MÉTODO PRÁTICO DE GERAÇÃO
DE
AVALIADORES DE ATRIBUTOS EM UM PASSO

Edmar Wienskoski Junior

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof: José Lucas Mourão Rangel Netto, Ph.D.
(Presidente)

Prof[a]: Sueli Bandeira Texeira Mendes, Ph.D.

Prof: Arndt Von Staa, Ph.D.

RIO DE JANEIRO, RJ - BRASIL
MARÇO DE 1990

WIENSKOSKI JUNIOR, EDMAR

Método prático de geração de avaliadores de atributos em um passo [Rio de Janeiro] 1990.

VII, 62 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas, 1990)

Tese - Universidade Federal do Rio de Janeiro, COPPE.

1. Gramática de Atributos I. COPPE / UFRJ II. Título (série).

À companheira de todas as horas

AGRADECIMENTOS

À prof^{a} Leila pelo apoio recebido para iniciar um trabalho na área de compiladores.

Ao prof Rangel pela compreensão e dedicação e cuja firmeza de propósitos e paciência sustentou-me até a finalização deste trabalho.

À minha esposa o incentivo para recomeçar sempre, derrotando o desânimo e ultrapassando os obstáculos.

Aos meus pais e minha irmã o esforço realizado com amor para minha educação.

Aos demais membros da banca que muito me honraram com sua participação.

À RACIMEC através do Dr. Simão Brayer a possibilidade de conciliar o trabalho com o curso de mestrado.

À Revista Nacional através do Sr. Mauritônio Meira pelo uso dos recursos materiais para a apresentação deste trabalho.

Resumo da Tese Apresentada à COPPE / UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M. Sc.)

MÉTODO PRÁTICO DE GERAÇÃO DE AVALIADORES DE ATRIBUTOS EM UM PASSO

EDMAR WIENSKOSKI JUNIOR

março de 1990

Orientador: José Lucas Mourão Rangel Netto
Programa: Engenharia de Sistemas e Computação

Trata-se da construção de uma ferramenta sob a forma de um gerador de programa avaliador de atributos.

A partir da descrição sintática e semântica da linguagem alvo, esta ferramenta irá gerar duas saídas: a lista das produções da linguagem alvo no formato exigido pelo gerador R*S desenvolvido por J. L. Rangel e S. M. Schneider, e o avaliador de atributos escrito na linguagem Pascal.

Para o analisador sintático do compilador alvo, o método adotado deriva do LR e se chama R*S. Para formar o avaliador de atributos, definimos uma gramática de atributos que deve ser avaliada em um percurso da árvore de derivação da esquerda para a direita como proposto por G. V. Bochmann.

A descrição da linguagem alvo deve ser feita através de uma linguagem criada para este fim, chamada LINDA (LINguagem de Descrição de Atributos). Esta tese descreve a sintaxe e o uso desta linguagem, descreve ainda uma implementação do compilador LINDA e detalha o desenvolvimento de um compilador exemplo usando a linguagem LINDA.

Abstract of Thesis presented to COPPE / UFRJ as partial fulfilment of the requirements for the degree of Master of Science (M. Sc.)

PRACTICAL METHOD FOR GENERATION
OF
ONE PASS ATTRIBUTE EVALUATORS

EDMAR WIENSKOSKI JUNIOR

march, 1990

Thesis Supervisor: José Lucas Mourão Rangel Netto
Department: Computer Science

This thesis describes the construction of a tool, a program generator of attribute evaluators.

Starting from a description of the syntax and semantics of the target language, this tool generates two outputs: The list of target language productions in R*S generator format and the attribute evaluator program, written in Pascal. The R*S generator was developed by J. L. Rangel and S. M. Schneider.

The syntactic analysis of the target compiler follows a method derived from the LR parsing, named R*S, and for attribute evaluation, we define an attribute grammar to be evaluated in one pass over the parse tree, from left to right as proposed by G. V. Bochmann.

The description of target language will be done in a specific defined language, named LINDA (Attribute Description Language). This thesis describes that language, a LINDA compiler implementation and the development of a compiler-example using the LINDA language.

ÍNDICE

CAPÍTULO I

INTRODUÇÃO

A estrutura interna de um compilador pode ser dividida em quatro blocos principais: analisador léxico, analisador sintático, analisador semântico e gerador de código.

O analisador léxico compreende a parte do compilador que recebe e examina o programa a ser compilado. Sua função básica é substituir uma cadeia de caracteres por uma cadeia de códigos, onde cada código representa um entre os elementos básicos da linguagem tais como identificadores, números, palavras-chave, etc. Os algoritmos utilizados para a implementação de um analisador léxico se baseiam em autômatos finitos, ou tabelas de classificação ou ainda são programados manualmente. Este bloco do compilador é de fácil realização e envolve técnicas bastantes conhecidas e estudadas (AHO, 1986).

O analisador sintático recebe então esta cadeia de códigos e verifica a exatidão das estruturas do programa que lhe foi submetido segundo a definição da linguagem. Para tanto, existem diversos algoritmos, sendo os mais usados e mais eficientes baseados em autômatos de pilha dirigidos por tabelas. Para a construção destas tabelas, o projetista do compilador tem ao seu alcance algoritmos que partem de uma descrição BNF da linguagem desejada. Para um estudo bem detalhado sobre diversos métodos, ver AHO et alia (1986).

Cabe ao analisador semântico verificar as restrições contextuais da linguagem, como por exemplo: definição e utilização das variáveis, regras de escopo de variáveis, conversão automática de tipos, etc. Tal é a diversidade de funções entre os diversos tipos de máquina e entre as diversas linguagens, que frequentemente o analisador semântico é implementado manualmente.

Atualmente o esforço de desenvolvimento se volta para a automação da análise semântica, com o objetivo de facilitar sua construção, de garantir sua eficiência e sua portabilidade, como aconteceu com as demais etapas da compilação.

Não podemos citar aqui as inúmeras técnicas que emergem dos centros de pesquisa, pois são diferentes entre si, e muitas delas ainda não foram suficientemente aplicadas em problemas práticos, ainda não foram bem difundidas e não estão consagradas. Entretanto a técnica que parece dar resposta a mais problemas do ponto de vista prático é a da **gramática de atributos.** DERANSART et alia (1988) dá uma idéia bastante exata do estado da arte numa publicação que classifica mais de 600 trabalhos nesta área e apresenta a maioria dos sistemas que se utlizam de gramática de atributos.

Objetivamos facilitar o desenvolvimento de um compilador, colocando à disposição do projetista uma ferramenta capaz de realizar manipulações, verificações, e outras operações na gramática por ele elaborada.

Esta tese visa a construção desta ferramenta, condensando várias técnicas existentes para descrever as ações semânticas da linguagem alvo através de uma linguagem projetada para esta finalidade. Estas ações semânticas estarão ligadas às regras de uma gramática que, em geral, não foi a mesma utilizada para a descrição sintática da linguagem alvo.

Esta ferramenta é um gerador de programas para avaliação de atributos. Ela recebe do projetista de linguagens uma gramática de atributos que define ações semânticas associadas às regras. O método utilizado visa combinar a facilidade de definição de gramáticas de

atributos em um percurso, da esquerda para a direita (BOCHMANN, 1976), com a facilidade de geração de analisadores sintáticos R*S (RANGEL, 1988).

Para a utilização do método citado, seria necessário dispor de um analisador descendente da gramática associada, o que dificulta sua definição e a dos atributos. Outra alternativa é construir a árvore de derivação utilizando algum analisador disponível, porém isto implicaria em perda de espaço.

Na figura I.1 mostramos um diagrama da forma proposta. A análise sintática será realizada por um analisador R*S, gerando uma tabela de nomes e uma sequência de derivação para uma árvore de derivação esquerda. Deste modo a gramática que define os atributos não precisará ser a mesma utilizada como entrada para o gerador R*S, porque uma vez conhecida a sequência de derivação da cadeia de entrada, esta informação pode ser aplicada diretamente até a uma gramática ambígüa sem incorrermos em erro. A única restrição é a de que a gramática usada para os atributos seja uma cobertura (AHO et alia, 1972 e NIJHOLT, 1980) da gramática que definiu a sintaxe da linguagem. Entretanto o projetista deverá fornecer uma relação de correspondência entre as regras da gramática da sintaxe abstrata e as produções da linguagem alvo. Na figura I.2 temos um exemplo de como se aplica esta correspondência.

Desta forma a nossa ferramenta toma a forma de um compilador, sendo as descrições das gramáticas e dos seus vínculos realizadas através de uma linguagem especialmente projetada chamada LINDA (LINguagem de Descrição de Atributos). Algumas idéias para a criação desta linguagem foram baseadas na tese de mestrado de Dora Toma Taguata orientada por Estevam Gilberto de Simone (TAGUATA, 1982).

O compilador LINDA fará uma verificação dos tipos dos atributos, verificará se a gramática da sintaxe abstrata obedece às regras descritas no método de BOCHMANN (1976), e irá gerar um programa avaliador de atributos incluindo as respectivas ações semânticas. Também fornecerá um arquivo com as produções sintáticas da linguagem alvo no formato de entrada do gerador R*S.

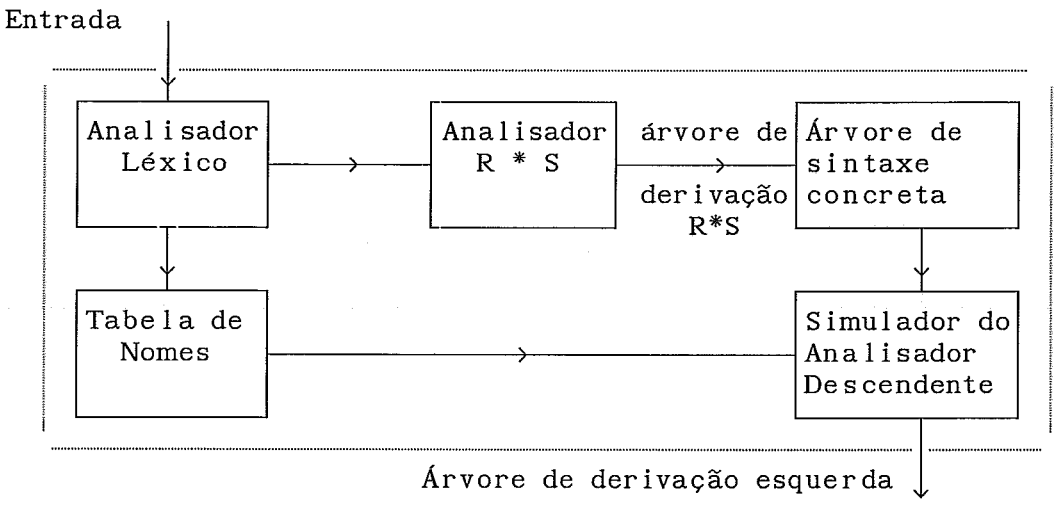


Figura I.1 - Simulador do analisador descendente

Gramática da sintaxe concreta

1 - E → E + T
2 - E → T
3 - T → T * F
4 -T → F
5 -F → (E)
6 -F → a

Gramática da sintaxe abstrata

Soma - E → E + E
Mult - E → E * E
Ident - E → a

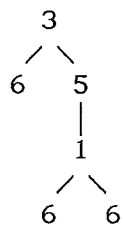
Correspondência entre as regras

1 - Soma
3 - Mult
6 - Ident

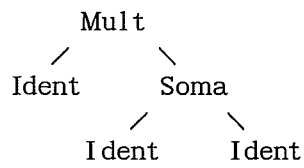
Exemplo para a * (a + a)

derivação direita: 6 4 6 4 2 6 4 1 5 3 2
derivação R*S: 6 6 6 1 5 3

árvore R*S



árvore da sintática abstrata



derivação esquerda: Mult Ident Soma Ident Ident

Figura I.2 - Correspondência entre sintaxe concreta e sintaxe abstrata

No capítulo II descrevemos brevemente o conceito de gramática de atributos e em especial o método citado.

O capítulo III descreve a especificação sintática da linguagem LINDA e mostra a utilização dos seus recursos com exemplos, sempre que possível.

No capítulo IV apresentamos a implementação de um compilador LINDA, com sua organização interna, métodos utilizados e as restrições de implementação desta versão.

No capítulo V apresentamos um exemplo completo de compilador, começando com a linguagem alvo, funções desejáveis e a escolha de atributos para estas funções. Descreve-se ainda a implementação dos analisadores léxico e sintático, e as rotinas complementares necessárias para o funcionamento com o avaliador de atributos.

No capítulo VI apresentamos conclusões e sugestões para outros trabalhos.

CAPÍTULO II

GRAMÁTICA DE ATRIBUTOS

II.1 - Fundamentos

O conceito de gramática de atributos foi primeiramente apresentado por KNUTH (1968) como uma extensão para traduções dirigidas pela sintaxe.

Cada nó da árvore gerada pelo analisador sintático, é **decorado** com atributos descrevendo as propriedades deste nó. As informações coletadas pelos atributos de um nó provém da vizinhança deste nó. É tarefa do analisador semântico computar estes atributos e verificar sua consistência.

O conceito de gramática de atributos tem provado ser de grande ajuda na representação da **decoração** (avaliação dos atributos) de uma árvore, porque ela constitui uma definição formal das propriedades da linguagem e é uma especificação formal da analise semântica. Quando projetamos esta especificação, não precisamos nos preocupar com a sequência em que os atributos serão calculados, porque a escolha dessa sequência, em parte, pode ser feita automaticamente. A forma do armazenamento dos atributos também não é refletida na especificação. Vamos começar por assumir que todos os atributos pertencentes a um nó serão guardados junto a ele, na árvore sintática. Mais tarde consideraremos a otimização deste armazenamento.

