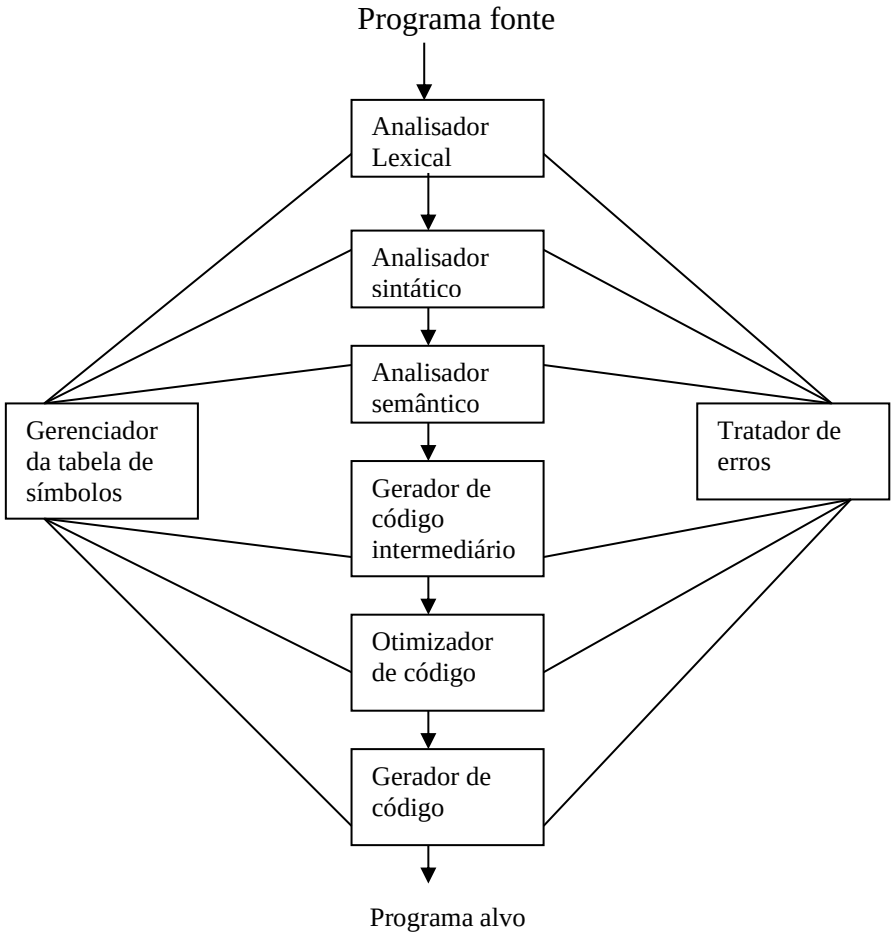
Compiladores

Ricardo Luís de Freitas

Índice

1 Estrutura Geral dos Compiladores

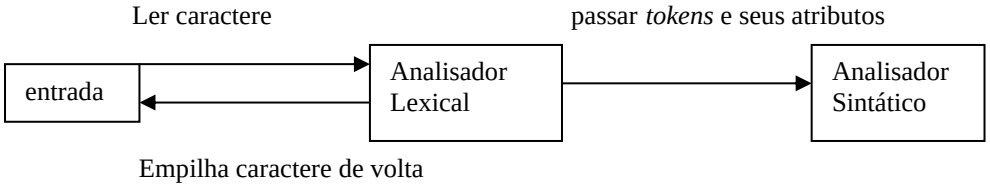


1.1 Funcionamento Básico de um Compilador

Este tópico será dedicado à uma breve explicação de cada módulo da estrutura do compilador. Note-se que esta estrutura é apenas uma representação didática do funcionamento do compilador. Isto pode dar uma impressão de simplicidade e que o compilador seja estático. Na verdade todos os módulos estão em constante atuação de forma quase simultânea. Iniciaremos com o primeiro módulo (análise Lexical) e seguiremos na ordem em que a figura indica para os demais módulos.

1.2 Analisador Lexical

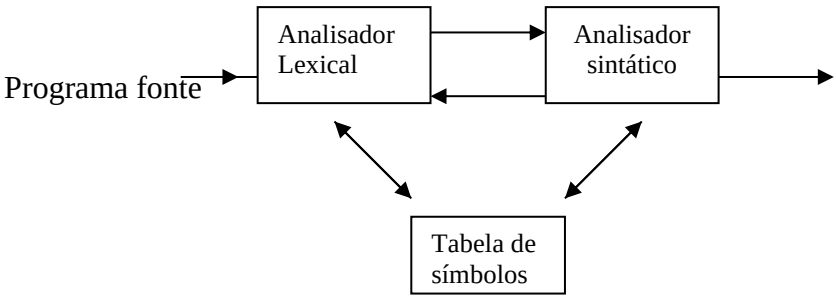
A análise Lexical é a primeira das três fases que compõe a análise do texto-fonte (*source*). Um Analisador Lexical, cumpre uma série de tarefas, não somente relacionadas a análise Lexical, de grande importância dentro do compilador. A principal função deste analisador é a de fragmentar o programa fonte de entrada em trechos elementares completos e com identidade própria. Estes componentes básicos são chamados *tokens.*



Conexão do Analisador Lexical com o Analisador Sintático

Do ponto de vista da implementação do compilador, o Analisador Lexical atua como uma interface entre o texto fonte e o analisador sintático, convertendo a sequência de caracteres que constituem o programa na sequência de átomos que o analisador sintático consumirá. Os átomos constituem-se em símbolos terminais da gramática. O Analisador Lexical do compilador tem como função varrer o programa fonte da esquerda para a direita, agrupando os símbolos de cada item léxico e determinando a sua classe. A seguir relacionamos algumas funções internas do Analisador Lexical.

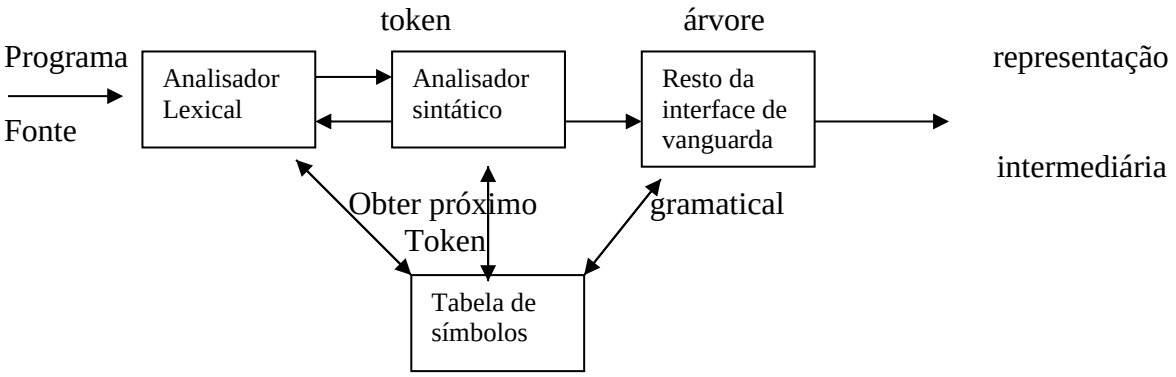
- Extração e classificação de átomos.
- Eliminação de delimitadores e comentários.
- Identificação de palavras reservadas.
- Recuperação de erros



Interação entre o Analisador Lexical, o analisador sintático e a tabela de símbolos.

1.3 Analisador Sintático

É o segundo grande bloco componente dos compiladores. A principal função deste módulo é a de promover a análise da sequência com que os átomos componentes do texto fonte se apresentam, a partir da qual se efetua a síntese da árvore sintática, com base na linguagem fonte. O analisador sintático irá processar a sequência de átomos, proveniente do texto fonte, extraídos pelo Analisador Lexical. A partir dessa sequência o analisador sintático efetua a verificação da ordem de apresentação na sequência, de acordo com a gramática na qual se baseia o reconhecedor. A análise sintática cuida exclusivamente da forma das sentenças da linguagem, e procura, com base na gramática, levantar a estrutura das mesmas.



Interação do analisador sintático com demais módulos

Para a representação de notação da gramática de uma linguagem, uma forma que vem ganhando muitos adeptos devido a legibilidade de seu aspecto gráfico são os diagramas sintáticos –já descritos anteriormente. Esses diagramas caracterizam-se por dois pontos diferentes: o ponto de partida e o de chegada. Os dois pontos são ligados entre si por um grafo orientado, cujos nódos representam pontos onde eventualmente podem ser feitas escolhas, e cujas arestas representam caminhos de percurso. Cada ramo pode incluir um terminal ou não terminal da gramática.

- Identificação de sentenças.
- Detecção de erros de sintaxe.
- Recuperação de erros.
- Montagem da árvore abstrata da sentença.
- Comando de ativação do Analisador Lexical.
- Ativação de rotinas de análise semântica.
- Ativação de rotinas de síntese do código objeto.

1.4 Analisador Semântico

O principal objetivo do analisador semântico é o de captar o significado das ações a serem tomadas no texto fonte. Ele compõe a terceira etapa de análise do compilador e refere-se à tradução do programa fonte para o programa objeto. Em geral a geração de código vem acompanhada de atividades da análise semântica. Denominamos semântica de uma sentença o significado por ela assumido dentro do texto analisado, enquanto semântica da linguagem é a interpretação que se pode atribuir ao conjunto de todas as sentenças. A fase da análise semântica, porém, é de difícil formalização, exigindo notações complexas. Devido a esse fato, na maioria das linguagens de programação adota-se especificações mais simplificadas, com bases informais, através de linguagens naturais.

A partir dessas informações, podemos afirmar que a principal função da análise semântica é criar, a partir do texto fonte, uma interpretação deste texto, expressa em alguma notação adequada que é, geralmente, uma linguagem intermediária do compilador. Esta operação é realizada com base nas informações existentes nas tabelas construídas pelos outros analisadores, tabelas de palavras reservadas, mapas e saídas dos demais analisadores. Deve ser observada a fundamental importância de uma análise semântica bem realizada, visto que

toda a síntese estará embasada na interpretação gerada por este analisador. Entre as ações semânticas encontramos tipicamente as seguintes:

- Manter informações sobre o escopo dos identificadores.
- Representar tipos de dados

1.5 Geração de Código

A geração de códigos é a fase final da compilação. Uma geração de códigos ótima é sempre difícil, sendo importante destacar as dificuldades decorrentes das especificidades de cada máquina. A importância do estudo deve-se ao fato de que um algoritmo de geração de códigos bem elaborado pode facilmente produzir códigos que serão executados até duas vezes mais rápido do que os códigos produzidos por um algoritmo pouco analisado. Deve-se destacar que a natureza exata do código objeto é altamente dependente da máquina e do sistema operacional.

PROGRAMAS OBJETO

Em geral, a entrada para uma rotina de geração de código é um programa em linguagem intermediária. A saída do gerador de código é o programa-objeto. Esta saída pode apresentar-se numa variedade de formas de código: linguagem de máquina absoluto, linguagem de máquina relocável, linguagem montadora (assembly), ou então outra linguagem qualquer de programação.

Para o caso de saída em código de linguagem de máquina absoluto, esta é colocada em uma posição fixa na memória e executada imediatamente, podendo ser compilada e executada em poucos segundos. Como exemplo, pode-se citar os compiladores usados para fins didáticos (student-jobs). A produção de códigos em linguagem de máquina relocável (módulo objeto), permite que sub-programas sejam compilados separadamente. Um conjunto de módulos objeto relocáveis podem ser reunidos e carregados por um montador (linker). O fato de ser possível compilar sub-rotinas separadamente e chamar outros programas compilados previamente a partir de um módulo objeto, permite uma grande flexibilidade. Muitos compiladores comerciais produzem módulos objeto relocáveis. A produção de um código em linguagem montadora como saída torna o processo de geração de códigos mais fácil. É possível gerar instruções simbólicas e utilizar as facilidades do montador para auxiliar a geração de códigos. O custo adicional é o passo em linguagem montada, depois da geração de código. Porém, como para a produção do código em linguagem montada o compilador não duplicará o trabalho do montador, esta escolha é outra alternativa razoável, especialmente para uma máquina com pouca memória disponível, onde um compilador precise utilizar diversos passos. A produção de uma linguagem de alto nível simplifica ainda mais a geração de códigos. Este tipo de implementação atua como pré-processador, e transfere os problemas de geração do código de máquina para um outro compilador.

1.6 Otimização Global e Local

As técnicas de otimização de código são geralmente aplicadas depois da análise sintática, usualmente antes e durante a geração de código. Essas técnicas consistem em detectar padrões no programa e substituí-los por construções equivalentes, porém mais

eficientes. Estes padrões podem ser locais ou globais, e a estratégia de substituição pode ser dependente ou independente da máquina.

Segundo Aho, o programa-fonte como entrada para um compilador é uma especificação de uma ação. O programa-objeto, a saída do compilador, é considerado como sendo outra especificação da mesma ação. Para cada programa-fonte, existem infinitos programas-objeto que implementam a mesma ação, ou seja, produzem a mesma saída para uma mesma entrada. Alguns desses programas-objeto podem ser melhores do que outros com relação a critérios tais como tamanho ou velocidade. O termo otimização de código refere-se às técnicas que um compilador pode empregar em uma tentativa para produzir um programa-objeto melhorado, para um dado programa-fonte. A qualidade de um programa-objeto pode ser medida por seu tamanho ou tempo de execução. Para programas grandes, o tempo de execução é particularmente importante. Para computadores de pequeno porte, o tamanho do código objeto pode ser tão importante quanto o tempo. Deve-se observar com cuidado o termo otimização, devido à dificuldade de se obter um compilador que produza o melhor programa-objeto possível, para qualquer programa-fonte, a um custo razoável. Assim, um termo mais exato para otimização de código seria melhoria de código ("code improvement").

1.7 Um Exemplo do Funcionamento de um Compilador

Como exemplo do funcionamento de um compilador vamos analisar um simples cálculo do perímetro de um triângulo. A primeira etapa consiste na análise do problema pelo programador. Se o problema é o cálculo de uma fórmula matemática, deve ser feito uma abstração matemática desta situação que é a seguinte:

Perímetro = Lado 1 + Lado 2 + Lado 3

O próximo passo seria converter os elementos da fórmula em variáveis associadas aos mesmos. Para Lado 1 chamaremos "b", para Lado 2 chamaremos "c", para Lado 3 chamaremos "d" e finalmente para Perímetro chamaremos "a". Em linguagem computacional (alto nível) a fórmula de perímetro representada pela seguinte equação:

a = b + c + d

O primeiro passo do compilador será então ler e identificar os símbolos, separando cada elemento. O analisador acessa a tabela de símbolos e palavras reservadas, montando assim uma relação entre os elementos e tokens da linguagem. Os tokens para este caso são: a,=,b,+,c,+,d,fim_de_linha. O caracter fim_de_linha é desprezado, assim como todos os espaços em branco que não interessam à compilação. A etapa seguinte será a análise sintática que verificará se as variáveis a, b, c, d são reconhecidas. Para isso o analisador sintático consulta a tabela interna de símbolos ou declarações explícitas, criada pelo analisador sintático. Deve-se observar que os analisadores estão continuamente em contato entre si, sendo mutuamente ativados. O passo seguinte é a verificação das operações "="e "+", que devem ser reconhecidas como pertencentes à gramática da linguagem. Na etapa seguinte de investigação, é necessário identificar o que deve ser construído. Neste exemplo deve-se abstrair o sentido da operação que se deseja efetuar. Trata-se de uma análise semântica muito simples. O analisador semântico constrói uma linguagem intermediária que deverá realizar a operação desejada e ser entendida pela máquina. Neste caso, o significado (semântica) que representa a equação do perímetro é a soma do Lado 1 com o Lado 2, o resultado desta soma

é somado com o Lado 3. Em uma linguagem intermediária isso seria representado da seguinte forma:

T 1 = b + c
T 2 = T 1 + d
A = T 2

Com esta etapa realizada o compilador passa para a fase de síntese, que produzirá a linguagem assembly (código produzido pela linguagem de montagem (assembler)), e, dependendo do compilador, efetuará duas otimizações, global e local. Na primeira fase da síntese o compilador irá buscar uma forma de otimizar, se possível, as sentenças obtidas pela análise semântica. Esta fase é chamada otimização global. Podemos notar que a sentença acima não foi otimizada. Uma possível otimização seria a eliminação das duas últimas sentenças em favor de uma que substituísse as duas como pode ser visto abaixo:

T 1 = b + c
A = T 1 + d

O passo seguinte seria a produção de linguagem assembly para o texto acima. Caso a linguagem a ser produzida fosse para a arquitetura **RISC_LIE,** o código que seria produzido a partir do texto não otimizado seria:

Ldxw b
Add c
Stxw t1
Ldxw t1
Add d
Stxw t2
Idwx t2
Stxw a

Ldxw: carrega palavra (correspondendo a load)
Stxw: armazena palavra (correspondendo a store)

O compilador poderia então, numa última etapa de síntese, efetuar uma otimização local, na linguagem assembly gerada. O resultado desta otimização seria:

Ldxw b
Add c
Add d
Stxw a

A linguagem assembly gerada é um mneumônico de código binário, que é entendido pela C. P. U. da máquina. Como as C. P. U. são dispositivos eletrônicos que operam sob forma de tensões elétricas em dois níveis (0 se não há tensão e 1 se há tensão), a linguagem assembly corresponderia a uma sequência de 0 e 1. Se o tamanho da palavra fosse 32 bits, teríamos a seguinte sentença correspondente em linguagem de máquina à instrução
Ldxw b :

0101 1000 0001 0000 0000 0010 1101 1001

2 Especificação de uma Linguagem Simplificada de Programação

Como já visto, consideramos linguagem uma determinada coleção de cadeias de símbolos de comprimentos finito. Estas cadeias são chamadas sentenças da linguagem e são formadas pela justaposição de elementos individuais, que são símbolos, átomos ou tokens da linguagem.

Podemos representar uma linguagem de três formas:

- Enumeração de todas as cadeias e símbolos possíveis que formam as sentenças;
- Regras de formação das cadeias reunidas em um conjunto de regras chamado gramática;
- Reconhecedor que atua através de regras de aceitação de cadeias .

A forma efetivamente utilizada é a última citada. As cadeias são reconhecidas como símbolos substituíveis, denominados não-terminais, e símbolos não substituíveis, denominados terminais.

Antes de se implementar uma linguagem deve-se procurar defini-la da maneira mais precisa possível. Em primeiro lugar devem ser especificadas todas as seqüências de símbolos que constituirão programas válidos para uma linguagem. Somente estas seqüências serão aceitas pelo compilador. Assim, tem-se especificada a sintaxe da linguagem. A segunda fase consiste na definição do significado associado a cada construção da linguagem, que compõem sua semântica. As regras que determinam a semântica de uma linguagem são muito mais difíceis de serem determinadas se comparadas as regras sintáticas.

Devido a dificuldade de se definir regras semânticas, uma vez que não é encontrado um padrão que especifique as semânticas de linguagens, a construção de compiladores corretos é bastante dificultada. Cada linguagem tem uma particularidade, o que exige um grau de significação diferenciado, interferindo diretamente na semântica da linguagem.

Os principais objetivos a serem considerados no projeto de compiladores são:

- Produzir um código objeto eficiente;
- Produzir códigos objetos pequenos;
- Minimizar o tempo necessário para compilar programas;
- Manter o compilador tão pequeno quanto possível;
- Produzir um compilador com boa capacidade de diagnóstico e recuperação de erros;
- Produzir um compilador confiável.

Como é fácil perceber os objetivos acima muitas vezes são conflitantes. Por exemplo, um compilador será mais lento e maior se o objetivo principal for um código mais eficiente, e vice-versa. Dessa forma, torna-se necessário definir qual dos objetivos é o principal e balizar o desenvolvimento do compilador nesse objetivo, porém sem perder de vista os restantes.

Outra decisão que deve ser tomada é quanto ao número de passagens que serão executadas pelo compilador. É considerada uma passagem cada vez que o compilador percorre o texto fonte. Do ponto de vista de simplicidade, os compiladores de uma única passagem são atraentes. No entanto, nem todas as linguagens permitem uma compilação deste tipo. Como compilador de uma passagem entende-se aqueles que necessitam percorrer o texto do código fonte apenas uma única vez para obter a compilação. Já os de várias passagens necessitam percorrer o texto várias vezes, sendo feita parcela da compilação a cada passagem. Como exemplo tem-se que na primeira passagem seria feita a análise Lexical, na segunda a sintática e assim por diante.

