# Construção de Compiladores

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

March 15, 2010

# 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

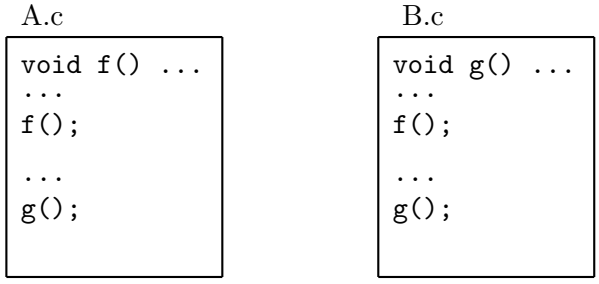


Figure 1.1: Dois arquivos em c que formam um programa

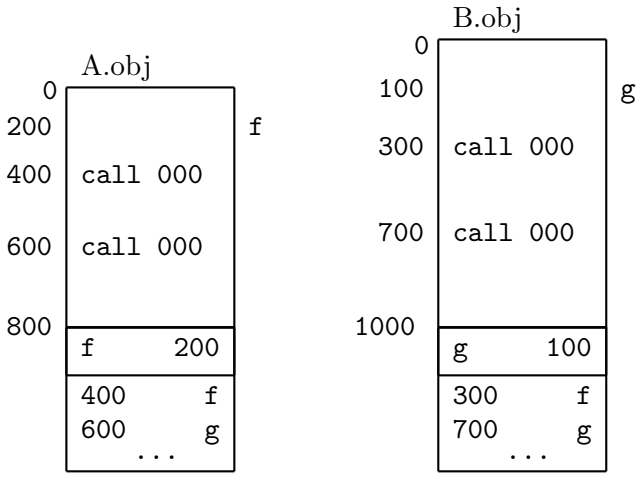


Figure 1.2: Configuração dos arquivos objeto

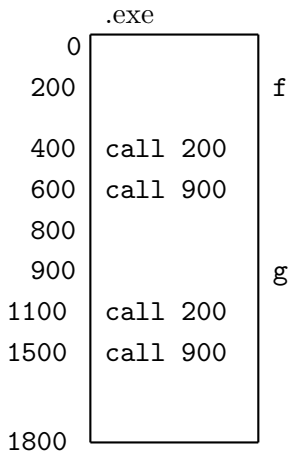


Figure 1.3: Configuração do arquivo executável

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.

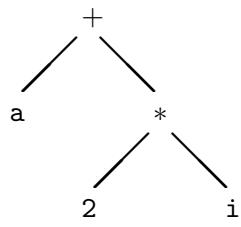

Figure 1.4: Árvore de sintaxe da sentença `a + 2*i`

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 $\text{S} \overset{*}{\Longrightarrow} \alpha$, $\alpha$ 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 $\alpha_i$ $(1 \leq i \leq n)$ 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.

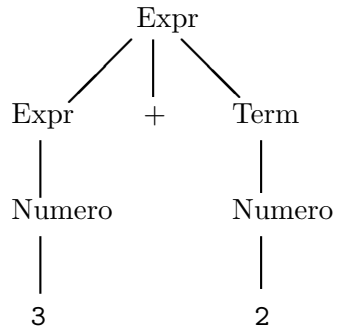


Figure 2.1: Árvore de análise de "3 + 2"

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 ::= $\beta$ $\alpha_1$ | $\beta$ $\alpha_2$

onde $\beta$, $\alpha_1$ e $\alpha_2$ são seqüências de terminais e não-terminais. Uma gramática com produções

A ::= $\beta_1$ | $\beta_1$ $\alpha_1$

A ::= $\beta_2$ | $\beta_2$ $\alpha_2$

pode ser fatorada à esquerda transformando estas produções em

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

$X_1$ ::= $\epsilon$ | $\alpha_1$

$X_2$ ::= $\epsilon$ | $\alpha_2$

onde $\epsilon$ 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 $\alpha$ de terminais e não-terminais.

Se uma gramática tiver produções

A ::= A$\alpha_1$ | A$\alpha_2$ | ... | A$\alpha_m$ | $\beta_1$ | $\beta_2$ | ... | $\beta_n$

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

A ::= $\beta_1$A′ | $\beta_2$A′ | ... | $\beta_n$A′

A′ ::= $\alpha_1$A′ | $\alpha_2$A′ | ... | $\alpha_m$A′ | $\epsilon$

Observe que as primeiras produções geram uma seqüência de zero ou mais $\alpha_i$ iniciada por um único $\beta_j$, como

$\beta_3\alpha_3\alpha_1\alpha_5$

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

$\beta_2$

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

A ::= $\beta_j$ A′

e que haverá uma repetição de um ou mais $\alpha_i$ por causa das produções

A′ ::= $\alpha_i$A′

A regra acima não funcionará quando A derivar A$\alpha$ 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, $\alpha_1$ e $\alpha_2$ iguais a "+ Term" e "− Term" e $\beta_1$ igual a Term. O resultado é

Expr ::= Term Expr′

Expr′ ::= + Term Expr′ | − Term Expr′ | $\epsilon$

fazendo o mesmo com Term, obtemos

Term ::= Numero Term′

Term′ ::= ∗ Numero Term′ | / Numero Term′ | $\epsilon$

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′ | $\epsilon$
T ::= N T′
T′ ::= ∗ N T′ | $\epsilon$
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:

2 + 3$
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:

2 + 3$
T E′

Existe também uma única alternativa para T:

2 + 3$
N T′ E′

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

2 + 3$
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′ | $\epsilon$), escolhemos derivar T' para $\epsilon$ 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′ ::= $\epsilon$ 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 $\epsilon$:

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 ::= $\alpha_1$ | $\alpha_2$ | ... | $\alpha_n$

que começa com a *string* "a". Se uma gramática não tiver nenhuma produção do tipo A ::= $\epsilon$, 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 | $\epsilon$
Term ::= Numero Term2
Term2 ::= ∗ Numero Term2 | / Numero Term2 | $\epsilon$
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′ ::= $\epsilon$
4. T ::= F T′ 5. T′ ::= $*$ F T′ 6. T′ ::= $\epsilon$
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 $\epsilon$.

- 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`($\alpha$) é o conjunto dos terminais que iniciam $\alpha$ em uma derivação. Assim, utilizando a gramática

E ::= T E′
E′ ::= + T E′ | $\epsilon$
T ::= N T′
T′ ::= ∗ N T′ | $\epsilon$

temos que

`first` (N) = { N }
`first` (T′) = { ∗, $\epsilon$ }

| 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 $r_3$ |
| $\epsilon$ | eof | aceita |

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

`first` (T) = { N }
`first` (E′) = { +, $\epsilon$ }
`first` (E) = { N }

A função `first` é definida formalmente como:

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

$\mathcal{P}(A)$ 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 ($\Sigma$) 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 ::= $\alpha$Xa

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

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

   Se $\mathrm{Y}_1$ não derivar $\epsilon$, então adicione `first`($\mathrm{Y}_1$) 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 ::= $\alpha$X$\beta$, então `first`($\beta$), exceto $\epsilon$, é colocado em `follow`(X);
3. se houver uma produção da forma A ::= $\alpha$X ou uma produção A ::= $\alpha$X$\beta$ e $\epsilon \in$ `first`($\beta$), 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 $\cup$ $\Sigma$ (todos os símbolos da gramática), aplique as regras acima até que nenhum terminal ou $\epsilon$ 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 $\in$ N $\cup$ {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′ ::= $\epsilon$
4. T ::= F T′
5. T′ ::= $*$ F T′
6. T′ ::= $\epsilon$
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′) = { +, $\epsilon$ }
`first`(T) = { id, ( } Note que não calculamos a função `first` completamente. Esta
`first`(T′) = { $*$, $\epsilon$ }
`first`(F) = { (, id }

função se aplica a qualquer cadeia de (N $\cup$ $\Sigma$)$^*$. 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 ::= B$\alpha$.

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 ::= $\alpha$, faça:

1. para cada a $\in$ `first`($\alpha$), adicione o número da regra
   A ::= $\alpha$
   em `M[A, a]`;

2. Se $\epsilon \in$ `first`($\alpha$), adicione o número da regra
   A ::= $\alpha$
   em `M[A, b]` para cada b $\in$ `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′ ::= $\epsilon$
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 ::= $\alpha$ | $\beta$ 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($\alpha$) e first($\beta$).
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 $\alpha$ 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 ::= $\alpha$ e A ::= $\beta$. Por exemplo,

   A ::= B — C
   B ::= Dd — d
   C ::= cC — f
   D ::= c — b

   Neste caso, B $\stackrel{*}{\Longrightarrow}$ c e C $\stackrel{*}{\Longrightarrow}$ c. Se c for o símbolo corrente e tivermos que expandir A, podemos escolher A $\Longrightarrow$ B ou A $\Longrightarrow$ C:
   A $\Longrightarrow$ B $\Longrightarrow$ Dd $\Longrightarrow$ cd
   ou A $\Longrightarrow$ C $\Longrightarrow$ cC

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. Veja o exemplo abaixo.

