK_PONT, VIRG);    }
;         { return token(TOK_PONT, PNTVIRG); }

.         { return token(TOK_ERRO, 0);   }
```

A primeira regra ignora os espaços, como vimos no exemplo anterior. A segunda é para ignorar os comentários. Um comentário é qualquer sequência que começa com duas barras e vai até o fim da linha. Como a barra é um caractere especial no flex, ele precisa ser especificado com a contrabarra antes, e portanto `//` vira `\/\/` no padrão. As outras regras não têm novidade, exceto que a regra para *strings* e a regra para identificadores obtêm o valor do *token* chamando as funções `adiciona_string` e `adiciona_simbolo`, como discutimos antes.

O analisador léxico normalmente não vai ser usado de forma isolada: ele é chamado pelo analisador sintático para produzir *tokens* quando necessário, como veremos no próximo capítulo. Mas aqui vamos testar o funcionamento do analisador léxico isoladamente, criando um programa principal que abre um arquivo de entrada e chama o analisador léxico. Esse programa está no arquivo `lex_teste.c`:

Código fonte code/cap2/minic/lex_teste.c

O programa principal é seguinte:

Função do principal do programa de teste do analisador léxico

```
int main(int argc, char **argv)
{
  Token *tok;

  if (argc < 2) {
    printf("Uso: mclex <arquivo>\n");
    return 0;
  }

  inicializa_analise(argv[1]);

  tok = yylex();
  while (tok != NULL) {
    imprime_token(tok);
    tok = yylex();
  }

  finaliza_analise();
```

```
  return 0;
}
```

A função principal obtém o nome do arquivo de entrada dos argumentos de linha de comando, inicializa a análise passando esse nome de arquivo, e para cada *token* encontrado na entrada imprime uma representação desse *token* na tela. A função `inicializa_analise` apenas tenta abrir o arquivo com o nome passado e configura o analisador léxico para usar esse arquivo, como discutimos antes:

Função que inicializa a análise léxica

```
void inicializa_analise(char *nome_arq)
{
  FILE *f = fopen(nome_arq, "r");

  if (f == NULL) {
    fprintf(stderr, "Nao foi possivel abrir o arquivo de entrada: %s\n ↩
       ", nome_arq);
    exit(1);
  }

  yyin = f;
}
```

E a função de finalização destrói as tabelas de símbolos e de *strings*, e fecha o arquivo de entrada:

Função de finalização da análise léxica

```
void finaliza_analise()
{
  // destroi tabelas
  destroi_tab_strings();
  destroi_tab_simbolos();

  // fecha arquivo de entrada
  fclose(yyin);
}
```

A função de impressão de *tokens* apenas imprime o tipo e o valor do *token*, mas usando uma *string* que representa o nome do tipo ao invés de simplesmente imprimir a constante numérica. A função completa pode ser vista no código fonte.

Para testar o analisador, vamos criar um pequeno programa na linguagem Mini C. Esse programa pode ser encontrado no arquivo `teste.c`.

Código fonte code/cap2/minic/teste.c

Arquivo de teste na linguagem Mini C

```
// teste para o compilador Mini C
#include <stdio.h>

int main()
{
  printf("Ola, mundo!\n");
```

```
  return 0;
}
```

Compilando o programa principal `lex_teste.c` para gerar um executável `mclex`, obtemos a seguinte saída para o arquivo `teste.c`:

Saída do teste do analisador léxico para o arquivo de teste