Embora existam diferentes notações para se especificar uma linguagem, a Forma Normal de Backus (BNF), tem se destacado devido a sua grande facilidade de compreensão. A BNF é uma metalinguagem, ou seja, uma linguagem usada para especificar outras linguagens. A formalização da sintaxe da linguagem é feita através da especificação das

regras de produção, ou regras de substituição, onde cada símbolo da metalinguagem é associada uma ou mais cadeias de símbolos, indicando as diversas possibilidades de substituição.

2.1 Descrição BNF da Linguagem Simplificada

A linguagem que será definida representa uma linguagem de programação estruturada semelhante à uma linguagem de programação estruturada. Esta linguagem receberá o nome de **LPD** (Linguagem de Programação Didática). O compilador a ser desenvolvido receberá o nome de **CSD** (Compilador Simplificado Didático). Os símbolos não terminais de nossa linguagem serão mnemônicos colocados entre parênteses angulares < e >, sendo os símbolos terminais colocados em negrito.

Descrição BNF da Linguagem Simplificada

<programa>::= **programa** <identificador> **;** <bloco> **.**

<bloco>::= [<etapa de declaração de variáveis>]
[<etapa de declaração de sub-rotinas>]
<comandos>

DECLARAÇÕES

<etapa de declaração de variáveis>::= **var** <declaração de variáveis> **;**
{<declaração de variáveis>**;**}

<declaração de variáveis>::= <identificador> {**,** <identificador>} **:** <tipo>

<tipo> ::= (**inteiro** | **booleano**)

<etapa de declaração de sub-rotinas> ::= (<declaração de procedimento>**;**|
<declaração de função>**;**)
{<declaração de procedimento>**;**|
<declaração de função>**;**}

<declaração de procedimento> ::= **procedimento** <identificador>**;**
<bloco>

<declaração de função> ::= **funcao** <identificador>**: <**tipo>**;**
<bloco>

COMANDOS

<comandos>::= **inicio**
<comando>{**;**<comando>}[**;**]
fim

<comando>::= (<atribuição_chprocedimento>|
<comando condicional> |
<comando enquanto> |
<comando leitura> |
<comando escrita> |
<comandos>)

<atribuição_chprocedimento>::= (<comando atribuicao>|
<chamada de procedimento>)

<comando atribuicao>::= <identificador> **:=** <expressão>

<chamada de procedimento>::= <identificador>

<comando condicional>::= **se** <expressão>
entao <comando>
[**senao** <comando>]

<comando enquanto> ::= **enquanto** <expressão> **faca** <comando>

<comando leitura> ::= **leia (** <identificador> **)**

<comando escrita> ::= **escreva (** <identificador> **)**

EXPRESSÕES

<expressão>::= <expressão simples> [<operador relacional><expressão simples>]

<operador relacional>::= (**!=** | **=** | **<** | **<=** | **>** | **>=**)

<expressão simples> ::= [**+** | **-**] <termo> {(**+** | **-** | **ou**) <termo> }

<termo>::= <fator> {(***** | **div** | **e**) <fator>}

<fator> ::= (<variável> |
<número> |
<chamada de função> |
(<expressão>**)** | **verdadeiro** | **falso |**
nao <fator>)

<variável> ::= <identificador>

<chamada de função> ::= <identificador >

NÚMEROS E IDENTIFICADORES

<identificador> ::= <letra> {<letra> | <dígito> | _ }

<número> ::= <dígito> {<dígito>}

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

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

COMENTÁRIOS

Uma vez que os comentários servem apenas como documentação do código fonte, ao realizar a compilação deste código faz-se necessário eliminar todo o conteúdo entre seus delimitadores.

delimitadores : **{ }**

Exercícios

1) Criar o diagrama sintático para a linguagem definida.

2) Represente a árvore de derivação para o seguinte programa

```
programa test;
var v: inteiro;
      i,max, juro,: inteiro;
inicio
 enquanto v != -1
 faca
 inicio
  {leia o valor  inicial}
  leia(v);
  {leia a taxa de juros }
  leia(juro);
  { Leia o periodo };
  leia(max);
  valor:= 1;
  i:= 1;
  enquanto i <= max          { (1+juro) elevado a n }
  faca inicio
          valor:= valor*(1+juro);
          i:= i+1
       fim;
   escreva(valor)
 fim
fim.
```

2.2 Fundamentos da Programação em LPD

2.2.1 A Linguagem LPD

Características da linguagem **LPD**:

- É um a linguagem altamente estruturada.
- Necessidades rigorosas na definição de variáveis e procedimentos. Você deve definir todas as variáveis, procedimentos, e funções, antes de usá-las.
- Funções e procedimentos. Estas são estruturas auto-delimitadas, nas quais você pode organizar cada uma das diferentes seções do seu programa.
- Variáveis locais e globais. Você pode declarar e usar variáveis locais em qualquer procedimento ou função. Os valores de dados de variáveis locais permanecem privados para as rotinas que os usem. Você pode também declarar variáveis globais que podem ser usadas em qualquer parte do programa.
- Decisões SE estruturadas. Elas permitem que você construa complexas tomadas de decisões.
- "Loop" estruturado. LPD fornece "loop" estruturado, no qual você pode especificar a condição sob a qual o "loop" deverá continuar (ENQUANTO).

2.2.2 Estrutura de um Programa em LPD

Agora vamos examinar a estrutura de um programa em LPD, focalizando os elementos individuais dessa estrutura, especificamente as declarações VAR, e de sub-rotinas (PROCEDIMENTO e FUNCAO), e a seção principal do programa.

Um programa em LPD consiste de diversos elementos estruturais, incluindo os seguintes:

- Um cabeçalho PROGRAMA, que fornece o nome do próprio programa. Este cabeçalho deve ser incluído em todos os programas, até nos mais triviais.
- O comando VAR que declara todas as variáveis globais usadas no programa. Esta declaração é opcional.
- Procedimentos e funções, que contém as instruções para tarefas definidas individualmente no programa.
- Uma seção principal do programa, que controla as ações do mesmo através das chamadas a procedimentos e funções incluídas no programa, ou através de comandos executáveis. Um esboço de um programa em LPD é visto a seguir:

```
programa NomePrograma;
var   {declaração de variaveis}
      {declaração de rotinas}
início {programa principal}
  {comandos da seção principal}
fim.  {programa principal}
```

As declarações VAR, de procedimentos e funções são opcionais; as únicas partes necessárias são o cabeçalho e a seção principal do programa. Consequentemente, a forma mais simples de um programa em LPD tem um cabeçalho e um bloco de comandos limitados por INÍCIO/FIM contendo um ou mais comandos executáveis.

programa NomePrograma;
início
  Comando;
  Comando;
  Comando;
fim.

2.2.3 Palavras Reservadas

As palavras que identificam as partes de um programa - por exemplo, **programa, var, procedimento, funcao, inicio, fim** - são chamadas palavras reservadas. Todas palavras reservadas possuem significados fixos dentro da linguagem, que não podem ser alterados. LPD possui as palavras reservadas listadas na tabela abaixo.

e	**inicio**	**booleano**	**div**	**faca**
senao	**fim**	**falso**	**funcao**	**se**
inteiro	**nao**	**ou**	**procedimento**	**programa**
leia	**entao**	**verdadeiro**	**var**	**enquanto**
escreva				

2.2.4 Identificadores Definidos pelo Usuário

Os identificadores na linguagem LPD podem ter comprimento de até 30 caracteres, onde todos os 30 são significativos. Os identificadores devem começar com uma letra do alfabeto, seguida de qualquer número de dígitos ou letras. O caractere sublinhado "_", também é legal dentro de um identificador.

2.2.5 Declarações VAR

Variáveis são identificadores que representam valores de dados num programa. LPD necessita que todas as variáveis sejam declaradas antes de serem usadas (a falha na declaração de uma variável resulta em um erro em tempo de compilação).

Você pode criar tantas variáveis quanto forem necessárias para satisfazer a necessidade de dados dentro de um determinado programa, dentro do limite de memória disponível. Tipos simples de variáveis são projetados para armazenar um único valor por vez; quando você atribui um novo valor a uma variável, o valor armazenado previamente será perdido.

Na seção VAR, você especifica explicitamente o tipo de cada variável que você cria. Aqui está o formato geral da seção VAR:

  .
  .
var
  NomedaVariável1 : Tipo1;
  NomedaVariável2 : Tipo2;
  .
  .

Alternativamente, você poderá usar o formato a seguir para declarar várias variáveis do mesmo tipo:

  .
  .
var

```
  NomedaVariável1,
  NomedaVariável2,
  NomedaVariável3 : Tipo;
    .
    .
```

Por exemplo, a seção a seguir declara uma variável do tipo inteiro e três variáveis do tipo booleano.

```
    .
var
  Codigo_Erro : inteiro;
  Sair,
  ExisteLista,
  Troca_Dados : booleano;
    .
```

2.2.6 Introdução aos Tipos de Dados

Os programas em LPD podem trabalhar com dados expressos como valores literais (tal como o número 6), ou como valores representados simbolicamente por identificadores (variáveis). Sem considerar como o valor está expresso, você deve tomar cuidado para distinguir os tipos de dados que LPD reconhece. O uso de dados de modo não apropriado resulta em erros em tempo de compilação.
LPD possui dois tipos de dados padrão:

- Tipo numérico inteiro.
- Tipo booleano.

Tipo Inteiro

LPD oferece o tipo numérico inteiro INTEGER. Este tipo fornece uma ampla faixa de valores de números inteiros.

Um inteiro não possui valor decimal, e é sempre perfeitamente preciso dentro de sua faixa.

Valores Booleanos

Um valor booleano pode ser tanto VERDADEIRO (verdadeiro), como FALSO (falso). (Os valores booleanos são assim chamados em homenagem ao matemático inglês do século dezenove *George Boole*. Usaremos algumas vezes o termo alternativo valor lógico.) LPD fornece um conjunto de operações lógicas e relacionais, que produzem expressões que resultam em valores VERDADEIRO ou FALSO. Veremos estas operações mais tarde.

O identificador padrão BOOLEANO define uma variável desse tipo, como neste exemplo:

```
    .
    .

var
  Achei : booleano;
    .
```

.

Os valores lógicos constantes VERDADEIRO e FALSO são também identificadores padrão na linguagem LPD, e tem significado fixo para esta.

2.2.7 Comando de Atribuição

Uma vez que você tenha declarado uma variável de qualquer tipo, poderá utilizar um comando de atribuição para armazenar um valor na mesma.

A sintaxe do comando de atribuição é a seguinte:
NomedaVariável := Expressão;

Para executar este comando, o CSD avalia a expressão localizada no lado direito do sinal de atribuição e armazena o valor resultante na variável à esquerda. O nome da variável aparece sempre sozinho, no lado esquerdo do sinal de atribuição. A expressão localizada do lado direito pode ser simples ou complexa. Considere estes exemplos:

```
    .
    .
var
  Achei : booleano;
  Numero,
  Linha,
  Coluna,
  Local_Erro : inteiro;
    .
    .
Achei := VERDADEIRO;
Numero := 5;
Local_Erro := Coluna * (Linha + 160);
    .
    .
```

O primeiro exemplo atribui o valor booleano VERDADEIRO à variável *Achei*, o segundo atribui o número 5 à variável *Numero*, o terceiro atribui o resultado de uma expressão aritmética à variável *Local_Erro*.

2.2.8 Procedimentos e Funções em LPD

Como já vimos, LPD suporta dois tipos de blocos estruturados : procedimentos e funções. Nós nos referimos a estes blocos genericamente como *sub-programas*, *sub-rotinas*, ou simplesmente *rotinas*. A estrutura de um procedimento é semelhante a de uma função (na LPD). A diferença principal é que a função é explicitamente definida para retornar um valor de um tipo especificado.

Uma chamada a um procedimento ou função é representada num programa em LPD por uma referência ao nome da própria rotina. Entretanto, o contexto de uma chamada depende da rotina ser procedimento ou função:

- A chamada de um procedimento é independente como um comando completo e resulta na execução do procedimento.
- A chamada de uma função é sempre parte de um comando maior. O nome da função representa o valor que a função definitivamente retorna. Em outras palavras, o nome de uma função é sempre um operando em um comando, nunca o próprio comando.

Aqui está o formato geral de um procedimento em LPD:

```
procedimento NomedaProcedimento;
var   {declaração de variáveis locais}

início
    {corpo do procedimento}
fim;
```

Nesta representação da sintaxe, *NomedaProcedimento* é o identificador que você cria como nome do procedimento. A declaração VAR dentro do bloco do procedimento define as variáveis locais para a rotina.

O formato geral de uma função é similar, mas tem duas diferenças importantes:

```
funcao NomedaFunção: TipodoResultado;
var   {declaração de variáveis locais}

inicio
    {corpo da função}
  NomedaFunção := ValordeRetorno;
fim;
```

NomedaFunção é o nome de identificação da função. Observe que o cabeçalho da função define o tipo da função - ou seja, o tipo do valor que a função retorna - com o identificador ***TipodoResultado*.**

Igualmente, dentro de toda função deverá existir um comando de atribuição que identifique o valor exato que a função retornará. Esse comando possui a seguinte forma:

```
  NomedaFunção := ValordeRetorno;
```

2.2.9 Chamadas a Procedimentos e Funções

Um comando de chamada a uma rotina direciona o CSD para executá-la. Uma chamada tem a forma mostrada abaixo:

```
  NomedaRotina
```

NomedaRotina faz referência à rotina definida dentro do programa. Por exemplo, considere a chamada a seguir da procedimento *Soma* definida anteriormente:

```
    .
    .
var  X,
      Y: inteiro;
    .
    .
  Soma;
    .
    .
```

Em contraste, a chamada a uma função não é independente como um comando, mas ao invés disso, aparece como parte de um comando maior. Geralmente uma função está relacionada a uma seqüência de operações que conduzem a um único valor calculado. Por exemplo, considere a função *Exp* a seguir:

```
funcao Exp: inteiro;
var Contador,
      Resultado,
      Base, Expoente : inteiro;
início
  leia(Base);
  leia(Expoente);
  Contador := 1;
  Resultado := 1;
  enquanto Contador <= Expoente faca
  início
    Resultado := Resultado * Base;
    Contador := Contador + 1;
  fim;
  Exp := Resultado;
fim;
```

A função está definida para retornar um valor do tipo INTEGER. O comando de atribuição no fim da função especifica que *Resultado* é o valor de retorno.

```
Auxiliar := Exp * 5;
```

A variável que recebe o valor retornado pela função, bem como o resto da expressão, deve ser do mesmo tipo da função. No exemplo acima, Auxiliar deve ser do tipo INTEGER.

2.2.10 Declaração Global X Local

Nós já vimos que os procedimentos e funções podem ter sua própria seção VAR. A localização de um comando de declaração determina o escopo dos identificadores correspondentes:

- Declarações em nível de programa (localizadas imediatamente abaixo do cabeçalho PROGRAMA), definem identificadores globais, ou seja, variáveis que podem ser usadas em qualquer ponto do programa.
- Declarações em nível de procedimentos e funções definem identificadores locais, ou seja, variáveis que só podem ser usadas dentro de uma rotina em particular, sendo removidas da memória quando a execução da rotina se completa.

Além disso, um procedimento ou função pode ter suas próprias definições de procedimentos ou funções aninhadas. Uma rotina que esteja localizada inteiramente dentro do bloco de definição de outra rotina, está aninhada dentro desta mesma rotina e é local para a rotina que a contém. Por exemplo, considere a rotina hipotética a seguir:

```
procedimento  MaisExterna;
var  {declaração de variáveis para MaisExterna};

  procedimento Dentro1;
  {Dentro1 e local para MaisExterna}
  var  {declaração de variáveis para Dentro1};

  início
     {comandos de Dentro1}
  fim;

  procedimento Dentro2;
  {Dentro2 também e local para MaisExterna}
  var  {declaração de variáveis para Dentro2};

       funcao Dentro3 : Booleano;
       {Dentro3 e local para Dentro2}
       var  {declaração de variáveis para Dentro3};

       início
          {comandos de Dentro3}
       fim;

   início
       {comandos de Dentro2}
   fim;

início
    {comandos de MaisExterna}
fim;
```

Neste exemplo, o procedimento *MaisExterna* possui dois procedimentos locais: *Dentro1* e *Dentro2*. Estes dois procedimentos só estão disponíveis para *MaisExterna*. Além disso, o procedimento *Dentro2* possui sua própria função local: *Dentro3*. *Dentro3* só está disponível para *Dentro2*. Os procedimentos e funções locais estão sempre localizados acima do bloco INICIO/FIM da rotina que as contém.

2.2.11 Operações

A linguagem LPD oferece duas categorias de operações, cada qual designada para ser executada sobre um tipo de operando específico. Estas categorias são as seguintes:

- Operações numéricas, que trabalham com tipos inteiros.
- Operações relacionais e lógicas, que resultam em valores booleanos.

Agora veremos cada uma destas categorias de operações.

Operações Numéricas

As quatro operações numéricas são representados na linguagem LPD pelos símbolos a seguir:

*	Multiplicação
div	Divisão
+	Adição
-	Subtração

Estas são chamadas operações binárias, por necessitarem sempre de dois operandos numéricos. Em contraste, o sinal de menos também pode ser utilizado para expressar um número negativo; a negação é uma operação unária. O operando DIV fornece o quociente inteiro da divisão de dois números; qualquer resto será simplesmente descartado. Por exemplo, considere a expressão a seguir:

```
a div b
```

Se a variável *a* possuir o valor 42, neste exemplo, e *b* o valor 8, o resultado da expressão será 5. O resto da divisão, que é 2, será descartado. Em qualquer expressão que contenha mais de uma operação, o CSD normalmente seguirá a ordem de precedência na avaliação de cada operação. Aqui está a ordem, da mais alta para a mais baixa precedência:

1. Negação
2. Multiplicação, Divisão
3. Soma, Subtração

Onde existir mais de uma operação com o mesmo nível de precedência, as operações com mesmo nível serão executadas da esquerda para direita. Por exemplo, considere o fragmento de programa, no qual todos os operandos são do tipo INTEGER:

```
a := 5;
b := 4;
c := 3;
d := 2;
x := a * b + c div d;
```

Para atribuir um valor a *x*, o CSD irá avaliar primeiramente as expressões do lado direito do sinal de atribuição, nesta ordem: a multiplicação, a divisão e, finalmente a adição. A variável *x* receberá, assim, o valor 21.

Se você desejar que o CSD avalie uma expressão numa ordem diferente do modo padrão, poderá fornecer parênteses dentro da expressão. O CSD executa operações dentro de parênteses antes de outras operações. Por exemplo, considere como a adição de parênteses afeta o resultado da expressão anterior:

```
x := a * (b + c) div d;
```

Neste caso, a adição é realizada primeiramente, depois a multiplicação e, por último, a divisão. Se as variáveis tiverem os mesmos valores que antes, o novo resultado será 17.

Os parênteses podem estar aninhados dentro de uma expressão, ou seja, um conjunto de parênteses pode aparecer completamente dentro de outro conjunto de parênteses. O CSD executa a operação localizada dentro dos parênteses mais internos primeiramente, e depois vai executando as operações contidas nos parênteses mais externos. Por

exemplo, no comando a seguir, a adição é executada primeiramente, depois a divisão e finalmente, a multiplicação, dando um resultado igual a 15:

```
x := a * ((b + c) div d);
```

Numa expressão aninhada, cada "abre parênteses" deve ser correspondido por um respectivo "fecha parênteses", ou ocorrerá um erro em tempo de compilação.

Operações Relacionais

Uma operação relacional compara dois itens de dados e fornece um valor booleano como resultado da comparação. Aqui estão os seis operadores relacionais e os seus significados:

```
=  Igual
<  Menor que
>  Maior que
<=  Menor ou igual a
>=  Maior ou igual a
!=  Diferente de
```

Você pode usar estas operações com dois valores do mesmo tipo. Por exemplo, digamos que a variável inteira *Escolha* contenha o valor 7. A primeira das expressões a seguir fornece um valor falso, e a segunda um valor verdadeiro:

```
Escolha <= 5
Escolha > 4
```

Operações Lógicas

A linguagem LPD possui três operações lógicas. Duas das quais - E, e OU - são operações binárias, usadas para combinar pares de valores lógicos em expressões lógicas compostas. A terceira, NAO, é uma operação unária que modifica um único operando lógico para o seu valor oposto. Fornecendo dois valores ou expressões lógicas, representadas por *Expressão1* e *Expressão2*, podemos descrever as três operações lógicas a seguir:

- Expressão1 E Expressão2 é verdadeiro somente se ambas, *Expressão1* e *Expressão2*, forem verdadeiras. Se uma for falsa ou se ambas forem falsas, a operação E também será falsa.
- Expressão1 OU Expressão2 é verdadeiro se tanto *Expressão1* como *Expressão2* forem verdadeiras. A operação OU só resulta em valores falsos se ambas, *Expressão1* e *Expressão2*, forem falsas.
- NAO Expressão1 avalia verdadeiro se *Expressão1* for falsa; de modo contrário, a operação NAO resultará em falso, se *Expressão1* for verdadeira.