Seja G = (N, T, P, Z) uma gramática livre de contexto onde **N** é o conjunto de símbolos não-terminais, **T** é o conjunto de símbolos terminais, **P** é o conjunto de produções da gramática e **Z** é o símbolo inicial.

Uma gramática de atributos é baseada numa gramática livre de contexto G = (N, T, P, Z). Ela associa um conjunto A(X) de atributos a cada símbolo **X**, no vocabulário de **G**. Cada atributo representa um valor local de uma característica do símbolo **X**, ou seja, os valores são sensíveis ao contexto. Denotamos **X.a** para indicar que o atributo **a** é um elemento de A(X).

Cada produção **p** pertencente ao conjunto **P** pode ser escrita da forma:

$$p : X_0 \longrightarrow X_1 \; X_2 \; \ldots \; X_{n_p}$$

onde X_i pertence ao conjunto **N** para qualquer **i** de 0 a n_p. Observar que retiramos das produções os símbolos terminais, formando assim o que chamamos de **fundamental** da gramática livre de contexto, ou ainda **gramática reduzida**. A relação entre atributos e símbolos terminais será discutida mais adiante.

Cada nó da árvore sintática de uma sentença de L(G) é associado a um conjunto particular de valores para os atributos de algum símbolo **X** do vocabulário de **G**. Estes valores são estabelecidos por regras semânticas. Cada produção **p** $\in$ **P** está associada a conjunto de regras semânticas R(p), que denotamos por:

$$R(p) = \{ \; X_i.a \longleftarrow f \; (X_j.b, \; \ldots \; X_k.c) \; \} \text{ onde } 0 \leq i,j,k \leq n_p$$

Cada regra define um atributo $X_i.a$ em termos dos atributos $X_j.b \; \ldots \; X_k.c$ de símbolos da mesma produção **p**.

Explicando de uma maneira mais informal, para cada não-terminal associamos variáveis e para cada produção associamos expressões envolvendo estas variáveis.

Para cada produção p : $X_0 \longrightarrow X_1 \ \ldots \ X_{np}$ pertencente a **P** o conjunto de ocorrências de definição de atributos é:

$$OD(p) = \{ X_i.a \mid X_i.a \leftarrow f\ (\ldots) \in R(p) \}$$

Um atributo X.a é chamado **sintetizado** se existe uma produção **p** em que o símbolo **X** aparece à esquerda da produção e o atributo X.a pertence ao OD(p), e é chamado **herdado** se existe uma produção **q** em que o símbolo **X** aparece à direita da produção e o atributo X.a pertence ao OD(p).

Atributos sintetizados de um nó são resultado imediato da computação da sub-árvore deste nó, enquanto que os valores dos atributos herdados provém da vizinhança superior do nó (pai e irmãos do nó).

Denotamos o subconjunto de atributos sintetizados do símbolo X como S(X) e o subconjunto de atributos herdados do símbolo X como H(X).

Neste trabalho definimos que os conjuntos de atributos devem satisfazer a seguinte condição:

$$\text{Para todo } X \in N,\ H(X) \cap S(X) = \varnothing$$

Da condição acima deduzimos que se um determinado atributo foi definido como sintetizado, este não poderá ser redefinido como herdado do mesmo não-terminal, e vice-versa.

Cada conjunto de regras semânticas, define a computação dos elementos de $S(X_0)$ e $H(X_i)$ $1 \leq i \leq np$, em termos dos elementos de $H(X_j)$ $\cup$ $S(X_j)$ $0 \leq j \leq np$.

Isto significa que para uma determinada produção, as suas regras semânticas computam os atributos sintetizados do lado esquerdo da produção, $S(X_0)$, e os atributos herdados do lado direito da produção, $H(X_i)$ $1 \leq i \leq np$, em função de quaisquer outros atributos. Observamos ainda que o cálculo dos atributos é definido dentro do

escopo de uma produção, ou seja, as regras semânticas não podem referenciar atributos de outras produções.

Os símbolos terminais de uma sentença aparecem sempre nas folhas da árvore sintática. Isto é a própria definição de símbolo terminal. Deste modo compreendemos que símbolos terminais só podem possuir atributos sintetizados, cujos valores serão usados pela vizinhança do nó.

Entretanto podemos declarar, em aparente contradição, que símbolos terminais só podem possuir atributos herdados, porque símbolos terminais só aparecem no lado direito das produções, o que combina com a definição de atributos herdados. Teoricamente aceitamos a existência de ambos para símbolos terminais. Neste trabalho mostraremos que símbolos terminais só poderão ter atributos herdados por uma restrição da ordenação das avaliações dos atributos adotada por nós.

Uma produção $p : X_0 \longrightarrow X_1\ X_2\ \ldots\ X_{np}$ tem uma ocorrência de atributo **(a, k)** se:

$$a \in (H(X_k) \cup S(X_k)) \text{ onde } 0 \leq k \leq np$$

Um conjunto de **dependência** é um conjunto de ocorrências de atributos cujos valores serão usados no cálculo de um outro atributo. Denotamos por $D^{(p)}_{(a,k)}$ o conjunto de ocorrências de atributos (i, j) da produção **p** para o cálculo de (a, k).

II.2 - Método de Bochmann

BOCHMANN (1976) propõe que a avaliação dos atributos seja feita sobre a árvore de parse, visitando a árvore a partir da raiz indo do lado esquerdo para o lado direito.

Dada uma gramática de atributos, os atributos de qualquer árvore de derivação podem ser calculados em um único percurso, da esquerda para a direita, se os conjuntos de dependências das regras semânticas de qualquer produção $p : X_0 \longrightarrow X_1\ X_2\ \ldots\ X_{np}$ da gramática satisfaz as seguintes condições:

$$D^{(p)}_{(S,0)} \cap Z = \varnothing \qquad (1)$$

$$D^{(p)}_{(I,k)} \cap \left\{ Z \cup \bigcup_{j=k}^{np} (H(X_j) \cup S(X_j)) \right\} = \varnothing \qquad (2)$$

Para $1 \leq k \leq np$ e $i \in H(X_k)$

Explicando as condições acima podemos dizer que:

(1) Nenhum dos atributos sintetizados depende do símbolo inicial **Z**.

(2) Nenhum dos atributos herdados de um símbolo não-terminal numa determinada posição de uma produção podem ter como dependentes o símbolo inicial **Z** nem outro atributo, sintetizado ou herdado, de qualquer símbolo que ocorra à sua direita na mesma produção.

Para cada expressão envolvendo atributos, verificaremos que:

(1) Caso o atributo que aparece à esquerda da expressão seja sintetizado, então à direita da expressão não poderá aparecer atributos do símbolo inicial Z.

(2) Caso o atributo que aparece à esquerda da expressão for do tipo herdado, então à direita da expressão não poderão aparecer referências a atributos de símbolos que estejam à direita do símbolo correspondente ao atributo que aparece a esquerda da expressão.

II.3 - Aplicação do método no avaliador de atributos

Atributos tais como o valor de uma constante ou o descritor de um identificador, os quais aparecem junto com a construção da árvore sintática, são chamados atributos **intrínsecos** (GOOS, 1984). Para manter a generalidade e a flexibilidade do compilador LINDA, o analisador léxico grava estas ocorrências em formato ASCII, tal como encontradas no programa fonte, em um arquivo intermediário. Durante a avaliação dos atributos, as regras semânticas podem invocar funções que leêm este

arquivo e retornam valores apropriados. Deste modo evitamos que a árvore sintática tenha que carregar dados de tamanho pré-fixado e dependente da linguagem alvo, além de economizar área de memória. Esta técnica se torna particularmente viável pelo fato de que o arquivo pode ser acessado sequencialmente, pois a ordem em que as regras semânticas pegam os dados coincide com a ordem que o analisador léxico varre o arquivo de entrada.

A partir da árvore de derivação construída pelo analisador sintático, o programa avaliador irá varrer a árvore recursivamente, a partir da raiz, calculando os atributos de cada nó.

Ao descer de um nó para seu filho, o avaliador calcula os atributos herdados deste nó filho, desce para outros nós, caso existam, numa operação recursiva, para então calcular os atributos sintetizados. A operação se repete para todos nós filhos até que não haja mais, em toda a árvore, nós a serem visitados.

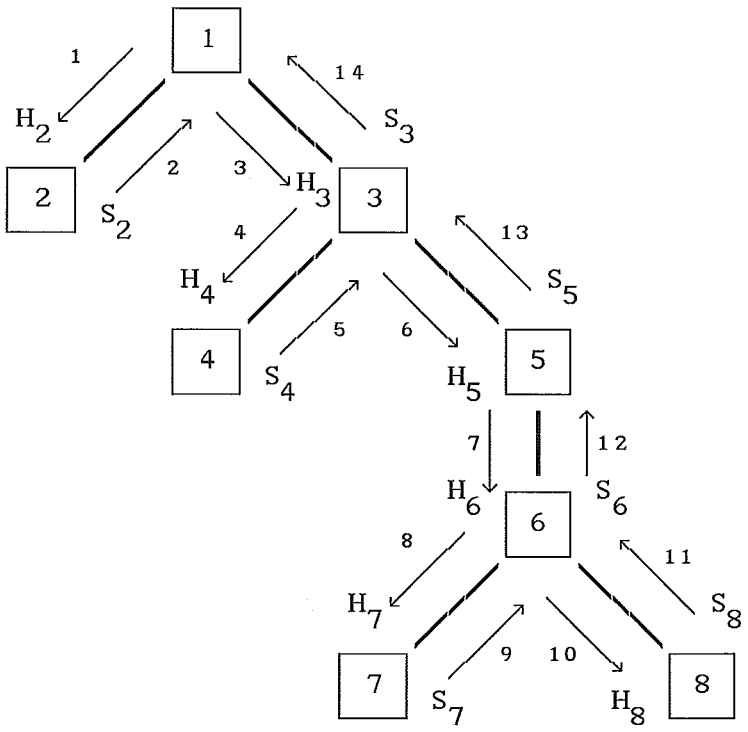


Figura II.1

Para ilustração, vamos tomar como exemplo a árvore de derivação da figura II.1.

As setas numeradas indicam a ordem em que os atributos serão calculados. Quando a seta aponta para baixo, na ponta da seta se encontra a indicação H_i denotando o conjunto de atributos herdados do nó indice **i**. A direção da seta indica que estes atributos serão calculados a partir de dados do nó pai. Quando a seta aponta para cima, na base da seta se encontra a indicação S_i denotando o conjunto de atributos sintetizados do nó indice **i** calculados neste e que serão **copiados** para o nó pai, ficando disponíveis para cálculos de outros atributos.

Considerando que o programa avaliador é uma rotina recursiva, e que como qualquer outra rotina pode receber parâmetros e retornar valores, vamos analisar o esquema de alocação de memória que será usado no avaliador.

Inicialmente o avaliador recebe como parâmetro os atributos herdados do nó. A seguir ele aloca espaço de memória para os atributos herdados e atributos sintetizados de seu primeiro nó filho (o nó filho mais à esquerda), então ocorre a recursividade, sendo os parâmetros dessa chamada os atributos herdados do nó filho, e os valores retornados serão guardados na área alocada para os atributos sintetizados. A operação de alocação de memória e a chamada recursiva se repete para cada um de seus nós filhos. Ao final, todos os atributos da sub-árvore abaixo de si estarão calculados. Os seus atributos sintetizados são então calculados, toda a memória alocada é liberada, e o avaliador termina retornando os atributos sintetizados do nó.

Entrando mais detalhadamente na questão, todos os parâmetros passados e recebidos são acessados por ponteiros. As rotinas de gerenciamento de memória (alocação e liberação) disponíveis no ambiente de programação Pascal também trabalham com ponteiros.

O compilador LINDA traduz as regras semânticas elaboradas pelo projetista da linguagem transformando todas as referências de atributos em acessos por ponteiros, usando como campo o nome do atributo e como ponteiro um daqueles blocos de memória que será alocado pelo avaliador. O compilador LINDA se encarrega de estabelecer o tamanho correto do bloco de memória a ser alocado através do controle do número e tipos dos atributos, bem como o emprego correto dos ponteiros destes blocos através da posição em que o símbolo referenciado aparece na produção.

CAPÍTULO III

LINGUAGEM LINDA

III.1 - Introdução

O objetivo desta tese é construir uma ferramenta capaz de facilitar o desenvolvimento de um compilador. Para tanto, estabelecemos no capítulo anterior os métodos a serem usados. Agora definiremos a maneira de utilizar os recursos desta ferramenta.

Entendemos como linguagem alvo, aquela para a qual queremos construir um compilador. Assim, se o projetista de linguagens, ou melhor, se o usuário desta ferramenta deseja um compilador para a linguagem *dBase*, isto é, cuja linguagem fonte é *dBase*, *dBase* será a linguagem alvo do sistema LINDA. A figura III.1 apresenta o esquema do sistema para *dBase*.

Para formalizar as definições da linguagem alvo, dadas pelo projetista, foi criada uma linguagem chamada LINDA, LINguagem de Descrição de Atributos.

O compilador LINDA gera código fonte em Pascal e por isso escolhemos a forma dos construtos e as palavras reservadas da linguagem LINDA o mais assemelhado possível com o Pascal. Alternativamente, o compilador aceita sinônimos em português das palavras reservadas.

A escolha das palavras reservadas foi orientada para a clareza do programa, documentação e evitar erros de programação.

