Construção de Compiladores

José de Oliveira Guimarães
Departamento de Computação
UFSCar - São Carlos, SP
Brasil
e-mail: jose@dc.ufscar.br

March 26, 2007

Contents

Chapter 1

Introdução

1.1 Compiladores e Ligadores

Um compilador é um programa que lê um programa escrito em uma linguagem L_1 e o traduz para uma outra linguagem L_2. Usualmente, L_1 é uma linguagem de alto nível como C++ ou Prolog e L_2 é assembler ou linguagem de máquina. Contudo, o uso de C como L_2 tem sido bastante utilizado. Utilizando C como linguagem destino torna o código compilado portável para qualquer máquina que possua um compilador de C, que são praticamente 100% dos computadores. Se um compilador produzir código em assembler, o seu uso estará restrito ao computador que suporta aquela linguagem assembler.

Quando o compilador traduzir um arquivo (ou programa) em L_1 para linguagem de máquina, ele produzirá um arquivo de saída chamado de arquivo ou programa objeto, usualmente indicado pela extensão "`.obj`". No caso geral, um programa executável é produzido pela combinação de vários arquivos objetos pelo ligador (*linker*). Veremos como o *linker* executa esta tarefa estudando um exemplo. Suponha que os arquivos "A.c" e "B.c" foram compilados para "A.obj" e "B.obj". O arquivo "A.c" define uma função `f` e chama uma função `g` definida em "B.c". O código de "B.c" define uma função `g` e chama a função `f`. Existe uma chamada a `f` em "A.c" e uma chamada a `g` em "B.c". Esta configuração é ilustrada na Figura 1.1. O compilador compila "A.c" e "B.c" produzindo "A.obj" e "B.obj", mostrados na Figura 1.2. Cada arquivo é representado por um retângulo dividido em três partes. A superior contém o código de máquina correspondente ao arquivo "A.c". Neste código, utilizamos

```
call  000
```

para a chamada de *qualquer* função ao invés de colocarmos uma chamada de função em código de

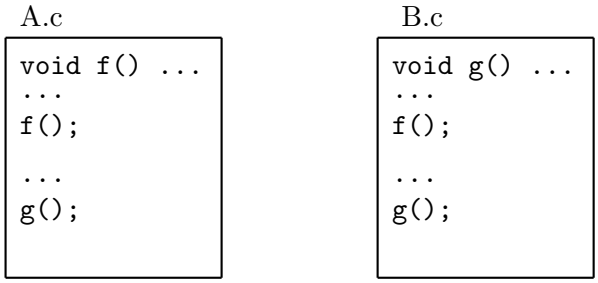

Figure 1.1: Dois arquivos em c que formam um programa

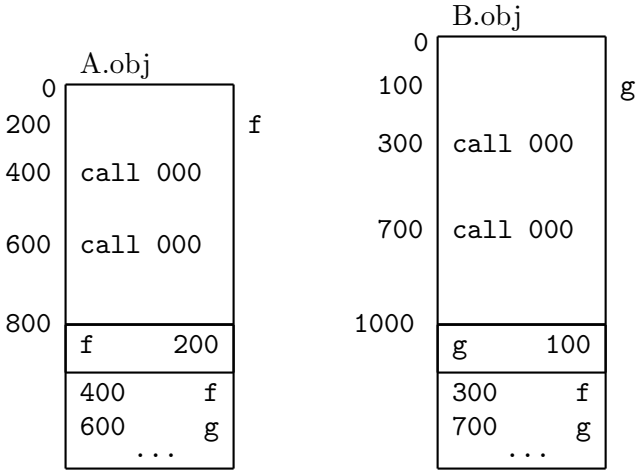

Figure 1.2: Configuração dos arquivos objeto

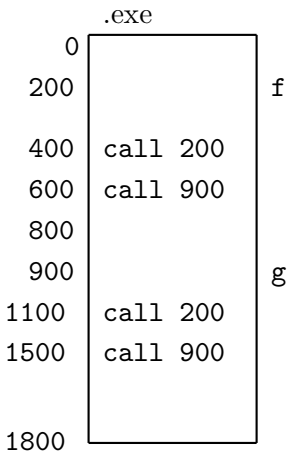

Figure 1.3: Configuração do arquivo executável

máquina, que seria composta apenas de números.

A parte intermediária do retângulo contém os nomes das funções definidas no arquivo juntamente com o endereço delas. Assim, o arquivo "A.obj" define uma função chamada `f` cujo endereço é `200`. Isto é, o endereço da primeira instrução de `f` é `200`. A última parte da definição de "A.obj", o retângulo mais embaixo, contém os nomes das funções que são chamadas em "A.obj" juntamente com o endereço onde as funções são chamadas. Assim, a função `f` é chamada no endereço `400` e a função `g`, em `600`. Os números utilizados acima (`200`, `400`, `600`) consideram que o primeiro byte de "A.obj" possui endereço `0`.

Para construir o programa executável, o *linker* agrupa o código de máquina de "A.obj" e "B.obj" (parte superior do arquivo) em um único arquivo, mostrado na Figura 1.3. Como "B.obj" foi colocado após "A.obj", o linker adiciona aos endereços de "B.obj" o tamanho de "A.obj", que é `800`. Assim, a definição da função `g` estava na posição `100` e agora está na posição `900` (`100 + 800`).

Como todas as funções estão em suas posições definitivas, o *linker* ajustou as chamadas das funções utilizando os endereços de `f` e `g`, que são `200` e `900`. O arquivo "A.c" chama as funções `f` e `g`, como mostrado na Figura 1.1. O compilador transforma estas chamadas em código da forma

```
...
call 000   /* chama f */
...
call 000   /* chama g */
...
```

onde `000` foi empregado porque o compilador não sabe ainda o endereço de `f` e `g`. O *linker*, depois de calculado os endereços definitivos destas funções, modifica estas chamadas para

```
...
call 200   /* chama f */
...
call 900   /* chama g */
...
```

Ao executar o programa ".exe", o sistema operacional carrega-o para a memória e pode ser necessário modificar todas as chamadas de função, adaptando-as para os endereços nos quais elas foram carregadas. Por exemplo, se o executável foi carregado para a possição 5000 na memória, a função que começa na posição 200 do executável estará na posição 5200 da memória. Assim, todas as chamadas a esta função, que são da forma

```
call 200
```

deverão ser modificadas para

```
call 5200
```

O carregamento do programa para a memória e possíveis realocações de endereços é chamada de carregamento.

1.2 Sistema de Tempo de Execução

Qualquer programa compilado requer um sistema de tempo de execução (*run-time system*). O sistema de tempo de execução (STE) é formado por todos os algoritmos de um programa executável que não foram especificados diretamente pelo programador. Algumas das funções do STE são descritas a seguir.

- Fazer a busca por um método em resposta a um envio de mensagem em C++.
- Procurar na pilha de funções ativas[1] qual função trata uma exceção levantada com `throw` em C++.
- Alocar memória para as variáveis locais de uma função antes do início da execução da mesma e desalocar a memória ao término da execução da função.
- Gerenciar o armazenamento do endereço de retorno da chamada da função.
- Chamar o construtor para uma variável cujo tipo é uma classe com construtor quando a variável for criada. Idem para o destrutor.
- Fazer a coleta de lixo de tempos em tempos. Todos os testes que o coletor de lixo possa ter inserido no programa fazem parte do sistema de tempo de execução.

[1] As que foram chamadas pelo programa mas que ainda não terminaram a sua execução.

- Fornecer a *string* contendo a linha de comando com a qual o executável foi chamado.[2] A linha de comando fornecida pelo sistema operacional.
- Converter valores de um tipo para outro. Quando alguma transformação for necessária, será feita pelo STE.

Parte do código do sistema de tempo de execução é fornecido como bibliotecas já compiladas e parte é acrescentado pelo compilador. No primeiro caso, temos o algoritmo de coleta de lixo e o código que obtém a linha de comando do sistema operacional. No segundo caso estão todos os outros itens citados acima.

O *linker* agrupa todos os ".obj" que fazem parte do programa com um arquivo ".obj" que contém algumas rotinas do sistema de tempo de execução.

1.3 Interpretadores

Um compilador traduz o código fonte para linguagem de máquina que é modificada pelo *linker* para produzir um programa executável. Este programa é executável diretamente pelo computador.