Os operadores lógicos (E, OU, e NAO), têm uma ordem de precedência mais alta que os operadores relacionais (=, !=, <, >, <=, >=). Os parênteses são, portanto, necessários para forçar o CSD a avaliar as expressões relacionais antes de avaliar a expressão E.

2.2.12 Decisões e "Loop"

Para controlar eventos complexos, um programa iterativo em LPD deve ser capaz de tomar decisões, escolher entre os cursos de ações disponíveis, e executar ações repetidamente. Todas estas atividades necessitam de avaliações bem sucedidas de condições lógicas, com o objetivo de decidir uma direção específica na execução atual ou de

determinar a duração de uma seqüência de repetição em particular. A linguagem LPD fornece uma estrutura de controle para executar decisões, e outra para executar "loop"; essas estruturas são o nosso próximo assunto.

2.2.12.1 Estrutura de decisão SE

Uma estrutura de decisão SE seleciona um entre dois comandos (ou grupo de comandos), para execução. A estrutura consiste de uma clausula SE (se), em companhia de uma cláusula ENTAO (então) e uma SENAO (senão):

se Condição
  entao {comando ou grupo de comandos que
        serão executados se a condição for VERDADEIRO}
  senao {comando ou grupo de comandos que
       serão executados se a condição for FALSO};

Durante uma execução, o CSD começa por avaliar a condição da cláusula SE. Esta condição deve resultar num valor do tipo booleano, embora possa tomar uma variedade de formas, ela pode ser uma expressão de qualquer combinação de operadores lógicos e relacionais, ou simplesmente uma variável do tipo BOOLEANO.

Se a condição for VERDADEIRO, o comando ou comandos localizados entre o ENTAO e o SENAO são executados e a execução pula a cláusula SENAO. Alternativamente, se a expressão for FALSO, a execução é desviada diretamente para o comando, ou comandos, localizados após a cláusula SENAO.

As cláusulas ENTAO e SENAO podem ser seguidas tanto de um comando simples como de um comando composto. Um comando composto deve estar entre os marcadores INÍCIO e FIM. A estrutura do comando SE usando comandos compostos é a seguinte:

se Condição
  entao início
    {comandos da cláusula ENTAO}
  fim
  senao início
    {comandos de cláusula SENAO}
  fim;

Não cometa o erro de omitir os marcadores INÍCIO/FIM nos comandos compostos de uma estrutura SE. Além disso, não coloque o ponto e virgula entre o FIM e o SENAO numa estrutura SE. O CSD interpretará esta marcação incorreta como o fim do comando SE sendo que a cláusula SENAO, a seguir, resultará num erro em tempo de compilação.Tendo em vista que a cláusula SENAO é opcional, a forma mais simples da estrutura SE é a seguinte:

se Condição
  entao {comando ou comandos que
        serão executados se a condição for VERDADEIRO};

Neste caso o programa executa o comando ou comandos após a cláusula ENTAO somente se a condição for verdadeira. Se ela for falsa, o comando SE não terá nenhuma ação. As estruturas SE podem estar aninhadas. Em outras palavras, toda uma estrutura de decisão, com as cláusulas SE, ENTAO, e SENAO, pode aparecer dentro de uma cláusula

ENTAO ou SENAO de uma outra estrutura. O aninhamento de estruturas SE pode resultar em seqüências de decisões complexas e poderosas. Como exemplo, considere o fragmento de programa a seguir:

```
    .
    .
se N < 2
  entao G := G + N
  senao
        início
          H := G;
          P (N - 1, H);
          G := H;
          P (N - 2, G);
        fim;
    .
    .
```

Neste exemplo, a decisão será baseada no valor de *N*. Se *N* for menor que 2, o programa soma *N* ao valor de *G* e armazena em *G*. Se o valor de *N* for maior ou igual a 2, o programa executa o comando composto localizado após a cláusula SENAO.

2.2.12.2 A Estrutura do "Loop" ENQUANTO

Aqui está a sintaxe do "loop" ENQUANTO:

```
enquanto Condição faca
Comando;
```

Se o bloco do "loop" possuir um comando composto, a forma geral é a seguinte:

```
enquanto Condição faca
início
  {comandos que serão executados uma
   vez a cada iteração do "loop"}
fim;
```

A condição é uma expressão que o CSD pode avaliar como VERDADEIRO ou FALSO. O CSD avalia a expressão antes de cada iteração do "loop". A repetição continua enquanto a condição for VERDADEIRO. Em algum ponto, a ação dentro do "loop" comuta a condição para FALSO, antes de uma nova iteração a expressão será avaliada novamente; como ela resultará em FALSO, o "loop" não será executado. Como exemplo, considere o fragmento de programa a seguir:

```
    .
    .
Contador := 0;
enquanto Contador <= 11 faca
inicio
  escreva (Contador);
  Contador := Contador + 1;
fim;
```

A condição que controla este "loop" em particular é o progresso da variável *Contador* dentro do próprio "loop". Antes de cada iteração, o programa testa o valor de

Contador para ver se é menor que 11. Enquanto a condição for VERDADEIRO, o programa executará o "loop". Neste exemplo, o "loop" será executado até que a instrução que incrementa a variável *Contador* armazene na mesma um valor maior que 11. O programa irá, então, avaliar a condição, obterá o valor FALSO, e pulará o "loop" passando a executar a instrução seguinte ao FIM. Podemos descrever a ação desse "loop" do seguinte modo:

enquanto {enquanto} (contador for menor que 11) **faca**
inicio
 {escreva o valor do contador na tela,
 e incremente o seu valor de 1}
fim;

3 Análise Lexical

O analisador lexical é a primeira fase de um compilador. Sua tarefa principal é a de ler os caracteres de entrada e produzir uma sequência de *token*s que o analisador sintático utiliza. Essa interação, sumarizada esquematicamente na Figura 4.1, é comumente implementada fazendo-se com que o Analisador Lexical seja uma sub-rotina ou uma co-rotina do analisador sintático. Ao receber do analisador sintático um comando "obter o próximo *token*", o Analisador Lexical lê os caracteres de entrada até que possa identificar o próximo *token.*

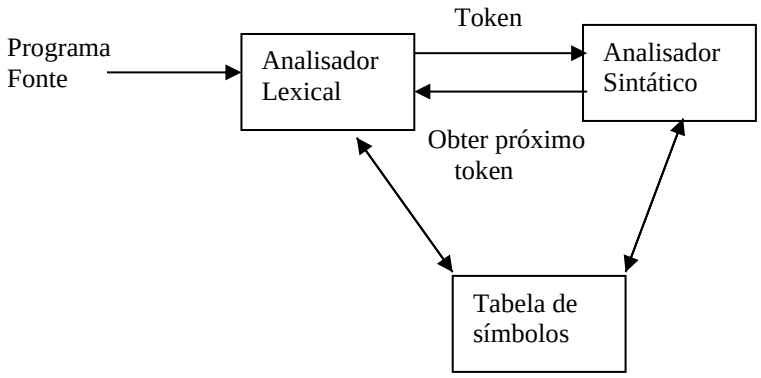


Figura 4.1: Interação do Analisador Lexical com o analisador sintático.

Como o Analisador Lexical é a parte do compilador que lê o texto-fonte, também pode realizar algumas tarefas secundárias ao nível da interface com o usuário. Uma delas é a de remover do programa-fonte os comentários e os espaços em branco, os últimos sob a forma de espaços, tabulações e caracteres de avanço de linha. Uma outra é a de correlacionar as mensagens de erro do compilador com o programa-fonte. Por exemplo, o Analisador Lexical pode controlar o número de caracteres examinados, de tal forma que um número de linha possa ser relacionado a uma mensagem de erro. Em alguns compiladores, o Analisador Lexical fica com a responsabilidade de fazer uma cópia do programa-fonte com as mensagens de erro associadas ao mesmo. Se a linguagem-fonte suporta algumas funções sob a forma de macros pré-processadas, as mesmas também podem ser implementadas na medida em que a análise Lexical vá se desenvolvendo.

Algumas vezes, os analisadores lexicais são divididos em duas fases em cascata, a primeira chamada de "varredura"(*scanning*) e a segunda de "análise Lexical". O *scanner* é responsável por realizar tarefas simples, enquanto o Analisador Lexical propriamente dito realiza as tarefas mais complexas. Por exemplo, um compilador poderia usar um *scanner* para eliminar os espaços da entrada.

3.1 Temas da Análise Lexical

Existem várias razões para se dividir a fase de análise da compilação em análise Lexical e análise sintática.

1. Um projeto mais simples talvez seja a consideração mais importante. A separação das análises Lexical e sintática frequentemente nos permite simplificar uma ou outra dessas fases. Por exemplo, um analisador sintático que incorpore as convenções para comentários e espaços em branco é significativamente mais complexo do que um que assuma que os mesmos já tenham sido removidos pelo Analisador Lexical. Se estivermos

projetando uma nova linguagem, separar as convenções lexicais das sintáticas pode levar a um projeto global de linguagem mais claro.

2. A eficiência do compilador é melhorada. Um Analisador Lexical separado nos permite construir um processador especializado e potencialmente mais eficiente para a tarefa. Uma grande quantidade de tempo é gasta lendo-se o programa-fonte e particionando-o em *token*s. Técnicas de **buferização** especializadas para a leitura de caracteres e o processamento de *token*s podem acelerar significativamente o desempenho de um compilador.
3. A portabilidade do compilador é realçada. As peculiaridades do alfabeto de entrada e outras anomalias específicas de dispositivos podem ser restringidas ao Analisador Lexical. A representação de símbolos especiais ou não-padrão, tais como ^ em Pascal, pode ser isolada no Analisador Lexical.

3.2 *Token*s, Padrões, Lexemas

Quando se fala sobre a análise Lexical, usamos os termos "*token*", "padrão" e "lexema" com significados específicos. Exemplos de seus usos são mostrados na Figura 4.2. Em geral, existe um conjunto de cadeias de entrada para as quais o mesmo *token* é produzido como saída. Esse conjunto de cadeias é descrito por uma regra chamada de um padrão associado ao *token* de entrada. O padrão é dito reconhecer cada cadeia do conjunto. Um lexema é um conjunto de caracteres no programa-fonte que é reconhecido pelo padrão de algum *token*. Por exemplo, no enunciado Pascal

```
const   pi  =  3.1416;
```

A subcadeia pi é um lexema para o *token* como símbolos terminais na gramática para a linguagem-fonte, usando nomes em negrito para representá-los. Os lexemas reconhecidos pelo padrão do *token* representa cadeias de caracteres no programa-fonte, e podem receber um tratamento conjunto, como instâncias de uma mesma unidade Lexical (por exemplo, instâncias de identificadores, números etc).

Na maioria das linguagens de programação, as seguintes construções são tratadas como *token*s: palavras-chave, operadores, identificadores, constantes, literais, cadeias e símbolos de pontuação, como parênteses, vírgulas e ponto-e-vírgulas. No exemplo acima, quando a sequência de caracteres pi aparece no programa-fonte, um *token* representando um identificador é repassado ao analisador sintático. O repasse de um *token* é frequentemente implementado transmitindo-se um inteiro associado ao *token*. É justamente esse inteiro que é designado por **id** na Figura 4.2.

Um padrão é uma regra que descreve o conjunto de lexemas que podem representar um *token* particular nos programas-fonte. O padrão para o *token* **const** na Figura 4.2 é exatamente a cadeia singela const, que soletra a palavra-chave. O padrão para o *token* **relacional** é o conjunto de todos os seis operadores relacionais.

Certas convenções de linguagem aumentam a dificuldade da análise Lexical. Linguagens como Fortran requerem certas construções em posições fixas na linha de entrada. Dessa forma, o alinhamento de um lexema pode ser importante na determinação da correção de um programa-fonte. A tendência no projeto moderno de linguagens de programação está na direção de entradas em formato livre, permitindo que as construções sejam colocadas em qualquer local da linha de entrada, e, por conseguinte, esse aspecto da análise Lexical se tornou menos importante.

O tratamento dos espaços varia grandemente de linguagem para linguagem. Em algumas linguagens, os espaços não são significativos, exceto quando dentro de literais do tipo cadeia de caracteres. Podem ser adicionados à vontade a fim de melhorar a legibilidade

de um programa. As convenções relacionadas aos espaços podem complicar grandemente a tarefa de identificação dos *token*s.

Um exemplo popular que ilustra a dificuldade potencial em se reconhecer *token*s é o enunciado DO de Fortran. No comando

DO 5 I = 1.25

Não podemos afirmar que DO seja parte do identificador DO5I, e não um identificador em si, até que tenhamos examinado o ponto decimal.
Por outro lado, no enunciado

DO 5 I = 1,25

Temos sete *token*s: a palavra-chave DO, o rótulo de enunciado 5, o identificador I, o operador =, a constante 1, a vírgula e a constante 25. Aqui não podemos estar certos, até que tenhamos examinado a vírgula, de que DO seja uma palavra-chave. Para aliviar esta incerteza, Fortran 77 permite que uma vírgula opcional seja colocada entre o rótulo e o índice do enunciado DO (no exemplo, a variável I). O uso dessa vírgula é encorajado porque a mesma ajuda a tornar o enunciado DO mais claro e legível.
Em muitas linguagens, certas cadeias são reservadas, isto é, seus significados são pré-definidos e não podem ser modificados pelo usuário. Se uma palavra-chave não for reservada, o Analisador Lexical precisará distinguir uma palavra-chave de um identificador definido pelo usuário. Em PL/I, as palavras-chave não são reservadas; consequentemente, as regras para essa distinção são um tanto complicadas, como o seguinte enunciado PL/I ilustra:

SE ENTAO ENTAO ENTAO = SENAO; SENAO SENAO = ENTAO;

TOKEN	LEXEMAS EXEMPLO	DESCRIÇÃO INFORMAL DO PADRÃO
Const	const	const
Se	se	se
Relacional	<, <=, =, !=, >, >=	< ou < = ou = ou != ou > = ou >
id	pi, contador, D2	**letra seguida por letras e/ou dígitos**
num	3 . 1416, 0, 6 . 02E23	**qualquer constante numérica**
literal	"conteúdo da memória"	**quaisquer caracteres entre aspas, exceto aspa**

Figura 4.2 Exemplo de *tokens.*

3.3 Atributos para os *Token*s

Quando um lexema for reconhecido por mais de um padrão, o Analisador Lexical precisará providenciar informações adicionais para as fases subsequentes do compilador a respeito do lexema particular que foi reconhecido. Por exemplo, o padrão **num** reconhece as duas cadeias 0 e 1, mas é essencial para o gerador de código ser informado sobre que cadeia foi efetivamente reconhecida, ou seja, não basta reconhecer **num,** tem que informar que ele vale 0 ou vale 1.

O Analisador Lexical coleta informações a respeito dos *token*s em seus atributos associados. Os *token*s influenciam decisões na análise gramatical; os atributos influenciam a

tradução dos *token*s. Do ponto de vista prático, o *token* possui usualmente somente um único atributo - um apontador para a entrada da tabela de símbolos na qual as informações sobre os mesmos são mantidas; o apontador se torna o atributo do *token*.

Para fins de diagnóstico, podemos estar interessados tanto no lexema de um identificador quanto no número da linha na qual o mesmo foi primeiramente examinado. Esses dois itens de informação podem, ambos, ser armazenados na entrada da tabela de símbolos para o identificador.

Exemplo: Os *token*s e os valores de atributos associados ao enunciado Fortran

E = M * C ** 2

São escritos abaixo como uma sequência de pares:

<**id**, apontador para a entrada da tabela de símbolos para E >

<**operador_de_atribuição,>**

<id, apontador para a entrada da tabela de símbolos para M>

<**operador_de_multiplicação,>**

<id, apontador a para a entrada da tabela de símbolos para C>

<**operador_de_exponenciação,>**

<num, valor inteiro 2>

Note-se que em certos pares não existe a necessidade de um valor de atributo; o primeiro componente é suficiente para identificar o lexema. Nesse pequeno exemplo, ao *token* **num** foi associado um atributo de valor inteiro. O compilador pode armazenar a cadeia de caracteres que forma o número numa tabela de símbolos e deixar o atributo do *token* **num** ser o apontador para a entrada da tabela.

3.4 Erros Léxicos

Poucos erros são distinguíveis somente no nível léxico, uma vez que um Analisador Lexical possui uma visão muito local do programa-fonte. Se cadeia fi for encontrada pela primeira vez num programa C, no contexto

fi (a == f(x)) ...

Um Analisador Lexical não poderá dizer se fi é a palavra-chave **if** incorretamente grafada ou um identificador de função não declarada. Como fi é um identificador válido, o Analisador Lexical precisa retornar o *token* identificador e deixar alguma fase posterior do compilador tratar o eventual erro.

Mas, suponhamos que aconteça uma situação na qual o Analisador Lexical seja incapaz de prosseguir, porque nenhum dos padrões reconheça um prefixo na entrada remanescente. Talvez a estratégia mais simples de recuperação seja a da “modalidade pânico”. Removemos sucessivos caracteres da entrada remanescente até que o Analisador Lexical possa encontrar um *token* bem-formado. Essa técnica de recuperação pode

ocasionalmente confundir o analisador sintático, mas num ambiente de computação interativo pode ser razoavelmente adequada.

Outras possíveis ações de recuperação de erros são:

1. remover um caractere estranho
2. inserir um caractere ausente.
3. substituir um caractere incorreto por um correto
4. transpor dois caracteres adjacentes.

Transformações de erros como essas podem ser experimentadas numa tentativa de se consertar a entrada. A mais simples de tais estratégias é a de verificar se um prefixo da entrada remanescente pode ser transformado num lexema válido através de uma única transformação. Essa estratégia assume que a maioria dos erros léxicos seja resultante de um único erro de transformação, uma suposição usualmente confirmada na prática, embora nem sempre.

Uma forma de se encontrar erros num programa é computar o número mínimo de transformações de erros requeridas para tornar um programa errado num que seja sintaticamente bem-formado. Dizemos que um programa errado possui K erros se a menor sequência de transformações de erros que irá mapeá-lo em algum programa válido possui comprimento K. A correção de erros de distância mínima é uma conveniente ferramenta teórica de longo alcance, mas que não é geralmente usada por ser custosa demais de implementar. Entretanto, uns poucos compiladores experimentais usaram o critério da distância mínima para realizar correções localizadas.

Outra forma que pode ser adotada é registrar como erro cada entrada que não se enquadra dentro dos padrões definidos para a linguagem.

3.5 Análise Lexical no CSD

Como foi dito anteriormente, a principal função do Analisador Lexical é ler os caracteres de entrada e produzir uma lista de *"token"* — neste processo são removidos do programa fonte os comentários e os espaços em branco — que o analisador sintático utilizará. Cada *"token"* representa um símbolo válido na linguagem LPD.

No **CSD** os *"token"* são representados internamente por uma estrutura formada por dois campos:

- Lexema: contém a sequência de caracteres — letras, números, etc. — que formam o token;

- Símbolo: este é um campo do tipo numérico que contém uma representação interna para o "token". Para aumentar a eficiência do sistema, toda a manipulação da estrutura do token será feita utilizando-se este campo. O **CSD** possui a seguinte tabela de símbolo:

TOKEN

Lexema	Símbolo
programa	sprograma
início	sinício
fim	sfim
procedimento	sprocedimento
funcao	sfuncao
se	sse
entao	sentao
senao	ssenao
enquanto	senquanto
faca	sfaca
:=	satribuição
escreva	sescreva
leia	sleia
var	svar
inteiro	sinteiro
booleano	sbooleano
identificador	sidentificador
número	snúmero
.	sponto
;	sponto_vírgula
,	svírgula
(	sabre_parênteses
)	sfecha_parênteses
>	smaior
>=	smaiorig
=	sig
<	smenor
<=	smenorig
!=	sdif
+	smais
-	smenos
*****	smult
div	sdiv
e	se
ou	sou
nao	snao
:	sdoispontos
verdadeiro	sverdadeiro
falso	sfalso

Por exemplo, na análise Lexical da sentença abaixo

```
se  contador > 10  { exemplo de comentário }
entao  escreva (contador)
senao   escreva (x);
```

É gerada a seguinte lista de *"tokens"*:

Lexema	Símbolo
se	sse
contador	sidentificador
>	smaior
10	snúmero
entao	sentao
escreva	sescreva
(	sabre_parênteses
contador	sidentificador
)	sfecha_parênteses
senao	ssenao
escreva	sescreva
(	sabre_parênteses
x	sidentificador
)	sfecha_parênteses
;	sponto_vírgula

Os comentários na linguagem **CSD** são feitos através do uso dos caracteres "{" — para abrir um comentário — e "}" — para fechar o comentário —, este caracteres assim como a seqüência de caracteres entre eles, não são considerados *"tokens"* pelo Analisador Lexical.

