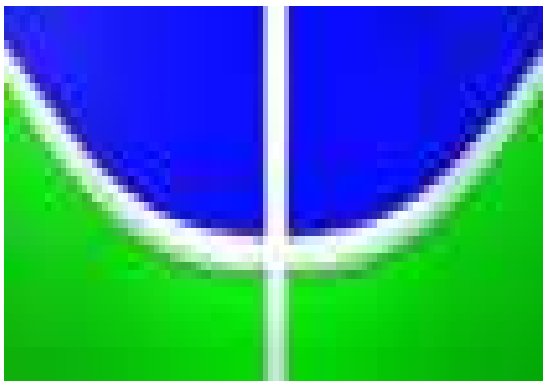
Universidade de Brasília
Departamento de Ciências da Computação
Projeto Final

PASCAL ZIM!

Implementação de um compilador contendo um subconjunto da linguagem Pascal



Luiz Reginaldo Almeida Fleury Curado

Dezembro de 1999

Orientadora : Maria Emília M. Telles Walter

Departamento de Ciências da Computação
Universidade de Brasília

PASCAL ZIM!

Implementação de um compilador contendo um subconjunto da linguagem Pascal

Luiz Reginaldo Almeida Fleury Curado

Dezembro de 1999

Banca Examinadora

- *Maria Emília M. Telles Walter (Orientadora)*
  CIC – UnB

- *Fernando A. A. C. de Albuquerque*
  CIC – UnB

- *Pedro Antônio Dourado de Rezende*
  CIC – UnB

Dedicatórias

Gostaria de dedicar esse trabalho a todas pessoas que de alguma forma estiveram presentes na minha vida no decorrer de todos esses anos de aprendizado... pois cada uma dessas pessoas me ensinou a ser um pouco do que hoje sou, e deixou comigo um pouco da sabedoria e vivência que hoje carrego.

Gostaria de dedicar esse trabalho a todos professores que me acompanharam nesses anos, e que me deram o maior dos tesouros: o conhecimento.

Gostaria de dedicar esse trabalho a todos os amigos, a todos colegas que me fizeram acreditar que um dia eu estaria escrevendo essas linhas de agradecimento.

Gostaria de dedicar esse trabalho a tantas pessoas....

Em especial, gostaria de dedicar esse trabalho aos meus pais, Luiz e Dolly, e ao meu irmão, Luciano, que estiveram presentes em cada momento da minha vida, me acompanharam em cada momento de conquista, tristeza e felicidade. Eu não teria conseguido chegar onde cheguei hoje sem o apoio de vocês....

“De tudo, ficaram três coisas:

A certeza de que estamos sempre começando...
A certeza de que precisamos continuar...
A certeza de que seremos interrompidos antes de terminar....

Portanto devemos:

Fazer da interrupção um caminho novo ...
Da queda um passo de dança...
Do medo, uma escada...
Do sonho, uma ponte...
Da procura, um encontro... ”

(Fernando Pessoa)

ÍNDICE

1. Introdução

O processo de compilação de um programa consiste, de forma simples e genérica, na tradução de um programa escrito numa certa linguagem (a linguagem *fonte* do compilador) em um programa equivalente, escrito numa outra linguagem (a linguagem *alvo* do compilador). Estes programas são denominados, respectivamente, de programa fonte e programa objeto. O processo de compilação é executado por programas especiais denominados de compiladores.

Ao longo dos anos 50, os compiladores foram considerados programas notoriamente difíceis de escrever. O primeiro compilador Fortran, por exemplo, necessitou de 18 homens-ano para ser implementado [Backus *et al*, 1970]. Desde então, foram sendo desenvolvidas técnicas sistemáticas para o tratamento das tarefas necessárias ao processo de compilação de um programa fonte. Além disso, assim como foram sendo desenvolvidas linguagens de implementação cada vez mais sofisticadas, permitindo simplificar o desenvolvimento de compiladores, através do uso intensivo de ambientes de programação mais amigáveis e ferramentas de *software* para os mais variados propósitos.

Nesse projeto final visamos implementar um compilador capaz de interpretar um subconjunto da linguagem Pascal padrão proposta por [Wirth, [3]]. O nome dado a este compilador, **Pascal Zim!,** deriva do compilador mais utilizado nos meios acadêmicos para o estudo da linguagem Pascal, o Turbo Pascal da Borland, refletindo o fato de que este compilador não implementa todas as funcionalidades de um compilador comercial.

Os princípios e técnicas envolvidas no projeto e construção de compiladores abrangem o estudo de tópicos relacionados à diversas áreas de pesquisa em Ciência da Computação, tais como as áreas de linguagens de programação, arquitetura de máquinas, teoria das linguagens, algoritmos e engenharia de software. Sendo assim, os estudos realizados para implementar este compilador são bastantes complexos, envolvendo diferentes temas estudados no decorrer do curso de graduação.

Neste contexto, o objetivo desse projeto é desenvolver um compilador Pascal que poderá ser utilizado pelos alunos da disciplina *Introdução à Ciência da Computação,* oferecida pelo Departamento de Ciências da Computação a vários cursos da Universidade de Brasília. Este compilador será uma ferramenta de apoio à aprendizagem desta disciplina.

Além da elaboração do compilador foi produzida uma documentação teórica, proveniente do estudo necessário para o projeto e desenvolvimento do compilador **Pascal ZIM!**. Este texto foi dividido em cinco capítulos. No Capítulo 2 serão apresentados os fundamentos provenientes da teoria de Linguagens Formais, além da descrição das diversas partes que constituem o processo de compilação de um programa fonte em um programa objeto. No Capítulo 3 será especificado o subconjunto da linguagem Pascal utilizado para a implementação do compilador. No Capítulo 4 serão mostrados detalhes de implementação utilizados na concepção do **Pascal ZIM!**.Finalmente, no Capítulo 5 concluímos este trabalho e sugerimos algumas extensões possíveis.

2. Fundamentação Teórica

Neste capítulo apresentamos a fundamentação teórica necessária para a implementação de um compilador Pascal.

Inicialmente, na seção 2.1 descreveremos conceitos de Linguagens Formais mais relacionados ao compilador **Pascal ZIM!**.

Em seguida, na seção 2.2 descreveremos as etapas do processo de compilação.

2.1. Linguagens Formais

Nesta serão descritos certos conceitos, da área de Linguagens Formais, que serão utilizado nas fases de análise e síntese do processo de compilação.

As expressões regulares e os autômatas serão usados para especificar o analisador léxico, enquanto que as Gramáticas Livres de Contexto serão usadas na especificação de um analisador sintático para o compilador **Pascal ZIM!**. Além disso, os conceitos abordados no estudo das Gramáticas serão utilizados na especificação da Análise Semântica e na fase de síntese do compilador.

2.1.1. Cadeias e Linguagens

O termo *alfabeto* ou *classe de caracteres* denota qualquer conjunto finito de símbolos. O conjunto {0,1}, por exemplo, é o *alfabeto binário*, que consiste dos símbolos "0" e "1". Os alfabetos de computadores EBCDIC e ASCII, consistindo no conjunto de caracteres segundo cada uma dessas convenções, é um outro exemplo de alfabeto de computadores.

Uma *cadeia* sobre algum alfabeto é uma sequência finita de símbolos retirados do mesmo. Os termos *sentença* e *palavra* são frequentemente usados como sinônimos para "cadeia".

O comprimento da cadeia s, usualmente escrito |s|, é o número de ocorrências de símbolos em s. A cadeia *vazia*, denotada $\in$, é uma cadeia especial de comprimento zero.

Uma *linguagem* denota qualquer conjunto de cadeias sobre algum alfabeto fixo. Linguagens abstratas como ϕ, o conjunto *vazio*, ou $\{\in\}$, o conjunto contendo somente a cadeia vazia, são exemplos de linguagens. Também o são o conjunto de todos os programas Pascal sintaticamente bem-formados assim como o conjunto de todas as sentenças gramaticalmente corretas em inglês.

Operações em Linguagens

Existem diversas operações que podem ser aplicadas às linguagens. Nesse tópico estaremos interessados, em particular, nas operações de *união*, *concatenação* e *fechamento.*

Dadas duas *linguagens*, L e M, podemos definir, para as mesmas:

- A operação de *união* de L e M (escrita L U M), denotada através do conjunto:

$$L \cup M = \{ s \mid s \text{ está em } L \text{ ou } s \text{ está em } M \}$$

- A operação de *concatenação* de L e M (escrita LM), denotada através do conjunto:

$$LM = \{ st \mid s \text{ está em } L \text{ e } t \text{ está em } M \}$$

- A operação de *fechamento de Kleene* de L (escrita L*), denotada através da definição:

$$L^* \text{ denota "zero ou mais concatenações de" } L$$

- A operação de *fechamento positivo* de L (escrito L^+), denotada através da definição

$$L^+ \text{ denota "uma ou mais concatenações de" } L$$

Podemos definir o operador de "exponenciação" através de uma generalização da operação de *concatenação* de linguagens. Assim, definimos- L^0 como sendo $\{\in\}$ e L^i como L^{i-1} L. Nessa notação, L^i é simplesmente L concatenada consigo mesma i-1 vezes.

Exemplo. Seja L o conjunto {A, B, ..., Z, a, b, ..., z} e D o conjunto {0, 1, ..., 9}:

1. L U D é o conjunto cujos elementos são letras ou dígitos
2. LD é o conjunto de cadeias consistindo em uma letra seguida por um dígito
3. L^4 é o conjunto de todas as cadeias contendo quatro letras
4. L* é o conjunto de todas as cadeias de letras, incluindo $\in$, a cadeia vazia.
5. L(LUD)* é o conjunto de todas as cadeias de letras e dígitos, que iniciam por uma letra.
6. D^+ é o conjunto de todas as cadeias de um ou mais dígitos.

2.1.2. Expressões Regulares

Uma *expressão regular* é definida através de um conjunto de regras recursivas sobre algum alfabeto $\sum$, onde associada a cada regra existe um especificação da linguagem denotada pela *expressão regular* sendo definida. As regras que definem uma *expressão regular são*:

1. Uma expressão regular que denota $\{\in\}$ (o conjunto que contém a cadeia vazia) é dado por $\in$.
2. Se *a* é um símbolo em $\sum$, então *a* é uma expressão regular que denota {*a*} (o conjunto contendo a cadeia *a*).
3. Sendo r e s *expressões regulares* denotando as linguagens L(r) e L(s), então:

    a) (r) | (s) é uma expressão regular denotando L(r) U L(s)
    b) (r) (s) é uma expressão regular denotando L(r) L(s)
    c) (r)* é uma expressão regular denotando (L(r))*
    d) $(r)^+$ é uma expressão regular denotando $(L(r))^+$

Cada expressão regular *r* denota uma linguagem L(r). As regras de definição especificam como L(r) pode ser formada através da combinação, em várias formas, de linguagens denotadas por subexpressões de r.

Os parênteses desnecessários podem ser evitados nas expressões regulares, se forem adotada as convenções de que:

1. Os operadores unários * e $^+$ possuem a maior precedência e seja associativos à esquerda.
2. A concatenação tenha a segunda maior precedência e seja associativa à esquerda
3. O operador de união | possua a menor precedência e seja associativo à esquerda.

Exemplo. Sob as convenções estipuladas acima, (*a*) | ((*b*) * (*c*)) é equivalente a *a* | *b***c*

A linguagem denotada por uma expressão regular é dita ser um *conjunto regular*.

Exemplo. Seja $\sum$ = {*a*,*b*}

1. A expressão regular *a* | *b* denota o conjunto {*a*,*b*}
2. A expressão regular (*a* | *b*) (*a* | *b*) denota {*aa*, *ab*, *ba*, *bb*}
3. A expressão regular *a** denota o conjunto de todas as cadeias de zero ou mais *a*'s, isto é, { $\in$, *a*, *aa*, *aaa*, ...}
4. A expressão regular (*a* | *b*)* denota o conjunto de todas as cadeias contendo zero ou mais instâncias de um *a* ou um *b*, ou seja, o conjunto de todas as cadeias de *a*'s e *b*'s.
5. A expressão regular *a* | *a**b denota o conjunto contendo a cadeia *a* e todas as cadeias consistindo em zero ou mais *a*'s seguidos por um *b*.

Definições Regulares

Por uma conveniência de notação, podemos desejar dar nomes à expressões regulares, bem como definir outras expressões que fazem uso desses nomes como se os mesmos tivessem sido definidos como sendo *símbolos* pertencentes a algum alfabeto $\sum$. Com essa finalidade definiremos agora as *definições regulares*.

Uma *definição regular* sobre um alfabeto $\sum$ é uma sequência de definições da forma

$d_1 \rightarrow r_1$
$d_2 \rightarrow r_2$
....
$d_n \rightarrow r_n$

onde cada d_i é um nome distinto e cada r_i, uma expressão regular sobre os símbolos em $\sum$ U (d_1, d_2, ... , d_{i-1}).

Com essa convenção, podemos construir uma expressão regular sobre $\sum$ para qualquer r_i substituindo-se repetidamente os nomes de expressões regulares pelas expressões que os mesmos denotam.

A fim de distinguir os nomes dos símbolos, os nomes das definições serão imprimidas em negrito.

Simplificações Notacionais

Algumas construções ocorrem de forma tão frequente nas expressões regulares, que é conveniente introduzir algumas simplificações notacionais para as mesmas.

1. *Uma ou mais ocorrências*. O operador unário pós-fixo $^+$ significa "uma ou mais ocorrências de". Se r for uma expressão regular que denote a linguagem L(r), então $(r)^+$ é uma expressão regular que denota a linguagem $(L(r))^+$.

Exemplo. A expressão regular a^+ denota o conjunto de todas as cadeias de um ou mais *a*'s.

2. *Zero ou mais ocorrências*. O operador pós-fixo unário ? significa "zero ou uma ocorrência de". A notação *r?* é uma simplificação para $r \mid \in$.

Exemplo. Se *r* for uma expressão regular, então (r)? denota a linguagem L(r) U $\{\in\}$.

3. *Classes de caracteres*. A notação [*abc*], onde *a*, *b* e *c* são símbolos de alfabeto, denota a expressão regular *a* | *b* | *c*. Uma classe de caracteres abreviada, tal como [*a*-*z*] denota a expressão regular *a* | *b* | ... | *z*.

Os números sem sinal em Pascal são cadeias como 52.80, 39.37, 6.33E64, 1.894E-4 ou ainda 3.E5 e 5. . A definição regular seguinte providencia uma precisa especificação para essa classe de cadeias:

Dígito $\rightarrow$ 0 | 1 | ... | 9
Dígitos $\rightarrow$ **Dígito**$^+$
Fração_Opcional $\rightarrow$ (. **Dígitos**) ? | .
Expoente_Opcional $\rightarrow$ (E (+ | -) ? **Dígitos**) ?
Num $\rightarrow$ **Dígitos Fração_Opcional Expoente_Opcional**

2.1.3. Autômatos Finitos

Um *reconhecedor* para uma linguagem é um programa que toma como entrada uma cadeia x e responde "sim" se x for uma sentença da linguagem e "não" em caso contrário. Expressões regulares podem ser compiladas num *reconhecedor* através da construção de um diagrama de transições generalizado chamado de autômato finito. Um autômato finito pode ser *determinístico* ou *não-determinístico*.

Autômatos Finitos Não-Determinísticos

Um *autômato finito não-determinístico* (AFN, simplificadamente) é um modelo matemático que consiste em

1. Um conjunto de estados *S*
2. Um conjunto de símbolos de entrada $\sum$ (o *alfabeto de símbolos de entrada*)
3. Uma função de transição, que mapeia pares *estado-símbolo* em conjuntos de estados.
4. Um estado s_0, que é distinguido dos outros estados como o *estado de partida* (ou *inicial*).
5. Um conjunto de estados *F* distinguidos como *estados de aceitação* (ou estados *finais*)

Um AFN pode ser representado através de um grafo dirigido e rotulado, um *grafo de transições*, no qual os nós são os estados e os lados rotulados representam a função de transição de um estado para outro. Um mesmo caracter pode rotular duas ou mais transições para fora de um mesmo estado. Os lados podem ser rotulados pelo símbolo especial $\in$ bem como pelos símbolos definidos em $\sum$.

Um AFN *aceita* uma cadeia de entrada *x* se e somente se existir algum percurso no grafo de transições, a partir do estado inicial até algum estado de aceitação, tal que os rótulos dos lados ao longo do percurso correspondam à cadeia *x*. Um percurso pode ser representado por uma sequência de transições de estados, que recebem a denominação de *movimentos* do autômata. Em geral, mais de uma sequência de movimentos pode levar a um estado de aceitação.

O *grafo de transições* para um AFN capaz de reconhecer a linguagem *(a|b)*abb* é mostrado na Figura 2.1.3.1. O conjunto de estados do AFN é {0,1,2,3} e o alfabeto de símbolos de entrada é {a,b}. O estado 0 é distinguido como o estado de partida e os estado de aceitação 3 é indicado por um círculo duplo.

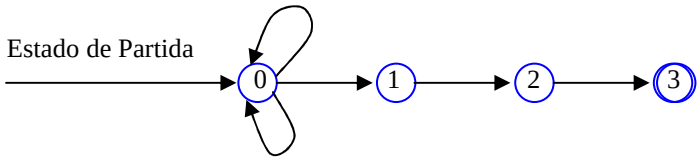


Figura 2.1.3.1. Um autômato finito não-determinístico

A função de transição de um AFN pode ser implementada de várias formas diferentes em um computador. A implementação mais fácil é através de uma *tabela de transições*, na qual existe uma linha para cada estado e uma coluna para cada símbolo de entrada e para $\in$ se necessário. A entrada para a linha *i* e símbolo *a* na tabela é o conjunto de estados que podem ser atingidos através do estado *i* e entrada *a*.

A tabela de transições para o AFN da Figura 2.1.3.1 é mostrado na tabela abaixo:

Estado	Símbolo de Entrada	
	a	*b*
0	{0,1}	{0}
1	-	{2}
2	-	{3}

Autômatos Finitos Determinísticos

Um *autômata finito determinístico* (AFD, simplificadamente) é um caso especial de autômato finito não-determinístico, no qual

1. Nenhum estado possui uma transição-$\in$, isto é, uma transição à entrada $\in$, e
2. Para cada estado *s* e símbolo de entrada *a* existe no máximo *um* lado rotulado *a* deixando *s*.

Se estivermos fazendo uso de uma *tabela de transições* para representar as funções de transição para um AFD, então cada entrada na tabela de transições consitirá de um único estado. Como consequência, é muito mais fácil determinar se um *autômato* aceita uma cadeia de entrada à partir de um *autômato finito determinístico* que à partir de um *autômato finito não-determinístico*, dado que existe no máximo um único percurso a ser seguido a partir do estado inicial *s*, rotulado por aquela cadeia, informando que o *autômato* aceita ou não a cadeia.

Tanto os *autômatos finitos determinísticos* quanto os *não-determinísticos* são capazes de reconhecer precisamente os conjuntos regulares. Entretanto, enquanto os *autômatos finitos determinísticos* podem levar a reconhecedores mais rápidos, eles tendem a ser muito maiores do que os *autômato finito não-determinístico* equivalentes.

2.1.4. Gramáticas Livres de Contexto

Uma *Gramática* é uma convenção útil para descrever a estrutura hierárquica de muitas construções inerentes à linguagens de programação. Um comando *if-else* na linguagem C, por exemplo, é dada pelo formato

if (*expressão* **)** *comando* **else** *comando*

O comando pode ser enxergado como uma *cadeia* que, vista de uma forma bem ampla, consiste de um agregado de unidades de informação significativas. Esse agregado consiste da palavra reservada **if**, de um parênteses à esquerda, uma *expressão*, um parênteses à direita, um *comando*, a palavra reservada **else** e um outro *comando*. Um *comando*, por sua vez, pode ser ainda outro comando *if-else*, como o estudado, e assim afora.

Uma *Gramática Livre de Contexto* é definida através de quatro componentes, a saber:

1. Um conjunto de *não-terminais*. Os não-terminais são *variáveis sintáticas* que denotam cadeias de caracteres. Os não-terminais definem conjuntos de cadeias que auxiliam a definição da linguagem gerada pela gramática. Impõem uma estrutura hierárquica na linguagem que é útil tanto para a análise sintática quanto para a tradução.
2. Um conjunto de *tokens*, conhecidos como *símbolos terminais* da gramática. Esses elementos são os símbolos básicos da *Gramática*, a partir dos quais as *cadeias* são formadas.
3. Um conjunto de produções, onde uma produção consiste em um não-terminal, chamado de *lado esquerdo* da produção, uma seta e uma seqüência de *tokens* e/ou *não-terminais*, chamados de *lado direito* da produção As produções de uma gramática especificam a forma pela qual os terminais e não-terminais podem ser combinados a fim de formar cadeias.
4. Uma designação a um dos não terminais como o *símbolo de partida*. O conjunto de cadeias que o mesmo denota é a linguagem definida pela gramática.

Dizemos que uma produção é *para um não-terminal* se o último figurar no lado esquerdo da primeira. Uma cadeia de *tokens* é uma seqüência de zero ou mais *tokens*. A cadeia contendo zero *tokens*, escrita $\in$, é chamada de *cadeia vazia*.

Uma gramática deriva cadeias de *tokens* começando pelo símbolo de partida e, então, substituindo repetidamente um não-terminal pelo lado direito de uma produção para aquele não-terminal. As cadeias de *tokens* que podem ser derivadas a partir do símbolo de partida formam a *linguagem* definida pela gramática.

Exemplo. A gramática com as seguintes produções define expressões aritméticas simples.

expr → *expr op expr*
expr → (*exp*)
expr → *– expr*
expr → **id**
op → +
op → –
op → *
op → /

Nesta gramática, os símbolos terminais são

id + - * / ( )

Os símbolos não-terminais são *expr* e *op*, e *expr* é o símbolo de partida.

Convenções Notacionais

A fim de tornar mais clara a distinção entre elementos da gramática serão utilizadas as seguintes convenções notacionais:

1. Símbolos terminais:

   I. Letras minúsculas do início do alfabeto, tais como *a, b, c*
   II. Símbolos de operadores, tais como +, -, etc
   III. Símbolos de pontuação, tais como parênteses, vírgula, etc
   IV. Os dígitos 0, 1, 2, ..., 9.
   V. Cadeias em negrito como **if** ou **id**.

2. Símbolos não-terminais:

   I. Letras maiúsculas do início do alfabeto, tais como A, B, C
   II. A letra S, que, quando aparece, é usualmente o símbolo de partida
   III. Os nomes em itálico formados por letras minúsculas, como *expr* ou *cmd*

3. As letras maiúsculas do final do alfabeto, tais como X, Y, Z representam *símbolos gramaticais*, isto é, terminais ou não-terminais.

4. Letras minúsculas, ao fim do alfabeto, principalmente *u, v, ..., z*, representam *cadeias de terminais*.

5. Letras gregas minúsculas, α, β, e γ, por exemplo, representam *cadeias de símbolos gramaticais*. Dessa forma, uma produção genérica poderia ser escrita como $A \rightarrow \alpha$, indicando que existe um único não-terminal A à esquerda da seta (o *lado esquerdo* da produção) e uma cadeia α de símbolos gramaticais à direita da seta (o *lado direito* da produção).

6. Se $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, ..., $A \rightarrow \alpha_k$ são todas as produções com A à esquerda (chamamos de *produções-A*), podemos escrever $A \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_k$. Chamamos α_1, α_2, ..., α_k de *alternativas* para A.

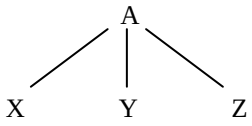
7. A menos que seja explicitamente estabelecido, o lado esquerdo da primeira produção é o símbolo de partida.

Exemplo. Usando as convenções notacionais sugeridas poderíamos escrever a gramática do *exemplo* anterior concisamente como

$E \rightarrow E\ A\ E \mid (E) \mid -E \mid \mathbf{id}$
$A \rightarrow + \mid - \mid * \mid /$

Árvores Gramaticais

Uma *árvore gramatical* mostra, pictoricamente, como o símbolo de partida de uma gramática deriva uma cadeia da linguagem. Se um *não-terminal* A possui uma produção $A \rightarrow XYZ$, então uma *árvore gramatical* pode ter um nó interior rotulado A, com três filhos rotulados X, Y e Z, da esquerda para a direita:



Formalmente, dada uma *Gramática Livre de Contexto*, uma *árvore gramatical* para essa gramática possui as seguintes propriedades:

1. A raiz da árvore é rotulada pelo símbolo de partida da gramática.

2. Cada folha da árvore é rotulada por um símbolo terminal (*token*) da gramática ou por $\in$

3. Cada nó interior da árvore é rotulado por um não-terminal da gramática.

4. Se A é um não-terminal rotulando algum nó interior e X_1, X_2, ..., X_n são os rótulos dos filhos daquele nó, da esquerda para a direita, então $A \rightarrow X_1X_2...X_n$ é uma produção. Aqui, X_1, X_2, ..., X_n figuram no lugar de símbolos que sejam terminais ou não-terminais. Como um caso especial, se $A \rightarrow \in$, então um nó rotulado A deve possuir um único filho rotulado $\in$.

As folhas da árvore gramatical, lidas da esquerda para a direita, formam o *produto* da árvore, que é a cadeia *gerada* ou *derivada* a partir do não-terminal à raiz da árvore gramatical.
Na figura 2.1.4.1, a cadeia *derivada* a partir de E é a cadeia – (**id + id**).

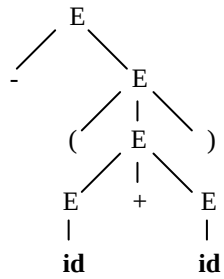


Figura 2.1.4.1. Árvore Gramatical para **– (id + id).**

A *linguagem gerada por uma gramática* é o conjunto de cadeias que podem ser geradas por alguma *árvore gramatical*.

O processo de encontrar uma árvore gramatical para uma dada cadeia de *tokens* é chamado de *análise sintática* daquela cadeia.

Derivações

Existem várias formas de se enxergar o processo pelo qual uma gramática define uma linguagem. No tópico anterior, examinamos esse processo como sendo o de construir árvores gramaticais.

Existe uma visão derivacional relacionada, que fornece uma precisa descrição da construção da da árvore gramatical do topo para as folhas (construção *top-down* da árvore gramatical), no qual o não-terminal mais á esquerda é substituído pela cadeia no lado direito da produção.

Exemplo. Consideremos a seguinte gramática para expressões aritméticas:

$E \rightarrow E + E \mid E * E \mid (E) \mid -E \mid \mathbf{id}$

A produção $E \rightarrow -E$ significa que uma expressão precedida por um sinal de menos também é uma expressão. Essa produção pode ser usada para gerar expressões mais complexas, permitindo que qualquer instância de E possa ser potencialmente substituída por –E. Podemos descrever essa ação escrevendo:

$E \Rightarrow -E$

que é lido “E deriva –E”.

A produção $E \rightarrow (E)$ diz que podemos substituir uma instância de um E em qualquer cadeia de símbolos gramaticais por (E), como por exemplo em

$E*E \Rightarrow (E)*E$ ou $E*E \Rightarrow E * (E)$.

Podemos, assim, tomar um único E e aplicar repetidamente as produções em qualquer ordem, a fim de obtermos uma seqüência de substituições, como em:

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$

Chamamos uma tal seqüência de substituições de uma *derivação* de –(**id**) a partir de E. Essa derivação providencia uma prova de que uma instância particular de uma expressão é a cadeia –(**id**).

Dizemos que $\alpha A \beta \Rightarrow \alpha\gamma\beta$ se $A \rightarrow \gamma$ for uma produção e α e β forem cadeias arbitrárias de símbolos gramaticais.

Se $\alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_n$, dizemos que α_1 *deriva* α_n. *em n passos*. Se desejamos dizer “deriva em zero ou mais passos” usamos o símbolo $\overset{*}{\Rightarrow}$ . .

Definimos, então, as seguintes propriedades:

1. $\alpha \overset{*}{\Rightarrow} \alpha$ para qualquer cadeia α, e
2. Se $\alpha \overset{*}{\Rightarrow} \beta$ e $\beta \Rightarrow \gamma$, então $\alpha \overset{*}{\Rightarrow} \gamma$

Usamos o símbolo $\overset{+}{\Rightarrow}$ para significar “deriva em um ou mais passos”.

Dada uma gramática G, com símbolo de partida S, podemos usar a relação $\overset{+}{\Rightarrow}$ para definir L(G), a *linguagem gerada por G*. Dizemos que uma cadeia de terminais *w* está em L(G) se e somente se $S \overset{+}{\Rightarrow} w$. A cadeia *w* é chamada de uma *sentença* de G. Uma linguagem que possa ser gerada por uma gramática é dita ser uma *linguagem livre de contexto*. Se duas gramáticas geram a mesma linguagem, as gramáticas são ditas *equivalentes*.

Se $S \overset{+}{\Rightarrow} \alpha$, onde α pode conter não-terminais, dizemos, então, que α é uma *forma sentencial* de G. Uma *sentença* é uma forma sentencial despida de não-terminais.

Exemplo. A cadeia –(**id** + **id**) é uma sentença da gramática de expressões usada no exemplo anterior, pois

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(\mathbf{id} + E) \Rightarrow -(\mathbf{id} + \mathbf{id})$$

A cada passo numa derivação existem duas escolhas a serem feitas.

- Primeiro, precisamos escolher qual não-terminal substituir e,
- Segundo, tendo feito tal escolha, que alternativa usar na substituição daquele não-terminal.

Certos analisadores sintáticos seguem uma sequência de derivações nas quais somente o não-terminal mais à esquerda em qualquer forma sentencial é substituído a cada passo. Tais derivações são ditas *mais à esquerda*. Se $\alpha \Rightarrow \beta$ for um passo no qual o não-terminal mais à esquerda em α será substituído, escrevemos $\alpha \Rightarrow_{mal} \beta$.

Usando nossas convenções notacionais, cada passo mais à esquerda pode ser escrito $wA\gamma \Rightarrow w\delta\gamma$, onde *w* consiste em terminais somente, $A \rightarrow \delta$ é a produção aplicada, e γ é uma cadeia de símbolos gramaticais. Para enfatizar o fato de que α deriva β por uma derivação mais à esquerda, escrevemos

$$\alpha \overset{*}{\underset{mal}{\Rightarrow}} \beta$$

Se $S \overset{*}{\underset{mal}{\Rightarrow}} \alpha$, dizemos, então, que α é uma *forma sentencial mais à esquerda* da gramática em questão. Definições análogas valem para derivações *mais à direita*, nas quais o não-terminal mais à direita é substituído a cada passo. Derivações mais à direita são chamadas de derivações *canônicas.*

Árvores Gramaticais e Derivações

Para se compreender a relação entre as derivações e as árvores gramaticais, consideremos uma derivação genérica $\alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_n$, onde α_1 é um não-terminal único A. Para cada forma sentencial α_i na derivação, construímos uma árvore gramatical cujo *produto* é α_i. O processo é uma indução em i. Como base da indução, a árvore para $\alpha_1 \equiv A$ é um único nó rotulado A. Para realizar a indução, suponhamos já ter construído uma árvore gramatical cujo produto seja $\alpha_{i-1} = X_1X_2...X_n$. Suponhamos que α_i seja derivada a partir de α_{i-1} pela substituição de X_i , um não-terminal, por $\beta = Y_1Y_2...Y_r$. Ou seja, no i-ésimo passo da derivação, a produção $X_j \rightarrow \beta$ é aplicada a α_{i-1} a fim de derivar $\alpha_i = X_1X_1... X_{j-1}\beta X_{j+1}...X_n$.

Para modelar esse passo de derivação, encontramos a j-ésima folha a partir da esquerda na árvore gramatical corrente. Esta folha é rotulada X_i . Damos a esta folha *r* filhos, rotulados $Y_1Y_2...Y_r$, a partir da esquerda. Como um caso especial, se $r = 0$, isto é, $\beta = \in$, então damos à j-ésima folha um filho rotulado $\in$.

Ambigüidade

Cada árvore gramatical possui associada a si uma única derivação mais à esquerda ou mais à direita. Uma gramática que produza mais de uma árvore gramatical para alguma sentença é dita ambígua.

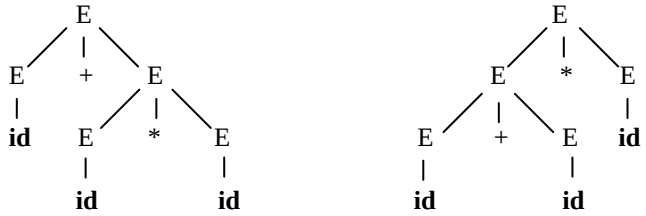
Colocado de outra forma, uma gramática ambígua é aquela que produz mais de uma derivação à esquerda, ou à direita, para a mesma sentença. Para certos tipos de analisadores sintáticos, é desejável que a gramática seja não-ambígua, porque se não o for, não poderemos selecionar, de forma única, a árvore gramatical para uma dada sentença.

Exemplo. Vamos considerar novamente a gramática de expressões.
A sentença **id** + **id** * **id** possui duas derivações distintas mais à esquerda:

$E \Rightarrow E + E \Rightarrow$ **id** $+ E \Rightarrow$ **id** $+ E * E \Rightarrow$ **id** $+$ **id** $* E \Rightarrow$ **id** $+$ **id** $*$ **id**

$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow$ **id** $+ E * E \Rightarrow$ **id** $+$ **id** $* E \Rightarrow$ **id** $+$ **id** $*$ **id**

Com as duas árvores gramaticais correspondentes:



Precedência de Operadores

A ambigüidade encontrada em gramáticas para expressões aritméticas pode ser evitada através de construções que fazem uso de definições de precedência entre os operadores da gramática. Em uma expressão aritmética contendo os operadores * e +, por exemplo, é claro que o operador * deve ter *precedência mais alta* do que o operador +. Dessa forma o operador * deverá capturar seus operandos antes do operador + o fazer.

A fim de ilustrar a ambigüidade pode ser evitada nesse caso específico são considerados dois não-terminais, *expr* e *termo*, para os dois níveis de precedência dos operadores, e um não-terminal extra, *fator*, para gerar as unidades básicas das expressões. Essas unidades básicas serão *dígitos* e *expressões parentetizadas*, como ilustra a produção a seguir.

fator → **dígito** | (*expr*)

Como o operador de multiplicação possui precedência mais alta que o operador de soma, as produções utilizando o mesmo devem ser encontrar mais próximas das unidades básicas das expressões aritméticas que as produções utilizando o operador de soma.

Assim, as produções para *termo* devem ser escritas da seguinte forma:

termo → *termo* * *fator* | *fator*

A precedência do operador de multiplicação pode agora ser alcançada através da seguinte produção:

expr → *expr* + *termo* | *termo*

A gramática resultante é , por conseguinte,

expr → *expr* + *termo* | *termo*
termo → *termo* * *fator* | *fator*
fator → **dígito** | (*expr*)

Esta gramática trata uma expressão como uma lista de fatores separados pelo operador multiplicativo. Assim, os operandos para o operador * são "capturados" antes do operador + o fazer.

Qualquer expressão parentizada é um fator e, por conseguinte, os parênteses permitem a construção de expressões com níveis arbitrários de aninhamento.

A gramática construída dessa forma, onde os operadores * e + não tem mesma precedência, não contém ambigüidades.

2.1.5. Expressões Regulares e Gramáticas Livre de Contexto

Cada construção que possa ser descrita por uma expressão regular também pode ser descrita por uma *Gramática Livre de Contexto*. Por exemplo, a expressão regular (*a* | *b*)* *abb* e a gramática

$A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$
$A_1 \rightarrow bA_2$
$A_2 \rightarrow bA_3$
$A_3 \rightarrow \in$

descrevem a mesma linguagem, o conjunto das cadeias de a's e b's terminadas em *abb*.

A conversão de um autômato finito não-determinístico (AFN) numa gramática que gera a mesma linguagem reconhecida pelo AFN é dada pelos seguintes passos:

- Para cada estado i do AFN, cria-se um símbolo não-terminal A_i.
- Se o estado i possui uma transição para o estado *j* no símbolo *a*, então deve-se introduzir na gramática a produção $A_i \rightarrow aAj$.
- Se o estado *i* vai para o estado *j* à entrada $\in$, deve-se introduzir a produção $A_i \rightarrow Aj$.
- Se *i* for um estado de aceitação, deve-se introduzir $A_i \rightarrow \in$.
- Se *i* for o estado de partida, fazer de A_i o símbolo de partida da gramática.

2.2. O processo de compilação

Basicamente, o processo de compilação pode ser subdividido em duas etapas: a fase de análise e a fase de síntese. De forma genérica, na fase de análise o programa fonte é dividido nas suas partes constituintes, sendo criada uma representação intermediária do mesmo. A fase de síntese, a partir da representação intermediária gerada, constrói o programa objeto desejado..

Mais especificamente, durante a fase de análise, as estruturas da linguagem reconhecida pelo compilador são identificadas e registradas numa estrutura hierárquica em forma de árvore, a *árvore de derivação* para o programa fonte analisado. Essa fase é comumente subdividida em três outras fases, que são bastante ligadas entre si:

1. ***Análise Léxica*** (ou análise linear): fase na qual a seqüência de caracteres que forma o programa fonte é lida, da esquerda para a direita, sendo esta sequência agrupada em *tokens*. Os *tokens* são unidades sintáticas básicas que possuem um significado próprio dentro de uma linguagem.
2. **Análise Sintática** (ou análise hierárquica): os *tokens* provenintes da análise léxica são agrupados de forma hierárquica de acordo com um conjunto de regras sintáticas definidas pela linguagem reconhecida pelo compilador.
3. **Análise Semântica:** são feitas determinadas verificações para assegurar que seja obedecido um conjunto de regras referentes à escopo e tipos, conforme definido na linguagem.

Na fase de síntese é gerado o programa objeto, sendo este normalmente escrito em código de máquina relocável ou código de montagem. Alternativamente, o compilador pode gerar um código intermediário, que pode ser interpretado por um programa denominado *interpretador*. Wirth [3], ao definir a linguagem Pascal, propôs também a definição de uma máquina virtual Pascal capaz de reconhecer um conjunto de instruções conhecido como P-Code. Esses conceitos serão mais detalhados fase de síntese do compilador e generalizadas, quando necessário, com o fim de definir estruturas de dados mais complexas do que aquelas apresentadas inicialmente pelo autor.