Existe uma outra maneira de executar um programa: interpretá-lo. Há diversas modos de interpretação:

1. o interpretador lê o texto do programa e vai executando as instruções uma a uma. Atualmente esta forma é raríssima;
2. o interpretador toma o texto do programa e o traduz para uma estrutura de dados interna (semelhante à ASA que veremos neste curso, um conjunto de objetos que representa todo o programa). Então o interpretador, após a traduação do programa para a estrutura de dados, percorre esta estrutura interpretando o programa;
3. um compilador traduz o texto do programa para instruções de uma máquina virtual (pseudo-código). Então um interpretador executa estas instruções. Esta máquina virtual é usualmente uma máquina não só simples como feita sob medida para a linguagem que se quer utilizar

Existem vantagens e desvantagens no uso de compiladores/*linkers* em relação ao uso de interpretadores, sintetizados abaixo. Ao citar interpretadores, temos em mente aqueles descritos nos itens 2 e 3 acima.

1. O código interpretado pelo interpretador nunca interfere com outros programas. O interpretador pode conferir todos os acessos à memória e ao hardware impedindo operações ilegais.
2. Com o código interpretado torna-se fácil construir um *debugger* pois a interpretação está sob controle do interpretador e não do hardware.
3. Interpretadores são mais fáceis de fazer, por vários motivos. Primeiro, eles traduzem o código fonte para um pseudo-código mais simples do que o assembler ou linguagem de máquina utilizados pelos compiladores (em geral). Segundo, eles não necessitam de *linkers* — a ligação entre uma chamada da função e a função é feita dinamicamente, durante a interpretação. Terceiro, o sistema de tempo de execução do programa está presente no próprio interpretador, feito em uma linguagem de alto nível. O sistema de tempo de execução de um programa compilado deve ser, pelo menos em parte, codificado em assembler ou linguagem de máquina.

[2] Esta *string* pode ser manipulada em C++ com o uso dos parâmetros `argv` e `argc` da função `main`.

4. Interpretadores permitem iniciar a execução de um programa mais rapidamente do que se estivermos utilizando um compilador (veja a descrição 2 de interpretadores). Para compreender este ponto, considere um programa composto por vários arquivos já traduzidos para o pseudo-código pelo interpretador. Suponha que o programador faça uma pequena modificação em um dos arquivos e peça ao ambiente de programação para executar o programa. O interpretador traduzirá o arquivo modificado para pseudo-código e imediatamente iniciará a interpretação do programa. Apenas o arquivo modificado deve ser re-codificado. Isto contrasta com um ambiente de programação aonde compilação é utilizada.

   Se um arquivo for modificado, freqüentemente será necessário compilar outros arquivos que dependem deste, mesmo se a modificação for minúscula. Por exemplo, se o programador modificar o valor de uma constante em um `define`

   ```
   #define Num 10
   ```

   de um arquivo "def.h", o sistema deverá recompilar todos os arquivos que incluem "def.h".

   Depois de compilados um ou mais arquivos, dever-se-á fazer a ligação com o *linker*. Depois de obtido o programa executável, ele será carregado em memória e os seus endereços realocados para novas posições. Só após isto o programa poderá ser executado. O tempo total para realizar todas esta operações em um ambiente que emprega compilação é substancialmente maior do que se interpretação fosse utilizada.

5. Compiladores produzem código mais eficiente do que interpretadores. Como o código gerado por um compilador será executado pela própria máquina, ele será muito mais eficiente do que o pseudo-código interpretado equivalente. Usualmente, código compilado é 10-20 vezes mais rápido do que código interpretado.

Da comparação acima concluimos que interpretadores são melhores durante a fase de desenvolvimento de um programa, pois neste caso o importante é fazer o programa iniciar a sua execução o mais rápido possível após alguma alteração no código fonte.

Quando o programa estiver pronto, será importante que ele seja o mais rápido possível, o que pode ser obtido compilando-o.

1.4 Aplicações

Os algoritmos e técnicas empregados em compiladores são utilizados na construção de outras ferramentas descritas a seguir.

1. Editores de texto orientados pela sintaxe. Um editor de texto pode reconhecer a sintaxe de uma linguagem de programação e auxiliar o usuário durante a edição. Por exemplo, ele pode avisar que o usuário esqueceu de colocar `(` após o while:

   ```
   while  i > 0 )
   ```

2. *Pretty printers*. Um *pretty printer* lê um programa como entrada e produz como saída este mesmo programa com a tabulação correta. Por exemplo, se a entrada for

   ```
   #include <iostream.h>
   void
       main()
   {  int i;       for ( i = 0;
   ```

```
     i < 10; i++
     )
   cout << endl;
        }
```

a saída poderá ser

```
#include <iostream.h>

void main()
{
  int i;

  for ( i = 0; i < 10; i++ )
    cout << endl;
  }
```

O comando indent do Unix formata e tabula um programa em C apropriadamente de acordo com algumas opções da linha de comando.

3. Analisadores estáticos de programas, que descobrem erros como variáveis não inicializadas, código que nunca será executado, situações em que uma rotina não retorna um valor, etc. O código abaixo mostra um erro que poderá ser descoberto por um destes analisadores.

```
void calcule( int *p, int *w )
{

  int soma = *p + *w;
  return soma + *p/*w;  /* retorna soma */;
  }
```

O Unix possui um analisador chamado lint que descobre erros deste tipo. Um analisador mais poderoso, disponível para o DOS e Windows, é o PC-lint.

4. Analisadores que retornam a cadeia de chamadas de um programa. Isto é, eles informam quem chama quem. Por exemplo, dado o programa

```
#include <stdio.h>

int fatorial( int n )
{
  if ( n >= 1 )
    return n *fatorial(n-1);
  else
    return 1;
  }

int fat( int n )
{
```

```
  return n > 1 ? n*fatorial(n - 1) : 1 ;
  }

void main()
{
  int n;

  printf("Digite n ");
  scanf( "%d", &n );
  printf("\nfat(%d) = %d\n", n, fat(n) );
  }
```

como entrada ao utilitário `cflow` do Unix, a saída será:

```
1   main: void(), <lx.c 19>
2       printf: <>
3       scanf: <>
4       fat: int(), <lx.c 14>
5           fatorial: int(), <lx.c 6>
6               fatorial: 5
```

5. Formatadores de texto como TEX e LATEX. Estes formatadores admitem textos em formato não documento com comandos especificando como a formatação deve ser feita e o tipo das letras. Por exemplo, o trecho

   A função `main` inicia todos ... é utilizada com freqüência para ...

   é obtido digitando-se

```
A fun\c{c}\~{a}o {\tt main} inicia todos ... \'{e} utilizada com
freq\"{u}\^{e}ncia para ...
```

6. Interpretadores de consulta a um Banco de Dados. O usuário pede uma consulta como
   `select  dia > 5 and dia < 20 and idade > 25`
   que faz o interpretador imprimir no vídeo os registros satisfazendo às três condições acima.

7. Interpretadores de comandos de um sistema operacional. Além de reconhecer se o comando digitado é válido, o interpretador deve conferir os parâmetros:

```
C:\>mkaedir so
C:\>dir *.cpp *.h
C:\>del *.exe
```

   o interpretador sinalizaria erro nos dois primeiros comandos.

8. Interpretadores de expressões. Alguns programas gráficos leêm uma função digitada pelo usuário e fazem o seu gráfico. A função é compilada para pseudo-código e que é então interpretado.

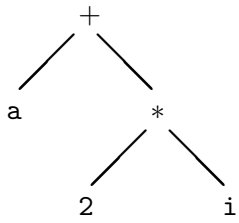

Figure 1.4: Árvore de sintaxe da sentença `a + 2*i`

9. Alguns editores de texto fazem busca por cadeias de caracteres utilizando expressões regulares. Por exemplo, para procurar por um identificador que começa com a letra `A` e termina com `0`, podemos pedir ao editor para buscar por

   ```
   A[a-zA-Z0-9_]*0
   ```

   Um conjunto entre `[` e `]` significa qualquer caráter do conjunto e `*` é repetição de zero ou mais vezes do caráter anterior. Assim, `[a-z]*` é repetição de zero ou mais letras minúsculas. Esta expressão é transformada em um autômato e interpretada.

