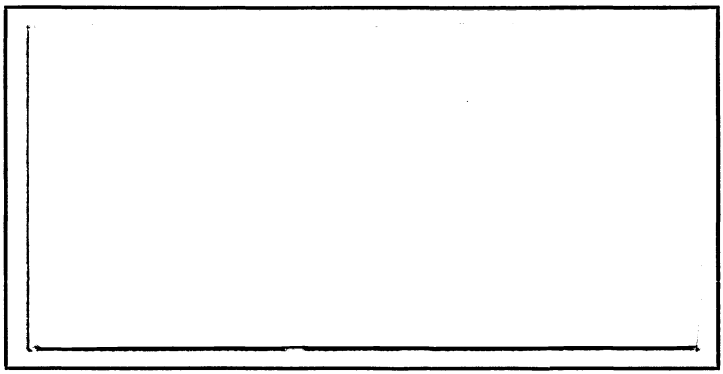
UNIVERSIDADE DE SÃO PAULO
CAMPUS DE SÃO CARLOS
INSTITUTO DE CIÊNCIAS MATEMÁTICAS DE SÃO CARLOS
I.C.M.S.C.



Relatório Técnicos do ICMSC - USP

ISSN - 0103-2509

Uso de Programação Lógica no Desenvolvimento de Compiladores

Maria Carolina Monard; João Luiz Franco

Nº 8

São Carlos (SP)
1991

Uso de Programação Lógica no Desenvolvimento de Compiladores [1]

MARIA CAROLINA MONARD

JOÃO LUIZ FRANCO

Universidade de São Paulo / ILTC
Instituto de Ciências Matemáticas de São Carlos
Departamento de Ciências de Computação e Estatística

Sumário

O objetivo deste trabalho é mostrar a adequação da linguagem de programação lógica Prolog no desenvolvimento de compiladores.

Uma linguagem de programação simples — denominada $\mathcal{CALCULA}$L — é definida para mostrar em detalhes o processo de compilação. A linguagem $\mathcal{CALCULA}$L possui os comandos comumente encontrados nas linguagens de programação procedimentais estruturadas.

A máquina para a qual essa linguagem é compilada é uma máquina virtual típica, que possui um único acumulador e um subconjunto das instruções comumente encontradas nas máquinas com arquitetura de von Newman.

A implementação das diversas fases do compilador proposto é mostrada em detalhes, bem como sugestões para estender a linguagem $\mathcal{CALCULA}$L e o compilador.

[1]Trabalho realizado com auxílio do CNPq e ILTC - Instituto de Lógica Filosofia e Teoria da Ciência

Conteúdo

1 Introdução

Em meio século de história, os computadores aumentaram em capacidade e em velocidade de processamento na razão direta da sua diminuição de custo. Desta forma, os computadores se popularizaram e passaram a fazer parte da vida de grande número de profissionais das mais diversas áreas. Esta evolução não ocorreu apenas a nível de hardware, mas também a nível de software. Os programas de computador não apenas se diversificaram como também aumentaram em complexidade.

Embora para os usuários não especialistas em computação seja cada vez mais agradável operar um aplicativo, o desenvolvimento desses sistemas continua sendo uma tarefa de grande complexidade. Se de um lado há um usuário que exige cada vez mais dos aplicativos — capacidade de armazenamento, interface amigável, etc. — do outro lado há uma máquina que, apesar de ser mais rápida e de possuir maior capacidade de armazenamento, continua entendendo somente a mesma linguagem que o velho ENIAC, ou seja, um conjunto de zeros e uns.

É neste contexto que aparecem os compiladores, uma importante ferramenta de programação. Sua função é fazer uma ponte entre a linguagem usada pelo programador de um sistema computacional e a linguagem entendida pelo computador. Os compiladores também acompanharam a evolução dos programas, pois atualmente já existem poderosos ambientes de programação, onde se pode editar, compilar e depurar um programa com muita facilidade, auxiliando o programador na tarefa de produção de software.

Após anos de pesquisas na área, pode-se afirmar que tanto a teoria quanto a técnica de construção de compiladores está bem entendida e estruturada, sendo que existem muitos bons livros textos e publicações nesta área.

O objetivo deste trabalho é mostrar a adequação da linguagem de programação lógica Prolog para implementar compiladores. Para isso, a linguagem para a qual será implementado o compilador é uma linguagem de programação procedimental simples, que possui os comandos comumente encontrados nas linguagens de programação estruturadas. A máquina para a qual essa linguagem é traduzida consiste de uma máquina virtual típica, que possui um único acumulador e um subconjunto das instruções comumente encontradas nas máquinas com arquitetura de Von Newman.

O núcleo da implementação Prolog do compilador proposto neste trabalho está baseado no excelente artigo publicado por David Warren [Warren 80]. Algumas idéias e partes do trabalho provêm de trabalhos desenvolvidos por diversos estudantes de pós-graduação do ICMSC-USP, área Computação, no curso SCE-717 — Teoria e Técnicas de Construção de Compiladores, ministrado no segundo semestre de 1988[2].

O trabalho está organizado da seguinte forma:

[2]Participaram os seguintes estudantes: Bráulio Coelho Ávila, Carlos Roberto Valêncio, João do E. S. Batista Neto, José Pacheco de Almeida Prado, Luiz Manoel da Silva Cunha, Maria Inés Castiñeira, Nivaldi Calonego Júnior, Pedro Luiz Pizzigatti Corrêa, Ricardo Luís de Freitas, Sandra Maria Alvísio Caldeira e Solange Rezende Rodrigues

Na seção 2 é discutido o processo básico de compilação.

Na seção 3 é mostrada a estrutura geral de um compilador e as fases em que está dividido.

A seção 4 trata o projeto de implementação, descrevendo a linguagem a ser compilada — denominada *CALTAL* —, a máquina virtual para a qual serão traduzidos os comandos dessa linguagem, bem como exemplos de compilação de programas *CALTAL*.

Na seção 5 a implementação Prolog das diversas fases do compilador proposto é mostrada em detalhes, enquanto a seção 6 contém a listagem completa dessa implementação.

Na seção 7 são apresentadas várias sugestões para estender a linguagem *CALTAL* e o compilador.

Finalmente, são apresentadas as conclusões.

2 Compiladores

2.1 Considerações Iniciais

O advento de computadores eletrônicos tornou evidente a existência de uma barreira de comunicação entre o homem e a máquina: enquanto as pessoas preferem expressar-se usando linguagem natural, os computadores baseados na arquitetura de Von Newman operam, em geral, manipulando dígitos binários, registradores, posições de memória, etc. Eles possuem uma linguagem própria, denominada linguagem de máquina, que em nada se parece com a linguagem humana. As linguagens de programação de alto nível foram introduzidas com a intenção de superar essa barreira de comunicação.

As primeiras linguagens introduzidas foram as linguagens de montagens, mas ainda eram muito próximas das linguagens de máquina. Um passo decisivo foi tomado na segunda metade da década de 50, com a introdução de linguagens de alto nível, como Fortran e Algol 60, superando assim a barreira de comunicação com a máquina. Desde então surgiram centenas de linguagens de alto nível, algumas das quais tiveram larga utilização.

Com a introdução de linguagens de alto nível, surgiu a necessidade de programas tradutores, isto é, sistemas que traduzissem programas escritos por programadores — *programas fonte* — para programas em linguagem de máquina — *programas objeto*.

Por motivos históricos, os tradutores para linguagens de montagem são chamados *montadores*, enquanto os utilizados para linguagens de alto nível são chamados *compiladores*.

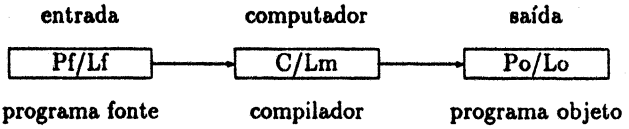
Existem outras soluções para a implementação de linguagens de alto nível. Uma possibilidade é o uso de *interpretadores*, que em vez de traduzir de uma vez o programa fonte para então executá-lo, decodificam unidades básicas do programa para executá-las imediatamente.