Estrutura da Lista de Tokens

A lista de Tokens é uma estrutura de fila — *"First in First out"* —, formada por uma lista encadeada onde cada nó possui o formato apresentado na figura abaixo.

Nó da lista de *"tokens".*

Os Algoritmos do Analisador Lexical no CSD

Uma vez defina a estrutura de dados do Analisador Lexical, é possível descrever seu algoritmo básico. No nível mais alto de abstração, o funcionamento do Analisador Lexical pode ser definido pelo algoritmo:

```
Algoritmo Analisador Lexical (Nível 0)
Inicio
   Abre arquivo fonte
   Enquanto não acabou o arquivo fonte
   Faça {
                Trata Comentário e Consome espaços
                Pega Token
                Coloca Token na Lista de Tokens
              }
   Fecha arquivo fonte
Fim
```

Na tentativa de aproximar o algoritmo acima de um código executável, são feitos refinamentos sucessivos do mesmo. Durante este processo, surgem novos procedimentos, que são refinados na medida do necessário.

```
Algoritmo Analisador Lexical (Nível 1)
Def. token: TipoToken
Inicio
   Abre arquivo fonte
   Ler(caracter)
   Enquanto não acabou o arquivo fonte
   Faça {Enquanto ((caractere = "{") ou
                        (caractere = espaço)) e
                        (não acabou o arquivo fonte)
                 Faça { Se caractere = "{"
                        Então {Enquanto (caractere ≠ "}" ) e
                                        (não acabou o arquivo fonte)
                                  Faça Ler(caracter)
                                  Ler(caracter)}
                          Enquanto (caractere = espaço) e
                                (não acabou o arquivo fonte)
                          Faça Ler(caracter)
                         }
         se caractere != fim de arquivo
         então {Pega Token
                  Insere Lista}
         }
   Fecha arquivo fonte
Fim.
```

```
Algoritmo Pega Token
Inicio
    Se caractere é digito
      Então Trata Digito
      Senão Se caractere é letra
              Então Trata Identificador e Palavra Reservada
                 Senão Se caractere = ":"
                          Então Trata Atribuição
                          Senão Se caractere ∈ {+,-,*}
                                   Então Trata Operador Aritmético
                                   Senão Se caractere ∈ {!,<,>,=}
                                            EntãoTrataOperadorRelacional
                                            Senão Se caractere ∈ {;  ,  (  ) .}
                                                     Então Trata Pontuação
                                                     Senão ERRO
Fim.


Algoritmo Trata Dígito
Def num : Palavra
Inicio
   num ← caractere
   Ler(caracter)
   Enquanto caractere é dígito
     Faça {
             num ← num + caractere
             Ler(caracter)
            }
   token.símbolo ← snúmero
   token.lexema ← num
Fim.

Algoritmo Trata Identificador e Palavra Reservada
Def id: Palavra
Inicio
   id ← caractere
   Ler(caracter)
   Enquanto caractere é letra ou dígito ou "_"
     Faça {id ← id + caractere
             Ler(caracter)
            }
   token.lexema ← id
    caso
       id = "programa" : token.símbolo ← sprograma
       id = "se" : token.símbolo ← sse
       id = "entao" : token.símbolo ← sentao
       id = "senao" : token.símbolo ← ssenao
       id = "enquanto" : token.símbolo ← senquanto
        id = "faca" : token.símbolo ← sfaca
        id = "início" : token.símbolo ← sinício
```

```
        id = “fim” : token.símbolo ← sfim
        id = “escreva” : token.símbolo ← sescreva
        id = “leia” :token.símbolo ← sleia
        id = “var” : token.símbolo ← svar
        id = “inteiro” : token.símbolo ← sinteiro
        id = “booleano” : token.símbolo ← sbooleano
        id = “verdadeiro” : token.símbolo ← sverdadeiro
        id = “falso” : token.símbolo ← sfalso
        id = “procedimento” : token.símbolo ← sprocedimento
        id = “funcao” : token.símbolo ← sfuncao
        id = “div” : token.símbolo ← sdiv
        id = “e” : token.símbolo ← se
        id = “ou” : token.símbolo ← sou
        id = “nao” : token.símbolo ← snao
      senão : token.símbolo ← sidentificador
Fim.
```

Exercícios:

Fazer os algoritmos para:

Trata Atribuição
Trata Operador Aritmético
Trata Operador Relacional
Trata Pontuação

4 Tabela de Símbolos

Uma função essencial do compilador é registrar os identificadores usados no programa fonte e coletar as informações sobre os seus diversos atributos. Esses atributos podem ser informações sobre a quantidade de memória reservada para o identificador, seu tipo, escopo, (onde é válido o programa) e, no caso de nomes de procedimentos, coisas tais como o número e os tipos de seus argumentos, o método de transmissão de cada um (por exemplo, por referência) e o tipo retornado, se algum. Para armazenar estas informações existe a tabela de símbolos.

Uma tabela de símbolos é uma estrutura de dados contendo um registro para cada identificador, com os campos contendo os atributos do identificador. As informações sobre o identificador são coletadas pelas fases de análise de um compilador e usada pelas fases de síntese de forma a gerar o código-alvo. Durante a Análise Lexical a cadeia de caracteres que forma um identificador é armazenada em uma entrada da tabela de símbolos. As fases seguintes do compilador acrescentam informações a esta entrada. A fase de geração utiliza as informações armazenadas para gerar o código adequado.

Um mecanismo de tabela de símbolos precisa permitir que adicionemos novas entradas e encontremos eficientemente as já existentes. É útil para um compilador ser capaz de crescer a tabela de símbolos dinamicamente, se necessário, em tempo de compilação. Se o tamanho da tabela for estático, ele deve ser grande o suficiente para tratar qualquer programa fonte que lhe seja apresentado. Tal tamanho fixo traz o problema de desperdício na maioria dos casos.

4.1 Entradas na Tabela de Símbolos

O formato das entradas na tabela de símbolos não tem que ser uniforme, porque a informação guardada a respeito de um nome depende do uso do mesmo. Cada entrada pode ser implementada como um registro consistindo em uma sequência de palavras consecutivas na memória. Para manter a tabela de símbolos uniforme, pode ser conveniente que algumas das informações a respeito de um nome sejam mantidas fora da entrada da tabela, com somente um apontador no registro apontando para cada uma destas informações.

Um aspecto importante que deve ser considerado é quanto aos procedimentos ou funções aninhados. Eles criam ambientes locais de variáveis. Para tratar isso temos as seguintes possibilidades:

Modelo de Pilha: uma vez terminada a compilação de um procedimento os símbolos locais são descartados.

Compilação em duas etapas: após a compilação de um procedimento ou função a árvore de programa continua fazendo referências aos símbolos locais.

A tabela de símbolos deve ser organizada de forma que uma vez terminada a compilação de um procedimento ou função, os símbolos locais continuem existindo, embora não façam parte do “caminho de busca” usado na compilação do restante do programa.

Algumas das estratégias para resolver esse problema:

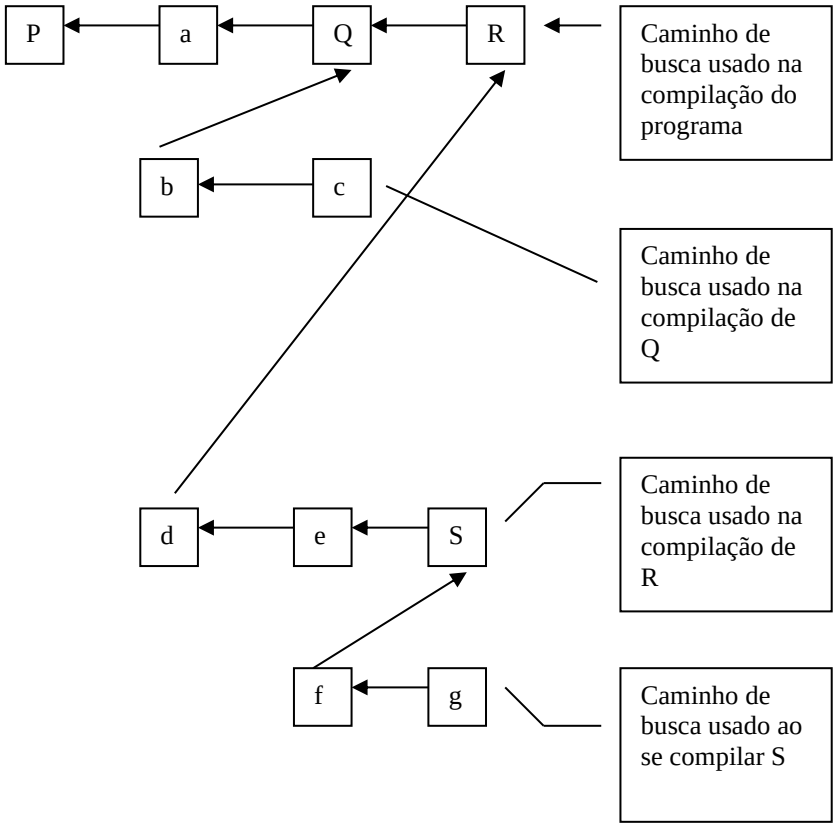
- gerar o código para um procedimento imediatamente após a árvore de programa correspondente ter sido construída. Após a geração de código, os símbolos locais podem ser descartados (o modelo de pilha continua válido);

- implementar a tabela de símbolos como uma lista ligada. Ao final da compilação de um procedimento, os símbolos locais são transferidos para uma outra lista. Dessa forma as referências aos símbolos feitas pela árvore de programa continuam válidas.

4.2 Tabela de Símbolos como "Árvore Invertida"

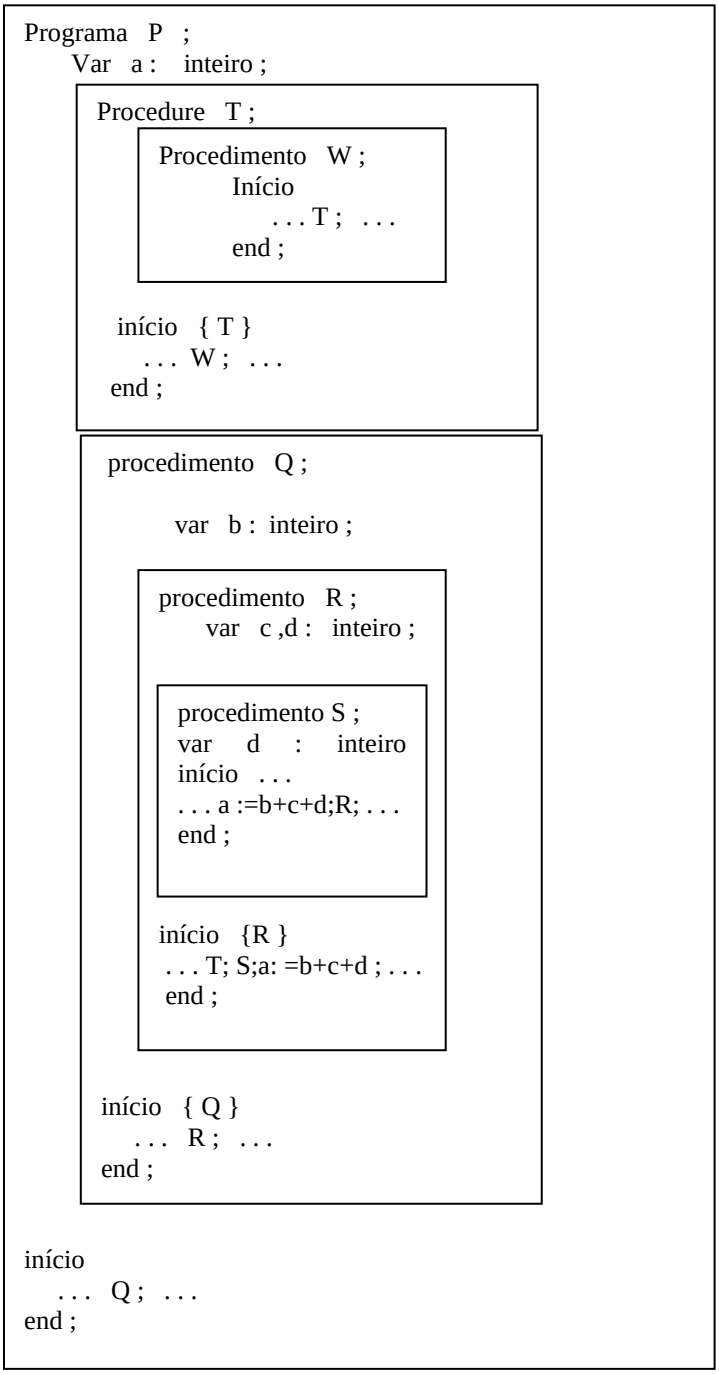
Uma forma simples de se implementar a tabela de símbolos é construir uma lista ligada onde cada nó aponta para o nó inserido anteriormente na tabela, manter um apontador para o último símbolo. Ao encerrar a compilação de um procedimento ou função, os símbolos locais continuam na lista e as novas inserções passam a ser feitas a partir do nó que escreve o procedimento.

```
Programa   p ;
   Var   a:   inteiro ;
      Procedimento   Q ;
       Var   b,c:   inteiro ;
        Início   . . .   fim;
         Procedimento   R ;
          Var   d,   e:   inteiro   ;
                Procedimento   S ;
                Var   f ,g:   inteiro   ;
                 Início   . . .   fim ;
            Início   . . .   fim ;
                 . . .
           fim   .
```



4.3 "Visibilidade" dos Símbolos

A tabela símbolos deve ser organizada de forma a refletir a visibilidade de símbolos definida na linguagem.



- O procedimento T, é "visível" dentro do procedimento W (embora o seu "corpo" possa ainda não ter sido compilado).
- S "enxerga" as variáveis a, b, e c definidas em nível mais externo. A variável d, definida em R é inacessível em S porque existe em S uma outra definição do nome d que "esconde" a anterior.
- R "enxerga" as variáveis a, b, c e d, e os procedimentos Q e T, mais externos. Vê também o procedimento S, interno a ele, mas não as definições internas a S, ou seja, a variável d.

EXEMPLO

No exemplo acima, os nomes visíveis em cada procedimento são:

- Em T, W e a são visíveis
- Em W, T e a são visíveis
- Em Q, T, a, b e R são visíveis
- Em R, T, a, Q, b, c, d e S são visíveis
- Em S, T, a, Q, b, c, R e d são visíveis
- Em P, T, a e Q são visíveis

Durante a compilação de um procedimento, devem estar visíveis os nomes definidos localmente, e os nomes definidos dentro dos escopos mais externos dentro dos quais o procedimento está contido.

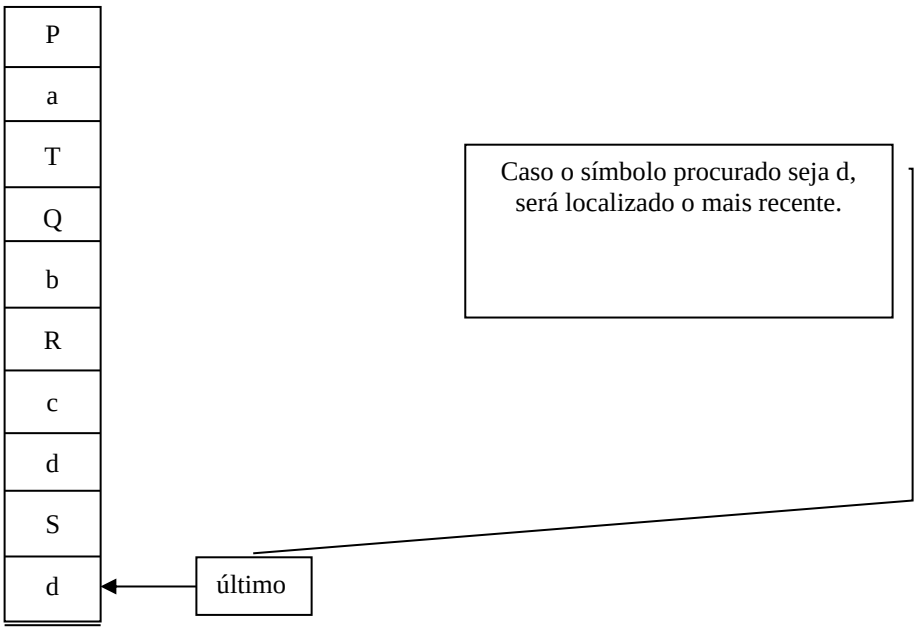
A prioridade de busca deve ser tal que, no caso de um mesmo nome estar definido em mais de um escopo, a definição feita no escopo mais interno prevalece.

Uma vez completada a compilação de um procedimento, os símbolos internos a ele deixam de ser visíveis. O próprio procedimento continua visível dentro do escopo onde ele é definido.

4.4 A Tabela de Símbolos Organizada como um Vetor

Uma forma simples de se organizar a tabela de símbolos é como um vetor onde os símbolos são inseridos na mesma ordem em que são declarados. As buscas nessa tabela são feitas sempre do mais recente para o mais antigo. Ao se encerrar a compilação de um procedimento, todos os símbolos locais são removidos da tabela.

Exemplo (situação de tabela durante a compilação do procedimento S, acima):



Como as inserções são sempre no fim da tabela e como as retiradas sempre são dos símbolos mais recentes (locais ao procedimento que acabou de ser compilado), a tabela funciona como uma pilha (!).

Este modelo para a tabela, usando um vetor, supõe que as buscas serão sequenciais. Isso pode ser proibitivo se o número de símbolos for muito grande. A mesma "lógica" de funcionamento pode ser aplicada a outras organizações de tabela visando a melhoria no tempo de acesso.

4.5 Tabela de Símbolos no CSD

A Tabela de Símbolos no CSD deverá seguir um dos padrões vistos anteriormente. A única restrição ficará por conta do registro a ser definido, que deverá representar as seguintes informações:

- Nome do identificador (lexema)
- Escopo (nível de declaração)
- Tipo (padrão do identificador)
- Memória (endereço de memória alocado)

Além do registro deverão ser implementados os seguintes procedimentos básicos:

Procedimentos:

- Insere na Tabela: inserir os identificadores na Tabela.

- Consulta a Tabela: percorre a Tabela procurando por um identificador. Devolve todos os campos do registro.

- Coloca Tipo nas Variáveis: percorre a tabela do final para o começo substituindo todos os campos tipo que possuem o valor *variável* pelo tipo agora localizado.

5 Análise Sintática

Cada linguagem de programação possui as regras que descrevem a estrutura sintática dos programas bem-formados. Em C, por exemplo, um programa é constituído por blocos, um bloco por comandos, um comando por expressões, uma expressão por *token*s e assim por diante. A sintaxe das construções de uma linguagem de programação pode ser descrita pelas gramáticas livres de contexto (usando possivelmente a notação BNF). As gramáticas oferecem vantagens significativas tanto para os projetistas de linguagens quanto para os escritores de compiladores.

- Uma gramática oferece, para uma linguagem de programação, uma especificação sintática precisa e fácil de entender.
- Para certas classes de gramáticas, podemos construir automaticamente um analisador sintático que determine se um programa-fonte está sintaticamente bem-formado. Como benefício adicional, o processo de construção do analisador pode revelar ambiguidades sintáticas bem como outras construções difíceis de se analisar gramaticalmente, as quais poderiam, de outra forma, seguir indetectadas na fase de projeto inicial de uma linguagem e de seu compilador.
- Uma gramática propriamente projetada implica uma estrutura de linguagem de programação útil à tradução correta de programas-fonte em códigos-objeto e, também, à detecção de erros. Existem ferramentas disponíveis para a conversão de descrições de traduções, baseadas em gramáticas, em programas operacionais.
- As linguagens evoluíram ao longo de um certo período de tempo, adquirindo novas construções e realizando tarefas adicionais. Essas novas construções podem ser mais facilmente incluídas quando existe uma implementação baseada numa descrição gramatical da linguagem.