Na seção 2.2.1 mostramos como é realizada a Análise Léxica, que visa determinar como um compilador é capaz de identificar um conjunto de *tokens* e como esta tarefa pode ser implementada.

Na seção 2.2.2 estudaremos a Análise Sintática. De forma geral, os dois métodos de análise sintática mais utilizados na prática são os métodos *Top Down* e *Bottom Up*. Mostraremos, nessa seção, como são construídas as *árvores de derivações* para programas escritos em uma linguagem particular, assim como faremos uma comparação entre estes dois métodos.

Na seção 2.2.3. descreveremos a Análise Semântica, fase na qual pode-se definir relações de dependência semântica entre os nós de uma *árvore de derivação*, que refletem um conjunto de regras semânticas de uma linguagem particular.

Finalmente, na seção 2.2.4. estudaremos, com uma ênfase mais direcionada ao lado prático, a fase de síntese do compilador. Descreveremos, nessa seção, a estrutura e o funcionamento de uma máquina virtual, o *Computador Pascal* proposto por Wirth [3], juntamente com um conjunto de instruções que podem ser *interpretadas* por essa máquina.

2.2.1. A Análise Léxica

A análise léxica constitui a primeira fase do processo de compilação de um programa. Sua tarefa principal é a leitura de uma sequência de caracteres de entrada e, a partir dos mesmos, produzir uma seqüência de unidades sintáticas significativas, denominadas *tokens*. Essa *sequência de tokens* é utilizada pelo analisador sintático na segunda fase da compilação (a análise sintática).

A interação entre a análise léxica e a análise sintática esquematizada na Figura 2.2.1.1. Ao receber do analisador sintático a solicitação de "obter o próximo *token*", o analisador léxico efetua a leitura do programa fonte até que seja identificado o próximo *token*. O *token* identificado é então passado ao analisador sintático.

Para certos tipos de *tokens* existe a necessidade de que seja guardado um conjunto de informações, coletadas em fases posteriores do processo de tradução. Para tanto, utiliza-se uma estrutura de dados denominada de Tabela de Símbolos, que guarda informações relativas aos *tokens*.

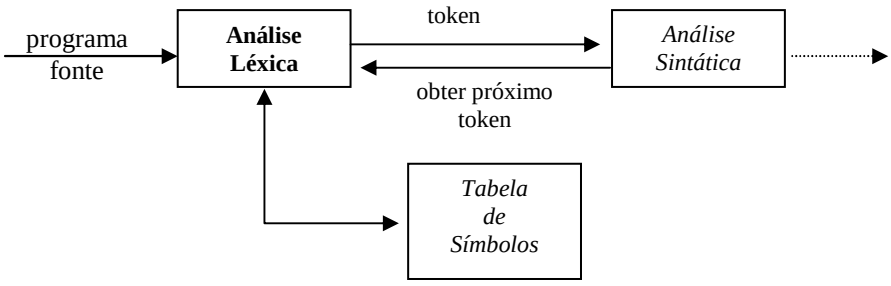


Figura 2.2.1.1. Interação entre o Analisador Léxico e o Analisador Sintático

Existem razões importantes para a distinção entre a *análise léxica* e *análise sintática*, a saber:

1. Simplificar o desenvolvimento do projeto: um analisador sintático que trate as convenções para comentários e espaços em branco é significativamente mais complexo. Essa função particular pode ser atribuída ao analisador léxico sem aumentar de forma considerável sua complexidade.
2. Aumentar a eficiência do compilador: um analisador léxico separado permite a construção de rotinas mais especializadas e potencialmente mais eficientes para a tarefa de reconhecimento das unidades sintáticas que compõem um programa fonte.

3. Incorporar portabilidade ao compilador: o tratamento dado ao alfabeto de entrada, assim como outras características específicas de dispositivos de entrada, podem ser tratadas especificamente pelo analisador léxico.

2.2.1.1. Tokens, Padrões, Lexemas

O estudo da análise léxica requer a definição de três entidades distintas:

- *Token*
- *Padrão*
- *Lexema*

Um *token* é definido como sendo a entidade básica utilizada pelo analisador sintático no reconhecimento de uma estrutura qualquer da linguagem para o qual o analisador sintático foi projetado.

Um padrão é definido como sendo um conjunto cujos elementos são todas as possíveis seqüências de caracteres passíveis de identificar um mesmo *token*.

Um lexema é definido como sendo um elemento pertencente a um padrão qualquer.

Por exemplo, a Tabela 2.2.1.1.1 mostra exemplos do uso de *tokens, lexemas* e *padrões.*

Token	*Exemplo de Lexema*	*Padrão*
Const	Const	Const
If	If	If
Identificador	Pi, D2, Ex_1, _nome	_ ou letra seguida por letras, _ e/ou dígitos
Número	3.1416, 0, 6.02E23	Qualquer constante numérica
Cadeia	“conteúdo da memória”	Quaisquer caracteres entre aspas, exceto aspa

Tabela 2.2.1.1.1. Tabela ilustrando o exemplo do uso de tokens, lexemas e padrões

A partir da tabela pode-se notar que:

- As palavras reservadas de uma linguagem, como *const* e *if*, podem ser unívocamente definidas por um padrão simples, consistindo de uma cadeia que o identifica.
- A definição de padrões para *tokens* complexos, como a utilizada para os *tokens identificador* e *número*, sugere o uso de uma notação capaz de identificar univocamente um conjunto de cadeias. As expressões regulares podem ser úteis nesse ponto.

Em geral, existe um conjunto de cadeias de entrada para as quais um mesmo *token* é produzido como saída. Na maioria das linguagens de programação as seguintes construções são tratadas como *tokens*:

- Palavras chaves,
- Operadores,
- Identificadores,
- Constantes,
- Literais,
- Cadeias
- Símbolos de pontuação

2.2.1.2. Atributos para os Tokens

O analisador léxico coleta informações a respeito de um *token* por meio de atributos associados. Geralmente, um *token* posssui nenhum ou somente um único atributo (um apontador para a entrada da tabela de símbolos na qual as informações sobre o *token* em estudo são mantidas). Os *tokens* influenciam as decisões a serem tomadas durante o a fase de análise sintática; os atributos se permitem a tradução dos *tokens*.

Exemplo. Os *tokens* e atributos associados à instrução

E := M * C + 2 – 3.1416;

são identificados abaixo através de uma sequência de pares *token* - *atributo*:

< **identificador**, apontador para a tabela de símbolos para o identificador E >
< **operador de atribuição**, >
< **identificador**, apontador para a tabela de símbolos para o identificador M >
< **operador de multiplicação**, >
< **identificador**, apontador para a tabela de símbolos para o identificador C >
< **operador de soma**, >
< **número**, valor inteiro 2 >
< **operador de subtração**, >
< **número**, apontador para a tabela de símbolos para 3.1416 >
< **ponto-e-vírgula**, >

Nota-se que certos pares não existe a necessidade de um atributo. Em tais casos, o primeiro componente do par é suficiente para identificar o *lexema* relacionado como o *token*.

2.2.1.3. Especificação e Reconhecimento de Tokens

Estudaremos agora como é possível identificar unívocamente um *token*, a partir de uma dada cadeia de caracteres. Nossa meta é encontrar um mecanismo capaz de isolar um *lexema* e produzir, como saída, um par consistindo do *token* apropriado e de um valor de atributo.

Diagramas de Transições

Para ter controle sobre as informações de uma cadeia de caracteres à medida em que cada caractere individual que a compõe vai sendo analisado, utiliza-se um *diagrama de transições*, que na verdade, é um autômata finito determinístico.

Os estados em um *diagrama de transições* são representados através de círculos, chamados de *estados*. Uma seta ligando dois *estados* indica uma *transição entre dois estados*, e recebe a denominação de *aresta*. As *arestas* que saem de um estado *s* são rotulados por caracteres, que servem para indicar que a devida transição entre dois estados é feita se na entrada for encontrado um caractere que rotula a *aresta*. Uma *aresta* é rotulada **outro** para se referir a qualquer caractere diferente dos outros que rotulam as *arestas* deixando um *estado*. Os estados diferenciados por círculos duplos são estado de aceitação. Os *diagramas de transições* são *determinísticos*, e dessa forma um mesmo símbolo não pode figurar como rótulo de dois *lados* diferentes que deixem um mesmo *estado*.

O funcionamento de um *diagrama de transições* pode ser sumarizado através dos seguintes passos:

- Um dos *estados* do *diagrama* é tomado como sendo o estado *de partida*. Este é o estado onde é iniciado o reconhecimento de uma cadeia de caracteres.
- Ao entrar num estado, é lido o próximo caractere de entrada. Se existir uma *aresta* a partir do estado corrente cujo rótulo seja igual a esse caractere de entrada tornamos o *estado* apontado pela *aresta* como estado corrente.
- Se o estado atingido for um estado de aceitação, então a cadeia de caracteres foi reconhecida com sucesso pelo *diagrama de transições*. Em caso contrário, a cadeia não pôde ser reconhecida.

A Figura 2.2.1.3.1 mostra um *diagrama de transições* para os padrões > e >=. O diagrama funciona da seguinte forma: seu estado inicial é o estado 0. Neste, lemos o próximo caractere de entrada. A *aresta* rotulada > a partir do estado 0 deve ser seguida até o estado 1, se esse caractere de entrada for >. Se não for, falhamos em reconhecer > ou >=.

Ao atingirmos o estado 1, lemos o próximo caractere de entrada. A *aresta* rotulada = a partir do estado 1 deve ser seguida até o estado 2, se o caractere de entrada for um =. Se não for, a *aresta* rotulada **outro** indica que devemos nos dirigir para o estado 3. O círculo duplo no estado 2 informa que este é um estado de aceitação, no qual o *token* >= foi encontrado.

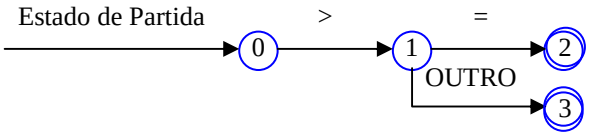


Figura 2.2.1.3.1. Diagrama de transições para os padrões >= e >.

Implementando um Diagrama de Transições

Um *diagramas de transições* pode ser transformado em um programa para o reconhecimento de um conjunto de *tokens*, onde cada estado recebe um segmento de código. Se existirem *arestas* deixando um estado, então o código para aquele *estado* lê um caractere e seleciona um *aresta* para seguir, se possível. Se existir uma *aresta* rotulada pelo caracter lido o controle é, então, transferido para o *estado* apontado por aquele lado. Se não existir tal *aresta* não existem ações a serem tomadas no *diagrama* para o caractere em questão, e uma rotina para tratamento de erros léxicos deve ser invocada.

O conjunto de cadeias reconhecido por um *diagrama de transições* pode ser especificado através de um conjunto de *expressões regulares*, onde cada *expressão regular* está relacionada com um *subdiagrama de transições* capaz de reconhecer um dado *token*.

2.2.1.4. Comparação de uso entre expressões regulares e gramáticas livres de contexto

Uma vez que cada *expressão regular* pode ser definida através de uma *gramática livre de contexto*, é razoável questionar a escolha da primeira abordagem na definição léxica dos *tokens* reconhecidos durante a análise léxica. As razões abaixo justificam a escolha das *expressões regulares*:

1. As regras léxicas de uma linguagem são freqüentemente simples e para descrevê-las não é preciso uma notação tão poderosa quanto a das gramáticas.
2. As *expressões regulares* geralmente fornecem, para os *tokens* da gramática, uma notação mais concisa e facilmente compreensível.
3. A partir de *expressões regulares* podem ser construídos automaticamente analisadores léxicos mais eficientes do que a partir de gramáticas.

Não existem diretrizes claras sobre o que colocar nas regras léxicas, ao contrário das regras sintáticas. As expressões regulares são mais úteis para descrever a estrutura de construções léxicas tais como identificadores, constantes, palavras-chave e assim por diante. Por outro lado, as gramáticas são mais úteis na descrição de estruturas aninhadas tais como parênteses balanceados, *begin-ends* emparelhados, *if-then-elses* correspondentes, estruturas que não podem ser descritas por expressões regulares.

2.2.2. A Análise Sintática

Uma linguagem de programação é definida segundo um conjunto de regras sintáticas que definem a estrutura de um programa escrito nessa linguagem. Em Pascal, por exemplo, um programa é constituído por blocos, um bloco por comandos, um comando por expressões, uma expressão por *tokens*. A análise sintática é o processo de determinar se uma cadeia de *tokens* pode ou não ser gerada por uma das regras sintáticas que compõem a gramática de uma linguagem de programação.

A sintaxe das construções de uma linguagem de programação pode ser descrita pelas gramáticas livres de contexto ou pela notação BNF (Forma de Backus-Naur). As gramáticas oferecem vantagens significativas tanto para os projetistas de linguagens como para os projetistas de compiladores:

- Uma gramática oferece, para uma linguagem de programação, uma especificação sintática precisa e fácil de entender.
- Pode ser usada para definir a estrutura de linguagens de programação
- Para certas classes de gramáticas é possível a construção de um analisador sintático capaz de determinar se a estrutura sintática de um programa está correta.
- É útil para a tradução correta de programas-fonte em código-objeto e também para detectar erros (léxicos, sintáticos e semânticos).

Como já foi dito anteriormente, no modelo de compilador utilizado neste projeto, o analisador sintático obtém uma seqüência de *tokens* proveniente do analisador léxico, e verifica se essa seqüência corresponde à estrutura sintática definida na gramática da linguagem fonte do compilador (Figura 2.3.1). A saída de um analisador sintático, após ter sido analisado todo o programa fonte é uma representação em forma de árvore gramatical para a sequência de *tokens* obtida do analisador léxico.

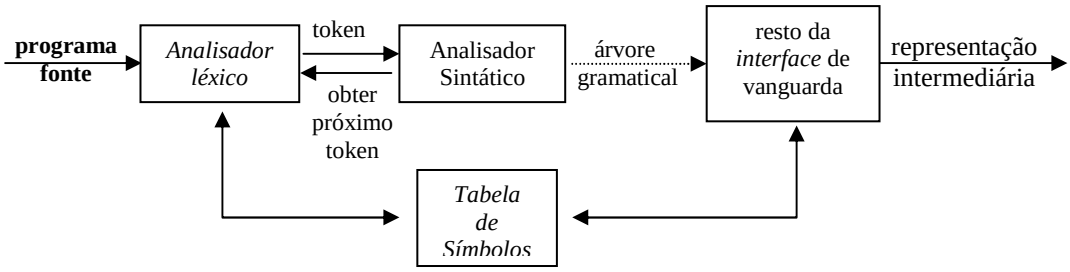


Figura 2.2.2.1. O analisador sintático na implementação de um compilador.

A maioria dos métodos de análise sintática pertence a uma dentre duas classes, chamadas de *top-down* e *bottom-up*. Esses termos referem à ordem pela qual os nós da árvore gramatical são construídos. No método *top-down*, a construção é iniciada na raiz e prossegue em direção às folhas (análise sintática descendente), enquanto que no *bottom-up*, a construção é iniciada nas folhas e prossegue em direção à raiz (análise sintática ascendente).

Os métodos de análise sintática mais eficientes, tanto *top-down* quanto *bottom-up*, trabalham somente em determinadas subclasses de gramáticas, mas várias dessas subclasses, como as das gramáticas LL e LR, são suficientemente expressivas para descrever as construções sintáticas das linguagens de programação. Os analisadores implementados sem ferramentas automatizadas trabalham freqüentemente com gramáticas LL, enquanto os da classe das gramáticas LR são usualmente construídos através de ferramentas automatizadas.

Na prática, existe um certo número de tarefas que poderiam ser conduzidas durante a análise sintática, tais como coletar informações sobre os vários *tokens* na tabela de símbolos, realizar 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* de vanguarda”.

Nessa seção serão estudados dois métodos de análise sintática, análise sintática descendente (*top-down*) e ascendente (*bottom-up*). No final da seção será feita uma breve comparação entre os dois métodos.

2.2.2.1. A Análise Sintática Descendente

A análise sintática descendente pode ser entendida como uma tentativa de encontrar uma derivação mais à esquerda para uma cadeia de entrada ϖ. Equivalentemente, pode ser vista como uma tentativa de construir, a partir da identificação de uma sequência de *tokens* em ϖ, uma árvore gramatical, onde a construção se inicia a partir da raiz e avança em direção à construção das folhas.

O *token* analisando durante o processo de análise sintática para uma cadeia de entrada ϖ recebe a denominação de *símbolo lookahead*. No início do processo de análise sintática, o *símbolo lookahead* é o *token* mais à esquerda da sequência definida em ϖ.

A construção de uma árvore gramatical no processo de análise sintática descendente é feita de forma recursiva, tomando-se como ponto de partida a raiz da árvore, rotulada pelo não-terminal de partida da gramática. Três passos definem a construção recursiva da árvore gramatical:

1. Para um nó *n* da árvore, rotulado por um não-terminal A, seleciona-se uma das produções para A.
2. Constrói-se na árvore, a partir de *n*, uma estrutura hierárquica, onde os símbolos definidos no lado direito da produção escolhida no primeiro passo figuram como filhos de *n*.
3. Repete-se os passos 1 e 2 até que não seja mais possível a adição de novos nós à árvore.

Para algumas gramáticas os passos acima podem ser implementados durante uma única leitura da cadeia de entrada, da esquerda para a direita. A identificação de cada um dos *tokens* que compõem essa cadeia de entrada, a partir do *token* mais à esquerda, permite a análise sintática para a cadeia.

Por exemplo, se consideremos a seguinte gramática:

tipo → *tipo_simples*
| **array** [*tipo_simples*] **of** *tipo*
tipo_simples → **integer**
| **char**
| num pontoponto num

e a seguinte cadeia de entrada:

array [**num pontoponto num**] **of integer**

O processo de análise sintática da cadeia de entrada, pode ser descrito da seguinte forma :

- Inicialmente, o *token* **array** é o símbolo *lookahead* e a parte conhecida da árvore gramatical sendo construída consiste na raiz, rotulada pelo não-terminal de partida *tipo*.
- O não-terminal *tipo*, agora, precisa derivar uma cadeia que inicie pelo símbolo *lookahead* (**array)**. Na gramática do exemplo, existe uma única produção que satisfaz essa exigência. Tal produção é então selecionada, e os filhos para a raiz da árvore gramatical são construídos, cada filho rotulado com um símbolos pertinente ao lado direito da produção.
- O nó em consideração na árvore gramatical passa a ser o nó rotulado pelo *lookahead* **array**.

Após terem sido feitos esses passos iniciais, o resto do processo de análise usa as seguintes convenções:

- Quando o nó em consideração na árvore gramatical é o de um terminal e este último corresponde ao símbolo *lookahead*, o próximo *token* à entrada se torna o novo símbolo *lookahead* e o próximo filho na árvore gramatical é considerado.
- Quando o nó em consideração na árvore gramatical é o de não-terminal o processo de selecionar uma produção para o não-terminal é feito. O nó em análise na árvore gramatical se torna, agora, o nó relativo ao filho mais à esquerda para o não-terminal expandido.

Na figura 2.2.2.1.1(a), os filhos da raiz foram construídos e o filho mais à esquerda, rotulado por **array**, está sendo examinado. As setas, na Figura 2.2.2.1.1 servem para identificar o atual símbolo *lookahead* durante o processo de análise sintática. Na figura 2.2.2.1.1 (b), os filhos para o não-terminal *tipo_simples* foram construídos e o filho mais à esquerda, rotulado por **[** , está em exame.

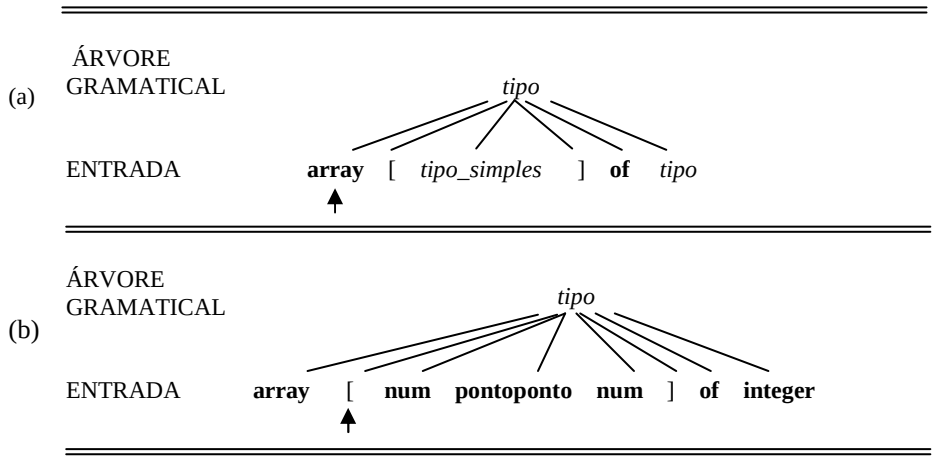


Figura 2.2.2.1.1. Exemplo de construção de uma arvore gramatical para
array [**num pontoponto num**] **of integer**

A seleção de uma produção para um não-terminal, durante a análise sintática descendente, pode envolver tentativa e erro. Assim sendo, se a escolha de uma produção falhar na derivação de uma dada cadeia de entrada, deve-se retroceder na construção da árvore de derivação e tentar fazer uso de uma outra produção inicializada pelo símbolo *lookahead* anteriormente considerado.

Por exemplo, se consideremos a gramática:

$S \rightarrow cAd$
$A \rightarrow ab \mid a$

e a cadeia de entrada $\varpi = cad.$, a construção de uma árvore gramatical para ϖ, iniciada a partir da raiz, é iniciada com a construção de um nó para S, o símbolo de partida da gramática. O símbolo *lookahead* é *c*, o primeiro símbolo de ϖ.

A primeira produção para S na expansão da árvore gera a árvore da Figura 2.2.2.1.2.(a).

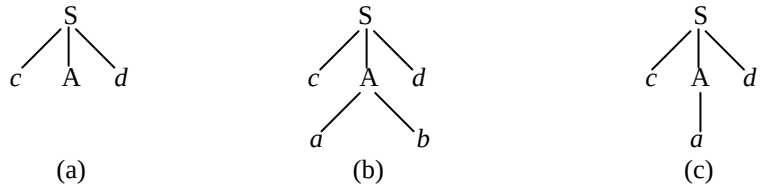


Figura 2.2.2.1.2. Exemplo da construção de uma árvore gramatical numa análise sintática *top-down*

A folha mais à esquerda, rotulada *c*, reconhece o primeiro símbolo de ϖ e, por conseguinte, deve-se analisar o próximo nó da árvore. O próximo símbolo esperado é o próximo símbolo de ϖ *a*, e o nó a ser considerado passa a ser o nó rotulado por A. Como A é um não-terminal, é feita uma expansão na árvore através do uso da primeira das duas produções para esse não-terminal. O nó em consideração passa a ser agora o nó rotulado por *a.* A árvore obtida é a árvore da Figura 2.2.2.1.2(b).

Como o *a* reconhece o segundo símbolo de ϖ, deve-se procurar agora uma correspondência para o terceiro símbolo da cadeia de entrada, *d*. A terceira folha da árvore, rotulada *b*, é comparada com *d*. Como *b* não corresponde ao símbolo esperado, *d*, deve-se "resgatar" a árvore que expandiu A, com o fim de se utilizar uma outra produção para esse não-terminal (uma que ainda não tenha ainda sido tentada), a fim de tentar produzir um reconhecimento para a cadeia ϖ.

Tentamos a segunda produção para A, obtendo a árvore da Figura 2.2.2.1.2(c). A folha *a* reconhece o segundo símbolo de *w* e a folha *d* o terceiro. Uma vez que produzimos uma árvore gramatical para ϖ, anunciamos o término com sucesso da análise sintática.

2.2.2.2. Análise Sintática Descendente Recursiva

A *análise sintática descendente recursiva*, também conhecida como *análise sintática preditiva*, é um método *top-down* de análise sintática no qual é executado um conjunto de procedimentos recursivos no processo de reconhecimento de uma cadeia de entrada, onde cada procedimento é associado a um não-terminal da gramática.

Nesse método de análise sintática o símbolo *lookahead* determina de forma não-ambígua o procedimento selecionado para cada não-terminal. A seqüência de procedimentos chamados define, implicitamente, a construção de uma árvore gramatical .

Por exemplo, o analisador sintático preditivo da Figura 2.3.1.3. consiste em procedimentos para os não-terminais *tipo* e *tipo_simples* da gramática de tipos esboçada dois exemplos atrás. O procedimento *reconhecer* é usado para simplificar o código para *tipo* e *tipo_simples; reconhecer* avança para o próximo *token* de entrada se seu argumento *t* for igual ao símbolo *lookahead*.

```
procedimento reconhecer (t: token);
início
  se lookahead = t então
  lookahead := próximo_token
  senão erro
fim;

procedimento tipo_simples;
início
  se lookahead = integer
    então reconhecer(integer);
  senão se lookahead = char então
        reconhecer(char);
  senão se lookahead = num então início
        reconhecer(num); reconhecer(pontoponto); reconhecer(num);
  fim
  senão erro
fim;

procedimento tipo;
início
  se lookahead está em { integer, char, num } então
  tipo_simples
 senão se lookahead = array então início
   reconhecer(array); reconhecer('['); tipo_simples;
   reconhecer(']'); reconhecer(of); tipo;
 fim
 senão erro
fim;
```

Figura 2.3.1.3. Pseudocódigo para um analisador gramatical preditivo

Para a cadeia de entrada

array [**num** **pontoponto** **num**] **of** **integer** (1)

a análise sintática é iniciada com uma chamada para o procedimento correspondente ao não-terminal de partida de nossa gramática, *tipo*. *Lookahead* é inicialmente o primeiro *token*, **array**.

O procedimento *tipo* executa o código a seguir:

reconhecer(**array**); *reconhecer*('['); *tipo_simples*;
reconhecer(']'); *reconhecer*(**of**); *tipo*;

correspondendo ao lado direito da produção

tipo $\rightarrow$ **array** [*tipo_simples*] **of** *tipo*

Nota-se que cada terminal presente no lado direito da produção para *tipo* é confrontado com o símbolo *lookahead*, e cada não-terminal leva a uma chamada de seu procedimento correspondente.

No fragmento de código (1), após os *tokens* **array** e [terem sidos reconhecidos, o símbolo *lookahead* é **num**. Neste ponto, o procedimento *tipo_simples* é chamado, sendo executado o código

reconhecer(**num**); *reconhecer*(**pontoponto**); *reconhecer*(**num**);

O símbolo *lookahead* guia a seleção da produção a ser usada. Se o lado direito de uma produção for iniciado por um *token*, então esta produção pode ser usada quando o símbolo *lookahead* for igual ao *token*.

A construção de um analisador sintático preditivo requer o conhecimento, dado um símbolo de entrada *a* e um não-terminal A a ser expandido, de qual das alternativas $A \rightarrow \alpha_1 \mid \alpha_1 \mid .... \mid \alpha_n$ irá derivar uma cadeia iniciando por *a*. Assim, essa forma particular de análise gramatical repousa na informação dos primeiros símbolos que podem ser gerados pelo lado direito de uma produção a ser expandida.

Partindo desse pressuposto podemos definir, para uma cadeia de símbolos gramaticais α, uma função FIRST(α), que retorna um conjunto de *tokens* passíveis de figurar como primeiro símbolo em uma ou mais cadeias derivadas a partir de α. Na definição da função, se α é $\in$ ou puder gerar $\in$, então $\in$ pertencerá a FIRST(α).

Exemplo. Para a Figura 2.3.1.3:

FIRST(*tipo_simples*) = { **integer**, **char**, **num** }
FIRST(**array** [*tipo_simples*] **of** *tipo*) = { **array** }

Tendo-se a coleção de conjuntos FIRST para cada não-terminal de uma dada gramática é possível a construção de uma analisador sintático preditivo para a essa gramática, com a seguinte ressalva:

- Se existirem duas produções $A \rightarrow \alpha$ e $A \rightarrow \beta$ a análise gramatical descendente recursiva sem retrocesso requer que FIRST(α) e FIRST(β) sejam disjuntos. O símbolo *lookahead* decide qual produção usar: se estiver *lookahead* estiver em FIRST(α), então, a produção com α do lado direito é usada, se o símbolo *lookahead* estiver em FIRST(β), então a produção com β do lado direito será usada. Se o símbolo *lookahead* estiver tanto em FIRST(α) como em FIRST(β), o analisador sintático o analisador sintático entrará em um estado inconsistente, no qual não sabe qual produção deve ser usada.

Recursão à esquerda

Um analisador sintático descendente recursivo pode entrar em laço infinito. O problema surge em produções recursivas à esquerda, quando o símbolo mais à esquerda do lado direito da produção é o mesmo que o não-terminal no lado esquerdo, como na produção abaixo:

expr→ *expr* + *termo*

A aplicação da produção *expr* faz com que o procedimento *expr* seja novamente chamado recursivamente, e essa série de chamadas prossegue indefinidamente.

Como generalização do problema, consideremos um não-terminal A, e duas produções

$A \rightarrow A\alpha \mid \beta$

onde α e β são seqüências de terminais e não-terminais não iniciados por A. A produção para A é *recursiva à esquerda* pelo fato de $A \rightarrow A\alpha$ possuir o não-terminal A como símbolo gramatical mais à esquerda no lado direito dessa produção.

A produção constrói uma seqüência consistindo de β, seguido de zero ou mais α's . Esse mesmo resultado pode ser obtido através da rescrita das produções para A da seguinte maneira:

$A \rightarrow \beta R$
$R \rightarrow \alpha R \mid \in$

Onde a recursividade à esquerda foi eliminada, em favor da *recursividade à direita n*a produção $R \rightarrow \alpha R$.

Uma gramática recursiva à esquerda pode levar um analisador sintático descendente recursivo, mesmo com retrocesso, a um laço infinito, pois quando tentamos expandir A, podemos eventualmente nos encontrar novamente tentando expandir A sem ter consumido nenhum símbolo da entrada.

Fatoração à esquerda

A *fatoração à esquerda* é uma transformação gramatical útil na definição de uma gramática sem conflitos para a análise sintática preditiva. A idéia básica para a fatoração á esquerda é definir uma regra que permita, nos casos de não ser possível decidir qual de duas produções alternativas usar na expansão um não-terminal A, reescrever as produções-A de forma que a escolha da produção a ser usada na expansão seja postergada até que se tenha visto o suficiente da cadeia de entrada.

Em geral, se tivermos $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ duas produções-A, e uma cadeia de entrada ϖ iniciada por uma cadeia não vazia derivada a partir de α, não é possível saber se A será expandida em $\alpha\beta_1$ ou em $\alpha\beta_2$.

Podemos, entretanto, postergar a decisão de expandir A para $\alpha A'$. A idéia é que, após termos enxergarmos a cadeia de entrada derivada a partir de α, possamos expandir A' em β_1 ou em β_2.

Exemplo. As produções $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ fatoradas à esquerda, se tornam:

$A \rightarrow \alpha A'$
$A' \rightarrow \beta_1 \mid \beta_2$

O uso dos artifícios de eliminação da recursão e fatoração à esquerda com o fim de adequar gramáticas ao uso da análise sintática preditiva recursiva possuem ao menos um incoveniente: acabam tornando as gramáticas bastante complexas, e difíceis de serem usadas para os requisitos de tradução.

2.2.2.3. Análise Sintática Descendente Não-Recursiva

A construção de um *analisador sintático descendente não-recursivo* pode ser feita com o uso de uma pilha, em substituição às chamadas recursivas. Três estruturas definem um analisador sintático desse tipo, a saber:

1. Um buffer de entrada
2. Uma pilha
3. Uma tabela sintática

O *buffer* de entrada armazena a cadeia a ser analisada, seguida por um $ à direita indicando o seu término. A pilha contém uma seqüência de símbolos gramaticais, com $ indicando o fundo da pilha. A tabela sintática usada pelo analisador sintático consiste de uma matriz bidimensional M[A,*a*], onde A é um não-terminal e *a* é um terminal ou o símbolo $, e é usada para definir qual produção deve ser aplicada.

O relacionamento entre essas três estruturas e o analisador sintático é ilustrado na figura 2.3.1.4.

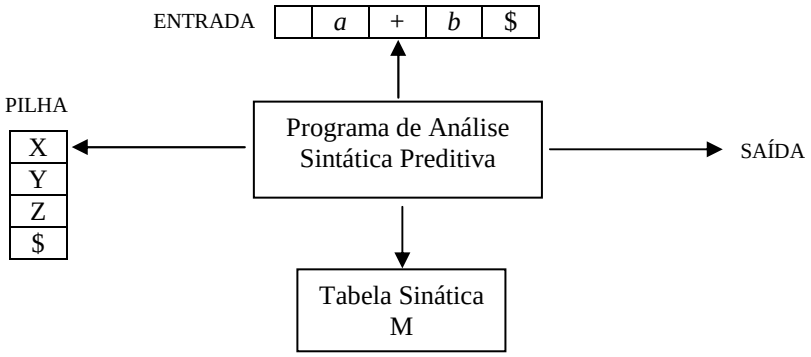


Figura 2.3.1.4. Modelo de um analisador sintático preditivo não-recursivo

O funcionamento básico desse analisador sintático é descrito através dos seguintes passos:

- Inicialmente, a pilha contém o símbolo inicial da gramática acima de $.
- O programa identifica X, o símbolo ao topo da pilha, e *a*, o símbolo corrente de entrada.
- A partir de X e *a* o analisador sintático define uma ação, que é uma dentre as três seguintes:

  1. Se X = *a* = $, o analisador pára e anuncia o término com sucesso da análise sintática.
  2. Se X = *a* ≠ $, o analisador sintático remove X da pilha e avança o apontador da entrada para o próximo símbolo.
  3. Se X é um não-terminal, o programa consulta a entrada M[X,*a*] da tabela sintática M. Essa entrada será uma produção-X da gramática ou uma entrada de erro. Se, por exemplo, M[X,*a*] = {X→ UVW}, o analisador substitui X no topo da pilha por WVU (com U no topo da pilha). No caso da entrada M[X,*a*] não estar definida, um erro é reportado.

Uma gramática para um analisador sintático preditivo cuja tabela sintática não possua entradas multiplamente definidas é dita LL(1). O primeiro "L" em LL(1) significa a varredura da entrada da esquerda para a direita (*left to right*); o segundo, a produção de uma derivação mais à esquerda (*left linear*); e o "1", o uso de um único símbolo de entrada como *lookahead* a cada passo para tomar as decisões sintáticas.

Exemplo. A entrada para M[S',e] na tabela sintática preditiva da Figura 2.3.1.5 contém duas produções:

1. S' → *e*S
2. S' → ∈ .

NÃO-TERMINAL	SÍMBOLO DE ENTRADA					
	A	*b*	*E*	*i*	*T*	$
S	S → *a*			S → *i*E*t*SS'		
S'			S → ∈ S → *e*S			S → ∈
E		E → *b*				

Figura 2.3.1.5. Exemplo de Tabela Sintática Preditiva

A presença de uma entrada duplicada em uma tabela sintática significa que não é possível decidir, para o par (não-terminal, terminal) sob o qual a entrada foi definida, qual produção deve ser usada. Uma gramática que gera uma tabela sintática com entradas multiplamente definidas é dita uma *gramática ambígua*. A gramática do exemplo é ambígua.

2.2.2.4. Conclusões sobre a análise sintática descendente

A dificuldade principal em usar a análise preditiva está em escrever uma gramática que não seja recursiva à esquerda, nem ambígua. Gramáticas escritas para utilização em analisadores sintáticos preditivos que possuem estas características podem ser transformadas para que sejam eliminadas a ambigüidade e a recursividade à esquerda. Mesmo com essa abordagem dois problemas ainda persistem:

- Apesar da eliminação da recursão à esquerda e da fatoração à esquerda serem fáceis de aplicar, ambas tornam a gramática resultante difícil de ler e usar para os fins da tradução.
- Existem algumas gramáticas para as quais nenhuma alteração irá produzir uma gramática LL(1).

Esses problemas sugerem a busca de um método de análise sintática que não seja tão restritivo e limitado. Com esse intuito, passamos agora ao estudo da Análise Sintática Ascendente.

2.2.2.5. Análise Sintática Ascendente

A análise sintática ascendente pode ser vista como uma tentativa de construir, a partir da identificação de uma seqüência de *tokens* constituintes de uma cadeia de entrada ϖ, uma árvore gramatical, onde a construção é iniciada a partir das folhas e avança em direção à raiz. Em virtude da maneira como é construída a árvore gramatical esse método comumente é chamado de *análise sintática ascendente.*

O processo de análise sintática ascendente pode ser visualizado como o processo de redução de uma cadeia ϖ ao símbolo de início de uma gramática. A cada passo da *redução*, uma subcadeia particular, que reconheça o lado direito de uma produção, é substituída pelo não-terminal à esquerda daquela produção e, se a subcadeia tiver sido escolhida corretamente a cada passo, uma derivação mais à direita terá sido detectada na ordem inversa.

Por exemplo, consideremos a gramática

S → *a*AB*e*
A → A*bc* | *b*
B → *d*

A cadeia ϖ = *abbcde* pode ser reduzida a S pelos seguintes passos:

abbcde
*a*A*bcde*
*a*A*de*
*a*AB*e*
S

Através de uma seqüência de quatro reduções é possível reduzir *abbcde* de S. Essas reduções rastreiam a seguinte derivação mais à direita, na ordem reversa:

$S \Rightarrow_{mad} aABe \Rightarrow_{mad} aAde \Rightarrow_{mad} aAbcde \Rightarrow_{mad} abbcde$

Handles

Um *handle* é uma subcadeia que reconhece o lado direito de uma produção, e cuja redução à um não-terminal do lado esquerdo dessa produção representa um passo ao longo do percurso de uma sequência de derivações mais à direita. No exemplo anterior, o *handle* para *abbcde* é A $\rightarrow$ *b* na posição 2.

A figura 2.3.2.1 mostra o formato genérico de um *handle* A $\rightarrow \beta$ em uma árvore gramatical para uma cadeia $\alpha\beta w$.