   1. S ::= Ae
   2. A ::= B | C

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

3. B ::= b | $\epsilon$
4. C ::= c | $\epsilon$

Se e for o símbolo corrente e A deve ser expandido, podemos escolher tanto a produção A ::= B como A ::= C:
S $\Longrightarrow$ Ae $\Longrightarrow$ Be $\Longrightarrow$ e
S $\Longrightarrow$ Ae $\Longrightarrow$ Ce $\Longrightarrow$ e

3. Se $\beta$ puder derivar $\epsilon$ ($\beta \stackrel{*}{\Longrightarrow} \epsilon$), b $\in$ `follow`(A) e $\alpha$ puder derivar b ($\alpha \stackrel{*}{\Longrightarrow}$ b$\gamma$), então tanto A ::= $\alpha$ quanto A ::= $\beta$ poderão ser escolhidas para expandir A quando b for o símbolo corrente da entrada. É claro que a utilização de A ::= $\alpha$ seria correta, já que b $\in$ first($\alpha$). Para enxergar porque a utilização de A ::= $\beta$ produziria o resultado correto, considere o seguinte exemplo: $\gamma$A$\phi$ está sendo expandido para reconhecer a entrada, b é o token corrente, $\gamma$ já foi reconhecido, A deve ser derivado, b $\in$ first($\phi$) e, portanto, b $\in$ follow(A). Escolhendo A ::= $\beta$ e $\beta \stackrel{*}{\Longrightarrow} \epsilon$, teríamos $\gamma$A$\phi \stackrel{+}{\Longrightarrow} \gamma\phi$ onde b seria reconhecido por $\phi$. Veja o exemplo abaixo.

   1. S ::= Ae
   2. A ::= B | C
   3. B ::= e
   4. C ::= c | $\epsilon$

   Se e for o símbolo corrente e estivermos expandindo S, temos duas opções:
   S $\Longrightarrow$ Ae $\Longrightarrow$ Be $\Longrightarrow$ ee
   S $\Longrightarrow$ Ae $\Longrightarrow$ Ce $\Longrightarrow$ e

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.

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.

# Chapter 4

# Geração de Código

## 4.1 Introdução

Estudaremos a geração de código para linguagem S2 a partir da ASA construída durante a análise sintática. O código será gerado na linguagem C. A tradução de S2 para C é trivial e será mostrada através de um exemplo. O programa S2 dado a seguir,

```
var
   i, n : integer;
   ok : boolean;
begin
ok = false;
read(n);
if (n > 0) and not ok
then
  i = 1;
  while i <= n do
    begin
    write(i);
    i = i + 1;
    end
  ok = true;
else
  if (n < 0) or ok
  then
    write(-n);
  else
    write(0);
  endif
endif
end
```

deve ser traduzido para o seguinte programa em C:

```
#include <stdio.h>
void main()
{
```

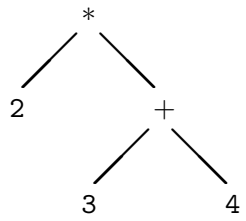

Figure 4.1: Asa da expressão `2*(3 + 4)`

```
const
  MaxCh = 127;
int i, n, ok;
char s[MaxCh + 1];

ok = 0;
gets(s);
sscanf(s,"%d",&n);
if ( n > 0 && ! ok ) {
  i = 1;
  while ( i <= n ) {
    printf("%d ",i);
    i = i + 1;
    }
  ok = 1;
  }
else {
  if ( n < 0 || ok ) {
    printf("%d ",-n);
    }
  else {
    printf("%d ",0);
    }
  }
}
```

A tradução de S2 para C pode ser feita colocando-se um método `gera` em cada classe da ASA ou aplicando-se o padrão Visitante. Neste último caso, criamos uma subclasse `VisitanteGera` da classe `Visitante` com os métodos para gerar código para cada classe da ASA. Não entraremos em mais detalhes sobre a tradução porque ela é muito semelhante à impressão do código fonte a partir da ASA vista nos capítulos anteriores. Veremos agora alguns detalhes da geração do código em C, mas em alto nível.

A variável `s` na tradução para C somente será declarada se houver um comando `read` no programa S2.

Uma expressão composta em S2 será transformada em um objeto da classe `CompositeExpression_no` sem nenhum indicativo se a expressão estava entre parênteses ou não. Assim, a expressão

```
2*(3 + 4)
```

será transformada na ASA mostrada na Figura 4.1.

Se o código para um objeto de `CompositeExpression_no` for gerado imprimindo-se a expressão à esquerda, o operador e a expressão à direita, o código gerado para a expressão `2*(3 + 4)` será

```
2*3 + 4
```

com o significado incorreto. Este erro pode ser corrigido:

- colocando-se uma variável em `CompositeExpression_no` informando se a expressão está entre parênteses no programa S2;
- colocando-se parênteses em todas as expressões compostas. Neste caso, o código gerado para a expressão acima seria

```
(2*(3 + 4))
```

Colocamos os comandos da parte `then` e `else` dos `if`'s entre chaves. Se houver um único comando na parte `then` as chaves podem não ser necessárias em alguns casos, embora elas o sejam no trecho de código mostrado a seguir.

```
if i > 0
then
   if i > 5
   then
     i = i - 1;
   endif
else
   i = 1;
endif
```

Este código seria traduzido incorretamente para o trecho em C dado abaixo se chaves não fossem empregadas.

```
if ( i > 0 )
  if ( i > 5 )
    i = i - 1;
else
   i = 1;
```

Entretanto, se houver um único comando na parte `else`, as chaves são desnecessárias, pois nunca haverá ambigüidades.

## 4.2 Uma Linguagem Assembler

Definiremos uma pequena linguagem assembler para dar suporte às discussões das seções seguintes sobre geração de código para assembler.

Utilizaremos uma máquina com uma pilha apontada pelo registrador `sp` cujo topo da pilha é `sp[t]`, onde `t` é um registrador contendo um número inteiro. Cada posição da pilha, como `sp[0]` ou `sp[1]` e cada um dos oito registradores `R0`, `R1`, ... `R7` pode armazenar um número inteiro, booleano ou um ponteiro para a memória. O registrador `bp` é utilizados como um ponteiro para a pilha e pode ser indexado como um vetor em C: `bp[0]` referir-se-á ao primeiro elemento da pilha se `bp` for igual a `sp`.

Todas as variáveis de um programa assembler são alocadas na pilha e referenciadas indexando-se `bp` ou `sp`:

```
    mov sp[0], 2
    mov sp[1], bp[2]
```

Na tradução de S2 para assembler, utilizaremos apenas `sp` para manipular variáveis. As variáveis locais são alocadas na pilha nas posições `sp[0]`, `sp[1]`, `sp[2]`, etc.

Na declaração das variáveis do programa

```
  var a, b, c: integer;
begin
   ...
end
```

as variáveis `a`, `b` e `c` seriam referenciadas como `sp[0]`, `sp[1]` e `sp[2]` no assembler. Contudo, utilizaremos os nomes `a`, `b` e `c` também no assembler, por uma questão de simplicidade.

Detalhamos a seguir todas as instruções do assembler. Os identificadores `a` e `b` utilizados nos exemplos referem-se a variáveis do assembler ou a registradores.

`not a`
Descrição: `!a`

`add a, b`
Descrição: `a` = `a` + `b`

`sub a, b`
Descrição: `a` = `a` − `b`

`mult a, b`
Descrição: `a` = `a` ∗ `b`

`div a, b`
Descrição: `a` = `a` / `b`

`cmp a, b`
Descrição: compara `a` com `b`, inicializando o registrador `cm` com:

- -1 se `a < b`;
- 0 se `a == b` ou;
- 1 se `a > b`.

O registrador `cm` é inicializado por esta instrução e utilizada pelas instruções de desvio incondicional (`goto>`, etc). Ele não é manipulado diretamente pelo programador.

`goto> L`, `goto>= L`, `goto< L`, `goto<= L`, `goto<> L`, `goto== L`

Descrição: Após comparar dois valores com `cmp`, cada uma destas instruções causará um desvio para o *label* `L` se o resultado da comparação for verdadeiro. Por exemplo, em

```
    cmp X, 3
    goto> L1
```

a última instrução causará um desvio para `L1` se `X > 3`.

`goto L`
Descrição: desvia para `L`

`goto v[i]`
Descrição: `v` é um ponteiro utilizado como vetor onde cada elemento do vetor é um endereço de uma instrução de programa (*label*). Esta instrução desvia a execução do programa para a posição `v[i]`.

`push X`
Descrição: empilha `X` na pilha, o que seria equivalente em C++ a fazer `sp[++t] = X`.

`pop X`
Descrição: coloca o topo da pilha em `X`, decrementando `t`:

```
X = sp[t--];
```

`call L`
Descrição: chama o procedimento do *label* `L`.

`call X`
Descrição: chama o procedimento cujo endereço está na variável `X`.

`ret`
Descrição: retorna do procedimento.

`mov a, b`
Descrição: copia `b` em `a`

`mov a, b[i]`
Descrição: copia `b[i]` em `a`, onde `b[i]` é uma indexação de um vetor

`mov b[i], a`
Descrição: copia `a` em `b[i]`

`mov a, -b`
Descrição: copia `-b` em `a`

`mov a, &b`
Descrição: copia o endereço de `b` em `a`. `b` deve ser uma variável. Se `b` for `sp[2]`, por exemplo, esta instrução copiará `sp + 2` em `a`.