O núcleo desta seção está devotado aos métodos de análise sintática que são tipicamente usados nos compiladores. Apresentamos primeiramente os conceitos básicos, em seguida as técnicas adequadas à implementação manual e finalmente os algoritmos usados.

5.1 O Papel do Analisador Sintático

Em nosso modelo de compilador, o analisador sintático obtém uma cadeia de *token*s proveniente do Analisador Lexical, como mostrado na Figura 6.1, e verifica se a mesma pode ser gerada pela gramática da linguagem-fonte. Esperamos que o analisador sintático relate quaisquer erros de sintaxe de uma forma inteligível.

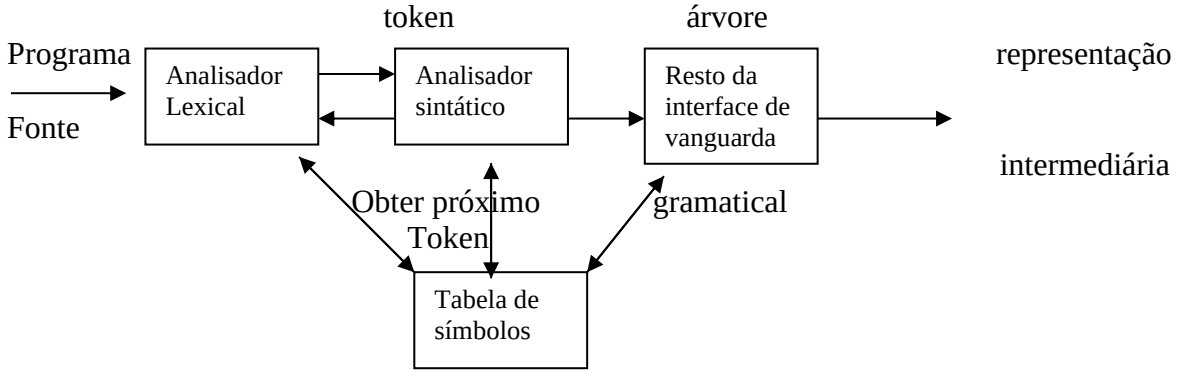


Figura 6.1: Posição de um analisador sintático num modelo de compilador

Existem três tipos gerais de analisadores sintáticos. Os métodos universais de análise sintática podem tratar qualquer gramática. Esses métodos, entretanto, são muito ineficientes para se usar num compilador comercial. Os métodos mais comumente usados nos compiladores são classificados como descendentes (*top-down*) ou ascendentes (*bottom-up*). Como indicado por seus nomes, os analisadores sintáticos descendentes constroem árvores do topo (raiz) para o fundo (folhas), enquanto que os ascendentes começam pelas folhas e trabalham árvore acima até a raiz. Em ambos os casos, a entrada é varrida da esquerda para a direita, um símbolo de cada vez.

Os métodos de análise sintática mais eficientes, tanto descendentes quanto ascendentes, trabalham somente em determinadas subclasses de gramáticas, mas várias dessas subclasses, como as das gramáticas LL(*left-left*) e LR(*left-right*), são suficientemente expressivas para descrever a maioria das construções sintáticas das linguagens de programação. Os analisadores implementados manualmente trabalham frequentemente com gramáticas LL. Os da classe mais ampla das gramáticas LR são usualmente construídos através de ferramentas automatizadas.

Nesta seção, assumimos que a saída de um analisador sintático seja alguma representação da árvore gramatical para o fluxo de *token*s produzido pelo Analisador Lexical. Na prática existe um certo número de tarefas que poderiam ser conduzidas durante a análise sintática, tais como coletar informações sobre os vários *token*s na tabela de símbolos, realizar a verificação de tipos e outras formas de análise semântica, assim como gerar o código intermediário. Juntamos todos esses tipos de atividades na caixa "resto da interface da vanguarda" da Figura 6.1.

5.2 Análise Sintática Ascendente

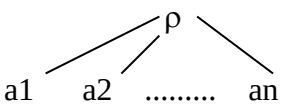
Neste tipo de análise sintática a construção da árvore de derivação para uma determinada cadeia começa pelas folhas da árvore e segue em direção de sua raiz. Caso seja obtida uma árvore cuja raiz tem como rótulo o símbolo inicial da gramática, e na qual a sequência dos rótulos das folhas forma a cadeia dada, então a cadeia é uma sentença da linguagem, e a árvore obtida é a sua árvore de derivação.

O processo de se partir das folhas em direção à raiz de uma árvore de derivação é conhecido como *redução*. A cada passo procura-se reduzir uma cadeia (ou sub-cadeia) ao seu símbolo de origem, objetivando-se atingir o símbolo inicial da gramática.

Funcionamento geral:

1) Toma-se uma determinada cadeia β.

2) Procura-se por uma sub-cadeia **a1,a2,...an** de β que possa ser substituída pelo seu símbolo de origem ρ**.** Ou seja, $\rho \rightarrow$ **a1,a2,...an**. Assim $\beta = \alpha 1\ \rho\ \alpha 2$**.**



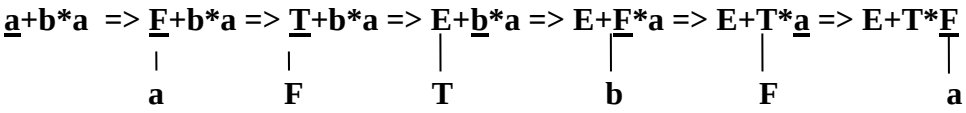
3) Repetir o passo (2) até que β **= símbolo inicial** da gramática. Caso isso não seja possível tem-se que a cadeia analisada não pertence a linguagem especificada.

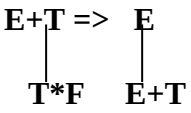
Exemplo:

G= ({E,T,F},{a,b,+,*,(,)},P,E)

P: $E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow a \mid b \mid (E)$

Verificando se a cadeia **a+b*a** pertence a linguagem utilizando-se de análise sintática ascendente.





É possível dizer que os maiores problemas na análise sintática ascendente residem na dificuldade de se determinar qual a parcela da cadeia (sub-cadeia) que deverá ser reduzida, bem como qual será a produção a ser utilizada na redução, para o caso de existirem mais do que uma possibilidade.

5.3 Análise Sintática Descendente

Neste tipo de análise sintática tem-se procedimento inverso ao da anterior. Aqui a análise parte da raiz da árvore de derivação e segue em direção as folhas, ou seja, a partir do símbolo inicial da gramática vai-se procurando substituir os símbolos não-terminais de forma a obter nas folhas da árvore a cadeia desejada.

Funcionamento geral:

1) Toma-se uma determinada cadeia **β** e o símbolo inicial **S** da gramática.

2) Procura-se por uma regra de derivação que permita derivar **β** em sua totalidade ou pelo menos se aproximar desta.

3) Repetir o passo (2) até que a cadeia não apresente mais símbolos não terminais. Nesta situação é verificado se a cadeia obtida coincide com **β.** Caso isso não seja possível tem-se que a cadeia analisada não pertence a linguagem especificada.

Embora apresente os mesmos problemas que a anterior, apresenta a possibilidade de se poder implementar facilmente compiladores para linguagens obtidas de gramáticas LL(1) (análise da cadeia da esquerda para a direita e derivações esquerdas observando apenas o primeiro símbolo da cadeia para decidir qual regra de derivação será utilizada).

Nas gramáticas LL(1) cada regra de derivação apresenta símbolo inicial da cadeia resultante diferenciado dos demais, o que facilita sua escolha. Como nas linguagens de programação isso ocorre com frequência, ela se mostra adequada para uso em compiladores.

No resto desta seção, consideraremos a natureza dos erros sintáticos e as estratégias gerais para sua recuperação. Duas dessas estratégias, chamadas “modalidade do desespero”e recuperação em nível de frase, são discutidas mais pormenorizadamente junto com os métodos individuais de análise sintática. A implementação de cada estratégia requer o julgamento do desenvolvedor do compilador, mas daremos algumas diretrizes gerais relacionadas a essas abordagens.

5.4 Tratamento dos Erros de Sintaxe

Se um compilador tivesse que processar somente programas corretos, seu projeto e sua implementação seriam grandemente simplificados. Mas os programadores frequentemente escrevem programas incorretos, e um bom compilador deveria assistir o programador na identificação e localização de erros. É gritante que, apesar dos erros serem lugar-comum, poucas linguagens sejam projetadas tendo-se o tratamento de erros em mente. Nossa civilização seria radicalmente diferente se as linguagens faladas tivessem as mesmas exigências de correção sintática que as das linguagens de computadores.

A maioria das especificações das linguagens de programação não descreve como um compilador deveria responder aos erros; tal tarefa é deixada para o projetista do compilador. O planejamento do tratamento de erros exatamente desde o início poderia tanto simplificar a estrutura de um compilador quanto melhorar sua resposta aos erros.

Sabemos que os programas podem conter erros em muitos níveis diferentes. Por exemplo, os erros podem ser:

- léxicos, tais como errar a grafia de um identificador, palavra-chave ou operador
- sintáticos, tais como uma expressão aritmética com parênteses não-balanceados
- semânticos, tais como um operador aplicado à um operando incompatível
- lógicos, tais como uma chamada infinitamente recursiva

Frequentemente, boa parte da detecção e recuperação de erros num compilador gira em torno da fase de análise sintática. Isto porque os erros ou são sintáticos por natureza ou são expostos quando o fluxo de *token*s proveniente do Analisador Lexical desobedece às regras gramaticais que definem a linguagem de programação.

Outra razão está na precisão dos modernos métodos de análise sintática; podem detectar muito eficientemente a presença de erros sintáticos num programa. Detectar precisamente a presença de erros semânticos ou lógicos em tempo de compilação é uma tarefa muito mais difícil.

O tratador de erros num analisador sintático possui metas simples de serem estabelecidas:

- Deve relatar a presença de erros clara e acuradamente.
- Deve se recuperar de cada erro suficientemente rápido a fim de ser capaz de detectar erros subsequentes.
- Não deve retardar significativamente o processamento de programas corretos.

A realização efetiva dessas metas apresenta desafios difíceis.

Felizmente, os erros comuns são simples e frequentemente basta um mecanismo de tratamento de erros relativamente direto. Em alguns casos, entretanto, um erro pode ter ocorrido muito antes de sua presença ter sido detectada e sua natureza precisa pode ser muito difícil de ser deduzida. Em casos difíceis, o tratador de erros pode ter que adivinhar o que o programador tinha em mente quando o programa foi escrito.

Vários métodos de análise sintática, tais como os métodos LL e LR, detectam os erros tão cedo quando possível. Mais precisamente, possuem a propriedade do prefixo viável, significando que detectam que um erro ocorreu tão logo tenham examinado um prefixo da entrada que não seja o de qualquer cadeia da linguagem.

Exemplo: A fim de ter uma apreciação dos tipos de erros que ocorrem na prática, vamos examinar os erros que Ripley e Druseikis encontraram numa amostra de programa Pascal de estudantes.

Ripley e Druseikis descobriram que os erros não ocorrem com tanta frequência; 60% dos programas compilados estavam semântica e sintaticamente corretos. Mesmo quando os erros ocorriam de fato, eram um tanto dispersos; 80% dos enunciados contendo erros possuíam apenas um, 13% dois. Finalmente, a maioria constituía-se de erros triviais; 90% eram erros em um único *token.*

Muitos dos erros poderiam ser classificados simplificadamente: 60% eram erros de pontuação, 20% de operadores e operandos, 15% de palavras-chave e os 5% restantes de outros tipos. O grosso dos erros de pontuação girava em torno do uso incorreto do ponto-e-vírgula.

Para alguns erros concretos, consideremos o seguinte programa em LPD (estendida com parâmetros).

```
(1)  programa  prmax;
(2)  var
(3)      x,  y:  inteiro;
(4)  funcao max  (i:  inteiro ;  j:  inteiro)  : inteiro;
(5)  {  retorna  maximo  de  i  e  j  }
(6)  inicio
(7)      se  i  >  j
(8)      entao  max  :=  i
(9)      senao  max  :=  j
(10) fim;
(11) inicio
(12)     leia  (x,  y)  ;
(13)     escreva  (max  (x,  y)  )
(14) fim.
```

Um erro comum de pontuação é o de se usar uma vírgula em lugar de ponto-e-vírgula na lista de argumentos de uma declaração de função (por exemplo, usar uma vírgula em lugar do primeiro ponto-e-vírgula à linha (4)); outro é o de omitir um ponto-e-vírgula obrigatório ao final de uma linha (por exemplo, o ponto-e-vírgula ao final da linha (4)); um terceiro é o de colocar um ponto-e-vírgula estranho ao fim de uma linha antes de um senao (por exemplo, colocar um ponto-e-vírgula ao fim da linha (7)).

Talvez uma razão pela qual os erros de ponto-e-vírgula sejam tão comuns é que seu uso varia grandemente de uma linguagem para outra. Em Pascal, um ponto-e-vírgula é um separador de enunciados; em PL/1 e C é um terminador. Alguns estudos têm sugerido que a última utilização é menos propensa a erros.

Um exemplo típico de um erro de operador é o de omitir os dois pontos em :=. Erros de grafia em palavras-chave são menos comuns, mas ocorrem principalmente no início da aprendizagem de uma nova linguagem de programação.

Muitos compiladores não têm dificuldades em tratar os erros comuns de inserção, remoção e transformação. De fato, vários compiladores irão tratar erro deste tipo como um erro comum de pontuação ou de operador; emitindo somente um diagnóstico de alerta, apontando a construção ilegal.

No entanto, um outro tipo de erro é muito mais difícil de se reparar corretamente. É o caso de um início ou fim ausente (por exemplo, a omissão da linha (10)). A maioria dos compiladores não irá tentar reparar esse tipo de erro.

Como deveria um tratador de erros reportar a presença de um erro? No mínimo, deveria informar o local no programa-fonte onde o mesmo foi detectado, uma vez que existe uma boa chance de o erro efetivo ter ocorrido uns poucos *tokens* antes. Uma estratégia comum empregada por muitos compiladores é a de imprimir a linha ilegal com um apontador para a posição na qual o erro foi detectado. Se existir um razoável prognóstico do que o erro realmente foi, uma mensagem de diagnóstico informativa é também incluída; por exemplo, "ponto-e-vírgula ausente nesta posição".

Uma vez que o erro tenha sido detectado, como deveria o analisador sintático se recuperar? Como veremos, existe um número de estratégias gerais, mas nenhum método claramente se impõe sobre os demais. Num tempo atrás, não era adequado para o analisador sintático encerrar logo após detectar o primeiro erro, porque o processamento da entrada restante ainda poderia revelar outros. Assim, existia alguma forma de recuperação de erros na qual o analisador tentava restaurar a si mesmo para um estado onde o processamento da entrada pudesse continuar com uma razoável esperança de que o resto correto da entrada fosse analisado e tratado adequadamente pelo compilador.

Um trabalho inadequado de recuperação pode introduzir uma avalancha de erros "espúrios", que não foram cometidos pelo programador, mas introduzidos pelas modificações no estado do analisador sintático durante a recuperação de erros. Numa forma similar, uma recuperação de erros sintáticos pode introduzir erros semânticos espúrios que serão detectados posteriormente pelas fases de análise semântica e de geração de código. Por exemplo, ao se recuperar de um erro, o analisador pode pular a declaração de alguma variável, digamos zap. Quando zap for posteriormente encontrada nas expressões, não haverá nada sintaticamente errado, mas como não há uma entrada na tabela de símbolos para zap, a mensagem "zap não definido" será gerada.

Uma estratégia cautelosa para o compilador é a de inibir as mensagens de erro que provenham de erros descobertos muito proximamente no fluxo de entrada. Em alguns casos, pode haver erros demais para o compilador continuar um processamento sensível (por exemplo, como deveria um compilador C responder ao receber um programa Java como entrada?). Parece que uma estratégia de recuperação de erros tem que ser um compromisso cuidadosamente considerado levando em conta os tipos de erros que são mais propensos a ocorrer e razoáveis de processar.

Como mencionamos, alguns compiladores tentavam reparar os erros, num processo em que tentam adivinhar o que o programador queria escrever. Exceto, possivelmente, num ambiente de pequenos programas escritos por estudantes principiantes, a reparação extensiva de erros não é propensa a pagar o seu custo. De fato, com a ênfase crescente na computação interativa e bons ambientes de programação, a tendência está na direção de mecanismos simples de recuperação de erros.

5.5 Estratégia de Recuperação de Erros

Existem muitas estratégias gerais diferentes que um analisador sintático pode empregar para se recuperar de um erro sintático. Apesar de nenhuma delas ter provado ser universalmente aceitável, uns poucos métodos têm ampla aplicabilidade. Introduzimos aqui as seguintes estratégias:

- modalidade do desespero
- nível de frase
- produções de erro
- correção global

Recuperação na modalidade do desespero. Este é o método mais simples de implementar e pode ser usado pela maioria dos métodos de análise sintática. Ao descobrir um erro, o analisador sintático descarta símbolos de entrada, um de cada vez, até que seja encontrado um *token* pertencente a um conjunto designado de *tokens* de sincronização. Os *tokens* de sincronização são usualmente delimitadores, tais como o ponto-e-vírgula ou o **fim**, cujo papel no programa-fonte seja claro. Naturalmente, o projetista do compilador precisa selecionar os *tokens* de sincronização apropriados à linguagem-fonte. A correção na modalidade do desespero, que frequentemente pula uma parte considerável da entrada sem verificá-la, procurando por erros adicionais, possui a vantagem da simplicidade e, diferentemente dos outros métodos a serem enfocados adiante, tem a garantia de não entrar num laço infinito. Nas situações em que os erros múltiplos num mesmo enunciado sejam raros, esse método pode ser razoavelmente adequado.

Recuperação de frases. Ao descobrir um erro, o analisador sintático pode realizar uma correção local na entrada restante. Isto é, pode substituir um prefixo da entrada remanescente por alguma cadeia que permita ao analisador seguir em frente. Correções locais típicas seriam substituir uma vírgula por ponto-e-vírgula, remover um ponto-e-vírgula estranho ou inserir um ausente. A escolha da correção local é deixada para o projetista do compilador. Naturalmente devemos ser cuidadosos, escolhendo substituições que não levem a laços infinitos, como seria o caso, por exemplo, se inseríssemos para sempre na entrada algo à frente do seu símbolo corrente. Esse tipo de substituição pode corrigir qualquer cadeia e foi usado em vários compiladores de correção de erros. O método foi primeiramente usado na análise sintática descendente. Sua maior desvantagem está na dificuldade que tem ao lidar com situações nas quais o erro efetivo ocorreu antes do ponto de detecção.

Regras de Produções para erro. Se tivéssemos uma boa ideia dos erros comuns que poderiam ser encontrados, poderíamos aumentar a gramática para a linguagem em exame com as produções que gerassem construções ilegais. Usamos, então, a gramática aumentada com essas produções de erro para construir um analisador sintático. Se uma produção de erro for usada pelo analisador, podemos gerar diagnósticos apropriados para indicar a construção ilegal que foi reconhecida na entrada.

Correção global. Idealmente, gostaríamos que um compilador fizesse tão poucas mudanças quanto possível, ao processar uma cadeia de entrada ilegal. Existem algoritmos para escolher uma sequência mínima de mudanças de forma a se obter uma correção global de menor custo. Dadas uma cadeia de entrada incorreta x e uma gramática G, esses algoritmos irão encontrar uma árvore gramatical para uma cadeia relacionada y, de tal forma que as inserções, remoções e mudanças de *tokens* requeridas para transformar x em y sejam tão pequenas quanto possível. Infelizmente, esses métodos são em geral muito custosos de implementar, em termos de

tempo e espaço e, então, essas técnicas são correntemente apenas de interesse teórico. Devemos assinalar que o programa correto mais próximo pode não ser aquele que o programador tinha em mente. Apesar de tudo, a noção de correção de custo mínimo fornece um padrão de longo alcance para avaliar as técnicas de recuperação de erros e foi usada para encontrar cadeias ótimas de substituição para a recuperação em nível de frase.

É evidente que com a tendência de uso de computadores pessoais interligados em rede, em substituição ao antigos computadores de grande porte, tem-se que a compilação passou a ser uma tarefa de característica interativa. Assim, a tendência concentra-se no projeto de compiladores que interrompem a compilação a cada erro encontrado, restringindo a recuperação simplesmente a emissão de uma mensagem compreensível do erro detectado.