1.5 As Fases de Um Compilador

Um compilador típico divide a compilação em seis fases, descritas abaixo. Cada fase toma como entrada a saída produzida pela fase anterior. A primeira fase, análise léxica, toma como entrada o programa fonte a ser compilado. A última fase, geração de código, produz um arquivo ".obj" com o código gerado a partir do programa fonte.

1. Análise léxica. Esta fase lê o programa fonte produzindo como saída uma seqüência de números correspondentes a pequenas partes do programa chamados *tokens*. Por exemplo, às palavras chave `while`, `if` e `return` estão associados os números `1`, `10`, `12` e os números `40` e `45` estão associados a identificadores e literais numéricas. Assim, a entrada

   ```
   if ( i > 5 )
     return j
   ```

   produziria a seqüência de *tokens* `10`, `50`, `40`, `60`, `45`, `51`, `12`, `40` e `66`. Observe que `50`, `60`, `51` e `66` estão associados a "`(`", "`>`", "`)`" e "`;`".

2. Análise sintática. Esta fase toma os números produzidos pela fase anterior e verifica se eles formam um programa correto de acordo com a gramática da linguagem. Em geral, uma árvore de sintaxe abstrata será também construída nesta fase. Esta árvore possuirá nós representando cada sentença da gramática. Por exemplo, a expressão `a + 2*i` seria representada como mostrada na Figura 1.4.

3. Análise semântica. Esta fase confere se o programa está correto de acordo com a definição semântica da linguagem. Como exemplo, o comando

   ```
   i = fat(n);
   ```

   em C++ requer que:

   - `i` e `n` tenham sido declarados como variáveis e `i` não tenha sido declarada com o qualificador `const`;
   - `fat` seja uma função que possua um único parâmetro para cujo tipo o tipo de `n` possa ser convertido;

- `fat` retorne alguma coisa e que o tipo de retorno possa ser convertido para o tipo de `i`;

Todas estas conferências são feitas pelo analisador semântico. Observe que "`i = fat(n)`" é um comando sintaticamente correto, independente de qualquer conferência semântica.

4. Geração de código intermediário. Nesta fase, o compilador gera código para uma máquina abstrata que certamente é mais simples do que a máquina utilizada para a geração do código final. Este código intermediário é gerado para que algumas otimizações possam ser mais facilmente feitas. Aho, Sethi e Ullman citam o exemplo do comando

```
p = i + r*60
```

compilado para o código intermediário

```
temp1 = inttoreal(60)
temp2 = id3*temp1
temp3 = id2 + temp2
id1 = temp3
```

otimizado para

```
temp1 = id3*60.0
id1 = id2 + temp1
```

Note que:

- o tipo de `i` e `r` é real, sendo necessário converter `60` para real;
- `id1`, `id2` e `id3` correspondem a `p`, `i` e `r`;
- o código intermediário não admite, como assembler, gerar o código

  ```
  id1 = id2 + id3*60
  ```

  diretamente. Tudo deve ser feito em partes pequenas;
- o compilador não gera o código otimizado diretamente. Primeiro ele gera um código ruim que então é otimizado. Isto é mais simples do que gerar o código otimizado diretamente.

5. Otimização de código. Como mostrado anteriormente, esta fase otimiza o código intermediário. Contudo, otimização de código pode ocorrer também durante a geração de código intermediário e na geração de código. Como exemplo do primeiro caso, o compilador pode otimizar

```
n = 20;
b = 4*k;
a = 3*n;
```

para

```
n = 20;
b = k << 2;
a = 60;
```

antes de gerar o código intermediário. Como exemplo do segundo caso, o compilador pode otimizar

```
i = i + 1
```

para

```
inc i
```

aproveitando o fato de que a máquina alvo possui uma instrução que incrementa o valor de um inteiro de um (`inc`). Admitimos que o código intermediário não possui esta instrução. Nesta fase também é feita a alocação dos registradores da máquina para as variáveis locais, o que também é um tipo de otimização.

6. Geração de código. Esta fase gera o código na linguagem alvo, geralmente assembler ou linguagem de máquina. Como visto na fase anterior, esta fase é também responsável por algumas otimizações de código.

As fases de um compilador são agrupadas em *front end* e *back end*. O *front end* é composto pelas fases que dependem apenas da linguagem fonte, como da sua sintaxe e semântica. O *back end* é composto pelas fases que dependem da máquina alvo.

Utilizando a divisão do compilador exposta anteriormente, o *front end* consiste da análise léxica, sintática e semântica, geração de código intermediário e otimização de código. O *back end* consiste apenas da geração de código. Obviamente, otimizações de código dependentes das características da máquina alvo fazem parte do *back end*.

Se tivermos um compilador para uma linguagem `L` que gera código para uma máquina A, poderemos fazer este compilador gerar código para uma máquina B modificando-se apenas o *back end*. Por este motivo, é importante produzir código intermediário para que nele possam ser feitas tantas otimizações quanto possíveis. Estas otimizações serão aproveitadas quando o compilador passar a gerar código para outras máquinas diferentes da original.

Em todas as fases, o compilador utilizará uma tabela de símbolos para obter informações sobre os identificadores do programa. Quando um identificador for encontrado durante a compilação, como `j` na declaração

```
int j;
```

ele será inserido na tabela de símbolos juntamente com outras informações, como o seu tipo. As informações da TS (tabela de símbolos) é utilizada para fazer conferências semânticas, como conferir que `j` foi declarado em

```
j = 1;
```

e que `1` pode ser convertido para o tipo de `j`. Ao gerar código para esta atribuição, o compilador consultará a TS para descobrir o tipo de `j` e então gerar o código apropriado.

Durante a análise léxica, todos os identificadores do programa estarão associados a um único número, que assumiremos ser `40`. Assim,

```
j = i + 1;
```

retornará a seqüência de número `40 33 40 63 45`. Para descobrir mais informações sobre o identificador, poderemos utilizar os métodos de um objeto `lex`. Por exemplo, para saber a *string* correspondente ao último identificador encontrado, fazemos

```
lex->getStrToken()
```

Com a *string* retornada, poderemos consultar a TS para obter mais informações sobre o identificador, como o seu escopo e tipo.

Nos próximos capítulos, veremos cada uma das fases do compilador em detalhes, embora com algumas diferenças das fases apresentadas nesta seção:

- o código gerado será em linguagem C;
- não haverá geração de código intermediário.

Chapter 2

A Análise Sintática

2.1 Gramáticas

A sintaxe de uma linguagem de programação descreve todos os progranas válidos naquela linguagem e é descrita utilizando-se uma gramática, em geral livre de contexto. A gramática representa, de forma finita, todos os infinitos[1] programas válidos na linguagem.

Uma gramática G é especificada através de uma quádrupla (N, Σ, P, S) onde

- N é o conjunto de símbolos não-terminais;
- Σ é o conjunto de símbolos terminais;
- P é um conjunto de produções;
- S é o símbolo não-terminal inicial da gramática.

Os programas válidos da linguagem são aqueles obtidos pela expansão de S.
Utilizando a gramática

Expr	::= Expr "+" Term | Expr "−" Term | Term
Term	::= Term "∗" Numero | Term "/" Numero | Numero
Numero	::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

temos que:

- N = { Expr, Term, Numero }
- Σ = { +, −, ∗, /, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }

  Os terminais serão colocados entre aspas quando houver dúvidas sobre quais são os terminais. Neste exemplo, está claro que os operadores aritméticos e os dígitos são terminais e portanto as aspas não foram utilizadas.
- P = { Expr ::= Expr + Term | Expr − Term | Term, Term ::= Term ∗ Numero | Term / Numero | Numero, Numero ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }
- S = Expr

[1] Em geral.

Nos exemplos deste curso, uma gramática será apresentada somente através de suas regras de produção com o símbolo inicial aparecendo do lado esquerdo da primeira regra. Assim, todos os elementos N, Σ, P e E estarão claramente identificados.