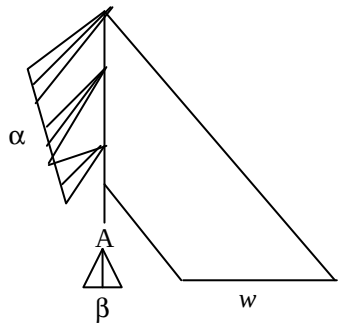


Figura 2.3.2.1. O *handle* A $\rightarrow \beta$ na árvore gramatical de $\alpha\beta w$

Na figura, o *handle* A $\rightarrow \beta$ representa a subárvore completa mais à esquerda para a cadeia $\alpha\beta w$, que consiste em um nó e todos os seus filhos. Esse nó é o nó mais ao fundo e mais à esquerda formando uma subárvore com todos os seus filhos na árvore completa. A redução de β para A em $\alpha\beta w$ é denominada a "poda do *handle*", e consiste na remoção dos filhos de A da árvore gramatical.

Uma derivação mais à direita para uma cadeia de entrada ϖ pode ser obtida através de uma sequência de "*podas*" em ϖ. A idéia para esse processo de derivação é baseado no fato de que, se ϖ é uma sentença da gramática em questão, então $\varpi = \gamma_n$, onde γ_n é a enésima forma sentencial à direita de alguma derivação mais à direita ainda desconhecida.

A derivação de ϖ, a partir do símbolo de partida da gramática, S, é generalizada através da seguinte sequência de reduções:

$S = \gamma_0 \Rightarrow_{mad} \gamma_1 \Rightarrow_{mad} \gamma_2 \Rightarrow_{mad} \Rightarrow_{mad} \ldots \Rightarrow_{mad} \gamma_{n-1} \Rightarrow_{mad} \gamma_n = \varpi$

Essa derivação pode ser obtida, na ordem inversa, a partir da execução dos seguintes passos:

- Localizamos o *handle* β_n em γ_n e substituímos β_n pelo lado direito de alguma produção $A_n \rightarrow \beta_n$, de modo a obtermos a enésima menos uma forma sentencial à direita γ_{n-1}.
- Localizamos o *handle* β_{n-1} em γ_{n-1} e o reduzimos de forma a obter uma forma sentencial à direita γ_{n-2}.
- Continuando esse processo, produzimos uma forma sentencial à direita consistindo somente no símbolo de partida S. O reverso da seqüência de produções usadas nas reduções é uma derivação mais à direita para a cadeia de entrada.

Por exemplo, considerando a cadeia de entrada ϖ = **id** + **id * id**, e a gramática ambígua

E $\rightarrow$ E + E | E * E | (E) | **id**

a seqüência de reduções mostrada na Figura 2.3.2.2. reduz ϖ à E.

Forma Sentencial À Direita	Handle	Produção Redutora
Id + **id * id**	**id**	E $\rightarrow$ **id**
E + **id * id**	**id**	E $\rightarrow$ **id**
E + E *** id**	**id**	E $\rightarrow$ **id**
E + E ***** E	E * E	E $\rightarrow$ E * E
E + E	E + E	E $\rightarrow$ E + E
E		

Figura 2.3.2.2. Reduções realizadas por um analisador sintático de empilhar e reduzir

2.2.2.6. Implementação de Pilha da Análise Sintática de Empilhar e Reduzir

Uma maneira conveniente de se implementar uma analisador sintático ascendente é fazendo uso de duas estruturas auxiliares:

- Uma pilha, para guardar os símbolos gramaticais
- Um *buffer* de entrada, que armazena a cadeia de entrada ϖ a ser decomposta.

Usamos o símbolo $ para marcar o fundo da pilha e também para marcar o final à direita da entrada. O funcionamento básico do analisador sintático segue os seguintes passos:

- O analisador sintático opera empilhando zero ou mais símbolos até que um *handle* β surja no topo da pilha.
- Nesse momento, β é reduzido para o lado esquerdo da produção apropriada.
- Esse ciclo é repetido até que tenha detectado um erro ou que a pilha contenha o símbolo de partida e a entrada esteja vazia

Existem quatro operações que o analisador sintático pode realizar:

1. empilhar
2. reduzir
3. aceitar
4. erro.

Essas ações são descritas abaixo:

- Numa ação de *empilhar*, o próximo símbolo de entrada é colocado no topo da pilha
- Numa ação de *reduzir*, o analisador sabe que o final à direita de um *handle* está no topo da pilha. Precisa, então, localizar o inicio à esquerda do *handle* dentro da pilha e decidir qual não-terminal irá substituir o *handle*.

- Numa ação de aceitar, o analisador anuncia o término com sucesso da operação de decomposição.
- Numa ação de *erro*, o analisador descobre que um erro sintático ocorreu e chama uma rotina de recuperação de erros.

Um analisador sintático ascendente com essas propriedades recebe a denominação de *analisador sintático de empilhar e reduzir*.

Por exemplo, a seqüência de ações tomadas por um *analisador sintático de empilhar e reduzir*, para reconhecer a cadeia de entrada ϖ = **id** + **id * id** é mostrada na figura 2.3.2.3.

PILHA	ENTRADA	AÇÃO
(1) $	**id + id * id** $	empilhar
(1) $ **id**	**+ id * id** $	reduzir por E → **id**
(1) $ E	**+ id * id** $	empilhar
(1) $ E +	**id * id** $	empilhar
(1) $ E + **id**	*** id** $	reduzir por E → **id**
(1) $ E + E	*** id** $	empilhar
(1) $ E + E *	**id** $	empilhar
(1) $ E + E * **id**	$	reduzir por E → **id**
(1) $ E + E * E	$	Reduzir por E → E * E
(1) $ E + E	$	Reduzir por E → E + E
(1) $ E	$	Aceitar

Figura 2.3.2.3. Configurações de uma analisador sintático de empilhar e reduzir para a cadeia de entrada **id** + **id** * **id**.

Observamos que o *handle* sempre aparece no topo da pilha.

2.2.2.7. Conflitos durante a Análise Sintática de empilhar e reduzir

Existem gramáticas livres de contexto para as quais a análise de empilhar e reduzir não pode ser usada. Nestas gramáticas, o analisador sintático pode atingir uma configuração na qual, mesmo conhecendo o conteúdo de toda a pilha e o próximo símbolo de entrada, não pode decidir entre empilhar ou reduzir (um *conflito empilhar*/*reduzir*) ou não pode decidir qual das diversas reduções alternativas realizar (um *conflito reduzir*/*reduzir*).

2.2.2.8. Analisadores Sintáticos LR

Estudaremos agora uma técnica eficiente de análise sintática ascendente, que pode ser usada para uma ampla classe de gramáticas livres de contexto. A técnica é chamada *análise sintática LR* (*k*), onde:

- O "L" significa varredura da entrada da esquerda para a direita (*left to right*)
- O "R", construção de uma derivação mais à direita (*rightmost derivation*)
- *k*, é o número de símbolos de entrada usados pelo *lookahead* que permitem tomar decisões na análise sintática. Quando (*k*) for omitido, assume-se que tem o valor 1.

A técnica de análise sintática LR é atrativa por uma série de razões:

- Analisadores sintáticos LR podem ser elaborados para reconhecer virtualmente todas as construções de linguagens de programação que podem ser reconhecidas por gramáticas livres de contexto.

- O método de decomposição LR é o mais geral dentre os métodos sem retrocesso de empilhar e reduzir conhecidos e pode ser implementado tão eficientemente quanto os demais métodos de empilhar e reduzir.
- A classe de gramáticas que podem ser utilizadas usando-se os métodos LR é um superconjunto próprio da classe de gramáticas que podem ser reconhecidas usando-se analisadores sintáticos preditivos.
- Um analisador sintático LR pode detectar um erro sintático tão cedo quanto possível numa varredura da entrada da esquerda para a direita.

A principal desvantagem deste método está no fato de bastante complexo construir um analisador sintático LR para uma gramática típica de linguagem de programação. Usa-se, em geral, uma ferramenta especializada – um gerador de analisadores LR, que tomando como entrada uma gramática livre de contexto para um linguagem, automaticamente produz um analisador sintático para a mesma.

Após ser discutida a operação básica de um analisador LR, serão apresentadas três técnicas para construção de tabelas sintáticas. A primeiro método, chamado LR simples (SLR), o mais fácil de implementar, é o menos poderoso dos três: pode falhar em produzir uma tabela sintática para algumas gramáticas em que os outros dois possam ter sucesso. O segundo método, chamado de LR canônico, é o mais poderoso e o mais caro. O terceiro, chamado LR *lookahead* (LALR), tem poder e custo intermediários.

2.2.2.9. O Algoritmo de Análise Sintática LR

A forma esquemática de um analisador sintático LR é mostrada na Figura 2.3.3.1. Consiste em uma entrada, uma saída, uma pilha, um programa diretor (denominado *analisador sintático LR*) e uma tabela sintática que possui duas partes (*ação e desvio*). O programa diretor é o mesmo para todos os três tipos de analisadores LR; somente a tabela sintática muda de uma analisador para outro.

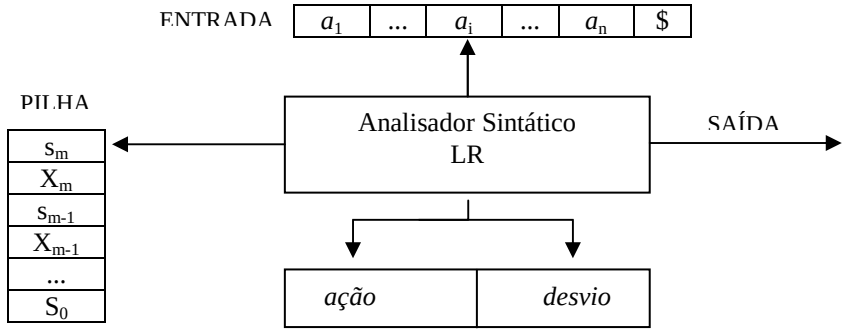


Figura 2.3.3.1. Modelo de um analisador LR

O analisador sintático processa uma sequência de *tokens* e usa uma pilha capaz de armazenar cadeias sob a forma $s_0X_1s_1X_2s_2..\ X_ms_m$, onde cada X_i é um símbolo gramatical, cada s_i, um símbolo chamado de *estado*, e s_m está no topo da pilha. Cada estado sumariza a informação contida na pilha abaixo dele e a combinação do estado no topo da pilha e o símbolo corrente de entrada é usada para indexar a tabela sintática e determinar qual ação será executada pelo analisador sintático.

A tabela sintática consiste em um par de funções sintáticas, *ação* e *desvio*, onde:

- A função *ação* toma toma como argumentos um estado e um *token*, realiza uma dentre as quatro ações definidas em um *analisador sintático de empilhar e reduzir*: empilhar, reduzir, aceitar ou erro.
- A função *desvio* toma como argumentos um estado e um símbolo gramatical, retornado um estado.

O programa diretor (analisador sintático LR) se comporta como se segue:

- Determina s_m, o estado correntemente no topo da pilha, e a_i, o símbolo corrente de entrada.
- Consulta, então, *ação*[s_m, a_i], a entrada da tabela de ações sintáticas para o estado s_m e a entrada a_i, que pode ter um dos quatro seguintes valores:

  1. empilhar s, onde s é um estado.
  2. reduzir através da produção gramatical $A \rightarrow \beta$.
  3. aceitar, e
  4. erro.

Uma *configuração* de um analisador sintático LR é um par, cujo primeiro componente é o conteúdo da pilha e cujo segundo componente é a entrada ainda não consumida:

$(s_0X_1s_1X_2s_2..\ X_ms_m\ ,\ a_ia_{i+1}..an\$)$

Esta configuração apresenta a forma sentencial à direita

$X_1X..\ X_ma_ia_{i+1}..an\$$

A tabela de ações é indexada por duas informações: a_i, o símbolo corrente de entrada, e s_m, um estado. A função *ação*[s_m ,a_i], faz com que o analisador sintático execute uma, dentre quatro ações possíveis a serem tomadas:

1. Se *ação*[s_m ,a_i] = empilhar s , o analisador executa um movimento de empilhar, entrando na configuração

   $(s_0X_1s_1X_2s_2..\ X_ms_ma_is\ ,\ a_{i+1}..an\$)$

   Aqui, o símbolo sintático empilhou tanto o símbolo corrente de entrada quanto o próximo estado s, que é dado por *ação*[s_m ,a_i]; a_{i+1} se torna o símbolo corrente de entrada.

2. Se *ação*[s_m ,a_i] = reduzir $A \rightarrow \beta$, o analisador sintático executa um movimento de redução, entrando na configuração

   $(s_0X_1s_1X_2s_2..\ X_{m-r}s_{m-r}As\ ,\ a_ia_{i+1}..an\$)$

   onde s = desvio[s_{m-r} , A] e *r* é o comprimento de β, o lado direito da produção. Aqui o analisador sintático remove primeiro 2*r* símbolos gramaticais para fora da pilha (*r* símbolos de estados e *r* símbolos gramaticais), expondo o estado s_{m-r}. Em seguida, empilha tanto A, o lado esquerdo da produção, quanto s, a entrada para desvio[s_{m-r} , A].

3. Se *ação*[s_m ,a_i] = aceitar, a análise sintática estará completa.

4. Se *ação*[s_m ,a_i] = erro, o analisador sintático descobriu um erro e chama um procedimento de recuperação de erros.

O algoritmo de análise sintática LR pode ser esquematizado da seguinte forma:

- Inicialmente, o analisador sintático possui s_0 na pilha, onde s_0 é o estado inicial, e *w*$ no *buffer* de entrada.
- O analisador sintático executa, então, o algoritmo abaixo, até que uma ação de aceitação ou de erro seja atingida.

```
fazer ip apontar para o primeiro símbolo de w$
repetir para sempre início
   seja s o estado ao topo da pilha e
   a o símbolo apontado por ip;
   se ação[s , a] = empilhar s', então início
                empilhar a e em seguida s' no topo da pilha;
                avançar ip para o próximo símbolo de entrada
   fim
    senão se ação[s , a] = reduzir A →β, então início
                desempilhar 2 * β símbolos para fora da pilha;
                seja s' o estado agora ao topo da pilha;
                empilhar A e em seguida desvio[s' , A];
   fim
senão se ação[s , a] = aceitar, então
                retornar
senão erro( )
fim
```

Figura x. O algoritmo do analisador sintático LR

2.2.2.10. O método SLR

A idéia central do método SLR é a construção de um autômata finito determinístico a partir de uma gramática.

Definimos um *item* LR(0), (*item*, simplificadamente) para uma gramática G, como sendo uma produção de G com um ponto em alguma de suas posições no lado direito.

Exemplo. A produção A → XYZ produz os quatro itens seguintes

A → •XYZ
A → X•YZ
A → XY•Z
A → XYZ•

A produção A → ∈ gera somente um item, A → •.

Um item indica quanto de uma produção já foi examinada num certo ponto do processo de análise sintática. O primeiro item no exemplo acima, por exemplo, indica que é esperada uma cadeia derivável a partir de XYZ. O segundo item indica que acabamos de examinar uma cadeia derivável a partir de X e que é esperada em seguida uma cadeia derivável a partir de YZ.

Uma coleção de conjuntos de itens LR(0) recebe a denominação de *uma coleção LR canônica*, e providencia a base para a construção de analisadores sintáticos SLR.

A construção de uma coleção canônica LR(0) para uma gramática pode ser feita a partir de uma *gramática aumentada* e duas funções, *fechamento* e *desvio*, definidas abaixo:

1. Definição de *Gramática Aumentada*

Se G for uma gramática com símbolo de partida S, então G', a *gramática aumentada* para G é G com um novo símbolo de partida, S', mais a produção S' → S.

2. *Definição da operação de Fechamento*

Se I for um conjunto de itens para uma gramática G, então o *fechamento*(I) é o conjunto de itens construídos a partir de I através da aplicação recursiva das seguintes regras:

1. Adicione cada item em I ao conjunto *fechamento*(I)
2. Seja $A \rightarrow \alpha \bullet B\beta$ um item em *fechamento*(I), e $B \rightarrow \gamma$ for uma produção para B. Adicione o item $B \rightarrow \bullet\gamma$ a I se o mesmo já não estiver lá.
3. Aplique novamente as regras a) e b) até que não seja mais possível adicionar novos itens ao conjunto *fechamento*(I).

A presença do item $A \rightarrow \alpha \bullet B\beta$ em *fechamento*(I) significa que, em algum ponto do processo de análise sintática, esperamos poder reconhecer na entrada uma cadeia derivável a partir de $B\beta$. Assim, se $B \rightarrow \gamma$ for uma produção, esperamos poder reconhecer uma cadeia derivável de γ àquele ponto., e por essa razão incluímos $B \rightarrow \bullet\gamma$ no *fechamento*(I).

Por exemplo, consideremos a seguinte gramática:

E' →E
E → E + T | T
T → T * F | F
F → (E) | **id**

Se I for o conjunto de um item dado por { [E' → E] }, então *fechamento*(I) contém os itens

E' → •E
E → •E + T
E → •T
T → •T * F
T → •F
F → •(E)
F → •**id**

E' → •E é colocado em *fechamento*(I) pela regra (a). Os outros itens são adicionadas a *fechamento*(I) pela regra (b).

3. *Definição da operação de Desvio*

Seja I um conjunto de itens, e X um símbolo gramatical. *Desvio*(I,X) é definida como sendo o fechamento do conjunto de todos os itens [$A \rightarrow \alpha X \bullet \beta$] tais que [$A \rightarrow \alpha \bullet X\beta$] esteja em I.

Exemplo. Se I for o conjunto de dois items { [E' →E], [E → E + T] }, estão o conjunto *desvio*(I,+) consiste nos seguintes itens:

E → E +• T
T → •T * F
T → •F
F → •(E)
F → •**id**

Para cada gramática conjunto de itens I a função *desvio* define um autômato finito determinístico que reconhece todos os prefixos de I.

Tendo definido uma *gramática aumentada* e as funções de *fechamento* e *desvio*, podemos agora definir a construção de uma *coleção canônica de conjuntos de itens* LR(0) para uma *gramática aumentada* G'. Essa construção é feita através de acordo com o algoritmo esboçado na figura.

Procedimento *itens* (G');
Início
C:= { fechamento ([S' → S]) } ; // *C* contém inicialmente o conjunto de itens I_0
Repetir
  Para cada conjunto de itens I_x em *C* e cada símbolo gramatical X
    tal que *desvio*(I,X) não seja vazio e não esteja em C **faça**
      Crie um novo conjunto de itens I_j ;
      Inclua em I_j *desvio*(I,X) ;
      Inclua em *C* *desvio*(I,X) ;
Até que não haja mais conjuntos de itens a serem incluídos em *C.*
Fim

Figura . Algoritmo para construir a coleção canônica de um conjunto de itens LR (0)

Por exemplo, a coleção canônica de conjuntos de itens LR(0) para a gramática usada nos exemplos anteriores é mostrada na figura 2.3.4.1. A função *desvio* para este conjunto de itens é mostrada como o diagrama de transições de um autômato finito determinístico.

I_0 : E' → •E
E → •E + T
E → •T
T → •T * F
T → •F
F → •(E)
F → •**id**

I_1 : E' → E•
E → E• + T

I_2 : E → T•
E → T• * F

I_3 : T → F•

I_4 : F → (•E)
E → •E + T
E → •T
T → •T * F
T → •F
F → • (E)
F → •**id**

I_5 : F → **id**•

I_6 : E → E + •T
T → •T * F
T → •F
F → •(E)
F → •**id**

I_7 : T → T * •F
F → •(E)
F → •**id**

I_8 F → (E•)
E → E• + T

I_9 :E → E + T•
T → T• * F

I_{10}: T → T * F•

I_{11}: F → (E)•

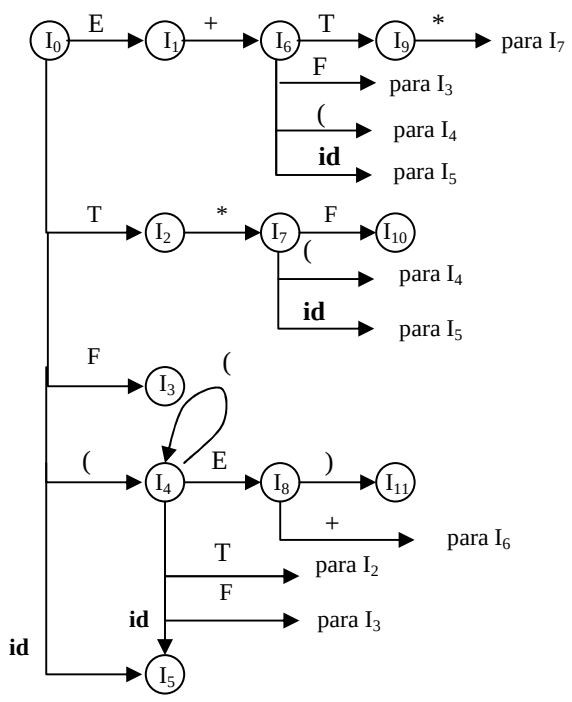


Figura 2.3.4.1. Coleção canônica LR(0) para a gramática de expressões aumentada, e diagrama de transições construído a partir dos itens

A construção de Tabelas Sintáticas SLR envolve dois requisitos:

1. Dada uma gramática G, devemos aumentá-la de forma a produzir G' a partir de G.
2. Construir *C*, a coleção canônica de conjuntos de itens para G'.

A construção das funções *ação* e *desvio* envolvem o conhecimento prévio de uma função auxiliar associada à gramática, FOLLOW, definida um não-terminal da gramática da seguinte forma:

1. Se A = S, colocamos $ em FOLLOW(S), onde S é o símbolo de partida e $ o marcador de fim de entrada à direita.
2. Se existir uma produção $A \rightarrow \alpha B \beta$, então tudo em FIRST(β), exceto $\in$, é colocado em FOLLOW(B)
3. Se existir uma produção $A \rightarrow \alpha B$ ou uma produção $A \rightarrow \alpha B \beta$ onde FIRST(β) contém $\in$, então tudo em FOLLOW(A) está em FOLLOW(B).

Exemplo. Consideremos a seguinte gramática:

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \in$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \in$
$F \rightarrow (E) \mid$ **id**

Então:

FIRST(E) = FIRST(T) = FIRST(F) = { (, **id** }
FIRST(E') = { *, $\in$ }
FIRST(T') = { *, $\in$ }
FOLLOW(E) = FOLLOW(E') = {), $ }
FOLLOW(T) = FOLLOW(T') = { +,), $ }
FOLLOW(F) = { +, *,), $ }

As funções *ação* e *desvio* são definidas através dos seguintes passos:

1. Construir C = { I_0 , I_1 , ... , I_n }, a coleção canônica de itens LR(0) para G'.
2. O estado i é construído a partir de do conjunto de itens I_i . As ações sintáticas para o estado i são determinadas como se segue:

   a) Se [$A \rightarrow \alpha \bullet a \beta$] estiver em I_i e *desvio*(I_i ., *a*) = I_j, então *ação*[i , *a*] = "empilhar *j*". Aqui, *a* precisa ser um terminal.
   b) Se [$A \rightarrow \alpha \bullet$] estiver em I_i então estabelecer *ação* [i, *a*] em "reduzir através de $A \rightarrow \alpha$, para todo *a* em FOLLOW(A); aqui, A não pode ser S'.
   c) Se [$S' \rightarrow S \bullet$] estiver em I_i então fazer *ação* [i , $] igual a "aceitar".

3. As transições de desvio para o estado *i* são construídas para todos os não-terminais A usando-se a seguinte regra: se *desvio*(I_i , A) = I_j , então *desvio*(i , A) = j.
4. Todas as entradas não definidas pelas regras (2) e (3) são tornadas "erro"
5. O estado inicial do analisador sintático é aquele construído a partir do conjunto de itens contendo [$S' \rightarrow S$].

A tabela sintática contendo as funções de *ação* e *desvio* é chamada de tabela SLR(1) para G. Um analisador sintático LR usando uma tabela SLR(1) para G é denominado um analisador sintático SLR(1) para G e uma gramática tendo uma tabela sintática SLR(1) é dita ser SLR(1).

Por exemplo, consideremos a seguinte gramática:

$E' \rightarrow E$
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid$ **id**

Temos então que:

- O item F → •(E) do conjunto de itens I_0 dá origem à entrada *ação*[0, ( ] = empilhar 4;
- O item F → •**id** do conjunto de itens I_0 dá origem à entrada *ação*[0, **id**] = empilhar 5;
- Os outros itens em I_0 não produzem ações;
- O item E' → E• em I_1 produz *ação*[1, $] = aceitar.
- Para o item E → T• em I_2 verificamos que FOLLOW(E) = { $, +,) }, isso dá origem às entradas *ação*[2, $] = *ação*[2, +] = *ação*[2,)] = reduzir E → T.

Toda gramática SLR(1) é não-ambígua, mas existem muitas gramáticas não-ambíguas que não são SLR(1).

Por exemplo, consideremos a seguinte gramática:

S → L = R
S → R
L → *R
L → **id**
R → L

A coleção canônica de conjuntos de itens LR(0) para essa gramática é mostrada na figura :

I_0 : S' → •S
S → •L = R
S → •R
L → •* R
L → •**id**
R → •L

I_1 : S' → S•

I_2 : S → L• = R
R → L•

I_3 : S → R•

I_4 : L → * •R
R → •L
L → •* R
L → •**id**

I_5 : L → **id**•

I_6 : S → L = •R
R → •L
L → •* R
L → •**id**

I_7 : L → * R•

I_8 R → L•

I_9 : S → L = R•

Figura 2.3.4.2. Coleção canônica de items LR(0) para a gramática do exemplo

Consideremos o conjunto de itens I_2 . O primeiro item nesse conjunto faz com que *ação*[2, =] seja igual a "empilhar 6". Uma vez que FOLLOW(R) contém =, o segundo item faz *ação*[2, =] igual a "reduzir por R → L". Dessa forma, a entrada *ação*[2, =] é multiplamente definida. Uma vez que existe tanto uma ação de empilhar quanto uma ação de reduzir em *ação* [2, =], o estado 2 possui um conflito de empilhar/reduzir para o símbolo de entrada =. A gramática citada no exemplo não é ambígua, mas o método de construção SLR produz para a mesma uma tabela sintática com conflitos nas ações sintáticas.

Os métodos LR canônico e LALR irão funcionar para um conjunto maior de gramáticas. No entanto, deve ser aqui assinalado que existem gramáticas não-ambíguas para as quais qualquer método de construção de analisadores sintáticos LR irá produzir tabelas sintáticas com conflitos nas ações sintáticas. Felizmente, tais gramáticas geralmente podem ser evitadas nas aplicações das linguagens de programação.

2.2.2.11. Construindo tabelas sintáticas LR canônicas

No método SLR, o estado *i* chama pela redução $A \rightarrow \alpha$ se o conjunto de itens I_i contiver o item [$A \rightarrow \alpha\bullet$] e *a* estiver em FOLLOW(A). Em algumas situações, entretanto, quando o estado *i* aparece no topo da pilha, o prefixo viável $\beta\alpha$ será tal que β não poderá ser seguido por *a* numa forma sentencial à direita. Por conseguinte, uma redução através de $A \rightarrow \alpha$ será inválida à entrada *a*.

Por exemplo, na figura 2.3.4.2, no estado 2, tínhamos o item $R \rightarrow L\bullet$, que corresponderia a $A \rightarrow \alpha$ acima, e *a*, que poderia ser o sinal de =, que está em FOLLOW(R). Dessa forma, o analisador sintático chama pela redução $R \rightarrow L$ no estado 2 com = como próxima entrada (a ação de empilhar também é chamada por causa do item $S \rightarrow L\bullet = R$ no estado 2). Entretanto, não existe, na gramática que gera a coleção canônica da figura 2.3.4.2, uma forma sentencial à direita que comece por R = ... Consequentemente, o estado 2, que é o estado correspondente ao prefixo viável L somente, não poderia chamar pela redução daquele L para R.

Pensando nesse problema, podemos então pensar na possibilidade de se carregar mais informações dentro de um estado, as quais irão proscrever algumas dessas reduções inválidas por $A \rightarrow \alpha$. A idéia é a redefinição dos itens, de forma que os mesmos contenham uma informação adicional: um símbolo terminal como um segundo componente.

A forma geral de um item, com a adição dessa nova unidade de informação, é definida através do formato [$A \rightarrow \alpha\bullet\beta$, *a*], onde $A \rightarrow \alpha\beta$ é uma produção e *a* um terminal ou marcador de fim à direita $. Um item definido nesse formato recebe a denominação de um *item* LR(1), onde o "1" se refere ao comprimento do segundo componente do item, o *lookahead*.

Os *lookaheads* de um item não possuem efeito em itens na forma [$A \rightarrow \alpha\bullet\beta$, *a*], onde β não é $\in$. Os itens no formato [$A \rightarrow \alpha\bullet$, *a*], no entanto, informam que deve ser efetuada uma redução através de $A \rightarrow \alpha$ somente se o símbolo de entrada for *a*.

O método para construção da coleção da coleção de conjuntos de itens LR(1) segue essencialmente o mesmo esquema que o utilizado para a construção da coleção canônica de conjuntos de itens LR(0). A diferença reside apenas numa outra definição das funções de *fechamento* e *desvio* (Figura).

```
Função fechamento (I: item LR(1) );
Início
  Repetir
    Para cada item [A → α•Bβ, a ] em I,
          cada produção B → γ em G',
          e cada terminal b em FIRST(β a)
          tal que [ B → •γ, b ] não está em I faça
          incluir [ B → •γ, b ] em I;
  até que não possam ser adicionados mais itens a I;
  retornar I
fim;


função desvio ( I , X )
Início
  Seja J o conjunto de itens [A → αX•β, a ] tais que
  [A → α•Xβ, a ] esteja em I;
  retornar fechamento (J)
fim;
```

Figura . Nova definição das funções de *fechamento* e *desvio* para o método LR canônico

Por exemplo, consideremos a seguinte gramática:

$S' \rightarrow S$
$S \rightarrow CC$
$C \rightarrow cC \mid d$

Iniciamos pelo cálculo do *fechamento* de { [S' → •S, $] }. Para tanto, confrontamos esse item com o item [A → α•Bβ, *a*] no procedimento *fechamento*, do qual tiramos: A = S', α = ∈, B = S, β = ∈ e *a* = $.

A função *fechamento* nos diz para adicionar o item [B → •γ , *b*] para cada produção B → γ e terminal *b* em FIRST(β*a*). Em termos da presente gramática, B → γ precisa ser S → CC, e uma vez que β é ∈ e *a* é $, *b* pode somente ser $. Consequentemente, adicionamos [S → •CC, $].

Continuamos a computar o fechamento, adicionando todos os itens [C → •γ , *b*] para *b* em FIRST(C$). Isto é, confrontando-se [S → •CC, $] com [A → α•Bβ, *a*], temos A = S, α = ∈, B = C, β = C e *a* = $. Como C não deriva a cadeia vazia, FIRST(C$) = FIRST(c). Uma vez que FIRST(C) contém os terminais *c* e *d*, adicionamos os itens [C → •cC, c], [C → •cC, d], [C → •d, c] e [C → •cC, d]. Nenhum dos novos itens possui um não-terminal imediatamente à direita do ponto, e , então, completamos nosso primeiro conjunto de itens LR(1). O conjunto inicial de itens é:

I_0 : S' → •S, $
S → •CC, $
C → •cC, c/d
C → •d, c/d

Os colchetes foram omitidos por conveniência de notação. Usamos a forma [C → •cC, c/d] como abreviação para dois itens [C → •cC, c] e [C → •cC, d].

Procedemos agora para a computação de *desvio*(I_0, X) para os vários valores de X. Para X =S, precisamos fechar o item [S' → •S, $]. Nenhum fechamento adicional é possível, uma vez que o ponto está à extremidade direita. Por conseguinte, temos o próximo conjunto de itens:

I_1 : S' → S•, $

Para X = C, fechamos [S → C•C, $]. Adicionamos as produções-C com segundo componente $ e, então, não podemos adicionar mais, produzindo:

I_2 : S → C•C, $
C → •cC, $
C → •d, $

Em seguida, seja X = c. Precisamos fechar [C → c•C, c/d]. Adicionamos as produções-C com o segundo componente *c*/*d*, produzindo:

I_3 : C → c•C, c/d
C → •cC, c/d
C → •d, c/d

Finalmente, para X = d, obtemos o conjunto de itens:

I_4 : C → d•, c/d

Terminamos de considerar *desvio* em I_0. Não temos conjuntos de I_1, mas I_2 possui *desvios* em C, *c* e em *d*. Em C obtemos:

I_5 : S → CC•, $

Em *c*, tomamos o fechamento de [C → c•C, $], para obter:

I_6 : C → c•C, $
C → •cC, $
C → •d, $

Continuando com a função *desvio* para I_2, *desvio*(I_2, *d*) é achado ser:

$I_7 : C \rightarrow d\bullet, \$$

Voltando agora para I_3, os *desvios* de I_3 para *c* e *d* são I_3 e I_4, respectivamente, e *desvio*(I_6, C) é

$I_8 : C \rightarrow cC\bullet, c/d$

I_4 e I_5 não possuem *desvio*s. Os *desvios* de I_6 em *c* e *d* são I_6 e I_7, respectivamente, e *desvio*(I_6, C) é:

$I_9 : C \rightarrow cC\bullet, \$$

Os conjuntos restantes de itens não produzem *desvios*, e, então, terminamos. A Figura xxx mostra os dez conjuntos de itens com seus respectivos *desvios*.

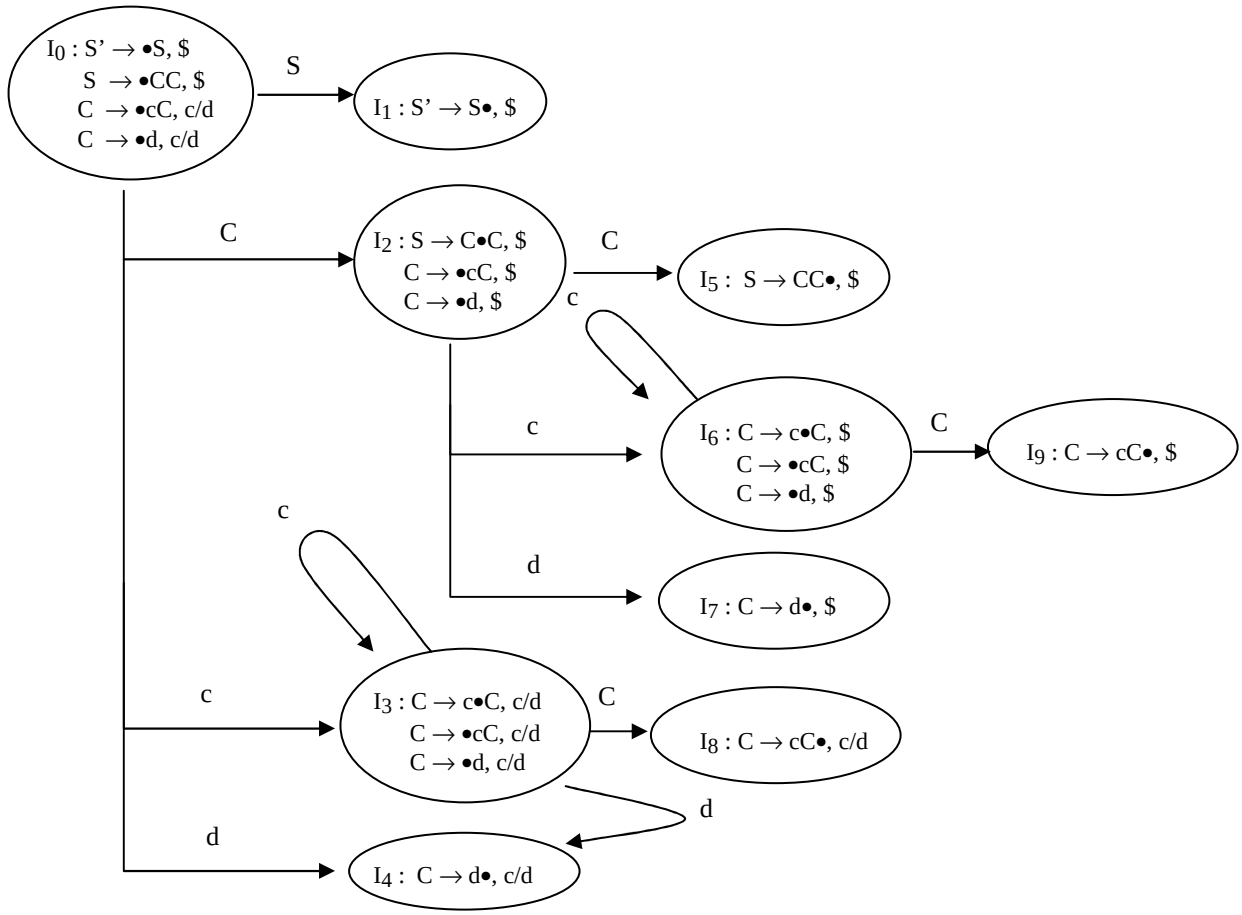


Figura 2.3.5.1. O grafo *desvio* para a gramática do exemplo

Fornecemos as regras pelas quais as funções sintáticas de ação e desvio são construídas a partir dos conjuntos de itens LR(1). As funções de ação e desvio são representadas por uma tabela como antes, a única diferença está nos valores das entradas.

O algoritmo para construção da tabela LR canônica segue os seguintes passos:

1. Construir C = { I_0, I_1, ..., I_n }, a coleção de conjuntos de itens LR(1) para G'
2. O estado *i* do analisador sintático é construído a partir de I_1. As funções sintáticas para o estado *i* são determinadas como se segue:

   a) Se [A → α•*a*β, *b*] estiver em I_i e *desvio*(I_i, *a*) = I_j, então fazer *ação*[i,*a*] igual a "empilhar *j*". Aqui, *a* é exigido ser um terminal.
   b) Se [A → α•, *a*] estiver em I_i A ≠ S', então fazer *ação*[i,*a*] igual a "reduzir A → α".
   c) Se [S' → S•, $] estiver em I_i A ≠ S', então fazer *ação*[i,$] igual a "aceitar".

3. As transições *desvio* para o estado *i* são determinadas como se segue : se *desvio* (I_i, A) = I_j, então *desvio*[i, *a*] = *j*.
4. Todas as entradas não definidas pelas regras (2) e (3) são tornadas "erro".
5. O estado inicial do analisador sintático é aquele construído a partir do subconjunto contendo o item [S' → •S, $].

A tabela formada a partir das funções sintáticas de ação e desvio, derivadas do algoritmo acima, recebe a denominação de *tabela sintática LR* (1) *canônica*. Um analisador sintático que faz uso dessa tabela é chamado de um analisador sintático LR(1) canônico. Se a função sintática de ação não possui entradas multiplamente definidas, então a gramática usada na construção da tabela sintática LR é dita ser uma *gramática* LR(1).