```
andrei$ ./mclex teste.c
Tipo: prologo - Valor: 0
Tipo: identificador - Valor: 0
Tipo: identificador - Valor: 1
Tipo: pontuacao - Valor: 1
Tipo: pontuacao - Valor: 2
Tipo: pontuacao - Valor: 3
Tipo: palavra chave - Valor: 4
Tipo: pontuacao - Valor: 1
Tipo: string - Valor: 0
Tipo: pontuacao - Valor: 2
Tipo: pontuacao - Valor: 6
Tipo: palavra chave - Valor: 3
Tipo: numero - Valor: 0
Tipo: pontuacao - Valor: 6
Tipo: pontuacao - Valor: 4
```

É interessante ver que o analisador ignorou a primeira linha com um comentário, e o primeiro *token* mostrado é o prólogo. A sequência de *tokens* segue corretamente do conteúdo do arquivo fonte. Também deve ser observado que o valor dos *tokens* de tipo *string* e identificador são os índices deles nas tabelas correspondentes. Para obter a *string* ou identificador encontradas pelo analisador, é preciso acessar as tabelas. Veremos exemplos disso no capítulo seguinte.

2.6 Conclusão

Neste capítulo vimos o que é a etapa de análise léxica do compilador, e como criar um analisador léxico para qualquer linguagem de entrada. Vimos o que são os *tokens* na análise léxica e como especificar os padrões que determinam os tipos de *tokens* usando expressões regulares. É possível criar um analisador léxico escrevendo o código diretamente a partir dos padrões dos *tokens*, mas é mais prático usar um gerador de analisadores léxicos como o flex. Vimos o uso do flex para linguagens bastante simples e também para linguagens de programação mais próxima da realidade. Neste capítulo também vimos uma introdução à linguagem Mini C, que será usada como exemplo no resto do livro. Um analisador léxico completo para a linguagem Mini C, usando o flex, foi mostrado e detalhado neste capítulo. Esse analisador pode servir como base para criar analisadores para outros tipos de linguagens reais.

No capítulo seguinte veremos como utilizar a saída do analisador léxico para fazer a análise sintática dos programas, e estabelecer a estrutura sintática. Essa estrutura é de importância central na compilação dos programas.

Capítulo 3

Análise Sintática

OBJETIVOS DO CAPÍTULO

Ao final deste capítulo você deverá ser capaz de:

- Entender a função do analisador sintático e como ele se integra ao resto do compilador
- Compreender o conceito de estrutura sintática
- Criar Gramáticas Livres de Contexto que capturam a estrutura sintática de linguagens
- Criar o analisador sintático para uma linguagem usando uma ferramenta geradora

A análise sintática é a etapa do compilador que ocorre após a análise léxica. O objetivo da análise sintática é determinar a estrutura sintática do código-fonte que está sendo compilado. Para isso, a análise sintática utiliza o fluxo de *tokens* produzido pela análise léxica.

No capítulo anterior mencionamos uma analogia entre compilação de um programa e leitura de um texto. Vimos que a análise léxica pode ser entendida como o agrupamento de letras individuais em palavras. Já a análise sintática é similar ao agrupamento de palavras para formar frases. Quando lemos, nosso cérebro determina automaticamente a estrutura sintática das frases, pois entender essa estrutura (mesmo que apenas intuitivamente) é necessário para compreender o significado da frase: quem é o sujeito (quem realiza a ação), quem é o objeto (quem sofre a ação), etc. A análise sintática em um compilador faz o mesmo tipo de tarefa, mas determinando a estrutura do código-fonte que está sendo compilado.

Neste capítulo vamos entender em mais detalhes a função do analisador sintático e como ele funciona. Assim como usamos o formalismo das expressões regulares para guiar a criação do analisador léxico, veremos o uso das *Gramáticas Livres de Contexto* para guiar a criação de analisadores sintáticos. Também veremos como usar o gerador bison como uma ferramenta para ajudar na criação do analisador sintático (assim como usamos o flex para a análise léxica).