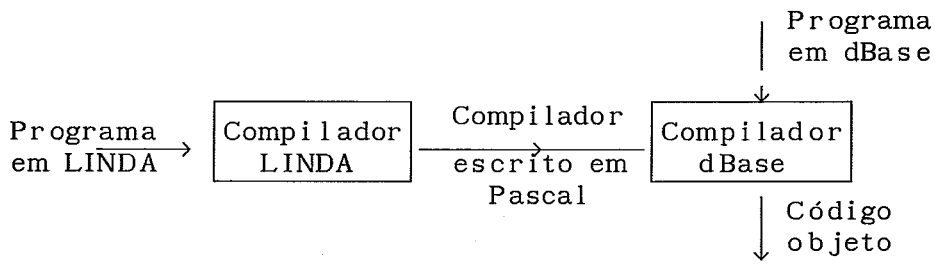


Figura III.1 - Esquema do sistema

III.2 - Elementos da linguagem LINDA

A linguagem LINDA, é formada por palavras reservadas, números inteiros, strings, identificadores e alguns delimitadores. Os números inteiros devem sempre ser representados na base decimal, as strings são delimitadas por aspas simples, identificadores devem começar com uma letra e para delimitar comentários usamos um par de chaves.

III.3 - Estrutura de um programa LINDA

Um programa LINDA possui duas partes que definem a especificação sintática e a especificação semântica da linguagem alvo. Para delimitar claramente estas partes definimos a seguinte estrutura:

Todo programa LINDA começa com a palavra reservada *Grammar* ou seu sinônimo em português *Gramatica*, seguido de um identificador e do delimitador dois pontos. Este identificador não terá outra finalidade senão a de documentar a gramática que está em desenvolvimento.

Em seguida, devemos dar início à especificação sintática da linguagem alvo usando a palavra reservada *Syntax* ou em português *Sintaxe*, e para separar esta parte da segunda usamos *Semantics*, ou em português *Semantica*.

O programa LINDA deve terminar com a palavra reservada *End* ou *Fim*.

III.4 - Especificação sintática da linguagem alvo

Esta parte compreende a declaração de símbolos e a descrição das produções da gramática concreta.

Na parte de especificação sintática podemos declarar dois tipos de símbolos: Não-terminais e Regras. A forma de declarar é usando a palavra reservada *Nonterminal* ou *Rule* respectivamente, seguida de uma lista de identificadores separados por vírgula e terminada por ponto e vírgula.

Os sinônimos em português são respectivamente: *Naoterminal* e *Regra.*

Exemplos:

```
Nonterminal  E, T, F;
Rule         Soma, Mult, Preced;
Naoterminal  Prog, Bloco, Decl, Cmd;
Regra        rBloco, rDecls, rDecl, rIf, rAtrib;
```

As produções da gramática concreta são formadas por símbolos não-terminais e terminais, sendo estes últimos representados por strings. Entre o símbolo não-terminal e a derivação do mesmo usamos o delimitador ::=.

À cada produção da gramática concreta podemos opcionalmente associar um símbolo do tipo *regra*, usando o delimitador => e um identificador previamente declarado. Este identificador será o elo de correspondência entre as gramáticas concreta e abstrata.

Para finalizar a produção devemos usar ponto e vírgula.

Para representar uma produção que reduz em epsilon, podemos recorrer a dois métodos: o primeiro é omitir a lista do lado direito da produção, e o segundo é usar a palavra reservada *epsilon.*

Exemplos:

```
E ::= E '+' T => rSoma;
E ::= '(' E ')';
Decls ::=        => rInicioLista;
Decls ::= epsilon => rInicioLista;
```

O compilador LINDA utiliza as informações contidas nesta parte do programa para:

- Registrar as produções na ordem em que aparecem e numerá-las. A numeração deve começar do número 1 (um) e ignorar as produções simples. Definimos como produção simples aquela em que o seu lado direito tem um e apenas um símbolo não-terminal. Este procedimento se torna necessário pelo fato de que o gerador R*S se utiliza deste critério para numerar as produções e gerar as tabelas de redução do analisador sintático. Esta numeração será utilizada pelo compilador para criar a correspondência entre as reduções do analisador sintático e as regras semânticas, definindo os símbolos do tipo *Regra* com as constantes correspondentes.

Exemplo:

```
E ::= E '+' T       => Soma;        [1]
E ::= T;
T ::= T '*' F       => Mult;        [2]
T ::= F;
F ::= '(' E ')'     => Preced;      [3]
F ::= 'ID'          => Ident;       [4]
```

Para o exemplo acima a numeração é a que está indicada entre colchetes do lado direito de cada produção. As segunda e quarta linhas do exemplo contém regras simples. A regra número 3 não é simples porque além do símbolo não-terminal há dois símbolos terminais. A definição de constantes emitida pelo compilador LINDA ficaria assim:

```
const
    Soma      = 1;
    Mult      = 2;
    Preced    = 3;
    Ident     = 4;
```

- Registrar a ocorrência dos símbolos terminais na ordem em que aparecem nas produções e numerá-los. A numeração deve começar do número 1 (um). O gerador R*S se utiliza do mesmo critério para gerar os códigos dos *tokens* da linguagem e assim preencher as tabelas do analisador sintático. Esta numeração será utilizada pelo compilador LINDA para gerar a tabela *Hash* do analisador léxico, e a tabela ASCII com os nomes dos terminais da linguagem alvo.

Para o mesmo exemplo teríamos a seguinte lista de símbolos terminais e correspondente numeração:

```
+       [1]
*       [2]
(       [3]
)       [4]
ID      [5]
```

- Gerar um arquivo texto com as produções sintáticas para posterior processamento pelo gerador R*S. Assim, deverá se agrupar as alternativas que tiverem o mesmo não-terminal do lado esquerdo através do uso do símbolo exclamação (ver manual do gerador R*S (RANGEL 1988)). Também a sintaxe de terminação da produção deverá ser observada (ponto e vírgula ao final da lista de derivações de cada não-terminal).

Concluindo o exemplo teríamos:

```
E = E '+' T
  ! T ;
T = T '*' F
  ! F ;
F = '(' E ')'
  ! 'ID' ;
```

III.5 - Especificação semântica da linguagem alvo

Nesta segunda parte do programa LINDA estão contidas as declarações de símbolos e seus atributos, e a descrição de cada regra semântica.

As declarações podem ser de: tipos, constantes, variáveis, funções, não-terminais e terminais.

A declaração de tipo é simplesmente uma lista de nomes com os quais o compilador irá verificar expressões aritméticas, passagens de parâmetros em funções e atribuições a variáveis. Estas verificações se restringem a comparar entre si os tipos dos operandos, comparar os tipos dos parâmetros reais com os tipos dos parâmetros formais, ou ainda os tipos resultantes de expressões aritméticas com os tipos das variáveis, respectivamente.

O tipo resultante de uma expressão aritmética será igual ao dos operandos para o caso de operadores aritméticos e lógicos, e igual ao tipo *boolean* quando o operador for relacional.

Existem tipos pré-definidos que o programa LINDA não precisa e não deve declarar, são eles: *Integer*, *Char*, *String* e *Boolean*.

É responsabilidade do projetista da linguagem alvo, incluir ao código gerado pelo compilador LINDA declarações completas em Pascal dos tipos definidos no programa LINDA, como já vimos, os tipos de um programa LINDA são apenas identificadores.

Para formar a declaração de tipo, utilizamos a palavra reservada *Type* ou em português *Tipo* e, em seguida, a lista de identificadores que desejamos declarar separados por vírgula. Para terminar a lista usamos ponto e vírgula.

Exemplos:

```
Type    tabsimb, instrucao;
Tipo    real;
```

A declaração de constantes e de variáveis, associa a um identificador um tipo. O valor em si da constante não é definido no programa LINDA. Apenas interessa ao compilador os nomes das constantes e seus respectivos tipos, cabendo ao projetista da linguagem alvo incluir ao código gerado pelo compilador LINDA declarações Pascal suplementares.

Para formar uma declaração de constantes ou de variáveis, utilizamos a palavra reservada *Const* ou *Var* respectivamente seguida de uma lista de identificadores separados por vírgula, terminando a lista com o delimitador dois pontos, um identificador previamente declarado como tipo e o caractere ponto e vírgula.

Exemplos:

```
Const   vazio : tabsimb;
Var     Topo, Limite : integer;
```

A declaração de função é similar a declaração de *Header* de função na programação Pascal. O compilador LINDA irá analisar o tipo que a função retorna no contexto da invocação da função, verificar se o número de parâmetros passados é igual ao número de parâmetros declarados, e se o tipo de cada parâmetro passado é igual ao respectivo tipo declarado. O compilador não fará crítica entre o uso de parâmetros passados por valor e por referência. O corpo da função deve ser escrito em Pascal e incluído ao código gerado pelo compilador LINDA (ver figura III.1). A palavra reservada *Function* pode ser substituida pelo seu sinônimo em português *Funcao*.

Exemplos:

```
funcao    junta (t1, t2 : ts) : ts;
function  pegasimb : string;
```

Associado a cada declaração de um símbolo, não-terminal ou terminal, poderemos ter um ou vários identificadores para representar os atributos deste símbolo. Cada um desses identificadores representa uma variável com seu tipo, e mais um atributo indicativo do modo de avaliação do símbolo. Este modo pode ser herdado ou sintetizado. O compilador LINDA transformará estas declarações em estruturas *Record* do Pascal. Portanto é permitido ao projetista da linguagem repetir os nomes dos atributos na declaração de outros símbolos.

Convém destacar que símbolos terminais que não tenham atributos não são declarados, pois os mesmos podem ser representados por *strings*.

A declaração de símbolo começa com a palavra reservada *Nonterminal* ou *Terminal* conforme o caso, mais uma lista de símbolos separados por vírgula e uma lista de atributos. Cada um dos símbolos possuirá a sua própria cópia da lista de atributos. Cada elemento da lista de atributos é formado pela palavra reservada indicativa do modo de avaliação, uma lista de identificadores separados por vírgula, o caractere dois pontos finalizando a lista, um identificador previamente declarado como tipo e o caractere ponto e vírgula.

O modo de avaliação dos atributos deve ser definido com o uso da palavra reservada *Inh* ou *Herd* se forem herdados ou, *Syn* ou *Sint* se forem sintetizados.

Exemplos:

```
Nonterminal E, T, F
      Inh tab : ts;
      Syn pos : integer;
      Syn cod : boolean;
Naoterminal S
      Herd a, b, c : simb;
      Sint x, y, z : param;
Terminal ELSE
      Herd marca : integer;
```

Como já vimos, a correspondência entre as gramáticas concreta e abstrata se dá por intermédio de um símbolo do tipo *Regra*. Entendemos como uma regra o conjunto de ações semânticas que deverão ser executadas cada vez que uma produção da gramática concreta é reduzida pelo analisador sintático. Após todas as declarações, o programador LINDA passa a descrever as regras semânticas.

Cada regra é composta de duas partes: especificação sintática e especificação semântica.

A especificação sintática de uma regra descreve a sintaxe da gramática abstrata. Escrevemos essa especificação do mesmo modo que se faz na primeira parte do programa LINDA. O único vínculo entre as duas descrições, é a necessidade de o número de símbolos não-terminais ser o mesmo em ambas. A razão disso é que a cada símbolo não-terminal encontrado na especificação sintática de uma regra, corresponde a construção de um nó na árvore sintática. Esta correspondência pode ser entendida analisando os dois lados da seguinte forma: o analisador sintático constrói uma sub-árvore de acordo com a gramática concreta, onde o número de filhos corresponde ao número de símbolos não-terminais da regra gramática usada. Por outro lado, o cálculo dos atributos é realizado a partir de uma base de indexação firmada pela posição relativa do símbolo não-terminal na descrição da gramática abstrata.

A especificação semântica de uma regra é composta de uma ou mais expressões de atribuição. Cada atribuição pode abranger variáveis

ordinárias, constantes, chamada de funções e variáveis com referências a atributos. As expressões podem ter operadores booleanos (And, Or e unário Not), aritméticos (Soma, Subtração, Multiplicação, Divisão, Resto de divisão e unário Menos) ou relacionais (Diferente, Igual, Maior, Menor, Maior ou igual e Menor ou igual), e parênteses para mudança de precedência.

Para formar as regras semânticas, usamos a palavra reservada *Rule* ou em português *Regra*, em seguida um identificador previamente declarado com tipo *Regra* e o delimitador =>. Seguem então a especificação sintática e a especificação semântica, e terminamos a regra com ponto e vírgula.

A especificação semântica pode assumir duas formas. A forma simples, onde todos os atributos herdados são copiados do lado esquerdo para o lado direito da produção e os atributos sintetizados copiados do lado direito para o lado esquerdo. Esta forma é especificada através da palavra reservada *Simple* ou, em português, *Simples.*

A segunda forma, que envolve expressões de atribuição, é uma lista que começa com a palavra reservada *Begin* (ou *Inicio*), e termina com *End* ou *Fim*. Cada expressão da lista é separada por um ponto e vírgula.

Cada símbolo não-terminal ou terminal pode ter um ou mais atributos. Quando o símbolo aparece numa expressão, devemos referenciar o atributo desejado escrevendo o caractere ! e o nome do atributo logo em seguida ao nome do símbolo. Este caractere foi preferido ao caractere '.', que é usado em Pascal, com sintaxe semelhante, mas com finalidade distinta.

Exemplos:

```
Rule rINT => decl ::= 'int' id
      begin
      decl!tsd := entrats (id!nome, decl!ts);
      end;
Regra preced => E ::= '(' E ')'
      Simples;
```

Sempre que ocorrer uma repetição de um símbolo dentro da

mesma especificação sintática da gramática abstrata, haverá a necessidade de diferenciar uma ocorrência da outra quando o símbolo aparece numa expressão. Sendo assim, o programador LINDA deve prover a especificação sintática de uma numeração. O critério adotado para numerar essas ocorrências é da esquerda para a direita iniciando-se a contagem com zero. Esta numeração é verificada pelo compilador e as eventuais discrepâncias serão assinaladas como erros. Para referenciar o símbolo escreveremos então o caractere # e um número inteiro logo após o nome do símbolo.