Cada gramática SLR(1) possui associada uma gramática LR(1), mas para uma gramática SLR(1) o analisador sintático LR pode ter mais estados do que o analisador sintático SLR para a mesma gramática.

2.2.2.12. Analisador sintático LALR

Estudaremos agora nosso último método para construção de analisadores sintáticos, a técnica LALR (*lookahead* LR). Este método é freqüentemente usado na prática porque as tabelas obtidas são menores do que as tabelas LR canônicas e, além do mais, a maioria das construções sintáticas comuns às linguagens de programação podem ser expressas convenientemente por gramáticas LALR.

A construção de tabelas LALR pode ser feita a partir de um conjunto de itens LR(1), através da combinação de estados contendo *núcleos comuns*; i.e. os conjuntos de itens tendo que possuem em comum os primeiros componentes.

Por exemplo, na figura 2.3.5.1 os estados I_4 e I_7 podem ser combinados, pois possuem um núcleo comum dado por { C → •d }. Da mesma forma, podem ser combinados os estados I_3 e I_6 , com núcleo comum { C → c•C, C → •cC , C → •d }, e I_8 e I_9 , com núcleo comum { C → •cC }.

A união de conjuntos de itens com núcleos comuns não gera conflitos do tipo empilhar/reduzir, a menos que tais conflitos já estivessem presentes em algum dos estados originais. Suponhamos, por hipótese, que na união de dois itens ocorra um conflito no *lookahead a* porque existe um item [A → α•, *a*] chamando por uma redução através de A → α, e existe um outro item [B → β•*a*γ , *b*] chamando por um empilhamento. Então, algum conjunto de itens a partir do quais a união foi realizada possui o item [A → α•, *a*] e, como os núcleos de todos esses estados são os mesmos, o conjunto também deve ter um item [B → β•*a*γ , *c*], para algum *c*. Assim, o estado unido já possui o conflito de empilhar/reduzir em *a*. Conclusão: a gramática já não era LR(1) como assumido.

É possível, entretanto, que a combinação de dois itens produza um conflito reduzir/reduzir, como o seguinte exemplo mostra.

Por exemplo, consideremos a gramática

S' → S
S → *a*A*d* | *bbd* | *b*B*e* | *b*A*e*
A → *c*
B → *c*

Essa gramática é LR(1), e contém os conjuntos de itens { [A → *c*•, *d*], [B → *c*•, *e*] } e {[A → *c*•, *e*], [B → *c*•, *d*] }. Nenhum desses conjuntos gera um conflito e seus núcleos são os mesmos. Sua união, entretanto, dada por

A → •*c*, *d* /e
B → •*c*, *d* /e

gera um conflito reduzir/reduzir, uma vez que as duas reduções, A → *c* e B → *c* são chamadas às entradas *d* e *e*.

O algoritmo para construção de uma tabela sintática LALR a partir de um conjunto de itens LR(1) segue os seguintes passos:

1. Construir C = { I_0, I_1, ..., I_n }, a coleção de conjuntos de itens LR(1) para G'
2. Para cada núcleo presente entre os conjuntos de itens LR(1), encontrar todos os conjuntos que tenham o mesmo núcleo e substituí-los pela sua união.
3. Seja C' = { J_0, J_1, ..., J_m }, a coleção de conjuntos de itens LR(1) resultante. Construir as ações sintáticas para o estado *i* a partir de J_i da mesma maneira que no algoritmo de construção de um tabela LR canônica.
4. Construir a tabela *desvio* Bcomo se segue. Se J for a união de um ou mais conjuntos de itens LR(1), isto é, J = $I_1 \cup I_2 \cup .... \cup I_k$, então os núcleos de *desvio*(I_1, X), *desvio*(I_2, X), ..., *desvio*(I_k, X) são os mesmos, uma vez que I_1 , I_2 , ... , I_k, possuem todos o mesmo núcleo. Seja K a união de todos os conjuntos de itens que tenham o mesmo núcleo que *desvio*(I_i, X). Então *desvio*(J,X) = K.

A tabela produzida por esse algoritmo recebe a denominação de *tabela sintática* LALR para G. Se não existem conflitos de ações sintáticas, então a gramática é dita LALR(1). A coleção de conjuntos de itens construída no passo (3) é chamada de coleção LALR(1).

Exemplo. Consideremos a gramática cujo grafo de *desvio* foi mostrado na figura 14. As funções de ação e desvio LALR para os conjuntos condensados de itens são mostrados na figura 15.

ESTADO	*Ação*			*desvio*	
	c	*d*	$	S	C
0	s36	s47		1	2
1			ac.		
2	s36	S47			5
36	s36	S47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Figura 2.3.6.1. Tabela sintática LALR para a gramática cujo grafo de *desvio* foi mostrado na figura 14

Para uma comparação do tamanho do espaço, as tabelas SLR e LALR para uma gramática possuem o mesmo número de estados, e esse número é tipicamente de várias centenas para uma linguagem como Pascal. A tabela LR canônica teria tipicamente vários milhares de estados para uma linguagem com o mesmo tamanho. Consequentemente, é mais fácil e econômico construir tabelas SLR e LALR do que tabelas LR canônicas.

2.2.2.13. Usando Gramáticas Ambíguas

Qualquer gramática ambígua falha em ser LR e consequentemente não está em quaisquer das classes estudadas SLR, LR *canônica* e LALR, pelo fato de gerar tabelas sintáticas com entradas multiplamente definidas. Certos tipos de gramáticas ambíguas, no entanto, são úteis na especificação e implementação de linguagens, e, por esse motivo, devem ser consideradas.

Em alguns casos podemos especificar regras de não-ambiguidade que permitam somente uma árvore gramatical para cada sentença, de forma que a especificação global da linguagem se mantenha também não-ambígua. Através dessas regras o analisador sintático pode, por exemplo, quando encontrar um conflito empilhar/reduzir na construção da tabela sintática, optar por armazenar a ação de "empilhar" em detrimento da ação de "reduzir", construindo assim uma tabela sintática não-ambígua.

Um caso particular destes conceitos pode ser ilustrado através do tratamento dado à seguinte gramática, que tem a ambiguidade do *else*-vazio:

cmd → **if** *exp* **then** *cmd* **else** *cmd*
| **if** *exp* **then** *cmd* **else** *cmd*
| **outro**

Para simplificar a discussão, consideremos uma abstração da gramática acima, onde *i* está no lugar de **if** *expr* **then**, *e* no de **else** e *a* no de "todas as demais produções.

Usando essa abstração temos que a gramática acima, adicionada da produção S' → S, pode ser rescrita da seguinte forma:

S' → S
S → *i*S*e*S | *i*S | *a*

Os conjuntos de itens LR(0) para essa gramática são mostrados na figura 2.3.7.1. A ambiguidade na gramática dá origem a um conflito de empilhar/reduzir no estador I_4: nesse, o item S → *i*S•*e*S chama por um empilhamento de *e* e, uma vez que FOLLOW(S) = { *e*, $ } ao mesmo tempo em que o item S → *i*S• chama por uma redução através de S → *i*S à entrada *e*.

I_0 : S' → •S
S → •*i*S*e*S
S → •*i*S
S → • *a*

I_1 : S' → S•

I_2 : S → *i*•S*e*S
S → *i*•S
S → •*i*S*e*S
S → •*i*S
S → •*a*

I_3 : S → *a*•

I_4 : S → *i*S•*e*S
S → *i*S•

I_5 : S → *i*S*e*•S
S → •*i*S*e*S
S → •*i*S
S → •*a*

I_6 : S → *i*S*e*S•

Figura 2.3.7.1. Estados LR(0) para a gramática abstrata aumentada do "else-vazio"

A análise desse caso particular nos leva à conclusão de que o conflito em I_4 deve ser resolvido em favor de empilhar à entrada *e*, pois o **else** está associado ao **then** prévio.

A tabela sintática SLR, para os conjuntos de itens da figura 2.3.7.1, construída de acordo com essa regra, é mostrada na Tabela 2.3.7.2.

ESTADO	*Ação*				*desvio*
	i	*e*	*a*	$	S
0	s2		s3		1
1			.	ac.	
2	s2		s3		4
3		r3		r3	
4		s5		r2	
5	s2		s3		6
6		r1		r1	

Tabela 2.3.7.2. Tabela sintática SLR para a gramática abstrata do "else-vazio"

2.2.2.14. Uma questão: LL x LR

Existe uma significativa diferença as gramáticas LL e as gramáticas LR. Para uma gramática ser LR(*k*), por exemplo, é necessário que seja possível o reconhecimento do lado direito de uma produção tendo visto tudo do que foi derivado a partir daquele lado direito mais o esquadrinhamento antecipado de *k* símbolos de entrada (isto é, *k* símbolos *lookahead*), exigência muito menos restritiva do que aquela para uma gramática LL(*k*), onde é necessário reconhecimento de uma produção examinando somente os primeiro *k* símbolos que seu lado direito pode derivar. Como consequência disso, as gramáticas LR podem descrever mais linguagens do que as gramáticas LL.

A prática tem mostrado que os analisadores sintáticos LALR(1) são poderosos o suficiente para serem usados na maioria dos compiladores modernos. Analisadores Sintáticos LR(1) normalmente não são usados por conterem muitos estados e serem ineficientes.

2.2.3. A Análise Semântica

Antes de traduzir os comandos de um programa fonte para uma sequência de instruções correspondentes de um programa objeto, o compilador precisa verificar se um conjunto definido de regras semânticas está sendo satisfeito. A verificação destas regras é feita durante a análise semântica.

Nesta fase, o compilador armazena na tabela de símbolos um conjunto de informações que serão utilizadas para validar as regras semânticas, e também para gerar código objeto. O conjunto de informações coletadas para uma construção de linguagem é definido como o conjunto de *atributos semânticos* associados à essa construção. Um tipo, uma cadeia de caracteres, ou uma localização de memória são exemplos de *atributos*.

As *definições dirigidas pela sintaxe* especificam o formato para a tradução das construções da linguagem por meio de atributos associados aos seus componentes sintáticos. Os *esquemas de tradução* especificam essas traduções. Ambos os formalismos serão discutidos nesse estudo.

2.2.3.1. Atributos semânticos

Um atributo semântico pode representar uma cadeia, um número, um tipo, uma localização de memória, ou qualquer outra informação pertinente a uma construção da linguagem. O valor para um atributo semântico em um nó da árvore gramatical é definido por uma regra semântica associada à produção usada naquele nó. Dois tipos distintos de atributos semânticos serão considerados nesse estudo: os atributos sintetizados e os atributos herdados.

Um atributo é *sintetizado* se o seu valor num nó da árvore gramatical é determinado a partir dos valores dos atributos dos filhos daquele nó.

Um atributo é *herdado* se o seu valor num nó da árvore gramatical é determinado a partir dos valores dos atributos dos irmãos e do pai daquele nó.

2.2.3.2. Definições dirigidas pela sintaxe e esquemas de tradução

As definições dirigidas pela sintaxe usam uma gramática livre de contexto para especificar a estrutura sintática de um programa fonte. A cada símbolo da gramática é associado um conjunto de atributos, e a cada produção é associado um conjunto de *regras semânticas*. O conjunto formado por uma gramática livre de contexto e um conjunto de *regras semânticas* constituem as *definições dirigidas pela sintaxe*.

Em uma *definição dirigida pela sintaxe*, cada produção $A \rightarrow \alpha$ tem associada à si conjunto de regras semânticas da forma $b := f(c_1, c_2, ..., c_k)$, onde *f* é uma função e ocorre uma das duas situações seguintes, mas não ambas:

1. *b* é um atributo sintetizado de A e $c_1, c_2, ..., c_k$ são atributos semânticos pertencentes aos símbolos da produção ou

2. *b* é um atributo herdado, pertencente a um dos símbolos do lado direito da produção, e c_1, c_2, ..., c_k são atributos semânticos pertencentes aos símbolos da produção.

Em ambos os casos, dizemos que o atributo *b* depende dos atributos c_1, c_2, ..., c_k.

Nesse formalismo, assume-se que os terminais tenham somente atributos sintetizados, pois a definição não fornece regras semânticas para terminais. Além disso, assume-se que o símbolo inicial não contém quaisquer atributos herdados, a menos que seja estabelecido o contrário.

Por exemplo, a tabela mostrada na Figura 2.2.3.2.1 mostra um conjunto de definições dirigidas pela sintaxe para uma linguagem que descreve uma calculadora de mesa. É associado um atributo sintetizado, *val*, com valor do tipo inteiro, a cada um dos não-terminais *E*, *T* e *F*. Para cada uma das produções, a regra semântica computa o atributo *val* para os não-terminais do lado direito.

Produção	Regras Semânticas
L → E**n**	*Imprimir* (E.*val*)
E → E + T	E.*val* := E.*val* + T.*val*
E → T	E.*val* := T.*val*
T → T * F	T.*val* := T.*val* x F.*val*
T → F	T.*val* := F.*val*
F → (E)	F.*val* := E.*val*
F → **digito**	F.*val* := **digito**.*lexval*

Figura 2.2.3.2.1. Definições dirigidas pela sintaxe de uma linguagem que descreve uma calculadora de mesa simples

O *token* **dígito** possui um atributo sintetizado *lexval*, cujo valor deverá ser fornecido pelo analisador léxico. A regra associada à produção L → E **n** para o não-terminal inicial L é um procedimento que imprime o valor da expressão aritmética gerada por E.

Um *esquema de tradução* é uma gramática livre de contexto na qual fragmentos de programas, chamados de *ações semânticas*, são inseridos nos lados direitos das produções. Um esquema de tradução gera uma saída para cada sentença *x*, que tenha sido gerada pela gramática, através da execução das ações na ordem especificada durante o caminhamento em profundidade de uma árvore gramatical para *x*.

Com esses dois formalismos, as definições dirigidas pela sintaxe e os esquemas de tradução, é possível efetuar a análise sintática de um fluxo de *tokens* de entrada, construir a árvore gramatical correspondente à esse fluxo de *tokens*, e também percorrer a árvore de forma a avaliar as regras semânticas definidas para cada nó. A avaliação das regras semânticas pode: gerar código objeto (essa decisão fica a cargo do projetista do compilador), salvar informações numa tabela de símbolos, emitir mensagens de erro e tratar da manutenção de estruturas de dados auxiliares ao processo de avaliação semântica, como uma pilha de funções.

2.2.3.3. Grafo de Dependências

As interdependências entre os atributos herdados e sintetizados nos nós da árvore gramatical podem ser mostradas através de uma estrutura chamada de *grafo de dependências.*

Um grafo de dependências possui um nó para cada atributo semântico, e uma aresta a partir de cada nó c_i para o nó *b*, se o atributo *b* depender do atributo c_i. Assim, a construção de um grafo de dependências para uma árvore gramatical, requer que cada regra semântica seja colocada sob a forma b := f(c_1, c_2, ..., c_k).

Por exemplo, suponhamos que A.*a* := f(X.*x*, Y.*y*) seja uma regra semântica para a produção A → XY. Essa regra define o atributo sintetizado A.*a* que depende dos atributos X.*x* e Y.*y*. Se essa produção vier a ser usada na árvore gramatical, existirão, no grafo de dependências, três nós, A.*a*, X.*x* e Y.*y*, com uma aresta para A.*a* a partir de X.*x*, uma vez que A.*a* depende de X.*x*, e uma aresta para A.*a*, a partir de Y.*y*, uma vez que A.*a* também depende de Y.*y* (Figura 2.2.3.3.1).

Os três nós do grafo de dependências, marcados por •, representam os atributos sintetizados A.*a*, X.*x* e *Y.y* na árvore gramatical. A aresta para A.*a* a partir de X.*x* mostra que A.*a* também depende de X.*x*. As linhas pontilhadas representam a árvore gramatical e não fazem parte do grafo de dependências.

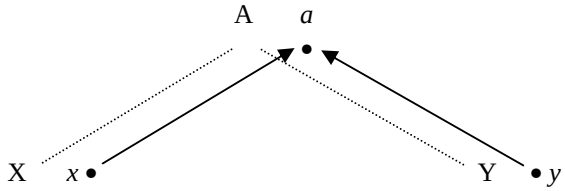


Figura 2.2.3.3.1. A.*a* é sintetizado a partir de X.*x* e *Y.y*.

Uma *classificação topológica* de um grafo acíclico dirigido é uma ordenação m_1, m_2, ..., m_k dos nós do grafo, de tal forma que as arestas saem dos primeiros nós da ordenação para os últimos, isto é, se $m_i \rightarrow m_j$ é uma aresta de m_i para m_j, então m_i aparece antes de m_j na ordenação.

Qualquer classificação topológica de um grafo de dependências fornece uma ordem válida a partir da qual as regras semânticas podem ser avaliadas, ou seja, na classificação topológica, os atributos c_1, c_2, ..., c_k, dos quais uma regra semântica b := $f(c_1, c_2, ..., c_k)$ depende, estarão disponíveis num nó antes de *f* ser avaliada.

As traduções especificadas por definições dirigidas pela sintaxe podem ser tornadas precisas como segue. A gramática é usada para construir uma árvore gramatical para o programa fonte. O grafo de dependências é construído conforme discutido acima. A partir de uma classificação topológica do grafo de dependências, obtemos uma ordem de avaliação para as regras semânticas.

2.2.3.4. Verificações Estáticas

Um compilador precisa verificar se o programa fonte segue as convenções sintáticas e semânticas da linguagem fonte. Essa tarefa, chamada de *verificação* estática, assegura que erros sintáticos e semânticos serão detectados e reportados.

Muitos compiladores Pascal combinam a verificação estática e a geração de código intermediário com a análise sintática. A verificação estática pode, no entanto, ser realizada entre a análise sintática e a geração de código como indicado na Figura 2.2.3.4.1. O uso dessa estratégia, embora diminua um pouco a eficiência do compilador, assegura a vantagem de separar claramente as etapas de análise sintática e análise semântica, possibitando que cada uma dessas etapas possa ser implementada de forma independente.

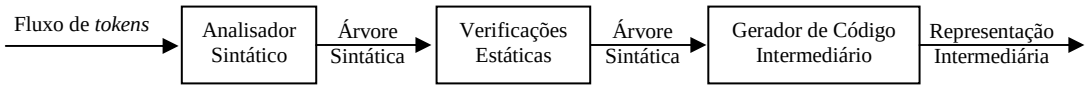


Figura 2.2.3.4.1. Posição do verificador estático no compilador

A verificação estática é bastante complexa, e envolve um conjunto variado de verificações semânticas. Duas dessas verificações são importantes do ponto de vista de um compilador: a *verificação de tipos* e a *verificação de unicidade e escopo*.

Na verificação de tipos é efetuado um conjunto de verificações com o fim de investigar se o tipo de uma construção corresponde exatamente àquele esperado no contexto. Alguns exemplos de verificações desse tipo são:

- Verificar se estão sendo adicionados duas variáveis de tipos compatíveis.
- Verificar se ambos os operandos para uma operação com o operador *mod* têm o tipo inteiro.
- Verificar se a indexação está sendo feita sobre um vetor.
- Verificar se uma chamada à função contém o número e tipos corretos de argumentos.

Na verificação de unicidade e escopo é efetuado um conjunto de verificações com o fim de assegurar que um identificador foi declarado univocamente dentro da visibilidade de um bloco.

2.2.3.5. Verificação de tipos

A verificação de tipos é baseada nas informações sobre as construções sintáticas da linguagem, a noção de tipos de dados e as regras para atribuição de tipos às construções das linguagens.

Expressões de tipo

O tipo de uma construção de linguagem é denotado por uma *expressão de tipo*. Informalmente, uma expressão de tipo é um tipo básico ou é formada através da aplicação de um operador (chamado *construtor de tipos*) a outras expressões de tipo. Os *tipos básicos* e os *construtores de tipos* dependem da linguagem sendo analisada.

Uma *expressão de tipo*, em Pascal, é definida através das seguintes regras:

1. Um tipo básico é uma expressão de tipo.
   Os tipos básicos, na linguagem Pascal, são os tipos **integer**, **boolean**, **char**, **real** e **string**.
2. Um nome de tipo é uma expressão de tipo.
   Os nomes de tipos são definidos, na linguagem Pascal, com o uso da palavra reservada **Type**.
3. Um construtor de tipos aplicado a uma expressão de tipo é uma expressão de tipo.
   Os construtores de tipo, na linguagem Pascal, abrangem:

   - *Vetores*. Se *T* é uma expressão de tipo, então *array* (*I*, *T*) é uma expressão de tipo denotando o tipo *vetor* de elementos com tipo *T* e conjunto de índices *I*, onde *I* é freqüentemente um intervalo de inteiros.

     Por exemplo, a declaração

     Var A**: array [** 1 **..** 10 **] of integer;**

     associa a expressão de tipo *array* (1..10, *integer*) à variável *A*.

   - *Registros*. Um construtor do tipo *registro* é aplicado a uma tupla formada a partir dos nomes de campos e dos tipos de campos.

     Por exemplo, a declaração

     ```
     Type linha = record
                    Endereço: integer;
                    Lexema: array [1..15] of char
                    End;
     ```

     representa o nome de tipo *linha* como representando a seguinte expressão de tipo

     Registro ((endereço x *inteiro*) x (lexema x (*array* (1..15, *caractere*)))

   - *Apontadores.* Se *T* é uma expressão de tipo, então *apontador* (*T*) é uma expressão de tipo denotando o tipo “apontador para um objeto do tipo T”.

     Por exemplo, a declaração

     Var p: ↑ linha;

     Declara a variável p como tendo o tipo *apontador* (linha).

- *Funções.* Em Matemática, uma função mapeia elementos de um conjunto, o *domínio*, em outro conjunto, o contradomínio. Devemos tratar as funções nas linguagens de programação como mapeando o *domínio de tipos D* em um contradomínio *de tipos* R. O tipo duma função será denotado pelas expressões de tipo $D \rightarrow R$.

  Por exemplo, a declaração Pascal

  function f (a, b: *char*) : ↑*integer*;

  Diz que o domínio de tipos de *f* é denotado por *caractere x caractere* e o contradomínio de tipos por *apontador* (*inteiro*).

Equivalência das expressões de tipo

A forma geral para validar regras semânticas para verificação de tipos baseia-se na definição de *equivalência de tipos*. A *equivalência de tipos* abrange dois esquemas distintos para determinar a igualdade de duas expressões de tipo: a *equivalência de nomes* e a *equivalência estrutural*.

A *equivalência por nome* enxerga cada nome de tipo como o de um tipo distinto, de tal forma que duas *expressões de tipo* são nomes equivalentes se, e somente se, os nomes forem idênticos. Na *equivalência estrutural*, os nomes são substituídos pelas expressões de tipos que os definem e, dessa forma, duas expressões são estruturalmente equivalentes se representarem duas expressões de tipo estruturalmente equivalente após a substituição de todos os nomes.

Os conceitos de equivalência por nome e estrutural são úteis para explicar as regras usadas pelas linguagens para associar tipos aos identificadores que figuram nas declarações.

Por exemplo, no conjunto de declarações Pascal

type link = ↑celula**;**
var próximo **:** link**;**
anterior **:** link**;**
p**:** ↑celula**;**
q**,** r**:** ↑celula**;**

o identificador *link* é declarado como sendo um nome para o tipo ↑celula. A questão envolvida é se as variáveis *próximo*, *anterior*, *p*, *q* e *r*, têm todas o mesmo tipo.

As expressões de tipo associadas à essas variáveis são dadas na tabela 2.2.3.5.1:

Variável	Expressão de tipo
próximo	*link*
anterior	*link*
p	*Apontador* (celula)
q	*Apontador* (celula)
r	*Apontador* (celula)

Tabela 2.2.3.5.1. Expressões de tipos para próximo, anterior, p, q e r

Pela equivalência por nome, as variáveis *próximo* e *anterior* têm o mesmo tipo, porque foram associadas às mesmas expressões de tipo. As variáveis *p*, *q* e *r* também têm o mesmo tipo, mas *p* e *próximo* não, uma vez que suas expressões de tipo associadas são diferentes. Pela equivalência estrutural, todas as cinco variáveis têm o mesmo tipo porque *link* é um nome para a expressão de tipo *apontador* (celula).

Conversões de tipos

A definição de uma linguagem de programação especifica quais são as conversões necessárias antes de aplicar um determinado operador a um conjunto de operandos.

A conversão de um tipo para outro é dita *implícita* se for realizada automaticamente pelo compilador. Conversões implícitas de tipo, também chamadas de *coerções*, estão limitadas em muitas linguagens às situações onde nenhuma informação é perdida em princípio. Por exemplo, um inteiro pode ser convertido num real mas não vice-versa. Na prática, alguma perda é possível quando um número real deve ser armazenado usando o mesmo número de *bits* que o inteiro.

A conversão é dita *explícita* se o programador precisa inserir comandos no programa fonte para que a conversão seja feita. As conversões explícitas agem como aplicações de funções para um verificador de tipos.

Exemplo. Consideremos expressões na forma $x + i$, onde x é do tipo real e i do tipo inteiro. Como a representação dos inteiros e reais é diferente dentro do computador e diferentes instruções de máquina são usadas para as operações sobre inteiros e reais, o compilador pode ter que primeiro converter um dos operandos de + para assegurar que ambos os operandos sejam do mesmo tipo quando a adição for efetuada.

Sobrecarga de operadores

Um símbolo *sobrecarregado* é aquele que possui diferentes significados dependendo do contexto em que é utilizado. Em Matemática, o operador de adição + é sobrecarregado, porque + em A + B possui diferentes significados quando A e B são inteiros, reais, números complexos ou matrizes.

A sobrecarga de operadores em uma linguagem de programação é *resolvida* quando um único significado para a ocorrência de um símbolo sobrecarregado é determinado. Por exemplo, se + denotar a adição inteira ou a real, então as duas ocorrências de + em $x + (i + j)$ podem denotar diferentes formas de adição, dependendo dos tipos de x, i e j.

2.2.3.6. Verificação de unicidade e escopo

As *verificações de unicidade e escopo* são efetuadas a partir de um conjunto definido de *regras de escopo*. As *regras de escopo* de uma linguagem determinam qual é a declaração de nome que deve ser aplicada para cada nome referenciado no programa fonte. A parte do programa à qual uma declaração se aplica é chamada de *escopo* daquela declaração.

As *regras de escopo léxico ou estático*, determinam a declaração que se aplica a um nome pelo exame isolado do texto do programa. As *regras de escopo dinâmico*, determinam a declaração aplicável a um nome em tempo de execução. Pascal, C e Ada estão entre as muitas linguagens que usam o *escopo estático*, enquanto Lisp, Apl e Snobol estão dentre as linguagens que usam o *escopo dinâmico*.

Amarração de nomes

Mesmo que num programa cada nome seja declarado uma única vez, o mesmo nome poderá denotar objetos de dados diferentes em tempo de execução. O termo informal “objeto de dados” corresponde a uma localização de memória que pode abrigar valores.

Na semântica das linguagens de programação, o termo *ambiente* se refere a uma função que mapeia um nome em uma localização de memória e o termo *estado* a uma função que mapeia uma localização de memória no valor guardado nela. (Figura 2.2.3.6.1)

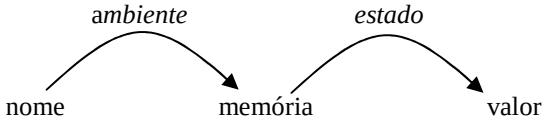


Figura 2.2.3.6.1. Um mapeamento de nomes em valores em dois estágios

Quando um ambiente associa uma localização de memória *s* a um nome *x*, dizemos que *x* está *amarrado* a *s*. Em Pascal, por exemplo, mais de uma ativação de um mesmo procedimento pode estar viva em determinado momento, de forma que o nome que define uma variável local ao procedimento estará amarrado a diferentes localizações de memória.

Regras de Escopo

Um bloco é definido como sendo um trecho de um programa fonte que contém suas próprias declarações de dados locais.

O conceito de bloco foi originado pela linguagem Algol. Em C, um bloco possui a sintaxe:

{ *declarações comandos* }

Uma característica dos blocos é a sua estrutura de aninhamento. Os delimitadores marcam o início e o final de um bloco. A linguagem C utiliza chaves, { e }, enquanto que por tradição de Algol diversas linguagens usam *begin* e *end*. Os delimitadores asseguram que um bloco é independente ou está aninhado dentro de outro bloco, isto é, não é possível, para dois blocos B_1 e B_2 se sobreporem, de forma que o primeiro, B_1, comece, em seguida B_2, mas que B_1 termine antes de B_2 terminar. Essa propriedade de aninhamento é algumas vezes referenciada como *estrutura de bloco*.

A ocorrência de um nome dentro de um bloco é dita *local* ao bloco se estiver no escopo de uma declaração dentro daquele bloco; caso contrário, a ocorrência é *não-local*. A distinção entre nomes locais e não-locais recai sobre qualquer construção sintática que possa ter declarações dentro de si.

Embora o escopo seja uma propriedade da declaração de um nome, é algumas vezes conveniente usar a abreviação "escopo de um nome *x*" em lugar de "escopo da declaração do nome *x* que se aplica a uma ocorrência de *x*".

As *regras de escopo* numa linguagem estruturada em blocos são dadas pela *regra do aninhamento mais interno*. Essa regra é definida através de duas outras regras:

1. O escopo de uma declaração num bloco *B* inclui *B*
2. Se um nome *x* não for declarado num bloco *B*, uma ocorrência de *x* em *B* estará no escopo de uma declaração de *x* num bloco B' envolvendo *B* tal que

   i. *B*' possui uma declaração para *x*
   ii. *B*' é o bloco mais internamente aninhado envolvendo *B*, em relação a qualquer outro bloco que contenha uma declaração para *x*.

2.2.3.7. Tabela de símbolos

Um compilador utiliza uma estrutura de dados denominada *tabela de símbolos* para guardar as informações relativas a nomes declarados no texto do programa fonte que está sendo analisado. Essas informações podem ser, por exemplo, um campo informando o tipo de objeto identificado com o nome (se o nome é um campo, vetor, registro ou subprograma, por exemplo), um campo informando o nome do bloco no qual o nome foi definido dentro do programa fonte, seu tipo, ou ainda uma cadeia de caracteres que o reconheça.

As duas estruturas de tabela de símbolos mais utilizadas são as listas lineares e as tabelas *hash*.

Entradas

Cada entrada da tabela de símbolos é destinada à declaração de um nome. O formato das entradas não tem que ser uniforme, porque os atributos semânticos de um nome dependem da sua função no programa fonte.

Em alguns casos, uma entrada na tabela de símbolos é definida a partir do analisador léxico, assim que um nome é encontrado no programa fonte. A descoberta, pelo compilador, de um mesmo nome denotando vários objetos diferentes durante a análise semântica, no entanto, exige que sejam feitas novas entradas para um mesmo nome, cada uma relacionada a um contexto onde o nome foi declarado.

Por exemplo, as declarações C

int *x*
struct x { *float y*, *z*; };

usam *x* tanto como um inteiro quanto como o rótulo de uma estrutura contendo dois campos. Esse trecho de código fará com que sejam criadas duas entradas para *x* na tabela de símbolos: uma reconhecendo *x* como um inteiro e outra como uma estrutura. Para cada uma dessas entradas serão armazenadas, durante a análise semântica, um conjunto próprio de informações, os *atributos semânticos* de referência ao nome.

Listas lineares

A estrutura de dados mais simples e fácil usada na implementação de uma tabela de símbolos consiste em uma lista linear de registros (Figura 2.2.3.7.1).

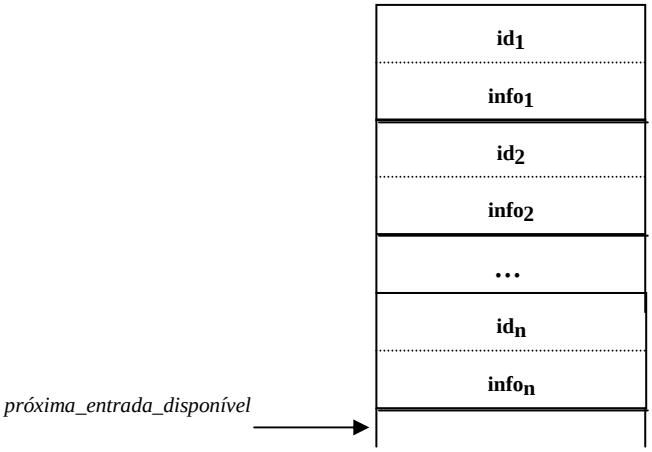


Figura 2.2.3.7.1. Uma lista linear de registros.

Novos nomes são adicionados à lista na ordem em que são encontrados. A posição do último elemento inserido na lista é marcada pelo apontador *próxima_entrada_disponível*. A pesquisa de um nome é feita do início da lista para o final. Se atingirmos o final da lista sem encontrar o nome, um erro é reportado – *um nome esperado não está na tabela de símbolos*.

Se a tabela de símbolos contém *n* nomes, o trabalho necessário para inserir um novo nome será constante se realizarmos a inserção sem verificarmos se o nome já pertence à tabela. Se não são permitidas múltiplas entradas para o mesmo nome, necessitamos, então, procurar ao longo de toda a tabela antes de descobrir se um nome já pertence a ela. Neste processo, a complexidade de tempo é O(*n*), pois para encontrarmos os dados a respeito de um nome, pesquisamos, em média, *n*/2 entradas.

Tabelas hash

As tabelas *hash* têm sido utilizadas em diversos compiladores. Aqui consideramos uma variante simples, conhecida como *hashing aberto*, onde o termo "aberto" se refere à propriedade de que não precisa haver limite no número de entradas que podem ser feitas numa tabela. Esse esquema nos dá a capacidade de realizar *e* entradas sobre *n* nomes num tempo de O(*n* (*n* + *e*)/ *m*), para qualquer constante *m* de nossa escolha. Uma vez que *m* pode ser feita tão grande quanto desejado, até o limite de *n*, esse método é geralmente mais eficiente do que as listas lineares e é um método comum para as tabelas de símbolos.

O esquema básico de tabelas *hash* é mostrado na Figura 2.2.3.7.2, e é constituído de duas estruturas de dados:

1. Um vetor fixo de *m* apontadores, que define a tabela *hash*,
2. *m* listas ligadas separadas, chamadas de *buckets*, que guardam as entradas da tabela

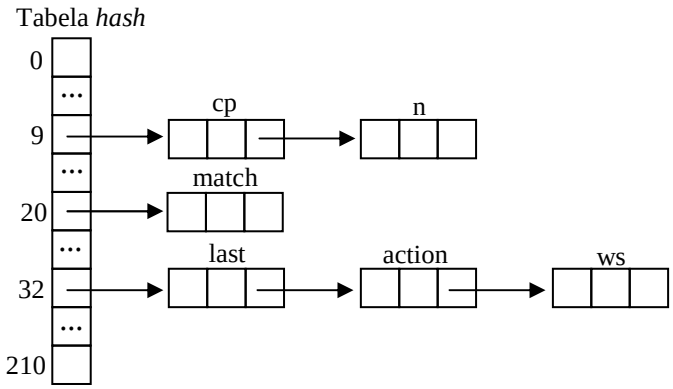


Figura 2.2.3.7.2. Uma tabela *hash* de tamanho 211.

Cada entrada da tabela de símbolos figura em uma das *m* listas. Para determinar se uma cadeia de caracteres *s* está armazenada na tabela de símbolos, aplicamos uma *função de hash h* a *s*, de tal forma que *h*(*s*) retorne um inteiro entre 0 e *m*-1. Se *s* estiver na tabela de símbolos, estará na lista numerada por *h*(*s*). Se *s* ainda não estiver na tabela de símbolos, é introduzida através da criação de um registro para a mesma, que é ligado ao início da lista indicada por *h*(*s*).

Em média, a lista tem um comprimento de *n*/*m* registros, se existirem *n* nomes numa tabela de comprimento *m*. Escolhendo *m* de forma que *n*/*m* esteja limitado por uma pequena constante, o tempo para acessar a uma entrada da tabela é essencialmente constante. O espaço ocupado pela tabela de símbolos consiste em *m* palavras para a tabela *hash* e *cn* palavras para as entradas da tabela, onde *cn* é o número de palavras por entrada na tabela. Portanto, o espaço para a tabela *hash* depende somente de *m* e o espaço para as entradas da tabela depende somente do número de entradas.

Atenção tem sido dada à questão de como projetar uma função de *hash* que seja fácil de computar, além de distribuir as entradas uniformemente entre as *m* listas.

2.2.4. Geração de Código

Nessa seção iremos descrever a estrutura de uma máquina virtual capaz de *interpretar* um conjunto de instruções *Pascal*, como proposto por Wirth [3]. Será descrita a organização de memória e o funcionamento básico da máquina virtual Pascal no processamento de uma sequência de instruções. Também será proposto, e explicado de forma sucinta, um conjunto de instruções suportadas pela máquina para gerar, paralelo à Análise Semântica do programa fonte, um conjunto de instruções em mais baixo nível.

2.2.4.1. Um computador ideal

O propósito de um compilador é reconhecer o formato de uma entrada bem-definida e, dessa forma, produzir uma saída bem-definida. O conceito de entrada e saída “bem-definidas” dependem, por sua vez, da estrutura da linguagem proposta pelo projetista do compilador e do tratamento dado à essa linguagem nas fases de análise semântica e geração de código. A saída para o compilador é um código especial, denominado de *código objeto*.

Uma questão importante na fase de geração de código é a decisão de optar entre gerar um código objeto para um processador específico, como por exemplo, instruções para o processador *Pentium*, ou gerar código objeto passível de ser interpretado por um outro programa, um *Interpretador de Código Objeto*.

Em vez de construir um compilador para um computador específico, será definida a estrutura de um computador ideal para um compilador *Pascal*, um *Computador Pascal*.

O *Computador Pascal* é ideal pelas seguintes razões:

1) As intruções geradas para o *Computador Pascal* têm uma correspondência direta com os conceitos da linguagem Pascal.

2) O conjunto de instruções geradas para o *Computador Pascal* tem praticamente a mesma sintaxe de um programa Pascal submetido ao compilador. Como consequência desse fato, o gerador de código é uma extensão trivial do analisador sintático.
3) O código objeto produzido poderá ser interpretado por um programa bastante simples, que tem a função única de interpretar as instruções geradas para o *Computador Pascal*, denominado *Interpretador Pascal.*

A vantagem essencial de construir e gerar instruções para uma máquina virtual, como a proposta com o *Computador Pascal*, é a propriedade de ser possível implantar o *Interpretador Pascal* em diferentes plataformas sem grandes esforços de programação, e sem a necessidade de fixar uma plataforma específica.