3.1 Estrutura sintática

A estrutura sintática de um programa relaciona cada parte do programa com suas sub-partes componentes. Por exemplo, um comando `if` completo tem três partes que o compõem: uma condição de teste, um comando para executar caso a condição seja verdadeira, e um comando para excutar caso a condição seja falsa. Quando o compilador identifica um `if` em um programa, é importante que ele

possa acessar esses componentes para poder gerar código executável para o `if`. Por isso, não basta apenas agrupar os caracteres em *tokens*, é preciso também agrupar os *tokens* em estruturas sintáticas.

Uma forma comum de representar a estrutura sintática é usando árvores. As árvores na ciência da computação são normalmente desenhadas de cabeça para baixo, com a raiz no topo. A raiz é um nó da árvore da qual todo o resto se origina. Cada nó pode ter nós filhos que aparecem abaixo dele, ligados por um *arco* ou *aresta*. Uma árvore que representa a estrutura sintática de um programa é normalmente chamada de *árvore sintática*. Para o exemplo do comando condicional mencionado antes, a árvore sintática seria similar à mostrada na Figura 3.1 [38]. Como filhos do nó `if` (a raiz) existem três nós, o primeiro representando a condição, o segundo o comando para o caso da condição ser verdadeira (C1) e o terceiro representando o comando para o caso da condição ser falsa (C2).

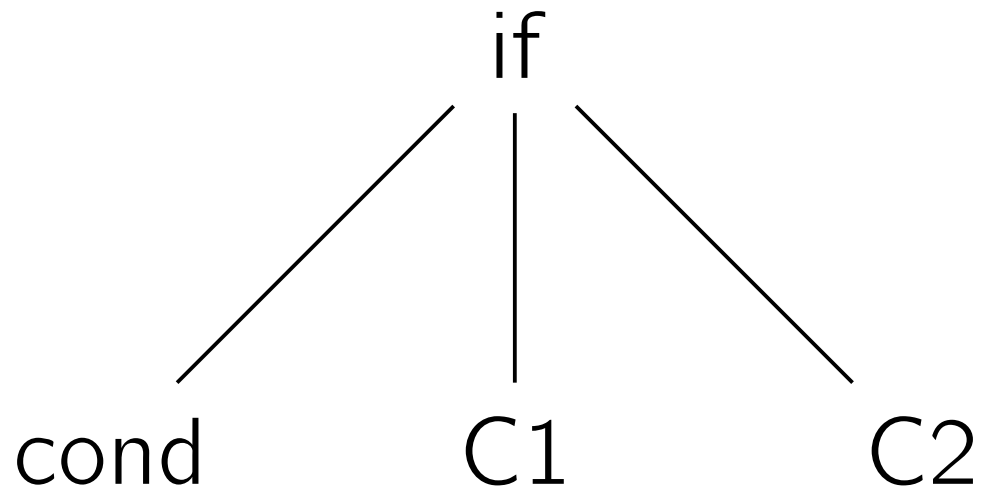

Figura 3.1: Árvore sintática para o comando condicional completo.

O papel da análise sintática é construir uma árvore como a da Figura 3.1 [38] para todas as partes de um programa. Essa árvore é incompleta pois a condição e os comandos C1 e C2 possuem estrutura também, e essa estrutura precisa estar representada na árvore completa. Para discutir a estrutura dessas partes, vamos ver um pouco mais sobre árvores sintáticas, especificamente sobre árvores de expressão.

3.1.1 Árvores de expressão

Árvores de expressão são árvores sintáticas construídas para expressões aritméticas, relacionais ou lógicas. Uma expressão é formada por *operadores*, que designam as operações que fazem parte da expressão, e *operandos*, que designam os valores ou sub-expressões sobre os quais os operadores agem. Em uma árvore de expressão, os operadores são nós internos (nós que possuem filhos) e os valores básicos são folhas da árvore (nós que não possuem filhos e portanto aparecem "nas pontas" da árvore).

Um exemplo é a expressão `2 + 3`, cuja árvore sintática é mostrada na Figura Figura 3.2 [39]a. O operador é `+` e os operandos são `2` e `3`. Uma expressão mais complexa é `(2 + 3) * 7`, cuja árvore é mostrada na Figura Figura 3.2 [39]b. O operando direito da multiplicação é o número `7`, mas o operando esquerdo é a sub-expressão `2 + 3`. Note que como a estrutura da expressão fica evidenciada pela estrutura da árvore, não é necessário usar parênteses na árvore para `(2 + 3) * 7`, apesar dessa expressão precisar de parênteses. Os parênteses são necessários para representar a

estrutura da expressão como uma *string*, uma sequência linear de caracteres, mas não na representação em árvore.

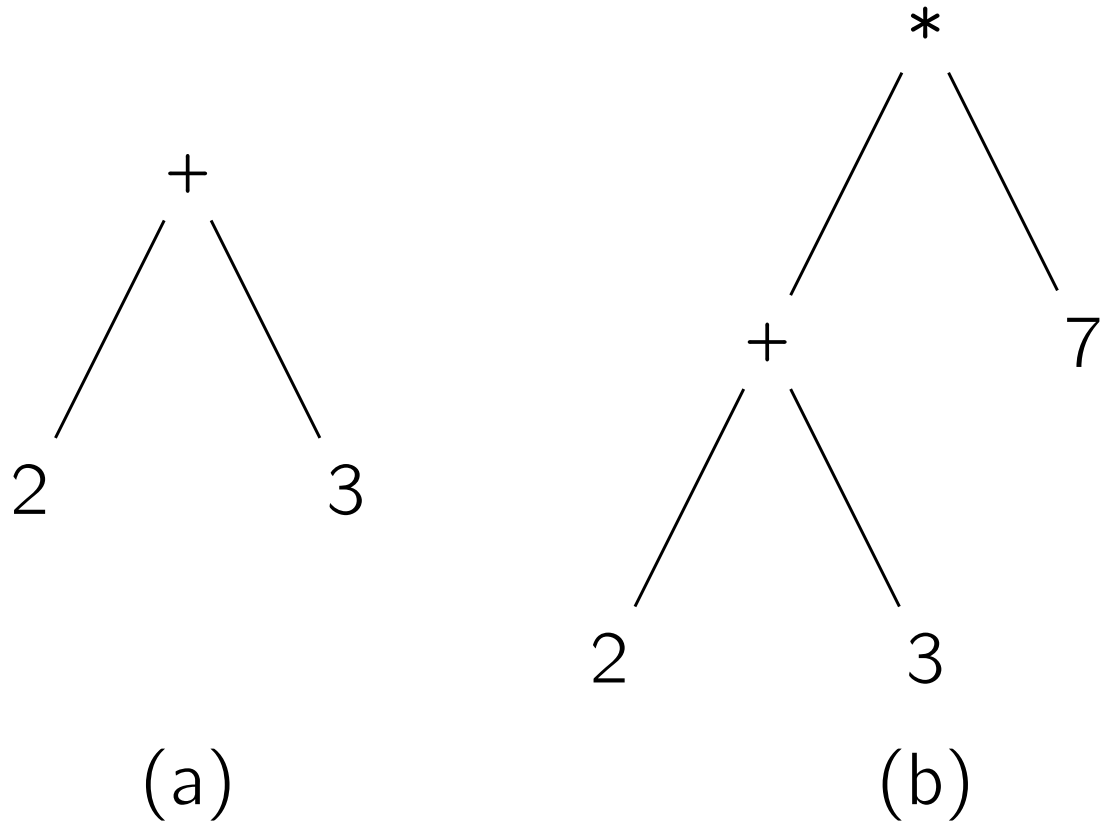

Figura 3.2: Árvore sintática para duas expressões, a) `2 + 3` b) `(2 + 3) * 7`.

As operações em expressões são geralmente binárias, porque precisam de dois operandos. Algumas operações são unárias, como a negação de um número ou o NÃO lógico. Por isso, as árvores de expressão são compostas por nós que podem ter 0, 1 ou 2 filhos. Esse tipo de árvore é normalmente chamado de *árvore binária*.

Em um comando como o condicional `if`, podemos pensar que o `if` é um operador com três operandos: condição, comando 1 e comando 2. Voltando à árvore para um comando condicional, digamos que um trecho do código do programa que está sendo processado pelo compilador é o seguinte:

Exemplo de comando condicional