`mov a, *b`
Descrição: copia em `a` o valor contido no endereço apontado por `b`.

`exit`
Descrição: termina a execução do programa e volta o controle ao sistema operacional.

## 4.3 Geração de Código para S2

Esta seção define como código em assembler deve ser gerado para programas em S2. Apresentamos a seguir as regras da gramática com o código que deve ser gerado para cada uma delas. Algumas regras não causam geração de código e, portanto, não são citadas. A geração de código não é descrita de maneira rigorosa pois isto está fora do escopo deste curso.

1. Assignment ::= Id "=" Expression

   Se Expression for uma variável, como em
   `a = b;`

o código gerado será

```
mov a, b
```

Se Expression tiver algum operador aritmético ou lógico (isto é, não for uma variável), como em

```
a = b*c + 3;
```

o valor da expressão será colocado em uma variável temporária que chamaremos de `t1` e o código gerado será

```
mov a, t1
```

2. Expression ::= SimpleExpression [ Relation SimpleExpression ]

Na expressão

```
a*b < b + c
```

considere que o valor das expressões `a*b` e `b + c` tenham sido colocados em `t1` e `t2`, duas variáveis temporárias. Então, o código gerado será

```
     cmp t1, t2
     goto< L1
     mov t3, 0
     goto L2
L1:  mov t3, 1
L2:
```

O resultado da expressão foi colocado em `t3`. Se esta expressão estiver em um `if` ou `while`, o resultado não precisa ser colocado em uma variável — veja as regras para estes comandos.

3. Factor ::= not Factor

A expressão

```
not Factor
```

será gerada como

```
not t1
```

admitindo que o resultado de Factor foi colocado em `t1`.

4. IfStat ::= “if” Expression “then” StatementList
[ “else” StatementList ] “endif”

Estudaremos a geração de código de um `if` sem `else`:

```
if a < b
then
  S
endif
```

onde `S` é uma seqüência de comandos. O código gerado será

```
     cmp a, b
     goto>= L1
```

```
     codigo para S
L1:
```

O código para

```
if a < b
then
  S1
else
  S2
endif
```

será

```
     cmp a, b
     goto>= L1
     codigo para S1
     goto L2
L1:  codigo para S2
L2:
```

5. ReadStat ::= “read” “(” IdList “)”

Admitiremos que existe uma função `_read` do sistema de tempo de execução que lê um (e apenas um) inteiro da entrada padrão e o coloca no registrador `R0`. Deste modo, o comando

```
     read(a, b, c);
```

será traduzido para

```
     call _read
     mov a, R0
     call _read
     mov b, R0
     call _read
     mov c, R0
```

6. SimpleExpression ::= [ Signal ] Term { LowOperator Term }
Term ::= Factor { HighOperator Factor }

O resultado de expressões aritméticas são colocadas em variáveis temporárias. Assim, o código

```
  a = b*c + 3
```

será traduzido para

```
     mov t1, b
     mult t1, c
     add t1, 3
     mov a, t1
```

que poderia ser otimizado para

```
mov a, b
mult a, c
add a, 3
```

já que a variável `a` não aparece na expressão.

Uma negação

```
a = -b
```

será traduzida para

```
mov a, -b
```

Em geral, expressões mais complexas, como

```
a = a*b + c*d
```

necessitarão de mais de uma variável temporária:

```
mov t1, a
mult t1, b
mov t2, c
mult t2, d
add t1, t2
mov a, t1
```

7. WriteStat ::= "write" "(" ExpressionList ")"

Cada expressão de ExpressionList será colocada em uma variável temporária que é passada como parâmetro à função `_write` do sistema de tempo de execução. Por exemplo,

```
write(a, b*c, d + 1);
```

será traduzido para

```
push a
call _write
mov t1, b
mult t1, c
push t1
call _write
mov t1, d
add t1, 1
call _write
```

Observe que a variável temporária `t1` foi utilizada em `b*c` e reutilizada em `d + 1`.

8. WhileStat ::= "while" Expression "do" UnStatBlock

A geração de código para

```
while i < n do
   S
```

onde `S` é uma lista de comandos entre `begin` e `end`, poderia ser

```
L1:  cmp i, n
     goto>= L2
     codigo para S
```

```
     goto L1
L2:
```

Contudo, existe uma forma mais eficiente:

```
     goto L1
L2:  codigo para S
L1:  cmp i, n
     goto< L2
```

A cada passo do laço, na primeira forma há um "`goto L1`" e um "`goto>= L2`" que falha.[1] Na segunda forma, a cada passo do laço há apenas um "`goto< L2`" que sucede. Mesmo havendo um "`goto L1`" no início da segunda forma, esta é mais eficiente do que a primeira.

## 4.4 Geração de Código para Vetores e Comando `case/switch`

### Geração de Código para Vetores

Suponha que S2 suporte vetores unidimensionais declarados como

```
var v : array(integer)[100];
```

e que podem ser indexados como em C++:

```
a = v[i];   // 1
v[i] = a;   // 2
```

Estas instruções serão traduzidas para

```
mov a, v[i]   // 1
mov v[i], a   // 2
```

em assembler. Se a instrução `mov` não admitisse a indexação de seus operandos, o endereço de `v[i]` teria que ser calculado previamente. Nesta situação, a instrução `1` seria traduzida para

```
mov t1, &v
add t1, i
mov a, *t1
```

onde `t1` é uma variável temporária. No caso geral, o endereço `t1` do `i`-ésimo elemento do vetor é calculado como

```
t1 = &v + i*sizeof(v[0])
```

considerando que a memória é indexada byte a byte. Na máquina que estamos utilizando, a memória é indexada de dois em dois bytes. Isto é, o endereço `t1` representa dois bytes, suficientes para um inteiro e `t1 + 1` também representa dois bytes. Assim, o endereço do `i`-ésimo elemento do vetor `v` é

```
t1 = &v + i;
```

### Geração de Código para o Comando `case/switch`

Suponha que tenha sido acrescentado a S2 um comando `case` da forma

[1] Este desvio condicional sucederá apenas quando o laço for terminar.

```
case expr of
  V1:
    S1;
  V2:
    S2;
  ...
  Vn:
    Sn;
  default:
    Sd
end { case }
```

onde `expr` é uma expressão inteira, `V1`, `V2`, ... `Vn` são constantes literais inteiras e cada `Si` é uma instrução ou uma seqüência de instruções entre `begin` e `end`.

Uma das maneiras de gerar código para este comando é fazer todas as comparações de `expr` com os `Vi` no fim da tradução, como é feito a seguir.

```
     Calcule expr e coloque o resultado em t
     goto testeCase
L1:
     codigo para S1
     goto fim
L2:
     codigo para S2
     goto fim
     ...
Ln:
     codigo para Sn
     goto fim
D:
     codigo para Sd
     goto fim
testeCase:
     cmp t, V1
     goto== L1
     cmp t, V2
     goto== L2
     ...
     cmp t, Vn
     goto== Vn
     goto   D
fim:
```

Os testes poderiam ser colocados logo após o cálculo de `expr` eliminando a necessidade da instrução "`goto testeCase`".

Uma outra alternativa de geração de codigo é colocar os testes junto com a codificação de cada `Si`:

```
L1:  cmp t, V1
     goto<> L2
```

```
codigo para S1
goto fim
...
```

# Chapter 5

# Otimização de Código

Otimizar um código é transformá-lo em um código que faz a mesma coisa mas que é mais rápido que a versão anterior. O código transformado pode ser o código fonte da linguagem, o código intermediário gerado pelo compilador, o código em assembler ou mesmo o programa em forma de árvore de sintaxe abstrata.

Emprega-se o termo "otimizar" não só com relação a aumentar a velocidade de execução do código como também diminuir o seu tamanho. Neste curso, a menos de menção em contrário, otimizar terá o sentido de "aumentar a velocidade de execução". O nome "otimizar" significa "tornar o melhor possível". Como apenas em situações excepcionais o código otimizado por um compilador é o melhor que se pode obter, este nome é utilizado incorretamente.

## 5.1 Blocos Básicos e Grafos de Fluxo de Execução

Um bloco básico é uma seqüência de instruções em assembler que estão em seqüência no código gerado pelo compilador de tal forma que o fluxo de controle se inicia na primeira instrução do bloco básico e termina na última. Nenhuma instrução do bloco básico, exceto a última, pode ser desvio condicional ou incondicional (`goto`, `goto<`, ...). Nenhuma instrução, exceto a primeira, é o alvo de um desvio condicional ou incondicional.

Aho, Sethi e Ullman [4] fornecem um algoritmo para encontrar os blocos básicos de uma seqüência de instruções em assembler, descrito a seguir.

1. Primeiro determinamos os *líderes* do código em assembler. Um líder é a primeira instrução de um bloco básico, e é encontrado pelas regras apresentadas abaixo.