Uma sentença válida na linguagem é composta apenas por terminais e derivada a partir do símbolo inicial S, que neste exemplo é Expr. A derivação de Expr começa substituindo-se este símbolo pelos símbolos que aparecem à esquerda de Expr nas regras de produção da gramática. Por exemplo,

$$\text{Expr} \Longrightarrow \text{Expr} + \text{Term}$$

é uma derivação de Expr. O lado direito de $\Longrightarrow$ pode ser derivado substituindo-se Expr ou Term. Substituiremos Expr:

$$\text{Expr} \Longrightarrow \text{Expr} + \text{Term} \Longrightarrow \text{Term} + \text{Term}$$

Substituindo-se sempre o não-terminal mais à esquerda, obteremos

$$\text{Expr} \Longrightarrow \text{Expr} + \text{Term} \Longrightarrow \text{Term} + \text{Term} \Longrightarrow \text{Numero} + \text{Term}$$
$$\Longrightarrow 7 + \text{Term} \Longrightarrow 7 + \text{Numero} \Longrightarrow 7 + 2$$

Uma seqüência de símbolos w de tal forma que

$$\text{S} \Longrightarrow \alpha_1 \Longrightarrow \alpha_2 \ldots \Longrightarrow \alpha_n \Longrightarrow \text{w}$$

$n \geq 0$, é chamada de sentença de G, onde G é a gramática (N, Σ, P, S) e w é composto apenas por terminais.

O símbolo $\overset{*}{\Longrightarrow}$ significa "derive em zero ou mais passos" e $\overset{+}{\Longrightarrow}$ significa "derive em um ou mais passos". Então, uma sentença w de G é tal que

$$\text{S} \overset{+}{\Longrightarrow} \text{w}$$

A linguagem gerada pela gramática G é indicada por L(G). Assim,

$$\text{L(G)} = \{\text{w} \mid \text{S} \overset{+}{\Longrightarrow} \text{w}\}$$

onde S é o símbolo inicial da gramática e w é composto apenas por terminais.

Uma derivação de S contendo símbolos terminais e não-terminais é chamada de *forma sentencial de G*. Isto é, se $\text{S} \overset{*}{\Longrightarrow} \alpha$, α é uma forma sentencial de G.

Neste curso, utilizaremos apenas derivações à esquerda e à direita. Em uma derivação à esquerda

$$\text{S} \Longrightarrow \alpha_1 \Longrightarrow \alpha_2 \Longrightarrow \ldots \Longrightarrow \alpha_n \Longrightarrow \text{w}$$

somente o não-terminal mais à esquerda de cada forma sentencial α_i $(1 \leq i \leq n)$ é substituído a cada passo. Indicamos uma derivação à esquerda $\alpha \Longrightarrow \beta$ por

$$\alpha \underset{lm}{\Longrightarrow} \beta$$

onde *lm* significa `leftmost`.

Derivação à direita possui uma definição análoga à derivação à esquerda, sendo indicada por

$$\alpha \underset{rm}{\Longrightarrow} \beta$$

onde *rm* significa `rightmost`.

Considerando a derivação

$$\text{Expr} \Longrightarrow \text{Expr} + \text{Term} \Longrightarrow \text{Term} + \text{Term} \Longrightarrow \text{Numero} + \text{Term}$$
$$\Longrightarrow 3 + \text{Term} \Longrightarrow 3 + \text{Numero} \Longrightarrow 3 + 2$$

podemos construir uma árvore de análise associada a ela, mostrada na Figura 2.1. Esta árvore abstrai a ordem de substituição dos não-terminais na derivação de Expr: a mesma árvore poderia ter sido gerada por uma derivação à direita.

2.2 Ambigüidade

Uma gramática que produz mais de uma derivação à esquerda (ou direita) para uma sentença é dita ser ambígüa. Isto é, uma gramática será ambígüa se uma sentença w puder ser obtida a partir do símbolo inicial S por duas derivações à esquerda (ou direita) diferentes.

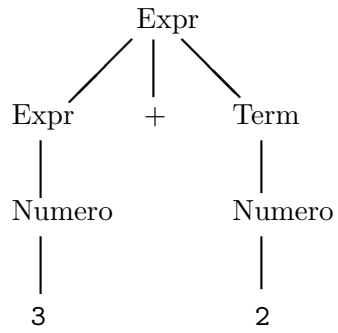

Figure 2.1: Árvore de análise de "3 + 2"

Uma gramática ambígüa produz mais de uma árvore de sintaxe para pelo menos uma sentença. Como exemplo de gramática ambígüa, temos

E::= E + E | E ∗ E | Numero

onde Numero representa qualquer número natural. A sentença

3 + 4 ∗ 5

pode ser gerada de duas formas usando derivação à esquerda:

$$E \Longrightarrow E + E \Longrightarrow Numero + E \Longrightarrow 3 + E \Longrightarrow 3 + E * E \Longrightarrow 3 + Numero * E$$
$$\Longrightarrow 3 + 4 * E \Longrightarrow 3 + 4 * Numero \Longrightarrow 3 + 4 * 5$$

e

$$E \Longrightarrow E * E \Longrightarrow E + E * E \Longrightarrow Numero + E * E \Longrightarrow 3 + E * E$$
$$\Longrightarrow 3 + Numero * E \Longrightarrow 3 + 4 * E \Longrightarrow 3 + 4 * Numero \Longrightarrow 3 + 4 * 5$$

Admitimos que +, ∗ e os números naturais são terminais.

Ambigüidade é indesejável porque ela associa dois significados diferentes a uma sentença como "3 + 4 ∗ 5". No primeiro caso, a primeira derivação é

$E \Longrightarrow E + E$

o que implica que a multiplicação 4 ∗ 5 será gerada pelo segundo E de "E + E". Isto significa que a multiplicação deverá ser avaliada antes da soma, já que o resultado de 4 ∗ 5 é necessário para E + E. Então, esta derivação considera que ∗ possui maior precedência do que +, associando 23 como resultado de 3 + 4 ∗ 5.

No segundo caso, a primeira derivação é

$E \Longrightarrow E * E$

e o primeiro E de "E ∗ E" deve gerar 3 + 4, pois + vem antes de ∗ na sentença "3 + 4 ∗ 5".[2] Sendo assim, esta seqüência de derivações admite que a soma deve ser calculada antes da multiplicação. Portanto é considerado que + possui maior precedência do que ∗.

2.3 Associatividade e Precedência de Operadores

Na linguagem S2 e na maioria das linguagens de programação, os operadores aritméticos +, −, ∗ e / são associativos à esquerda. Isto é, "3 + 4 − 5" é avaliado como "(3 + 4) − 5". Quando um operando, como 4, estiver entre dois operadores de mesma precedência (neste caso, + e −), ele será associado

[2]Se o segundo E gerasse um sinal +, teríamos algo como E ∗ E + E, que não se encaixa na sentença 3 + 4 ∗ 5.

ao operador mais à esquerda — neste caso, +. Por isto dizemos que os operadores aritméticos são associados à esquerda.

A gramática

Expr ::= Expr + Term | Expr − Term | Term

Term ::= Term ∗ Numero | Termo / Numero | Numero

com terminais +, −, ∗, / e Numero, associa todos os operadores à esquerda. Para entender como + é associativo à esquerda, basta examinar a produção

Expr ::= Expr + Term

Tanto Expr quanto Term podem produzir outras expressões. Mas Term nunca irá produzir um terminal +, como pode ser facilmente comprovado examinando-se a gramática. Então, todos os símbolos + obtidos pela derivação de "Expr + Term" se originarão de Expr. Isto implica que, em uma derivação para obter uma sentença com mais de um terminal +, como

2 + 6 + 1

o não-terminal Expr de "Expr + Term", deverá se expandir para resultar em todos os símbolos +, exceto o último:

$$\text{Expr} \Longrightarrow \text{Expr} + \text{Term} \Longrightarrow \text{Expr} + 1 \Longrightarrow \text{Expr} + \text{Term} + 1 \stackrel{+}{\Longrightarrow} 2 + 7 + 1$$

Implícito nesta derivação está que, para fazer a soma

Expr + Term

primeiro devemos fazer todas as somas em Expr, que está à direita de + em "Expr + Term". Então, os operadores + à esquerda devem ser utilizados primeiro do que os + da direita, resultando então em associatividade à esquerda.

Voltando à gramática, temos que a regra

Expr ::= Expr + Term | Expr − Term | Term

produz, no caso geral, uma seqüência de somas e subtrações de não-terminais Term:

$$\text{Expr} \stackrel{+}{\Longrightarrow} \text{Term} + \text{Term} - \text{Term} + \text{Term} - \text{Term}$$

Então, as somas e subtrações só ocorrerão quando todos os não-terminais Term forem avaliados. Examinando a regra para Term,

Term ::= Term ∗ Numero | Term / Numero | Numero

descobrimos que Term nunca gerará um + ou −. Então, para avaliar Expr, devemos avaliar uma seqüência de somas e subtrações de Term. E, antes disto, devemos executar todas as multiplicações e divisões geradas por Term. Isto implica que ∗ e / possuem maior precedência do que + e −.