```
if (x > 2)
  y = (x - 2) * 7;
else
  y = x + 2 * 5;
```

Nesse comando, a condição é uma expressão relacional (de comparação), `x > 2`, o comando 1 é a atribuição do valor da expressão `(x - 2) * 7` à variável `y`, e o comando 2 é a atribuição do valor da expressão `x + 2 * 5` à variável `y`. A árvore para esse comando começa da forma que já vimos, mas inclui a estrutura das partes do comando. Essa árvore pode ser vista na Figura 3.3 [40].

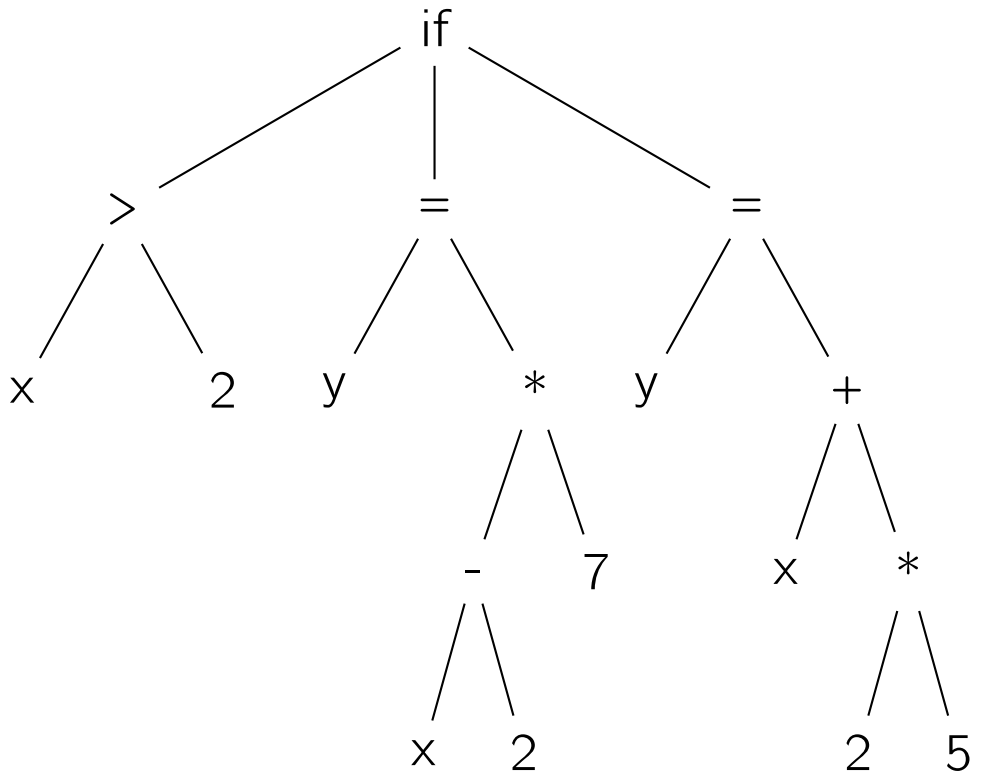

Figura 3.3: Árvore sintática completa para um comando condicional.

O analisador sintático de um compilador vai gerar, para o código mostrado antes, a árvore na Figura 3.3 [40]. As técnicas para fazer isso serão estudadas no resto do capítulo.

3.2 Relação com o Analisador Léxico

O analisador sintático é a etapa que vem logo após o analisador léxico no compilador, e isso acontece porque as etapas estão fortemente relacionadas. A tarefa do analisador sintático é muito mais simples de realizar partindo dos *tokens* da entrada, ao invés dos caracteres isolados.

Em teoria, como vimos no Capítulo 1, a comunicação entre o analisador léxico e o analisador sintático é sequencial: o analisador léxico produz toda a sequência de *tokens* (criada a partir do arquivo de entrada) e passa essa sequência inteira para o analisador sintático. Essa ideia é mostrada na Figura 3.4 [40].

Figura 3.4: Relação simplificada entre analisadores léxico e sintático.

Na prática, as duas etapas são organizadas de forma diferente nos compiladores reais. Não é necessário, para o analisador sintático, ter acesso a toda a sequência de *tokens* para fazer a análise sintática. Na maioria dos casos, é possível construir a árvore sintática examinando apenas um, ou um número

pequeno de *tokens* de cada vez. Por isso, o mais comum é fazer com que o analisador sintático e o analisador léxico funcionem em conjunto, ao invés do léxico terminar todo seu processamento e passar o resultado para o sintático. Nesse arranjo, o analisador sintático está no comando, por assim dizer: é o analisador sintático que aciona o analisador léxico, quando necessário para obter o próximo *token* da entrada. O analisador léxico deve manter controle sobre que partes da entrada já foram lidas e a partir de onde começa o próximo *token*. Essa relação é ilustrada na Figura 3.5 [41], onde `proxtoken()` é a função do analisador léxico que deve ser chamada para obter o próximo *token*; o analisador sintático chama essa função sempre que necessário, obtem o próximo *token*, e continua com a análise.

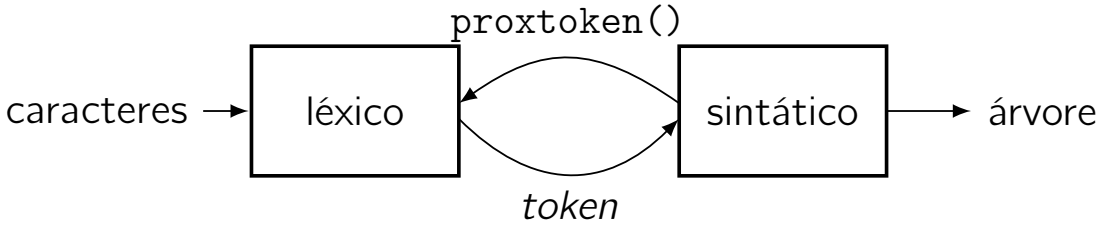

Figura 3.5: Relação entre analisadores léxico e sintático, na prática.

Um dos motivos que levaram a essa organização das duas primeiras etapas de análise foi que em computadores antigos era pouco provável ter memória suficiente para armazenar toda a sequência de *tokens* da entrada (a não ser que o programa de entrada fosse pequeno), então fazia mais sentido processar um *token* de cada vez. Da mesma forma, não havia memória suficiente para guardar toda a árvore sintática do programa. Por isso, os compiladores eram organizados de maneira que o analisador sintático comandava todo o processo de tradução: obtia o próximo *token* do analisador léxico e, se fosse possível, passava uma sub-estrutura completa do programa (uma sub-árvore) para ser processada pelas etapas seguintes, já gerando o código-destino para essa parte. Em seguida, essa parte da árvore era descartada e o analisador sintático passava para a próxima parte da árvore.

Esse tipo de organização de um compilador era conhecida como *tradução dirigida pela sintaxe*. Hoje em dia, com os computadores tendo quantidades de memória disponível muito maiores, é menos comum ver compiladores reais seguindo esse esquema, e muitos constroem a árvore sintática inteira do programa, que é passada para as etapas seguintes. Isso porque vários processos das etapas seguintes podem funcionar melhor se puderem ter acesso à árvore sintática inteira, ao invés de apenas um pedaço de cada vez.

Mas a relação entre o analisador léxico e o analisador sintático continua a mesma mostrada na Figura 3.5 [41] até hoje, mesmo tendo mais memória, pois para a maioria das linguagens de programação não é mesmo necessário acessar toda a sequência de *tokens* da entrada. Alguns tipos de analisador sintático armazenam e analisam os últimos *n tokens*, para *n* pequeno (ao invés de analisar apenas um *token* por vez). Mesmo assim, isso não muda a relação entre os analisadores léxico e sintático, o analisador sintático apenas chama a função de obter o próximo *token* quantas vezes precisar.

3.3 Gramáticas Livres de Contexto

As gramáticas formais são ferramentas para descrição de linguagens. Usamos aqui o adjetivo gramáticas *formais* para distinguir de outros sentidos da palavra "gramática", por exemplo na frase "a gramática da língua portuguesa", mas daqui para frente, sempre que usarmos a palavra "gramática", estaremos nos referindo às gramáticas formais, a não ser que haja indicação do contrário.