   - A primeira instrução é um líder. Isto é, a instrução onde o código inicia a sua execução é um líder.
   - As instruções que são alvos de desvios são líderes. Estas instruções possuem obrigatoriamente *label* no assembler utilizado.
   - Qualquer instrução que se segue a um desvio é um líder.

2. O bloco básico correspondente a cada líder se inicia nele e termina imediatamente antes do fim do próximo líder ou no fim do programa.

Como exemplo, considere que o programa em S2

```
  var i, j : integer;
begin
i = 0;
while i < 10 do
  begin
  if i > 5
  then
    j = 10;
    while j > 0 do
      j = j - 1;
  else
    j = 5;
  endif
  i = i + 1;
  end
end
```

seja traduzido para

```
( 1)     mov i, 0
( 2)     goto L1
( 3) L2: cmp i, 5
( 4)     goto<= L3
( 5)     mov j, 10
( 6)     goto L4
( 7) L5: sub j, 1
( 8) L4: cmp j, 0
( 9)     goto> L5
(10)     goto L6
(11) L3: mov j, 5
(12) L6: add i, 1
(13) L1: cmp i, 10
(14)     goto< L2
(15)     exit
```

Os blocos básicos deste código são

```
1  2
3  4
5  6
7
8  9
10
11
12
13  14
15
```

Blocos básicos possuem duas características importantes:

Figure 5.1: Grafo do Fluxo de Execução de um programa S2

1. uma vez que o fluxo de execução começa na primeira instrução, ele prossegue até a última sem interrupção por desvios e;

2. não há desvios para o meio de um bloco básico. A execução sempre se inicia na primeira instrução.

### Grafos de Fluxo de Execução

Um grafo G = (V, E) é composto por um conjunto de vértices V e arestas E. Uma aresta é um par (v, w) onde v e w são vértices do grafo. Dizemos que v e w estão ligados por uma aresta. Em um *grafo dirigido*, a aresta (w, v) é diferente de (v, w), sendo esta última uma aresta *de* v *para* w.

Um caminho em um grafo é uma seqüência de arestas ($v_1$, $v_2$), ($v_2$, $v_3$), ($v_3$, $v_4$), ... ($v_{n-1}$, $v_n$) que ligam $v_1$ a $v_n$. Um ciclo é um caminho onde $v_1$ a $v_n$.[1]

O fluxo de execução de um programa pode ser visualizado criando-se um grafo dirigido a partir dos seus blocos básicos. Cada vértice do grafo é um bloco básico. Existirá uma aresta de um bloco B1 para um bloco B2 se B2 puder ser executado imediatamente após B1, o que acontecerá se:

1. houver um desvio (`goto`) condicional ou incondicional da última instrução de B1 para a primeira de B2;

2. B2 seguir-se a B1 no programa e B1 não terminar com um `goto` incondicional. Neste caso, existe em algum ponto do programa um desvio para a primeira instrução de B2: foi esta a razão pela qual B2 foi separado de B1.

Um grafo construido de acordo com as regras acima será chamado de *Grafo de Fluxo de Execução*. Para o programa apresentado na seção 5.1, o grafo de fluxo de execução é aquele mostrado na Figura 5.1.

Nas seções seguintes, descrevemos as otimizações mais comuns sem entrar em detalhes sobre os algoritmos necessários para realizá-las.

## 5.2 Otimizações em Pequena Escala

Otimizações em pequena escala (*peephole optimizations*) possuem este nome porque são feitas examinando-se poucas instruções do código gerado, em geral do assembler.

Para compreender as otimizações *peephole* (e otimizações em geral), devemos ter em mente que:

1. Um desvio incondicional, como `goto>`, `goto<=`, etc, nem sempre causa um desvio. Quando não causa, o seu tempo de execução é menor do que o de um `goto`. Quando causa, o `goto` é mais rápido.

[1] Um grafo dirigido sem ciclos é chamado em Inglês de *direct acyclic graph*, abreviado por DAG.

2. Uma multiplicação ou divisão é muito mais lenta do que uma soma, subtração ou deslocamento de bits.

3. Operadores de manipulação de bits como & e | binários de C++ são muito mais rápidos do que as operações aritméticas.

4. O uso de constantes é mais rápido do que o de variáveis:
```
mov a, 2
```
é mais rápido do que
```
mov a, b
```

Colocaremos o código original seguido do código otimizado dentro de um retângulo *ou* o código original seguido de uma barra horizontal seguido do código otimizado. As otimizações triviais não serão comentadas.

1.
```
mov t, a
mov a, t
```

---

```
mov t, a
```

Este código pode ter sido o resultado da tradução de mais de uma linha do código fonte original, como
```
t = a;
a = t + 1;
```

2.
```
push a
pop a
```

---

```
(nada)
```

3.
```
    cmp 5, 3
    goto<> L2
L1: mov a, 1
L2:
```

---

```
    goto L2
L1: mov a, 1
L2:
```

A comparação será sempre verdadeira e não precisa ser feita.

Embora seja improvável que o programador gere este código, ele pode ser o resultado do uso de constantes no programa, como neste exemplo:

```
#define Max 5
...
if ( Max == 3 )
   a = 1;
```

Admitimos que o registrador `cm` é zerado pela instrução `goto<>`. Se não fosse, o código otimizado produziria um resultado diferente do original por causa deste registrador. Na versão original, o registrador seria inicializado e, não otimizada, não.

4. Eliminação de salto sobre salto.

```
     cmp a, 1
     goto== L1
     goto L2
L1:  add x, y
L2:
```

---

```
     cmp a, 1
     goto<> L2
L1:  add x, y
L2:
```

5.
```
     goto L1
     ...
L1:  goto L2
     ...
L2:  ...
```

---

```
     goto L2
     ...
L1:  goto L2
     ...
L2:
```

6.
```
     goto L1
     ...
     goto L4
```

```
L1:  cmp a, b
     goto> L2
L3:  ...
```

```
L1:  cmp a, b
     goto> L2
     goto  L3
     ...
     goto L4
     L3:
```

As duas versões possuem o mesmo número de comandos. Contudo, a última versão é mais rápida porque o `goto L3` só será executado quando a comparação falhar, enquanto que na versão não otimizada, `goto L1` sempre será executado.

7. Simplificações Algébricas.

   Estas otimizações serão apresentadas em C++ por uma questão de clareza.

   ```
   a = b + 0
   a = b * 1
   a = b * 0
   a = b / 1
   a = 0 - b
   a = b - 0
   ```

   ```
   a = b
   a = b
   a = 0
   a = b
   a = -b
   a = b
   ```

   Instruções como as acima geralmente não são produto da codificação direta do programador, mas o resultado da substituição de constantes:

   ```
   #define Max 1
   ...
   size =  n*Max;
   ```

8. Transformações Utilizando Operações Dependentes de Máquinas.

   ```
   8*a
   ```

```
a << 3
```

```
a/16
```

---

```
a >> 4
```

```
a%2
```

---

```
a&1
```

```
160*a
```

---

```
(a << 7) + (a << 5)
```

Esta otimização é obtida por:

```
160*a = 32*5*a = 32*(4 + 1)*a = 128*a + 32*a = (a << 7) + (a << 5)
```

Muitas máquinas possuem uma instrução `inc x` que incrementa `x` de `1`. Esta operação é muito mais rápida, geralmente, do que somar `1` a `x` com "`add x, 1`".

Assim, soma pode ser otimizada:

```
add  x, 1
```

---

```
 inc x
```

```
add x, 2
```

---

```
inc x
inc x
```

Em geral é mais rápido chamar `inc` duas vezes do que utilizar `add x, 2`.

## 5.3 Otimizações Básicas

1. Subexpressões Comuns.

   Uma expressão será chamada de "subexpressão comum" se ela aparecer em dois lugares diferentes e se as suas variáveis não tiverem mudado de valor entre o cálculo de um expressão e outra. Pode-se calcular o valor da subexpressão apenas uma vez. Como exemplo, o codigo:

```
a = 4*i;
b = c;
e = 4*i;
```

   pode ser otimizado para

```
a = 4*i;
b = c;
e = a;
```

   A subexpressão comum é `4*i`, sendo que o valor de `i` não é modificado entre as duas avaliações. Freqüentemente, o valor da subexpressão é colocado em uma variável temporária `t`:

```
t = 4*i
a = t;
b = c;
e = t;
```

   Esta otimização pode ser local a um bloco básico ou global, envolvendo vários blocos. Neste último caso, será necessário empregar algoritimos de análise de fluxo de execução para descobrir quais são as subexpressões comuns do programa. Veja o exemplo abaixo:

```
a = 4*i;
if ( i > 10 ) {
  i++;
  b = 4*i;
  }
else
  c = 4*i;
```

   Este código é transformado em

```
a = 4*i;
if ( i > 10 ) {
  i++;
  b = 4*i;
  }
else
  c = a;
```

2. Propagação de Cópia (*Copy Propagation*) .

   Sempre que houver uma atribuição

   ```
   b = c;
   ```

   a variável `c` poderá substituir `b` após esta instrução, desde que nenhuma das duas variáveis mude de valor.

   ```
   a = b;
   c = a + x;
   ```

   ---

   ```
   a = b;
   c = b + x;
   ```

   Com esta otimização, esta atribuição poderá tornar-se desnecessária e ser eliminada.