2.2.4.2. A descrição do Computador Pascal

A memória de um Computador Pascal é uma pilha (ou vetor) de inteiros, denominada de *vetor de execução*. Cada elemento desta pilha e seus índices correspondentes são denominados, respectivamente, *palavra* e *endereço de memória*.

O *vetor de execução* armazena em suas células o código objeto e o espaço necessário para endereçamento das variáveis de um programa Pascal. O código objeto, que possui um comprimento fixo, é colocado no início do vetor de execução. O resto do vetor é usado para armazenar valores de variáveis, e, também, durante a execução dos comandos, pode ainda armazenar resultados temporários obtidos através da avaliação de comandos, por exemplo o resultado para uma expressão.

código objeto do programa Pascal
espaço para endereçamento de variáveis
espaço livre (usado para trabalho)

Figura 2.2.4.2.1. Formato genérico da memória para o *Computador Pascal*

O *Computador Pascal* possui três registradores, de nomes *p*, *b* e *s*. Quando na execução de um programa, o *registrador de programa p* contém o endereço da instrução corrente, o *registrador de base b* é usado para endereçar variáveis, e o *registrador de execução s* armazena o endereço da última célula de memória utilizada no espaço livre do vetor de execução.

Um endereço no vetor de execução é denotado por *vetor_execução[x]*.

Funcionamento básico do Computador Pascal

Inicialmente apenas o código-objeto de um programa Pascal encontra-se armazenado no vetor de execução. Quando da execução do programa, o *Computador Pascal* aloca espaço no vetor de execução para armazenar valores para as variáveis definidas no bloco principal do programa, e esse espaço permanecerá alocado até que a execução do programa termine. Quando é solicitada a ativação de um subprograma é alocado espaço no vetor de execução para as variáveis locais definidas no mesmo. Esse espaço é liberado do vetor de execução quando a execução do procedimento termina.

2.2.4.3. Registros de Ativações

As informações utilizadas durante a execução de um subprograma são gerenciadas com o uso de um bloco contíguo de armazenamento, definido sob a nomenclatura de *registro de ativação* (Figura 2.5.3.1). Um *registro de ativação* consiste em uma coleção de campos, que armazenam:

1. Espaço para armazenamento de valores temporários, tais como aqueles que surgem da avaliação de expressões. Esses valores são armazenados em um espaço disponível do *vetor de execução*, o espaço para *temporários*, durante a execução de comandos.
2. Espaço para armazenamento das variáveis locais a uma execução do subprograma.

3. Um campo para armazenamento do valor retornado pelo subprograma para o subprograma que o chamou.
4. Um campo que armazena o estado da máquina exatamente antes do subprograma ser chamado. Estas informações podem incluir o valor de um contador de programa e o conteúdo dos registradores de máquina que precisam ser restaurados quando o controle retornar do subprograma.
5. Um *elo de acesso*, um campo usado para refenciar as variáveis definidas no escopo visível do subprograma.
6. Um *elo de controle*, um campo que guarda a referência do registro de ativação do subprograma chamador.
7. Espaço para armazenamento de dos parâmetros atuais para o subprograma, que é usado pelo subprograma chamador para fornecer os parâmetros do procedimento chamado.

A Figura 2.5.3.1. ilustra um registro de ativação contendo esses campos.

Figura 2.2.4.3.1. Um registro de ativação genérico.

Dentro de um registro de ativação, os parâmetros e as variáveis locais são identificados de acordo com a ordem pela qual foram definidos dentro do bloco do qual fazem parte. Além disso, se um subprograma é ativado recursivamente, cada ativação cria outra instância do registro de ativação no vetor de execução.

No caso específico do *Computador Pascal*, o registro de ativação consiste de quatro partes:

1) A parte dos parâmetros
2) A parte de contexto
3) A parte das variáveis
4) A parte temporária

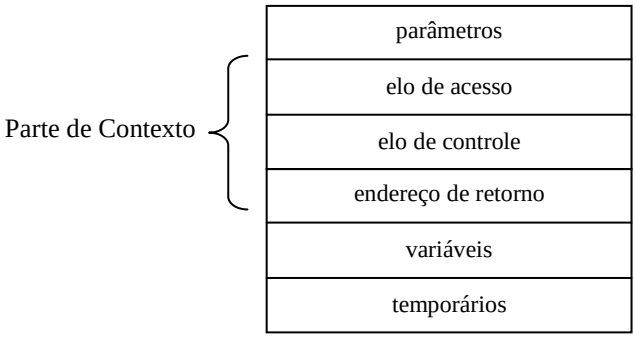


Figura 2.2.4.3.2. Estrutura de um registro de ativação

A parte dos parâmetros contém espaço para o armazenamento dos parâmetros definidos para o bloco. A parte de contexto contém o *elo de acesso,* o *elo de controle e* o *endereço de retorno*. Esses endereços definem o contexto no qual o bloco foi ativado. O *endereço de retorno* contém o endereço de retorno para a próxima instrução após a chamada do subprograma, instrução essa executada após a execução de todos os comandos contidos no corpo do subprograma chamado. A parte das variáveis contém espaço para o armazenamento das variáveis locais ao bloco. A parte temporária armazena operandos e resultados durante a execução dos comandos.

2.2.4.4. Variáveis

Em Pascal, qualquer variável possui um tipo fixo. Esse tipo pode ser um tipo pré-definido (inteiro, real, caracter ou cadeia), um tipo vetor, ou ainda um tipo registro.

Uma variável cujo tipo é um dos pré-definidos na linguagem ocupa um número definido de palavras de memória. Como vetores e registros são combinações de um número fixo de elementos de um tipo pré-definido (ou ainda, de forma mais geral, de vetores ou registros que são combinações de um número fixo de elementos de um tipo pré-definido), é claro que cada variável ocupa também um número fixo de palavras de memória. Esse número é denominado o *comprimento* da variável.

Quando um subprograma é ativado, o *ComputadorPascal* cria um registro de ativação e faz com que o registrador de base *b* aponte para o endereço de base desse registro de ativação. O *interpretador* pode, então, computar o endereço relativo de cada variável dentro do registro de ativação com relação a esse registrador.

Os endereços em memória relativos ao registrador de base são denominados de *deslocamentos* em relação a esse registrador.

Por exemplo, dado o seguinte fragmento de código:

```
Procedure Quicksort (m, n: integer);
Var i, j: integer;
Begin ... End;
```

O registro de ativação correspondente à ativação desse procedimento é:

-2	m
-1	n
0	elo de acesso
1	elo de controle
2	endereço de retorno
3	i
4	j

As variáveis declaradas neste subprograma possuem os seguintes deslocamentos:

Variável	Deslocamento
M	-2
N	-1
I	3
J	4

Estas variáveis podem ser acessadas através da soma do seu deslocamento com o valor do registrador *b*:

Variável	Deslocamento
m	b -2
n	b -1
i	b + 3
j	b + 4

Após o término da execução do subprograma, o *Computador Pascal* remove o correspondente registro de ativação do vetor de execução. Cada registro de ativação é ligado ao registro de ativação que o chamou através do *elo de controle*, que contém o endereço base do registro de ativação que o chamou. Quando um subprograma termina, o endereço de base do subprograma chamador, armazenado no *elo de controle* do corrente registro de ativação, é tomado como o novo endereço de base para o registrador de base *b*.

O encadeamento de *elos de controle* durante a execução de um programa, que reflete uma sequência de ativações de subprogramas em um dado momento, recebe a denominação de *ligação dinâmica*, onde a denominação "dinâmica" está relacionada com a sequência dinâmica na qual os blocos vão sendo ativados.

O acesso às variáveis definidas dentro de outros registros de ativações é possível através do encadeamento dos *elos de acesso* entre as diversas ativações de blocos de subprogramas (consequência direta da análise de escopo na análise semântica). Esse encadeamento define o conjunto de variáveis que podem ser acessadas de dentro do bloco corrente. O encadeamento de *elos de acesso* recebe a denominação de *ligação estática*, denominação que ressalta a estrutura estática do escopo das variáveis de dentro de um bloco de subprograma.

O conjunto de variáveis acessíveis a partir de um bloco é denominado de *contexto corrente* do programa. Em um dado momento o contexto corrente é definido pelo encadeamento estático que se inicia a partir do registrador *b*.

2.2.4.5. Instruções que definem o Computador Pascal

Existe um conjunto de instruções que definem o código objeto gerado para um *Computador Pascal*. Essas intruções estão relacionadas com os seguintes temas: *acesso a variáveis, avaliação de expressões, execução de comandos* e *chamada de subprogramas.*

Instruções usadas para acessar variáveis

O acesso a uma variável dentro do *contexto corrente* é possível através de duas informações:

1) O registro de ativação que contém a variável
2) O endereço da variável dentro desse registro de ativação, em relação ao endereço de base.

Durante a análise semântica, o compilador associa a cada variável duas informações: uma informação de escopo e o endereço da variável dentro do escopo no qual foi definida. A primeira delas, dito de forma breve, é o número de *elos de acesso* que o *Computador Pascal* deve seguir através da *ligação estática* para encontrar o registro de ativação que contém a variável solicitada. A segunda é a posição da variável, em memória, relativa ao endereço de base para o registro de ativação no qual está definida a variável.

Toda vez que um programa se refere a uma variável pelo seu nome é gerada uma instrução na forma

VAR (Nível, Deslocamento)

A instrução consiste de duas partes:

1) Uma parte que contém o operador VAR, utilizado para calcular o endereço absoluto de uma variável;
2) Dois argumentos que definem as informações necessárias para obtenção do endereço em memória da variável.

A parte da instrução que contém o operador e os argumentos ocupam uma palavra de memória cada. Durante a execução da instrução, o *registrador de programa p* aponta para o operador da instrução. O computador executa a instrução em cinco passos:

1) O *registrador de execução* é incrementado de um com o fim de criar uma área de memória para o armazenamento do endereço da variável.
2) O endereço base da variável é encontrado percorrendo-se recursivamente a *ligação estática;*
3) O endereço absoluto da variável é computado adicionando-se ao endereço base o deslocamento da variável.
4) O endereço absoluto é armazenado na área de memória criada no passo 1;
5) O *registrador de programa* é incrementado de 3, e aponta para a próxima instrução.

O procedimento esboçado pelo algoritmo abaixo define a instrução VAR. Os parâmetros formais denotam os argumentos da instrução.

```
void VAR(int level, int displ)
 {
  int x;

  s++;
  x = b;
  while (level > 0)
    {
     x = vetor_execucao[x];
     level--;
    }
 vetor_execucao[s] = x + displ;
  p = p + 3;
 }
```

Instruções VAR são usadas para obter o endereço de variáveis e parâmetros passados por valor. Parâmetros passados por referência são acessados de uma forma diferente. No vetor de execução, em vez do endereço do parâmetro local x, necessitamos do endereço da variável para o qual essa localização aponta. Dessa forma, deve-se criar outra instrução, que recupera um endereço de memória e o associa à outro (o endereço da variável).

Essa instrução, VARPARAM, é definida como:

VARPARAM (Nível, Deslocamento)

onde os argumentos da instrução são semelhantes aos argumentos da já definida instrução VAR.

A instrução VARPARAM é definida pelo algoritmo abaixo:

```
void VARPARAM(int level, int displ)
 {
  int x;

  s++;
  x = b;
  while (level > 0)
    {
     x = vetor_execucao[x];
     level--;
    }
 vetor_execucao[s] = vetor_execucao[x + displ];
  p = p + 3;
 }
```

O acesso a variáveis envolve ainda dois casos especiais:

1. O acesso a uma posição fixa de memória dentro de um vetor.
2. O acesso a posição de memória correspondente a um campo de um registro

Estes casos, denominados *casos seletores*, são expressos na linguagem Pascal através da seguinte regra sintática:

Variável → **id** idtail
idtail → **.id** idtail | **[** expressão **]** | ∈

As instruções geradas para o conjunto de produções acima poderiam ser pensadas da seguinte forma:

1. O conjunto de instruções geradas para obter o endereço de uma variável consiste de uma instrução de acesso à variável, seguido de um conjunto de instruções para acesso à uma posição de memória referente a um ou mais seletores, que podem ser campos ou posições fixas dentro de um vetor;
2. A instrução para o acesso á uma variável é uma instrução VAR ou uma instrução VARPARAM.
3. A instrução para acesso a uma posição de memória referente a um seletor é um conjunto de instruções consistindo de:

   - Um conjunto finito de instruções referentes a uma expressão seguida por uma instrução INDEX, que determina uma posição fixa de memória dentro de um vetor, posição esta diretamente relacionada ao resultado obtido através da avaliação da expressão, ou
   - Uma instrução FIELD, que determina a posição de memória referente a um campo.

Durante a fase de geração de código, a estrutura do código gerado pode também ser representada por regras sintáticas. Com essa convenção é possível associar, a um conjunto de produções que definem uma linguagem, um conjunto de instruções que possuem uma correspondência direta com as regras sintáticas que definem a linguagem Pascal.

A definição sintática para as instruções de acesso à variáveis é dada pelas produções abaixo:

Acesso à variável → **VAR** Seletor | **VARPARAM** Seletor
Seletor → Expressão **INDEX** | **FIELD** Seletor

Uma instrução INDEX adiciona um deslocamento ao endereço obtido previamente da avaliação de outras instruções de acesso à variável (VAR, VARPARAM ou ainda recursões de FIELD) para obter o endereço de uma posição de memória fixa referente ao vetor. O cálculo do deslocamento é feito através da multiplicação de um valor (o resultado obtido com a avaliação da expressão passada como referência ao vetor) pelo tamanho do tipo de dados definido para o vetor.

A instrução INDEX verifica ainda se a posição de memória referenciada pelo vetor encontra–se definida dentro da faixa de intervalo definida na declaração do vetor, e imprime uma mensagem de erro de execução caso isso não tenha sido satisfeito.

O algoritmo que define a instrução INDEX é:

```
void INDEX(int lower, int upper, int tamanho)
 {
  int i;
  i = vetor_execucao[s];
  if ((i<lower) || (i>upper))
   {
    printf("\n >>> Erro nos limites do vetor!");
   }
  else
   vetor_execucao[s] = vetor_execucao[s] + (i – lower)*tamanho;

  p = p+4;
 }
```

Por exemplo, para a seguinte declaração:

Program teste**;**
Var i **: integer;**
A**: array [** 1 **..** 10 **] of integer;**

temos o acesso a uma variável na forma A[i] codificado por:

```
    VAR ( 0 , 4 )
    VAR ( 0 , 3 )
  * VAL( 1 )
    INDEX ( 1, 10, 1 )
```

notando-se que a instrução VAL retorna o valor armazenado na variável *i*

A primeira instrução recupera o endereço em memória onde está armazenada a variável A. As duas instruções seguintes são responsáveis por avaliar a expressão *i* . A última instrução do conjunto, INDEX, adiciona o valor obtido com a avaliação de *i* à posição de memória onde está armazenada a variável A, obtendo, assim, a posição em memória para A[i].

Uma instrução FIELD adiciona o deslocamento do campo em questão (deslocamento esse relativo ao início do registro no qual o campo foi definido) ao endereço obtido previamente na avaliação de outras instruções de acesso à variável (VAR, VARPARAM, INDEX ou ainda recursões de FIELD) para obter o endereço onde está armazenado o campo em questão.

O algoritmo que define a instrução FIELD é:

```
void FIELD(int displ)
   {
    vetor_execucao[s] = vetor_execucao[s] + displ;
    p = p + 2;
   }
```

Por exemplo, para a seguinte declaração:

Program teste**;**
Type R = **record** f, g **: integer;** h **: boolean; end;**
Var x **:** R**;**

o acesso a uma variável na forma *x.g* é codificado por:

VAR (0 , 3)
FIELD (1)

O efeito dessas instruções é mostrado na Figura 2.2.4.5.1.

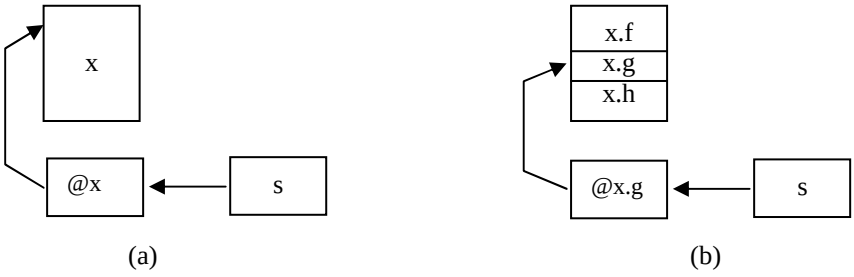


Figura 2.2.4.5.1. Acesso ao campo de um registro usando a instrução FIELD

A instrução VAR coloca o endereço da variável *x* no vetor de execução, como mostrado na figura 2.2.4.5.1.a. A instrução FIELD adiciona o deslocamento do campo *g* ao endereço da variável *x* para obter o endereço do campo *x.g* (veja figura 2.2.4.5.1.b)

Instruções usadas para avaliar expressões

O *Computador Pascal* avalia uma expressão no espaço utilizado para armazenamento de valores temporários do registro de ativação atual. Para isto, ele faz uso do registrador *s* para obter os operandos antes de calcular uma operação sobre eles.

A avaliação de expressões evidencia a necessidade de uma instrução capaz de recuperar o valor armazenado em um endereço de memória relacionado a uma variável. Uma instrução desse tipo usa como argumento o endereço em memória de uma variável (gerado através de uma instrução VAR ou VARPARAM) e recupera o valor armazenado naquele endereço de memória. A função responsável por efetuar a segunda dessas operações, VAL, obtém do vetor de execução o endereço de uma variável, inserindo em seu lugar o valor armazenado naquele endereço.

A instrução VAL é definida através do algoritmo abaixo, onde o argumento para a instrução informa o tamanho da valor a ser recuperado (em palavras de memória).

```
void VAL(int lenght)
 {
  int x,i;

  x = vetor_execucao[s];
  i = 0;
  while (i<lenght)
    {
     vetor_execucao[s+i] = vetor_execucao[x+i];
     i++;
    }
  s = s + lenght - 1;
  p = p + 2;
 }
```

Expressões são hierarquicamente definidas por *fatores*, *termos* e *expressões simples*. Os fatores tem a seguinte sintaxe:

Fator → Constante | Variável | **(** Expressao **)** | Negacao Fator

A instrução gerada na identificação de uma constante é responsável por inserir um valor inteiro correspondente à constante no espaço para armazenamento de valores temporários do vetor de execução, a partir do registrador *s*, para que essa constante possa, posteriormente, ser utilizada como argumento em uma expressão ou comando. Uma instrução com essas propriedades é definida como uma instrução CONST.

Uma constante consistindo de um valor inteiro, booleano ou um único caractere ocupa uma única *palavra de memória* no vetor de execução. Assim, uma instrução CONST para constantes de algum desses tipos faz com que um único valor inteiro correspondente à constante seja inserido na parte temporária do vetor de execução apontada pelo registrador *s*. O tratamento dado a cadeias de caracteres é uma generalização da idéia acima, onde para cada caractere é gerada da cadeia é gerada uma instrução CONST.

O formato da instrução CONST é o seguinte:

CONST (*valor inteiro*)

O algoritmo que implementa a instrução é mostrado abaixo:

```
void CONST(int value)
  {
   s++;
   vetor_execucao[s] = value;
   p = p + 2;
  }
```

O caso particular das constantes numéricas em ponto flutuante requer que um tratamento mais complexo seja feito, pois os dados armazenados no vetor de execução são valores inteiros.

Uma solução para o problema consiste em “quebrar” a constante real em constantes inteiras. Na linguagem C variáveis definidas em ponto flutuante são representadas por 32 bits de memória, enquanto variáveis inteiras são representadas por 16 bits. Pode-se pensar, então, que variáveis do primeiro tipo podem ser “quebradas” em dois inteiros, o primeiro contendo os 16 bits mais à esquerda na representação binária da constante, e o segundo contendo os 16 bits restantes. Assim, constantes em ponto flutuante podem ser representadas através de uma instrução CONSTREAL, que recebe *n* argumentos. A partir desses argumentos é possível recuperar o valor original de uma constante em ponto flutuante.

O caso das expressões entre parênteses é simples: os parênteses não existem no código objeto, e assim uma expressão entre parênteses simplesmente produz as instruções para a própria expressão.

A negação de fatores é compilada em dois passos: um conjunto de instruções responsável por avaliar um fator, seguido por uma instrução que nega o resultado obtido no passo anterior.

Nesse ponto surge uma importante característica das expressões. No texto do programa, uma expressão é escrita na notação convencional *infixa*, onde um operador, como **not**, que tem um operando simples, aparece antes do operando. O código compilado para uma expressão, no entanto, resulta como se a expressão tivesse sido escrita na notação *posfixa*, onde os operandos de um operador aparecem antes deste.

A notação posfixa é ideal para um computador capaz de avaliar expressões utilizando uma pilha. Esta idéia é utilizada pelo Computador Pascal para avaliação de expressões, da seguinte forma:

1) O computador "varre" uma expressão da esquerda para a direita, de forma tal que, quando um operando é referenciado, seu valor é imediatamente "empilhado" no topo da pilha.
2) Quando um operador é encontrado na "varredura", o computador retira da pilha seus operandos previamente armazenados, aplica a operação relacionada ao operador em questão aos operandos, e armazena no topo da pilha o valor resultante dessa operação.

As duas instruções responsáveis pela negação de fatores são as instruções NOT (para fatores booleanos) e MINUS (para fatores numéricos).

O algoritmo que descreve a instrução de negação para tipos booleanos, NOT, é definido abaixo:

```
void NOT()
  {
   if (vetor_execucao[s] == 1)
     vetor_execucao[s] = 0;
   else
     vetor_execucao[s] = 1;
   p = p + 1;
  }
```

A instrução usada para negação de fatores numéricos (MINUS) é trivial, com a ressalva do tratamento especial dado à negação de constantes numéricas em ponto flutuante. Nesse caso a constante real a ser negada deve ser recuperada (a partir dos *n* inteiros que a representam) antes de ser submetida à operação de negação, e então novamente "quebrada" em *n* inteiros. Em seguida, a constante é armazenada novamente no vetor de execução.

Um termo, na linguagem Pascal, é definido através da seguinte regra sintática:

Termo → Termo { Operador Multiplicativo Fator } | Fator
Operador Multiplicativo → ***** | **div** | **/** | **mod** | **and**

O algoritmo abaixo define a instrução MULTIPLY, responsável por efetuar a multiplicação de dois inteiros. As operações DIV e MOD, responsáveis por calcular, respectivamente, o resultado e o resto inteiros da divisão de dois inteiros, seguem raciocínio semelhante.

```
void MULTIPLY()
  {
   s = s - 1;
   vetor_execucao[s] = vetor_execucao[s] * vetor_execucao[s+1];
   p = p + 1;
  }
```

A operação de multiplicação de números em ponto flutuante envolve o tratamento especial para constantes numéricas desse tipo (conversão dos *n* inteiros na constante real) antes que seja efetuada a operação de multiplicação. O resultado obtido da multiplicação, uma constante numérica em ponto flutuante, é convertido em *n* inteiros antes de ser novamente armazenada na parte temporária do vetor de execução.

A instrução que define o “AND” lógico entre dois operandos booleanos, AND, é definida pelo algoritmo abaixo:

```
void AND()
 {
  s--;
  if (vetor_execucao[s] == 1)
    vetor_execucao[s] = vetor_execucao[s+1];
  p = p + 1;
 }
```

As instruções correspondentes às produções para um termo são definidas através da seguinte regra sintática:

Termo → Termo { Fator Operador Multiplicativo } | Fator
Operador Multiplicativo → **MULTIPLY** | **DIV** | **DIVIDE** | **MOD** | **AND**

Uma expressão simples, na linguagem Pascal, é definida através da seguinte regra sintática:

Expressão Simples → Expressão Simples { Operador Aditivo Termo } | Termo
Operador Aditivo → + | - | **or**

As operações ADD e SUBTRACT, responsáveis por calcular, respectivamente, a soma e a subtração de dois inteiros, seguem raciocínio semelhante aquele usado na construção do algoritmo para multiplicação de inteiros. As operações de adição para números em ponto flutuante envolvem o tratamento especial para constantes numéricas desse tipo (conversão dos *n* inteiros na constante real) antes que seja efetuada a operação de adição. O resultado obtido, uma constante numérica em ponto flutuante, é convertido em *n* inteiros antes de ser armazenada na parte temporária do vetor de execução.

A instrução que define o “OR” lógico entre dois operandos booleanos, AND, é definida pelo algoritmo abaixo:

```
void OR()
 {
  s--;
  if (vetor_execucao[s] == 0)
    vetor_execucao[s] = vetor_execucao[s+1];
  p = p + 1;
 }
```

As instruções correspondentes às produções para uma expressão simples são definidas através da seguinte regra sintática:

Expressão Simples → Expressão Simples { Termo Operador Aditivo } | Termo
Operador Aditivo → **ADD** | **SUBTRACT** | **OR**

Uma expressão completa na linguagem Pascal, é definida através da seguinte regra sintática:
Expressão → Expressão { Operador Relacional Expressão Simples } | Expressão Simples
Operador Relacional → < | = | > | <= | <> | >=

As operações LESS, EQUAL, BIG, LESSEQUAL, DIFERENT e BIGEQUAL, responsáveis por calcular para dois inteiros, respectivamente, se o primeiro deles é menor, igual, maior, menor ou igual, diferente ou maior ou igual ao segundo, seguem raciocínio semelhante ao usado na construção do algoritmo para multiplicação de inteiros. As operações relacionais cujos operandos são números em ponto flutuante envolvem o tratamento especial para constantes numéricas desse tipo (conversão dos *n* inteiros na constante real) antes que seja efetuada a operação relacional. O resultado obtido através da avaliação de uma operação relacional consiste em um valor booleano.

As instruções correspondentes às produções para uma expressão completa são definidas através da seguinte regra sintática:

Expressão → Expressão { Expressão Simples Operador Relacional } | Expressão Simples
Operador Relacional → **LESS** | **EQUAL** | **BIG** | **LESSEQUAL** | **DIFERENT** | **BIGEQUAL**

2.2.4.6. Execução de comandos

Os comandos são, basicamente, divididos em 5 categorias:

- Comandos de atribuição
- Comandos de repetição
- Comandos condicionais
- Comandos de entrada/saída
- Chamadas a subprogramas

As chamadas a subprogramas, por envolver uma certa complexidade, serão consideradas em tópico separado.

Os comandos de atribuição, de forma bem geral, envolvem uma variável (localizada à esquerda do operador de atribuição) e uma expressão (localizada à direita do operador de atribuição), e a regra sintática que define um comando desse tipo é definida através da seguinte produção:

Comando de Atribuição → Variável **:=** Expressão **;**

A idéia para um comando de atribuição é que o valor obtido através da avaliação de uma expressão localizada no lado direito do sinal de atribuição deve ser associado ao endereço da variável localizada no lado esquerdo desse mesmo sinal. Estando no vetor de execução o endereço da variável a receber a atribuição e o valor obtido através da avaliação da expressão a ser atribuída, a instrução deve recuperar ambos os operandos e guardar o valor armazenado no segundo deles no endereço em memória especificado pelo primeiro. Com o fim de tornar a instrução mais flexível, a mesma deve ainda ser informada do tamanho (em *palavras*) do valor a ser atribuído.

A instrução de atribuição, ASSIGN, é definida através do seguinte algoritmo:

```
void ASSIGN(int length)
 {
  int x,y,i;

  s = s - length - 1;
  x = vetor_execucao[s+1];
  y = s + 2;
  i = 0;
  while (i < length)
    {
     vetor_execucao[x+i] = vetor_execucao [y+i];
     i++;
    }
  p = p + 2;
 }
```

A regra sintática que define o comando é dada pela seguinte produção:

Comando de Atribuição → Acesso à Variável Expressão **ASSIGN**

Os comandos de repetição, assim como os **comandos condicionais**, usam instruções que possuem a propriedade de desviar para endereços de memória pré-estabelecidos. Isso é suportado pelo compilador pelo fato do mesmo conhecer o endereço e o comprimento de cada instrução gerada (que possui um tamanho fixo).

Os comandos que usam desvios em sua estrutura são definidos através de duas instruções, DO e GOTO, definidas abaixo:

- A instrução DO recebe um argumento inteiro e decide entre prosseguir para a próxima instrução na sequência de instruções ou desviar para outra instrução, de acordo com um valor booleano obtido da parte temporária do vetor de execução apontada pelo registrador *s*. Se o valor booleano obtido for 1 (*verdadeiro*), a próxima instrução a ser executada é a próxima na sequência das instruções do programa objeto, caso contrário a próxima instrução a ser executada será aquela identificada através do endereço passado como parâmetro para a instrução.
- A instrução GOTO é uma instrução que recebe um argumento inteiro e executa um desvio para a instrução identificada no endereço correspondente à esse argumento.

Os algoritmos que definem essas duas instruções são dados abaixo:

```
void DO(int displ)
 {
  if (vetor_execucao[s] == 1)
    p = p+2;
  else
   p = displ;
  s--;
 }

void GOTO(int displ)
 {
  p = displ;
 }
```

Um comando de repetição **while**, como

While B **do** S

gera uma sequência de instruções no seguinte formato:

```
L1 :  B
       DO ( L2 )
       S
      GOTO ( L1 )
L2:
```

Essas instruções são executadas através dos seguintes passos:

1) A expressão B é avaliada, produzindo como resultando um valor booleano na parte temporária do vetor de execução.
2) A instrução DO remove o valor booleano obtido do vetor de execução. Se o valor é *verdadeiro*, o computador prossegue para o passo 3. Caso contrário, a execução do comando while é terminado através de um desvio para a instrução presente no endereço identificado com o rótulo L2.
3) A sequência de instruções de comandos dada por S é executada sequencialmente, após o qual o passo 1 é repetido por um desvio para a instrução para a instrução presente no endereço identificado com o rótulo L1, através do comando GOTO.

A regra sintática que define a sintaxe para o código objeto gerado por um comando while é definida através da seguinte produção:

Comando While $\rightarrow$ Expressão **DO** Comandos **GOTO**

Um comando condicional **if**, como

If B **then** S

É compilado na seguinte sequência de instruções:

```
    B
    DO ( L )
    S
L:
```

Se a avaliação da expressão B produz uma valor booleano *falso*, a instrução DO desvia para o endereço L localizado ao final do comando **if**. Caso contrário, a sequência de instruções dada por S é executada.

A regra sintática que define a sintaxe para o código objeto gerado por um comando **if** é definida através da seguinte produção:

Comando If $\rightarrow$ Expressão **DO** Comandos

Os comandos de entrada e saída são responsáveis pela leitura/ escrita de dados a partir de um dispositivo de entrada/ saída pré-definido (teclado, monitor ou arquivo). Os argumentos para um comando dessa classe deve especificar:

- O dispositivo de entrada/saída com o qual o comando deve lidar (teclado, monitor ou arquivo)
- O tipo de dado a ser lido/escrito do/no dispositivo

Uma instrução WRITE, usada para impressão de dados, recebe como argumento uma constante inteira que identifica o tipo da expressão a ser impressa. A instrução pressupõe que a expressão a ser impressa encontra-se disponível na parte temporária do vetor de execução. A instrução obtém, assim, da parte temporária do vetor de execução, o valor obtido com a avaliação da expressão passada como argumento ao comando Pascal **write** e imprime esse resultado na tela.

Um comando de escrita de dados, como

Write (Expressão)

é compilado na seguinte sequência de instruções:

Instruções para Expressão
Instrução **WRITE**

Uma instrução READ, usada para ler um valor do teclado, recebe como argumento uma constante inteira (que identifica o tipo do valor a ser lido), e pressupõe que o endereço da variável que armazenará este valor se encontra disponível na parte temporária do vetor de execução. A instrução obtém, assim, da parte temporária do vetor de execução, o endereço em que deve ser armazenado o valor lido do teclado, e guarda esse último na posição de memória identificada.

Assim, um comando Pascal de leitura de dados, como

Read (variável)

é compilado na seguinte sequência de instruções:

Acesso á Variável
Instrução **READ**

2.2.4.6. Subprogramas

A ativação de subprogramas é feita por dois comandos, responsáveis pelas seguintes ações:

- Criar a parte de contexto para o registro de ativação do subprograma
- Alocar espaço para o armazenamento das variáveis locais definidas no subprograma
- Definir a primeira instrução do corpo do subprograma como a próxima instrução a ser executada.

A parte de contexto do registro de ativação para um subprograma consiste de três endereços:

1. O *elo estático*, endereço a partir do qual será possível acessar as visíveis dentro do escopo do subprograma.
2. O *elo dinâmico*, que contém o endereço base do registro de ativação chamou o subprograma.
3. O *endereço de retorno*, que contém o endereço da instrução seguinte à instrução de chamada para o subprograma.

A instrução responsável por criar a parte de contexto para um subprograma e promover também um desvio no código objeto para o endereço de memória onde está localizada a primeira instrução do subprograma sendo chamado é uma instrução CALLFUNC. Duas informações devem ser fornecidas à instrução para a execução dessas ações, sendo elas:

- Uma informação de escopo, indicando o número de *elos de acesso* que o computador deve percorrer através da *ligação estática* para encontrar o registro de ativação onde está definido o subprograma.
- Um deslocamento, que identifica o endereço da primeira instrução do subprograma sendo chamado, em relação ao endereço onde a instrução CALLFUNC foi definida.

Dessa forma, a instrução CALLFUNC pode ser definida através do seguinte algoritmo:

```
void CALLFUNC(int level, int displ)
 {
  int x;

  s = s++;
  x = b;
  while (level > 0)
    {
     x = vetor_execucao[x];
     level--;
    }
  vetor_execucao[s] = x;       { Ligação Estática }
  vetor_execucao[s+1] = b;     { Ligação Dinâmica }
  vetor_execucao[s+2] = p + 3; { Endereço de Retorno }
  b = s;
  s = b + 2;
  p = p + displ;
 }
```

A identificação dos parâmetros passados a um subprograma é feito anteriormente à criação do registro do seu registro de ativação correspondente. Assim, quando o registro de ativação de um subprograma é criado, seus argumentos já estão definidos, resultes da avaliação de alguma expressão. Em parâmetros passados por valor os argumentos correspondem a valores de inicialização para os mesmos, enquanto que em parâmetros passados por referência os argumentos correspondem a endereços de memória relacionados com os parâmetros.

Por exemplo, uma chamada de subprograma, como em

Fatorial (n)

Produz, através de uma passagem de parâmetros por valor, o seguinte código:

VAR (... , ...)
VAL (.....)

O código produzido por uma passagem de parâmetros por referência, por sua vez, produz uma única instrução VAR:

VAR (.... , ...)

Que corresponde ao endereço da variável passada como argumento para o subprograma.

Depois de criada a parte de contexto para o registro de ativação através da instrução CALLFUNC, o computador executa a primeira instrução do subprograma: uma instrução FUNC, responsável por alocar espaço para o endereçamento das variáveis locais do subprograma e definir como a próxima instrução a ser executada a primeira instrução contida no corpo de comandos para o subprograma.

As informações necessárias para a instrução FUNC poder executar esses passos são:

- O tamanho total do bloco de variáveis locais definido no subprograma, para que seja reservado espaço no vetor de execução para o endereçamento das mesmas. No caso específico das funções, à esse tamanho deve ser acrescentada a quantidade de memória necessária para armazenamento do valor de retorno para a função.
- O tamanho da área de memória utilizada para trabalho (tamanho da parte temporária) para o subprograma.
- Um deslocamento, que identifica a primeira instrução do corpo de comandos do subprograma. Esse deslocamente é, na verdade, a diferença entre o endereço da primeira instrução do corpo de comandos do subprograma e o endereço da instrução FUNC.

O *Computador Pascal*, ao interpretar a instrução FUNC, aloca espaço para as variáveis locais de um subprograma incrementando o registrador *s* pelo tamanho do bloco de variáveis. Após isso, ele verifica se existe memória suficiente no vetor de execução para a criação da parte temporária para o subprograma e desvia para a parte de comandos do procedimento.

A instrução FUNC é definida através do seguinte algoritmo:

```
void FUNC(int varlength, int templength, int displ)
{
 s = s + varlength;
 if ((s + templength) > MAX)
   {
    printf(">>> ERRO - ESTOURO DE PILHA");
    running = 0;
   }
 else
  p = p + displ;
}
```

Ao final da execução de todas as instruções internas ao subprograma, o *Computador Pascal* deve remover o registro de ativação correspondente à ativação do mesmo. O novo endereço de base (registrador *b*) deve ser definido através do valor armazenado no *elo de controle* do subprograma e a próxima instrução após o término da execução do subprograma, identificada através do registrador *p*, deve ser recuperada através do endereço de retorno do subprograma. No caso particular das funções, o valor a ser retornado pela função deve ser ainda recuperado e armazenado no vetor de execução para utilizaçãofutura.

Todos esses passos, com exceção do último (esboçamos abaixo somente o caso mais simples da finalização de procedimentos), é efetuado por uma instrução ENDPROC, que utiliza como argumento o tamanho total dos parâmetros passados ao subprograma, e é definido segundo o algoritmo abaixo:

```
void ENDPROC(int paramlength)
 {
  s = b - paramlength - 1;
  p = vetor_execucao[b+2];
  b = vetor_execucao[b+1];
 }
```

2.2.4.7. A Execução do Programa

A inicialização do programa principal é semelhante à inicialização dos subprogramas, com a diferença de que o programa principal não contém nem endereço de retorno nem argumentos iniciais.

A instrução de inicialização do programa principal é uma instrução PROGRAM, cujos argumentos são semelhantes aos passados à instrução FUNC.

```
void PROGRAM(int varlength, int templength, int displ)
   {
    b = posicao_vetor_execucao - 1;
    s = b + 3 + varlength;
    if ((s + templength) > MAX)
     {
      printf(">>> ERRO - ESTOURO DE PILHA");
      running = 0;
     }
    else
    p = p + displ;
   }
```

Depois de carregado o programa em memória, o *Computador Pascal* associa uma variável booleana *Running* o valor *verdadeiro*, e executa a primeira instrução do programa, a instrução PROGRAM.