5.6 Análise Sintática no CSD

Para fornecer um guia na implementação do analisador sintático do **CSD**, tem-se a seguir os algoritmos levando em consideração as características de sintaxe da Linguagem **LPD**. Considera-se que sempre que encontrar um ERRO tem-se o processo de compilação interrompido. No algoritmo a seguir, existem exemplos não exaustivos de trechos do analisador semântico (em azul) e da geração de código para rótulos (em vermelho).

```
Algoritmo Analisador Sintático <programa>
Def rotulo inteiro
início
  rotulo:= 1
  Léxico(token)
  se token.simbolo = sprograma
  então início
            Léxico(token)
            se token.simbolo = sidentificador
            então início
                     insere_tabela(token.lexema,"nomedeprograma","","")
                      Léxico(token)
                      se token.simbolo = spontovirgula
                      então início
                                analisa_bloco
                                se token.simbolo = sponto
                                então se acabou arquivo ou é comentário
                                          então sucesso
                                          senão ERRO
                                senão ERRO
                               fim
                       senão ERRO
                     fim
            senão ERRO
          fim
  senão ERRO
fim.

Algoritmo Analisa_Bloco <bloco>
início
  Léxico(token)
  Analisa_et_variáveis
  Analisa_subrotinas
  Analisa_comandos
fim
```

```
Algoritmo Analisa_et_variáveis  <etapa de declaração de variáveis>
início
 se token.simbolo = svar
 então início
          Léxico(token)
          se token.símbolo = sidentificador
          então enquanto(token.símbolo = sidentificador)
                  faça início
                          Analisa_Variáveis
                          se token.símbolo = spontvirg
                          então Léxico (token)
                          senão ERRO
                         fim
           senão ERRO
fim

Algoritmo Analisa_Variáveis <declaração de variáveis>
início
   repita
       se token.símbolo = sidentificador
        então início
                Pesquisa_duplicvar_ tabela(token.lexema)
                se não encontrou duplicidade
                então início
                        insere_tabela(token.lexema, “variável” ,””,””)
                          Léxico(token)
                          se (token.símbolo = Svírgula) ou
                             (token.símbolo = Sdoispontos)
                          então início
                                  se token.símbolo = Svírgula
                                   então início
                                           léxico(token)
                                           se token.simbolo = Sdoispontos
                                           então ERRO
                                          fim
                                 fim
                          senão ERRO
                       fim
                senão ERRO
          senão ERRO
      até que (token.símbolo = sdoispontos)
      Léxico(token)
      Analisa_Tipo
fim


Algoritmo Analisa_Tipo    <tipo>
início
  se (token.símbolo ≠ sinteiro e token.símbolo ≠  sbooleano))
  então ERRO
  senão coloca_tipo_tabela(token.lexema)
  Léxico(token)
fim

Algoritmo Analisa_comandos  <comandos>
início
  se token.simbolo = sinicio
  então início
            Léxico(token)
            Analisa_comando_simples
            enquanto (token.simbolo ≠ sfim)
```

```
            faça início
                    se token.simbolo = spontovirgula
                    então início
                              Léxico(token)
                              se token.simbolo ≠ sfim
                              então Analisa_comando_simples
                            fim
                     senão ERRO
                   fim
            Léxico(token)
          fim
  senão ERRO
fim

Analisa_comando_simples  <comando>
início
  se token.simbolo = sidentificador
  então Analisa_atrib_chprocedimento
  senão
  se token.simbolo = sse
  então Analisa_se
  senão
  se token.simbolo = senquanto
  então Analisa_enquanto
  senão
  se token.simbolo = sleia
  então Analisa_leia
  senão
  se token.simbolo = sescreva
  então Analisa_ escreva
  senão
 Analisa_comandos
fim

Algoritmo Analisa_atrib_chprocedimento <atribuição_chprocedimento>
início
  Léxico(token)
  se token.simbolo = satribuição
  então  Analisa_atribuicao
  senão  Chamada_procedimento
fim

Algoritmo Analisa_leia  <comando leitura>
início
  Léxico(token)
  se token.simbolo = sabre_parenteses
  então início
            Léxico(token)
            se token.simbolo = sidentificador
            então se pesquisa_declvar_tabela(token.lexema)
                    então início (pesquisa em toda a tabela)
                              Léxico(token)
                              se token.simbolo = sfecha_parenteses
                              então Léxico(token)
                              senão ERRO
                            fim
                     senão ERRO
            senão ERRO
          fim
   senão ERRO
fim
```

```
Algoritmo Analisa_escreva  <comando escrita>
início
   Léxico(token)
   se token.simbolo = sabre_parenteses
   então início
              Léxico(token)
              se token.simbolo = sidentificador
              então se pesquisa_ declvarfunc_tabela(token.lexema)
                        então início
                                  Léxico(token)
                                  se token.simbolo = sfecha_parenteses
                                  então Léxico(token)
                                  senão ERRO
                              fim
                         senão ERRO
              senão ERRO
             fim
    senão ERRO
fim

Algoritmo Analisa_enquanto  <comando repetição>
Def auxrot1,auxrot2 inteiro
início
   auxrot1:= rotulo
   Gera(rotulo,NULL,´          ´,´          ´)     {início do while}
    rotulo:= rotulo+1
   Léxico(token)
   Analisa_expressão
   se token.simbolo = sfaça
   então início
              auxrot2:= rotulo
              Gera(´          ´,JMPF,rotulo,´          ´)        {salta se falso}
              rotulo:= rotulo+1
              Léxico(token)
              Analisa_comando_simples
              Gera(´          ´,JMP,auxrot1,´          ´)   {retorna início loop}
              Gera(auxrot2,NULL,´          ´,´          ´)   {fim do while}
            fim
   senão ERRO
fim


Algoritmo Analisa_se  <comando condicional>
início
   Léxico(token)
   Analisa_expressão
   se token.símbolo = sentão
   então início
              Léxico(token)
              Analisa_comando_simples
              se token.símbolo = Ssenão
              então início
                        Léxico(token)
                        Analisa_comando_simples
                       fim
            fim
    senão ERRO
fim
```

```
Algoritmo Analisa_Subrotinas <etapa de declaração de sub-rotinas>
Def. auxrot, flag inteiro
Início
  flag = 0
  if (token.simbolo = sprocedimento) ou
            (token.simbolo = sfunção)
  então início
        auxrot:= rotulo
        GERA(´        ´,JMP,rotulo,´          ´)      {Salta sub-rotinas}
        rotulo:= rotulo + 1
        flag = 1
      fim
  enquanto (token.simbolo = sprocedimento) ou
            (token.simbolo = sfunção)
   faça início
        se (token.simbolo = sprocedimento)
        então analisa_declaração_procedimento
        senão analisa_ declaração_função
        se token.símbolo = sponto-vírgula
        então Léxico(token)
        senão ERRO
      fim
    if flag = 1
    então Gera(auxrot,NULL,´          ´,´          ´)    {início do principal}
fim

Algoritmo Analisa_ declaração_procedimento <declaração de procedimento>
início
  Léxico(token)
  nível := "L" (marca ou novo galho)
  se token.símbolo = sidentificador
  então início
        pesquisa_declproc_tabela(token.lexema)
        se não encontrou
        então início
              Insere_tabela(token.lexema,"procedimento",nível, rótulo)
                                    {guarda na TabSimb}
              Gera(rotulo,NULL,´      ´,´        ´)
                    {CALL irá buscar este rótulo na TabSimb}
               rotulo:= rotulo+1
              Léxico(token)
              se token.simbolo = sponto_vírgula
              então Analisa_bloco
              senão ERRO
            fim
        senão ERRO
      fim
  senão ERRO
  DESEMPILHA OU VOLTA NÍVEL
fim
```

```
Algoritmo Analisa_ declaração_função <declaração de função>
início
  Léxico(token)
  nível := “L” (marca ou novo galho)
  se token.símbolo = sidentificador
  então início
          pesquisa_declfunc_tabela(token.lexema)
          se não encontrou
          então início
                  Insere_tabela(token.lexema,””,nível,rótulo)
                  Léxico(token)
                  se token.símbolo = sdoispontos
                  então início
                          Léxico(token)
                          se (token.símbolo = Sinteiro) ou
                              (token.símbolo = Sbooleano)
                          então início
                                  se (token.símbolo = Sinteger)
                                  então TABSIMB[pc].tipo:=
                                                  “função inteiro”
                                  senão TABSIMB[pc].tipo:=
                                                  “função booleana”
                                  Léxico(token)
                                 se token.símbolo = sponto_vírgula
                                 então Analisa_bloco
                                fim
                           senão ERRO
                          fim
                   senão ERRO
                 fim
          senão ERRO
        fim
  senão ERRO
  DESEMPILHA OU VOLTA NÍVEL
Fim

Algoritmo Analisa_expressão  <expressão>
início
  Analisa_expressão_simples
  se (token.simbolo = (smaior ou smaiorig ou sig
                       ou smenor ou smenorig ou sdif))
  então inicio
          Léxico(token)
          Analisa_expressão_simples
         fim
fim

Algoritmo Analisa_expressão_simples <expressão simples>
início
  se (token.simbolo = smais) ou (token.simbolo = smenos)
  então Léxico(token)
  Analisa_termo
  enquanto ((token.simbolo = smais) ou (token.simbolo = smenos) ou
              (token.simbolo = sou))
  faça inicio
          Léxico(token)
          Analisa_termo
        fim
fim
```

```
Algoritmo Analisa_termo   <termo>
início
  Analisa_fator
  enquanto ((token.simbolo = smult) ou (token.simbolo = sdiv) ou
              (token.simbolo = se))
  então início
          Léxico(token)
          Analisa_fator
        fim
fim


Algoritmo Analisa_fator <fator>
Início
  Se token.simbolo = sidentificador              (* Variável ou Função*)
  Então inicio
          Se pesquisa_tabela(token.lexema,nível,ind)
          Então Se (TabSimb[ind].tipo = "função inteiro") ou
                   (TabSimb[ind].tipo = "função booleano")
                Então Analisa_chamada_função
                Senão Léxico(token)
          Senão ERRO
        Fim
   Senão Se (token.simbolo = snumero)          (*Número*)
          Então Léxico(token)
          Senão Se token.símbolo = snao            (*NAO*)
                  Então início
                          Léxico(token)
                          Analisa_fator
                        Fim
                  Senão Se token.simbolo = sabre_parenteses
                                          (* expressão entre parenteses *)
                        Então início
                                Léxico(token)
                                Analisa_expressão(token)
                                Se token.simbolo = sfecha_parenteses
                                 Então Léxico(token)
                                 Senão ERRO
                              Fim
                        Senão Se (token.lexema = verdadeiro) ou
                                    (token.lexema = falso)
                               Então Léxico(token)
                               Senão ERRO
Fim
```

Exercício:

Elaborar os algoritmos para os seguinte procedimentos:

Analisa Chamada de Procedimento
Analisa Chamada de Função

6 Analisador Semântico

Até agora a preocupação foi em analisar sintaticamente os programas-fonte de acordo com as suas respectivas gramáticas, considerando que um programa-fonte fosse simplesmente uma sequência de caracteres. Entretanto, para gerar corretamente os códigos-objeto de um programa-fonte, um compilador deve ser capaz de reconhecer os tipos de símbolos (por exemplo, se eles são do tipo inteiro, do tipo ponto flutuante ou de um novo tipo construído) e a consistência do seu uso (por exemplo, o operador aritmético / em C ou o operador %).

Os programas em linguagem de alto nível contém normalmente uma seção de declaração de dados (identificadores) e uma de comandos e expressões propriamente ditos. A consistência do uso de um identificador na seção de comandos depende da sua declaração (explícita ou implícita). Este tipo de dependência só pode ser satisfatoriamente contemplado por uma gramática sensitiva ao contexto. Entretanto, existem poucos resultados práticos que formalizam e processam, de forma satisfatória, as gramáticas sensitivas.

6.1 Gramática com Atributos

Uma solução prática comumente adotada para a especificação da semântica de uma linguagem consiste em atribuir às produções da sua gramática as ações semânticas, que "qualificam semanticamente" os símbolos que aparecem na árvore sintática de uma sentença. Essa gramática estendida é conhecida como gramática com atributos. A principal função de uma gramática com atributos é incluir restrições semânticas às construções sintaticamente corretas.

As ações semânticas estabelecem a dependência semântica entre os símbolos de uma gramática. Através delas, pode-se propagar os valores conhecidos de atributos pela árvore sintática e determinar os desconhecidos. Com uma árvore sintática com os atributos, é possível controlar em muitas situações a consistência semântica.

Um analisador descendente é apropriado para incluir este mecanismo de atribuição de valores para os identificadores. Normalmente, os valores dos atributos de cada símbolo são identificados e armazenados numa tabela de símbolos durante o processo de análise sintática. A partir daí o analisador semântico usa as informações existentes na tabela para validar os tipos de dados. Em muitos compiladores, o analisador semântico é considerado como parte do analisador sintático, porque a construção da árvore sintática e a inferência dos valores dos atributos de seus símbolos ocorrem paralelamente.

6.2 Tabela de Símbolos

Como visto anteriormente, a tabela de símbolos guarda a definição dos tipos de dados construídos pelos usuários. Em linguagens que suportam o uso de um mesmo nome simbólico em diferentes níveis de blocos de programas, os compiladores usam a tabela de símbolos para distinguir o **escopo de validade** de cada símbolo. Outra utilidade da tabela de símbolos durante o processo de análise semântica e reconhecer o **escopo** para o qual um símbolo é ativado. Em programas escritos em linguagem estruturada, como C, o conceito de **escopo de validade** ou **nível** léxico é importante, porque para estas linguagens um nome não precisa ter uma entrada única na tabela de símbolos. É admissível o uso de um mesmo nome em níveis distintos com distintas "interpretações semânticas".

Os outros atributos associados a um símbolo dependem da categoria dele. Por exemplo, ao nome de um procedimento deve-se especificar ainda o número de parâmetros

formais, tipo, mecanismo de passagem de cada parâmetro e, dependendo da linguagem, o tipo de dado retornado; e ao nome de uma variável precisa-se acrescentar o seu tipo de dado. Devido a diversidade do número de atributos associados a cada símbolo, é comum utilizar em implementações o mecanismo de referência para atributos cuja quantidade e cujo comprimento são variáveis.

A informação sobre a categoria de símbolos na tabela é usada pelo analisador semântico para, por exemplo:

- evitar que o nome de uma variável seja utilizado no contexto do nome de uma sub-rotina a ser chamada;
- evitar que o nome de um procedimento apareça no lado esquerdo do operador de atribuição ou uma função numa expressão sem a sua lista de argumentos;
- otimizar o uso de memória atribuindo a todas as constantes idênticas um mesmo endereço.

A tabela de símbolos provém ainda um campo para o atributo <endereço> de cada símbolo. Este endereço pode ser textual ou pode ser uma posição de memória, dependendo dos códigos intermediários a serem gerados

Linguagens que suportam a construção de novos tipos de dados, como C, requerem que os compiladores armazenem na tabela de símbolos os nomes dos novos tipos construídos e os seus componentes. Com isso, o analisador semântico pode validar facilmente as referências aos campos de uma variável do tipo construído. A medida que o analisador sintático reconhece um símbolo, o analisador semântico verifica se o símbolo já está na **tabela de símbolos.** Se não estiver, ele é armazenado. Caso contrário, o analisador semântico procura verificar a compatibilidade entre os atributos do símbolo inserido e a forma corrente do seu uso. A complexidade de pesquisa na tabela de símbolos é um fator importante para analisar o desempenho de um compilador.

6.3 Erros Semânticos no CSD

A análise semântica no compilador **CSD** será composta basicamente das seguintes tarefas:

1) **Verificação da ocorrência da duplicidade na declaração de um identificador** (nome do programa, procedimento, função ou variável). Sempre que for detectado um novo identificador, deve ser feita uma busca para verificar as seguintes possibilidades:

   a) se ele for uma variável verificar se já não existe outro identificador (variável ou nome da subrotina onde está inserida) visível[1] de mesmo nome no mesmo nível de declaração.

   b) Se for o nome de um procedimento ou função verificar se já não existe um outro identificador visível de qualquer tipo em nível igual ao inferior ao agora analisado.

2) **Verificação de uso de identificadores não declarados**. Sempre que for detectado um identificador, verificar se ele foi declarado (está visível na tabela de símbolos) e é compatível com o uso (exemplo: variável usada que existe como nome de programa ou de procedimento na tabela de símbolos deve dar erro).

[1] Por visível considera-se como aquele que pode ser encontrado na Tabela de Símbolos atual.

3) **Verificação de compatibilidade de tipos**. Sempre que ocorrer um comando de atribuição, verificar se a expressão tem o mesmo tipo da variável ou função que a recebe.

4) Verificação dos comandos escreva e leia (variável inteira).

5) Verificação de chamadas de procedimento e função.

6) **Verificação dos operadores unários** – **,** + **,** nao.

É fácil perceber que as chamadas para o analisador semântico não passam de linhas de comandos a serem inseridos no "corpo" do analisador sintático, nos locais apropriados.

Vale lembrar que a Linguagem **LPD** não permite a passagem de parâmetros nos procedimentos e funções. Caso isso fosse permitido, então deveríamos também verificar a consistência no número de argumentos e parâmetros, bem como sua compatibilidade de tipos.

7 Máquina Virtual e Geração de Código

Esta seção foi retirada do livro do Kowaltowski (vide bibliografia), o qual, embora com edição esgotada, apresenta uma arquitetura para Máquina Virtual bastante simples na compreensão e implementação. As únicas diferenças residem na nomenclatura adotada.

O nosso objetivo final é construir um compilador para LPD, isto é, construir um programa que traduz um programa-fonte em LPD para um programa-objeto em linguagem de máquina de um dado computador. Antes de empreender a tarefa de escrever um compilador, devemos estabelecer precisamente a tradução que corresponde a cada construção de LPD. Entretanto, traduzir para a linguagem de máquina de um computador real é, em geral, uma tarefa muito trabalhosa. As linguagens de máquina têm muitas características com as quais teríamos que nos preocupar, perdendo de vista a correspondência entre as construções do programa-fonte e a sua tradução. Ao invés de traduzir, então, os programas em LPD para a linguagem de máquina de um computador real, definiremos uma máquina hipotética, mais conveniente para nossas finalidades. Esta abordagem é muito comum na implementação de linguagens de programação.

Os próximos tópicos são dedicados, então, ao desenvolvimento de um computador hipotético que chamaremos de MVD (Máquina Virtual Didática). A mesma sigla será usada para denotar a linguagem de montagem desta máquina. O desenvolvimento será gradual, acompanhando a discussão de vários mecanismos de LPD. Frequentemente, seremos obrigados a rever as decisões já tomadas sobre o funcionamento da MVD, a fim de acomodar um mecanismo, Desta maneira ficarão mais claras as razões para as várias características da MVD, e a sua relação com as construções de LPD.

7.1 Características Gerais da MVD

Descreveremos nesta seção a estrutura básica da MVD, que é bastante simples. Como veremos nas seções subsequentes, é o repertório de instruções que é complexo.

Uma decisão importante que tomaremos agora é que a MVD deverá ser uma máquina a pilha. Isto é muito natural, pois a LPD permite o uso de procedimentos recursivos, e a recursão está intimamente ligada com o uso da pilha. As razões para esta decisão ficarão mais claras, portanto, quando tratarmos de procedimentos recursivos.

A nossa máquina terá uma memória composta de duas regiões:

- A *região de programa P* que conterá as instruções da MVD. O formato exato de cada instrução é irrelevante para a nossa discussão.
- A *região da pilha de dados M* que conterá os valores manipulados pelas instruções da MVD. Suporemos que esta região compõe-se de palavras que podem conter valores inteiros ou então indefinidos.

Suporemos que cada uma das duas regiões têm palavras numeradas com 0, 1, 2,..., e não nos preocuparemos com as limitações de tamanho de cada região, nem de cada palavra.

A MVD terá dois registradores especiais que serão usados para descrever o efeito das instruções:

- O registrador do programa *i* conterá o endereço da próxima instrução a ser executada, que será, portanto, *P [i]*.
- O registrador *s* indicará o elemento no topo da pilha cujo valor será dado, portanto, por *M [s]*.

Uma vez que o programa da MVD está carregado na região *P*, e os registradores têm os seus valores iniciais, o funcionamento da máquina é muito simples. As instruções indicadas pelo registrador *i* são executadas até que seja encontrada a instrução de parada, ou ocorra algum erro. A execução de cada instrução incrementa de um o valor de *i*, exceto as instruções que envolvem desvios.