2.2 O Processo de Compilação

Denomina-se compilador um programa C que tem por finalidade traduzir um programa Pf, escrito em uma linguagem Lf, para um outro programa equivalente Po escrito em uma linguagem Lo, onde :

Pf	é o programa fonte
Lf	é a linguagem fonte
Po	é o programa objeto
Lo	é a linguagem objeto

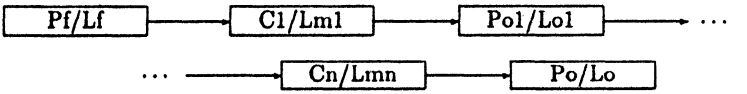
Este processo pode ser representado graficamente da seguinte forma:



Como entrada para o compilador C tem-se o programa fonte Pf escrito na linguagem Lf, onde este compilador C já está traduzido para a linguagem de máquina Lm do computador onde ele é processado, que produz como resultado da tradução um programa objeto Po escrito em uma linguagem objeto Lo.

Geralmente, a linguagem fonte Lf é uma linguagem de alto nível (como por exemplo PASCAL, FORTRAN, C, COBOL), enquanto a linguagem objeto pode ser uma linguagem de montagem (Assembly) La. Quando isto ocorre, é necessário que haja mais de uma fase de tradução da linguagem La para a linguagem Lm correspondente ao computador que está sendo usado.

Um compilador C pode ser composto de vários passos, partindo do programa fonte Pf/Lf, passando por compiladores Ci/Lmi, com programas fonte Pfi/Lfi e produzindo programas objeto Poi/Loi, até chegar no programa objeto Po/Lo desejado, como mostrado a seguir :



3 Estrutura Geral de um Compilador

Os primeiros compiladores construídos não apresentavam uniformidade de estrutura. Entretanto, a tarefa do planejamento da construção de compiladores — independentemente da linguagem a ser compilada e da máquina a que se destina o código objeto por ele gerado — é essencialmente a mesma, diferindo nos detalhes inerentes a particularidades das linguagens e das máquinas objeto ou na sua organização física.

Compilar um programa é, de forma geral, ler uma sequência de caracteres de um meio externo, reconhecer nestes caracteres as estruturas elementares da linguagem — *átomos* — e verificar se estas estruturas estão organizadas de acordo com as regras da linguagem — *sintaxe*. Nesta etapa, o compilador também deve ser capaz de indicar os erros de sintaxe, para que eles possam ser corrigidos posteriormente. Finalmente, o compilador deve captar o significado do texto fonte — semântica — para que a tradução para o programa objeto possa ser feita.

Foi pelo acúmulo de experiência, pela observação dos resultados e, principalmente, pelo desenvolvimento de teorias relacionadas à tarefa da análise e síntese de programas, que se tornou possível criar concenso acerca de uma estruturação adequada para desenvolver compiladores.

O funcionamento básico de um compilador está indicado de maneira esquemática na Figura 1, dividido em seis fases:

Análise Léxica
Análise Sintática
Análise Semântica
Otimização Global
Geração de Código
Otimização Local.

Em geral, as tabelas do compilador são consultadas durante todas estas fases. Algumas destas tabelas têm conteúdo fixo, como por exemplo a tabela de palavras reservadas e caracteres especiais da linguagem. Outras, como a tabela de símbolos, são criadas a partir do próprio programa fonte e utilizadas nas várias fases de construção do compilador.

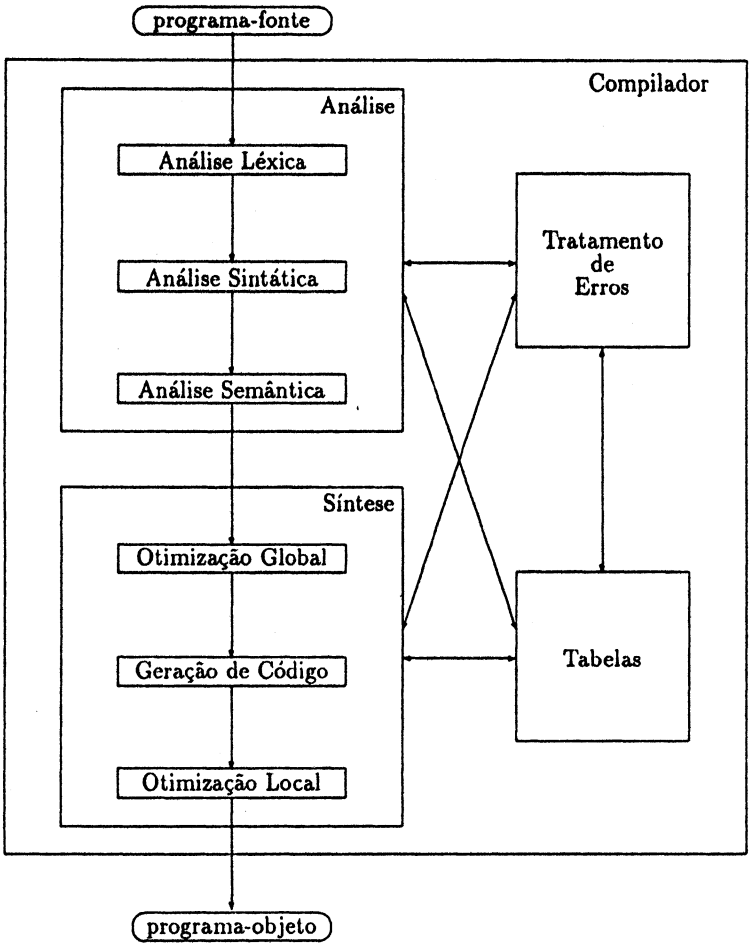


Figura 1: Funcionamento Básico de um Compilador

3.1 Análise Léxica

A principal função do Analisador Léxico é a de fragmentar o programa fonte de entrada em componentes básicos, indentificando trechos elementares completos e com identidade própria, que são chamados de átomos ou *tokens*.

Em geral, os átomos, extraídos do programa fonte pelo analisador léxico, são representados na metalinguagem que descreve a linguagem fonte na mesma forma que aparecem no texto. Para descrever a linguagem fonte a ser compilada, será usada a notação BNF — Backus-Naur Form ou Forma Normal de Backus. Os átomos representam, portanto, os símbolos terminais da gramática. Do ponto de vista de implementação do compilador, o analisador léxico atua como uma interface entre o analisador sintático e o texto de entrada, convertendo a sequência de caracteres de que o programa se constitui na sequência de átomos que o analisador sintático consumirá.

Para a consecução de seus objetivos, o analisador léxico executa usualmente uma série de funções, não obrigatoriamente ligadas à análise léxica, porém todas de grande importância para a correta operação das outras partes do compilador. Algumas das funções do analisador léxico são discutidas a seguir:

- **Extração e Classificação dos Átomos:** sua principal função é fazer um mapeamento do texto do programa fonte em outro texto, formado por átomos, que representam os símbolos do programa fonte. Entre a classe de átomos mais encontradas em analisadores léxicos usuais, os que se destacam são: identificadores, palavras reservadas, números inteiros, cadeias de caracteres, caracteres especiais simples e compostos.

- **Eliminação de Delimitadores e Comentários:** ainda que importantes no programa fonte por razões de legibilidade para o programador, os espaços em branco, os símbolos separadores e os comentários são irrelevantes do ponto de vista de geração de código. Portanto, podem ser eliminados pelo analisador léxico.

- **Identificação de Palavras Reservadas:** sempre que for encontrado um identificador no programa fonte, é necessário verificar se ele pertence ao conjunto das palavras reservadas. Não há nenhuma distinção entre os dois do ponto de vista formal. Em geral, a separação entre as duas classes é feita em duas etapas: primeiramente o átomo é reconhecido como identificador; em seguida, é verificado se ele pertence ao conjunto de palavras reservadas. Caso isto ocorra, muda-se a classe do átomo.

- **Recuperação de Erros:** a ocorrência de cadeias de caracteres que não obedecem a nenhuma lei de formação das classes de átomos que o analisador léxico tem condição de reconhecer determina a constatação de erros léxicos no programa fonte. Para que o analisador léxico possa prosseguir a análise do programa fonte é necessário criar mecanismos de recuperação de erros. A idéia seria prosseguir a análise, para que se possa mostrar o maior número de erros encontrados no programa fonte.

Em um compilador de um passo, uma opção seria a de percorrer o programa fonte à procura de algum delimitador, que pode ser um sinal de pontuação ou uma palavra reservada especial, descartando-se os caracteres intermediários, e levando-se o analisador sintático a um estado de início de reconhecimento de símbolos. Um outro meio seria o analisador léxico comportar mais uma classe de átomos, que seria composta de todas as cadeias de caracteres não identificáveis no programa fonte. Assim, ele sempre devolveria um átomo, deixando a recuperação de erros para o analisador sintático.