A última instrução do programa é uma instrução que associa o valor *falso* para *Running*, fazendo com que o *Computador Pascal* termine a execução do programa.

```
void ENDPROG()
 {
  running = 0;
 }
```

3. Especificação da linguagem PascalZIM!

Wirth projetou a linguagem Pascal em 1972 em Genebra, Suíça. A linguagem Pascal, que recebeu esse nome em homenagem ao filósofo francês Blaise Pascal, foi projetada para ser uma ferramenta de aprendizagem, mas descobriu-se que era tão poderosa que tornou-se uma linguagem de programação generalizada.

Em parte, isto é devido à *Borland* e à sua série de compiladores Turbo Pascal, introduzida em 1985. O compilador **Turbo Pascal** tornou a linguagem ainda mais popular, especialmente na plataforma PC, devido à uma combinação de simplicidade e poder.

O compilador **Pascal ZIM!**, concebido para fins educacionais, implementa a tradução de um subconjunto da linguagem Pascal, contendo as estruturas de dados e comandos mais utilizados por iniciantes, descritos nessa seção.

3.1. Identificadores

Um *identificador* na linguagem Pascal ZIM! é qualquer seqüência de caracteres que obedeça às seguintes regras:

- É iniciada por um caractere alfabético;
- Possui, após o primeiro caractere, uma seqüência de até 125 caracteres adicionais, que podem ser caracteres alfabéticos, numéricos ou ainda o caractere _ .
- Não seja um dos elementos pertencentes ao conjunto das *palavras reservadas da linguagem*.

Não há diferença quanto ao formato maiúsculo ou minúsculo na seqüência de caracteres que compõe um identificador. Por exemplo, PASCAL, pascal e Pascal são considerados identificadores idênticos.

3.2. Palavras Reservadas da Linguagem Pascal ZIM!

O conjunto das *palavras reservadas* da linguagem Pascal ZIM! é dado na tabela abaixo:

APPEND	*TEXTCOLOR*
ARRAY	*CHR*
ASSIGN	*RED*
BEGIN	*:*
BOOLEAN	*YELLOW*
CHAR	*LIGHTCYAN*
CLOSE	*LIGHTGREEN*
CLRSCR	*;*
CONST	*LENGTH*
DO	*TEXTBACKGROUND*
DOWTO	*BLINK*
ELSE	*>*
END	*NOT*
FALSE	*LIGHTGRAY*
FOR	*GREEN*
FUNCTION	*/*
GOTOXY	*LIGHTMAGENTA*
IF	*TEXT*
INTEGER	*MOD*
OF	*LIGHTBUE*
ORD	*MAGENTA*
PROCEDURE	***
PROGRAM	*OR*
READ	*.*
READKEY	*[*
READLN	*-*
REAL	*BROWN*
RECORD	*(*

REPEAT	*,*
RESET	*=*
REWRITE	*DARKGRAY*
STRING	*AND*
THEN	*<*
TO	*BLUE*
TRUE	*CYAN*
TYPE	*)*
UNTIL	*LIGHTRED*
VAR	*WHITE*
WHILE	*DIV*
WRITE	*]*
WRITELN	*EOF*

3.3. O formato básico de um programa Pascal ZIM!

Um programa escrito na linguagem **Pascal ZIM!** é basicamente dividido em três partes:

- O cabeçalho, onde é dado um nome ao programa;
- A *seção de definição e declaração de dados;*
- A *seção de comandos*, que define as ações executadas pelo programa.

O cabeçalho de um programa é iniciado com a palavra reservada **Program**, seguido de um nome identificador do programa, e um sinal de ponto e vírgula. Por exemplo,

Program *MeuPrograma* **;**

A *seção de definição e declaração de dados* segue o cabeçalho do programa, e é o local onde são definidas as constantes e tipos que serão usados dentro do programa. Nesta seção também são declaradas também as variáveis globais do programa, e definidas as funções e procedimentos que podem ser utilizados pelo programa. Essa seção consiste das seguintes partes:

- A parte para declaração de constantes;
- A parte para definição de tipos;
- A parte para declaração de variáveis;
- A parte para definição de funções e procedimentos;

A definição de cada uma dessas partes é opcional, mas deve seguir a ordem estabelecida. Por exemplo, uma função não pode ser definida antes da declaração de uma variável**.**

Em seguida, deve ser elaborada a *seção de comandos*. Esta seção é iniciada com a palavra reservada **Begin** e terminada com a palavra reservada **End**, seguida de um ponto. Entre as palavras **Begin** e **End** devem ser colocados os comandos do programa.

O formato genérico de um programa na linguagem **Pascal ZIM!** segue a seguinte especificação:

Program *identificador* ;

< Seção de definições e declarações >

Begin
< Comandos >
End.

3.4. Tipos

As variáveis declaradas em um programa escrito na linguagem **Pascal ZIM!** são especificadas através de um *tipo*. Um *tipo* é uma especificação que:

- Indica o espaço em memória necessário para o armazenamento de um dado (ou um conjunto de dados)
- Define o conjunto de operações que pode ser aplicada a um dado (ou um conjunto de dados)

Os tipos, na linguagem **Pascal ZIM,** são classificados em três categorias:

- Tipos predefinidos
- Tipos estruturados
- Tipos definidos

3.4.1. Tipos Predefinidos

Os tipos predefinidos na linguagem **Pascal ZIM!** são os seguintes:

- **Boolean**

  O tipo **boolean** define dois valores: **FALSE** ou **TRUE**.
  Um dado do tipo booleano ocupa um byte de espaço na memória.

- **Char**

  O tipo **char** define os elementos do conjunto de caracteres que compõem o alfabeto ASCII, e os caracteres representados pelos códigos de 128 a 255.
  Um dado do tipo **char** ocupa um byte de espaço na memória.

- **Integer**

  O tipo **integer** define os valores inteiros no intervalo de -32767 até 32767.
  Um dado do tipo **integer** ocupa dois bytes de espaço na memória.

- **Real**

  O tipo **real** define os valores reais definidos no intervalo de 3.4*(10**-38) até 3.4 * (10**+38).
  Um dado do tipo **real** ocupa quatro bytes de espaço na memória.

- **String**

  O tipo **string** define uma cadeia de caracteres. Se nenhum tamanho for especificado, armazena uma sequência contendo até 255 caracteres, ocupando 255 bytes de espaço na memória.

  Uma cadeia de caracteres de tamanho definido (contendo menos de 255 caracteres), onde o tamanho especifica o número máximo de caracteres contidos na cadeia, deve ser indicada entre colchetes, logo após a palavra reservada **string**, Por exemplo, **string [**6**]** define uma cadeia de 6 caracteres.
  Uma cadeia de caracteres definida com *n* caracteres ocupa *n* bytes de espaço na memória.

3.4.2. Tipos Estruturados

Os tipos predefinidos podem ser organizados em tipos mais complexos, denominados *tipos estruturados*. O compilador **Pascal ZIM!** oferece dois destes tipos:

- Vetores
- Registros

3.4.2.1. Vetores

Um vetor contém um número fixo de dados agrupados por um mesmo tipo, que pode ser qualquer um dos tipos predefinidos (**integer**, **char, boolean ou string)**, um tipo *vetor*, um tipo *registro* definido ou ainda um tipo definido pelo usuário.

O número de elementos de um vetor é determinado pelo seus índice, especificado entre colchetes por duas constantes ordinais, separadas por dois pontos.

A sintaxe para definição de vetores segue o seguinte formato:

array[$limite_1$ **..** $limite_2$] **of** *tipo*;

Onde:

- **array** e **of** são palavras reservadas da linguagem **Pascal ZIM!** usadas para declarar vetores
- $limite_1$ e $limite_2$ são constantes ordinais;
- *Tipo* define o tipo básico do vetor

Por exemplo, a declaração abaixo define um vetor do tipo inteiro, identificado por *Dias*:

```
Var
  Dias : array [ 1 .. 24 ] of  integer;
```

A referência ao elemento de um vetor identificado pelo índice *x* é dado da seguinte forma:

Nome da variável **[** *x* **]**

onde *Nome da variável* é uma variável do tipo *vetor*.

3.4.2.2. Vetores com várias dimensões

Vetores podem ter mais de uma dimensão. Nesses casos, cada dimensão nova é declarada de acordo com as regras do item anterior, e as *n* dimensões do vetor são separadas por vírgulas.

A sintaxe para definição vetores *n*-dimensionais segue o seguinte formato:

array[$limite_1$ **..** $limite_2$ **,** $limite_3$ **..** $limite_4$ **,** ... , $limite_{n-1}$ **..** $limite_n$ **]** **of** *tipo*;

Por exemplo, a declaração abaixo define um vetor de duas dimensões do tipo inteiro:

```
Var
  Matriz : array [ 1 .. 10, 1.. 20 ] of  integer;
```

3.4.2.3. Registros

Um *registro* é um tipo composto por um conjunto de dados de tipos diferentes, onde cada um dos dados é definido como sendo um *campo.*

Um tipo *registro* é declarado pela palavra reservada **record**, seguida por uma série de declaração de *campos*. A palavra reservada **end** seguida de um ponto e vírgula encerra a definição de um registro.

A sintaxe genérica para definição de *registros* segue o seguinte formato:

Record
Identificador de campo **:** *tipo*;
Identificador de campo **:** *tipo*;

Identificador de campo **:** *tipo*;
End;

Exemplo. Declaração de um registro simples:

```
Var
  Dados : Record
            Numero : integer;
            Caracter : char;
            Preenchido : boolean;
          End;
```

Exemplo. Declaração de um registro contendo registros aninhados:

```
Var
  Dados2 : Record
             Numero : integer;
             Dado :   Record
                        Caracter : char;
                      End;
             Preenchido: boolean;
           End;
```

A referência a um campo de um registro é feita através do nome da variável do tipo registro seguida por um ponto e pelo nome do campo, como por exemplo,

Dados.Numero

3.4.3. Tipos definidos

A definição de um novo tipo é feita na *seção de definição de tipos*, contida na *seção de definição e declaração de dados*.

O início da *seção de definição de tipos* é indicada através da palavra reservada **Type**. A definição de novos tipo de dados é dada segundo a seguinte sintaxe:

Type
Identificador de tipo = tipo construído **;**

Neste caso, *tipo construído* é um tipo estruturado *vetor* ou *registro*.

A definição de novos tipos de dados não tem nenhum efeito em um programa até que seja declarada uma variável tendo o tipo definido. A palavra reservada **Type** deve aparecer uma única vez dentro da *seção de definição e declaração de dados*.

3.5. Declaração de constantes

As constantes são declaradas na *seção de declaração de constantes*, contida na *seção de definição e declaração de dados*. O início da *seção de declaração de constantess* é indicada através da palavra reservada **const**.

A palavra reservada **const** marca o início da seção de definições de constantes, e deve aparecer somente uma única vez dentro da seção de declarações e definições**.**

A sintaxe para declaração de constantes segue o seguinte formato:

Const
Identificador$_1$*, identificador*$_2$**, ... ,** *identificador*$_n$ *= constante* **;**

Nesta declaração, *constante* pode ser uma constante inteira, real, uma cadeia de caracteres ou um único caractere.

Por exemplo, a declaração abaixo declara uma constante inteira cujo valor é 10

Const
Dez = 10 **;**

3.6. Declaração de variáveis

A declaração de uma variável faz com que o compilador reserve uma quantidade de espaço em memória suficientemente grande para armazenar um tipo de dados, além de associar também um "nome" a esta posição de memória.

As variáveis são declaradas na *seção de declaração de variáveis*, contida na *seção de definição e declaração de dados*. O início da *seção de declaração de variáveis* é indicada através da palavra reservada **var**.

A palavra reservada **Var** deve aparecer somente uma única vez dentro da *seção de definição e declaração de dados*.

A sintaxe para declaração de variáveis segue o seguinte formato:

Var
Identificador$_1$**,** *identificador*$_2$**, ... ,** *identificador*$_n$ **:** *tipo* **;**

Nesta declaração, cada identificador (ou grupo de identificadores separados por vírgulas).

Por exemplo, a declaração abaixo declara três variáveis, dos tipos inteiro, caractere e booleano.

Var
inteiro**: integer;**
caracter **: char;**
booleano**: boolean;**

3.7. Expressões

O termo *expressão* se refere a qualquer combinação de uma ou mais constantes ou identificadores de variáveis, com um ou mais *operadores*. As constantes e variáveis que aparecem numa expressão são chamadas de *operandos*.

Quando mais de um operador aparece numa expressão, a seqüência de cálculo efetuada pelo compilador depende da precedência definida para cada operador da linguagem, onde o operador com mais alta precedência é o primeiro a capturar seus operandos. No caso de dois ou mais operadores terem o mesmo nível de precedência, o cálculo é feito da esquerda para a direita.

São definidos quatro níveis de precedência para os operadores da linguagem, definidos abaixo em ordem decrescente:

1. - (menos unário), **not**
2. *****, **Div**, **mod**, **and**
3. **+**, **-**, **or**
4. =, <>, <, >, <=, >=

Parênteses alteraram a ordem de precedência de um conjunto de operadores, forçando o programa a calcular a expressão dentro dos parênteses antes das outras.

Por exemplo, a adição é calculada antes da multiplicação em 5* (3+4).

3.8. Operadores definidos na linguagem

Grande parte da manipulação de dados que ocorre na *seção de comandos* é feita através pelo uso de um *operador*. Um *operador* na linguagem **Pascal ZIM!** pertence a uma dentre três categorias básicas:

- *operadores aritméticos*
- *operadores lógicos*
- *operadores relacionais*

3.8.1. Operadores Aritméticos

Operadores aritméticos são usados em expressões aritméticas.
Os operadores aritméticos definidos na linguagem **Pascal ZIM!** são:

- \- (Menos Unário)
  Tipo de operando permitido: inteiro ou real
  Operação executada: Inverte o valor numérico do operando

- **DIV**
  Tipo de operandos permitidos: inteiros
  Operação executada: O operando à esquerda do DIV é dividido pelo operando à sua direita, sendo o resultado desta operação um valor inteiro resultante da divisão.

- **MOD**
  Tipo de operandos permitidos: inteiros
  Operação executada: O operando à esquerda do MOD é dividido pelo operando à sua direita, sendo o resultado desta operação o resto inteiro da divisão.

- \+
  Tipo de operandos permitidos: inteiros, reais, cadeias de caracteres
  Operação executada: No caso de inteiros e reais o operando à esquerda do + é somado ao operando a sua direita, sendo o tipo do resultado dessa operação dependente de seus operandos:

  - Se os dois operandos são inteiros, o resultado da soma é inteiro.
  - Se os dois operandos são reais, o resultado da soma é real.
  - Se os um dos operandos é real, e o outro é inteiro, o resultado da soma é real.

  No caso dos operandos serem ambos cadeias de caracteres o resultado da operação é dada pela cadeia obtida pela concatenação da cadeia dada pelo segundo operando com a cadeia dada pelo primeiro operando.

- \-
  Tipo de operandos permitidos: inteiros, reais
  Operação executada: O operando à esquerda do - é subtraído do operando a sua direita, sendo o tipo do resultado dessa operação dependente de seus operandos:

  - Se os dois operandos são inteiros, o resultado da operação é inteiro.
  - Se os dois operandos são reais, o resultado da operação é real.
  - Se os um dos operandos é real, e o outro é inteiro, o resultado da operação é real.

- *****
  Tipo de operandos permitidos: inteiros, reais
  Operação executada: O operando à esquerda do * é multiplicado pelo operando a sua direita, sendo o tipo do resultado dessa operação dependente de seus operandos:

  - Se os dois operandos são inteiros, o resultado da operação é inteiro.
  - Se os dois operandos são reais, o resultado da operação é real.
  - Se os um dos operandos é real, e o outro é inteiro, o resultado da operação é real.

- /
  Tipo de operandos permitidos: inteiros, reais
  Operação executada: O operando à esquerda do / é dividido pelo operando a sua direita, sendo o resultado dessa operação real.

3.8.2. Operadores Lógicos

Operadores lógicos são usados em expressões lógicas, com operandos do tipo booleano.
Os operadores lógicos definidos na linguagem **Pascal ZIM!** são:

- **not**
  Operação executada: O operador inverte o valor verdade de um operando booleano.

- **and**
  Operação executada: É feito um **and** lógico entre os dois operando do operador, sendo o resultado da operação verdadeiro quando ambos operandos são verdadeiros.

- **or**
  Operação executada: É feito um **or** lógico entre os dois operando do operador, sendo o resultado da operação verdadeiro se um dos operandos for verdadeiro.

A tabela verdade para os três operadores lógicos é mostrada a seguir:

Primeiro Operando	Operador	Segundo Operando	Resultado
Verdadeiro	Not	----	Falso
Falso	Not	----	Verdadeiro
Verdadeiro	And	Verdadeiro	Verdadeiro
Verdadeiro	And	Falso	Falso
Falso	And	Verdadeiro	Falso
Falso	And	Falso	Falso
Verdadeiro	Or	Verdadeiro	Falso
Verdadeiro	Or	Falso	Verdadeiro
Falso	Or	Verdadeiro	Verdadeiro
Falso	Or	Falso	Falso

3.8.3. Operadores Relacionais

Operadores relacionais são usados em expressões condicionais. Os tipos de operandos permitidos para esses operandos são:

- Ambos operandos do mesmo tipo primitivo (**integer**, **char**, **boolean**, **char** ou **string**)
- Operandos de tipos diferentes, onde:

  - Um operando é do tipo **integer** e outro do tipo **real**
  - Um operando é do tipo **string** e outro do tipo **char**

O resultado para expressões envolvendo operadores relacionais é um valor booleano, definido de acordo com a tabela a seguir.

Operador	Resultado
=	Verdadeiro se os dois operandos para o operador forem iguais Falso em caso contrário.
<>	Verdadeiro se os dois operandos para o operador forem diferentes. Falso em caso contrário.
<	Verdadeiro se o operando à esquerda do operador for menor que o operando à direita Falso em caso contrário.
<=	Verdadeiro se o operando à esquerda do operador for menor ou igual o operando à direita Falso em caso contrário
>	Verdadeiro se o operando à esquerda do operador for maior do que o operando à direita Falso em caso contrário.
>=	Verdadeiro se o operando à esquerda do operador for maior ou igual que o operando à direita Falso em caso contrário.

3.9. Comandos

Os comandos são definidos na *seção de comandos*, e podem ser classificados em sete categorias:

1. Comandos de atribuição
2. Comandos compostos
3. Comandos de repetição
4. Comandos condicionais
5. Comandos para tratamento de arquivos
6. Comandos de entrada e saída
7. Comandos auxiliares

O ponto e vírgula é usado na linguagem **Pascal ZIM!** como separador de comandos, servindo para separar um comando dos comandos subsequentes.

3.9.1. Comandos de Atribuição

Um comando de atribuição é definido através da seguinte sintaxe:

Variável **:=** *Expressão*

onde:

- *Variável* é uma variável
- *Expressão* é uma expressão

A atribuição é definida apenas para tipos *predefinidos*. O tipo da expressão deve ser igual ao tipo da variável, com exceção de dois casos especiais onde:

- A variável é do tipo **real** e a expressão é do tipo **integer**
- A variável é do tipo **string** e a expressão é do tipo **char**

Por exemplo, sendo dados

```
Var
  Item: integer;
  Saída : boolean;
  Soma, Valor : real;
  Caractere : char;
  Cadeia : string
```

podemos ter os seguintes comandos de atribuição:

```
Item := 0;
Saida := FALSE;
Soma := Valor1 + Valor2;
Caracter:= 'a';
Cadeia := 'Isso é uma cadeia de caracteres';
Soma := 9;
Cadeia := 'a';
```

3.9.2. Comandos Compostos

Além de marcar o início e o fim da *seção de comandos*, o par **begin** e **end** formam um par de instruções usado para combinar qualquer número de comandos em um *comando composto*.

Um *comando composto* é formado por qualquer tipo de comandos, incluindo outros comandos compostos.

3.9.3. Comandos de Repetição

Os comandos de repetição permitem a repetição da execução de um conjunto de comandos. Os comandos de repetição definidos na linguagem **Pascal ZIM!** são os seguintes:

- **Repeat**
- **While**
- **For**

O comando repeat executa repetidamente uma sequência de comandos até que uma condição, dada através da avaliação da uma expressão booleana, seja *verdadeira*.

A sintaxe de um comando **repeat** é

repeat $comando_1$**; ...;** $comando_n$**; until** *expressão*

onde *expressão* é uma expressão condicional.

Os comandos internos ao **repeat** são executados ao menos uma vez, pois a condição de parada da repetição é avaliada somente após a primeira repetição.

Exemplo:

```
repeat
  K := I mod J;
  I := J;
  J := K;
until J = 0;
```

O comando while se assemelha ao comando **repeat**, com a diferença de que a condição para a execução repetida de comandos é avaliada antes da execução de qualquer comando interno da repetição. Dessa forma, se a condição inicial para o **while** for *falsa*, a sequência de comandos definidos para o **while** não será executada nenhuma vez.

A sintaxe de um comando **while** é:

while *expressão* **do** *comando*

onde *expressão* é uma expressão condicional e *comando* pode ser um comando composto.

Exemplo.

```
while Data[I] <> X do I := I + 1;
```

O comando for, diferentemente dos comandos de repetição **repeat** e **while**, permite que uma sequência de comandos seja executada um número definido de vezes.
A sintaxe de um comando **for** é**:**

for *contador* **:=** *ValorInicial* **to** *ValorFinal* **do** *comando*

ou

for *contador* **:=** *ValorInicial* **downto** *ValorFinal* **do** *comando*

onde:

- contador é uma variável do tipo **integer**
- *ValorInicial* e *ValorFinal* são expressões inteiras
- *comando* é um comando, podendo ser um comando composto

O comando **for** associa o valor *ValorInicial* à variável *contador*, e então executa a sequência de comandos *comando* repetidamente, incrementando ou decrementando *contador* após cada iteração (o **for**...**to** incrementa a variável *contador*, enquanto **for** ... **downto** decrementa a variável *contador*). Quando *contador* armazena um valor maior que *ValorFinal* a repetição termina.

Exemplo.

```
for I := 2 to 63 do
  if Data[I] > Max then
    Max := Data[I];
```

3.9.4. Comandos Condicionais

Estes comandos permitem restringir a execução de um certo conjunto de comandos. O comando condicional definido na linguagem **Pascal ZIM!** é o **if**, que pode ter duas formas:

- **if..then**
- **if...then...else.**

A sintaxe de um comando **if...then** é:

if *expressão* **then** *comando*

onde *expressão* é uma expressão condicional e *comando* pode ser um comando composto.

O funcionamento do comando é simples: se *expressão* for TRUE, então *comando* é executado; caso contrário *comando* não é executado.

Exemplo:

```
if J <> 0 then Result := I/J;
```

A sintaxe de um comando **if...then...else** é:

if *expressão* **then** *comando*$_1$ **else** *comando*$_2$

onde *expressão* é uma expressão condicional e *comando*$_1$ *e comando*$_2$ podem ser comandos compostos.

O funcionamento do comando é simples: se *expressão* for TRUE, então *comando*$_1$ é executado; caso contrário, *comando*$_1$ é executado.

Exemplo.

```
if J = 0 then
  write(J)
else
  write(M)
```

Em uma série de comandos **if** aninhados a cláusula **else** está ligada ao **if** mais próximo no aninhamento. Uma seqüência de comandos como:

if *expressão*$_1$ **then if** *expressão*$_2$ **then** *comando*$_1$ **else** *comando*$_2$**;**

É reconhecido pelo compilador da seguinte forma:

if *expressão*$_1$ **then** [**if** *expressão*$_2$ **then** *comando*$_1$ **else** *comando*$_2$]**;**

3.9.5. Comandos para tratamento de arquivos

Os comandos para tratamento de arquivos incluem comandos para identificação, abertura e fechamento de arquivos.

O uso de arquivos na linguagem **Pascal ZIM!** é feito através da definição de variáveis de um tipo especial, TEXT, que identifica uma variável do tipo arquivo. Essa variável deve ser definida na parte para *declaração de variáveis* na *seção de definição e declaração de dados.*

Os comandos para tratamento de arquivos definidos na linguagem **Pascal ZIM!** são os seguintes:

- **Assign**
- **Reset**
- **Rewrite**
- **Append**
- **Close**

O comando assign associa o nome de um arquivo externo a um variável definida com o tipo TEXT. A sintaxe para o comando é:

assign (*VariavelArquivo* **,** *NomeArquivo* **);**

onde:

- *VariavelArquivo* é uma variável definida com o tipo TEXT
- *NomeArquivo* é uma cadeia de caracteres contendo o nome do arquivo associadoDo not use AssignFile on a file variable that is already open.

O comando reset abre um arquivo já existente. A sintaxe do comando é:

Reset (*VariavelArquivo* **);**

onde *VariavelArquivo* é uma variável definida com o tipo TEXT.

O comando rewrite cria e abre um arquivo. A sintaxe para o comando é:

Rewrite (*VariavelArquivo* **);**

onde *VariavelArquivo* é uma variável definida com o tipo TEXT.

O comando append abre um arquivo já existente para escrita no seu final. A sintaxe para o comando é:

Append (*VariavelArquivo* **);**

onde *VariavelArquivo* é uma variável definida com o tipo TEXT.

O comando close fecha um arquivo. A sintaxe para o comando é:

Close (*VariavelArquivo* **);**

onde *VariavelArquivo* é uma variável definida com o tipo TEXT.

3.9.6. Comandos de Entrada e Saída

Os comandos usados para leitura e escrita de dados são definidos na linguagem **Pascal ZIM!** através de quatro comandos:

- Read
- Readln
- Write
- Writeln

Os comandos **read** e **readln** são usados para ler o valor de uma variável de um dispositivo de entrada de dados. A diferença entre os dois comandos é que o comando **readln** processa uma quebra de linha após a leitura do valor de uma variável.

Os comandos **write** e **writeln** são usados para imprimir o valor de uma sequência de expressões em um dispositivo de saída de dados. A diferença entre os dois comandos é que o comando **writeln** processa uma quebra de linha após imprimir o valor de uma sequência de expressões.

A leitura e escrita de dados pode ser direcionada para um arquivo, identificado através de uma variável do tipo TEXT.

A sintaxe de um comando **read** para leitura a partir do teclado é:

READ (V*ariável* **);**

A sintaxe de um comando **read** para leitura a partir de um arquivo é:

READ (*VariavelArquivo,* V*ariável* **);**

onde *VariavelArquivo* é uma variável definida com o tipo TEXT.

A sintaxe de um comando **write** para impressão na tela de uma sequência de expressões é:

WRITE ($expressão_1$ **,** $expressão_2$ **, ... ,** $expressão_n$ **)**

A sintaxe de um comando **write** para impressão em arquivo de uma sequência de expressões é:

WRITE (*VariavelArquivo,* $expressão_1$ **,** $expressão_2$ **, ... ,** $expressão_n$ **)**

onde *VariavelArquivo* é uma variável definida com o tipo TEXT.

3.9.7. Comandos Auxiliares

É definido, na linguagem **Pascal ZIM!**, um conjunto de comandos auxiliares, a saber:

- **Textcolor**
- **TextBackground**
- **Clrscr**
- **Readkey**

O comando textcolor define a cor usada para impressão de texto na tela. Sua sintaxe é dada por:

Textcolor (*ListaDeCores* **)**

onde *ListaDeCores* pode ser uma constante inteira ou qualquer uma dentre as cores seguintes:

- BLUE
- GREEN
- CYAN
- RED
- MAGENTA
- BROWN
- LIGHTGRAY
- DARKGRAY
- LIGHTBLUE
- LIGHTGREEN
- LIGHTCYAN
- LIGHRED
- LIGHMAGENTA
- YELLOW
- WHITE
- BLACK

Pode ser utilizada uma combinação das mesmas, dada pelo sinal de adição +, como em

Textcolor (RED + BLUE)

O comando textbackground define a cor de fundo usada na impressão de texto na tela. Sua sintaxe é:

Textbackground (*ListaDeCores* **)**

onde *ListaDeCores* é definido como em **textcolor**

O comando clrscr limpa a tela de impressão. Sua sintaxe é:

Clrscr;

O comando readkey solicita a leitura de um caracter. Sua sintaxe é:

readkey;

Além desses comandos, são implementados no compilador um conjunto de funções auxiliares:

- **Chr**
- **Ord**
- **Length**
- **Eof**

A função auxiliar chr recebe como parâmetro um inteiro e retorna o caracter ASC II correspondente ao código identificado com esse inteiro.

A função auxiliar ord recebe como parâmetro um caractere e retorna o inteiro correspondente ao código ASC do caracter.

A função auxiliar length recebe como parâmetro uma cadeia de caracteres e retorna um inteiro denotando o número de caracteres da cadeia.

A função auxiliar eof recebe como parâmetro uma variável do tipo TEXT e retorna *verdadeiro* se o arquivo denotado pela variável está no seu final, e *falso* em caso contrário.

3.10. Subprogramas

Subprogramas são partes de um programa contendo um *cabeçalho*, uma *seção de definição e declaração de dados* e uma *seção de comandos*.

Os subprogramas são definidos na *seção de definição e declaração de dados*, e podem ser de dois tipos:

- Procedimentos
- Funções

A diferença essencial entre *procedimentos* e *funções* é o fato de que as *funções* retornam valores, enquanto os procedimentos não. O valor retornado por uma função é qualquer um dos tipos primitivos **char**, **integer**, **boolean**, **real** ou **string**.

A ativação de um subprograma é feita através de uma *chamada* ao subprograma. Quando um subprograma é *chamado* a sequência de comandos definida na *seção de comandos* do subprograma é executada, após o qual a execução do programa retorna para instrução seguinte à chamada do subprograma. Um subprograma é chamado através do nome que o define.

Os subprogramas podem ser embutidos; isto é, um subprograma pode ser definido dentro do bloco de declarações de um outro subprograma. Um subprogama embutido pode ser chamado somente pelo subprograma que o contém, sendo visível somente para o subprograma que o contém.

A chamada a um procedimento é reconhecida pelo compilador como um comando, enquanto que uma chamada á uma função é reconhecida como uma expressão.

Funções e Procedimentos

A declaração de procedimentos e funções difere apenas no cabeçalho. O cabeçalho de um procedimento segue a seguinte sintaxe:

Procedure *NomeProcedimento***;**

onde *NomeProcedimento* define o nome do procedimento

O cabeçalho de uma função segue a seguinte sintaxe:

Function *NomeFunção* **:** *tipo***;**

onde:

- *NomeFunção* define o nome da função
- *tipo* define o tipo de dado retornado pela função, que pode ser um dos tipos primitivos **char**, **integer, boolean**, **real** ou **string**.

A *seção de definição e declaração de dados* segue o cabeçalho do subprograma, e é o local onde são definidas as constantes e tipos passíveis de uso. Também nessa seção são declaradas as variáveis locais do subprograma, e definidas funções e procedimentos que podem ser utilizados pelo subprograma.

A *seção de comandos* segue a *seção de definição e declaração de dados*. É iniciada com a palavra reservada **Begin** e terminada com a palavra reservada **End**, seguida de um ponto e vírgula. Entre as palavras **Begin** e **End** são colocados os comandos da função.

A forma de atribuir um valor de retorno para uma função é atribuindo ao nome identificador da função o valor a ser retornado pela função, em alguma parte da *seção de comandos* da função.

Parâmetros

Um subprograma pode receber parâmetros. A especificação dos parâmetros passados a um subprograma deve ser especificada no cabeçalho do subprograma, dentro de parênteses, seguindo o identificador do subprograma.

A especificação de cada lista de identificadores de parâmetros deve estar separada por um sinal de ponto e vírgula, e declarada como tendo um dos tipos predefinidos da linguagem **Pascal ZIM!**, ou um tipo definido pelo usuário.

A sintaxe do cabeçalho de uma função contendo parâmetros é:

Function *identificador* ($parâmetro_1$: ***tipo*** ; $parâmetro_2$: ***tipo*** ; ... ; $parâmetro_n$: ***tipo***) **:** *tipo*;

A passagem de parâmetros na linguagem **Pascal ZIM!** pode ser de dois tipos, a saber:

- Passagem por valor
- Passagem por referência

No primeiro caso, o parâmetro assume o valor passado pela rotina de chamada, e no segundo caso o parâmetro assume o endereço da variável passada pela rotina de chamada.

A passagem por referência é diferenciada da passagem por valor pela presença da palavra reservada **var** antes do nome identificador do parâmetro, como em.

Procedure exemplo **(var** ParametroPassadoPorReferencia **: integer);**

A chamada a esse procedimento poderia ser feita, através de um comando tal como:

exemplo **(** *x* **) ;**

onde *x* é uma variável do tipo inteiro.

Funções Recursivas

Uma função pode chamar a si mesma de dentro de sua própria *seção de comandos*. Quando isto é feito, a função é denominada *função recursiva*.

O uso de *funções recursivas* consegue fornecer soluções elegantes para certos tipos de programas, como mostrado no exemplo abaixo, que calcula, para um número inteiro n, seu fatorial:

```
function fat (n :integer ) : integer;
 begin
  if n > 1 then
    fat :=  n * fat (n-1)
  else
    fat := 1;
 end;
```

3.11. Comentários

Os comentários são usados dentro de um programa com o fim de documentar o programa, e não afetam sua execução.

Comentário devem estar inclusos entre chaves **{ }**.

Exemplo.

```
Program teste;  { Esse é meu programa de teste }
 Begin
   Write('Olá, mundo!');   { Imprime a mensagem 'Olá, mundo!' }
 End.
```

3.12. Regras de escopo

As regras de escopo definidas para um programa escrito na linguagem **Pascal-ZIM** obedecem às seguintes convenções:

- Um identificador definido na *seção de definição e declaração de dados* do programa principal é acessível por todos subprogramas;
- Um identificador definido na *seção de definição e declaração de dados* de um subprograma é acessível na *seção de comandos* do subprograma na qual foi definido e também na *seção de comandos* de todos subprogramas declarados na sua *seção de definição e declaração de dados*, a menos que esse identificador seja redefinido.
- Os identificadores definidos em um subprograma função não existem antes nem depois da chamada àquela função e, por isso, não podem ser referenciados nesses momentos.

3.13. Tratamento de overFlow

O tratamento de overflow é feito para constantes inteiras, reais e também para cadeias de caracteres. O tratamento consiste em verificar se, durante uma determinada operação, uma constante ultrapassa o valor máximo permitido para constantes do tipo em questão.

O intervalo de valores válidos para constantes numéricas é:

- Para constantes reais: 3.4 * (10**-38) à 3.4 * (10**+38)
- Para constantes inteiras: 32767 à -32767.

O tratamento de *overflow* dado às cadeias de caracteres depende do tamanho da cadeia, especificado quando da declaração de uma variável. Variáveis do tipo **string** podem ocupar até 255 posições de memória, enquanto variáveis declaradas com o tipo **string [** *n* **]** podem ocupar até *n* posições de memória.

4. Descrição da Implementação

Neste capítulo descreveremos detalhes utilizados na implementação do **Pascal ZIM!**.

4.1. Analisador Léxico

Palavras reservadas, operadores e símbolos identificados pelo analisador léxico

O conjunto das *palavras reservadas e operadores* Pascal identificados pelo analisador léxico é listado na tabela abaixo.

APPEND	TEXTCOLOR
ARRAY	CHR
ASSIGN	RED
BEGIN	:
BOOLEAN	YELLOW
CHAR	LIGHTCYAN
CLOSE	LIGHTGREEN
CLRSCR	;
CONST	LENGTH
DO	TEXTBACKGROUND
DOWTO	BLINK
ELSE	>
END	NOT
FALSE	LIGHTGRAY
FOR	GREEN
FUNCTION	/
GOTOXY	LIGHTMAGENTA
IF	TEXT
INTEGER	MOD
OF	LIGHTBUE
ORD	MAGENTA
PROCEDURE	*
PROGRAM	OR
READ	.
READKEY	[
READLN	-
REAL	BROWN
RECORD	(
REPEAT	,
RESET	=
REWRITE	DARKGRAY
STRING	AND
THEN	<
TO	BLUE
TRUE	CYAN
TYPE	)
UNTIL	LIGHTRED
VAR	WHITE
WHILE	DIV
WRITE	]
WRITELN	

Função de hash usada na manutenção da tabela de símbolos

Em vários experimentos, a função de *hash hashpjw* (figura 4.1.2.1.), implementada no compilador C de P. J. Weinberger, mostrou-se eficiente [AHO, 189]. Os tamanhos de tabela testados incluíam os primeiros números primos maiores do que 100, 200, ... , 1500. A função *hashpjw* é computada começando com *h* = 0. Para cada caractere *c*, deslocam-se os *bits* de *h* 4 posições à esquerda e adiciona-se *c*. Se qualquer um dos 4 *bits* de mais alta ordem de *h* for 1, deslocam-se os quatro *bits* em 24 posições à direita, faz-se o *ou exclusivo* dos mesmos com *h* e zera-se qualquer *bit* de mais alta ordem que seja 1.

```
int hashpjw(char *cadeia)
  {
   char *p;
   unsigned h =0, g;

   for ( p = cadeia; *p != EOS ; p = p+1)
     {
      h = (h << 4) + (*p);
      if (g == (h&0xf0000000))
        {
         h = h ^ (g >> 24);
         h = h ^ g;
        }
     }
   return h % 211;
  }
```

Figura 4.1.2.1. Função de *hash hashpjw*, escrita em C.

Por questão de otimização, cadeia numéricas reais são armazenadas em uma tabela de símbolos independente durante a análise léxica. Ambas tabelas fazem uso da função de *hash hashpjw* para o armazenamento e pesquisa de uma cadeia particular.

Atributos passados ao analisador sintático pelo analisador léxico

Para cada *token* identificado pelo analisador léxico são retornados ao analisador sintático dois atributos, identificados como a seguir:

- Para *Cadeias De Caracteres* e *Identificadores*

  ATRIB1: valor inteiro obtido com a aplicação da função de hash **hashpjw** à cadeia/identificador
  ATRIB2: valor inteiro indicando posição da cadeia/ identificador na lista hash

- Para *Caracteres* e *Números Inteiros*

  ATRI2B1: valor inteiro do caracter/inteiro
  ATRIB2: -1

- Para *Números Reais*

  ATRIB1: valor inteiro obtido com a aplicação da função de hash **hashpjw** a cadeia numérica.
  ATRIB2: valor inteiro indicando posição da cadeia numérica na lista hash numérica.