3. Eliminação de Código Morto (*Dead Code Elimination*)

   Código Morto é o código que nunca será executado, independente do fluxo de execução do programa.

   ```
   int f (int n )
   {
     int i = 0;

     while ( i < n ) {
       if ( g == h ) {
         break;
         g = 1;  // morto
         }
       i++;
       g--;
       }
     return g;
     g++;      // morto
     }
   ```

   Esta função pode ser otimizada para

   ```
   int f (int n )
   {
     int i = 0;

     while ( i < n ) {
       if ( g == h )
         break;
       i++;
       g--;
       }
   ```

```
  return g;
  }
```

Código morto pode ser identificado fazendo-se uma busca no grafo de fluxo de execução da função, começando-se na primeira instrução. Os blocos básicos não alcançados por esta busca nunca poderão ser executados. Como exemplo, traduziremos o programa acima para assembler:

```
     mov i, 0
     goto L1
L2:  cmp g, h
     goto<> L3
     goto L4
     mov g, 1
L3:  add i, 1
     sub g, 1
L1:  cmp i, n
     goto< L2
L4:  ret
     add g, 1
```

O grafo de fluxo de execução está na Figura 5.2. Observe que os blocos básicos `B4` e `B8` nunca podem ser executados se a execução começar em `B1`. Neste caso específico, nenhuma flecha chega a `B4` ou `B8`. Mas eles poderiam ser código morto mesmo se houvesse referências a eles. Examine o trecho de código a seguir:

```
return 1;
L1:  goto L2
     g = 1;
L2:  g++;
     goto L1;
```

4. Avaliação de Expressões Constantes.

```
const
  Max = 100*20,
  Tam = Max + 1;

a = 1 + a + 3;
```

---

```
const
  Max = 2000,
  Tam = 2001;

a = a + 4;
```

Figure 5.2: Grafo do Fluxo de Execução com código morto

Esta avaliação pode ser combinada com eliminação de código morto:

```
#define debug 0
...
if ( debug )
  g = 1;
else
  g = -1;
```

---

```
#define debug 0
...
g = -1;
```

5. Fatoração de Código

   Esta é uma otimização que troca velocidade por espaço, poupando este último. Duas seqüencias de instruções idênticas no código fonte causam a geração de apenas uma seqüencia de instruções no executável.

```
gets(s);
while ( *s != '\n' ) {
  cout << strupr(s) << endl;
  gets(s)  ;
  }
```

---

```
     goto L1;
L2:  cout << strupr(s) << endl;
L1:  gets(s);
     if ( *s != '\n' ) goto L2;
```

Em S2:

```
i = i + 1;
while i < 10 do
  begin
  j = j + 1;
  i = i + 1;
  end
```

---

```
     goto L1
```

```
L2:  add j, 1
L1:  add i, 1
     cmp i, 10
     goto < L2
```

Esta otimização é comum em `switch`'s:

```
switch (n) {
  case 1:
    f();
    g();
    puts(s);
    i++;
    break;
  case 2:
    write(fp);
    break;
  case 3:
    g();
    puts(s);
    i++;
  }
```

---

```
switch (n) {
  case 1:
    f();
  case 3:
    g();
    puts(s);
    i++;
    break;
  case 2:
    write(fp);
  }
```

6. Otimizações de `if`'s e `switch`'s

   Uma seqüencia de `if`'s aninhados como

```
if ( n == 1 )
  S1;
else if ( n == 2 )
```

```
  S2;
else if ( n == 3 )
  S3;
else
  S4;
```

onde n é um inteiro, pode ser otimizada para um comando `switch`:

```
switch (n) {
  case 1:
    S1;
    break;
  case 2:
    S2;
    break;
  case 3:
    S3;
    break;
  default:
    S4;
  }
```

A tradução do `switch` para assembler descrita anteriormente pode ser otimizada através do comando `goto v[i]` que permite desviar para um dos endereços armazenados em um vetor. O comando `switch` acima pode ser traduzido para:

```
     cmp n, 1
     goto<  L1
     cmp n, 3
     goto> L1
     goto ender[n]
L2:  codigo para S1
     goto fim
L3:  codigo para S2
     goto fim
L4:  codigo para S3
     goto fim
L1:  codigo para S4
fim:
```

Admitimos que o vetor `ender` tenha sido inicializado com os endereços L2, L3 e L4.

Nos casos em que existirem muitas opções do `switch` e os números presentes no `case` não estiverem em ordem, pode-se utilizar uma tabela *hash* para obter o endereço fornecendo-se o valor de `n` (ou a expressão do `switch`) como chave:

```
calcula a funcao hash usando n
goto R0
...
```

Na tradução acima, admitimos que em `R0` foi colocado o resultado do cálculo da função *hash*.

Existem algoritmos que geram uma função *hash* dado um conjunto de números ou *strings* como entrada. A tabela *hash* gerada a partir dos números poderá ser:

- perfeita, quando a função *hash* não colocar mais de um elemento em cada posição da tabela. Isto é, se `v` for a tabela, `v[i]` não apontará para um lista com mais de um nó;
- mínima, quando cada `v[i]` apontar para uma lista com pelo menos um nó.

Em uma tabela *hash* perfeita e mínima cada posição da tabela aponta para uma lista de exatamente um elemento. Neste caso, uma tabela *hash* para `k` elementos será implementada como um vetor de tamanho `k`.

7. Eliminação de Variáveis Inúteis

   Uma variável local a um procedimento será inútil se ela for usada apenas do lado esquerdo de atribuições ou não for utilizada de modo algum. Na função:

```
int fat( int n )
{
  int i, j, p, k;

  i = p = 1;
  for ( j = 2; j <= n; j++ )
    p *= j;
  return p;
  }
```

   as variáveis `i` e `k` são inúteis e podem ser eliminadas do código, resultando em

```
int fat( int n )
{
  int  j, p;

  p = 1;
  for ( j = 2; j <= n; j ++ )
    p *= j;
  return p;
  }
```

## 5.4 Otimizações de Laços

1. Movimentação de Código (*Code Motion*)

Expressões que são constante dentro de laços podem ser avaliadas antes do laço e o resultado reaproveitado. Exemplos:

```
i = 0;
while i < n - 1 do
  begin
  write(i);
  i = i + 1;
  end
```

---

```
i = 0;
t1 = n - 1;
while i < t1 do
  begin
  write(i);
  i = i + 1;
  end
```

O código

```
s = 0;
for ( i = 0; i < n; i++ )
  s += a*b/i;
```

pode ser otimizado para

```
s = 0;
t1 = a*b;
for ( i = 0; i < n; i++ )
  s += t1/i;
```

2. Redução em Poder (*Strengh Reduction*)

   Redução em poder refere-se a transformar operações lentas (como multiplicações) em operações rápidas (como somas) dentro de laços. A cada iteração é aproveitado o resultado obtido pela iteração anterior, eliminando a necessidade de operações mais complexas. A multiplicação `4*i` dentro do laço

   ```
   for ( i = 0; i < n; i++ )
     f( 4*i );
   ```

   pode ser transformada em soma:

```
t = 0;
for ( i = 0; i < n; i++ ) {
  f(t);
  t += 4;
  }
```

A variável `t` é chamada de variável de indução.

Existe uma operação de multiplicação implícita quando vetores são indexados. Redução em poder pode também ser utilizado neste caso.

```
floot v[ Max ];
for ( i = 0; i < Max; i++ )
  v[i] = 0;
```

---

```
floot v[ Max ], *p;

p = v;
for ( i = 0; i < Max; i++ )
  *p++ = 0;
```

Contudo, em algumas máquinas a primeira forma será mais rápida.

3. Supressão de Testes de Limites Redundantes

   Alguns compiladores inserem testes que conferem se a variável (ou expressão) que indexa um vetor está dentro dos limites permitidos. Se o vetor `v` for declarado como:

   ```
   var v : array(integer)[100];
   ```

   a atribuição

   ```
   v[i] = a;
   ```

   será traduzida para

   ```
   cmp i, 0
   goto< Erro
   cmp i , 100
   goto>= Erro
   mov v[i], a
   ...
   Erro:
   ```

   Na posição do *label* `Erro` estará um código que imprime uma mensagem de erro e termina o programa. Esta mensagem poderia informar o número da linha do código fonte onde aconteceu o erro, o valor de `i` e o nome do vetor.

   Uma alternativa a imprimir uma mensagem com erro é sinalizar uma exceção, se a linguagem suportar esta construção. Deste modo, a exceção poderia ser tratada pelo programa evitando o seu término forçado.

Observe que as instruções que se seguem ao *label* `Erro` que testam se `i >= 0` e `i < 100` fazem parte do sistema de tempo de execução, pois pertencem ao código gerado mas não foram inseridas diretamente pelo programador.