Exemplo:

```
Rule SOMA => E#0 ::= E#1 '+' E#2
     begin
     E#1!pos := E#0!pos;
     E#2!pos := E#0!pos;
     E#0!cod := E#1!cod and E#2!cod;
     E#0!tam := E#1!tam + E#2!tam + 1;
     end;
```

III.6 - Arquivos gerados pelo compilador LINDA

Como resposta da compilação de um programa LINDA, obteremos três arquivos:

Um arquivo com a sintaxe concreta para ser processado posteriormente pelo gerador R*S. O nome desse arquivo será o mesmo do programa fonte LINDA mais a extensão ".GRM".

Um arquivo com programa fonte em Pascal que contém duas tabelas ("tabpalres" e "palres") e uma constante inteira para cada símbolo do tipo regra. A tabela de *strings* de nome "tabpalres" contém os nomes de todos os terminais da linguagem alvo. Usando o código do *token* como índice da tabela, obtemos uma string com a imagem ASCII do mesmo. Esta tabela é útil durante a depuração dos programas. A tabela Hash "palres" contém os símbolos terminais da linguagem alvo que comecem com uma letra. Fica disponível ao projetista um esqueleto de analisador léxico escrito em Pascal onde se encontra uma função hash com o mesmo algoritmo usado na construção da tabela. As constantes serão utilizadas pelo avaliador de atributos para determinar qual o conjunto de regras semânticas que deve ser aplicado a um nó. O nome

desse arquivo terá a extensão ".HTB".

Um arquivo com duas funções escritas em Pascal, "Avaliador" e "Desce", contendo as instruções de cálculos de atributos que, junto com o analisador sintático gerado, formará o compilador da linguagem alvo. Ao terminar a análise do programa fonte, o avaliador de atributos é invocado recebendo como parâmetro a raiz da árvore sintática. O nome do arquivo terá a extensão ".AVL".

As declarações de símbolos terminais e não-terminais serão convertidas para declarações Pascal de *record type*, onde o nome do símbolo será o nome do *type* e cada atributo será um elemento do *record*.

Exemplo:

```
Nonterminal E :
    syn livre, addr : integer;
    inh forma : integer
```

```
Type E = record
    livre, addr : integer;
    forma : integer;
    end;
```

Antes de calcular os atributos de um nó e da sub-árvore diretamente abaixo dele, precisamos alocar espaço de memória para seus atributos. Essa memória será desalocada mais adiante quando terminarmos de calcular todos os atributos da mesma regra semântica. Usaremos para isso as funções Pascal *GetMem* e *FreeMem*.

Exemplos:

```
GetMem (p0, SizeOf(E))
GetMem (p1, SizeOF(T))
FreeMem (p1, SizeOf(T))
FreeMem (p0, SizeOf(E))
```

Cada acesso a um atributo será traduzido para um acesso por ponteiro, onde o nome do símbolo será usado como *typecasting* do ponteiro.

Exemplo:

```
E#0!livre := T!addr;
```

```
E(p0^).livre := T(p1^).addr
```

Nesse exemplo, o trecho E ( ) forma o *typecasting*, p0^ é um ponteiro para um registro e .livre acessa um campo dentro desse registro.

Para compreendermos melhor o esquema geral de alocação de memória para cálculo dos atributos, já discutido no capítulo II, apresentamos o seguinte exemplo:

```
Seja a regra semântica S ::= S1 S2
A alocação de memória será:
      Aloca memória para S1
      Calcula atributos herdados de S1
      Calcula a sub-árvore abaixo de S1 retornando os
                  atributos sintetizados de S1
      Aloca memória para S2
      Calcula atributos herdados de S2
      Calcula a sub-árvore abaixo de S2 retornando os
                  atributos sintetizados de S2
      Calcula os atributos sintetizados de S
      Desaloca memória para S2, S1
```

Os nomes das regras semânticas foram definidas como constantes para que o avaliador pudesse ser estruturado como um grande *case*. Cada regra semântica está toda contida em cada entrada desse *case* e o nome da regra servindo como chave de entrada.

Apresentamos agora um exemplo completo de tradução de regra semântica. Algumas observações foram colocadas como comentários.

Exemplo:

```
Regra SOMA => E#0 ::= E#1 '+' E#2
      Inicio
      E#1!pos := E#0!pos;
      E#2!pos := E#0!pos;
      E#0!cod := E#1!cod and E#2!cod;
      E#0!tam := E#!tam + E#2!tam + 1;
      Fim;
```

Sua tradução será:

```
SOMA:              { Os atributos de E#0 estão apontador por r }
      begin
      GetMem (p1, Sizeof(E));
      E(p1^).pos := E(r^).pos;
      Desce (p1);                    { Calcula sub-árvore e sintet.}
      GetMem (p2, Sizeof(E));
      E(p2^).pos := E(r^).pos;
      Desce (p2);                    { Calcula sub-árvore e sintet.}
      E(r^).cod := E(p1^).cod and E(p2^).cod;
      E(r^).tam := E(p1^).tam + E(p2^).tam + 1;
      FreeMem (p2, Sizeof(E));
      FreeMem (p1, Sizeof(E));
      end;
```

CAPÍTULO IV

IMPLEMENTAÇÃO DO COMPILADOR LINDA

IV.1 - Introdução

A linguagem Pascal é a mais usada no meio acadêmico, principalmente por ser bem estruturada, fácil de aprender e os programas serem mais claros. Trabalhos que antecederam a este, quando a linguagem LINDA teve sua origem, também foram feitos em Pascal. A própria linguagem LINDA é assemelhada ao Pascal. O objeto gerado pelo compilador LINDA é um programa Pascal. Por essas razões, naturalmente, o compilador LINDA foi inteiramente escrito em Pascal sendo usada a versão 5.0 da BORLAND (1988).

O compilador LINDA consiste de três blocos principais: analisadores léxico, sintático e semântico. A compilação de um programa LINDA, é dirigida pelo analisador sintático. À medida em que se vai processando a cadeia de entrada, o analisador sintático invoca o analisador léxico para que este devolva o próximo *token* da cadeia de entrada. Quando o analisador sintático executa uma ação de redução, o analisador semântico é invocado com o número da regra que foi usada.

Inicialmente realizamos uma coletânea de sub-rotinas de manuseio de tabelas e pilhas. Dividimos estas rotinas em duas bibliotecas para uso neste projeto, bem como para auxiliar o projetista na realização do compilador alvo, e colocamos estas bibliotecas em *units* do Pascal.

A primeira biblioteca se chama *GTABSIM* e possui rotinas para acessar tabela de símbolos e agrupar símbolos em buffers. O método de acesso escolhido foi o de função Hash com uma lista para cada colisão. A segunda se chama *GARVSEM* e possui rotinas para construção de árvores, manuseio de pilhas e movimentação dentro da árvore.

O método de análise sintática escolhido foi o R*S devido à facilidade de implementação das tabelas e simplicidade e eficiência do algoritmo. Também é um motivo, o fato de que o compilador alvo usará o mesmo método. Futuramente o compilador LINDA poderá ser reescrito na própria linguagem LINDA com menor esforço de programação.

A programação do analisador semântico foi desenvolvida utilizando a técnica de tradução dirigida pela sintaxe (AHO 1986 e BARRET 1979).

Passamos a descrever cada um dos itens citados acima.

IV.2 - Biblioteca de rotinas para tabela de símbolos

Na biblioteca *GTABSIM* estão contidas as seguintes funções:

i) *Hash* - Recebe a cadeia de caracteres que compõe o símbolo e devolve um número entre 0 e 255. Este número servirá como índice para uma tabela de símbolos.

ii) *InsereNaTS* - A partir do índice calculado pela função Hash, verifica se o símbolo que se deseja inserir já está presente na tabela. Se não estiver, procura uma posição livre para colocar o símbolo. Retorna a posição na tabela em que efetivamente ficou o símbolo (encontrado ou recém inserido).

No caso de conflito, ou seja, caso em que a função Hash retorna o mesmo índice para dois ou mais símbolos diferentes, é formada uma lista com estes símbolos e o início desta lista fica enganchada na tabela.

iii) *ExsiteNaTS* - Pesquisa a tabela de símbolos para verificar a existência ou não de um determinado símbolo.

iv) *PutNomes* - Guarda a cadeia de caracteres que forma o símbolo num buffer e retorna um ponteiro deste buffer posicionado no início da cadeia.

Esta função é usada pela InsereNaTS, mas também é de uso geral.

v) *GetNomes* - Retorna uma cadeia de caracteres a partir de um ponteiro.

Esta função é usada pela ExisteNaTS, mas também é de uso geral.

vi) *GetPtrNomes* - A partir de um índice da tabela de símbolos, esta função retorna, caso esta posição da tabela esteja ocupada por um símbolo, o ponteiro onde foi guardado a cadeia de caracteres deste símbolo.

vii) *UpStr* - Transforma uma cadeia de caracteres em outra cadeia em que os caracteres minúsculos são substituídos pelos correspondentes maiúsculos.

Esta função é usada por InsereNaTS e ExisteNaTS para verificar a equivalência entre dois símbolos, entretanto, o que fica armazenado no buffer é a forma original. Também é de uso geral.

viii) *ListaTS* - Esta função lista na console todo o conteúdo da tabela de símbolos. Sua utilização é para a depuração de programas.

IV.3 - Biblioteca de rotinas para árvores e pilhas

Na biblioteca *GARVSEM* estão contidas funções para manuseios de pilhas e árvores. As árvores são compostas de *nós*, onde cada nó é uma estrutura de dados definida pelo usuário da biblioteca tendo no mínimo os seguintes campos:

PRIM - ponteiro para um nó;
PROX - ponteiro para um nó;
ULTIMO - variável booleana;
COD - variável inteira.

As pilhas são implementadas por listas, onde cada elemento é um ponteiro para uma estrutura de dados do tipo nó.

São as seguintes as funções:

i) *PushSem* - Esta função coloca no topo da pilha um ponteiro para um nó.

ii) *PopSem* - Retira o ponteiro que estiver no topo da pilha e retorna.

iii) *CriaNo* - Esta função aloca memória do tamanho de um nó, inicializa o campo COD com o número recebido como parâmetro e inicializa os campos PRIM, PROX e ULTIMO com ponteiros nulos. Coloca no topo da pilha um ponteiro para o bloco de memória alocada.

iv) *BuildN* - Chama a rotina CriaNo e depois inicializa os campos PRIM e PROX formando uma lista com os nós que estiverem no topo da pilha. O tamanho desta lista é passado como parâmetro.

v) *UnOp* - Desempilha um nó e modifica os campos PRIM e PROX do nó que ficar no topo para apontar para o nó desempilhado.

vi) *BinOp* - Semelhante à função UnOp, porém são retirados três nós da pilha e o nó intermediário terá seus campos modificados e depois reempilhado.

vii) *Merge* - Forma uma lista com os dois nós que estiverem no topo da pilha.

viii) *Pai* - Retorna um ponteiro para o pai de um nó.

ix) *MudaCod* - Modifica o campo COD do nó que estiver no topo da pilha.

x) *ListaNo* - Lista o conteúdo dos campos de um nó na console. Sua utilização é para a depuração de programas.

xi) *ListaAll* - Lista o conteúdo dos campos de um nó, e de seus filhos, recursivamente, até as folhas das árvores. Sua utilização também é para a depuração de programas.

IV.4 - Implementação do analisador léxico

Quando solicitado, o analisador léxico retira caracteres do arquivo fonte até formar um *token* da linguagem LINDA.

Durante este processo de leitura, os delimitadores de comentários são reconhecidos e processados. Um contador é incrementado a cada linha lida para que as demais rotinas possam informar a localização de um erro.

Ao encontrar um identificador, o analisador léxico verifica se ele não é uma das palavras reservadas da linguagem consultando uma tabela Hash.

Se o analisador encontrar o final de arquivo de maneira inesperada, seja antes de fechar um comentário ou antes de fechar uma *string*, será emitida uma mensagem de erro apropriada. Também ao encontrar um caractere de controle ASCII um erro será assinalado. A seguir apresentamos as mensagens de erro do analisador léxico:

"Erro léxico na linha: <n> / <c> Comentário não fechado"
"Erro léxico na linha: <n> / <c> String não fechada"
"Erro léxico na linha: <n> / <c> Caractere Inesperado"

onde <n> e <c> são o número da linha e da coluna respectivamente, onde o erro foi encontrado.

IV.5 - Implementação do analisador sintático

O método de análise sintática adotada foi o R*S, sendo usado o gerador R*S (RANGEL 1988) para a construção das tabelas.

Partimos de um algoritmo padrão, onde o analisador léxico é invocado para buscar o próximo *token* da cadeia de entrada. O mesmo acontece com o analisador semântico nas ações de redução.

Se em função da cadeia de entrada, do estado interno do analisador sintático e das tabelas de redução o algoritmo chegar a um ponto em que não for possível uma ação de redução nem de empilhamento, então teremos chegado a um estado de erro. Neste caso a seguinte mensagem será emitida:

"Erro sintático: <q> / <p> Linha: <n> / <c>"

onde <q> e <p> mostram o estado interno da pilha e o símbolo que ocasionou o erro (ver manual R*S (RANGEL 1988)), e <n> e <c> indicam a linha e coluna em que estava o processamento quando o erro foi detectado.