- **Geração de Listagens:** embora não se trate de uma tarefa de análise, os analisadores léxicos construídos para linguagens específicas fornecem listagens do programa fonte, ou eventual reedição do programa num formato mais legível, usando identação e/ou enumeração de linhas.

- **Geração de Referências Cruzadas:** durante a leitura dos átomos e das linhas do programa fonte, pode-se armazenar informações sobre a ocorrência de símbolos em uma tabela. Deste modo, ao final da análise léxica, poderá ser gerada uma listagem indicativa dos símbolos encontrados e sua localização.

3.2 Análise Sintática

A análise sintática cuida exclusivamente da forma das sentenças da linguagem, baseando-se na gramática que define a linguagem. Como centralizador das atividades da compilação, o analisador sintático opera em compiladores dirigidos por sintaxe como elemento de comando da ativação dos demais módulos do compilador, efetuando decisões acerca de qual módulo deve ser ativado em cada situação da análise do programa fonte.

O analisador sintático é responsável pela recepção de uma sequência de átomos que foram extraídos do programa fonte pelo analisador léxico. A partir desta sequência, o analisador sintático efetua uma verificação acerca da ordem de apresentação dos átomos, identificando em cada situação o tipo da construção sintática por eles formada, de acordo com a gramática na qual se baseia o reconhecedor. Na maioria das linguagens de alto nível, as construções sintáticas são definidas através de gramáticas livres de contexto.

A análise sintática engloba, em geral, diversas funções de grande importância, tais como:

- **Identificação de Sentenças:** o analisador sintático pode ser visto como aceitador de cadeias. O conjunto dessas cadeias forma a linguagem a que se refere o analisador.

- **Detecção de Erros de Sintaxe:** se uma cadeia não pertence à linguagem, o analisador sintático deve identificar sua ocorrência, acusando a presença de erros de sintaxe, de preferência através de uma indicação impressa na qual o programador seja informado sobre o ponto de detecção do erro, o tipo de erro detectado e, se for possível, a causa do erro.

- **Recuperação de Erros:** muitos compiladores incorporam, no mecanismo de análise sintática, meios de ressincronizar o reconhecedor sempre que forem identificadas construções não pertencentes à linguagem. Isto é importante para que o restante do programa fonte possa continuar sendo analisado, em busca de todos os erros que o compilador puder detectar.

3.3 Análise Semântica

A terceira grande tarefa do compilador refere-se à tradução do programa fonte para o programa objeto. Em geral, a geração de código vem acompanhada, em muitas implementações, das atividades de análise semântica, que são responsáveis pela captação do significado do texto fonte. Esta operação é essencial à realização da tradução do texto fonte por parte das rotinas de geração de código.

Denomina-se semântica de uma sentença o significado por ela assumido dentro do contexto em que se encontra. Semântica de uma linguagem é a interpretação que se pode atribuir ao conjunto de todas as suas sentenças. Ao contrário da sintaxe, que é facilmente formalizável, a semântica exige para isso notações mais complexas. Assim, na maioria das linguagens de programação, a semântica tem sido especificada informalmente, geralmente através de linguagem natural.

As atividades de tradução, exercidas pelos compiladores, baseiam-se em uma perfeita compreensão da semântica da linguagem a ser compilada, uma vez que é disto que depende a criação das rotinas de geração de código, responsáveis pela obtenção do código objeto a partir do programa fonte.

A principal função da análise semântica é criar, a partir do texto fonte, uma interpretação deste texto fonte, expressa em alguma notação adequada — geralmente uma linguagem intermediária do compilador. Isto é feito com base nas informações das tabelas e nas saídas dos outros analisadores. Tipicamente, as ações semânticas englobam funções como:

- **Criação e Manutenção da Tabela de Símbolos:** em geral, esta tarefa é executada pelo analisador léxico, mas, por razões de reaproveitamento dos programas de análise léxica — e mesmo sintática — as ações semânticas podem incorporar essa tarefa. A cada ocorrência de um identificador no texto fonte, a tabela de símbolos é consultada ou alterada. Sempre que um objeto da linguagem é declarado ou encontrado em um contexto, ele deve ser inserido na tabela de símbolos, se já não estiver presente. A criação e manutenção da tabela de símbolos é vital para o funcionamento do compilador, pois é nela que são guardados os nomes dos objetos definidos pelo programador e que são referenciados ao longo do programa. São, em geral, organizadas de modo que reflitam a estrutura do programa fonte, guardando, por exemplo, informações sobre o escopo em que os identificadores são definidos.

- **Associar aos Símbolos os Atributos Correspondentes:** é necessário acrescentar, para cada identificador da tabela de símbolos, um conjunto de informações que

seja suficiente para caracterizá-lo como sendo correspondente a um determinado objeto, indicando todas as características que tal objeto exige e que sejam de interesse para o processo de geração de código.

- **Verificar a Compatibilidade de Tipos:** a checagem de tipos é um recurso auxiliar para as atividades de geração de código, uma vez que o uso de dados cujos tipos sejam diferentes — embora coerentes — impõem ao gerador de código a tarefa de efetuar as conversões necessárias. Isto deve ser feito para permitir que as operações especificadas pelos comandos da linguagem sejam realizadas adequadamente.

3.4 Geração de Código

Com base na tradução prévia para uma linguagem intermediária, que eventualmente pode ser a linguagem de uma máquina real, o código objeto do programa pode ser construído. Em geral, a construção deste código pode ser feita de duas maneiras: ou sem cuidados adicionais, gerando um código que é uma tradução literal da interpretação do programa fonte (geralmente de qualidade inferior), ou então preparado para ser otimizado (caso o compilador tenha um otimizador incorporado). Em geral, o código gerado é relocável.

3.5 Otimização

A maioria dos bons compiladores modernos estão, cada vez mais, explorando técnicas de otimização, com a finalidade de construir automaticamente código de qualidade comparável com o gerado manualmente.

Na otimização global, que é independente de máquina, são efetuadas manipulações da árvore sintática, com a finalidade de reduzí-la. Nesta classe encontram-se: as otimizações de expressões, a eliminação de subexpressões comuns, a fatoração do cálculo de subexpressões de uso frequente, as otimizações das construções iterativas, etc.

Outra classe de otimização é a otimização local, na qual a máquina em que vai ser executado o programa influi na otimização a ser feita. Nesta classe estão: a otimização do uso de registradores, a otimização do uso do conjunto de instruções de máquina, a otimização do tempo de execução do programa, etc. Geralmente, essa otimização opera por transformação do código gerado previamente.

3.6 Fases do Compilador

O processo de compilação pode ser realizado em cinco fases — Figura 2. Neste trabalho serão tratados em detalhes as quatro primeiras fases.

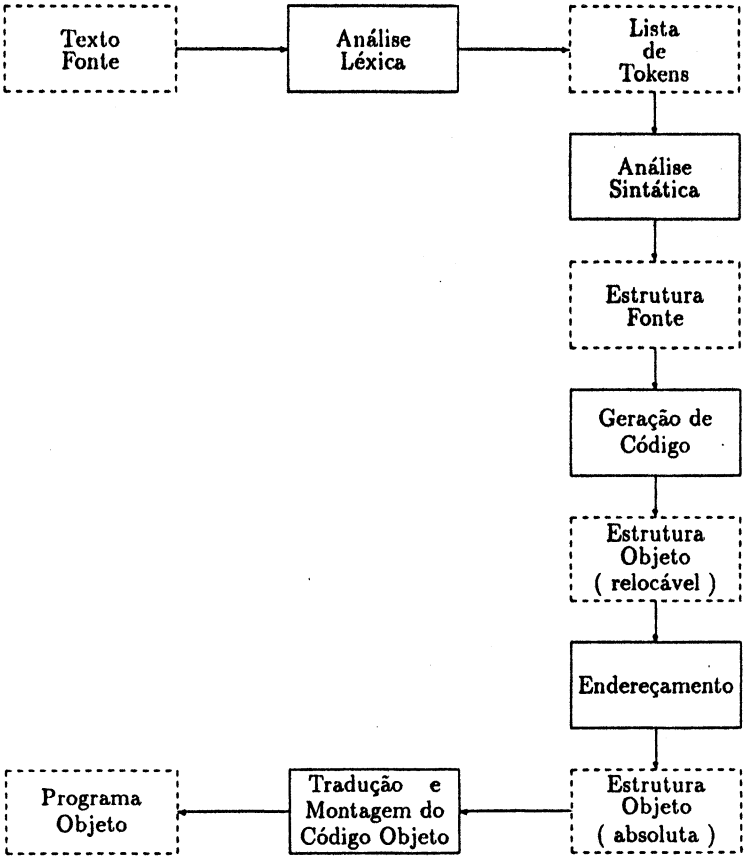


Figura 2: Etapas da Compilação

4 Projeto de Implementação

4.1 Máquina Virtual

Antes de executar a tarefa de elaborar um compilador para determinada linguagem, faz-se necessário o estabelecimento da tradução para os comandos da linguagem em questão. Como as linguagens de máquina se preocupam com muitos detalhes específicos da máquina, o trabalho de tradução torna-se muito grande. Uma possível solução é, em vez de se traduzir diretamente para a linguagem de máquina de um computador real, definir uma máquina hipotética e uma linguagem de montagem na qual a máquina é programada.