Um operador ∗ possuirá maior precedência do que + quando ∗ tomar os seus operandos antes do que +. Isto é, se houver um operando (número) entre um ∗ e um +, este será ligado primeiro ao ∗. Um exemplo é a expressão

3 + 4 ∗ 5

onde 4 está entre + e ∗ e é avaliada como

3 + (4 ∗ 5)

2.4 Modificando uma Gramática para Análise

As seções seguintes descrevem como fazer a análise sintática para uma dada gramática. O método de análise utilizado requer que a gramática :

1. não seja ambígüa;
2. esteja fatorada à esquerda e;
3. não tenha recursão à esquerda.

O item 1 já foi estudado nas seções anteriores. O item 2, fatoração à esquerda, requer que a gramática não possua produções como

A ::= β α_1 | β α_2

onde β, α_1 e α_2 são seqüências de terminais e não-terminais. Uma gramática com produções

A ::= β_1 | β_1 α_1

A ::= β_2 | β_2 α_2

pode ser fatorada à esquerda transformando estas produções em

A ::= $\beta_1 X_1$ | $\beta_2 X_2$

X_1 ::= ϵ | α_1

X_2 ::= ϵ | α_2

onde ϵ corresponde à forma sentencial vazia. Observe que fatoração à esquerda é semelhante à fatoração matemática, como transformar $x + ax^2$ em $x(1 + ax)$.

O item 3 requer que a gramática não tenha recursão à esquerda. Uma gramática será recursiva à esquerda se ela tiver um não-terminal A de tal forma que exista uma derivação $A \stackrel{+}{\Longrightarrow} A\alpha$ para uma seqüência α de terminais e não-terminais.

Se uma gramática tiver produções

A ::= Aα_1 | Aα_2 | ... | Aα_m | β_1 | β_2 | ... | β_n

onde nenhum β_i começa com A, a recursão à esquerda pode ser eliminada transformando-se estas produções em

A ::= β_1A′ | β_2A′ | ... | β_nA′

A′ ::= α_1A′ | α_2A′ | ... | α_mA′ | ϵ

Observe que as primeiras produções geram uma seqüência de zero ou mais α_i iniciada por um único β_j, como

$\beta_3\alpha_3\alpha_1\alpha_5$

$\beta_5\alpha_7\alpha_1\alpha 1\alpha_1\alpha_5\alpha_3$

β_2

Após a eliminação da recursão à esquerda, torna-se claro que um β_j aparecerá primeiro na forma sentencial derivada de A por causa das produções

A ::= β_j A′

e que haverá uma repetição de um ou mais α_i por causa das produções

A′ ::= α_iA′

A regra acima não funcionará quando A derivar Aα em dois ou mais passos, como acontece na gramática

S::= Aa | b

A::= Ac | Sd | E

extraída de Aho, Sethi e Ullman [4]. S é recursivo à esquerda, pois S $\Longrightarrow$ Aa $\Longrightarrow$ Sda. Aho et al. apresentam um algoritmo (página 177) capaz de eliminar este tipo de recursão para algumas gramáticas. Este algoritmo não será estudado neste curso.

Utilizando as técnicas descritas acima, eliminaremos a recursão à esquerda e fatoraremos a gramática

Expr ::= Expr + Term | Expr − Term | Term

Term ::= Term ∗ Numero | Term / Numero | Numero

Começamos eliminando a recurssão à esquerda de Expr, considerando A igual a Expr, α_1 e α_2 iguais a "+ Term" e "− Term" e β_1 igual a Term. O resultado é

Expr ::= Term Expr′

Expr′ ::= + Term Expr′ | − Term Expr′ | ϵ

fazendo o mesmo com Term, obtemos

Term ::= Numero Term′

Term′ ::= ∗ Numero Term′ | / Numero Term′ | ϵ

A gramática resultante já está fatorada.

Freqüentemente, a recursividade de um não terminal como Expr′ é transformado em iteração pelo uso dos símbolos { e } na gramática, que indicam "repita zero ou mais vezes". Assim,

A ::= a{b}

gera as sentenças

a
ab
abbbb

aplicando esta notação para a gramática anterior, obtemos

Expr ::= Term { (+|−) Term }
Term ::= Numero { (∗|/) Numero }

onde +|− significa + ou −.

2.5 Análise Sintática Descendente

A análise sintática (*parsing*) de uma seqüência de *tokens* determina se ela pode ou não ser gerada por uma gramática. A seqüência de *tokens* é gerada pelo analisador léxico a partir do programa fonte a ser compilado.

Existem diversos métodos de análise sintática e o método que estudaremos a seguir é chamado de descendente, pois ele parte do símbolo inicial da gramática e faz derivações buscando produzir como resultado final a seqüência de *tokens* produzida pelo analisador léxico. Um exemplo simples de análise descendente é analisar

2 + 3

a partir da gramática

E ::= T E′
E′ ::= + T E′ | ϵ
T ::= N T′
T′ ::= ∗ N T′ | ϵ
N ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

onde +, ∗ e os dígitos 0...9 são terminais. Inicialmente, marcamos o primeiro símbolo da entrada com sublinhado e começamos a derivar a partir do símbolo inicial E:

$\underline{2}$ + 3$
$\underline{E}$

O $ foi acrescentado ao final da entrada para representar o *token* "fim de arquivo". Marcaremos com sublinhado o próximo símbolo sendo considerado da forma sentencial derivada de E. À esquerda do sublinhado estarão apenas terminais que já foram acasalados com a entrada.

Como só existe uma alternativa para derivar E, assim fazemos:

$\underline{2}$ + 3$
$\underline{T}$ E′

Existe também uma única alternativa para T:

$\underline{2}$ + 3$
$\underline{N}$ T′ E′

Agora N possui 10 alternativas e escolheremos aquela que acasala (*match*) com o *token* corrente da entrada:

$\underline{2}$ + 3$
$\underline{2}$ T′ E′

Quando o *token* corrente da entrada, também chamado de símbolo *lookahead*, for igual ao terminal da seqüência derivada de E, avançamos ambos os indicadores de uma posição, indicando que o *token* da

entrada foi aceito pela gramática:

2 $\underline{+}$ 3\$
2 $\underline{\text{T}'}$ E′\$

Agora, T′ deveria derivar uma *string* começada por +. Como isto não é possível (T′ ::= ∗ N T′ | ϵ), escolhemos derivar T' para ϵ e deixar que + seja gerado pelo próximo não terminal, E′:

2 $\underline{+}$ 3\$
2 $\underline{\text{E}'}$

E′ possui duas produções, E′ ::= + T E′ e E′ ::= ϵ e escolhemos a primeira por que ela acasala com a entrada:

2 $\underline{+}$ 3\$
2 $\underline{+}$ T E′

O + é aceito resultando em

2 + $\underline{3}$\$
2 + $\underline{\text{T}}$ E′

T é substituido pela sua única produção resultando em

2 + $\underline{3}$\$
2 + $\underline{\text{N}}$ T′ E′

Novamente, escolhemos a produção que acasala com a entrada, que é N ::= 3 :

2 + $\underline{3}$\$
2 + $\underline{3}$ T′ E′

Aceitando 3, temos

2 + 3$\underline{\$}$
2 + 3 T′ E′

Como não existem mais caracteres a serem aceitos, ambos T′ e E′ derivam para ϵ:

2 + 3$\underline{\$}$
2 + 3

Obtendo uma derivação a partir do símbolo inicial E igual à entrada estamos considerando que a análise sintática obteve sucesso.

Em alguns passos da derivação acima, o não terminal sendo considerado possuiu várias alternativas e escolhemos aquela que acasala a entrada. Por exemplo, em

$\underline{2}$ + 3\$
$\underline{\text{N}}$ T′ E′

escolhemos derivar N utilizando N ::= 2 porque o *token* corrente da entrada era 2.

No caso geral de análise, não é garantido que escolheremos a produção correta (N ::= 2 neste caso) a ser derivada, mesmo nos baseando no *token* corrente de entrada.