IV.6 - Implementação do analisador semântico

Com a ajuda das bibliotecas GTABSIM e GARVSEM, foi possível construir e manter uma tabela de símbolos com todas as características necessárias para implementar a linguagem LINDA.

O analisador semântico determina um conjunto de operações para cada produção da linguagem. Quando o analisador sintático reduz uma determinada produção, as operações associadas a esta produção são executadas. Por exemplo, quando o analisador sintático reduz uma declaração de símbolo, o analisador semântico o inclui na sua tabela de símbolos.

A maior parte dessas operações envolve verificações e testes de condições. Sempre que uma discrepância for encontrada uma mensagem de erro é emitida. A estrutura geral das mensagens é a seguinte:

Erro semântico <e> Linha:<n> / <c> "<token>" <texto>

onde <e> é o número do erro semântico, <n> e <c> são a linha e coluna, <token> é o último símbolo analisado e <texto> é uma mensagem explicativa do erro.

As seguintes mensagens foram redigidas buscando sempre trazer as informações do contexto onde ocorreu o erro:

1 - Esperado símbolo do tipo "Não-Terminal"
2 - Esperado símbolo do tipo "Regra"
3 - Símbolo não declarado
4 - Símbolo já foi declarado
5 - Produção "simples" não pode ter regra associada
6 - Esperado símbolo do tipo "Type"
7 - Número inteiro inválido
8 - Tipos incompatíveis
9 - Chamada de função com menos parâmetros que o declarado
10 - Chamada de função com mais parâmetros que o declarado
11 - Utilização inválida de atributo
12 - Atributo não pertence ao identificador
13 - Numeração de ocorrência de símbolo inconsistente
14 - Esperado símbolo do tipo Não-terminal ou Terminal
15 - Construção inválida para símbolo tipo Terminal
16 - Utilização inválida de número de ocorrência
17 - Número de ocorrência inconsistente com o declarado
18 - Símbolo não pertence a esta regra semântica
19 - Utilização inválida de atributo sintetizado
20 - Utilização inválida de atributo herdado
21 - Expressão referencia atributo herdado de simb. que está à direita do simb. da variável
22 - Expressão referencia atributo sintetizado do simb. que a produção reduz
23 - Expressão referencia atributo sintetizado do próprio símbolo ou de símbolo que está à direita do simb. da variável
24 - Ordem de expressão não obedece critério de Bochmann
25 - Símbolo do tipo regra não está referenciado a uma produção
26 - Regra não pode ser "simples"

CAPÍTULO V

EXEMPLO DE CONSTRUÇÃO DE UM COMPILADOR

V.1 - Linguagem "G"

Desejamos projetar uma linguagem de programação, a qual chamaremos de G. A linguagem G conterá as características mais comuns de todas as linguagens de programação: declaração de variáveis, expressões aritméticas, comando condicional, comando de repetição e aninhamento de blocos.

Com este exemplo abordaremos os principais problemas na construção de compiladores. As técnicas aqui mostradas podem ser aplicadas na maioria das linguagens práticas onde os problemas reais e mais complexos são resolvidos com variações destas técnicas.

Na linguagem G as variáveis poderão ser de dois diferentes tipos: inteiro ou caractere. Os operadores serão: soma, subtração, multiplicação e divisão. As constantes serão numéricas ou literais. Os comandos serão: atribuição, if, if com cláusula else e while-do.

Na figura V.1 mostramos a primeira parte do programa LINDA onde está descrita a sintaxe da linguagem.

```
gramatica g :

{=============================================================================}
{       Gramatica exemplo "G"

  Esta gramatica define uma linguagem de programacao estruturada onde
  ha' declaracoes de variaveis, comandos, expressoes aritimeticas,
  e blocos identados.
}
{=============================================================================}

sintaxe
   naoterminal  Prog, Bloco, Decls, Cmds, Decl, Cmd,
                Expr, Term, Fat, Opad, Opmult, V, Cte;

   regra        rProg, rBloco, rBlocoCmd, rDecls, rCmds,
                rInt, rChar, rAtrib, rWhile, rIf, rIfelse,
                rAdicao, rSoma, rSub, rFatoracao, rMult, rDiv,
                rPreced, rCteNum, rCteLit, rVar, rId;

      Prog    ::= Bloco '.'                                  => rProg;
      Bloco   ::= 'begin' Decls ';' Cmds 'end'               => rBloco;
      Bloco   ::= 'begin' Cmds 'end'                         => rBlocoCmd;
      Decls   ::= Decls ';' Decl                             => rDecls;
      Decls   ::= Decl;
      Cmds    ::= Cmds ';' Cmd                               => rCmds;
      Cmds    ::= Cmd ;

      Decl    ::= 'int' '_id'                                => rInt;
      Decl    ::= 'char' '_id'                               => rChar;

      Cmd     ::= Bloco ;
      Cmd     ::= V ':=' Expr                                => rAtrib;
      Cmd     ::= 'if' Expr 'then' Cmd 'fi'                  => rIf;
      Cmd     ::= 'if' Expr 'then' Cmd 'else' Cmd 'fi'       => rIfelse;
      Cmd     ::= 'while' Expr 'do' Cmd                      => rWhile;

      Expr    ::= Expr Opad Term                             => rAdicao;
      Expr    ::= Term ;
      Opad    ::= '+'                                        => rSoma;
      Opad    ::= '-'                                        => rSub;
      Term    ::= Term Opmult Fat                            => rFatoracao;
      Term    ::= Fat ;
      Opmult  ::= '*'                                        => rMult;
      Opmult  ::= '/'                                        => rDiv;
      Fat     ::= '(' Expr ')'                               => rPreced;
      Fat     ::= Cte;
      Fat     ::= '_id'                                      => rId;
      Cte     ::= '_num'                                     => rCteNum;
      Cte     ::= '_lit'                                     => rCteLit;
      V       ::= '_id'                                      => rVar;
```

Figura V.1 - Especificação sintática da linguagem G

V.2 - Funções desejáveis e atributos

Vamos enumerar as funções semânticas as quais desejamos que o compilador da linguagem G execute, os atributos necessários para tanto e como eles agem. As regras semânticas que envolvem estes atributos serão discutidas na seção seguinte.

1 - Reconhecimento dos identificadores através do manuseio de uma tabela de símbolos, inserção e remoção de identificadores e controle do escopo das variáveis.

Sendo G uma linguagem estruturada desejamos que as variáveis só sejam definidas dentro do bloco em que foram declaradas. O controle do escopo deve prever a redefinição temporária das variáveis por outras declaradas em blocos mais internos. Neste caso, a variável retomará a definição original ao terminar a compilação do bloco mais interno.

Formaremos a tabela de símbolos como sendo uma lista (atributo herdado ts). Assim, quando entramos num bloco mais interno a lista de declarações deste bloco (atributo sintetizado tsd) é encadeada ao final da lista mais externa, e quando terminamos o bloco retiramos esta lista.

Para que o esquema de redefinição temporária das variáveis possa funcionar, é imprescendível que a pesquisa do símbolo seja feita do final da lista para o ínicio. A primeira ocorrência do símbolo será então a que deve prevalecer no momento.

Para controle do tamanho de cada lista de declarações temos um atributo sintetizado, tam, que é calculado durante a formação da lista da tabela de símbolos.

2 - Controle de alocação das variáveis.

Considerando o exposto no item anterior, decidimos que as variáveis serão alocadas dinamicamente no início do bloco onde foram declaradas e desalocadas ao final do mesmo bloco.

Para controle do espaço alocado às variáveis, usaremos um atributo herdado, pos, que guarda o próximo endereço livre de memória. Juntamente com o atributo tam podemos determinar o valor que será passado para o bloco mais interno.

3 - Controle de alocação de variáveis temporárias.

Durante o cálculo de expressões aritméticas, em algumas situações, temos que guardar um resultado intermediário para ser usado mais tarde. O uso otimizado do espaço de memória para esta finalidade também é realizado pelo atributo pos, o que fará com que as váriaveis declaradas e as variáveis temporárias fiquem agrupadas no mesmo segmento de memória.

4 - Controle dos tipos das variáveis.

Diferentes tipos podem requerer diferentes tamanhos de memória e quando misturados em expressões aritméticas é comum a necessidade de conversão de tipos. Neste exemplo, limitamo-nos a emitir uma mensagem de erro quando os tipos são diferentes. O atributo sintetizado format nos dará esta informação através do número de bytes ocupados pela variável.

5 - Geração de código objeto e expressões aritméticas. Determinação de endereços e instruções de código objeto e seus diferentes tamanhos.

O atributo herdado ini aponta para a primeira posição livre do segmento de memória reservado para as instruções. Poderia ser feito um controle do tamanho de instruções do mesmo modo que fizemos com as variáveis, mas como nossa máquina é hipotética, preferimos simplificar o exemplo fazendo a suposição de que todas as instruções possuam o tamanho de um byte.

Para geração do código das expressões aritméticas, necessitamos do uso do atributo sintetizado ender, com a finalidade de transportar o endereço do resultado parcial de uma expressão para ser usado em um nível de cálculo mais alto. Na prática, este atributo irá

guardar o endereço das variáveis temporárias, quando estas forem necessárias.

Para determinar qual a instrução aritmética a ser usada, temos o atributo sintetizado opcode para transportar esta informação até o momento da geração do código objeto.

6 - Instruções de salto para a frente.

Para gerarmos uma instrução de salto cujo endereço não conhecemos, utilizaremos o seguinte recurso técnico:

Criamos um atributo herdado chamado marca, que terá a localização da instrução de salto. Esta instrução será gerada com o endereço destino em branco. Quando conseguirmos determinar o endereço destino com exatidão, voltamos ao endereço guardado em **marca** e então geramos a instrução de salto completa.

7 - Controle de erro semântico.

Neste exemplo reduzimos o tratamento de erros para um simples aborto da geração de código objeto.

O atributo sintetizado cod é uma variável booleana cujo valor inicial *true*, é transportado por toda a árvore. No caso de haver um erro, a função que o encontrar torna a variável *false* inibindo as demais funções de geração de código.

V.3 - Programa LINDA

Na figura V.2 apresentamos os símbolos da linguagem **G** e os seus respectivos atributos. O tipo tabsimb deve ser definido em Pascal, externamente ao programa LINDA, e posteriormente adicionado ao avaliador de atributos gerado pelo compilador LINDA.

```
naoterminal Prog:
    sint cod      : boolean;    { Indica a presenca de erro ou nao          }
naoterminal Decl:
    herd pos      : integer;    { Endereco livre para alocar variaveis      }
    herd ts       : tabsimb;    { Tabela de simbolos validos                }
    sint tsd      : tabsimb;    { Tabela de simbolos declarada              }
    sint tam      : integer;    { Tamanho do espaco usado na declaracao     }
naoterminal Cmd:
    herd ini      : integer;    { Endereco da primeira instrucao do cmd     }
    herd ts       : tabsimb;    { Tabela de simbolos validos                }
    herd pos      : integer;    { Endereco livre para alocar variaveis      }
    sint cod      : boolean;    { Indica a presenca de erro ou nao          }
    sint tam      : integer;    { Tamanho do espaco usado na declaracao     }
naoterminal Expr:
    herd ini      : integer;    { Endereco da primeira instrucao do cmd     }
    herd ts       : tabsimb;    { Tabela de simbolos validos                }
    herd pos      : integer;    { Endereco livre para alocar variaveis      }
    sint cod      : boolean;    { Indica a presenca de erro ou nao          }
    sint tam      : integer;    { Tamanho do espaco usado na declaracao     }
    sint formt    : integer;    { Formato do resultado da expressao         }
    sint ender    : integer;    { Endereco do resultado da expressao        }
naoterminal V:
    herd ts       : tabsimb;    { Tabela de simbolos validos                }
    sint formt    : integer;    { Formato do resultado da expressao         }
    sint ender    : integer;    { Endereco do resultado da expressao        }
naoterminal Opad, Opmult:
    sint opcod    : integer;    { Tipo de operacao a ser executada          }
terminal Id, Num, Lit:
    herd nome     : string;     { String proveniente do anal. lexico        }
terminal xthen, xelse, xdo:
    herd marca    : integer;    { Guarda um endereco para backpatching      }
```

Figura V.2 - Declaração de símbolos e atributos

```
regra rProg        => Prog ::= Cmd
    inicio
    Cmd!ini := 0;
    Cmd!ts  := vazio;
    Cmd!pos := 0;
    Prog!cod := Cmd!cod;
    fim;
```

Figura V.3 - Regra semântica para símbolo inicial

Na figura V.3 vemos a regra semântica do símbolo inicial prog. Observamos que aqui estão a inicialização da tabela de símbolos e os segmentos de memória de dados e de código.

Na figura V.4 podemos ver o emprego dos atributos ts e tsd formando a regra de escopo das variáveis. A função juntats é responsável por unir duas listas de símbolos em uma única e ao modificarmos a variável toposimb, estamos deletando a última lista de símbolos. As demais operações são triviais.

Na figura V.5 encontramos as declarações de variáveis. A função pegasimb lê uma string de um arquivo intermediário gerado pelo analisador léxico. No exemplo deste capítulo, o analisador léxico grava os nomes, as constantes literais e as constantes numéricas. Neste caso, a *string* contém o nome do identificador que está sendo declarado. A função entrats pesquisa a lista de símbolos para verificar se este já foi declarado no presente bloco, caso contrário acrescenta-o no final da lista e retorna a lista inteira.