As gramáticas livres de contexto estão associadas às *linguagens livres de contexto*. Assim como a classe das linguagens regulares é usada na análise léxica, a classe das linguagens livres de contexto é essencial para a análise sintática. Aqui não vamos nos preocupar com linguagens livres do contexto em geral, apenas usando as gramáticas como ferramentas para fazer a análise sintática.

Uma gramática livre do contexto G é especificada por quatro componentes: o conjunto de símbolos terminais T, o conjunto de símbolos variáveis (ou não-terminais) V, o conjunto de produções P e o símbolo inicial S, sendo que o símbolo inicial deve ser um dos símbolos variáveis ($S \in V$).

As gramáticas funcionam como um formalismo gerador, similar às expressões regulares: começando pelo símbolo inicial, é possível usar as produções para gerar cadeias ou sentenças da linguagem que desejamos. Os símbolos terminais representam símbolos que aparecem na linguagem, enquanto que os símbolos variáveis são usados como símbolos auxiliares durante as substituições. Veremos alguns exemplos para tornar essas ideias mais claras.

3.3.1 Exemplo: Palíndromos

O primeiro exemplo é uma linguagem bastante simples que gera cadeias que são palíndromos. Um *palíndromo* é uma palavra ou frase que é lida da mesma forma de frente para trás e de trás para frente, como "roma e amor" ou "socorram-me, subi no ônibus em marrocos". Vamos trabalhar com palíndromos construídos com um alfabeto bastante limitado, de apenas dois símbolos: `a` e `b`. Alguns palíndromos nesse alfabeto são `abba`, `aaa` e `ababa`.

Existem dois tipos de palíndromos, que podemos chamar de *palíndromos pares* e *palíndromos ímpares*. Os palíndromos pares, como `abba`, contêm um número par de símbolos, com a segunda metade igual ao reverso da primeira metade. No caso de `abba`, as metades são `ab` e `ba`, sendo que a segunda metade, `ba`, é o reverso da primeira, `ab`. Cada símbolo em uma metade deve ocorrer na outra também.

Os palíndromos ímpares, como `ababa`, possuem um número ímpar de símbolos, com uma primeira parte, um símbolo do meio, e uma última parte; a última parte é o reverso da primeira, mas o símbolo do meio pode ser qualquer um. No caso do alfabeto com símbolos `a` e `b`, tanto `ababa` quanto `abbba` são palíndromos ímpares com primeira e última partes idênticas, mas símbolos do meio diferentes.

A gramática para essa linguagem de palíndromos tem dois símbolos terminais (`a` e `b`), um símbolo variável (S) que também é o símbolo inicial, e quatro produções:

$$\begin{aligned} S &\to aSa \\ S &\to bSb \\ S &\to a \\ S &\to b \\ S &\to \varepsilon \end{aligned}$$

Cada uma dessas produções representam uma forma em que o símbolo S pode ser transformado para gerar cadeias da linguagem. O símbolo ε representa uma cadeia vazia, ou seja, uma cadeia sem nenhum símbolo. Quando temos várias produções para o mesmo símbolo variável, como no caso da gramática para palíndromos, podemos economizar espaço usando a seguinte notação:

$$S \to aSa \mid bSb \mid a \mid b \mid \varepsilon$$

Todas as produções para o símbolo S aparecem na mesma linha, separadas por barras. Podemos ler essa gramática como "S pode produzir aSa ou bSb ou ...".

O processo de geração de uma cadeia seguindo as regras de produção de uma gramática é chamado de *derivação*, e será explicado a seguir.

3.3.2 Derivação

Vamos começar estabelecendo algumas definições necessárias:

Uma *sentença* em uma gramática é uma sequência de símbolos terminais. Para a gramática de palíndromos com a e b, `abba` é uma sentença.

Uma *forma sentencial* de uma gramática é uma sequência de símbolos terminais e variáveis. Uma forma sentencial pode ser formada apenas por símbolos variáveis, apenas por símbolos terminais, ou uma mistura dos dois tipos. Dessa forma, toda sentença é uma forma sentencial, mas uma forma sentencial que inclua algum símbolo variável não é uma sentença. Para a gramática de palíndromos em a e b, `aSa` é uma forma sentencial (mas não é sentença), enquanto `aaa` é uma forma sentencial que também é uma sentença.

Uma *derivação* na gramática G é uma sequência de formas sentenciais tal que:

1. A primeira forma sentencial da sequência é apenas o símbolo inicial da gramática G
2. A última forma sentencial é uma sentença (ou seja, só tem símbolos terminais)
3. Cada forma sentencial na sequência (exceto a primeira) pode ser obtida da forma sentencial anterior pela substituição de um símbolo variável pelo lado direito de uma de suas produções

Um exemplo simples de derivação na gramática de palíndromos é:

$$S \Rightarrow a$$