Como exemplo, considere a gramática

S ::= cAd
A ::= ab | a

tomada de um exemplo de Aho, Sethi e Ullman [4]. Com a entrada "cad" temos a seguinte análise:

$\underline{\text{c}}$ad\$
$\underline{\text{S}}$

$\underline{\text{c}}$ad\$
$\underline{\text{c}}$Ad

c$\underline{\text{a}}$d\$
c$\underline{\text{A}}$d

ca<u>d</u>$
ca<u>b</u>d

ca<u>d</u>$
ca<u>b</u>d

Na última derivação há um erro de síntaxe: os dois terminais, da entrada e da forma sentencial derivada, são diferentes. Isto aconteceu porque escolhemos a produção errada de A. Portanto, temos que fazer um retrocesso (*backtracking*) à última derivação:

ca<u>d</u>$
c<u>A</u>d

e escolher a segunda produção de A, que é A ::= a:

ca<u>d</u>$
ca<u>d</u>

que resulta em

ca<u>d</u>$
ca<u>d</u>

que aceita a entrada.

Em muitos casos, podemos escrever a gramática de tal forma que o retrocesso nunca seja necessário. Um analisador sintático descendente para este tipo de gramática é chamado de analisador preditivo. Em uma gramática para este tipo de analisador, dado que o símbolo corrente de entrada (*lookahead*) é "a" e o terminal a ver expandido (ou derivado) é A, existe uma única alternativa entre as produções de A,

A ::= α_1 | α_2 | ... | α_n

que começa com a *string* "a". Se uma gramática não tiver nenhuma produção do tipo A ::= ϵ, a condição acima será suficiente para garantir que um analisador preditivo pode ser construído para ela. As condições necessárias e suficientes para a construção de um analisador preditivo serão apresentadas no Capítulo 3.

As gramáticas de linguagens de programação em geral colocam uma palavra chave antes de cada comando exceto chamada de procedimento ou atribuição, facilitando a construção de analisadores preditivos. Utilizando-se uma gramática

Stmt ::= "if" Expr "then" Stmt "endif"
Stmt ::= "return" Expr
Stmt ::= "while" Expr "do" Stmt
Stmt ::= ident AssigOrCall
AssigOrCall ::= "=" Expr | "(" ExprList ")"

onde os símbolos entre aspas são terminais, pode-se analisar a entrada

```
if achou
then
  i = 1;
  return 0;
endif
```

sem retrocesso.

2.6 Análise Sintática Descendente Recursiva

Análise sintática descendente recursiva é um método de análise sintática descendente que emprega uma subrotina para cada não terminal da gramática. As subrotinas chamam-se entre si, em geral

havendo recursão entre elas.

Estudaremos apenas analisadores preditivos, isto é, sem retrocesso. Analisadores deste tipo podem ser construídos para, provavelmente, todas as linguagens de programação. Assim, não é uma limitação nos restringir a analisadores preditivos.

Uma gramática adequada para a construção de um analisador recursivo descendente não pode ser ambígua, deve estar fatorada à esquerda e não deve ter recursão à esquerda.

Estas condições são necessárias mas não suficientes para que possamos construir um analisador preditivo para a gramática. As condições necessárias e suficientes para que um analisador preditivo possa ser construído serão apresentadas nas próximas seções.

A gramática

Expr ::= Expr + Term | Expr − Term | Term
Term ::= Term ∗ Numero | Term / Numero | Numero
Numero ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

não é adequada para análise preditiva pois não está fatorada à esquerda e possui recursão à esquerda. Esta gramática foi modificada para

Expr ::= Term Expr2
Expr2 ::= + Term Expr2 | − Term Expr2 | ϵ
Term ::= Numero Term2
Term2 ::= ∗ Numero Term2 | / Numero Term2 | ϵ
Numero ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

na Seção 2.4, eliminando estes problemas.

O analisador preditivo para esta gramática é apresentada abaixo. Assume-se que exista um analisador léxico `nextToken` que coloca o token corrente em `token`.

Para cada não terminal da gramática existe uma função que o analisa. Sempre que o token corrente da entrada for igual ao terminal esperado pela função, a função `nextToken` é chamada para avançar para o próximo *token*. Isto é, quando o analisador estiver na configuração

$2 \underline{+} 3\$$
$2 \underline{+}$ Term Expr′

ele fará o teste

```
if ( token == mais_smb ) {
  nextToken();
  ...
  }
```

aceitando o + e passando para o próximo *token* da entrada.

```
void expr()
{
  term();
  expr2();    // utilizamos expr2 para Expr'
  }

void expr2()
{
  if ( token == mais_simb ) {
      // Expr' ::= + Term Expr'
    nextToken();
    term();
```

```
    expr2();
    }
  else
    if (token == menos_simb) {
         // Expr' ::= - Term Expr'
      nextToken();
      term();
      expr2();
      }
    /* se nenhum dos dois testes for verdadeiro (+ ou -), a alternativa
       escolhida sera Expr' ::= vazio e esta funcao nao deve fazer nada
    */
  }

void term()
{
 if ( token == numero_smb ) {
   nextToken();
   term2();
   }
 else
     // Como nao ha alternativa Term ::= vazio, houve um erro
   erro();
 }

void term2()
{
  switch (token) {
    case mult_smb:
      nextToken();
      if (token == numero_smb)
        nextToken();
      else
        erro();
      term2();
      break;
    case div_smb:
      nextToken();
      if (token == numero_smb)
        nextToken();
      else
        erro();
      term2();
      break;
    default:
      // Ok, existe producao Term' ::= vazio
    }
  }
```

```
void erro()
{
  puts("Erro de sintaxe");
  exit(1);
  }
```

Chapter 3

Análise Sintática Descendente Não Recursiva

3.1 Introdução

Um analisador descendente não recursivo utiliza uma pilha e uma tabela durante a análise.[1] A tabela é preenchida antes da análise de acordo com um método que será estudado posteriormente. A pilha contém, durante a análise, símbolos terminais e não terminais.

Um único analisador, mostrado na Figura 3.1, é utilizado para todas as linguagens, embora a tabela seja dependente da gramática. Dada a gramática

1. E ::= T E′ 2. E′ ::= + T E′ 3. E′ ::= ϵ
4. T ::= F T′ 5. T′ ::= $*$ F T′ 6. T′ ::= ϵ
7. F ::= (E) 8. F ::= id

onde +, $*$, (,) e id são terminais, a tabela associada, que chamaremos de `M`, é

	id	(	)	+	$*$	eof
E	1	1	–	–	–	–
E′	–	–	3	2	–	3
T	4	4	–	–	–	–
T′	–	–	6	6	5	6
F	8	7	–	–	–	–

O símbolo – deve ser lido 0 (zero).

O algoritmo da Figura 3.1 será utilizado a seguir para analisar a sentença

id + (id $*$ id)

A cada passo da análise, mostraremos a pilha, os elementos da entrada ainda não analisados e a ação a ser tomada. Na pilha, o topo é colocado mais à esquerda. Assim, T estará no topo se a pilha for

E′ T

Após uma operação desempilha, teremos

E′

[1] Parte do texto e os exemplos deste capítulo foram retirados de [6] e [4].

```
void analiseNaoRecursiva()

{
    /* Analisa um programa cujos tokens sao fornecidos pelo objeto
       lex. A tabela de analise é a matriz M, ambos globais. */

  Pilha pilha;

  pilha.crie();
  pilha.empilhe(S);  // S é o simbolo inicial

  while ( ! pilha.vazia() )
    if ( pilha.getTopo() == lex->token ) {
      pilha.desempilha();
      lex->nextToken();
      }
    else
      if ( M[ pilha.getTopo(), lex->token ] == 0 )
        ce->signal (erro_de_sintaxe_err);
      else {
        P = pilha.desempilha();
        empilhe todos os simbolos do lado direito da producao
        M[P, lex->token], da direita para a esquerda
        }

  if ( lex->token != eof_smb )
    ce->signal ( erro_de_sintaxe_err );
  else
    aceite a entrada

  } // analiseNaoRecursiva
```

Figure 3.1: Algoritmo de análise não recursiva

Ao empilhar o lado direito de uma regra, como F T′ da regra T ::= F T′, primeiro inverteremos os símbolos, obtendo T′ F, e então acrescentaremos o resultado à pilha, resultando em E′ T′ F.