Na figura V.6 observamos o uso das funções geraatrib e geracond cujo objetivo é simplesmente emitir uma sequência de instruções, LOAD / STORE e JUMP <condição>, respectivamente. As instruções são guardadas sequencialmente, a partir do endereço dado pelo atributo ini. Os operadores das instruções e a condição a ser usada na instrução de salto são passadas como parâmetros. Também observamos a função backpatch que retrocede na sequência de instruções geradas para modificar o endereço destino de uma instrução de salto. O ponto para qual deve retroceder e o endereço a ser colocado são passados como parâmetro.

A figura V.7 traz as regras semânticas de formação das expressões aritméticas. Observe o uso dos atributos pos para armazenar as variáveis temporárias. A função pegatam retorna a informação de quantos bytes foram usados no armazenamento de temporárias. A função geraoper emite uma sequência de três instruções LOAD / <oper> / STORE, onde **<oper>** se refere a uma das quatro operações aritméticas.

```
regra rBloco        => Cmd#0 ::= 'begin' decl ';' Cmd#1 'end'
    inicio
    Decl!pos  := Cmd#0!pos;
    Decl!ts   := Cmd#0!ts;
    Cmd#1!ini := Cmd#0!ini;
    Cmd#1!ts  := juntats (Cmd#0!ts, decl!tsd);
    Cmd#1!pos := Cmd#0!pos + Decl!tam;
    Cmd#0!cod := Cmd#1!cod ;
    Cmd#0!tam := Cmd#1!tam ;
    TopoSimb  := Cmd#0!ts;
    fim;
regra rBlocoCmd  => Cmd#0 ::= 'begin' Cmd#1 'end'
    simples;
regra rDecls        => decl#0 ::= decl#1 ';' decl#2
    inicio
    Decl#1!pos := Decl#0!pos;
    Decl#1!ts  := Decl#0!ts;
    Decl#2!pos := Decl#0!pos + Decl#1!tam;
    Decl#2!ts  := Decl#0!ts;
    Decl#0!tam := Decl#1!tam + Decl#2!tam;
    Decl#0!tsd := juntats (decl#1!tsd, decl#2!tsd);
    fim;
regra rCmds         => Cmd#0 ::= Cmd#1 ';' Cmd#2
    inicio
    Cmd#1!ini := Cmd#0!ini;
    Cmd#1!ts  := Cmd#0!ts;
    Cmd#1!pos := Cmd#0!pos;
    Cmd#2!ini := Cmd#1!ini + Cmd#1!tam;
    Cmd#2!ts  := Cmd#0!ts;
    Cmd#2!pos := Cmd#0!pos;
    Cmd#0!cod := Cmd#1!cod and Cmd#2!cod;
    Cmd#0!tam := Cmd#1!tam + Cmd#2!tam;
    fim;
```

Figura V.4 - Formação da estrutura em blocos

```
regra rInt          => decl ::= 'int' id
    inicio
    Id!nome := pegasimb;
    Decl!tsd := entrats (id!nome, tipoint, Decl!pos, Decl!ts);
    Decl!tam := 2;
    fim;
regra rChar         => decl ::= 'char' Id
    inicio
    Id!nome := pegasimb;
    Decl!tsd := entrats (Id!nome, tipochar, Decl!pos, Decl!ts);
    Decl!tam := 1;
    fim;
```

Figura V.5 - Construção da tabela de símbolos

```
regra rAtrib      => Cmd ::= V ':=' Expr
   inicio
   V!ts      := Cmd!ts;
   Expr!pos  := Cmd!pos;
   Expr!ts   := Cmd!ts;
   Expr!ini  := Cmd!ini;
   Cmd!cod   := geraatrib (Expr!cod, V!formt, Expr!formt,
                           Expr!ini+Expr!tam, V!ender, Expr!ender);
   Cmd!tam   := Expr!tam + 2; { load + store }
   fim;
regra rIf         => Cmd#0 ::= 'if' Expr xthen Cmd#1 'fi'
   inicio
   Expr!ini  := Cmd#0!ini;
   Expr!pos  := Cmd#0!pos;
   Expr!ts   := Cmd#0!ts;
   xthen!marca := geracond (Expr!ini+Expr!tam, Expr!ender, Condfalse);
   Cmd#1!ini := Expr!ini + Expr!tam + 2;  { tamanho do jump    }
   Cmd#1!ts  := Cmd#0!ts;
   Cmd#1!pos := Cmd#0!pos + pegatam(Expr!formt);
   Cmd#0!cod := backpatch (xthen!marca, Cmd#1!ini+Cmd#1!tam)
                and Expr!cod and Cmd#1!cod;
   Cmd#0!tam := Expr!tam + Cmd#1!tam + 2; { jump condicional }
   fim;
regra rIfelse     => Cmd#0 ::= 'if' Expr xthen Cmd#1 xelse Cmd#2 'fi'
   inicio
   Expr!ini  := Cmd#0!ini;
   Expr!pos  := Cmd#0!pos;
   Expr!ts   := Cmd#0!ts;
   xthen!marca := geracond (Expr!ini+Expr!tam, Expr!ender, Condfalse);
   Cmd#1!ini := Expr!ini + Expr!tam + 2;  { tamanho do load + jump      }
   Cmd#1!ts  := Cmd#0!ts;
   Cmd#1!pos := Cmd#0!pos + pegatam(Expr!formt);
   xelse!marca := geracond (Cmd#1!ini+Cmd#1!tam, 0, Incond);
   Cmd#2!ini := Cmd#1!ini + Cmd#1!tam + 1; { tamanho do jump incond.   }
   Cmd#2!ts  := Cmd#0!ts;
   Cmd#2!pos := Cmd#0!pos + pegatam(Expr!formt);
   Cmd#0!cod := backpatch (xthen!marca, Cmd#2!ini) and backpatch (xelse!
    marca, Cmd#2!ini+Cmd#2!tam) and Expr!cod and Cmd#1!cod and Cmd#2!cod;
   Cmd#0!tam := Expr!tam + Cmd#1!tam + Cmd#2!tam + 3; {load + 2 * Jump }
   fim;
regra rWhile      => Cmd#0 ::= 'while' Expr xdo Cmd#1 'end'
   inicio
   Expr!ini  := Cmd#0!ini;
   Expr!pos  := Cmd#0!pos;
   Expr!ts   := Cmd#0!ts;
   xdo!marca := geracond (Expr!ini+Expr!tam, Expr!ender, Condfalse);
   Cmd#1!ini := Expr!ini + Expr!tam + 2;  { tamanho do load + jump      }
   Cmd#1!ts  := Cmd#0!ts;
   Cmd#1!pos := Cmd#0!pos + pegatam(Expr!formt);
   Cmd#0!cod := backpatch (geracond (Cmd#1!ini+Cmd#1!tam, 0, Incond),
                Expr!ini) and backpatch (xdo!marca, Cmd#1!ini+Cmd#1!tam+1)
                and Expr!cod and Cmd#1!cod;
   Cmd#0!tam := Expr!tam + Cmd#1!tam + 3; {load + 2 * Jump }
   fim;
```

Figura V.6 - Comandos da linguagem G

```
regra rAdicao      => Expr#0 ::= Expr#1 Opad Expr#2
   inicio
   Expr#1!ini   := Expr#0!ini;
   Expr#1!pos   := Expr#0!pos;
   Expr#1!ts    := Expr#0!ts;
   Expr#2!ini   := Expr#1!ini + Expr#1!tam;
   Expr#2!pos   := Expr#0!pos + pegatam(Expr#1!formt);
   Expr#2!ts    := Expr#0!ts;
   Expr#0!ender:= Expr#0!pos;
   Expr#0!cod   := geraoper (Expr#1!cod and Expr#2!cod, Expr#1!formt,
                  Expr#2!formt, Expr#2!ini + Expr#2!tam,
                  Expr#1!ender, Opad!opcod, Expr#2!ender, Expr#0!ender);
   Expr#0!formt:= Expr#1!formt;
   Expr#0!tam   := Expr#1!tam + Expr#2!tam + 3;
   fim;
regra rFatoracao   => Expr#0 ::= Expr#1 Opmult Expr#2
   inicio
   Expr#1!ini   := Expr#0!ini;
   Expr#1!pos   := Expr#0!pos;
   Expr#1!ts    := Expr#0!ts;
   Expr#2!ini   := Expr#1!ini + Expr#1!tam;
   Expr#2!pos   := Expr#0!pos + pegatam(Expr#1!formt);
   Expr#2!ts    := Expr#0!ts;
   Expr#0!ender:= Expr#0!pos;
   Expr#0!cod := geraoper (Expr#1!cod and Expr#2!cod, Expr#1!formt,
                 Expr#2!formt, Expr#2!ini + Expr#2!tam, Expr#1!ender,
                 Opmult!opcod, Expr#2!ender, Expr#0!ender);
   Expr#0!formt:= Expr#1!formt;
   Expr#0!tam   := Expr#1!tam + Expr#2!tam + 3;
   fim;
regra rPreced      => Expr#0 ::= '(' Expr#1 ')'
   simples;
```

Figura V.7 - Expressões aritméticas

```
regra rCteNum     => Expr ::= Num
   inicio
   Num!nome := pegasimb;
   Expr!cod := true;
   Expr!tam := 0;
   Expr!formt := tipoint;
   Expr!ender := encadeia (Num!nome, tipoint);
   fim;
regra rCteLit      => Expr ::= Lit
   inicio
   Lit!nome := pegasimb;
   Expr!cod := true;
   Expr!tam := 0;
   Expr!formt := tipochar;
   Expr!ender := encadeia (Lit!nome, tipochar);
   fim;
```

Figura V.8 - Tratamento de constantes

Na figura V.8 temos o tratamento de números e constantes literais. Após retirar o símbolo do arquivo intermediário com a função pegasimb, o nome é processado pela função encadeia. Essa função é responsável por manter um segmento de memória com as constantes, convertendo as *strings* para um formato aceito pela nossa hipotética máquina, guardando estas constantes e retornando a sua localização.

Na figura V.9 encontramos o tratamento de identificadores. A função peganats recebe como parâmetro uma string, e pesquisa a existência dessa string na tabela de símbolos. Duas informações são retornadas: o endereço da variável e o seu tipo.

A figura V.10 completa o conjunto de regras semânticas deste exemplo.

Na figura V.11 temos um exemplo de programa escrito na linguagem **G**, e na figura V.12 o resultado da sua compilação.

```
regra rId          => Expr ::= Id
    inicio
    Id!nome := pegasimb;
    Expr!cod := true;
    Expr!tam := 0;
    Expr!ender := peganats(Id!nome, Expr!formt);
    fim;
regra rVar         => V ::= Id
    inicio
    Id!nome := pegasimb;
    V!ender := peganats (Id!nome, V!formt);
    fim;
```

Figura V.9 - Introdução de identificadores na tabela de símbolos

```
regra rSoma       => Opad ::= '+'
    inicio
    Opad!opcod := OperSoma;
    fim;
regra rSub        => Opad ::= '-'
    inicio
    Opad!opcod := OperSub;
    fim;
regra rMult       => Opmult ::= '*'
    inicio
    Opmult!opcod := OperMult;
    fim;
regra rDiv        => Opmult ::= '/'
    inicio
    Opmult!opcod := OperDiv;
    fim;
```

Figura V.10 - Símbolos terminais da linguagem G

```
[1]  {=========================================}
[2]  { Programa de teste escrito na linguagem G
[3]    Tese de mestrado da COPPE 25/out/89 }
[4]  {=========================================}
[5]
[6]  begin
[7]      int I;
[8]      char C;
[9]      int J;
[10]     int K;
[11]
[12] I := J + K;
[13] J := I * (J + K + I * (K + I)) + I;
[14] C := 'a';
[15] while C do
[16]      begin
[17]      int I;
[18]      char Local;
[19]      I := K;
[20]      Local := 'z';
[21]      if I
[22]      then K := J
[23]      else K := I fi
[24]      end;
[25] I := K
[26] end.
```

Figura V.11 - Exemplo de um program escrito em linguagem G

```
[ 0] Load     3    { - J}
[ 1] Add      5    { - K}
[ 2] Store    7    { T1 ← J + K          (linha 12)}
[ 3] Load     7    { - T1}
[ 4] Store    0    { I ← J + K           (linha 12)}
[ 5] Load     3    { - J}
[ 6] Add      5    { - K}
[ 7] Store    9    { T2 ← J + K          (linha 13)}
[ 8] Load     5    { - K}
[ 9] Add      0    { - I}
[10] Store    13   { T4 ← K + I          (linha 13)}
[11] Load     0    { - I}
[12] Mult     13   { - T4}
[13] Store    11   { T3 ← I * (K + I)    (linha 13)}
[14] Load     9    { - T2}
[15] Add      11   { - T3}
[16] Store    9    { T2 ← J + K + I*(K + I)}
[17] Load     0    { - I}
[18] Mult     9    { - T2}
[19] Store    7    { T1 ← I*(J + K + I*(K + I))}
[20] Load     7    { - T1}
[21] Add      0    { - I}
[22] Store    7    { T1 ← I*(J + K + I*(K + I)) + I}
[23] Load     7    { - T1}
[24] Store    3    { J ← T1}
[25] Load     0    { segmento de constantes 'a'}
[26] Store    2    { C ← 'a'             (linha 14)}
[27] Load     2    { - C}
[28] Jzero    41   {                     (linha 15)}
[29] Load     5    { - K}
[30] Store    8    { I ← K               (linha 19)}
[31] Load     1    { segmento de constantes 'z'}
[32] Store    10   { Local ← 'z'         (linha 20)}
[33] Load     8    { - I}
[34] Jzero    38   {                     (linha 21)}
[35] Load     3    { - J}
[36] Store    5    { K ← J               (linha 22)}
[37] Jump     40
[38] Load     8    { - I}
[39] Store    5    { K ← I               (linha 23)}
[40] Jump     27   { volta para controle do while}
[41] Load     5    { - K}
[42] Store    0    { I ← K               (linha 25)}
[43] Stop          {                     (linha 26)}

Segmento de constantes:
[ 0] a     tam:1
[ 1] z     tam:1
```

Figura V.12 - Saída do compilador G

V.4 - Analisador léxico

O analisador léxico do compilador da linguagem alvo deve ser suprido pelo projetista da linguagem, devendo ser uma subrotina que busque um *token* da linguagem alvo no arquivo fonte e retorne o código deste *token*.