Essa derivação tem apenas duas formas sentenciais: S, que é o símbolo inicial, e a, que é uma sentença. Para separar as formas sentenciais em uma derivação usamos o símbolo $\Rightarrow$. A derivação demonstra que a cadeia a é uma sentença da linguagem gerada pela gramática, e ela é obtida a partir do símbolo S pelo uso da terceira produção da gramática, $S \rightarrow a$. Como especificado pela produção, substituímos o símbolo S pelo símbolo a, gerando assim a segunda forma sentencial; nesse caso, a segunda forma sentencial já é uma sentença, e a derivação termina por aí (até porque não existem mais símbolos variáveis na forma sentencial).

Uma derivação com um passo a mais seria:

$$S \Rightarrow aSa \Rightarrow aa$$

A sentença gerada nessa derivação é aa. No primeiro passo da derivação, substituímos o símbolo S por aSa, usando a sua primeira produção. No segundo passo o símbolo S entre os dois a é substituído pela cadeia vazia (a última produção na gramática), desaparecendo e deixando apenas os dois a's.

Agora vejamos a derivação para gerar a cadeia `abba`:

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abba$$

Os dois primeiros passos mostram S sendo substituído por aSa e bSb, nesta ordem. O último passo mais uma vez substitui o S pela cadeia vazia, fazendo com que ele desapareça da forma sentencial.

Para gerar `ababa` a derivação é similar, mudando apenas no último passo:

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow ababa$$

Desta vez, ao invés de substituir S pela cadeia vazia no último passo, substituímos por a, obtendo o resultado esperado. Podemos ver que a derivação para um palíndromo par termina com a substituição

de *S* pela cadeia vazia no último passo, enquanto que a derivação para um palíndromo ímpar termina com *S* substituído por *a* ou *b*.

Qualquer derivação usando a gramática para palíndromos vai gerar, ao final, uma sentença que é um palíndromo usando os dois símbolos `a` e `b`. Não há como, seguindo as produções da gramática, gerar uma sentença que não é um palíndromo usando esses dois símbolos. O conjunto de todas as sentenças geradas por uma gramática livre de contexto é a *linguagem* gerada pela gramática.

A ideia é usar as gramáticas para descrever as estruturas sintáticas que podem ser formadas na linguagem que queremos analisar. Isso é parecido com o que vimos na análise léxica, de usar expressões regulares para descrever os padrões de *tokens* que podem ser usados na linguagem.

Agora que já entendemos como especificar uma gramática livre de contexto e o processo de derivação a partir dela, vamos ver mais alguns exemplos de linguagens e suas estruturas sintáticas descritas por gramáticas.

3.3.3 Exemplo: Expressões Aritméticas

Um exemplo mais similar às linguagens de programação é uma linguagem simples para expressões aritméticas, como vimos no Capítulo 2. Aqui veremos uma gramática para uma linguagem de expressões aritméticas formadas por números inteiros e as quatro operações básicas.

Diferente do exemplo anterior dos palíndromos, para a linguagem de expressões não é interessante trabalhar com caracteres isolados. Afinal, vimos como criar um analisador léxico justamente para agrupar os caracteres em *tokens*, o que facilita muito a análise sintática. Por isso, nesse exemplo e em praticamente todos daqui para a frente, os símbolos terminais não serão caracteres, mas sim *tokens*. Alguns *tokens* são formados por apenas um caractere, mas para a gramática não faz diferença; a análise sintática vai ser realizada com base nos *tokens*.

Para a linguagem de expressões, temos *tokens* de três tipos: números, operadores e pontuação. Os operadores são os símbolos para as quatro operações, e o tipo pontuação é para os parênteses. Lembrando do capítulo anterior, cada *token* tem um tipo e um valor; um *token* do tipo operador vai ter um valor associado que determina qual dos quatro operadores o *token* representa. O mesmo acontece com o valor dos *tokens* de tipo pontuação: o valor especifica se é um parêntese abrindo ou fechando. Para os *tokens* de tipo número, o valor é o valor numérico do *token*.

Uma gramática para a linguagem de expressões é a seguinte:

$$E \rightarrow E + E \mid E * E \mid E - E \mid E / E \mid (E) \mid \textbf{num}$$

Essas produções representam o fato que uma expressão pode ser:

- Uma soma (ou multiplicação, subtração, divisão) de duas expressões
- Uma expressão entre parênteses
- Uma constante numérica (representada aqui por um token de tipo **num**)

Todos os símbolos nas produções dessa gramática são variáveis ou são *tokens*; para deixar a notação mais leve, usamos o caractere + para representar um *token* de tipo operador e valor que representa um operador de soma. Isso não deve causar problema; deve-se apenas lembrar que todos os terminais são *tokens*. No caso do *token* de tipo **num**, o valor dele não aparece na gramática porque não é relevante para a estrutura sintática da linguagem. Qualquer *token* de tipo número, independente do valor, faz parte dessa mesma produção (diferente dos *tokens* de operadores).

Vejamos algumas derivações nessa gramática. Começando por uma expressão simples, `142 + 17`. A sequência de *tokens* associada a essa expressão é `<num, 142> <op, SOMA> <num, 17>`. Na derivação a seguir vamos representar os tokens da mesma forma que na gramática (ou seja, `<op, SOMA>` vira apenas +, e qualquer *token* de tipo número é representado apenas como **num**):