Com isto, simulamos a derivação

T E′ $\Longrightarrow$ F T′ E′

Primeiro o símbolo mais à esquerda na sentença a ser derivada, T, é removido da pilha e então substituído pelo lado esquerdo da regra T ::= F T′.

A sentença T E′ é representada na pilha como E′ T. Com a derivação T E′ $\Longrightarrow$ F T′ E′, o topo T é desempilhado e os símbolos T′ F são empilhados, nesta ordem.

Estudaremos agora a análise da sentença

```
id + (id*id)
```

utilizando o algoritmo da Figura 3.1. Cada passo da análise corresponde à aplicação de uma das regras descritas a seguir.

- emp r_i

  Para desempilhar o topo da pilha e empilhar o lado direito da regra i. Esta regra é `M[pilha.getTopo(), lex->token]`.

- desemp

  Para desempilhar o topo da pilha e passar para o próximo símbolo. Esta ação será tomada quando o topo da pilha for um terminal igual ao *token* corrente da entrada.

- aceita

  Para considerar a sentença válida. Este passo será utilizado quando a entrada for eof e a sentença a ser derivada for ϵ.

- erro

  Quando o topo da pilha for terminal diferente do *token* corrente da entrada.

A análise da sentença id + (id∗id) é detalhada na Figura 3.2.

Análise sintática descendente não recursiva pode ser utilizada com gramáticas que obedecem às mesmas restrições da análise recursiva: não ser ambígua, estar fatorada à esquerda e não ter recursão à esquerda.

3.2 A Construção da Tabela `M`

Para construir a tabela `M` utilizamos as relações `first` e `follow`, descritas a seguir.

`first`(α) é o conjunto dos terminais que iniciam α em uma derivação. Assim, utilizando a gramática

E ::= T E′
E′ ::= + T E′ | ϵ
T ::= N T′
T′ ::= ∗ N T′ | ϵ

temos que

`first` (N) = { N }
`first` (T′) = { ∗, ϵ }

Pilha	Entrada	Ação
E	$\underline{\text{id}}$ + (id∗id)	emp r_1
E′ T	$\underline{\text{id}}$ + (id∗id)	emp r_4
E′ T′ F	$\underline{\text{id}}$ + (id∗id)	emp r_8
E′ T′ id	$\underline{\text{id}}$ + (id∗id)	desemp
E′ T′	$\underline{+}$ (id∗id)	emp r_6
E′	$\underline{+}$ (id∗id)	emp r_2
E′ T+	$\underline{+}$ (id∗id)	desemp
E′ T	$\underline{(}$ id∗id)	emp r_4
E′ T′ F	$\underline{(}$ id∗id)	emp r_7
E′ T′) E (	$\underline{(}$ id∗id)	desemp
E′ T′) E	$\underline{\text{id}}$ ∗id)	emp r_1
E′ T′) E′ T	$\underline{\text{id}}$ ∗id)	emp r_4
E′ T′) E′ T′ F	$\underline{\text{id}}$ ∗id)	emp r_8
E′ T′) E′ T′ id	$\underline{\text{id}}$ ∗id)	desemp
E′ T′) E′ T′	$\underline{*}$ id)	emp r_5
E′ T′) E′ T′ F ∗	$\underline{*}$ id)	desemp
E′ T′) E′ T′ F	$\underline{\text{id}}$)	emp r_8
E′ T′) E′ T′ id	$\underline{\text{id}}$)	desemp
E′ T′) E′ T′	$\underline{)}$	emp r_6
E′ T′) E′	$\underline{)}$	emp r_3
E′ T′)	$\underline{)}$	desemp
E′ T′	eof	emp r_6
E′	eof	emp r_3
ϵ	eof	aceita

Figure 3.2: A análise da sentença id + (id∗id)

`first` (T) = { N }
`first` (E′) = { +, ϵ }
`first` (E) = { N }

A função `first` é definida formalmente como:

`first`: $(\mathrm{N} \cup \Sigma)^* \longrightarrow \mathcal{P}(\Sigma \cup \{\epsilon\})$
`first`(α) = $\{\mathrm{a} \mid \alpha \stackrel{*}{\Longrightarrow} \mathrm{a}\beta, \mathrm{a} \in \Sigma \cup \{\epsilon\}\}$

$\mathcal{P}(A)$ é o conjunto de todas as partes do conjunto A. Se $A = \{a, b\}$, $\mathcal{P}(A) = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$. Se A é um conjunto, A^* é o conjunto de todas as strings, cadeias ou sentenças[2] compostas por símbolos de A. Por exemplo, se $A = \{a, b\}$, $A^* = \{a, aa, aba, babb, \ldots\}$. O símbolo $(\mathrm{N} \cup \Sigma)^*$ significa o conjunto de todas as cadeias composta por símbolos terminais (Σ) e não terminais (N).

`follow`(X) é o conjunto de terminais que sucedem X em qualquer derivação partindo do símbolo inicial S.

Se existir uma produção

Y ::= αXa

então a $\in$ `follow`(X). O mesmo será válido se tivermos

Y ::= αXZ
Z ::= aH | bG

e neste caso também teremos b $\in$ `follow`(X).

Formalmente, `follow` é definida como

`follow`: $(\mathrm{N} \cup \Sigma) \longrightarrow \mathcal{P}(\Sigma \cup \{\mathrm{eof}\})$
`follow`(X) = $\{\mathrm{a} \in \Sigma \cup \{\mathrm{eof}\} \mid \mathrm{S} \stackrel{*}{\Longrightarrow} \alpha\mathrm{Xa}\beta$ com $\alpha, \beta \in (\mathrm{N} \cup \Sigma)^*\}$

`first`(X) é computado como:

1. se X for um terminal, `first`(X) = { X };
2. se X ::= ϵ for uma produção, adicione ϵ a `first`(X);
3. se X for não terminal e X ::= $\mathrm{Y}_1\mathrm{Y}_2$... Y_k for uma produção, então coloque f em `first`(X) se, para algum i, f $\in$ `first`(Y_i) e $\epsilon \in$ `first`(Y_j), j = 1, 2,...$i-1$. Se $\epsilon \in$ `first`(Y_j), j = 1, 2,... k, então coloque ϵ em `first`(X). Este item pode ser explicado como se segue.

   Se Y_1 não derivar ϵ, então adicione `first`(Y_1) a `first`(X). Se $\mathrm{Y}_1 \Longrightarrow \epsilon$, adicione `first`($\mathrm{Y}_2$) a `first`(X). Se $\mathrm{Y}_1 \Longrightarrow \epsilon$ e $\mathrm{Y}_2 \Longrightarrow \epsilon$, adicione `first`($\mathrm{Y}_3$) a `first`(X) e assim por diante.

`follow`(X) é computado como:

1. coloque eof em `follow`(S), onde S é o símbolo inicial;
2. se houver uma produção da forma A ::= αXβ, então `first`(β), exceto ϵ, é colocado em `follow`(X);
3. se houver uma produção da forma A ::= αX ou uma produção A ::= αXβ e $\epsilon \in$ `first`(β), então colocaremos os elementos de `follow`(A) em `follow`(X).

A relação `first` deverá ser calculada antes de `follow` (por quê ?). Para encontrar `first`(X) para todo X $\in$ N $\cup$ Σ (todos os símbolos da gramática), aplique as regras acima até que nenhum terminal ou ϵ possa ser adicionado a nenhum conjunto `first`(Y). Isto é necessário porque a adição de elementos a um conjunto pode implicar, pela regra 3 de `first`, na adição de elementos a outros conjuntos.

[2] Aqui utilizados como sinônimos.

Para calcular `follow`(A) para A ∈ N ∪ {eof} (não terminais e eof), aplique as regras anteriores até que nenhum elemento possa ser adicionado em qualquer conjunto `follow`(B).

Calcularemos as relações `first` e `follow` para a gramática apresentada na introdução deste capítulo, que é

1. E ::= T E′
2. E′ ::= + T E′
3. E′ ::= ϵ
4. T ::= F T′
5. T′ ::= ∗ F T′
6. T′ ::= ϵ
7. F ::= (E)
8. F ::= id

Os símbolos +, ∗, (,) e id são terminais. Observe que esta gramática obedece as restrições para a análise não recursiva, que são: não ser ambígua, estar fatorada à esquerda e não ter recursão à esquerda.