- Para *palavras reservadas* e *operadores* da linguagem **Pascal ZIM!**

  Nenhum atributo é retornado.

Identificação de padrões durante o reconhecimento dos tokens

Abaixo são listadas as *expressões regulares* que definem o padrões dos *tokens* reconhecidos pelo analisador léxico.

- *Token* ***Id*** *(Identificador)*

  Id → [A-Za-z_] [A-Za-z_0-9]*

- *Token* ***Num*** *(Número Inteiro ou Real)*

  Dígito → 0 | 1 | ... | 9
  Dígitos → **Dígito**$^{+}$
  Fração_Opcional → (. **Dígitos**) ? | .
  Expoente_Opcional → (E (+ | -) ? **Dígitos**) ?
  Num → **Dígitos Fração_Opcional Expoente_Opcional**

- *Token* ***Cadeia*** *(Cadeia de Caracteres)*

  ASC → Qualquer caracter ASCII (com exceção do caracter de avanço de linha)
  Cadeia → " ASC* "

- *Token* ***Caractere***

  ASC → Qualquer caracter ASCII (com exceção do caracter de avanço de linha)
  Cadeia → " ASC "

- *Palavras reservadas, operadores e símbolos* da linguagem **Pascal ZIM!**

  Identificado a partir dos caracteres que soletram a *palavra reservada* ou *operador*.

Tratamento dado a comentários e espaços em branco

Comentários e espaços em branco são ignorados pelo analisador léxico. As expressões regulares que definem esse tratamento são listadas abaixo.

- Tratamento de *Comentários*

  ASC → Caracteres ASCII
  Comentário → { ASC* }

- Tratamento de *Espaços em Branco, Avanços de Linha e Tabulações*

  Delimitadores → Espaço em Branco | Tabulação | Avanço de Linha
  WS → Delimitadores*

AFD para o reconhecimento dos tokens

Um autômato finito foi utilizado para implementar o analisador léxico. A Figura 4.1.6.1. ilustra o AFD implementado para a identificação dos *tokens* da linguagem **Pascal ZIM!**. Os estados rotulados por *Acumula_Identificador*, *Acumula_Número* e *Acumula_Cadeia*, nesta figura são descritos detalhadamente em seguida.

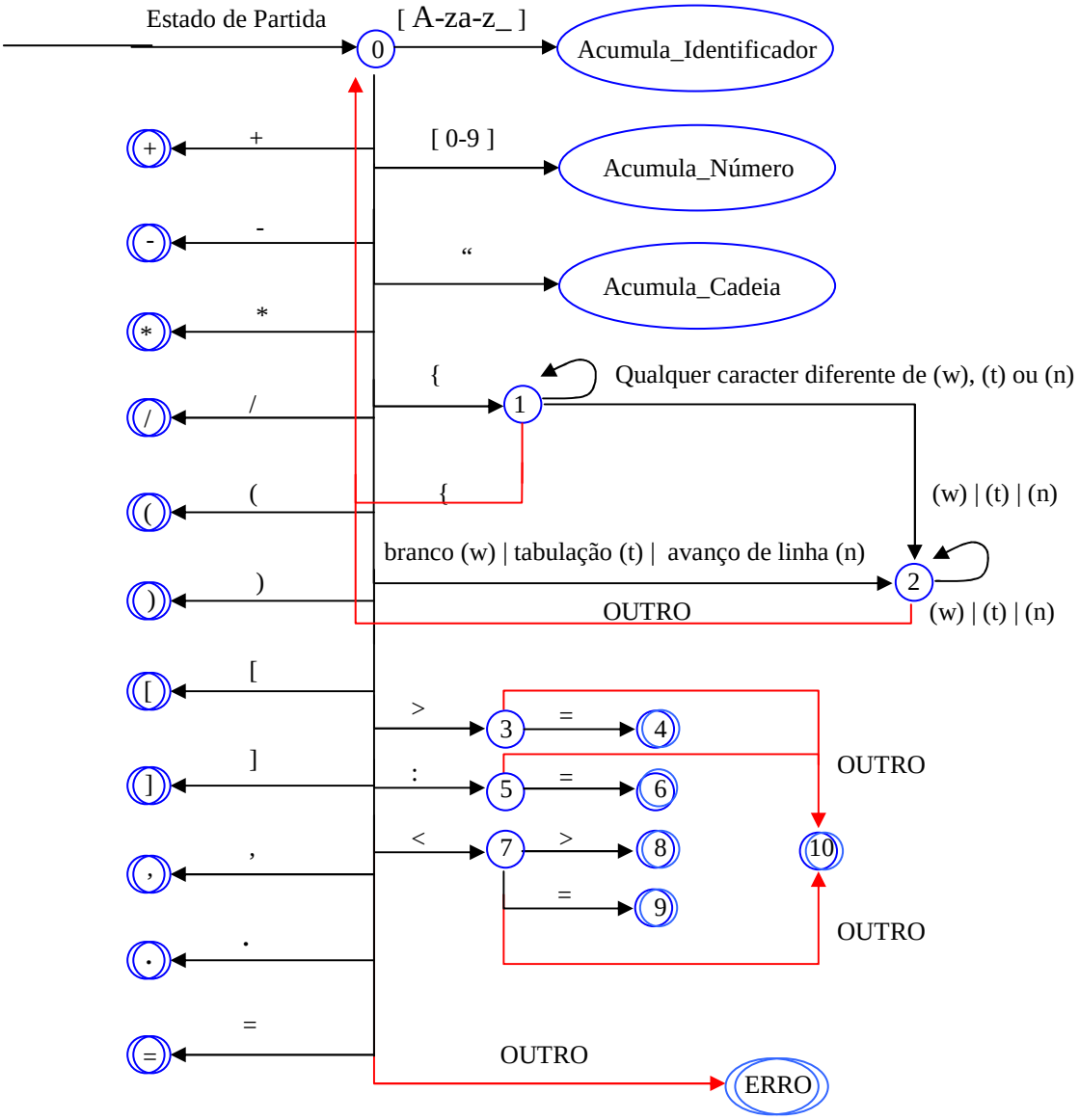


Figura 4.1.6.1. AFD para reconhecimento dos *tokens* pelo analisador léxico

No estado **0**, antes que feita a ação de ler um novo caractere do buffer de entrada, é verificado se o conteúdo da variável *lookahead* é um caractere *branco* ou não. Caracteres brancos são ignorados pelo analisador léxico. Se *lookahead* não contiver um caractere *branco*, o caractere armazenado é tomado por este estado como caracter corrente, e nenhum caractere é lido do buffer de entrada.

A transição dos estados **3, 5** e **7** para o estado reconhecedor **10,** assim como a transição do estado **1** para o estado **0**, faz com que o último caractere lido do buffer de entrada seja armazenado na variável *lookahead*.

O estado **1** é responsável por processar comentários, e o estado **2** é responsável por retirar do buffer de entrada caracteres como avanço de linha, tabulação e espaço em branco.

O estado rotulado **Erro** indica um estado onde um caractere não identificado foi encontrado no buffer de entrada.

O estado rotulado “.” executa um tratamento especial: ele verifica se a variável *prox_dado* armazena um “.” Nessa condição, o estado armazena em *lookahead* esse último caractere. O objetivo dessa estratégia será explicado adiante.

No início da leitura de dados do buffer de entrada *dado_anterior* armazena o caracter *branco*.

AFD`s detalhados

- AFD para o estado rotulado *Acumula_Identificador*

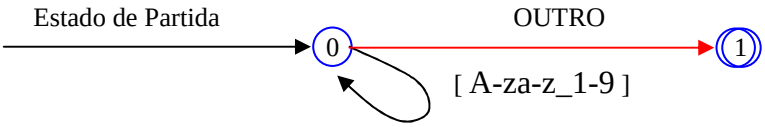


Figura 4.1.7.1. AFD detalhado para o padrão *identificador*

Nesse diagrama, o estado reconhecedor **1** identifica se o padrão reconhecido é ou não uma das palavras reservadas da linguagem. Em caso afirmativo o estado simplesmente identifica o *token* a ser retornado, enquanto que se não for uma palavra reservada o o estado executa uma operação mais complexa: retorna um *token* para **identificador**, e insere o padrão reconhecido em uma tabela de símbolos. Sua posição na tabela de símbolos é retornada como atributo para o *token*. Neste estado, o último caractere lido do buffer de entrada é armazenado na variável *lookahead*.

- AFD para o estado rotulado *Acumula_Cadeia*

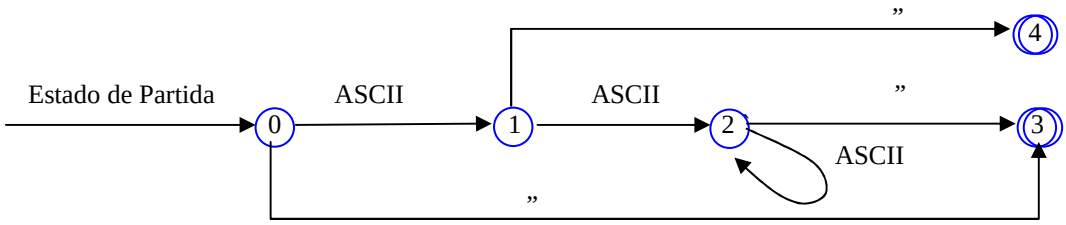


Figura 4.1.7.2. AFD detalhado para os padrões *cadeia* e *caracter*

Nesse diagrama os rótulos *ASCII* referem-se a qualquer caractere ASCII diferente do caractere de avanço de linha. Os estados **3** e **4** correspondem, respectivamente, aos estados de reconhecimento de cadeias de caracteres contendo um ou vários caracteres.

A transição do estado **1** para o estado **4** indica que foi reconhecida uma cadeia contendo um único caractere, e nesse caso um *token* indicando o reconhecimento de um caracter é retornado pelo estado reconhecedor. A esse *token* é passado como atributo um valor inteiro correspondente.

O reconhecimento de padrões pelo estado **3** faz com seja retornado um *token* indicando o reconhecimento de uma *cadeia de caracteres* (uma **string**). Nesse estado a cadeia reconhecida é armazenada em uma tabela de símbolos, e sua posição na mesma é retornada como atributo para o *token*.

- AFD para o estado rotulado *Acumula Número*

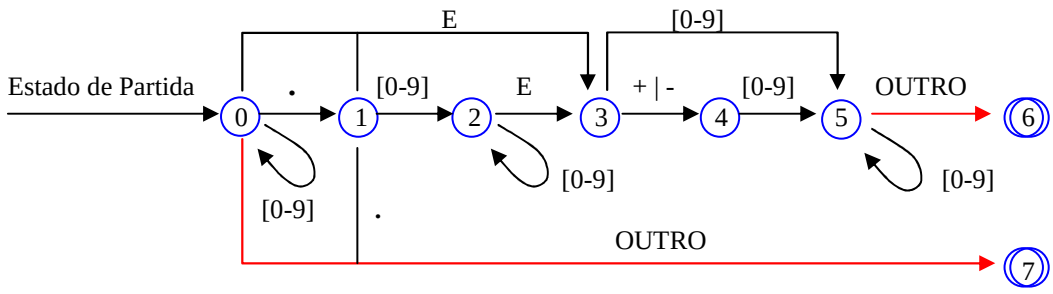


Figura 4.1.7.3. AFD detalhado para o padrão n*um*

Nesse diagrama, os estados **6** e **7** correspondem, respectivamente, aos estados de reconhecimento para números reais e números inteiros. A transição do estado 1 para o estado 7 foi necessária em virtude da existência, na declaração de vetores em Pascal, do formato **<constante inteira> . . <contante inteira>**. Aqui, tendo lido apenas o primeiro ".", o analisador léxico não é ainda capaz de decidir se o próximo *token* é um número real no formato **<digitos>.xxx** ou se vai encontrar **. .** , como exigido na declaração de vetores. Dessa forma, somente após ter identificado o próximo caractere seguindo o "." é que o analisador léxico é capaz de decidir qual o reconhecimento que deve ser efetuado.

A transição do estado **5** para o estado **6**, assim como a transição do estado **0** para o estado **7**, faz com que seja efetuada a ação de guardar em *lookahead* o último caracter lido.

A transição do estado **1** para o estado **7** faz com que seja guardado um "." em *lookahead* e também um outro "." em *prox_dado*. Essa última variável foi necessária para guardar a informação de que, nesse caso específico, foram avançados dois *tokens* pelo analisador léxico (um para cada ".").

O reconhecimento de padrões através do estado **6** faz com seja retornado um *token* indicando o reconhecimento de um *número real*. Nesse estado o número reconhecido é armazenado em uma tabela de números, e sua posição na tabela é retornada como atributo para o *token*.

O reconhecimento de padrões através do estado **7** faz com seja retornado um *token* indicando o reconhecimento de um *número inteiro*. Nesse estado o próprio número reconhecido é retornado como atributo para o *token*.

Restrições e observações a respeito do analisador léxico

- Um comentário pode conter qualquer número de caracteres entre chaves ({ }). O analisador léxico ignora todos os caracteres encontrados entre esses dois símbolos da linguagem.

- Uma cadeia de caracteres pode conter até 255 caracteres ASCII diferentes do caracter de retorno de linha. Se forem encontrados mais de 255 caracteres durante o reconhecimento de uma cadeia o analisador léxico toma a decisão de ignorar caracteres do arquivo fonte até que seja encontrado um caracter de fechamento de cadeia ("), ou até que seja atingido o final de arquivo. Em qualquer um desses dois casos um erro léxico é reportado, informando a linha do arquivo fonte onde foi encontrado o erro.

- Inteiros válidos não possuem nem fração nem expoente. O analisador léxico reconhece constantes inteiras válidas entre 0 e 32767 (em módulo), reportando um erro de overflow se forem encontrados no arquivo fonte valores inteiros fora desse intervalo.

- Números reais válidos são números compreendidos no intervalo de 3.4 (não incluindo 3.4) elevado a um expoente de -38 até 3.4 elevado a um expoente de +38.
  Os números reais reconhecidos pelo analisador léxico são truncados em 11 dígitos significativos, e armazenados na tabela de números segundo o formato

  Dígito . **Dígitos*** E (+ | -) ? **Dígitos***

  onde a sequência de caracteres definida no primeiro **Dígitos** contém até 10 caracteres, e a cadeia de caracteres identificada definida no segundo contém até 2 caracteres.

- Um identificador válido contém até 127 caracteres do conjunto alfanumérico ASCII adicionado do caracter _. Se forem encontrados mais de 127 caracteres durante o reconhecimento de um identificador o analisador léxico toma a decisão de ignorar caracteres do arquivo fonte até que seja encontrado um caractere que não se encontre nesse conjunto. Nesse caso um erro léxico é reportado, informando a linha do arquivo fonte onde foi encontrado o identificador inválido.

- Caracteres ASCII representando espaços em branco, tabulações e retorno de linha são ignorados pelo analisador léxico.

- A presença de caracteres ASCII que não fazem parte de nenhum padrão, operador ou símbolo da linguagem não são reconhecidos pelo analisador léxico, ocasionando um erro léxico.

4.2. Analisador Sintático

Construção do analisador sintático

Para a construção do analisador sintático foi utilizada uma ferramenta computacional – **FACC** (Ferramenta de Apoio a Construção do Compiladores).

A ferramenta, na verdade, é um gerador de analisadores LALR. Com ela é possível, a partir da especificação de uma gramática livre de contexto, produzir automaticamente um analisador sintático LALR e uma tabela sintática LALR.

A ferramenta define uma interface com o analisador léxico, produzindo ainda a especificação mínima de um analisador semântico, implementando um esquema eficiente de compilação em dois passos, baseado em gramáticas de atributos (uma gramática onde os tokens são passíveis de possuírem atributos associados).

Abaixo estão relacionadas algumas características do **FACC**.

1. Separa a *Análise Sintática* do *Processamento Semântico*;
2. Gera um *Analisador Sintático* dirigido por tabelas;
3. Não impõe restrições à gramática (recursão à esquerda, fatoração);
4. Usa uma *Representação Intermediária Sintética* (RIS) entre os passos da compilação. A RIS é uma estrutura de dados que corresponde à seqüência de reduções ocorrida durante o a construção da árvore gramatical, em ordem prefixada, e uma lista de atributos dos símbolos terminais. Com a RIS, o processador semântico pode decidir que rotinas processar e pode ainda recuperar o valor dos atributos a serem processados;
5. Gera um processador semântico com um procedimento para cada símbolo não terminal da gramática.

Ações sintáticas tomadas pela ferramenta de apoio

A ocorrência de ambigüidades na gramática que define uma linguagem implica na existência de mais de possiblidade de ação sintática durante o processamento de determinada regra sintática da linguagem.

O analisador sintático gerado pela ferramenta constrói a árvore gramatical das folhas para a raiz, onde as folhas são os símbolos terminais da gramática e os nós intermediários são os símbolos não terminais da gramática.

A ferramenta define duas operações básicas na montagem da árvore gramatical:

1. SHIFT (ação de empilhar): consume um *token* da entrada;
2. REDUCE (ação de reduzir): aplica uma das produções da gramática, reduzindo um conjunto de terminais a um não-terminal presente no lado esquerdo de alguma produção da gramática.

O tratamento de ambigüidades pela ferramenta considera dois casos:

a) CONFLITO DE SHIFT/REDUCE:
A ferramenta realiza a operação de SHIFT;

b) CONFLITO DE REDUCE/REDUCE:
A ferramenta realiza uma operação de REDUCE pela primeira produção envolvida no conflito;

No caso particular do tratamento da ambigüidade envolvendo o comando **if-then-else**, a ferramenta opta pela operação de SHIFT. O conjunto de produções abaixo:

```
Comando ->  'if' expressao 'then' comandos 'else' comandos
          | 'if' expressao 'then' comandos ;
```

por exemplo, gera um conflito SHIFT/REDUCE, que é tratado da seguinte forma:

- CONFLITO SHIFT/REDUCE P(2) e P(3) com o terminal 'else' SHIFT assumido !!!

Gramática Descritiva da Linguagem

O conjunto contendo as produções que compõem a gramática da linguagem para o compilador **Pascal ZIM!**,submetida à ferramenta FACC está no anexo II.

Interface com os analisadores léxico e semântico

A interface entre o analisador sintático e os analisadores léxico e semântico é feita através de um conjunto de variáveis e funções, enumeradas abaixo:

a) int **le_token()** : função que deve ser provida pelo analisador léxico, que retorna um inteiro correspondente ao *token* atual.

b) int **ATRIB1**, **ATRIB2** : variáveis definidas no analisador léxico que armazenam os valores dos atributos de um *token*. Um terminal possui atributos se na gramática descritiva da linguagem for definido como um terminal com atributos associados.
O analisador sintático gerado pela ferramenta define um atributo para cada terminal, denominada *atributo_terminal*. A construção eficiente da tabela de símbolos no analisador léxico, através do uso da função de *hash hashpjw*, forçou a necessidade de dois atributos para um terminal.

c) **topo_reducao_prefix** : variável referenciada e alterada pelo processador semântico, que guarda a base da pilha de reduções;

d) **fila_atributo** : variável referenciada e alterada pelo processador semântico, que guarda a base da fila de atributos;

4.3. Analisador Semântico

Definição da informação de escopo para identificadores

Nosso objetivo é identificar o escopo de cada *identificador* presente *seção de definição e declaração de dados* de um programa. Para tanto, faremos uso de um ponteiro auxiliar, denominado p_escopo, responsável por "guardar" na tabela de símbolos a posição contendo o escopo atual, além de uma pilha denominada *pilha de funções*, usada simular o aninhamento de subprogramas.

A informação de escopo para cada *identificador* (que corresponde a uma entrada na tabela de símbolos) será dada por um campo *escopo*, que é um ponteiro para uma entrada na tabela de símbolos.

A *estratégia* para alcançar o objetivo proposto é definida nos seguintes passos:

a) A partir dos atributos do token **program** (que identificam sua posição na tabela de símbolos), posicionamos p_escopo na posição da tabela de símbolos identificada com essa palavra reservada.
b) Fazemos com que o escopo do *identificador* para o programa (o nome do programa) aponte para a posição atual de p_escopo na tabela de símbolos (nesse caso, a posição de **program**).
c) Inserimos uma entrada na pilha de funções, contendo um apontador para a posição atual de p_escopo na tabela de símbolos (nesse caso, a posição de **program**).

Adotamos, então, as seguintes alternativas para as declarações:

d) Ao encontrar uma declaração de subprograma, fazemos com que o escopo para o *identificador* da declaração aponte para a posição atual de p_escopo na tabela de símbolos. Movemos então p_escopo para a posição na tabela de símbolos identificada por esse *identificador*. Finalmente, inserimos uma entrada na pilha de funções contendo um apontador para a nova posição atual de p_escopo na tabela de símbolos.

e) Ao sair de uma declaração de função removemos a entrada no topo da pilha de funções e reposicionamos p_escopo a partir do novo topo da pilha de funções.

f) Para cada *identificador* encontrado na *seção de definição e declaração de dados* do programa fazemos seu escopo apontar para a posição atual de p_escopo. Uma função *posiciona_escopo*, que recebe como parâmetros os atributos do identificador (que identificam sua posição na tabela de símbolos) e faz com que o escopo desse identificador aponte para a posição na tabela de símbolos atualmente identificada por p_escopo.

No caso dos registros a definição da informação de escopo é mais complexa, pois são permitidas declarações do tipo:

```
Type a,b,c = Record
              Campo:integer;
             End;
```

Podemos constatar, nesse exemplo que o campo *Campo* pertence ao escopo de a, b, c. Ao contrário de funções, um campo pode possuir vários escopos.

A solução adotada para esse problema é manter, além de uma pilha de funções, uma pilha de registros, com a diferença de que nessa nova pilha podem haver vários elementos no topo em vez de um único. Esses vários elementos são ligados através de uma lista.

A estratégia para definição da informação de escopo para campos é definida através dos seguintes passos:

a) Ao entrar em uma declaração de registro, empilhamos o primeiro elemento da lista de identificadores nessa declaração, e inserimos os outros demais elementos em uma lista ligada a esse elemento. Fazemos p_escopo apontar para a posição na tabela de símbolos apontada pelo primeiro elemento dessa lista.

b) Ao sair de uma declaração de registro removemos as entradas no topo da pilha de registros e reposicionamos p_escopo na posição apontada pelo primeiro elemento da lista no novo topo da pilha de registros. Se a pilha de registros estiver vazia, posicionamos p_escopo na posição apontada pelo elemento localizado no topo da pilha de funções.

c) Para inserir a informação de escopo em campos executamos os seguintes passos:

i. Fazemos com que o escopo do campo aponte para a posição identificada por p_escopo.
ii. Se existir um elemento seguinte na lista do topo da pilha de registros movemos p_escopo para a posição apontada pelo próximo elemento na lista do topo da pilha de registros. Criamos uma nova entrada para o *identificador* na tabela de símbolos, fazendo com que o escopo dessa entrada aponte para a posição atual de p_escopo.
iii. Repetimos o passo (b) até que tenhamos percorrido toda a lista de registros.
iv. Reposicionamos p_escopo na posição apontada pelo primeiro elemento da lista do topo da pilha de registros.

Definição da informação de tipo para identificadores

O tipo de um *identificador* pode ser especificado por meio de uma, entre as seguintes formas:

a) Em uma declaração do tipo <lista de identificadores> : *tipo***;**
b) Em funções, onde se tem usualmente **function** *identificador* **(** *argumentos* **) :** *tipo***;**
c) Em definições de constantes
d) Em definições de vetores e registros

onde *tipo* , no caso (a) corresponde à produção

Tipo → **real** | **char** | **integer** | **boolean** | **string** | **id**

e *tipo* , no caso (b) corresponde à produção

Tipo → **real** | **char** | **integer** | **boolean** | **string**

A solução para o caso (a) é alcançada através da seguinte estratégia:

1. Guardamos cada *identificador* referenciado em *lista_identificadores* em uma lista de identificadores.
2. No tratamento dado à produção *Tipo* → .... fazemos o seguinte:

    2.1. Procuramos na tabela de símbolos a entrada que armazena o símbolo do lado direito da produção para *Tipo* (**real**, **char**, **integer**, **boolean**, **string** ou **id**), guardando essa posição em um ponteiro *p_tipo.*
    2.2. "Esvaziamos" a lista de identificadores, inserindo como tipo, para cada identificador, um ponteiro para *p_tipo*.

A solucão para o caso (b) é alcançada através da seguinte estratégia:

1. Guardamos a posição do *identificador* da função na tabela de símbolos em um ponteiro marcador, *p_marcador*.
2. Antes da chamada à produção *tipo*, inserimos o *identificador* apontado por *p_marcador* na lista de identificadores.
3. No tratamento dado à produção *Tipo* → .... fazemos o seguinte:

    3.1. Procuramos na tabela de símbolos a entrada que armazena o símbolo do lado direito da produção, guardando essa posição em um ponteiro *p_tipo.*.
    3.2. Removemos o *identificador* da lista de identificadores, inserindo como tipo, um ponteiro para *p_tipo*.

A solução para o caso (c), dado pela regra sintática

<lista de identificadores> : *constante***;**

onde *constante* pode ser uma constante numérica, uma cadeia de caracteres, um caractere ou um valor booleano, é similar à solução para o caso (a), com a diferença de que no passo 2.1 p_tipo é identificado a partir do tipo de constante em análise.

É interessante notar que as soluções para os casos (a), (b) e (c) efetivamente associam tipos aos *identificadores* durante a análise da produção *Tipo* → .... . Dessa forma, a definição de tipos se concentra-se principalmente nessa produção.

O caso mais complexo para definição da informação de tipos, dado pelo caso (d), requer um campo adicional em cada entrada da tabela de símbolos, denominado *objeto*. Esse caso é solucionado através da seguinte estratégia:

1. Nas produções < *lista de identificadores* > = **array [** *limite do vetor* **] of** .... procedemos da seguinte forma:

    1. Guardamos cada *identificador* referenciado em *lista_identificadores* em uma lista de identificadores.
    2. Procuramos na tabela de símbolos a entrada que armazena a palavra reservada **array**, guardando essa posição em um ponteiro *p_tipo.*.
    3. Percorremos toda a lista de identificadores, guardando, para cada elemento da lista, as seguintes informações:

- No campo *objeto* guardamos a informação "A", indicando que o *identificador* é um vetor.
- Inserimos como tipo, para o *identificador*, um ponteiro para *p_tipo*.

2. Nas produções < *lista de identificadores* > = **record** *campos* **end ;** , para cada *identificador*, procedemos da seguinte forma:

   1. Guardamos cada *identificador* referenciado em *lista_identificadores* em uma lista de identificadores.
   2. Percorremos toda a lista de identificadores, guardando, para cada elemento da lista, no campo *objeto*, a informação "R", indicando que o *identificador* é um registro.

Os passos (1) e (2), em conjunto com a estratégia (a), produzá, para cada uma das declarações abaixo, as seguintes definições para o tipos do *identificador* **a**:

Var A**: Array [** 1**..** 3**] of integer;**

Tipo de A = *integer*
Definição de objeto para A = 'A'

Var A**: Array [** 1**..** 3**] of record** .... **;**

Tipo de A = *array*
Definição de objeto para A = 'R'

Var A**: record** .... **;**

Tipo de A = nenhum
Definição de objeto para A = 'R'

Esquema básico para a verificação de escopo

A verificação das *regras de escopo* é feita segundo a *regra do aninhamento mais interno*, fazendo uso de uma pilha. As ações executadas durante essa verificação são:

1. Ao "entrarmos" em um subprograma empilhamos uma referência para este;
2. Ao "sairmos" de um subprograma desempilhamos o subprograma no topo da pilha.
   A propriedade de aninhamento de subprogramas garante que o subprograma referenciado no topo da pilha trata-se do subprograma sendo analisado.
3. Ao procurarmos pelo escopo de um nome *x* percorremos a pilha de subprogramas do topo até a base, em busca de um subprograma onde *x* tenha sido declarado.

As regras acima não valem para verificação de escopo para campos. Para este, adota-se uma outra abordagem:
A referência a um campo é feita segundo a partir das seguintes regras sintáticas:

```
1. variavel -> id idtail ;

2. idtail -> . id idtail | [ lista_expressoes ] idtail | #  ;
```

Para as produções

`variavel -> id idtail e idtail -> . id idtail`,

adotamos a seguinte estratégia:

1. Verificamos se **id** é um vetor , e se for, guardamos em uma variável *espera_vetor* o valor 1.

2. Verificamos se **id** é um registro, e se for, guardamos em uma variável *espera_registro* o valor 1. Verificamos se o tipo de **id** é um registro, e se for, guardamos a entrada na tabela de símbolos correspondente ao tipo de **id** em um ponteiro auxiliar, p2. Em caso contrário, guardamos a entrada a entrada corresponde a **id** em p2.
3. Na segunda produção, verificamos se o valor armazenado na variável *espera_vetor* é igual a zero e se o valor armazenado na variável *espera_registro* é igual a um. Caso essa condição não seja satisfeita, um erro deve ser reportado – é esperada a referência a um vetor, ou um campo não é esperado.

Na produção

```
idtail -> . id idtail
```

Verificamos o escopo de **id** da seguinte forma: verificamos se o tipo de **id** corresponde à entrada na tabela de símbolos guardada pelo ponteiro p2. Se não for, um erro deve ser reportado – o campo **id** não pertence ao registro sendo analisado.

Na produção

```
idtail -> [ lista_expressoes ]
```

Verificamos se o valor armazenado na variável *espera_vetor* é diferente de zero, caso contrário um erro é reportado – uma referência a vetor está sendo feita em uma variável que não é um vetor.

Antes da avaliação de *lista_expressões* guardamos os valores armazenados nas variáveis *espera_vetor, espera_registro* e p2, pois essas variáveis podem ser modificadas durante a avaliação de *lista_expressões.*

Após a avaliação de *lista_expressões*, recuperamos os valores de *espera_registro*, *espera_vetor* e p2.

Se, durante a avaliação da produção

```
idtail -> #
```

Alguma das variáveis *espera_vetor* ou *espera_registro* contiver o valor 1, então um erro semântico é reportado – é esperada a referência aos índices de um *vetor* ou a um *campo*.

Esquema básico para a verificação de tipos

Ao invés de descrever minuciosamente de que modo foi implemenatada a verificação de tipos no **Pascal ZIM!**, iremos detalhar o esquema de verificação utilizado pelo compilador, fazendo uso das notações desenvolvidas na seção 2.2.3.

As regras para verificação de tipos em expressões são enumeradas abaixo:

1. Usamos uma função *procurar_tipo* (*e*) para recuperar o tipo guardado na entrada da tabela de símbolos denotada por *e*. Quando um identificador aparece numa expressão, seu tipo é recuperado e atribuído ao atributo *tipo* da expressão.

   $E \rightarrow$ **id** { *E.tipo* := *procurar* (**id**.*entrada*) }

2. A expressão formada pela aplicação do operador *mod* a duas subexpressões de tipo *inteiro* possui tipo *inteiro*; caso contrário, seu tipo é denotado por um tipo particular, um *tipo_erro*. A regra é

   $E \rightarrow E_1$ **mod** E_2 { *E.tipo* := **se** E_1*.tipo = inteiro* **e** E_2*.tipo = inteiro* **então** *inteiro* **senão** *tipo_erro* }

3. As regras semânticas para outros operadores da linguagem (operadores aritméticos, lógicos e condicionais, definidos na seção 3.8) segue raciocínio semelhante.Numa referência a um vetor, definido por *array* E_1 [E_2], a expressão de índice E_2 precisa ser do tipo inteiro, caso em que o resultado é o elemento de tipo *t* obtido a partir do tipo *array* (*s*, *t*) de E_1.

   $E \rightarrow E_1$ [E_2] { *E.tipo* := **se** E_2*.tipo* = *inteiro* **e** E_1*.tipo* = *array* (*s*,*t*) **então** *t* **senão** *tipo_erro* }

As regras para verificação de tiposem comandos são enumeradas abaixo:

1. Para comandos de atribuição:

S → **id** := *E* { **Se id**.*tipo* ≠ *E.tipo* **então** ((**se id**.tipo ≠ **string e** *E*.tipo ≠ **char**) **e** (**se id**.tipo ≠ **real e** *E*.tipo ≠ **integer**)então *acusa_erro_semtantico* }

2. Para comandos de repetição (por exemplo, o comando **while**):

S → **while** *E* **then** S_1 { **Se** *E.tipo* ≠ *booleano* **então** *acusa_erro_semtantico* }

A regra para o comando **repeat** segue raciocínio semelhante.

3. Para comandos condicionais (por exemplo, o comando **if**):

S → **if** *E* **then** S_1 { **Se** *E.tipo* ≠ *booleano* **então** *acusa_erro_semtantico* }

A regra para o comando **if** com **else** segue raciocínio semelhante.

4. Para comandos de tratamento de arquivos (por exemplo, o comando **close**):

S → **close (id)** { **Se** *procurar* (**id**.*entrada*) ≠ TEXT **então** *acusa_erro_semtantico* }

5. Para subprogramas:

$E \rightarrow$ **id** (E_2) { *E.tipo* := **se** E_2*.tipo* = *s* **e id**.*tipo* = $s \rightarrow t$ **então** t **senão** *tipo_erro* }

Esta regra diz que numa expressão formada pela aplicação de E_2 a **id**, o tipo de **id** precisa ser uma função $s \rightarrow t$, onde o tipo de E_2 (domínio da função) é *s*, e o contradomínio da função é *t*.

Outras verificações semânticas inclusas na verificação de tipos são:

- Verificar se o número de índices especificado numa referência a um vetor está de acordo com a definição do vetor;
- Verificar se o número de parâmetros passados a um subprograma está de acordo com a definição do subprograma.

4.4. Gerador de Código

Tamanhos e Tipos de Dados

Estabelecemos as seguintes convenções para os tipos de dados referenciados pelo compilador:

- Os inteiros serão representados por uma posição de memória na pilha de execução;
- Os caracteres serão representados por uma posição de memória na pilha de execução;
- Os reais serão representados por duas posições de memória na pilha de execução;
- As cadeias de caracteres declaradas sem restrição de tamanho serão representadas por 255 posições de memória na pilha de execução;
- As cadeias de caracteres declaradas com restrições de tamanho serão representadas por *n* posições de memória na pilha de execução, onde *n* é o (tamanho da cadeia de caracteres;

Cálculo do tamanho e localização em memória para variáveis e campos

Para possibilitar o acesso à variáveis é necessário ter as informações referentes à localização em memória (deslocamento em relação ao endereço do registrador de base *b* para um registro de ativação) e à quantidade de memória ocupada (deslocamento individual) para cada uma das variáveis, constantes e campos de registros declarados no programa fonte.

Para a localização em memória das variáveis e constantes, o deslocamento é calculado como a posição de memória no vetor de execução relativa ao endereço de base do registro de ativação no qual a variável (ou constante) foi definida, lembrando que:

- O endereço relativo zero no registro de ativação de um subprograma corresponde ao *elo de acesso*,
- O endereço relativo 1 no registro de ativação de um subprograma corresponde ao *elo de controle*,
- O endereço relativo 2 no registro de ativação de um subprograma corresponde ao *endereço de retorno* para o subprograma
- O endereço relativo 3 no registro de ativação de um subprograma corresponde à primeira posição de memória usada para endereçamento das variáveis locais ao subprograma.

Por exemplo, na seguinte declaração:

```
Var Var1: char;
    Var2: array [1.. 3] of integer;
    Var3: boolean;
```

devem ser armazenadas, para cada uma das variáveis declaradas, as seguintes informações:

- Variável: Var1
  Quantidade de memória ocupada pela variável: 1
  Localização em memória da variável (relativa ao endereço de base do subprograma): 3

- Variável: Var2
  Quantidade de memória ocupada pela variável: 3
  Localização em memória da variável (relativa ao endereço de base do subprograma): 4

- Variável: Var3:
  Quantidade de memória ocupada pela variável: 1
  Localização em memória da variável (relativa ao endereço de base do subprograma): 7

As operações de cálculo e armazenamento das informações referentes à quantidade de memória ocupada e à localização em memória das variáveis (ou constantes) podem ser feitas durante a análise semântica, paralelamente à operação de reconhecimento de tipos.

No caso dos campos de registros, a posição em memória é calculada como a posição de memória, no vetor de execução, relativa ao início do bloco de registro no qual o campo foi definido.

Por exemplo, na seguinte declaração:

```
Var Registro: record
                Campo1: integer;
                Registro_aninhado: record
                      Campo2: boolean;
                      Campo3: boolean;
                end;
            end;
```

teremos, para cada um dos campos do registro R*egistro*:

- Campo: Campo1
  Quantidade de memória ocupada pelo campo: 1
  Localização em memória do campo (relativa ao início do registro no qual foi definido): 0

- Campo: Campo2
  Quantidade de memória ocupada pelo campo: 1
  Localização em memória do campo (relativa ao início do registro no qual foi definido): 0

- Campo: Campo3
  Quantidade de memória ocupada pelo campo: 1
  Localização em memória do campo (relativa ao início do registro no qual foi definido): 1

- Campo: Registro_aninhado
  Quantidade de memória ocupada pelo campo: 2
  Localização em memória do campo (relativa ao início do registro no qual foi definido): 1

As operações de cálculo e armazenamento das informações referentes à quantidade de memória ocupada e à posição em memória de campos, definidos com algum tipo, podem ser feitas durante a análise semântica, paralelamente à operação de reconhecimento de tipos. No caso de registros aninhados, essas informações podem ser armazenadas quando o registro em análise estiver sendo retirado do topo da pilha de registros.

Para os tipos definidos no programa fonte deve-se guardar apenas a quantidade de memória necessária para armazenar, no vetor de execução, uma variável definida com esse tipo. Essa informação será associada à quantidade de memória ocupada para uma variável ou campo que venha a ser declarado com esse tipo no programa fonte.

A geração de instruções para acesso de variáveis

Uma vez armazenadas as informações referentes à posição em memória para cada variável declarada no programa fonte, além da identificação do registro de ativação onde se encontra a declaração da variável (informação de escopo) é possível gerar as instruções VAR e VARPARAM.

O cálculo do registro de ativação no qual se encontra declarada uma variável pode ser feito durante a análise semântica (mais especificamente, durante a análise de escopo), com a seguinte estratégia:

1. No início da análise de escopo devemos analisar a presença (ou ausência) da variável na função do topo da pilha de funções. Uma variável *nivel*, que guardará a informação de escopo, é inicializada com o valor zero.