Neste trabalho será usada uma máquina hipotética composta de um acumulador e um registrador CP — contador de programa. O funcionamento da máquina é muito simples. As instruções indicadas pelo registrador CP são executadas até que seja encontrada a instrução de parada. A execução de cada instrução provoca um incremento no registrador CP, excetuando-se o caso de instruções que envolvam desvios.

Define-se como *p-code* [3] o código entendido por esta máquina hipotética. Observe que é possível definir qualquer p-code, mas é preferível que o p-code definido seja um código intermediário entre as linguagens de máquina de computadores reais. As instruções desta máquina hipotética são:

Operações com Literais	Operações com Memória	Transferencia de Controle	Entrada/Saida Outras
ADDC	ADD	JUMP	READ
SUBC	SUB	JUMPEQ	WRITE
MULTC	MULT	JUMPNE	HALT
DIVC	DIV	JUMPLT	BLOCK
LOADC	LOAD	JUMPGT	
	STORE	JUMPLE	
		JUMPGE	

Tabela 1: Instruções da Máquina Virtual

[3] O termo p-code foi utilizado primeiramente para definir uma arquitetura especializada para Pascal, e continua sendo usado para linguagens com características semelhantes

4.2 Instruções da Máquina Virtual

Segue-se uma descrição detalhada da ação de cada uma das instruções da Tabela 1.

1. **Operações com Literais**

   LOADC Carrega o acumulador com o valor do operando

   ADDC Soma o conteúdo do acumulador ao valor do operando

   SUBC Subtrai do conteúdo do acumulador o valor do operando

   MULTC Multiplica o conteúdo do acumulador pelo valor do operando

   DIVC Divide o conteúdo do acumulador pelo valor do operando

2. **Operações com Memória**

   LOAD Carrega o acumulador com o conteúdo do endereço indicado pelo operando

   STORE Armazena o conteúdo do acumulador no endereço indicado pelo operando

   ADD Soma o conteúdo do acumulador ao conteúdo do endereço indicado pelo operando

   SUB Subtrai do conteúdo do acumulador o conteúdo do endereço indicado pelo operando

   MULT Multiplica o conteúdo do acumulador pelo conteúdo do endereço indicado pelo operando

   DIV Divide o conteúdo do acumulador pelo conteúdo do endereço indicado pelo operando

3. **Transferência de Controle**

   JUMP Desvia para a posição do programa indicada pelo rótulo

   JUMPEQ Desvia para a posição do programa indicada pelo rótulo se o conteúdo do acumulador for igual a zero

   JUMPNE Desvia para a posição do programa indicada pelo rótulo se o conteúdo do acumulador for diferente de zero

   JUMPLT Desvia para a posição do programa indicada pelo rótulo se o conteúdo do acumulador for menor que zero

   JUMPGT Desvia para a posição do programa indicada pelo rótulo se o conteúdo do acumulador for maior que zero

   JUMPLE Desvia para a posição do programa indicada pelo rótulo se o conteúdo do acumulador for menor ou igual a zero

   JUMPGE Desvia para a posição do programa indicada pelo rótulo se o conteúdo do acumulador for maior ou igual a zero

4. **Entrada/Saída - Outras**

READ Lê do dispositivo de entrada e armazena no acumulador

WRITE Escreve o conteúdo do acumulador no dispositivo de saída

HALT Interrompe a execução do programa

BLOCK Reserva um número de posições de memória para alocação de variáveis

4.3 Linguagem a ser Compilada

A linguagem fonte utilizada neste trabalho é uma subconjunto da linguagem Pascal [Wirth 82], que será denominada CALTAL . Os comandos da linguagem CALTAL são: **for-to-do**, **repeat-until**, **if-then-else**, **while-do**, **read**, **write**, **begin-end**, além de alguns operadores aritméticos e de comparação para números inteiros. A linguagem admite um único escopo, simplificando deste modo o tratamento semântico feito pelo compilador.

A definição da gramática da linguagem CALTAL na notação BNF é:

```
<program> ::= program <identificador> ; <comando> .
<comando> ::= begin <comando> <resto_comando> |
              <identificador> := <expressao> |
              if <teste> then <comando> else <comando> |
              for <identificador> := <expressao> to <expressao>
                                     do <comando> |
              repeat <nucleo_repeat> until <teste> |
              read <identificador> |
              write <expressao> |
              while <teste> do <comando>
<resto_comando> ::= ; <comando> <resto_comando> | ; end | end
<nucleo_repeat> ::= <comando> |
                    <comando> ; <nucleo_repeat>
<teste> ::= <expressao> <op_comp> <expressao>
<expressao> ::= <expressao> <op_1> <termo> | <termo>
<termo> ::= <termo> <op_2> <fator> | <fator>
<fator> ::= <identificador> | <numero> | ( <expressao> )
<identificador> ::= <nome>
<op_1> ::= + | -
<op_2> ::= * | /
<op_comp> ::= < | <= | <> | > | >= | =
<numero> ::= <digito> <numero> | <digito>
<digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<identificador> ::= <letra> <nros_letras>
<letra> ::= a | b | c | d | e | f | g | h | i | j | k | l | m |
```

```
          n | o | p | q | r | s | t | u | v | w | x | y | z
<nros_letras> ::= <digito> <nros_letras> |
                  <letra> <nros_letras> |
                  λ [4]
```

A linguagem permite dois tipos de comentários: os cometários começando pelo símbolo %, que encerram-se ao final da linha, e os comentários delimitados por chaves ({ e }), que podem inclusive ser aninhados.

Pode-se observar que todos os comandos da linguagem são compostos por letras minúsculas. No programa, os comandos só poderão ser escritos utilizando letras maiúsculas se o analisador léxico transformar em letras minúsculas todas as letras maiúsculas.

[4] λ representa a cadeia vazia

4.4 Exemplos de Compilação de Programas CALIDAL

Seguem-se dois programas escritos na linguagem CALIDAL. O primeiro, **MediaAritmetica**, calcula a média aritmética de **N** números lidos. O segundo, **Expressao**, calcula o valor de uma expressão aritmética simples.

As Tabelas 2 e 3 mostram a tradução em linguagem de máquina produzida pelo compilador implementado. Deve-se observar que a coluna denominada identificador não faz parte da saída produzida pelo compilador.

```
Program MediaAritmetica;
{  Este programa calcula a media aritmetica entre n numeros }
Begin
  read(N);
  cont := 0; % teste de comentario } {{
  soma := 0;
  for i := 1 to N do begin
     read(valor);
     cont := cont + 1;
     soma := soma + valor;
  end;
  media := soma/cont;
  write(media);
End.
```

```
Program Expressao;
Begin
  read(X);
  read(Y);
  A := 4 * (X + y * 7 - 9)/(5 - X);
  write(3+A-x*(7+Y));
End
```

identificador	endereço	instrução	operando
	01	read	29
	02	loadc	0
	03	store	30
	04	loadc	0
	05	store	33
	06	loadc	1
	07	store	31
	08	load	31
	09	sub	29
	10	jumpgt	22
	11	read	34
	12	load	30
	13	addc	1
	14	store	30
	15	load	33
	16	add	34
	17	store	33
	18	load	31
	19	add	1
	20	store	31
	21	jump	8
	22	load	33
	23	div	30
	24	store	32
	25	load	32
	26	write	0
	27	halt	0
	28	block	6
n	29		
cont	30		
i	31		
media	32		
soma	33		
valor	34		