Para a construção do analisador léxico da linguagem G, partimos de um esqueleto escrito em Pascal onde se encontra algumas rotinas de manuseio de *buffer* e uma função Hash. Adaptando este esqueleto conforme características da linguagem G, tais como delimitadores, formação de números e identificadores, delimitadores de comentários, etc., e acrescentando as informações geradas pelo compilador LINDA formamos então o analisador léxico.

Estas informações estão listadas nas seguintes figuras:

A figura V.13 mostra a tabela Hash com os símbolos terminais da linguagem G que comecem com uma letra. São eles: *begin*, *end*, *int*, *char*, *if*, *then*, *fi*, *else*, *while* e *do*.

Devemos observar que os terminais *_id*, *_num* e *_lit* começam com o símbolo "_" propositadamente, para não estarem presentes nesta tabela. Cada um deles representa uma classe de terminais que é devidamente tratada pelo analisador léxico (identificadores, números inteiros e constantes literais, respectivamente) onde recebem um único código para cada uma das classes.

Também nesta figura se encontra em comentário, o tamanho máximo da sequência de colisões na tabela Hash. Para evitar que em cada pesquisa na tabela tenhamos que esgotar todas as possibilidades, é fornecida ao projetista esta constante, que lhe permite antecipar o final da pesquisa quando o símbolo não é um terminal da linguagem. No nosso caso esta constante é 1, significando que todos os símbolos terminais podem ser identificados na primeira tentativa.

A figura V.14 mostra a tabela de *strings* com o nome de todos os símbolos terminais da linguagem G. Usando o código do *token* como índice desta tabela, obtemos uma *string* com a imagem ASCII do mesmo. Esta tabela é útil durante a depuração do programa.

```
{ ATENCAO, Lookup maximo = 1 }
const palres : array [byte] of byte =
    (255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
     255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
     255,  12, 255, 255, 255, 255, 255, 255, 255, 255,
     255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
     255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
     255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
     255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
     255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
     255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
     255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
     255, 255, 255, 255, 255, 255, 255, 255,   9, 255,
     255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
     255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
     255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
     255, 255, 255, 255,   2, 255,  13, 255, 255, 255,
     255, 255, 255, 255, 255, 255, 255, 255, 255,  10,
      11, 255, 255, 255, 255, 255, 255, 255, 255, 255,
     255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
     255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
     255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
      14, 255, 255, 255, 255,   6, 255, 255, 255, 255,
     255, 255,  21,   4, 255, 255, 255, 255, 255, 255,
     255, 255,   7, 255, 255,   5, 255, 255, 255, 255,
     255, 255, 255, 255,  22, 255, 255, 255, 255, 255,
     255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
     255, 255, 255, 255, 255, 255);
```

Figura V.13 - Tabela Hash

```
const tabpalres : array [1..22] of string [11] =
        ('.',        'BEGIN',   ';',       'END',
         'INT',      '_ID',     'CHAR',    ':=',
         'IF',       'THEN',    'FI',      'ELSE',
         'WHILE',    'DO',      '+',       '-',
         '*',        '/',       '(',       ')',
         '_NUM',     '_LIT');
```

Figura V.14 - Tabela de nomes dos símbolos terminais

V.5 - Analisador sintático

Formado por rotinas padrão para o método R*S o analisador sintático incorpora também: rotinas para construção da árvore sintática na memória, o avaliador de atributos gerado pelo compilador LINDA e as funções que descrevemos no item V.3

O compilador da linguagem G é então composto pelo código objeto do analisador léxico, do analisador sintático e da tabela emitida pelo gerador R*S.

V.6 - Funções complementares

Para implementar a tabela de símbolos, utilizamos um arranjo onde cada elemento possui todas as informações sobre uma variável. À medida que queremos acrescentar símbolos, vamos empilhando estes símbolos no arranjo. A variável toposimb marca o final da lista (figura V.15).

Na figura V.16 a função entrats pesquisa a existência de um símbolo na lista. Se não existir, acrescenta-o no topo da lista. A função juntats une duas tabelas de símbolos, retornando o índice que englobar o outro. A função peganats pesquisa a existência de um símbolo em toda a lista e se encontrar retorna os dados da variável. A função pegasimb lê um registro do arquivo intermediário gravado pelo analisador léxico. A função pegatam retorna o número de bytes ocupados por um determinado tipo de variável.

```
type tabsimb = integer;
var  ListaSimb : array [0..1000] of record
                    nome   : string [30];
                    formt  : byte;
                    pos    : byte;
                    end;
     TopoSimb  : tabsimb;
const vazio = 0;
```

Figura V.15 - Tabela de símbolos do compilador G

```
function juntats (t1, t2 : tabsimb):tabsimb;
    begin
    if t1 > t2 then juntats := t1
                else juntats := t2;
    end; {juntats}
function entrats (nome:string; f,pos:integer;
                ts:tabsimb) : tabsimb;
        var i : integer;
    begin
    for i:=toposimb-1 downto ts
    do if ListaSimb [i].nome = nome
       then begin
            writeln (^G, 'Simbolo ja'' declarado');
            entrats := i;
            exit;
            end;
    ListaSimb [topoSimb].nome := nome;
    ListaSimb [topoSimb].formt := f;
    ListaSimb [topoSimb].pos := pos;
    TopoSimb := TopoSimb + 1;
    entrats :=  topoSimb;
    end; {entrats}
function peganats (nome:string; var f:integer) : integer;
        var i : integer;
    begin
    for i:= toposimb-1 downto 0
    do if ListaSimb[i].nome = nome
       then begin
            peganats := ListaSimb[i].pos;
            f := ListaSimb [i].formt;
            exit;
            end;
    writeln (^G, 'Simbolo nao declarado');
    peganats := -1;
    end; {peganats}
function pegasimb : string;
        var s : string;
    begin
    readln (arqnomes, s);
    pegasimb := s;
    end; {pegasimb}
function pegatam (formato : integer) : integer;
    begin
    if formato = tipoint  then pegatam := 2;
    if formato = tipochar then pegatam := 1;
    end; {pegatam}
```

Figura V.16 - Funções de manuseio da tabela de símbolos

Na figura V.17 encontramos a estrutura de dados escolhida para a geração do código objeto, uma lista de quádruplas que no nosso exemplo foi simplicada para apenas dois campos: operador e operando.

Na figura V.18 temos as funções geraatrib, geracond e geraoper responsáveis pela emissão de instruções.

Na figura V.19 temos a função backpatch que modifica um trecho de código já emitido, e a função encadeia que vai empilhando as constantes do programa em um segmento à parte.

```
type instrucao = (Stop, Load, Store, Jump, Jnotzero, Jzero,
                  Add, Sub, Mult, Divisao);
type quadrupla = record
          oper  : instrucao;
          opnd1 : byte;
          end;
var  codigo : array [0..1000] of quadrupla;
     ListaCte : array [0..200] of record
                       nome  : integer;
                       formt : integer;
                       end;
     topo      : integer;
const tipochar = 1; tipoint = 2;
      Incond = 0; Condtrue = 1; Condfalse = 2;
      OperSoma = 1; OperSub = 2; OperMult = 3; OperDiv = 4;
```

Figura V.17 - Variáveis de geração de código

```
function geraatrib (Cod:boolean; f1,f2,ini,Vend,Eend:integer):boolean;
    begin
    if Cod then begin
          if f1 <> f2 then begin
                writeln (^G, 'Tipos conflitantes');
                geraatrib := false;
                end
          else begin
                codigo [ini].oper     := Load;
                codigo [ini].opnd1    := Eend;
                codigo [ini+1].oper   := Store;
                codigo [ini+1].opnd1 := Vend;
                geraatrib := true;
                end;
          end
    else geraatrib := false;
    end; {geraatrib}
function geracond (ini,ender,condicao:integer) : integer;
    begin
    if condicao = Incond then begin
          codigo [ini].oper  := Jump;
          geracond := ini;
          end
    else begin
          codigo [ini].oper     := Load;
          codigo [ini].opnd1    := ender;
          geracond := ini + 1;
          end;
    if condicao = Condtrue then codigo [ini+1].oper := Jnotzero;
    if condicao = Condfalse then codigo [ini+1].oper := Jzero;
    end; {geracond}
function geraoper (Cod : boolean; f1,f2,ini,E1,Op,E2,E0:integer):boolean;
    begin
    if Cod then begin
          if f1 <> f2 then begin
                writeln (^G, 'Tipos conflitantes');
                geraoper := false;
                end
          else begin
                codigo [ini].oper     := Load;
                codigo [ini].opnd1    := E1;
                if Op = OperSoma then codigo [ini+1].oper  := Add;
                if Op = OperSub  then codigo [ini+1].oper  := Sub;
                if Op = OperMult then codigo [ini+1].oper  := Mult;
                if Op = OperDiv  then codigo [ini+1].oper  := Divisao;
                codigo [ini+1].opnd1 := E2;
                codigo [ini+2].oper  := Store;
                codigo [ini+2].opnd1 := E0;
                geraoper := true;
                end;
          end
    else geraoper := false;
    end; {geraoper}
```

Figura V.18 - Funções de geração de código

```
function backpatch (marca:integer; ender:integer) : boolean;
    begin
    codigo [marca].opnd1 := ender;
    backpatch := true;
    end; {backpatch}
function encadeia (s: string; f:integer):integer;
    begin
    ListaCte [topo].nome := PutNomes (s);
    ListaCte [topo].formt := f;
    encadeia := topo;
    topo := topo + 1;
    end; {encadeia}
```

Figura V.19 - Funções de alteração e de controle de constantes

CAPÍTULO VI

CONCLUSÕES

VI.1 - Resultados alcançados

Conseguimos um método prático para a utilização de gramática de atributos. Através de uma ferramenta, podemos construir um avaliador de atributos a partir da gramática correspondente. Deste modo, esta ferramenta tomou a forma de um compilador para uma linguagem, especialmente projetada para a descrição da gramática, chamada LINDA (LINguagem de Descrição de Atributos).

Esta gramática não precisa ser tão complexa como a que descreve a sintaxe da linguagem. Ao lançar mão de ambigüidades na gramática, o projetista da linguagem simplifica a gramática de atributos, aumentando a clareza e diminuindo o número de símbolos. Através de dispositivos da linguagem LINDA o projetista estabelece uma relação direta entre as gramáticas abstrata e concreta, estabelecendo uma clara documentação do projeto e permitindo ao compilador LINDA implementar o avaliador de atributos eficientemente.

Combinamos a técnica citada ao analisador R*S. Este elimina reduções em produções simples, permitindo a economia do espaço de memória que seria alocado para os atributos, e a otimização do processamento do avaliador de atributos por evitar "cópias" dos atributos.

A linguagem LINDA permite ainda transformar regras não

simples em regras semanticamente simples, o que significa estender as vantagens do analisador R*S para, virtualmente, qualquer produção da linguagem.

Todos os resultados obtidos da aplicação desta ferramenta estão em código fonte Pascal, facilitando a interferência direta do projetista quando este o desejar.

A criação destas ferramentas trouxe sub-produtos que podem formar um ambiente de trabalho. Assim, para formar o compilador da linguagem alvo, o projetista disporia de:

1 - Esqueleto de analisador léxico com função Hash padrão.
2 - Tabelas para o analisador léxico:
    1) Tabela hash com os símbolos terminais da linguagem alvo que comecem com uma letra;
    2) Tabela de strings com os nomes de todos os terminais da linguagem alvo;
    3) Tamanho máximo da sequência de colisões na tabela hash.
3 - Analisador sintático R*S padrão.
4 - Tabelas para o analisador sintático emitidas pelo gerador R*S.
5 - Tabela para o analisador sintático fornecida pelo compilador LINDA contendo o número de ocorrências de não-terminais de cada produção. Será usado na montagem da árvore sintática na memória.
6 - Rotina recursiva de percurso em árvore para cálculo dos atributos das regras semânticas.
7 - Bibliotecas de funções para manipulação de tabelas de símbolos e árvores semânticas.

Para completar o compilador, o projetista deverá providenciar:

1 - Modificar o esqueleto do analisador léxico de acordo com os *tokens* da linguagem alvo. Os códigos dos tokens devem combinar com as tabelas emitidas pelo gerador R*S.
2 - Modificar o analisador léxico caso a sequência máxima de colisões seja maior que 1.

3 - Projetar a gramática de atributos.

4 - Incorporar ao analisador sintático as declarações e as funções necessárias ao funcionamento da gramática de atributos.

VI.2 - Sugestões para aperfeiçoamento

O compilador da linguagem LINDA necessita da inclusão de um recuperador de erro para tornar o processo de compilação mais eficiente. Sugerimos implementar um esquema o mais simples possível de modo a permitir a visualização de vários erros com uma só compilação.

As mensagens de erros sintáticos estão codificadas pelo estado interno do analisador e pelo último *token*. Embora a maioria dos erros sintáticos possa ser resolvidos com apenas a localização dos mesmos, sugiro incluir uma rotina que indexe uma tabela de mensagens com informações mais diretas, por exemplo, "Esperado ;" ou "Símbolo inválido".