Passaremos a desenvolver nas seções seguintes o repertório de instruções da MVD motivado pelas várias construções da LPD.

7.2 Avaliação de Expressões

Suponhamos que *E* é uma expressão em LPD da forma *E = E1 ∇ E2*, onde *E1* e*E2* são duas expressões mais simples, e ∇ é um operador binário como +, -, *, <, =, etc. A expressão *E* deve ser avaliada calculando-se em primeiro lugar os valores de *E1* e *E2* e aplicando-se, em seguida, a operação correspondente a ∇. Este cálculo pode ser implementado de várias maneiras, guardando-se os valores intermediários de *E1* e *E2* em localizações de memória especiais. Numa máquina como a MVD, uma maneira conveniente é guardar estes valores intermediários na própria pilha *M*. Assim, supondo que os valores *v1 e v2* das expressões *E1* e *E2* são calculados de maneira semelhante, a pilha terá as configurações sucessivas indicadas na Figura 8.1. Denotados por *m* o endereço do topo da pilha antes de iniciar a avaliação da expressão *E*, e por *v*, o valor final calculado. O símbolo ? Denota valores irrelevantes que já estavam na pilha.

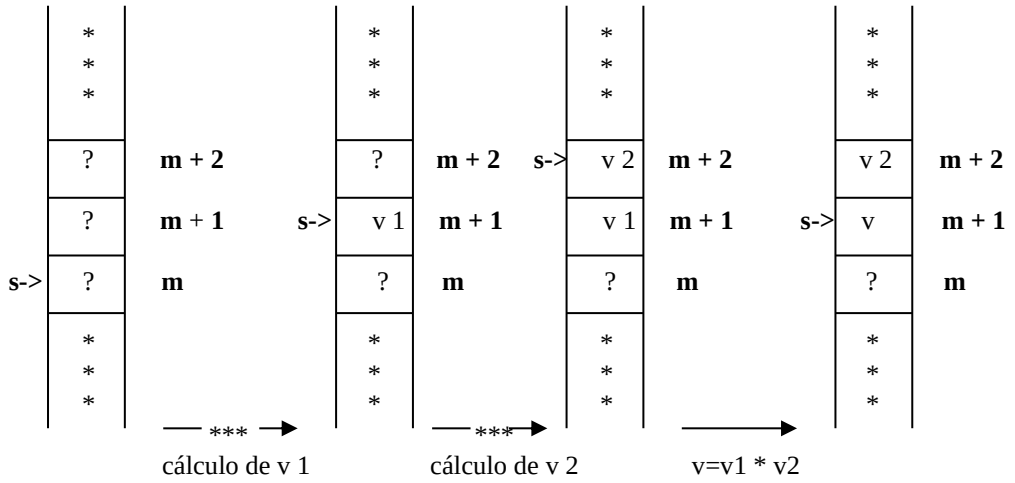


Figura 8.1

Note-se que estamos resolvendo o problema de avaliar expressões de maneira indutiva, supondo em primeiro lugar que sabemos resolvê-lo para expressões mais simples, isto é, mais curtas, como *E1* e *E2*. Assim, supusemos na Figura 8.1 que os valores *v1* e *v2* são calculados possivelmente em vários passos, ficando no fim o valor correspondente no topo da pilha, uma posição acima da posição inicial. A base desta indução corresponde às expressões o mais simples possível, isto é, às constantes e variáveis. Para estas, basta colocar os seus valores respectivos no topo da pilha. Os operadores unários serão tratados de maneira análoga. O caso de chamadas de funções que devolvem resultados será tratado mais adiante, mas é importante notar desde já que qualquer que seja a implementação destas chamadas, o seu efeito deverá ser sempre o de deixar o resultado final no topo da pilha, uma posição acima da inicial.

Podemos concluir da discussão acima que a MVD deve possuir instruções que carregam na pilha valores de constantes e de variáveis, e outras que executam operações correspondentes aos operadores da LPD. Definiremos, portanto, uma série de instruções para

a MVD, mas devemos notar que algumas dessas definições são provisórias, e serão modificadas mais adiante. O efeito de cada instrução está descrito numa notação semelhante à da LPD, indicando as modificações no estado dos registradores e da memória da MVD. Omitimos nesta descrição a operação ***i:=i+1*** que está implícita em todas as instruções, exceto quando há desvio. Adotaremos, também, a convenção de representar os valores booleanos por inteiros: *verdadeiro* por *1* e *falso* por *0.*

LDC *k* (Carregar constante):
S:=s + 1 ; M [s]: = *k*
LDV *n* (Carregar valor):
S:=s+1 ; M[s]:=M[n]
ADD (Somar):
M[s-1]:=M[s-1]+M[s]; s:=s-1
SUB (Subtrair):
M[s-1]:=M[s-1]-M[s]; s:=s-1
MULT (Multiplicar):
M[s-1]:=M[s-1]*M[s]; s:=s-1
DIVI (Dividir):
M[s-1]:=M[s-1] div M[s]; s:=s-1
INV (Inverter sinal):
M[s]:=-M[s]
AND (Conjunção):
Se M [s-1]=1 **e M**[s]=1 **então M**[s-1]:=1 **senão M**[s-1]:=0; S:=s-1
OR (Disjunção):
Se M[s-1]=1 **ou M**[s]=1 **então M**[s-1]:=1 **senão M**[s-1]:=0; s:=s-1
NEG (Negação):
M[s]:=1-M[s]
CME (Comparar menor):
Se M[s-1]<M[s] **então M**[s-1]:=1 **senão M**[s-1]:=0; s:=s-1
CMA (Comparar maior):
Se M[s-1] >M[s] **então M**[s-1]:=1 **senão M**[s-1]:=0;s:=s-1
CEQ (comparar igual):
Se M[s-1]=M[s] **então M**[s-1]:=1 **senão M**[s-1]:=0;s:=s-1
CDIF (Comparar desigual):
Se M[s-1] ≠ M[s] **então M**[s-1]:=1 **senão M**[s-1]:=0; s:=s-1
CMEQ (Comparar menor ou igual)
Se M[s-1] ≤ M[s] **então M**[s-1]:=1 **senão M**[s-1]:=0;s:=s-1
CMAQ (Comparar maior ou igual):
Se M[s-1] ≥ M[s] **então M**[s-1]:=1 **senão M**[s-1]:=0; s:=s-1

Exemplo:

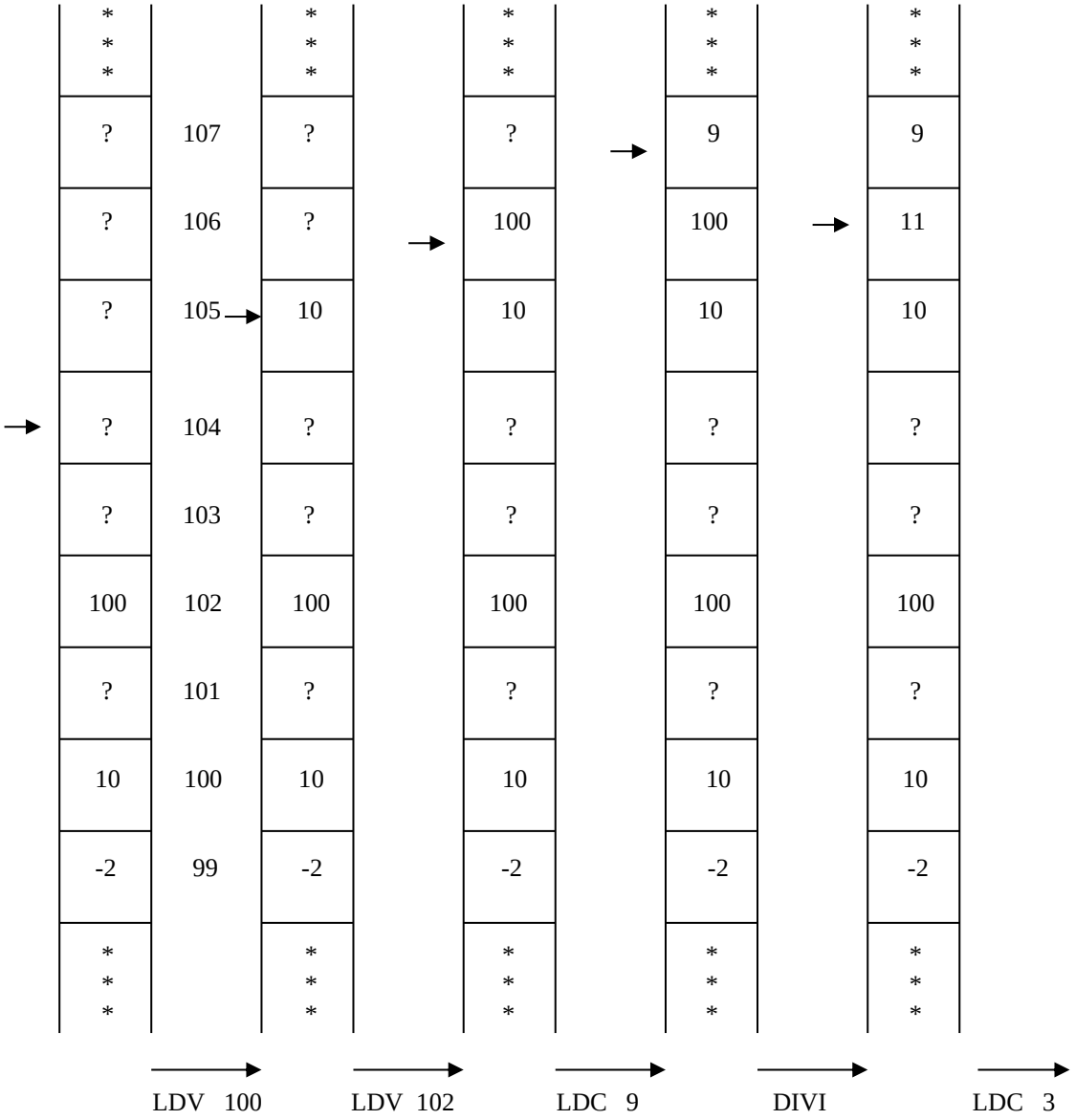
Consideremos a expressão a + (b **div** 9 - 3) * c, e suponhamos que os endereços atribuídos pelo compilador às variáveis a, b e c são, respectivamente, 100, 102, e 99. Então o trecho do programa-objeto correspondente à tradução desta expressão seria:

```
LDV   100
LDV   102
LDC   9
DIVI
LDC   3
```

```
SUB
LDV    99
MULT
ADD
```

Suponhamos que os valores armazenados nas posições 99, 100 e 102 da pilha são –2, 10 e 100, respectivamente, e que o registrador s contém o valor 104. As configurações sucessivas da pilha ao executar as instruções acima são dadas na Figura 8.2. Os valores sucessivos de s estão indicados pelas flechas.

Uma observação interessante é que o código da MVD gerado para expressões está diretamente ligado com a notação pós-fixa, que no caso da expressão do exemplo acima seria ab9 **div** 3-c*+. Esta sequência de símbolos deve ser comparada com a sequência de instruções da MVD desse exemplo.



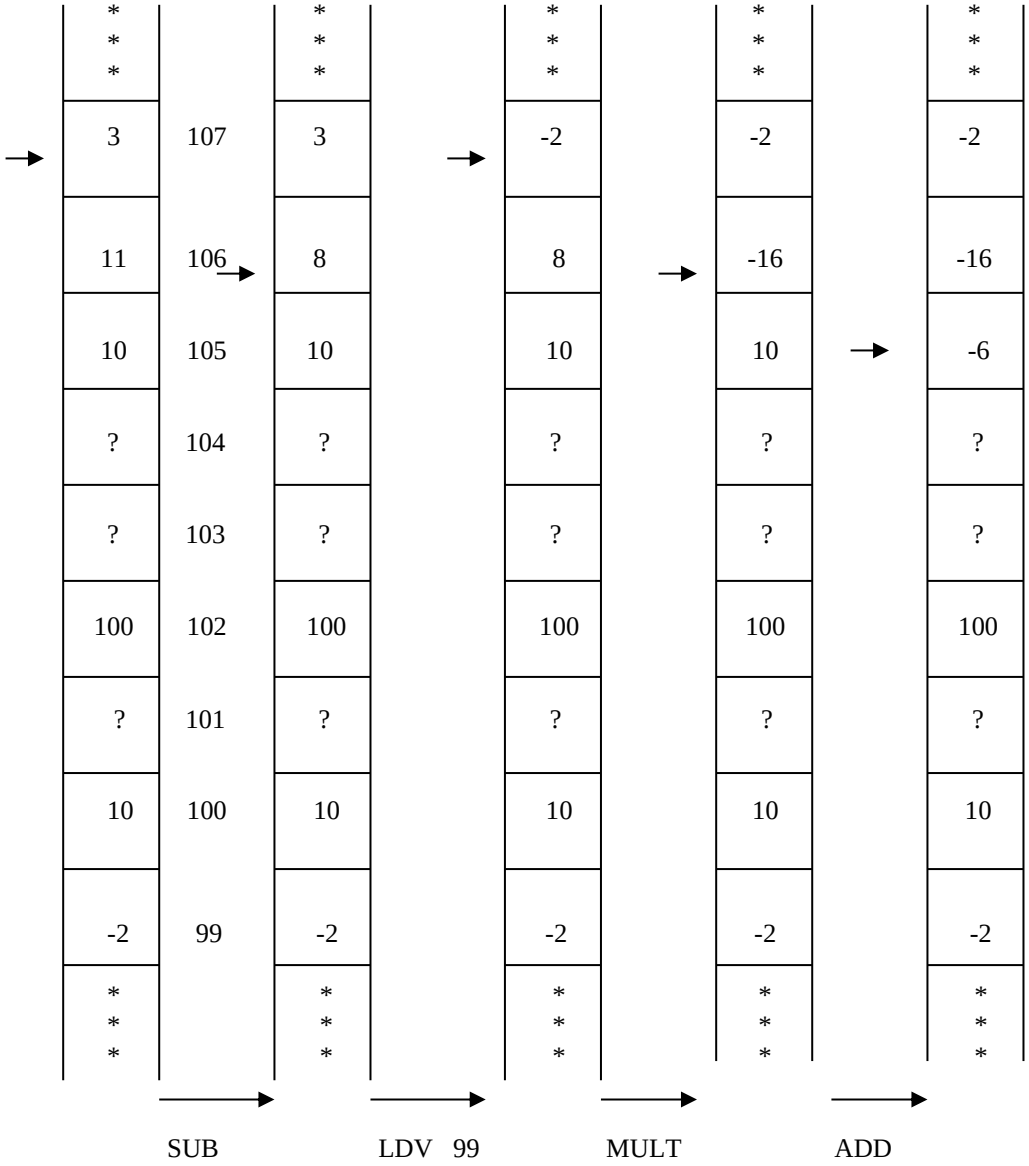


Figura 8.2

7.3 Comandos de Atribuição

Consideramos, por enquanto, apenas as atribuições a variáveis simples da forma V:=E, onde V é o nome de uma variável e E é uma expressão que já sabemos traduzir. Uma maneira simples de implementar este comando é dada por uma instrução de armazenamento:

STR *n* (Armazenar valor):

M[*n*]:=M[s]; s:=s-1

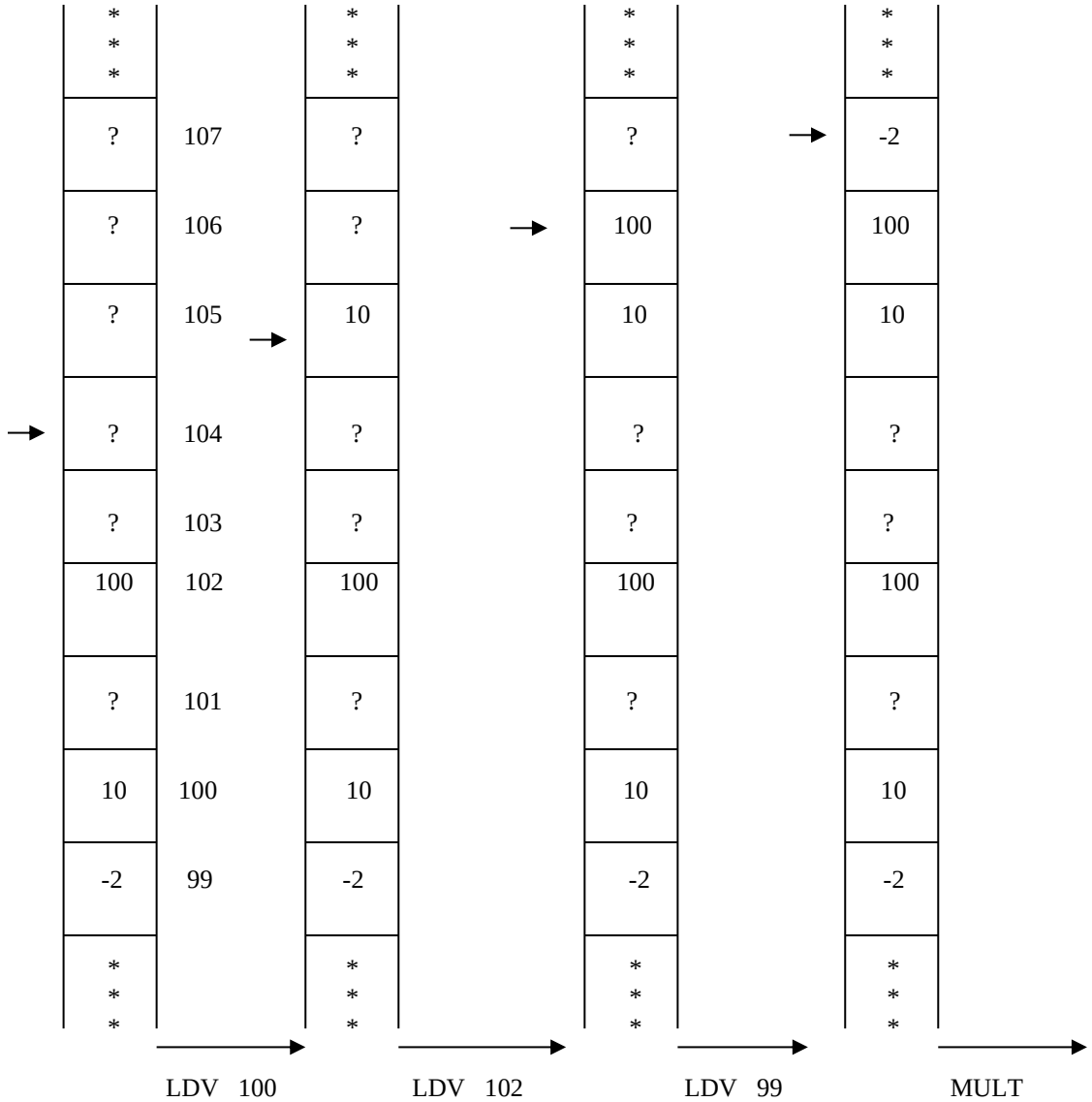
Esta instrução seguirá o código-objeto que corresponde à expressão E, e *n* será o endereço atribuído pelo compilador à variável V.

Exemplo: Consideremos o comando a:=a+b*c e suponhamos que os endereços e os valores destas variáveis são os mesmos do exemplo anterior. O código da MVD para este comando será:

```
LDV    100
LDV    102
LDV     99
MULT
ADD
STR    100
```

A Figura 8.3 apresenta as configurações sucessivas da pilha ao ser executado este código-objeto.

Note-se que, devido à maneira como definimos a instrução STR, o valor final do registrador s, após a execução do código-objeto correspondente a um comando de atribuição, será igual ao seu valor inicial. Esta é uma propriedade importante que será verdadeira para qualquer comando em LPD, como verificaremos mais tarde. As únicas exceções são os comandos de desvio, e de chamada de procedimentos e funções quando estes não retornam por causa de desvios.