Tabela 2: p-code do programa MediaAritmetica

identificador	endereço	instrução	operando
	01	read	31
	02	read	33
	03	loadc	5
	04	sub	31
	05	store	34
	06	load	33
	07	multc	7
	08	subc	9
	09	store	35
	10	load	31
	11	add	35
	12	div	34
	13	store	34
	14	loadc	4
	15	mult	34
	16	store	32
	17	loadc	7
	18	add	33
	19	store	34
	20	load	31
	21	mult	34
	22	store	34
	23	load	32
	24	sub	34
	25	store	34
	26	loadc	3
	27	add	34
	28	write	0
	29	halt	0
	30	block	5
x	31		
a	32		
y	33		
temp0	34		
temp1	35		

Tabela 3: p-code do programa Expressao

5 Implementação

Na implementação Prolog a seguir, será usada a documentação Prolog padrão (+,-,?) para indicar a instanciação das variáveis dos programas no instante em que são ativados. O módulo principal do compilador é `compile/1`:

```
compile(Arq) :-
  abrir_arquivo(Arq,ArqE),
  lexico(ArqE,Tokens),              % Analisador Lexico
  sintatico(Tokens,Estrutura),      % Analisador Sintatico
  codifique(Estrutura,Codigo,Dicc), % Gerador de Codigo
  assembler(Codigo,Assembly,Dicc),  % Enderecamento
  imprime_codigo(Assembly,Dicc).    % Impressao de Codigo
```

Onde:
`Arq` unifica com o nome do arquivo onde se encontra o programa fonte a ser compilado;
o programa `lexico/2` lê o programa fonte armazenado em `Arq` e gera uma lista de átomos — `Tokens` — onde cada átomo corresponde a um ítem léxico do programa fonte;
o programa `sintatico/2` verifica se a sequência de átomos está de acordo com as regras de formação da linguagem especificada, produzindo uma árvore — `Estrutura` — que corresponde à estrutura sintática do programa;
o programa `codifique/3` manipula a árvore produzida por `sintatico/2` e gera o código correspondente a cada uma das estruturas que formam essa árvore. Todas as variáveis são armazenadas e relacionadas com endereços de memória em `Dicc`, que corresponde à tabela de símbolos;
os endereços são alocados às variáveis por `assembler/3` que, além de completar a tabela de símbolos, gera o código absoluto — `Assembly` — a partir do código relocável — `Codigo`.

A seguir, cada um desses programas é explicado em detalhes.

5.1 Analisador Léxico

O analisador léxico implementado lê um programa armazenado em disco e monta a lista de átomos correspondente a esse programa. Quando uma cadeia de caracteres é lida, todos os caracteres maiúsculos são transformados no caracter minúsculo correspondente. Deste modo, as cadeias *Bolacha, BOLACHA, bolacha, bOLacHA* e *boLAclIa* serão todas transformadas no átomo *bolacha.*

Em todos os programas explicados a seguir, o nome de variável `Arquivo`, quando for o primeiro argumento de um predicado, refere-se ao *handle* do arquivo onde se encontra armazenado o texto de entrada — programa fonte a ser compilado.

O analisador léxico é definido pelo seguinte programa `lexico/2`:

```
lexico(+Arquivo,-Lista_de_atomos)
```

onde `Lista_de_atomos` é a lista dos átomos correspondentes ao texto de entrada.

O programa `lexico/2` está definido por uma única cláusula:

```
lexico(Arquivo,[Palavra|Tokens]) :-
  le_caracter(Arquivo,Caracter),
  leia_palavra(Arquivo,Caracter,Palavra,UltCar),
  tokenize(Arquivo,Palavra,UltCar,Tokens).
```

O programa `le_caracter/2` possui dois argumentos :

`le_caracter(+Arquivo,-Caracter)`

onde `Caracter` é uma estrutura correspondente ao caracter lido do texto fonte. Essa estrutura é montada pelo programa `identifica_caracter/2`, explicado adiante, e pode ser uma das quatro estruturas descritas a seguir, de acordo com o caracter lido. Em todos os casos, `Cod` é o código ASCII correspondente ao caracter lido. Os possíveis valores que `Caracter` pode assumir são:

`identificador(Tipo,Cod)`: letras e números. `Tipo` identifica se é uma letra ou um número;

`operador(Cod)`: operadores e

`comentario(InicTerm,Tipo)`: delimitadores de comentários. `InicTerm` pode assumir os valores `inicio` ou `termino`, enquanto os valores que `Tipo` pode assumir são `tipo_linha` ou `tipo_abre_fecha`.

O programa `le_caracter/2` está definido por uma única cláusula:

```
le_caracter(Arquivo,T) :-
  get0(Arquivo,C),
  name(X,[C]),write(X),
  identifica_caracter(C,T),
  !.
```

O programa `identifica_caracter/2` possui dois argumentos :

`identifica_caracter(+Codigo,-Estrutura)`

onde `Codigo` é o código ASCII do caracter lido e `Estrutura` unifica com a estrutura que identifica esse caracter. Consta de nove cláusulas.

A primeira cláusula:

```
identifica_caracter(C,identificador(letra,C)) :-
  C >= 'a,
  C =< 'z.
```

verifica se o caracter é uma letra minúscula.

A segunda cláusula

```
identifica_caracter(C,identificador(letra,L)) :-
  C >= 'A,
  C =< 'Z,
  L is C + 32.
```

verifica se o caracter lido é uma letra maiúscula e, neste caso, troca-o pela letra minúscula correspondente.

A terceira cláusula

```
identifica_caracter(C,identificador(numero,C)) :-
  C >= '0,
  C =< '9.
```

verifica se o caracter lido é um número, enquanto que a quarta cláusula

```
identifica_caracter(C,operador(C)) :-
  C >= '(,
  C =< '>.
```

verifica se o caracter lido é um operador.

As quatro cláusulas seguintes verificam se o caracter lido é um delimitador de comentário:

```
identifica_caracter('{,comentario(inicio,tipo_abre_fecha)).

identifica_caracter('},comentario(termino,tipo_abre_fecha)).

identifica_caracter('%,comentario(inicio,tipo_linha)).

identifica_caracter(10,comentario(termino,tipo_linha)).
```

Na cláusula anterior, 10 é o código ASCII correspondente ao final de linha, que é o caracter que fecha o comentário tipo linha.

Finalmente, a última cláusula é utilizada para os caracteres que não se enquadram em nenhum dos casos acima:

```
identifica_caracter(_,separador).
```

O programa `leia_palavra/4` possui quatro argumentos, como mostrado a seguir:

`leia_palavra(+Arquivo,+PrimCarac,-Palavra,-UltCar)`

Ela unifica `Palavra` com uma palavra lida do texto de entrada e `UltCar` com o último caracter lido após essa palavra.

O programa `leia_palavra/4` está definido por seis cláusulas, que são ativadas de acordo com o valor de `PrimCarac`, que corresponde ao primeiro caracter da palavra a ser lida.

A primeira cláusula

```
leia_palavra(Arquivo,comentario(inicio,Tipo),Palavra,UltCar) :-
  le_caracter(Arquivo,Caracter),
  termina_documentacao(Arquivo,Tipo,Caracter,0),
  le_caracter(Arquivo,Caracter),
  leia_palavra(Arquivo,Caracter,Palavra,UltCar).
```

verifica se o caracter lido é um início de comentário. Neste caso, deverá ignorar todos os caracteres subsequentes até encontrar o final do comentário (ver `termina_documentacao/4`).

A segunda cláusula

```
leia_palavra(Arquivo,operador(C),Simbolo,UltCar) :-
  pode_compor(C),!,
  le_caracter(Arquivo,Car),
  compoe_simbolo(Arquivo,C,Car,Simbolo,UltCar).
```

verifica se o caracter é um operador e se ele pode ser composto (`pode_compor/1`). Neste caso, um novo caracter deverá ser lido e, a seguir, será verificado se eles formam um operador composto (`compoe_simbolo/5`).

A terceira cláusula

```
leia_palavra(Arquivo,operador(C),Simbolo,UltCar) :-
  name(Simbolo,[C]),
  le_caracter(Arquivo,UltCar).
```

será ativada quando `pode_compor/2` falhar, ou seja, quando o caracter for um operador que não pode ser composto.

A quarta cláusula

```
leia_palavra(Arquivo,identificador(letra,C),Palavra,UltCar) :-
  le_caracter(Arquivo,Caracter),
  le_identificador(Arquivo,Caracter,Lista,UltCar),
  name(Palavra,[C|Lista]).
```

será ativada se o primeiro caracter da palavra for uma letra. Neste caso, irá ler os caracteres seguintes até encontrar algum que não seja classificado como identificador, ou seja, diferente de uma letra ou um número. Isto é feito pelo programa `le_identificador/4`.