Se um vetor for indexado pela variável de repetição de um laço, os testes de limites poderão ser feitos uma única vez antes da execução do laço. Por exemplo,

```
  var  v : array(integer)[30];
  ...
begin
i = 0;
while i <= 12 do
  begin
  v[i] = 1;
  i = i + 1;
  end
...
end
```

Normalmente traduzido para

```
     mov i, 0
     goto L1
L2:  cmp i, 0
     goto< Erro
     cmp i, 30
     goto>= Erro
     mov v[i], 1
     add i, 1
L1:  cmp i, 12
     goto<= L2
     ...
     Erro:
```

pode ser otimizada para

```
     mov i, 0
     cmp 0, 0
     goto< Erro
     cmp 12, 30
     goto>= Erro
     goto L1
L2:  mov v[i], 1
     add i, 1
L1:  cmp i, 12
     goto<= L2
```

Como as comparações utilizam constantes, uma otimização a mais resulta em

```
     mov i, 0
     goto L1
```

```
L2:  mov v[i], 1
     add i, 1
L1:  cmp i, 12
     goto<= L2
```

Este último passo não seria possível ser `i` fosse inicializado com uma variável e o limite superior fosse também uma variável:

```
i = k1;
while i <= k2 do
  begin
  v[i] = 1;
  i = i + 1;
  end
```

4. Desdobramento de Laço (*Loop Unrolling*)

   Quando o número de repetições de um laço for conhecido em tempo de compilação, pode-se eliminar o laço e gerar o código de dentro da repetição o número de vezes em que o laço seria executado.

```
i = 0;
while i < 4 do
  begin
  s = s + v[i];
  i = i + 1;
  end
```

---

```
i = 0;
s = s + v[i];
i = i + 1;
s = s + v[i];
i = i + 1;
s = s + v[i];
i = i + 1;
s = s + v[i];
i = i + 1;
```

Um bom compilador poderia ainda otimizar este código para

```
i = 4;
s = s + v[0];
s = s + v[1];
s = s + v[2];
s = s + v[3];
```

e mesmo para

```
i = 4;
s = s + v[0] + v[1] + v[2] + v[3];
```

Se o número de vezes que o laço será executado não for conhecido em tempo de compilação, o corpo do laço pode ser duplicado, permitindo outras otimizações como eliminação de subexpressões comuns. Pittman e Peters [7] citam como exemplo o código

```
while ( a < b ) {
  b = b - a*k;
  a++;
  }
```

transformado em

```
while ( 1 ) {
  if ( a >= b )
    break;
  b = b - a*k;
  a++;
  if ( a >= b )
    break;
  b = b - a*k;
  a++;
  }
```

O valor do primeiro cálculo de `a*k` pode ser colocado em uma variável temporária `t1` e reutilizado na segunda atribuição

```
b = b - a*k;
```

que pode ser modificada para

```
b = b - t1 + k;
```

já que

```
a1*k = (a0 + 1)*k = a0*k + k = t1 + k
```

onde `a0` e `a1` correspondem aos dois valores da variável `a` neste laço.

Se o número de repetições for conhecido e par, pode-se duplicar o corpo do laço reduzindo-se pela metade o número de testes de fim de laço:

```
for ( i = 0; i < 100; i++ )
  s[i] = 0;
```

---

```
     i = 0;
     goto L1;
L2:  s[i] = 0;
     i++;
```

```
     s[i] = 0;
     i++;
L1:  if ( i < 100 ) goto L2
```

## 5.5 Otimizações com Variáveis

1. Colocação de Variáveis em Registradores

   A manipulação de registradores é muito mais rápida do que a manipulação de memória RAM. Por esta razão, é importante colocar as variáveis mais usadas de cada procedimento[2] em registradores. Em geral, as variáveis mais usadas são aquelas empregadas nos laços mais internos. No exemplo

   ```
   for ( i = 0; i < n; i++ )
     for ( j = 0; j < n; j++ )
       for ( k = 0; k < n; k++ )
         s[i][j][k] = 0;
   ```

   as variáveis mais utilizadas são, na ordem,

   ```
   k > j > i == s
   ```

   Assim, se houver apenas dois registradores disponíveis, eles serão alocados para `k` e `j`. O compilador pode otimizar este código por "Movimentação de Expressões Constantes". O endereço de `s[i][j]` é uma constante para o último laço, o do `k`:

   ```
   int *p;
   for ( i = 0; i < n; i++ )
     for ( j = 0; j < n; j++ ) {
       p = &s[i][j];
       for ( k = 0; k < n; k++ )
         *p++ = 0;
       }
   ```

   Neste caso, melhor seria colocar `p` e `k` em registradores.

   Registradores não possuem endereço e, portanto, uma variável que tem o seu endereço tomado (com & em C++) não pode ser colocada em um registrador.

   Suponha que existam dois procedimentos `P` e `Q` sendo que ambos utilizam os registradores `R0` e `R1` como variáveis locais. Após `P` chamar `Q`, os valores dos registradores `R0` e `R1` utilizados em `P` terão sido alterados por `Q`. Então, antes de chamar `Q`, o procedimento `P` deve empilhar estes registradores:

   ```
   push R0
   push R1
   call Q
   pop R1
   pop R0
   ```

[2] Chamaremos procedimentos qualquer subrotina, rotina ou função.

Ou `Q` deve salvar estes registradores antes de usá-los.

2. Reuso de Registradores e Variáveis Locais/Temporárias

   Um registrador/variável está *vivo* do ponto em que recebe um valor ao ponto onde é utilizado pela última vez. Se duas variáveis locais a uma subrotina nunca estão vivas ao mesmo tempo, elas podem ocupar a mesma posição de memória ou registrador. Assim, no código

```
void f()
{
  int i, j;

  for ( i = 0; i < 10; i++ )
    cout << i << endl;
  for ( j = 10; j > 0; j-- )
    cout << j << endl;

  }
```

   `i` e `j` podem ser a mesma variável:

```
void f()
{
  int i;

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

   Esta técnica também é utilizada para diminuir o número de variáveis temporárias necessárias a uma subrotina.

## 5.6 Otimizações de Procedimentos

1. Passagem de Parâmetros/Valor de Retorno por Registradores

   O compilador pode adotar a convenção de passar os primeiros parâmetros de uma chamada de procedimento em determinados registradores utilizados apenas para esta finalidade. Sem esta otimização, os parâmetros são passados pela pilha, que é muito mais lenta. O mesmo raciocínio se aplica aos valores de retorno de funções.

2. Expansão em Linha de Procedimentos

   Procedimentos pequenos podem ser expandidos nos locais onde são chamados, eliminando a sobrecarga de uma chamada de subrotina. Por exemplo,

```
inline getValor()
{
  return valor;
```

```
  }
...
i = getValor() + 1;
```

é otimizado para

```
...
i = valor + 1;
```

Em C++, funções que devem ser substituídas em linha podem ser declaradas com a palavra chave `inline`, como mostrado acima.

Pode acontecer de haver chamadas recursivas entre as rotinas "inline". Neste caso, uma das rotinas não pode ser expandida.

Alguns compiladores expandem em linha funções de bibliotecas como `memset`, `strlen`, `strcpy`, etc. Mesmo que o usuário não declare alguma função como `inline` em C++, ela poderá ser expandida automaticamente pelo compilador.

Esta otimização é muito importante. Tipicamente, 40% do tempo de execução de um programa é gasto em chamadas de subrotinas. Cada chamada envolve: a) passagem de parâmetros, b) empilhamento do endereço de retorno, c) salto para a função, d) salvamento e inicialização de um registrador que permitirá manipular as variáveis locais e) alocação das variáveis locais f) destruição das variáveis locais g) salto para o endereço de retorno.

Entre e) e f) acontece a execução do corpo da função. Quando uma função for colocada em linha, os passos b)-g) e talvez a) serão eliminados. Esta otimização é particularmente importante porque parte considerável das funções é chamada apenas uma única vez em todo o programa. Assim, o uso intensivo desta otimização pode até tornar o programa menor.

3. Recursão de Cauda (*Tail Recursion*)

   Quando existir uma chamada de procedimento recursivo ao fim da execução do procedimento, esta poderá ser substituída por um desvio incondicional para o início do procedimento.