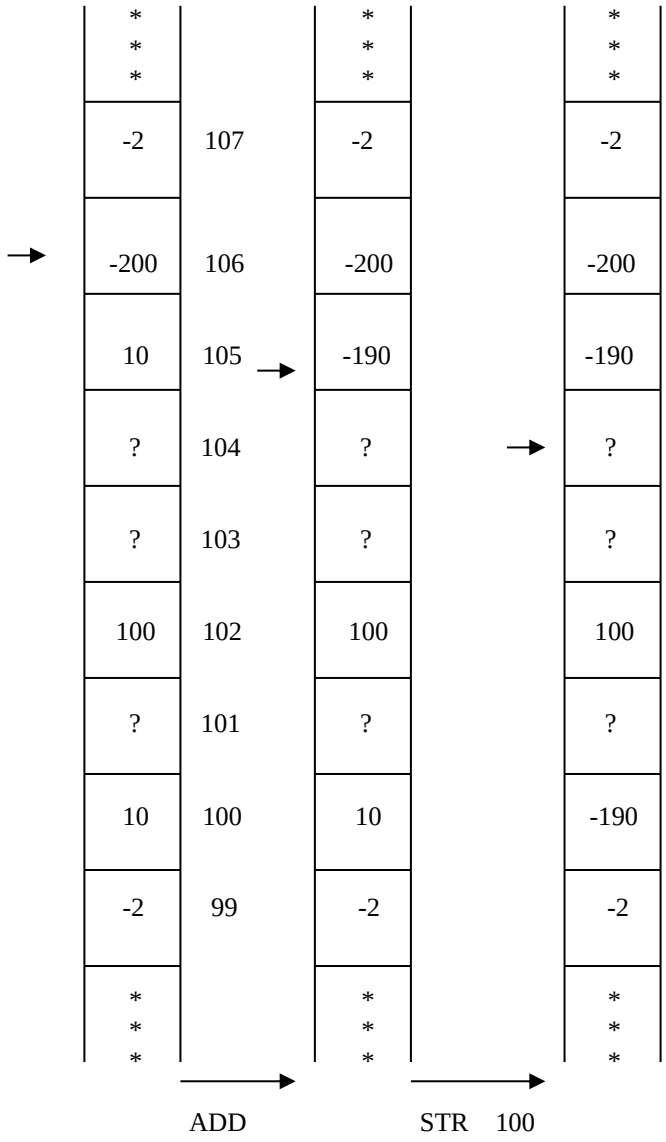


Figura 8.3

7.4 Comandos Condicionais e Iterativos

Para implementar comandos condicionais definiremos duas instruções de desvio para a MVD:

JMP p (Desviar sempre):
i:=p
JMPF p (Desviar se falso):
Se M[s]=0 **então** i:=p **senão** i:=i+1;
S:=s-1

Nestas instruções, p é um número inteiro que indica um endereço de programa da MVD. Nos exemplos que se seguem utilizaremos rótulos simbólicos no lugar de números. Note-se que, haja ou não desvio, a instrução JMPF elimina o valor que foi testado, e que está no topo da pilha.

Introduziremos, por conveniência, mais uma instrução que não é estritamente necessária, mas que simplifica o processo de tradução e que não tem nenhum efeito sobre a execução:

NULL (Nada): não faz nada.

Um comando condicional da forma **se *E* entao *C* 1 senao *C* 2,** onde *E* é uma expressão e *C* 1 e *C*2 são comandos, será traduzido por:

```
*
*                   tradução de E
*
JMPF  l 1
*
*                   tradução de C 1
*
JMP    l 2
L 1 NULL
*
*                   tradução de C 2
*
l 2 NULL
```

Caso o comando tenha a forma **se** *E* **entao** C, podemos traduzi-lo por:

```
*
*                  tradução de E
*
JMPF  l
*
*                  tradução de C
*
l NULL
```

As mesmas instruções de desvio podem ser usadas para implementar comandos repetitivos. No caso do comando **enquanto** E **faca** C teremos a tradução:

```
L 1 NULL
*
*                 tradução de E
*
JMPF  l 2
*
*                 tradução de C
*
JMP    l 1
L 2 NULL
```

É fácil mostrar que estas traduções de comandos condicionais e repetitivos são tais que, ao serem executadas, fazem com que o nível final da pilha seja igual ao inicial.

Exemplo:

Indicaremos a seguir as traduções correspondentes a três comandos em LPD:

1. **se** q **entao** a:=1 **senao** a:=2

```
        LDV    Q
        JMPF   L1
        LDC    1
        STR    A
        JMP    L2
    L1  NULL
        LDC    2
        STR    A
    L2  NULL
```

2. **se** a>b **entao** q:=p **e** q
   senao se a < 2*b **entao** p:=verdadeiro
   senao q:=falso

```
        LDV    A
        LDV    B
        CMA
        JMPF   L3
        LDV    P
        LDV    Q
        AND
        STR    Q
        JMP    L4
    L3  NULL
        LDV    A
        LDC    2
        LDV    B
        MULT
        CME
        JMPF   L5
        LDC    1
        STR    P
        JMP    L6
    L5  NULL
        LDC    0
        STR    Q
    L6  NULL
    L4  NULL
```

3. **enquanto** s <= n **faca** s:=s+3*s

```
L7  NULL
    LDV    S
    LDV    N
    CMEQ
    JMPF   L8
    LDV    S
    LDC    3
    LDV    S
    MULT
    ADD
    STR    S
    JMP    L7
L8  NULL
```

7.5 Comandos de Entrada e de Saída

Um comando da forma leia (v1) deve ler o próximo valor inteiro de entrada e atribuí-lo à variável inteira v1. A fim de implementar o comando de leitura definiremos a seguinte instrução para a MVD.

RD (Leitura):
S:=s+1; M[s]:= "próximo valor de entrada".

Usando esta instrução, podemos traduzir leia(v1) por:

```
RD
STR    V1
```

Onde V1 é o endereço da variável v1.

O comando de saída de LPD tem a forma escreva (v1), indicando que o valor da variável inteira v1 deve ser impresso no arquivo de saída. Definiremos então a instrução:

PRN (Impressão):
"Imprimir M[s]"; s:=s-1

Assim, escreva (v1) pode ser traduzido por:

```
LDV    V1
PRN
```

Exemplo:

1. leia(a)

```
RD
STR    A (endereço de a)
```

2. escreva(x)

```
LDV    X (endereço de x)
PRN
```

7.6 Sub-Programas

A Figura 8.4 mostra um programa muito simples, sem procedimentos. Deveria ser claro que, neste caso, o programa-objeto deveria reservar as cinco posições iniciais da pilha para as variáveis, e em seguida começar a executar as instruções

```
programa  exemplo5;
var  n, k: inteiro;
       f1,  f2,  f3: inteiro;

inicio
    leia(n);
     f1:=0;
     f2:=1;
     k:=1;
     enquanto k<=n
     faca inicio
              f3:=f1+f2;
              f1:=f2; f2:=f3;
              k:=k+1
          fim;
       escreva (n);
       escreva (f1)
   fim.
```

Figura 8.4

Que correspondem ao bloco de comandos. Definiremos então duas instruções para a MVD.

```
START          (Iniciar programa principal):
                S:=-1
ALLOC   m      (Alocar memória*):
                S:=s+m
```

***** mais a frente alteraremos a instrução ALLOC para atender a demanda por variáveis locais e recursividade.

```
DALLOC   m     (liberar memória):
                S:=s-m
```

A instrução **START** será sempre a primeira instrução de um programa-fonte. Uma declaração da forma v1, v2,...vm: tipo será traduzida por **ALLOC** m. Note-se que o compilador pode calcular facilmente os endereços das variáveis v1, v2, ...,vm que deverão ser 0, 1,...,m-1, respectivamente (quando esta é a primeira declaração de variáveis).

Para completar a tradução de programas, definiremos uma instrução de término de execução:

```
HLT     (Parar):
         "Pára a execução da MVD"
```

Exemplo:

Indicamos a seguir o programa-objeto que resulta da tradução do programa da Figura 8.4. Fragmentos do programa-fonte são usados como comentários a fim de tornar mais clara a tradução:

```
        START       programa
        ALLOC  2    var n, k            (0=>n , 1=>k , 2=>f1 , 3=>f2 , 4=>f3)
        ALLOC  3    fl, f2, f3
        RD
        STR    0    leia (n)
        LDC    0
        STR    2    fl:=0
        LDC    1
        STR    3    f2:=1
        LDC    1
        STR    1    k:=1
L1      NULL        enquanto
        LDV    1
        LDV    0
        CMEQ        k<=n
        JMPF   L2   faca
        LDV    2
        LDV    3
        ADD
        STR    4    f3:=f1+f2
        LDV    3
        STR    2    f1:=f2
        LDV    4
        STR    3    f2:=f3
        LDV    1
        LDC    1
        ADD
        STR    1    k:=k+1
        JMP    L1
L2      NULL
        LDV    0
        PRN         escreva (n)
        LDV    2
        PRN         escreva (f1)
        DALLOC 3
        DALLOC 2
        HLT         fim.
```

Note-se que a tradução de um comando composto delimitado pelos símbolos **inicio** e **fim** é obtida pela justaposição das traduções dos comandos componentes.

Uma observação importante sobre o nosso sistema de execução é que ele é muito complicado para o caso de considerar apenas os programas sem procedimentos. É fácil perceber que, neste caso, o compilador pode determinar os endereços de todas as variáveis e de todas as posições intermediárias na pilha. Poderíamos ter definido, portanto, instruções mais simples para MVD que fizessem acesso a localizações de memória sem a manipulação

explícita da pilha. Esta observação mantém-se também verdadeira para programas que têm procedimentos mas que não são recursivos.

7.7 Procedimentos sem Parâmetros

Consideremos o programa indicado na Figura 8.5, em que p é um procedimento recursivo sem parâmetros. Supondo que o valor lido pelo comando de entrada seja 4, obtém-se o diagrama de execução indicado na Figura 8.6. O número de ativações do procedimento p dependerá, em geral, do valor de n que foi lido, não sendo possível determinar-se um limite. Consequentemente, num certo instante de execução, poderão existir várias "instanciações" da variável local z de p. Por outro lado, as instruções da MVD que definimos até agora usam endereços fixos de memória, como por exemplo **LDV 3** ou **STR 7.** Isto significa que, ao traduzir expressões que envolvem a variável z, o compilador deverá usar um endereço fixo para esta variável. Entretanto, as várias instanciações não podem ocupar a mesma posição de memória.

```
programa exemplo6;
var x, y: inteiro;

    procedimento p;
      var z: inteiro;
    inicio
        z:= x; x:=x-1;
        se  z>1 entao p  (1)
                 senao  y:=1;
        y:=y*z
    fim { p };
inicio
   leia(x);
   p;  (2)
   escreva (y);
   escreva (x)
fim.
```

Figura 8.5

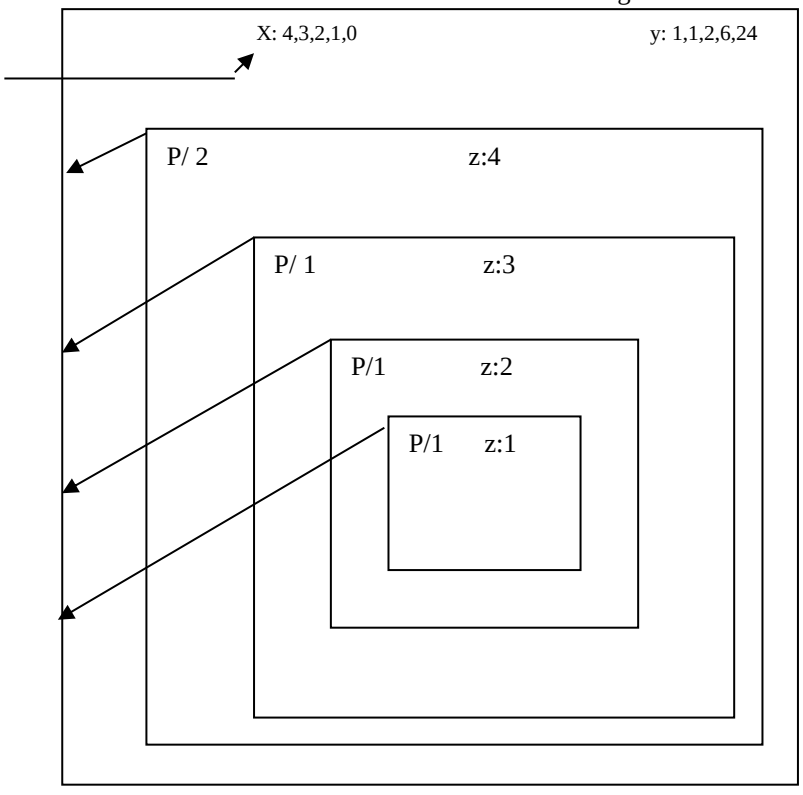


Figura 8.6

O problema pode ser resolvido notando-se que a criação e a destruição das instanciações de z seguem uma disciplina de pilha, isto é, a última instância a ser alocada é a primeira a desaparecer. Um outro fato, que podemos concluir observando as flechas estáticas da Figura 8.6, é num dado instante o programa só tem acesso à instância de z criada por último. O problema de endereçamento será resolvido, então, usando-se a própria pilha. Sempre que o procedimento p for chamado, o valor anterior de z será temporariamente guardado no topo da pilha, liberando a posição fixa, atribuída pelo compilador à variável z, para a próxima instância. O valor anterior de z será restaurado antes de executar o retorno. A Figura 8.7 indica as configurações sucessivas da pilha para cada chamada do procedimento p. Após cada retorno, a configuração prévia será restaurada. Os símbolos z1,z2,z3 e z4 denotam os valores das instanciações sucessivas de z.

Não é difícil ver que esta solução aplica-se no caso geral de procedimentos sem parâmetros. Cada procedimento alocará a sua memória, salvando no topo da pilha o conteúdo prévio das posições de memória correspondentes às suas variáveis locais. Estes valores serão restaurados antes de executarem-se os retornos. A fim de tornar o processo de tradução uniforme, o programa inteiro também será tratado como um procedimento. Redefiniremos, então, a instrução **ALLOC**, e introduziremos uma nova instrução **DALLOC:**

```
ALLOC  m,n      (Alocar memória):
                  Para k:=0 até n-1 faça
                        {s:=s+1;M[s]:=M[m+k]}

DALLOC  m,n     (Desalocar memória):
                  Para  k:=n-1  até 0  faça
                        {M[m+k]:=M[s];s:=s-1}
```

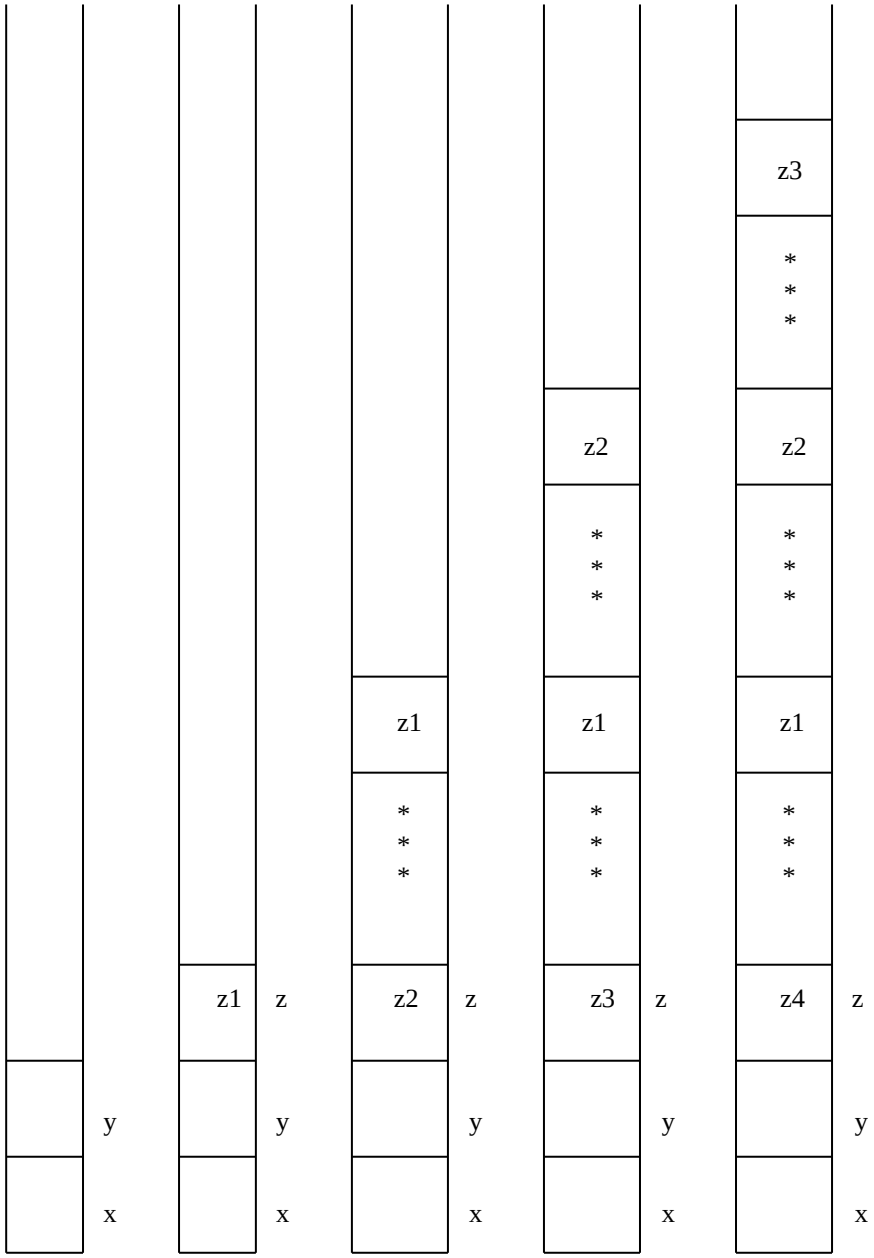


Figura 8.7

Um outro problema a ser resolvido é o próprio mecanismo de chamada de procedimentos. A única informação necessária para executar o retorno, neste estágio da MVD, é o endereço da instrução à qual a execução deve retornar. O lugar natural para guardar esta informação é a própria pilha. Definiremos, portanto, as instruções:

```
CALL   p  (Chamar procedimento ou função):
           S:=s+1;
           M[s]:=i+1;
           i:=p
```

```
RETURN  (Retornar de procedimento):
            i:=M[s];
            s:=s-1
```

Note-se que a instrução RETURN sempre encontra a informação correta no topo da pilha.

Exercício: fazer retorno de função: RETURNF.

Exemplo:

O programa da Figura 8.5 produz a seguinte tradução:

```
        START           programa
        ALLOC   0,2     var x,y
        JMP     L1
L2      NULL            procedimento p
        ALLOC   2,1     var z
        LDV     0
        STR     2       z:=x
        LDV     0
        LDC     1
        SUB
        STR     0       x:=x-1
        LDV     2
        LDC     1
        CMA             se z>1
        JMPF    L3      entao
        CALL    L2      p
        JMP     L4
L3      NULL            senao
        LDC     1
        STR     1       y:=1
L4      NULL
        LDV     1
        LDV     2
        MULT
        STR     1       y:=y*z
        DALLOC  2,1     fim
        RETURN
L1      NULL
        RD
        STR     0       leia(x)
        CALL    L2      p
        LDV     0
        PRN             escreva (x)
        LDV     1
        PRN             escreva (y)
        DALLOC  0,2     fim.
        HLT
```

Note-se o uso da instrução JMP L1 para pular o código do procedimento p ao iniciar-se a execução. Desta maneira, a tradução segue a mesma ordem do programa-fonte. Deve-se notar, também, a maneira como o compilador atribui endereços às variáveis x, y e z, que são, no caso, 0, 1 e 2, respectivamente.

Uma regra geral que pode ser adotada para atribuir endereços a variáveis, por quanto da ausência de parâmetros é: se um procedimento q, que tem k variáveis locais, está diretamente encaixado dentro de um procedimento p cuja última variável local recebeu endereço m, então as variáveis de q recebem os endereços m+1, m+2,..., m+k. Suporemos que o programa principal é um procedimento encaixado num outro tal que m=-1; consequentemente, as k variáveis do programa principal têm endereços 0, 1,...,k-1.

É importante notar que, também no caso de comandos que são chamadas de procedimentos, vale a propriedade enunciada anteriormente, ou seja, de que a execução das instruções do programa-objeto correspondentes mantém o nível da pilha final (após o retorno) igual ao inicial (antes da chamada).

8 Bibliografia

- Aho,A.V.; Sethi,R.; Ullman,J.D. *Compiladores: Princípios, Técnicas e Ferramentas* - Livros Técnicos e Científicos.

- Hopcroft,J.E; Ullman,J.D. *Formal Languages end their relation to Automata* - Addison-Wesley Series.

- Hopcroft,J.E; Ullman,J.D. *Introduction do Automata Theory, Languages and Computation* - Addison-Wesley Series.

- Kowaltowski,T. *Implementação de Linguagens de Programação* - Ed. Guanabara Dois.