Como esta tese envolve a implementação prática de uma técnica, podemos sugerir alguns aprimoramentos da técnica de tradução para melhorar a performance do sistema:

1 - Avaliador por pedaços de rotinas, cada qual com o seu arranjo de ponteiros dimensionado para cada regra semântica. - Menos memória

2 - Substituir a instrução *case* por uma tabela de ponteiros para *procedures* (ver item 1). - Maior velocidade e menos memória.

VI.3 - Sugestões para outros trabalhos

Implementar um sistema de correção de erro dirigido por tabela ao analisador sintático padrão. Trabalho semelhante já foi desenvolvido por L. C. Zancanella para analisadores descendentes.

REFERÊNCIAS BIBLIOGRÁFICAS

AHO, A. V., ULLMAN, J. D., (1972) "The theory of Parsing Translation and Compiling", Vol I : Parsing, Prentice Hall (1972).

AHO, A. V., SETHI, R., ULLMAN, J. D., (1986) "Compilers, Principles, Techniques and Tools", Addison Wesley, (1986).

BARRET, W.A., COUCH, J.D., (1979) "Compiler Construction Theory and Practice", Science Research Associates (1979).

BOCHMANN, GREGOR V., (1976) "Semantic evaluation from left to right", Communications of the ACM, feb. 1976, vol 19, nº 2, pp. 55-62.

BORLAND, (1988) "Turbo Pascal Owner's Handbook", Borland International (1988).

DERANSART, P., JOURDAN, M. e LORHO, B., (1988) "Attribute grammars: Definitions, Systems and Bibliography", Springer-Verlag, Lecture Notes in Computer Science, vol 323, (1988).

KNUTH, D. E., (1968) "Semantics of Context Free Languages", Math. Systems Theory 2, jun. 1968, pp. 127-145, Correção em: Math. Systems Theory 5, mar 1971, pp. 95-96.

NIJHOLT, A., (1980) "Context-Free Grammars: Covers, Normal forms, and Parsing", Springer-Verlag, Lecture Notes in Computer Science, vol 93, (1980).

RANGEL, J. L., (1988) "Manual de operação do sistema de geração de analisadores sintáticos R*S simples", Departamento de informática PUC-RJ, (1988).

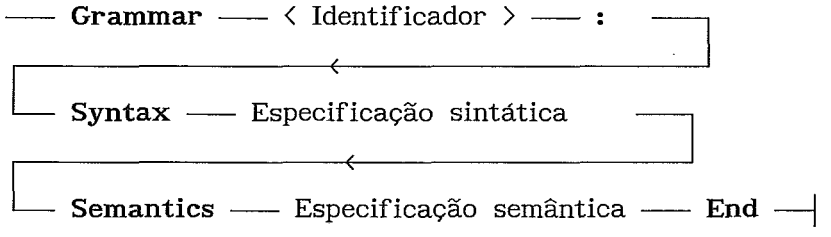
TAGUATA, D. T. (1982) "Uma linguagem de especificação para a gramática de atributos UPED", Tese de mestrado COPPE / UFRJ, (1982).

APÊNDICE A

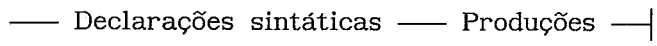
SINTAXE DA LINGUAGEM LINDA

Este apêndice contém a especificação sintática da linguagem LINDA. Na nomenclatura usada por nós a ocorrência dos colchetes angulares ('<' e '>') denotam que o programador deve suprir aquele espaço com um objeto do tipo indicado entre os colchetes, e nomes em negrito são os símbolos terminais.

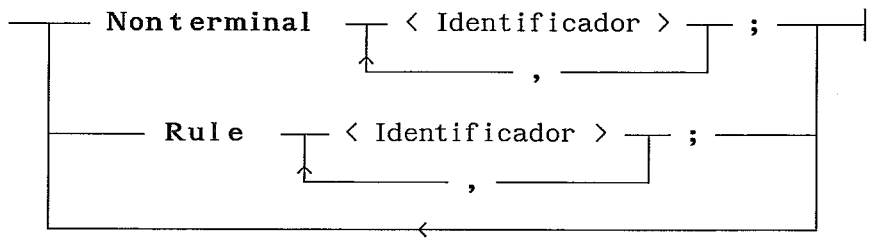
Programa =>



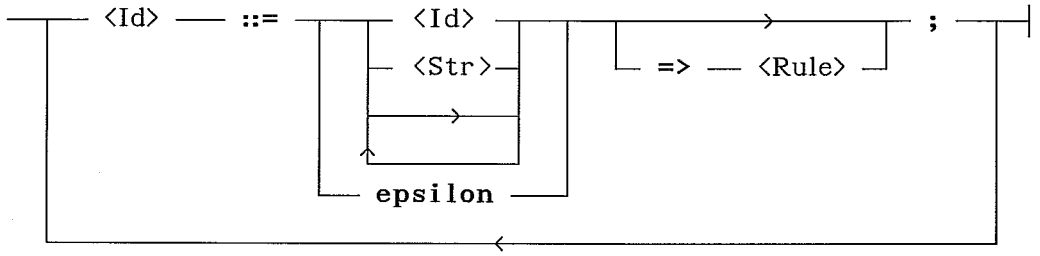
Especificação Sintática =>



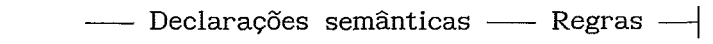
Declarações sintáticas =>



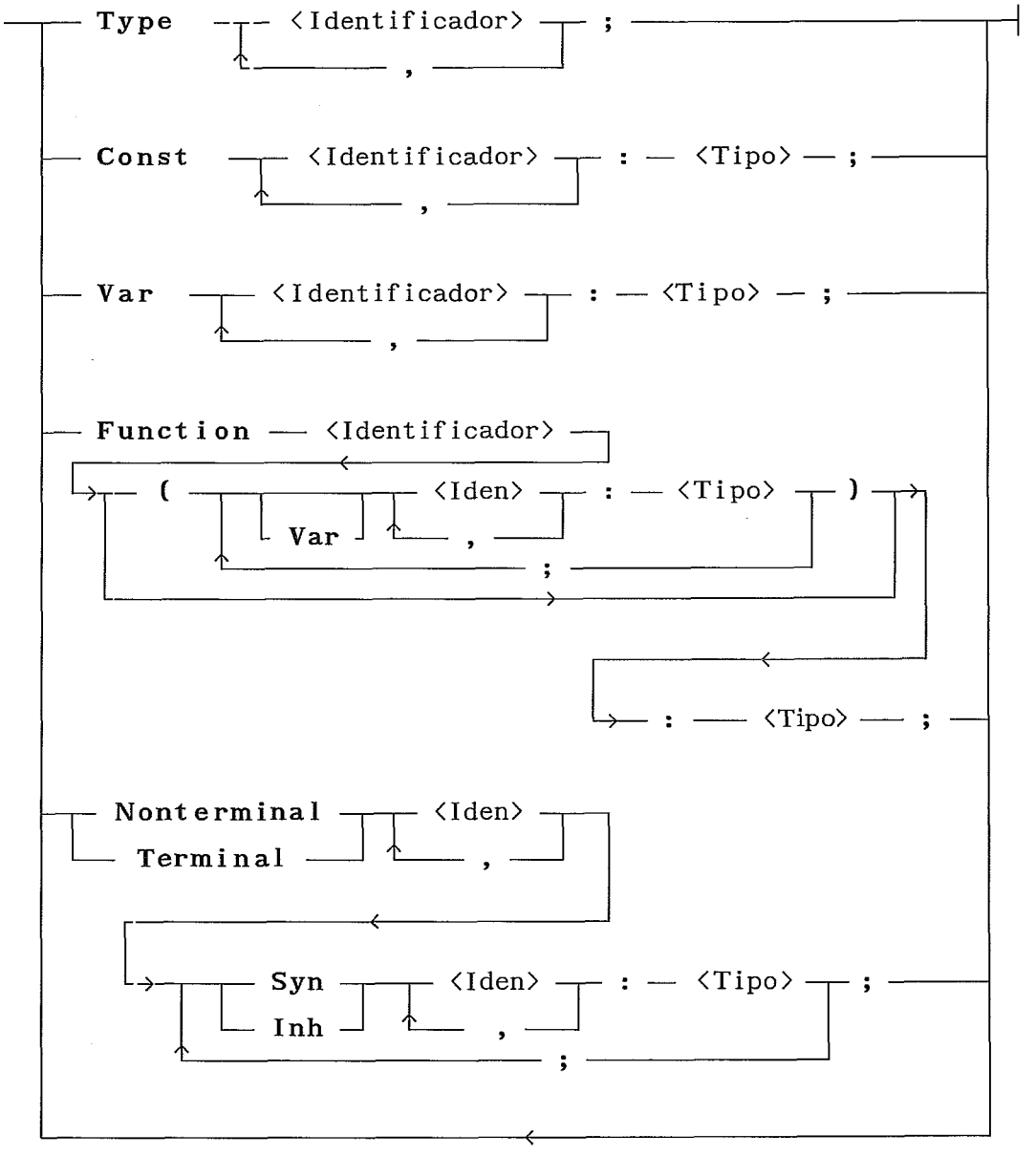
Produções =>



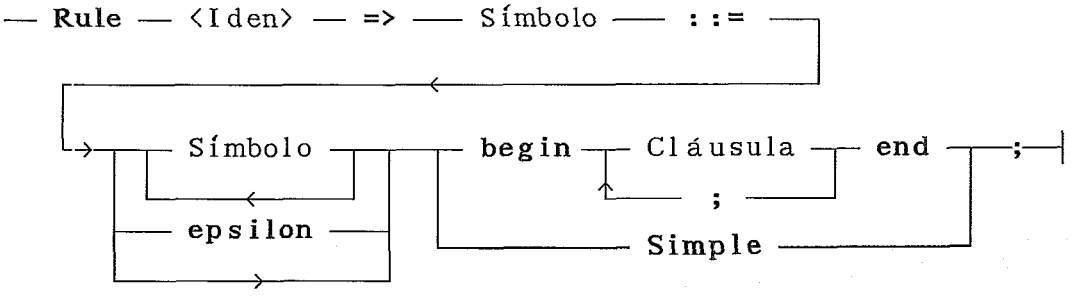
Especificação semântica =>



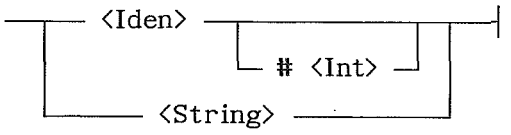
Declarações semânticas =>



Regras =>

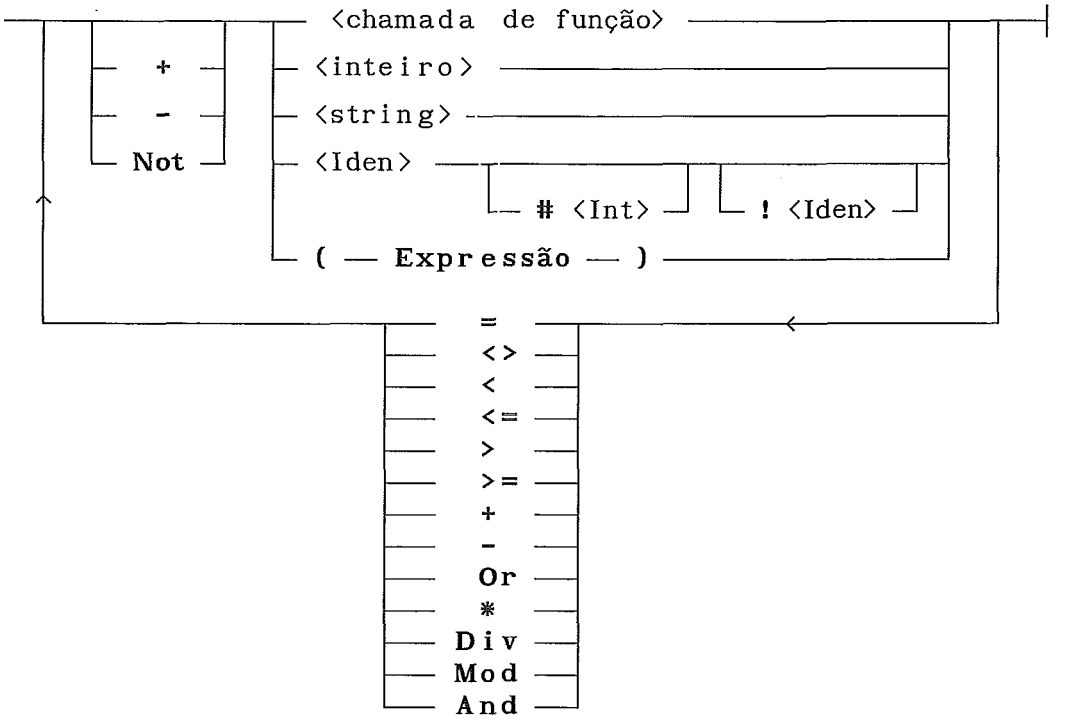


Símbolo =>



Cláusula =>

Expressão =>



Lista de símbolos terminais e sinônimos em português:

Grammar	Gramatica
Syntax	Sintaxe
Semantics	Semantica
Begin	Inicio
End	Fim
Nonterminal	Naoterminal
Terminal	
Rule	Regra
Epsilon	
Type	Tipo
Const	
Var	
Function	Funcao
Syn	Sint
Inh	Herd
Simple	Simples

Operadores aritméticos:

+ - * Div Mod

Operadores booleanos:

Or And Not

Operadores relacionais:

= <> < <= > >=

Outros símbolos terminais:

: ; , ::= => ( ) := # !