```
int E2 ()
{
  if ( lex->token == mais_smb ) {
    lex->nextToken();
    cout << "+";
    T();
    E2();
    }
  else
    if ( lex->token == menos_smb ) {
      lex->nextToken();
      cout << "-";
      T();
      E();
      }
  }
```

```
int E2()
{
  L:
  if ( lex->token == mais_smb ) {
    lex->nextToken();
    cout << "+";
    T();
    goto L;
    }
  else
    if ( lex->token == menos_smb ) {
      lex->nextToken();
      cout << "-";
      T();
      goto L;
      }
  }
```

Este exemplo é uma adaptação de um exemplo de Aho et al. [4]. Recursão de cauda poderá ser otimizada mesmo se o procedimento possuir parâmetros:

```
void P( int a )
{
  if ( a > 2 )
    P( a - 1 );
  else  if ( a == 2 )
    cout << "0" << endl;
  else
    P(10);
  }
```

```
void P( int a )
{
  L:
  if ( a > 2 ) {
    a = a - 1;
    goto L;
    }
  else
    if ( a == 2 )
      cout << "0" << end;
```

```
    else {
      a = 10;
      goto L;
      }
  }
```

Observe que

```
void P( int a )
{
  if ( a > 2 )
    P( a - 1 );
  else
    cout << "0" << endl;
  cout << "P" << endl;
  }
```

não pode ser otimizado porque há uma instrução após "`P( a - 1 )`".

4. Transformação de Variáveis Locais em Estáticas

   Quando um procedimento é chamado, deve-se alocar memória na pilha para as suas variáveis locais, que são manipuladas por meio do registrador `bp`. O código do início de um procedimento `P`, com duas variáveis locais, deve ser

   ```
   push bp
   mov bp, t
   add t, 2
   ```

   sendo que `t` é o registrador contendo o topo da pilha da máquina. A primeira variável local será manipulada por `bp[1]` e a segunda, por `bp[2]`. O valor de `bp` é salvo porque ele é utilizado também pelo procedimento que chamou `P`. Este valor é restaurado ao final da execução de `P`, juntamente com a desalocação das variáveis locais:

   ```
   sub t, 2
   pop bp
   ret
   ```

   A alocação e desalocação de variáveis locais na pilha é lenta e pode ser substituída, em alguns casos, por alocação estática, feita uma única vez antes do início da execução do programa.

   Se o procedimento não for direta ou indiretamente recursivo, haverá, no máximo, um conjunto de suas variáveis locais na pilha do computador. Sendo assim, as variáveis podem ser alocadas estaticamente. Se o procedimento for recursivo, não saberemos, em tempo de compilação, quantas vezes ele chamará direta ou indiretamente a si mesmo e, portanto, não sabemos quantos conjuntos de suas variáveis locais serão necessários em tempo de execução.

   Pode-se descobrir quais são os procedimentos recursivos de um programa através de uma busca em um grafo dirigido onde os vértices são os procedimentos e existe uma aresta de `v` para `w` se o procedimento `v` pode chamar `w` em tempo de execução. Em outras palavras, haverá aresta (`v`, `w`) se houver uma chamada

   ```
   w();
   ```

   em algum lugar do procedimento `v`.

Figure 5.3: Grafo de chamadas de procedimentos

Um exemplo de um grafo construído segundo esta regra é mostrado na Figura 5.3. Existe uma chamada de `m` a si mesmo e, portanto, `m` é recursivo. O procedimento `g` chama `p` que chama `q` que chama `g`, existindo então uma recursão `g-p-q-g`.

Nas frases acima, utilizamos "`x` chama `y`" para significar "no código fonte de `x` existe uma chamada a `y`". Talvez esta chamada nunca ocorra em tempo de execução, quaisquer que sejam os dados fornecidos ao programa. Como é impossível afirmar com certeza se `x` irá mesmo chamar `y`, admitiremos que isto *pode* acontecer.

Sempre que houver um círculo no grafo de chamadas, poderá haver recursão em tempo de execução e assim os procedimentos envolvidos em recursão devem ter alocação dinâmica de variáveis locais, na pilha. Os outros procedimentos podem utilizar alocação estática.

Observe que os procedimentos `f` e `r` nunca estarão na pilha ao mesmo tempo. Portanto, podemos alocar uma única área de memória para as variáveis de `f` e `r`. Este raciocínio pode ser estendido para todo o grafo.

Para explicar este ponto, considere um grafo representando um programa em C. A "raiz" do grafo é a função `main` e as folhas são procedimentos que não chamam ninguém. Se todas as funções forem recursivas, todas as variáveis locais devem ser alocadas dinamicamente na pilha.

Caso contrário, existe pelo menos uma função não recursiva. Considere que $\mathtt{p}_1$, $\mathtt{p}_2$, ..., $\mathtt{p}_n$ sejam todos os caminhos que começam em `main` e terminam em a) uma folha ou b) uma função recursiva. Sendo $\mathtt{esp(q}_j\mathtt{)}$ o número de bytes necessários para as variáveis locais de $\mathtt{q}_j$, o número $\mathtt{esp(p}_i\mathtt{)}$ de bytes necessários para as variáveis locais de todas as funções no caminho $\mathtt{p}_i = \mathtt{q}_1\mathtt{q}_2\mathtt{...q}_k$ é dado por

$$\mathtt{esp(p}_i\mathtt{)} = \sum_{j=1}^{k} \mathtt{esp(q}_j\mathtt{)}$$

É necessário alocar estaticamente para todo o programa um número de bytes igual a

$$\mathtt{max(esp(p}i\mathtt{))},\ 1 \leq i \leq n$$

Como exemplo, o grafo de chamadas da Figura 5.4 mostra ao lado de cada procedimento quantos bytes são necessários para as variáveis locais. Para este programa, o caminho que necessitará de mais memória será

$$\mathtt{main} \Longrightarrow \mathtt{g} \Longrightarrow \mathtt{n}$$

em um total de 38 bytes.

Figure 5.4: Grafo de chamadas de procedimentos

## 5.7 Dificuldades com Otimização de Código

Algoritmos de otimização de código são complexos e as transformações que eles fazem podem não corresponder precisamente à definição semântica da linguagem utilizada. Isto é, uma otimização pode transformar um código em outro mais eficiente mas que não é equivalente ao primeiro.

Veremos a seguir algumas transformações incorretas que um otimizador de código pode fazer.

i) Ao avaliar uma expressão constante, como em

```
x = 3.5/7*9 + 3/2;
```

o compilador ou otimizador deve empregar as mesmas regras de avaliação que as definidas pela linguagem. A maneira mais segura de avaliar expressões é gerar um pequeno programa que avalie esta expressão. Assim, se a linguagem utilizada for C, pode-se gerar o programa

```
#include <stdio.h>

void main()
{
  printf("%f\n", 3.5/7*9 + 3/2 );
  }
```

executá-lo, tomar o resultado e substituir a atribuição à variável x pela atribuição ao resultado.

ii) Uma multiplicação `2*n` pode ser transformada em `n << 1`. Contudo, haverá um erro se a instrução "rolamento à esquerda" ( que é `<<` ) da máquina alvo colocar o último bit do número na primeira posicão. Por exemplo, se `n` for

```
10011010
```

o resultado de `n << 1` poderá ser

```
00110101
```

iii) A movimentação de expressões constantes para fora de laços pode retirar testes de proteção para a expressão contra divisão por zero, estouro de limites, etc. Como exemplo,

```
s = 0;
```

```
for ( i = 0; i < n; i++ )
  if ( b != 0 )
    s += v[i] + a/b;
```

poderia ser incorretamente otimizado para

```
s = 0;
t1 = a/b;
for ( i = 0; i < n; i++ )
  if ( b != 0 )
    s += v[i] + t1;
```

iv) Aliás (*alias*) é o uso de dois nomes para uma mesma posição de memória. Como exemplo, no código

```
void  m( int &a, int &b )
{

  a = 2;
  b = 5;
  b = a*2;
  }
```

haverá um aliás se `m` for chamado por

```
m( x, x );
```

Os parâmetros `a` e `b` irão referenciar `x`. Por este motivo, a função `m` não pode ser otimizada para

```
void m ( int &a, int &b )
{
  a = 2;
  b = 4;
  }
```

Aliás pode acontecer em C++ na presença de variáveis passadas por referência e ponteiros. Então, sempre que houver variáveis deste tipo temos que assumir que elas podem ser modificadas por qualquer atribuição ou chamada de função, o que impede muitas otimizações: variáveis que eram constantes em determinado trecho (como `a` no exemplo acima) não podem mais ser consideradas como tal.

A linguagem C++ permite conversão de ponteiros de um tipo para outro desde que se utilize um *cast*:

```
  char *s;
  float f;
  int *p;
...
p = (int *) s;
...
p = &f;
```

Isto significa que um ponteiro pode referenciar variáveis e áreas de memória de qualquer tipo. Deste modo, uma atribuição

```
*p = 0;
```

pode, potencialmente, alterar qualquer variável, vetor ou objeto, dificultando a realização de otimizações como eliminação de subexpressões comuns, propagação de variáveis e movimentação de expressões constantes para fora de laços.

No exemplo

```
void m( int *p, int i )
{
  int a = i, b, c;

  if ( i > 0 ) p = &a;
  c = 3*a;
  *p = 5;
  b = 3*a;
  ...
  }
```

A subexpressão `3*a` não pode ser calculada uma única vez porque o valor de `a` pode ser alterado entre as atribuições "`c = 3*a`" e "`b = 3*a`". Se o endereço de uma variável for tomado, como em "`p = &a`", deve-se admitir que o seu conteúdo pode ser modificado indiretamente. Observe que o uso de ponteiros não impede sempre o compilador de fazer otimizações, mas torna a análise do que é constante em um certo trecho de código muito difícil.

Se a função `m` fosse definida como

```
void m( int *p, int i )
{
  int a = i, b, c;

  c = 3*a;
  b = 3*a;
  if ( i > 0 ) p = &a;
  *p = 5;
  ...
  }
```

a expressão `3*a` poderia ser calculada uma única vez porque as atribuições a `b` e `c` precedem, no grafo de fluxo de execução desta função, à tomada de endereço de `a`.

Se houver mais de um vetor como parâmetro, como em

```
void q( int v[], int w[], int n )
{
  int i = 0;

  v[0] = 3*w[0];
  a = 3*w[0];
  ...
  }
```

o compilador deve considerar que a escrita em uma posição de `v` pode alterar outra posição de `w`, pois os dois vetores podem se referir à mesma área de memória ou áreas sobrepostas. Assim, a função acima não pode ser otimizada para