A quinta cláusula

```
leia_palavra(Arquivo,identificador(numero,C),Numeral,UltCar) :-
  le_caracter(Arquivo,Caracter),
  le_numero(Arquivo,Caracter,Lista,UltCar),
  name(Numeral,[C|Lista]).
```

será utilizada quando o primeiro caracter é um dígito. Neste caso, será ativado o programa `le_numero/4`, que lê os próximos caracteres até encontrar algum que não seja um dígito.

A sexta cláusula

```
leia_palavra(Arquivo,separador,Palavra,UltCar) :-
  le_caracter(Arquivo,Caracter),
  leia_palavra(Arquivo,Caracter,Palavra,UltCar).
```

ignora o caracter lido, lê o próximo e faz a chamada recursiva.

O programa `termina_documentacao/4` é ativado quando for encontrado um delimitador de início de comentário, ou seja, `'%'` ou `'{'` . Ele deverá ignorar todos os caracteres até encontrar o fim dos comentários. Ele possui quatro argumentos:

`termina_documentacao(+Arquivo,+Tipo,+Caracter,+N)`

onde
`Tipo` poderá unificar com os valores `tipo_abre_fecha` ou `tipo_linha`
`Caracter` indica o último caracter lido
`N` indica o nível de aninhamento de comentários (número de chaves abertas). Seu valor é importante somente nos comentários tipo abre-fecha. Nos comentários tipo linha seu valor será sempre zero.

O programa `termina_documentacao/4` está definido por quatro cláusulas. A primeira delas

```
termina_documentacao(Arquivo,Tipo,comentario(termino,Tipo),0) :- !.
```

indica que o caracter lido é o fecha-comentário e só há um delimitador de comentário a ser fechado.

A segunda cláusula

```
termina_documentacao(Arquivo,tipo_abre_fecha,
                     comentario(inicio,tipo_abre_fecha),N) :-
  N1 is N + 1,!,
  le_caracter(Arquivo,Caracter),
  termina_documentacao(Arquivo,tipo_abre_fecha,Caracter,N1).
```

é utilizada quando o comentário for inicializado com '{' e um outro caracter idêntico é encontrado agora. Neste caso, deve-se aumentar o valor de N para indicar que um delimitador a mais deverá ser fechado.

A terceira cláusula

```
termina_documentacao(Arquivo,Tipo,comentario(termino,Tipo),N) :-
  N1 is N - 1,!,
  le_caracter(Arquivo,Caracter),
  termina_documentacao(Arquivo,Tipo,Caracter,N1).
```

é para o caso semelhante ao anterior quando o caracter lido mais recentemente for '}'. Neste caso, o valor de N deve ser decrementado para indicar que um dos delimitadores abertos acaba de ser fechado.

A quarta cláusula

```
termina_documentacao(Arquivo,Tipo,_,N) :-
  le_caracter(Arquivo,C),
  termina_documentacao(Arquivo,Tipo,C,N).
```

indica que foi lido um outro caracter qualquer e, portanto, deve ser ignorado.

A saída produzida pelo analisador léxico para os exemplos de programas escritos na linguagem CELTAL da página 16 é a seguinte:

MEDIA ARITMETICA

```
[program,mediaaritmetica,;,begin, read,(,n,),;, cont,:=,0,;,
 soma,:=,0,;,for,i,:=,1,to,n,do,begin,read,(,valor,),;,cont,
 :=,cont,+,1,;,soma,:=,soma,+,valor,;,end,;,media,:=,soma,/,
 cont,;,write,(,media,),;,end,.]
```

EXPRESSAO

```
[program,expressao,;,begin,read,(,x,),;,read,(,y,),;,a,:=,4,
*,(,x,+,y,*,7,-,9,),/,(,5,-,x,),;,write,(,3,+,a,-,x,*,(,7,+,
y,),),;,end,.]
```

5.2 Analisador Sintático

Uma importante aplicação da linguagem Prolog é a implementação de analisadores sintáticos. Na verdade, a linguagem Prolog surgiu como uma tentativa de usar lógica para expressar regras gramaticais e formalizar o processo de análise sintática.

A técnica construída dentro de Prolog é a de gramáticas de cláusulas definidas, que são uma generalização das gramáticas livres de contexto. Devido à sua importância, esta técnica será discutida a seguir com mais detalhes.

5.2.1 Notação de Regras Gramaticais

A Notação de Regras Gramaticais — NRG — está presente em diversas implementações da linguagem Prolog. Ela foi desenvolvida como um auxílio para a implementação de analisadores de sentenças, utilizando gramáticas de cláusulas definidas. Na verdade, as cláusulas escritas em NRG são uma variação notacional de uma classe de programas Prolog, que utilizam uma notação mais concisa para esconder as informações de menor interesse. Deste modo, o código torna-se mais fácil de ser lido.

A sintaxe da NRG é similar à notação BNF. O nome de um elemento da linguagem sendo definida deve estar presente no lado esquerdo da definição. O símbolo ::= é substituído pelo símbolo -->. A definição, que fica do lado direito da expressão, é uma sequência de um ou mais elementos separados por vírgulas. Um elemento da linguagem é um símbolo não terminal ou uma sequência de símbolos terminais.

Os símbolos não terminais são escritos como átomos Prolog; os símbolos terminais são listas de átomos. Isto é feito para facilitar a tradução das gramáticas para programas Prolog.

O exemplo apresentado a seguir mostra a implementação de um pequeno subconjunto da gramática da língua portuguesa, implementado utilizando NRG.

```
sentenca --> sintagma_nominal,sintagma_verbal.

sintagma_nominal --> artigo,nome.

sintagma_verbal --> verbo,sintagma_nominal.
sintagma_verbal --> verbo.

artigo --> [o].
artigo --> [os].
artigo --> [a].
artigo --> [as].

nome --> [garotinho].
nome --> [abacate].
```

```
nome --> [batatas].

verbo --> [come].
verbo --> [comem].
verbo --> [canta].
verbo --> [cantam].
```

Algumas sentenças que pertencem à linguagem definida por esta gramática são:

o garotinho come o abacate
o garotinho canta
o abacate canta o garotinho
as garotinho comem o batatas
a abacate cantam os garotinho

5.2.2 Adicionando Argumentos Extras

As gramáticas livres de contexto podem ser estendidas adicionando-se argumentos extras aos símbolos não terminais. Este recurso permite que as diversas partes da linguagem possam trocar algumas informações. No exemplo anterior, poderia ser indicada a concordância em gênero e número da seguinte forma:

```
sentenca --> sintagma_nominal(G,N),sintagma_verbal(N).

sintagma_nominal(G,N) --> artigo(G,N),nome(G,N).

sintagma_verbal(N) --> verbo(N),sintagma_nominal(Gn,Nn).
sintagma_verbal(N) --> verbo(N).

artigo(masculino,singular) --> [o].
artigo(masculino,plural) --> [os].
artigo(feminino,singular) --> [a].
artigo(feminino,plural) --> [as].

nome(masculino,singular) --> [garotinho].
nome(masculino,singular) --> [abacate].
nome(feminino,plural) --> [batatas].

verbo(singular) --> [come].
verbo(plural) --> [comem].
verbo(singular) --> [canta].
verbo(plural) --> [cantam].
```

Algumas sentenças que pertencem à linguagem definida por esta gramática são:

o garotinho come o abacate
o garotinho canta
o abacate canta o garotinho
o garotinho come as batatas

Deve ser observado que as sentenças que pertenciam à linguagem definida anteriormente, sem a devida concordância em gênero e número, não são mais reconhecidas. Isto mostra a importância da adição dos argumentos extras aos símbolos não terminais. Com esta extensão das gramáticas livres de contexto, tem-se uma nova classe de gramáticas: as Gramáticas de Cláusulas Definidas — GCD. A NRG é utilizada para implementar em Prolog analisadores sintáticos para as linguagens que podem ser definidas por GCD.

5.2.3 Conversão da Notação de Regras Gramaticais para Cláusulas Prolog

O símbolo --> é um operador Prolog padrão, definido em Arity Prolog como:

```
op(600,xfy,-->).
```

Na transformação das cláusulas em NRG para cláusulas Prolog, dois argumentos são adicionados à cláusula. Caso haja a necessidade da inclusão de cláusulas Prolog dentro de cláusulas em NRG, elas deverão estar entre chaves .