$$E \Rightarrow E + E \Rightarrow \textbf{num} + E \Rightarrow \textbf{num} + \textbf{num}$$

Em cada passo de derivação substituímos um símbolo variável pelo lado direito de uma de suas produções. Na derivação anterior, quando chegamos na forma sentencial $E + E$, temos a opção de substituir o E da esquerda ou o da direita; no caso, escolhemos o da esquerda. Mas o resultado seria o mesmo se tivéssemos começado pelo E da direita. Apenas a sequência de passos da derivação apareceria em outra ordem, mas o resultado final seria o mesmo, e a estrutura sintática da expressão seria a mesma.

Podemos estabelecer algumas ordens padronizadas, por exemplo em uma *derivação mais à esquerda*, quando há uma escolha de qual símbolo variável substituir, sempre escolhemos o símbolo mais à esquerda (como no exemplo anterior). Da mesma forma podemos falar de uma *derivação mais à direita*.

Mas existe uma forma melhor de visualizar uma derivação, uma forma que deixa mais clara a estrutura sintática de cada sentença derivada, e que não depende da ordem dos símbolos variáveis substituídos. Essa forma são as *árvores de derivação*.

3.3.4 Árvores de Derivação

Uma alternativa para representar derivações em uma gramática é usar as *árvores de derivação* ao invés de sequências lineares de formas sentenciais que vimos até agora. Uma árvore de derivação é semelhante às árvores sintáticas que vimos antes, mas incluem mais detalhes relacionados às produções da gramática utilizada. Uma árvore sintática não inclui nenhuma informação sobre símbolos variáveis da gramática, por exemplo. Mais à frente, um dos nossos objetivos será obter a árvore sintática de um programa, mas para fazer a análise sintática é importante entender as árvores de derivação.

Em uma árvore de derivação, cada nó é um símbolo terminal ou variável. As folhas da árvore são símbolos terminais, e os nós internos são símbolos variáveis. Um símbolo variável V vai ter como filhos na árvore os símbolos para os quais V é substituído na derivação. Por exemplo, sejam as seguintes derivações na gramática de expressões:

$$E \Rightarrow \textbf{num}$$
$$E \Rightarrow E + E \Rightarrow \textbf{num} + E \Rightarrow \textbf{num} + \textbf{num}$$

As árvores de derivação correspondentes são:

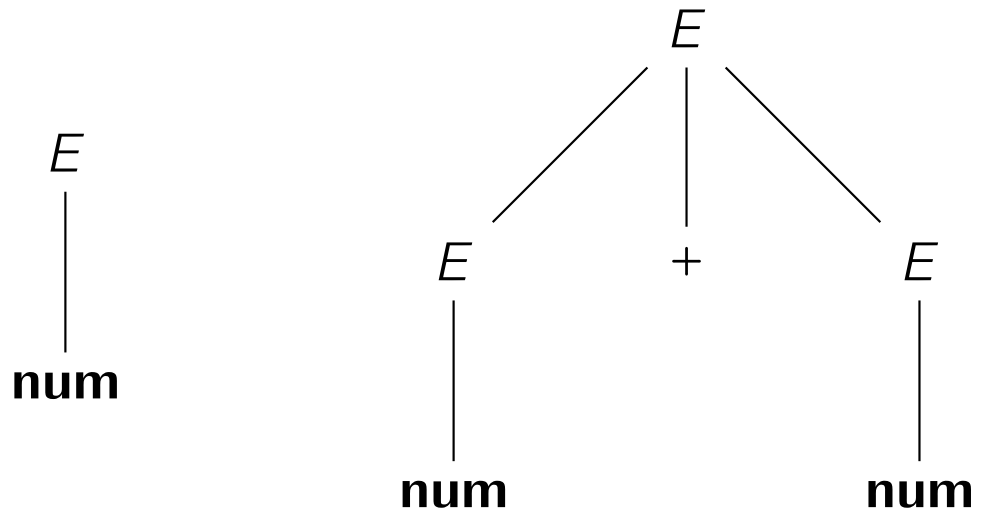

Figura 3.6: Árvores de derivação para duas expressões.

Vemos que quando o símbolo E é substituído apenas por **num**, o nó correspondente na árvore só tem um filho (ver árvore esquerda na Figura 3.6 [46]). Quando o símbolo E é substituído por $E + E$, isso significa que o nó correspondente na árvore terá três filhos (ver árvore direita na Figura 3.6 [46]).

Para uma árvore como a que está mostrada no lado direito da Figura 3.6 [46], não importa a ordem de substituição dos dois símbolos E na forma sentencial $E + E$; qualquer que seja a ordem, a árvore de derivação será a mesma.

Entretanto, existem sentenças geradas por essa gramática de expressões para as quais nós podemos encontrar mais de uma árvore de derivação. Quando temos mais de uma árvore de derivação para uma mesma sentença, dizemos que a gramática é *ambígua*, e a ambiguidade de uma gramática é um problema, como veremos a seguir.

3.3.5 Ambiguidade

O exemplo anterior demonstra um problema importante que pode ocorrer com gramáticas livres de contexto: ambiguidade. Uma gramática é *ambígua* quando existe pelo menos uma sentença gerada pela gramática que pode ser gerada de duas ou mais formas diferentes; ou seja, essa sentença terá duas ou mais árvores de derivação diferentes.

A ambiguidade é um problema pois significa que uma mesma sentença pode ter duas estruturas sintáticas diferentes, na mesma gramática. A estrutura sintática de uma sentença vai influenciar no seu significado e como ela é interpretada pelo compilador, por exemplo. Desta forma, uma gramática ambígua para uma linguagem de programação significa que certos programas poderiam funcionar de duas (ou mais) maneiras diferentes, dependendo de como o compilador interprete as partes ambígua. Obviamente é importante que uma linguagem tenha programas que funcionem sempre de uma mesma maneira, caso contrário o programador teria dificuldade para aprender como trabalhar com a linguagem.

No exemplo da gramática de expressões, uma ambiguidade ocorre quando misturamos operadores como soma e multiplicação. Na expressão `6 * 5 + 12`, deve ser efetuada primeiro a soma ou a multiplicação? Em termos de estrutura sintática, a pergunta é se a expressão é

1. uma soma, com operando esquerdo `6 * 5` e operando direito `12`

2. ou uma multiplicação com operando esquerdo `6` e operando direito `5 + 12`

Nós somos acostumados com a convenção de sempre fazer multiplicações e divisões antes de somas e subtrações, então para nós o mais natural é seguir a primeira interpretação. Mas a gramática que vimos não estabelece nenhuma interpretação, possibilitando as duas. Para essa mesma sentença, nesta gramática, duas árvores de derivação podem ser construídas:

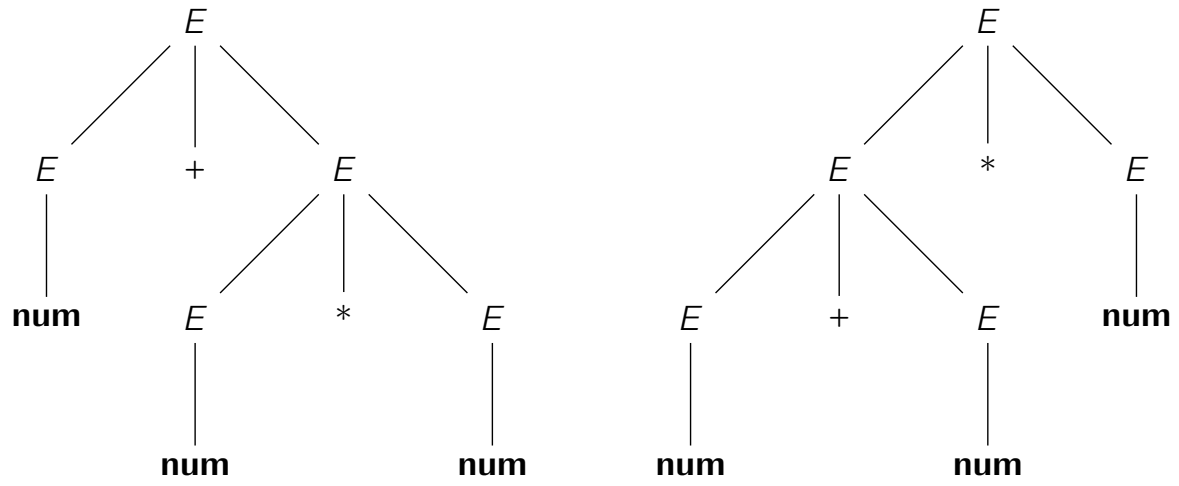

Figura 3.7: Duas árvores de derivação para a sentença 6 * 5 + 12

Cada uma das árvores representa uma das duas interpretações para a expressão. A árvore da esquerda representa a primeira interpretação: para realizar a soma é necessário obter o valor dos seus dois operandos, sendo que o operando esquerdo da soma é a multiplicação `6 * 5`; portanto, a multiplicação seria realizada primeiro. A árvore direita da Figura 3.7 [47] representa a segunda interpretação, que seria calcular primeiro a soma `5 + 12` e depois multiplicar por 6.

O que queremos é que a própria gramática evite a ambiguidade, determinando apenas uma das duas árvores de derivação para uma sentença como `6 * 5 + 12`, e que essa árvore corresponda à interpretação esperada: que a multiplicação deve ser efetuada antes da soma. Para isso precisamos construir uma nova gramática, que codifica nos símbolos variáveis os diferentes níveis de precedência dos operadores:

$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid T/F \mid F$$
$$F \rightarrow (E) \mid \textbf{num}$$

Essa gramática tem três símbolos variáveis E, T e F (que podemos pensar como *expressão*, *termo* e *fator*). Cada um representa um nível de precedência:

- o símbolo E representa a precedência mais baixa, onde estão os operadores de soma e subtração.
- T representa o próximo nível de precedência, com os operadores de multiplicação e divisão.
- F representa o nível mais alto de precedência, onde ficam os números isolados e as expressões entre parênteses; isso significa que o uso de parênteses se sobrepõe à precedência de qualquer operador, como esperado.

Esta gramática gera as mesmas sentenças que a primeira gramática de expressões que vimos, mas sem ambiguidade. Nesta gramática, existe apenas uma árvore de derivação para a sentença `6 * 5 + 12`, mostrada na Figura 3.8 [48].

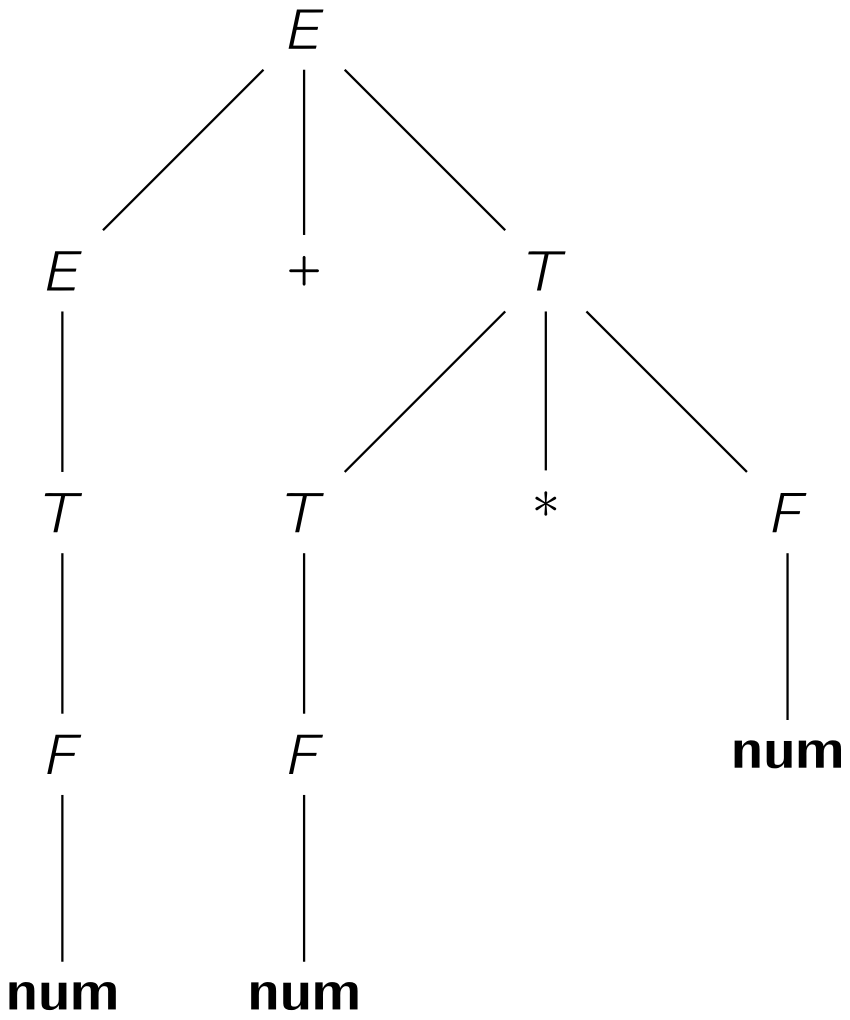

Figura 3.8: Árvore de derivação para a sentença 6 * 5 + 12 na nova gramática

A árvore mostrada na Figura 3.8 [48] é mais complexa do que as árvores da Figura 3.7 [47], mas essa complexidade adicional é necessária para evitar a ambiguidade.

Toda linguagem de programação tem uma parte para expressões aritméticas, relacionais e lógicas. Isso significa que a gramática para uma linguagem de programação vai incluir uma parte para expressões. Essa parte da gramática de qualquer linguagem de programação segue a mesma ideia vista no último exemplo: é usado um símbolo variável para cada nível de precedência. Como as expressões em uma linguagem de programação completa pode ter vários níveis de precedência (bem mais do que três), essa acaba se tornando uma parte grande da gramática da linguagem. A seguir veremos um exemplo de gramática para uma linguagem de programação simples.

3.3.6 Exemplo: Linguagem de programação simples

Agora que já vimos as características das gramáticas livres de contexto e alguns exemplos, vamos ver uma gramática para uma linguagem de programação simples, que demonstra o tipo de situações com

as quais teremos que lidar para criar o analisador sintático de um compilador.

$$
\begin{aligned}
&C \rightarrow \texttt{print}\ \ \textbf{string} \\
&C \rightarrow \texttt{if}\ R\ \texttt{then}\ C\ \texttt{else}\ C \\
&C \rightarrow \textbf{num} := E \\
&R \rightarrow R = E \mid R < E \mid E \\
&E \rightarrow E + T \mid E - T \mid T \\
&T \rightarrow T * F \mid T / F \mid F \\
&F \rightarrow (E) \mid \textbf{num} \mid \textbf{id}
\end{aligned}
$$

3.4 Geradores de Analisadores Sintáticos

Os geradores de analisadores sintáticos funcionam de maneira bastante similar aos geradores de analisadores léxicos vistos no Capítulo 2. Para gerar um analisador sintático, usamos a ferramenta geradora passando como entrada uma especificação da estrutura sintática da linguagem que queremos analisar; a saída do gerador é um analisador sintático na forma de código em alguma linguagem de programação (no nosso caso, um arquivo na linguagem C). Esse analisador recebe um fluxo de *tokens* na entrada e gera uma árvore sintática na saída. A Figura 3.9 [49] mostra um diagrama de blocos que representa o uso de um gerador de analisadores sintáticos, como descrito. No nosso caso, a ferramenta de geração é o bison, versão do projeto GNU para o utilitário yacc do Unix.

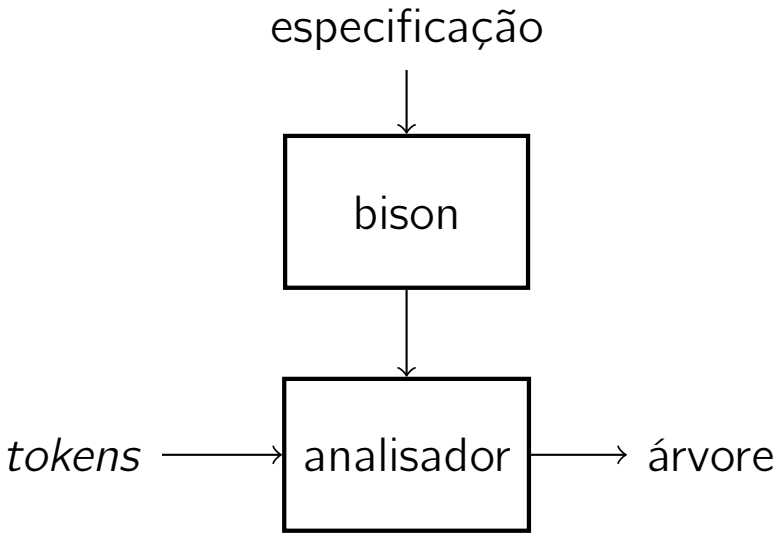

Figura 3.9: Uso de um gerador de analisadores sintáticos

Capítulo 4

Análise Semântica

OBJETIVOS DO CAPÍTULO

Ao final deste capítulo você deverá ser capaz de:

- objetivo 1
- objetivo 2
- objetivo N

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. == Representação Intermediária

OBJETIVOS DO CAPÍTULO

Ao final deste capítulo você deverá ser capaz de:

- objetivo 1
- objetivo 2
- objetivo N

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.