2. Se a variável não se encontra declarada no escopo atual, realizamos a busca no escopo imediatamente inferior. Isso corresponde à avançar uma posição abaixo na pilha de funções. Ao ser feita essa ação, o valor armazenado na variável *nivel* é incrementado.
3. Repetimos o passo 2 até que seja encontrada uma função na qual a variável tenha sido declarada. No caso de não ser mais possível avançar na pilha de funções, para buscar uma função onde a variável foi declarada, estamos em um caso particular de erro semântico, e nesse caso a infomação buscada não pode ser definida.

A instrução INDEX pode ser generalizada para tratar vetores com mais de uma dimensão, utilizando os princípios da linearização de matrizes. A idéia para essa generalização utiliza o fato de que um vetor n-dimensional $m_1 \times m_2 \times ... \times m_n$ é armazenado na memória em células contíguas, que podem ser endereçadas a partir de um endereço de base indicando o início do vetor.

O cálculo de uma posição fixa dentro de um vetor *n*-dimensional pode ser deduzido através de *n*.

- Quando n = 1, a posição de memória em que está localizada a célula m_j é a célula identificada na posição j.

- Quando n = 2, a posição de memória em que está localizada a célula m_{ij} em um vetor de tamanho m_1 x m_2 é a célula identificada na posição (m_2 x i) + j;
- Quando n = 3, a posição de memória em que está localizada a célula m_{ijk} em um vetor de tamanho m_1 x m_2 x m_3 é a célula identificada na posição (m_3 x i) (m_2 x j) + k;
- Quando n > 3, a posição de memória em que está localizada a célula $m_{z...kij}$ em um vetor de tamanho m_1 x m_2 x m_3 x ... x m_n é a célula identificada na posição (m_1 x z)... (m_{n-1} x k)(m_n x i) + j;.

A idéia para tratar a referências a vetores n-dimensionais é generalizar a instrução INDEX, da seguinte maneira:

- Para cada referência a uma das *n* dimensões do vetor em questão é gerada uma instrução INDEX, que mantém na parte temporária do vetor de execução o valor obtido com a avaliação da expressão da dimensão em estudo, e insere também o tamanho da dimensão (dado pela diferença entre a posição final e a posição inicial da dimensão + 1). (Figura 4.4.1)

```
void INDEX(int lower, int upper) {
  int i;
  i = vetor_execucao[s];
  if ((i<lower) || (i>upper))
   {
    printf(">>> Erro nos limites do vetor!");
    running = 0;
   }
  else
   vetor_execucao[s] = i - lower;

   // Guardamos o tamanho da dimensao atual
   s++;
   vetor_execucao[s] = upper-lower+1;

  p = p+3;
 }
```

Figura 4.4.1. O algoritmo que implementa a instrução INDEX

A referência a um vetor *n*-dimensional gera, assim, *n* instruções INDEX. Assim, ao final da análise de cada uma dessas *n* dimensões estarão alocadas, na parte temporária do vetor de execução, 2*n* posições de memória, guardando as informações de índice e tamanho de cada dimensão.

A partir desse conjunto de informações é possível calcular a posição dentro do vetor no qual se encontra armazenada uma célula em particular. Com esse propósito é gerada uma nova instrução, CALCPOSVET, que recebe como parâmetros o número de dimensões *n* do vetor em questão e seu tamanho (em palavras de memória), e, a partir de um endereço de memória armazenado na parte temporária do vetor de execução, calcula o endereço de memória em que se encontra armazenada uma célula de um vetor. (Figura 4.4.2)

```
void CALCPOSVET( int n , int tamanho_vetor)
 {
  int posicao_vetor;
  int multiplicando_matriz;
  int i;

  posicao_vetor = vetor_execucao[s- 1];
  if ( n > 1 )
   {
    multiplicando_matriz = vetor_execucao[s];
```

```
    for (i = 1; i<n; i++)
     {
      posicao_vetor = posicao_vetor +
      ( multiplicando_matriz * (vetor_execucao[s-((2*i)+1)]));
      multiplicando_matriz= multiplicando_matriz*vetor_execucao[s-(2*i)];
     }
   }
  s = s - 2*n;
  vetor_execucao[s]= vetor_execucao[s] + posicao_vetor*tamanho_vetor;
  p = p + 3;
 }
```

Figura 4.4.2. O algoritmo que implementa a instrução CALCPOSVET

A instrução FIELD é simples. Seu único parâmetro é simplesmente o deslocamento do campo relativo ao registro no qual foi declarado, tendo sido este calculado previamente.

A fim de generalizar a operação de acesso a variáveis, podemos verificar que uma posição de memória pode ser encontrada através da fórmula:

Endereço da Variável Composta = Endereço da Variável + Deslocamento do Seletor

Onde *Deslocamento do Seletor* é dada pela soma do deslocamentos de zero ou mais seletores.
Essa propriedade torna simples o endereçamento de variáveis complexas, como:

A [1, 2, 3]. Campo1. Campo2 [4, 5]. Campo3

Instruções usadas na avaliação de expressões

Quatro aspectos devem ser considerados na implementação de instruções usadas na avaliação de instruções, durante a geração de código intermediário: o mapeamento de constantes inteiras e reais, o *overflow* (estouro de memória), a sobrecarga de operadores e a conversão implícita de tipos de dados para operações. Estudaremos cada caso separadamente.

O mapeamento de constantes inteiras e reais é importante em virtude do fato de que todos os valores armazenados no vetor de execução são constantes inteiras. O gerador de código deve prover, assim, duas funções de mapeamento: uma função x que "distribui" os bits de uma constante real em n constantes inteiras e sua inversa x^{-1} que, tendo como argumento essas mesmas n constantes inteiras, seja capaz de recuperar uma constante real.

No caso específico do compilador **Pascal ZIM!** o código para o compilador foi escrito na linguagem C. Ali, Nesta linguagem, a representação de uma constante inteira e real em memória ocupam, respectivamente, 16 e 32 bits. Uma constante real contém o dobro de bits de uma constante inteira. Portanto, uma constante real pode ser mapeada em dois inteiros, o primeiro deles contendo os 16 bits mais à direita da constante, e o segundo contendo os 16 bits restantes.

A estrutura **union**, usada para compartilhar uma mesma área de memória entre vários tipos de dados diferentes mostrou-se interessante para a solução desse problema. Através de uma área de memória compartilhada contendo um número real, e um vetor de inteiros contendo duas posições, pode-se obter, de forma simples, a distribuição de bits desejada. A atribuição de uma constante real à variável real declarada dentro da área compartilhada automaticamente faz com que os 32 bits da constante real sejam distribuídos nos 32 bits do vetor de inteiros, como desejado.

A implementação da função que faz o mapeamento real → inteiro x inteiro é mostrada abaixo:

```
void converte_real_para_inteiros (float numero_real) {
union {         short int inteiro[2];         float real;
} u;  u.real = numero_real;  inteiro1 = u.inteiro[0];
inteiro2 = u.inteiro[1];}
```

O mapeamento inteiro x inteiro → real segue raciocínio semelhante, e é mostrado abaixo:

```
float converte_inteiros_para_real(int int1, int int2) {
union {         short int inteiro[2];         float real;
} u;  u.inteiro[0]= inteiro1;  u.inteiro[1]= inteiro2;
  return u.real;
 }
```

O tratamento de overflow é feito para constantes inteiras, reais e também para cadeias de caracteres. A idéia é bem simples. Basta verificar se, durante uma determinada operação a constante obtida ultrapassa o valor máximo permitido para constantes do tipo em questão.

O intervalo de valores válidos para constantes numéricas é:

- Para constantes reais: 3.4 * (10**-38) à 3.4 * (10**+38)
- Para constantes inteiras: 32767 à -32767.

O tratamento de *overflow* dado às cadeias de caracteres depende do tamanho da cadeia, definido quando uma variável é declarada. Variáveis declaradas como o tipo **string** podem ocupar até 255 posições de memória, enquanto variáveis declaradas com o tipo **string [** *n* **]** podem ocupar até *n* posições de memória.

A sobrecarga de operadores permite que operações diferentes para um mesmo operador sejam realizadas de acordo com o tipo dos operandos. As operações de soma, subtração, multiplicação e divisão são sobrecarregadas, possibilitando o cálculo para operandos inteiros ou reais. A soma de duas constantes inteiras produz uma instrução ADD, que efetua a soma de duas constantes inteiras, enquanto a soma de duas constantes reais produz uma instrução ADDREAL, que efetua a soma de duas constantes reais.

As expressões envolvendo operandos do tipo real requerem o uso das funções de mapeamento entre reais e inteiros x e x^{-1}. Antes de ser realizada a operação adequada, os operandos reais são recuperados através do uso da função x, que faz um mapeamento *inteiro* x *inteiro* → *real*. O resultado obtido com a operação, se for também uma constante real, é "quebrado" em inteiros através da função x^{-1}, que faz o mapeamento *real* → *inteiro* x *inteiro*, e esses são armazenados no vetor de execução.

As operações de soma e comparação (=, <>, <, >, <=, >=) são sobrecarregadas para inteiros, reais, caracteres e cadeias de caracteres. A operação de '+' para duas cadeias de caracteres concatena a segunda cadeia com a primeira. As operações de comparação são indutivas e óbvias para todos os quatro tipos de dados. As operações de comparação '=' e '<>' são suportadas também para booleanos.

A conversão implícita de tipos é feita para possibilitar que dois operandos de tipos diferentes possam ser utilizados em uma dada operação. Uma soma envolvendo um operando inteiro e um operando real, por exemplo, gera uma instrução para soma de dois reais. Antes que a soma seja efetuada o primeiro operando é convertido para real, e o resultado da operação é do tipo real.

A conversão implícita de tipos é feita de acordo com duas regras. Primeiro, se um dos operandos de uma operação envolve um inteiro e um real, o inteiro é convertido para real. Segundo, se um dos operandos de uma operação envolve um caracter *c* e uma cadeia de caracteres o caracter *c* é reconhecido como uma cadeia de caracteres.

Instruções usadas em comandos

Os comandos de atribuição envolvem dois operandos: um valor (que pode ser uma constante numérica, uma constante booleana, um caracter ou uma cadeia de caracteres) e um endereço em memória que armazenará o primeiro operando.

O caso envolvendo atribuição de constantes inteiras, booleanas, caracteres e reais é dado pela instrução ASSIGN. O caso mais complexo, envolvendo atribuição de cadeias de caracteres, deve possuir um parâmetro para verificar se o tamanho da cadeia sendo atribuída é maior que o espaço em memória disponível para armazenamento da cadeia pela variável. Assim, o tamanho máximo para a cadeia esperada deve ser passado como parâmetro para a atribuição de cadeias.

Para otimização de memória, o final de uma cadeia de caracteres, quando representada na área de memória temporária do vetor de execução, é “marcada” por uma constante de final de cadeia (-1). Dessa forma, uma cadeia de caracteres não ocupa, necessariamente, 255 posições de memória na área de trabalho, o que permite “economizar” células de memória.

Da mesma forma, para uma variável do tipo cadeia de caracteres, se o tamanho da cadeia armazenada for menor que a quantidade de memória disponível para seu armazenamento é inserido, ao final da cadeia, uma constante de final de cadeia (-1). Esse artifício permite que a recuperação de uma cadeia de caracteres seja feita de forma otimizada, pois apenas as constantes pertencentes à cadeia são recuperadas.

Essas otimizações levaram à definição de uma instrução própria para atribuição de cadeias, uma instrução ASSIGNSRT, que recebe como argumento o tamanho máximo esperado para a cadeia a ser atribuída. A instrução é responsável por armazenar, em um endereço de memória, uma cadeia de caracteres (presente na área temporária do vetor de execução) possuindo ou esse tamanho ou ainda um tamanho menor, nesse caso univocamente identificada através de uma constante marcando o seu fim (-1). (Figura 4.4.3)

```
void ASSIGNSTR( int length )
  {
   int x,y,i;
   int contador;
   int posicao;

   posicao = s;
   contador = 0;
    while (vetor_execucao[posicao] != -1)
      {
       contador++;
       posicao--;
      }
   if (contador > length)
     {
      cprintf("Erro de overflow!");
      running = 0;
     }
  else
    {
     s = posicao - 1;
     x = vetor_execucao[s];
     y = s + 2;
     i = 0;
     while (i < contador)
       {
        vetor_execucao[x+i] = vetor_execucao [y+i];
        i++;
       }
    }
  if (running)
    if (i < length )
      {
       vetor_execucao[x+i] = -1;
      }
 p = p + 3;
}
```

Figura 4.4.3. O algoritmo que implementa a instrução ASSIGNSTR

Na linguagem Pascal, além do comando **while**, existem mais dois comandos de repetição: o **Repeat** e o **For**.

Um comando **Repeat**, como em

Repeat S **Until** B

produz uma sequência de instruções da forma:

```
L1:  S
     B
DO ( L1 )
```

Se avaliação da expressão booleana B produz um valor *falso*, então a execução das instruções pertencentes ao corpo da repetição é repetida através de um desvio para a instrução armazenada no endereço L1. Caso contrário, a instrução seguinte ao **DO** será executada.

O conjunto de instruções geradas por um comando de repetição **for**, como em

For variável contadora **:=** expressão **to** expressão **do**
S**;**

produz a seguinte sequência de instruções:

```
        variavel( )
        expressao ( )
        ASSIGN ( 1 )
L1:     expressao( )
        VAR (...)
        VAL (...)
        BIGEQUAL
        DO ( L2 )
        Comandos
        VAR (...)
        VAR (...)
        VAL ( 1 )
        CONST( 1 )
        ADD
        ASSIGN( 1 )
        GOTO ( L1 )
L2:
```

A idéia para este comando segue os seguintes passos:

- Inicialmente, um valor inteiro é atribuído à variável de controle do comando.
- É definido um valor inteiro máximo para o qual a repetição deverá ser interrompida, no caso em que o valor armazenado na variável de controle ultrapassar esse valor inteiro máximo.
- Se o valor armazenado na variável de controle ultrapassar o valor máximo fixado, então a execução vai para a próxima instrução após a repetição (rotulada por L2)
- Caso contrário, os comandos da repetição são executados em sequência, o valor armazenado na variável contadora é incrementado de um, e o programa pula para a instrução rotulada por L1.

Um comando condicional **if** com **else** como em

If B **then** S_1 **else** S_2

é definido através da seguinte sequência de instruções:

Expressão ( )
DO (L1 **)**
Comandos do IF
GOTO (L2 **)**
(L1): Comandos do ELSE
(L2):

A avaliação da expressão booleana para o **if** produz um valor booleano *verdadeiro* ou *falso*. A instrução **DO** avalia esse valor, e prossegue para a execução da sequência de comandos inerentes ao **if** se esse valor for *verdadeiro*. Em caso contrário, a instrução **DO** desvia a execução de instruções para a sequência de instruções inerentes ao **else**. No primeiro caso, após as instruções do **if** terem sido executadas, a execução do programa é desviada para a próxima instrução imediatamente após o comando de seleção.

Os **comandos de entrada/ saída** podem fazer uso de arquivos.

A diferença essencial entre os comandos de entrada/ saída em arquivo e os comandos de entrada/ saída padrão são que os primeiros devem recuperar a informação sobre qual é o arquivo a ser utilizado.

O nome de um arquivo é guardado no endereço de uma variável do tipo TEXT através do comando Pascal **assign**, como em

Assign (arq**,** nome do arquivo **)**

Esse comando, na geração de código, produz duas instruções:

- Uma instrução VAR, que identifica o endereço da variável do tipo TEXT.
- Uma instrução ASSIGNSTR, que armazena no endereço da variável a cadeia de caracteres que identifica o nome do arquivo relacionado com a variável TEXT.

A leitura/escrita em arquivo são feitas da forma descrita em seguida:

Durante a análise semântica, ao ser reconhecida a solicitação para escrita em arquivo, como em

Write (arq, expressão **)**

onde o primeiro argumento para um comando **write** é uma variável do tipo text, é gerada uma instrução VAR, que informa o endereço da variável que guarda o nome do arquivo a ser utilizado na escrita de dados. Além disso, à uma variável global *escrever_no_arquivo* é atribuído o valor 1 para indicar que a escrita de dados foi direcionada para um arquivo. Para cada expressão a ser impressa em arquivo, então, é gerada uma instrução WRITEARQ, que recebe como argumento um inteiro identificando o tipo para a expressão a ser impressa.

A instrução WRITEARQ executa basicamente três ações durante sua execução, que são:

- Recupera a constante a ser impressa, constante esta armazenada na parte temporária do vetor de execução
- Recupera o nome do arquivo de impressão a partir do endereço de memória também armazenado na parte temporária do vetor de execução.
- Escreve a constante no arquivo

Após a escrita de todas as expressões, a memória utilizada para o armazenamento do endereço da variável contendo o nome do arquivo deve ser liberada. Isso é feito através de uma instrução LIBERAESPAÇOVETOR, que simplesmente decrementa o registrador *s* de uma posição.

A idéia utilizada para implementação do comando para leitura de dados a partir de um arquivo é semelhante, mas utiliza a leitura de dados em vez da escrita.

O comando Pascal **reset**, para abertura de um arquivo**,** no formato

Reset (Variável do tipo TEXT)

gera uma instrução VAR, que informa o endereço em memória de uma variável do tipo TEXT, a partir do qual é possível recuperar o nome de um arquivo, e uma instrução RESETARQ, responsável pela abertura de um arquivo para leitura ou escrita no seu início.

A instrução RESETARQ, a partir do endereço da variável TEXT, armazenado na parte temporária do vetor de execução, recupera o nome do arquivo armazenado nesse endereço, e promove a abertura do arquivo. (Figura 4.4.4):

```
RESETARQ()
 {
  char *cadeia;
  char *caracter = "\x0";
  FILE *arq;

   cadeia = new char [255];
   strcpy(cadeia,"\x0");
   while (vetor_execucao[s] != -1)
    {
     *caracter = vetor_execucao[s];
     strcat(cadeia,caracter);
     s--;
    }
   s--;
   arq = fopen(cadeia, "r+t");
   p++;
 }
```

Figura 4.4.4. O algoritmo que implementa a instrução RESETARQ

A implementação da instrução para criação e abertura de um arquivo, REWRITEARQ, assim como a implementação da instrução de abertura de um arquivo para leitura e escrita no seu final, dado por uma instrução APPENDARQ, segue passos análogos.

Outras instruções auxiliares implementadas no compilador, como READKEY, CLRSCR, TEXTCOLOR e TEXTBACKGROUND (correspondentes aos comandos de mesmo nome na linguagem Pascal) possuem implementação bastante simples. (Figura 4.4.5)

```
void READKEY()
 {
  getch();
  p = p + 1;
 }
void CLRSCR()
 {
  clrscr();
  p = p + 1;
 }
void TEXTCOLOR(int color)
 {
  textcolor(color);
  p = p + 2;
 }

void BACKGROUNDCOLOR(int color)
 {
  textbackground(color);
  p = p + 2;
 }
```

Figura 4.4.5. Algoritmos que implementam as instruções READKEY, CLRSCR, TEXTCOLOR e TEXTBACKGROUND

Instruções de ativações de subprogramas

As instruções de ativações de subprogramas tratam, além da própria ativação do término. A instrução responsável pelo primeiro desses passos, dada pela instrução FUNC, foi discutida anteriormente na seção 2.2.4.6. O segundo tipo de instruções, que trata do término de um subprograma, é diferenciado para procedimentos e funções, e foi anteriormente analisado somente para o primeiro desses casos. Para funções, a sua finalização deve manter, na parte temporária do vetor de execução, o valor de retorno esperado para a função.

A instrução responsável pelo término de uma função ENDFUNC, é diferente da instrução para término de um procedimento, ENDPROC, nos seguintes pontos:

- À instrução ENDFUNC deve ser passado como argumento o tamanho (em palavras de memória) do valor a ser retornado pela função
- Como o valor a ser retornado pela função pode ser uma cadeia de caracteres, por causa do tratamento especial dado às mesmas a função deve receber uma informação indicando se o valor retornado é uma cadeia de caracteres.
- A instrução deve deixar disponível, na área temporária do vetor de execução, o retornado pela função.

```
void ENDFUNC(int paramlength,int tamanho_retorno,int retorna_cadeia)
 {
  int endereco_retorno;
  int novo_endereco_base;

  // Guardamos o endereço de retorno para a função, assim
  // como o novo endereco base
  endereco_retorno = vetor_execucao[b+2];
  novo_endereco_base = vetor_execucao[b+1];

  // Recuperamos o valor de retorno para a função
  s = b - paramlength - 1;
  VAR(0,4);
  if (retorna_cadeia == 1)
   {
    int x,i;

     x = vetor_execucao[s];
     vetor_execucao[s] = -1;
     s++;
      i = 0;

     while ( (i<255)&&(vetor_execucao[x+i] != -1) )
       {
        vetor_execucao[s+i] = vetor_execucao[x+i];
        i++;
       }
     s = s + i - 1;
   }
  else
    VAL(tamanho_retorno);

  p = endereco_retorno;
  b = novo_endereco_base;
```

Figura 4.4.6. Algoritmo que implementa a instrução ENDFUNC

```
}
```

5. Conclusões

Neste trabalho, foi desenvolvido um compilador que implementa um subconjunto da linguagem Pascal padrão, proposta por Wirth [3], denominado **Pascal ZIM!**. Este compilador deverá ser utilizado como ferramenta de aprendizado pelos alunos da disciplina *Introdução à Ciência da Computação*, oferecida pelo Departamento de Ciências da Computação a diversos cursos da Universidade de Brasília. Os recursos da linguagem Pascal utilizados em ICC foram totalmente implementados, viabilizando portanto o seu uso já no próximo semestre.

Algumas extensões que poderiam ser feitas são a implementação de um módulo de ajuda (já iniciado), a implementação de outras funções matemáticas (como funções trigonométricas) e um manual de uso do **Pascal ZIM!**.

Anexo I

Interface gráfica para o compilador Pascal ZIM!

O compilador **Pascal ZIM!** foi implementado para funcionar por linha de comando, seguindo a sintaxe:

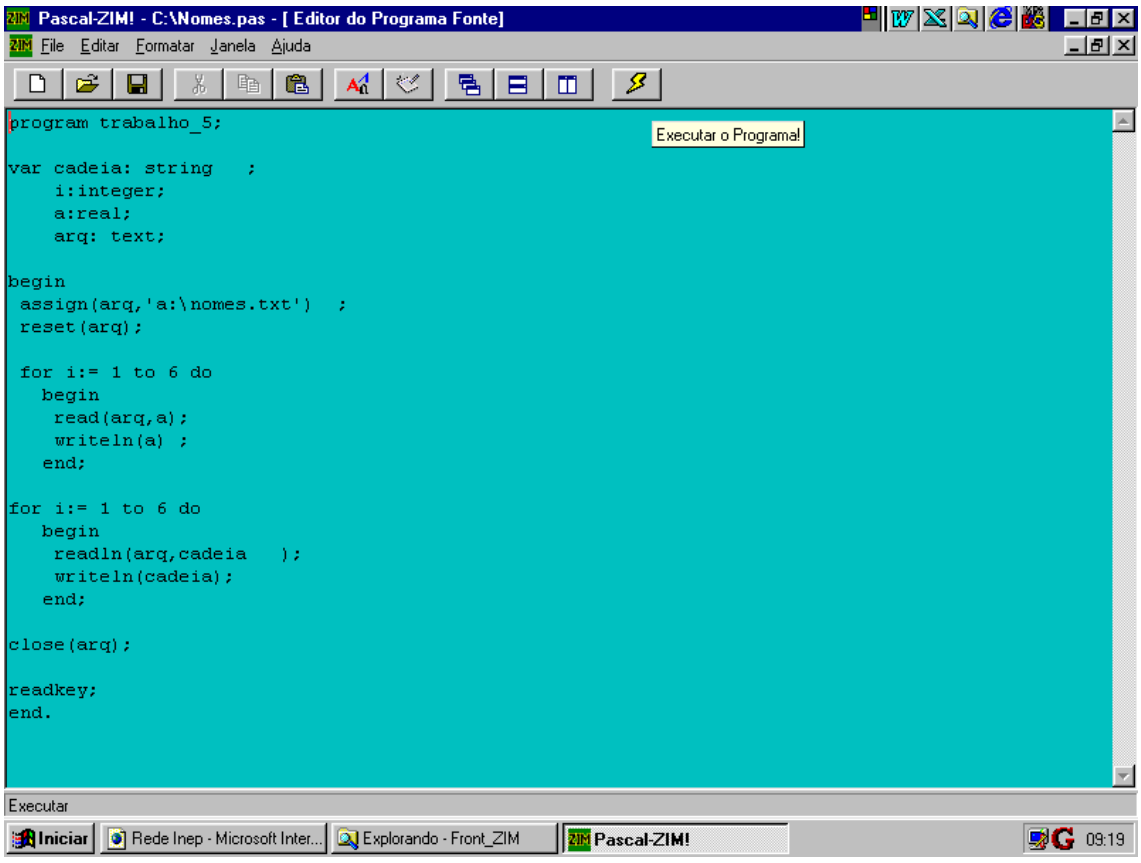
Pzim < *Nome do Arquivo* >

onde *Nome do Arquivo* deve ser um arquivo com a extensão .pas

Diante da necessidade de um ambiente de programação produtivo e de fácil utilização, foi implentado, em conjunto com o compilador, uma interface gráfica.

Algumas características implementadas nessa interface são:

- Um editor de texto, que pode ser configurado de acordo com a necessidade do programador;
- Implementação das operações de *copiar*, *colar* e *recortar* texto no editor.
- Implementação de um mecanismo de pesquisa de palavras
- Possibilidade de criar, abrir e salvar arquivos com extensão .pas
- Comunicação com o compilador **Pascal ZIM!**, para execução de programas
- Visualização do programa objeto gerado para um programa fonte
- Dispozição das janelas contendo o editor de texto e o programa objeto em *cascata*, organizadas verticalmente ou horizontalmente.
- Implementação de um arquivo de ajuda (help), construído a partir do texto descrito no Capítulo 3 deste projeto.



Anexo II

Gramática da linguagem implementada no compilador Pascal ZIM!

```
S -> programa   ;

/* ========== DEFINICAO DE PROGRAMA ========== */

programa -> cabecalho_programa declaracoes bloco_comandos '.' ;

cabecalho_programa -> 'program'.prgatrib 'id'.idatrib  ';' ;

declaracoes -> declaracao_constantes declaracao_tipos declaracao_variaveis
declaracao_subprogramas;

declaracao_subprogramas -> declaracao_subprogramas declaracao_subprograma
| # ;

declaracao_subprograma -> declaracao_funcao
| declaracao_procedimento ;

/* ========== DEFINICAO DE CONSTANTES ========== */

declaracao_constantes -> 'const' bloco_constantes
| #;

bloco_constantes-> lista_constantes
| bloco_constantes lista_constantes;

lista_constantes -> lista_identificadores '=' 'num'.numatrib ';'
| lista_identificadores '=' valor_booleano ';'
| lista_identificadores '=' 'cadeia'.cadeiaatrib ';'
| lista_identificadores '=' 'caracter'.caracatrib ';'
| lista_identificadores '=' '-' 'num'.numatrib ';';

valor_booleano -> 'TRUE'
| 'FALSE';

lista_identificadores -> identificador
| lista_identificadores ',' identificador;

identificador -> 'id'.idatrib ;

/* ========== DEFINICAO DE TIPOS ========== */

declaracao_tipos -> 'type' bloco_tipos
| # ;

bloco_tipos ->  bloco_array
|  bloco_array bloco_tipos
|  bloco_record
|  bloco_record bloco_tipos ;

/* == VETORES == */

bloco_array-> lista_identificadores '=' 'array'.artrib '[' limites_vetor
']' 'of'
tipo ';' ;

limites_vetor -> dimensao
| limites_vetor ',' dimensao;

dimensao -> 'num'.numatrib1 '.' '.' 'num'.numatrib2 ;

/* == REGISTROS == */
```

```
bloco_record -> lista_identificadores '=' 'record' campos 'end' ';' ;

campos-> lista_campos
| lista_campos campos ;

lista_campos -> lista_identificadores ':' tipo ';' ;

tipo -> tipo_definido
| 'array'.artrib '[' limites_vetor ']' 'of' tipo
| 'record' campos 'end' ;

tipo_definido -> tipo_predefinido
| 'id'.tipoatrib ;

tipo_predefinido -> 'string'.tpatrib string_tail
| 'integer'.tpatrib
| 'char'.tpatrib
| 'boolean'.tpatrib
| 'real'.tpatrib ;

string_tail -> '[' 'num'.stratrib ']'
| #;

/* ========== DEFINICAO DE VARIAVEIS ========== */

declaracao_variaveis -> 'Var' bloco_variaveis
| #;

bloco_variaveis-> lista_variaveis
| lista_variaveis bloco_variaveis;

lista_variaveis-> lista_identificadores ':' 'text'.tpatrib ';'
| lista_identificadores ':' tipo ';';

/* ========== DEFINICAO DE FUNCOES ========== */

declaracao_funcao -> cabecalho_funcao  declaracoes bloco_comandos ';' ;

cabecalho_funcao -> 'function' 'id'.funcatrib argumentos ':'
tipo_predefinido ';' ;

argumentos -> '(' lista_parametros ')'
| # ;

lista_parametros -> parametros
| lista_parametros ';' parametros ;

parametros -> lista_identificadores ':' tipo_definido
| 'Var' lista_identificadores ':' tipo_definido ;

/* ========== DEFINICAO DE PROCEDIMENTOS ========== */

declaracao_procedimento -> cabecalho_procedimento declaracoes
bloco_comandos ';' ;

cabecalho_procedimento ->  'procedure' 'id'.procatrib argumentos ';' ;

/* ========== DEFINICAO DE COMANDOS ========== */

bloco_comandos -> 'begin' comandos_bloco 'end' ;

comandos_bloco -> lista_comandos
| # ;

lista_comandos -> comandos ';'
| lista_comandos comandos ';' ;

comandos -> variavel ':=' expressao
```

```
| bloco_comandos
| 'id'.procatrib
| 'id'.procatrib2 '(' lista_expressoes ')'
| 'while' expressao 'do' comandos
| 'for' variavel ':=' expressao 'to' expressao 'do' comandos
| 'for' variavel ':=' expressao 'downto' expressao 'do' comandos
| 'repeat' lista_comandos 'until' expressao
| 'read' '(' parametros_read ')'
| 'readln' '(' parametros_read ')'
| 'write' '('lista_expressoes ')'
| 'writeln' '('lista_expressoes ')'
| 'assign' '(' 'id'.asnatrib ',' parametro_string ')'
| 'reset' '(' 'id'.restatrib ')'
| 'rewrite' '(' 'id'.restatrib ')'
| 'append' '(' 'id'.restatrib ')'
| 'close' '(' 'id'.restatrib ')'
| 'readkey'
| 'clrscr'
| 'textcolor' '(' lista_cores ')'
| 'textbackground' '(' lista_cores ')'
| 'gotoxy' '(' expressao ',' expressao ')'
|  'if' expressao 'then' comandos 'else' comandos
|  'if' expressao 'then'comandos;


parametro_string -> 'cadeia'.cdatrib
| 'id'.cadeiatrib ;

lista_cores -> cor
| lista_cores '+' cor ;

cor -> 'num'.numatrib
| 'blue'
| 'green'
| 'cyan'
| 'red'
| 'magenta'
| 'brown'
| 'lightgray'
| 'darkgray'
| 'lightblue'
| 'lightgreen'
| 'lightcyan'
| 'lighred'
| 'lighmagenta'
| 'yellow'
| 'white'
| 'blink'
| 'black';

variavel -> 'id'.varatrib idtail ;

idtail -> '.' 'id'.idatrib idtail
| '[' lista_expressoes ']' idtail
| #  ;

/* =========== DEFINICAO DE EXPRESSOES =========== */

expressao -> expressao_aritmetica
| expressao '=' expressao_aritmetica
| expressao '<' expressao_aritmetica
| expressao '>' expressao_aritmetica
| expressao '<>' expressao_aritmetica
| expressao '>=' expressao_aritmetica
| expressao '<=' expressao_aritmetica ;

expressao_aritmetica -> termo
|  expressao_aritmetica '+' termo
```

```
|  expressao_aritmetica '-' termo
|  expressao_aritmetica 'OR' termo ;

termo -> fator
| termo '*' fator
| termo '/' fator
| termo 'DIV' fator
| termo 'MOD' fator
| termo 'AND' fator ;

fator -> variavel
| 'num'.numatrib
| 'caracter'.caracatrib
| 'cadeia'.cadeiaatrib
| valor_booleano
| '(' expressao ')'
| 'id'.funcatrib '(' lista_expressoes ')'
| 'not' fator
| '-' fator
| 'length' '(' parametro_string ')'
| 'chr' '(' expressao_aritmetica ')'
| 'ord' '(' expressao_aritmetica ')'
| 'eof' '('id'.eofatrib ')' ;


lista_expressoes -> expressao_lista
| lista_expressoes ',' expressao_lista ;

expressao_lista -> expressao;

parametros_read -> variavel
| 'id'.arqatrib ',' variavel;
```

Anexo III

Exemplos de programas submetidos ao **Pascal ZIM!**

1. Programa Fonte:

```
program fatorial;

var n: integer;

function fat(n:integer):integer;
  begin
   if n > 1 then
     fat := n*fat(n-1)
   else
     fat:= 1;
  end;

begin
clrscr;
textcolor(green);
textbackground(blue);
clrscr;
write('Entre com um valor para o fatorial de n: ');
read(n);
write('O fatorial de ',n, ' : ',fat(n));
readkey;
end.
```

1. Programa Objeto

```
PROGRAM(1,4000,53)
FUNCTION(1,1000,4)
VAR(0,-1)
VAL(1)
CONST(1)
BIG
DO(42)
VAR(0,4)
VAR(0,-1)
VAL(1)
VAR(0,-1)
VAL(1)
CONST(1)
SUBTRACT
CALLFUNC(1,-30)
MULTIPLY
ASSIGN(1)
GOTO(49)
VAR(0,4)
CONST(1)
ASSIGN(1)
ENDFUNC(1,1,0)
CLRSCR
TEXTCOLOR(2)
BACKGROUNDCOLOR(1)
CLRSCR
CONST(-1)
CONST(32)
CONST(58)
CONST(110)
CONST(32)
CONST(101)
CONST(100)
CONST(32)
CONST(108)
CONST(97)
CONST(105)
CONST(114)
CONST(111)
CONST(116)
CONST(97)
CONST(102)
CONST(32)
CONST(111)
CONST(32)
CONST(97)
CONST(114)
CONST(97)
CONST(112)
CONST(32)
CONST(114)
CONST(111)
CONST(108)
CONST(97)
CONST(118)
CONST(32)
CONST(109)
CONST(117)
CONST(32)
CONST(109)
CONST(111)
CONST(99)
CONST(32)
CONST(101)
CONST(114)
CONST(116)
CONST(110)
CONST(69)
WRITE(6)
VAR(0,3)
READ(1)
CONST(-1)
CONST(32)
CONST(101)
CONST(100)
CONST(32)
CONST(108)
CONST(97)
CONST(105)
CONST(114)
CONST(111)
CONST(116)
CONST(97)
CONST(102)
CONST(32)
CONST(79)
WRITE(6)
VAR(0,3)
VAL(1)
WRITE(1)
CONST(-1)
CONST(32)
CONST(58)
CONST(32)
WRITE(6)
VAR(0,3)
VAL(1)
CALLFUNC(0,-200)
WRITE(1)
READKEY
ENDPROG
```

2. Programa Fonte:

```
program Matrizes;

var a: array [1..5,1..6] of integer;
i,j,contador: integer;

begin
  contador:= 1;
  for i:= 1 to 5 do
   for j:= 1 to 6 do
     begin
       a[i,j] := contador;
       contador:= contador + 1;
     end;

  for i:= 1 to 5 do
    for j:= 1 to 6 do
      writeln(a[i,j]);

end.
```

2. Programa Objeto:

```
PROGRAM(33,4000,4)
VAR(0,35)
CONST(1)
ASSIGN(1)
VAR(0,33)
CONST(1)
ASSIGN(1)
CONST(5)
VAR(0,33)
VAL(1)
BIGEQUAL
DO(119)
VAR(0,34)
CONST(1)
ASSIGN(1)
CONST(6)
VAR(0,34)
VAL(1)
BIGEQUAL
DO(104)
VAR(0,3)
VAR(0,33)
VAL(1)
INDEX(1,5,9)
VAR(0,34)
VAL(1)
INDEX(1,6,9)
CALCPOSVET(2,1)
VAR(0,35)
VAL(1)
ASSIGN(1)
VAR(0,35)
VAR(0,35)
VAL(1)
CONST(1)
ADD
ASSIGN(1)
VAR(0,34)
VAR(0,34)
VAL(1)
CONST(1)
ADD
ASSIGN(1)
GOTO(35)
VAR(0,33)
VAR(0,33)
VAL(1)
CONST(1)
ADD
ASSIGN(1)
GOTO(18)
VAR(0,33)
CONST(1)
ASSIGN(1)
CONST(5)
VAR(0,33)
VAL(1)
BIGEQUAL
DO(213)
VAR(0,34)
CONST(1)
ASSIGN(1)
CONST(6)
VAR(0,34)
VAL(1)
BIGEQUAL
DO(198)
VAR(0,3)
VAR(0,33)
VAL(1)
INDEX(1,5,15)
VAR(0,34)
VAL(1)
INDEX(1,6,15)
CALCPOSVET(2,1)
VAL(1)
WRITE(1)
WRITE(5)
VAR(0,34)
VAR(0,34)
VAL(1)
CONST(1)
ADD
ASSIGN(1)
GOTO(143)
VAR(0,33)
VAR(0,33)
VAL(1)
CONST(1)
ADD
ASSIGN(1)
GOTO(126)
ENDPROG
```

Bibliografia

1. AHO, A. V., SETHI, R. e ULLMAN, J.D. *Compiladores: Princípios, Técnicas e Ferramentas.* Rio de Janeiro, LTC, 1995.

2. SILVA, José Carlos G. Linguagens de Programação : Conceito e Avaliação. São Paulo, McGraw - Hill, 1988

3. HANSEN, P. B. *On Pascal Compiler* . Prentice Hall Publ, 1985.

4. WIRTH, N. *The design of a Pascal compiler*. Software--Practice and Experience, v. 1 (1971)