A seguir estão alguns exemplos de transformação de cláusulas em NRG para a notação Prolog:

```
a) sintagma_nominal --> artigo,nome.

   sintagma_nominal(S0,S) :- artigo(S0,S1),nome(S1,S).

b) verbo --> [canta].

   verbo([canta|S],S).

c) elem(A,B,C) --> transf(A,B),verif(A,B,C).

   elem(A,B,C,S0,S) :- transf(A,B,S0,S1),
                       verif(A,B,C,S1,S).
```

```
d) elem(A,B,C) --> [a],transf(A,B),[bola],verif(B,C),[milho].

   elem(A,B,C,[a|S0],S) :- transf(A,B,S0,[bola|S1]),
                           verif(B,C,S1,[milho|S]).

e) teste --> legal(X), {joia(X,Y)} , puxa(Y).

   teste(S0,S) :- legal(X,S0,S1),
                  joia(X,Y),
                  puxa(Y,S1,S).
```

5.2.4 Montagem da Árvore de Derivação

Outra importante aplicação das GCD na análise de sentenças é na montagem da árvore de derivação sintática da sentença. Como exemplo, considere-se, juntamente com a gramática definida na seção 5.2.1, a seguinte sentença:

"o garotinho come o abacate"

O analisador tenta reconhecer se ela é uma sentença. Para isto, ele verifica se é possível reconhecer um sintagma nominal seguido de um sintagma verbal.

```
sentenca --> sintagma_nominal,sintagma_verbal.
```

Para reconhecer um sintagma nominal, o analisador tenta reconhecer um artigo seguido de um nome.

```
sintagma_nominal --> artigo,nome.
```

O analisador reconhece então o artigo "o".

```
artigo --> [o].
```

Deste modo, este elemento é consumido, restando ao analisador reconhecer

"garotinho come o abacate".

A seguir, é reconhecida a palavra "garotinho" como um nome.

```
nome --> [garotinho].
```

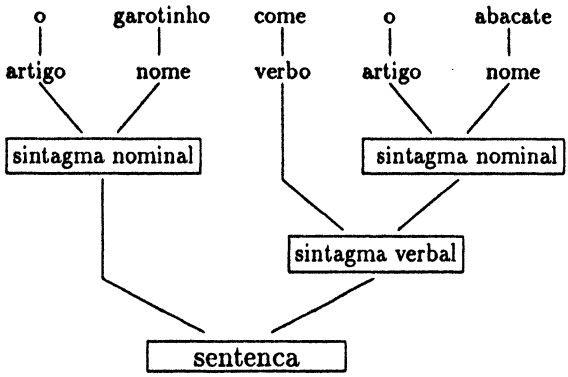


Figura 3: Árvore de derivação da sentença "o garotinho come o abacate"

Deste modo, o analisador conseguiu reconhecer um sintagma nominal, consumindo parte da sentença de entrada. Resta-lhe agora tentar consumir "come o abacate" e, para isto, deverá reconhecer um sintagma verbal.

Trabalhando do mesmo modo, o analisador reconhece "come" como sendo um verbo, "o" como um artigo e "abacate" como nome, reconhecendo a combinação "o abacate" como um sintagma nominal e "come o abacate" como um sintagma verbal.

Para construir a árvore de derivação, argumentos que representam as partes da árvore de derivação devem ser adicionados aos predicados do programa. Em geral, estes argumentos são estruturas Prolog.

Por exemplo, a primeira cláusula do programa na seção 5.2.1 pode ser estendida usando o termo composto `sent(SN,SV)` que representa a árvore de derivação. `SN` e `SV` representam, respectivamente, a derivação correspondente a `sintagma_nominal` e `sintagma_verbal`. Os outros predicados, que representam outras partes da sentença, devem também ser trocados, como mostrado a seguir:

```
sentenca(sent(SN,SV)) --> sintagma_nominal(SN),
                          sintagma_verbal(SV).

sintagma_nominal(sn(A,N)) --> artigo(A),nome(N).

sintagma_verbal(sv(V,SN)) --> verbo(V),sintagma_nominal(SN).
sintagma_verbal(sv(V)) --> verbo(V).

artigo(art) --> [o].
```

```
artigo(art) --> [os].
artigo(art) --> [a].
artigo(art) --> [as].
nome(nome(garoto)) --> [garotinho].
nome(nome(fruta)) --> [abacate].
nome(nome(legume)) --> [batatas].

verbo(verbo(comer)) --> [come].
verbo(verbo(comer)) --> [comem].
verbo(verbo(cantar)) --> [canta].
verbo(verbo(cantar)) --> [cantam].
```

Ao se perguntar

```
?- sentenca(X,[o,garotinho,come,o,abacate],[]).
```

tem-se como resposta

```
X = sent(sn(art,nome(garoto)),
sv(verbo(comer),sn(art,nome(fruta))))
```

Pode-se observar que esta é uma estrutura que representa a árvore de derivação mostrada na Figura 3.

5.2.5 Implementação do Analisador Sintático

O programa com a implementação do analisador sintático para a linguagem CLIPL é apresentado a seguir. Ele está baseado no programa apresentado por Sterling e Shapiro [Sterling 87].

A primeira cláusula do programa identifica a estrutura básica de um programa escrito em CLIPL , que consiste na palavra `program` seguida do nome do programa, de um ponto e vírgula e do comando que corresponde ao corpo do programa. Esse comando pode ser composto por vários outros comandos. Uma palavra que deve estar presente porque faz parte da gramática, tal como `program`, é chamada *palavra reservada.*

```
programa(E) --> [program],id(X),[';'],comando(E),['.'].
```

A estrutura que o analisador sintático retorna é a cláusula com a estrutura correspondente ao corpo do programa, pois o restante é irrelevante para o gerador de código.

A primeira cláusula de `comando` considera a existência de um bloco contendo vários comandos. Os comandos são delimitados pelas palavras reservadas `begin` e `end`.

```
comando((C ; Cresto)) --> [begin],comando(C),
                          resto_comando(Cresto).
```

Esta primeira cláusula reconhece a palavra reservada **begin** seguida pelo primeiro comando, cuja estrutura unificará com C, e pelos comandos restantes do bloco, cuja estrutura deve unificar com Cresto. A estrutura final que deve retornar é (C;Cresto), que é a composição das estruturas C e Cresto separadas pelo operador infixo ';'. Como tanto C quanto Cresto podem ser comandos compostos, ou mesmo conter comandos compostos, a estrutura que retorna do analisador sintático é montada recursivamente. Isto é feito pela cláusula resto_comando. O átomo vazio é usado para marcar o fim do bloco na estrutura de saída do analisador sintático.

```
resto_comando(fim) --> [end];[';',end].
resto_comando((C;Cresto)) --> [';'],comando(C),
                              resto_comando(Cresto).
```

A próxima cláusula é a que reconhece o comando de atribuição, que é formado por um identificador seguido do operador := e de uma expressão aritmética.

```
comando(atribua(X,Expr)) --> id(X),[':='],expressao(Expr).
```

A estrutura de retorno tem a forma atribua(X,Expr), onde a variável Expr representa a estrutura da expressão aritmética, enquanto X representa o nome da variável à qual será atribuída o valor da expressão. As expressões são identificadas pelas cláusulas a seguir:

```
expressao(E) --> termo(T),resto_expressao(E,T).
resto_expressao(E,E1) --> op1(Op),termo(T),
                          resto_expressao(E,expr(Op,E1,T)).
resto_expressao(E,E) --> [].
termo(T) --> fator(F),resto_termo(T,F).
resto_termo(T,T1) --> op2(Op),fator(F),
                      resto_termo(T,expr(Op,T1,F)).
resto_termo(T,T) --> [].

fator(E) --> ['('],expressao(E),[')'].
fator(F) --> identificador(F).

op1('-') --> ['-'].
op1('+') --> ['+'].
op2('*') --> ['*'].
op2('/') --> ['/'].

identificador(X) --> id(X).
```

```
identificador(X) --> inteiro(X).

id(nome(X)) --> [X],{atom(X)}.
inteiro(numero(X)) --> [X],{integer(X)}.
```

Pode-se observar que a estratégia utilizada na montagem da estrutura, no caso das expressões, diverge um pouco do caso geral. Isto é necessário para que a estrutura respeite a associatividade de operação da esquerda para a direita, no caso de operadores de mesma prioridade. Deste modo, uma expressão do tipo

$$4 - 5 - 7$$

terá sua estrutura representada como

```
expr(-,expr(-,4,5),7)
```

enquanto que, se fosse utilizada a mesma estratégia das outras cláusulas do analisador sintático, a estrutura seria

```
expr(-,4,expr(-,5,7))
```