```
void q( int v[], int w[], int n )
{
  int i = 0, t1;

  v[0] = t1 = 3*w[0];
  a = t1;
  ...
  }
```

onde `t1` é uma variável temporária, porque esta função poderia ser chamada como

```
int s[100];
...
q( s, s, 100 );
```

e, neste caso, `v[0]` seria igual a `w[0]`.

Em muitos compiladores, uma opção de compilação "Assume no alias" pode ser ligada quando o programador tiver certeza de que não haverá nenhum aliás em chamadas de função. Neste caso, a função `q` poderia ser otimizada porque o programador estaria afirmando que chamadas como "`q( s, s, 100)`" nunca ocorrerão. Claramente, esta opção é muito perigosa e deve ser ligada em muito poucos casos.

# Appendix A

# A Linguagem S2

A linguagem S2 é um pequeno subconjunto de Pascal com algumas modificações. O nome S2 provém de SS, Super Simples. A linguagem Simples é um superconjunto de S2 com suporte a orientação a objetos. Ela será estudada na parte 2 desta apostila. Simples é também um subconjunto de Green [8] [**?**].

Um exemplo de um pequeno programa em S2 que imprime os `n` primeiros números inteiros, `n` lido do teclado, é mostrado na Figura A.1. Este programa possui os principais elementos da linguagem. O programa começa com a declaração das variáveis globais, parte que é opcional. Em seguida, há o corpo do programa entre `begin` e `end`, com zero ou mais instruções. Os comandos são semelhantes aos de Pascal, exceto que o `if` é terminado por `endif` e não há "." após o `end` que termina o programa. O ";" termina as atribuições e comandos `read` e `write`. Como qualquer outra linguagem de programação, os terminais do programa são separados por branco, fim de linha ou caráter de tabulação.

A seguir detalhamos o significado de cada um dos elementos de S2.

## A.1 Comentários

Comentários na linguagem são colocados entre "{" e "}". Comentários aninhados não são permitidos, como

```
{  comentario { outro comentario }  fim primeiro }
```

A linguagem S2 também admite comentários do tipo `//` iguais aos de C++. Qualquer coisa após `//` até o fim da linha é ignorado pelo compilador. Os símbolos { e } dentro de um comentário iniciado por `//` não significam comentário. O mesmo se aplica a `//` entre { e }.

## A.2 Tipos e Literais Básicos

Existem apenas dois tipos em S2, `integer` e `boolean`. Literais do tipo `integer` devem estar entre os limites `0` e `32767`, sendo que qualquer número de zeros antes de um número é permitido. Assim, os números

```
00000000000001
00000000000000
```

são válidos. O tipo `boolean` possui apenas dois valores: `false` e `true`.

Os operadores `<`, `<=`, `>`, `>=`, `==`, `<>` de comparação podem ser aplicados a valores inteiros ou booleanos sendo que `false < true`. Naturalmente, ambos os operados de uma destas operações devem ser do mesmo tipo.

```
  var i, n : integer;
begin
read(n);
if n > 0
then
  i = 1;
  while i <= n do
    begin
    write(i);
    i = i + 1;
    end
endif
end
```

Figure A.1: Um programa em S2

| | | |
|---|---|---|
| `and` | `begin` | `boolean` |
| `do` | `else` | `end` |
| `endif` | `false` | `if` |
| `integer` | `not` | `or` |
| `read` | `then` | `true` |
| `var` | `while` | `write` |

Figure A.2: As palavras chave de S2

Os operadores `+`, `*`, `-` e `/` aplicam-se a valores inteiros resultando em inteiros. A semântica destes operadores é a mesma dos operadores da linguagem C.

Os operadores binários `and` e `or` e o unário `not` aceitam operandos booleanos e possuem o significado usual. A avaliação da expressão

```
expr1 and expr2
```

começa em `expr1`. Se `expr1` for `false`, toda a expressão será considerada falsa, mesmo sem o cálculo de `expr2`. Se for `true`, `expr2` será avaliada.

A expressão

```
expr1 or expr2
```

será considerada `true` se `expr1` for `true`. Caso contrário, esta expressão terá o valor de `expr2`.

## A.3 Identificadores

Identificadores são formados por letras, dígitos e o caráter sublinhado ("_"), iniciando-se por uma letra. Exemplos de identificadores válidos são:

```
getNum   x0   y1   get_Num
```

Os identificadores

```
_main    3ab    get$Num     write
```

são ilegais. "`write`" é ilegal por ser uma palavra chave. A lista das palavras chave da linguagem é exibida na Figura A.2.

Os primeiros 31 caracteres de um identificador são significativos e letras maiúsculas e minúsculas são consideradas diferentes. Assim, os identificadores

```
a12345678901234567890123456789012345678901Um
a12345678901234567890123456789012345678901Dois
```

são iguais. Todos os identificadores devem ser declarados antes de serem usados e nenhum pode ser declarado duas vezes.

## A.4 Atribuição

A atribuição de uma expressão `expr` a uma variável `aa` é

```
aa = expr
```

onde o tipo de `aa` e `expr` devem ser iguais.

## A.5 Comandos de Decisão

O comando `if` de S2 possui a forma

```
if expr
then
  StatementList
else
  StatementList
endif
```

onde a parte

```
else
  StatementList
endif
```

é opcional. `expr` é uma expressão booleana e `StatementList` é uma lista de zero ou mais comandos.

## A.6 Comandos de Repetição

O comando

```
while expr do
  Statement;
```

repete `Statement` enquanto a avaliação da expressão booleana `expr` resultar em `true`. Este comando também possui a forma

```
while expr do
  begin
  StatementList
  end
```

onde `StatementList` possui zero ou mais comandos.

## A.7 Entrada e Saída

A entrada de dados é feita pelo comando `read`:

```
read( IdList );
```

onde `IdList` é uma lista de uma ou mais variáveis inteiras. O comando

```
read( a1, a2, ...  an )
```
é equivalente a
```
read( a1 )
read( a2 )
...
read( an )
```
onde `read( a )` é equivalente ao código
```
gets(s);
sscanf(s, "%d", &a);
```
em C. Isto é, cada variável é lida em uma linha separada da entrada padrão.

O comando
```
write( expr1, expr2, ...  exprn )
```
escreve as expressões na saída padrão, sendo equivalente a
```
write( expr1 )
write( expr2 )
...
write( exprn )
```
O número `n` de expressões deve ser maior do que zero. O comando `write(expr)` é equivalente ao código
```
printf("%d ", expr);
```
em C. Apenas expressões inteiras podem ser parâmetros de `write`.

## A.8 A Gramática de S2

Esta seção define a gramática da linguagem. As palavras reservadas e símbolos da linguagem são mostrados entre “ e ”. Qualquer seqüência de símbolos entre { e } pode ser repetida zero ou mais vezes e qualquer seqüência de símbolos entre [ e ] é opcional. O prefixo Un em um nome significa a união de duas ou mais regras.

| | |
|---|---|
| Assignment | ::= Id “=” Expression |
| BasicType | ::= “integer” \| “boolean” |
| BasicValue | ::= IntValue \| BooleanValue |
| Block | ::= “begin” StatementList “end” |
| BooleanValue | ::= “true” \| “false” |
| Digit | ::= “0” \| ... \| “9” |
| Expression | ::= SimpleExpression [ Relation SimpleExpression ] |
| ExpressionList | ::= Expression { “,” Expression } |
| Factor | ::= BasicValue \| Id \| “(” Expression “)” \| “not” Factor |
| HighOperator | ::= “∗” \| “/” \| “and” |
| Id | ::= Letter { Letter \| Digit \| “_” } |
| IdList | ::= Id { “,” Id } |
| IfStat | ::= “if” Expression “then” StatementList [ “else” StatementList ] “endif” |
| IntValue | ::= Digit { Digit } |
| Letter | ::= “A” \| ... \| “Z” \| “a” \| ... \| “z” |
| LocalDec | ::= “var” VarDec { VarDec } |
| LowOperator | ::= “+” \| “−” \| “or” |

| | |
|---|---|
| Program | ::= [ LocalDec ] Block |
| ReadStat | ::= “read” “(” IdList “)” |
| Relation | ::= “==” \| “<” \| “>” \| “<=” \| “>=” \| “<>” |
| Signal | ::= “+” \| “−” |
| SimpleExpression | ::= [ Signal ] Term { LowOperator Term } |
| Statement | ::= Assignment “;” \| IfStat \| WhileStat \| ReadStat “;” \| WriteStat “;” \| “;” |
| StatementList | ::= { Statement } |
| Term | ::= Factor { HighOperator Factor } |
| UnStatBlock | ::= Statement \| Block |
| WriteStat | ::= “write” “(” ExpressionList “)” |
| VarDec | ::= IdList “:” BasicType “;” |
| WhileStat | ::= “while” Expression “do” UnStatBlock |

# 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.

[9] Guimarães, José de Oliveira. The Green Language. Computer Languages, Systems & Structures, Vol. 32, Issue 4, pages 203-215, December 2006.