A relação `first` é:

`first`(E) = { id, (}
`first`(E′) = { +, ϵ }
`first`(T) = { id, (}
`first`(T′) = { ∗, ϵ }
`first`(F) = { (, id }

Note que não calculamos a função `first` completamente. Esta função se aplica a qualquer cadeia de (N ∪ Σ)∗. Então podemos calcular `first`(T E′) ou `first`(∗ F T′). No primeiro caso, `first`(T E′) = `first`(T), pois $\epsilon \notin$ `first`(T). No segundo caso, temos `first`(∗ F T′) = ∗, pois ∗ é um terminal.

Os últimos conjuntos são calculados antes dos outros porque deles dependem os primeiros: `first`(A) dependerá de `first`(B) se houver uma produção A ::= Bα.

A relação `follow` é:

`follow`(E) = { eof,) }
`follow`(E′) = { eof,) }
`follow`(T) = { +, eof,) }
`follow`(T′) = { +, eof,) }
`follow`(F) = { ∗, +, eof,) }

A relação `follow` para os terminais não foi calculada, apesar de ser válida, porque não será utilizada no exemplo a seguir.

A tabela `M` utilizada pelo algoritmo de análise sintática não recursiva é calculada como se segue. Para cada produção da forma A ::= α, faça:

1. para cada a ∈ `first`(α), adicione o número da regra
   A ::= α
   em `M[A, a]`;

2. Se $\epsilon \in$ `first`(α), adicione o número da regra
   A ::= α
   em `M[A, b]` para cada b ∈ `follow`(A);

Torne indefinidas todas as outras entradas:

`M[A, a]` = 0

A tabela construída de acordo com estas regras foi apresentada na introdução deste capítulo.

3.3 Gramáticas LL(1)

Seja G uma gramática sem recursão à esquerda e fatorada à esquerda a partir da qual foi construída uma tabela `M` pelo método apresentado na seção anterior. Se G for ambígua, haverá uma entrada `M`[X, a] com mais de um elemento. Por exemplo, considere a gramática

1. S ::= A
2. S ::= B
3. A ::= a C
4. B ::= ab
5. C ::= b

Podemos obter a sentença "ab" com duas derivações à esquerda diferentes:

S $\Longrightarrow$ A $\Longrightarrow$ a C $\Longrightarrow$ a b

S $\Longrightarrow$ B $\Longrightarrow$ a b

Isto caracteriza a ambigüidade. A tabela de análise desta gramática é

	a	b	eof
S	1, 2		
A	3		
B	4		
C		5	

Como `M`[S, a] é { 1, 2 }, então poderemos utilizar a produção 1 ou 2 para expandir S quando o símbolo corrente da entrada for "a".

Contudo, uma gramática pode ter mais de um número em certa entrada da tabela e não ser ambígua. Por exemplo, a gramática

1. S ::= A
2. S ::= B
3. A ::= a C
4. B ::= a
5. C ::= b

possui a mesma tabela de análise que a gramática anterior, mas não é ambígua. É fácil ver porquê. A linguagem desta gramática só possui "ab" e "a" como sentenças (a gramática só produz estas sentenças) e só existe uma forma de produzir cada uma delas:

S $\Longrightarrow$ A $\Longrightarrow$ a C $\Longrightarrow$ a b

S $\Longrightarrow$ B $\Longrightarrow$ a

A gramática

1. S ::= "if" E "then" S S′
2. S ::= a

3. S′ ::= “else” S
4. S′ ::= ϵ
5. E ::= b

tomada de Aho et al. [4] possui a seguinte tabela de análise:

	a	b	“else”	“if”	“then”	eof
S	2			1		
S′			3, 4			4
E		5				

A entrada `M`[S′, “else”] possui dois elementos, implicando em considerar o “else” da sentença

```
if b then a if b then a else a
```

associado ao último `if` (3) ou ao primeiro (4).

Uma gramática será chamada de LL(1) se a sua tabela de análise tiver no máximo um elemento por cada entrada `M`[X, a]. O primeiro L de “LL(1)” informa que a análise para a gramática é feita da esquerda (*Left*) para a direita. O segundo L implica que, na análise, é produzida a derivação mais à esquerda sendo que em cada passo da derivação a ação a ser tomada é decidida com base em um (“1” de “LL(1)”) símbolo da entrada (o *lookahead*).

Pode ser provado que uma gramática G é LL(1) se e somente se sempre que A ::= α | β forem duas produções distintas de G, os itens a seguir são verdadeiros.[3]

1. Não há um terminal a pertencente ao mesmo tempo a first(α) e first(β).
2. Ou $\alpha \stackrel{*}{\Longrightarrow} \epsilon$ ou $\beta \stackrel{*}{\Longrightarrow} \epsilon$, nunca estas duas condições ao mesmo tempo.
3. Se $\beta \stackrel{*}{\Longrightarrow} \epsilon$, então α não derivará qualquer *string* começando com um terminal que pertence a `follow`(A).

Discutiremos porque a desobediência a um destes itens implica na ambigüidade da gramática.

1. Se $\alpha \stackrel{*}{\Longrightarrow}$ a e $\beta \stackrel{*}{\Longrightarrow}$ a puderem ocorrer e tivermos que expandir A com a como o símbolos corrente da entrada (*lookahead*), teremos duas produções a escolher: A ::= α e A ::= β.
2. Se for possível $\alpha \stackrel{*}{\Longrightarrow} \epsilon$ e $\beta \stackrel{*}{\Longrightarrow} \epsilon$ e tivermos que expandir A com b como símbolo corrente de entrada, b $\notin$ `first`(A), teremos novamente duas produções escolher.
3. Se β puder derivar ϵ ($\beta \stackrel{*}{\Longrightarrow} \epsilon$), b $\in$ `follow`(A) e α puder derivar b ($\alpha \stackrel{*}{\Longrightarrow}$ bγ), então tanto A ::= α quanto A ::= β poderão ser escolhidas para expandir A quando b for o símbolo corrente da entrada. É claro que a utilização de A ::= α seria correta, já que b $\in$ first(α). Para enxergar porque a utilização de A ::= β produziria o resultado correto, considere o seguinte exemplo: γAϕ está sendo expandido para reconhecer a entrada, b é o token corrente, γ já foi reconhecido, A deve ser derivado, b $\in$ first(ϕ) e, portanto, b $\in$ follow(A). Escolhendo A ::= β e $\beta \stackrel{*}{\Longrightarrow} \epsilon$, teríamos γA$\phi \stackrel{+}{\Longrightarrow} \gamma\phi$ onde b seria reconhecido por ϕ.

Nem todas as gramáticas são ou podem ser transformadas em LL(1). Este é o principal obstáculo ao uso de analisadores preditivos. Felizmente, o problema de entradas na tabela com mais de um elemento pode ser resolvido escolhendo-se um deles, removendo assim a ambigüidade.

[3] Todo este trecho é uma tradução quase literal de um parágrafo da página 192 de Aho et al. [4].

Analisadores ascendentes (*botton-up*), que derivam o símbolo inicial a partir dos terminais, podem ser utilizados com um número maior de gramáticas do que analisadores descendentes preditivos. Estes analisadores são, em geral, gerados automaticamente por *geradores de analisadores sintáticos* a partir da gramática da linguagem. Analisadores ascendentes empregam grandes tabelas que seriam difíceis de serem gerados por pessoas sem o auxílio de programas. Como exemplos de geradores de analisadores sintáticos, temos o CUP, JavaCC, YACC, Bison, ANTLR.

Bibliography

[1] Stroustrup, Bjarne. *The C++ Programming Language*. Second Edition, Addison-Wesley, 1991.

[2] Lippman, Stanley B. *C++ Primer*. Addison-Wesley, 1991.

[3] Deitel, H.M. e Deitel P.J. *C++ How to Program*. Prentice-Hall, 1994.

[4] Aho, Alfred e Sethi, Rave e Ullman, Jeffrey. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, 1986.

[5] Gamma, Erich; Helm, Richard; Johnson, Ralph e Vlissides, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series, Addison-Wesley, Reading, MA, 1994.

[6] Zorzo, Sérgio Donizetti. Notas de aula de Construção de Compiladores, 1995.

[7] Pittman, Thomas; Peter, James. *The Art of Compiler Design: Theory and Practice*. Prentice Hall, 1992.

[8] Guimarães, José de Oliveira. *The Green Language*. Disponível em http://www.dc.ufscar.br/ jose/green/green.htm.