A próxima cláusula corresponde ao condicional *if-then-else*. A estrutura montada é `condicao(Teste,C1,C2)`, onde `Teste` é a estrutura correspondente à expressão lógica que representa a condição a ser testada. `C1` é a estrutura referente ao comando que deve ser executado quando a condição for satisfeita, e `C2` corresponde ao comando que deve ser executado quando a condição não for satisfeita.

```
comando(condicao(Teste,C1,C2)) --> [if],teste(Teste),
                                   [then],comando(C1),
                                   [else],comando(C2).
```

A cláusula `teste` consiste de duas expressões aritméticas e um operador de comparação entre elas. A estrutura de retorno tem a forma `teste(OP,E1,E2)`, onde `E1` e `E2` são as estruturas relativas às expressões e `OP` é o operador utilizado.

```
teste(teste(OP,E1,E2)) --> expressao(E1),
                           comparacao(OP),
                           expressao(E2).

comparacao('=') --> ['='].
comparacao('<') --> ['<'].
comparacao('>') --> ['>'].
comparacao('>=') --> ['>='].
comparacao('<=') --> ['<='].
comparacao('<>') --> ['<>'].
```

A cláusula a seguir reconhece a estrutura de repetição *while-do*, que consiste de um teste e um comando que será executado enquanto o resultado do teste for verdadeiro. A estrutura montada é `enquanto(Teste,C)`, onde `C` é a estrutura do comando e `Teste` é a estrutura do teste.

```
comando(enquanto(Teste,C)) --> [while],teste(Teste),
                               [do],comando(C).
```

Outra importante estrutura de repetição da linguagem *CALTAL* é *repeat-until*.

```
comando(repita(C,Teste)) --> [repeat],nucleo_repeat(C),
                             [until],teste(Teste).
```

A estrutura montada é `repita(C,Teste)`, onde `C` é a estrutura referente a um comando ou a uma sequência de comandos. Como os comandos existentes entre *repeat* e *until* não precisam estar delimitados por *begin-end*, é necessário a cláusula `nucleo_repeat`:

```
nucleo_repeat(C) --> comando(C).
nucleo_repeat((C;Cresto)) --> comando(C),[';'],
                              nucleo_repeat(Cresto).
```

A última estrutura de repetição é o comando *for-to-do*.

```
comando(para(J,Inicio,Fim,C)) --> [for],id(J),
                                  [':='],expressao(Inicio),
                                  [to],expressao(Fim),
                                  [do],comando(C).
```

A estrutura que é montada é `para(J,Inicio,Fim,C)`, onde `J` corresponde à variável que controla o loop, `Inicio` é a estrutura que corresponde à expressão que indica o valor inicial de `J`, `Fim` é uma estrutura que corresponde à expressão cujo valor `J` deve atingir para encerrar o loop, e `C` é a estrutura equivalente ao comando que deve ser executado a cada iteração.

Os últimos comandos são referentes às operações de entrada e saída:

```
comando(leia(X)) --> [read],['('],id(X),[')'].
comando(escreva(X)) --> [write],['('],expressao(X),[')'].
```

A seção 6.3, na página 48, apresenta a listagem completa do analisador sintático. Deve ser observado que, não se levando em conta os argumentos utilizados para montar a estrutura que retorna do analisador sintático, a implementação é muito semelhante à especificação da linguagem *CALTAL* definida na seção 4.3.

As estruturas produzidas pelo analisador sintático, após manipular as listas de átomos geradas pelo analisador léxico para os programas `MediaAritmetica` e `Expressao` — página 24 — são, respectivamente:

MEDIA ARITMETICA

```
(leia(n)  ; atribua(cont,numero(0))  ; atribua(soma,numero(0)) ;
 para(nome(i),numero(1),nome(n) , (leia(valor)  ;  atribua(cont,
 expr(+,nome(cont),numero(1))) ; atribua(soma,expr(+,nome(soma),
 nome(valor)))  ; fim))  ; atribua(media,expr(/,nome(soma),nome(
 cont))) ; escreva(nome(media)) ; fim)
```

EXPRESSAO

```
(leia(x) ; leia(y) ; atribua(a, expr(/,expr(*,numero(4),
 expr(- , expr(+,nome(x) , expr(*, nome(y), numero(7))),
 numero(9))) , expr(- , numero(5), nome(x)))) ; escreva(
 expr(-,expr(+,numero(3),nome(a)),expr(*,nome(x),expr(+,
 numero(7),nome(y))))) ; fim)
```

5.3 Gerador de Código

5.3.1 O Codificador

Concluída a etapa da análise sintática, tem-se como saída do programa `sintatico/2` uma lista de comandos, separados por ponto e vírgula, pronta para ser manipulada pelo gerador de código implementado pelo programa `codifique/3`. O código gerado por este programa será descrito para as várias estruturas produzidas pelo analisador sintático, representando os vários comandos da linguagem fonte.

As variáveis utilizadas no programa fonte — bem como as variáveis temporárias criadas para resolução de expressões complexas — devem ser armazenadas e relacionadas com endereços de memória. Para representar a tabela de símbolos que armazena estas relações foi utilizada uma árvore de busca binária, denominada *dicionário*, e detalhada mais adiante. Por enquanto basta saber que o predicado `procure(Nome,Dicc,Ende)` acessa a estrutura dicionário `Dicc` tanto para recuperar o endereço `Ende` da variável `Nome` quanto para inserir a variável `Nome`, caso ela ainda não se encontre no dicionário.

Os três parâmetros do programa `codifique/3` representam, respectivamente, a estrutura de saída do analisador sintático, o código relocável gerado e a tabela de símbolos *Dicc*. A primeira cláusula de `codifique/3` se encarrega de codificar a saída do analisador sintático quando esta não é um único comando, mas uma lista de comandos separados pelo operador ';'.

```
codifique((E;Resto),(E1;Resto1),Dicc) :-
    codifique(E,E1,Dicc),!,
    codifique(Resto,Resto1,Dicc).
```

A saída do analisador sintático para o comando de atribuição

< Nome > := < Expressao >

será

```
(atribua(nome(Nome),Expressao)).
```

A segunda cláusula de codifique mostra que a tradução adequada para esta estrutura é primeiro codificar Expressao, cujo resultado ficará no acumulador, para depois armazenar este valor no endereço da variável Nome. Este endereço é obtido do dicionário Dicc pelo programa procure/3.

```
codifique(atribua(Nome,Expressao),(CodExpr;instr(store,Endereco)),Dicc):-
    procure(Nome,Endereco,Dicc),
    cod_expressao(Expressao,0,CodExpr,Dicc).
```

A codificação de Expressao é feita por cod_expressao/4, definido a seguir. O primeiro parâmetro é a expressão gerada pelo analisador sintático e o terceiro é a expressão codificada. O quarto parâmetro é o dicionário *Dicionário* e o segundo parâmetro controla o uso das variáveis temporárias.

Se a expressão for somente um nome ou um número, a codificação é direta.

```
cod_expressao(numero(X),_,instr(loadc,X),_).
cod_expressao(nome(X),_,instr(load,End),Dicc) :-
    procure(X,End,Dicc).
```

Se, na expressão expr(OP,E1,E2), E2 é uma instrução simples, ou seja, um nome ou um número, então basta codificar E1 e E2 separadamente; a codificação final é (CodE1;CodE2), que são as codificações de E1 e E2, respectivamente.

```
cod_expressao(expr(OP,E1,E2),N,(CodE1;Instr),Dicc) :-
   instr_simples(OP,E2,Instr,Dicc),
   cod_expressao(E1,N,CodE1,Dicc).

instr_simples(OP,numero(N),instr(OpCod,N),Dicc) :-
   operador_memoria(OP,OpCod).
```

```
instr_simples(OP,nome(N),instr(OpCod,Endereco),Dicc) :-
   procure(N,Endereco,Dicc),
   operador_literal(OP,OpCod).

operador_memoria('+',addc).
operador_memoria('-',subc).
operador_memoria('*',multc).
operador_memoria('/',divc).

operador_literal('+',add).
operador_literal('-',sub).
operador_literal('*',mult).
operador_literal('/',div).
```

Se, na expressão `expr(OP,E1,E2)`, `E2` é uma instrução complexa, ou seja, é uma expressão contendo um operador com seus argumentos, é necessário utilizar variáveis temporárias. A variável `Endereco` corresponde ao endereço da variável temporária, que será utilizado para armazenar resultados intermediários. O código final é

```
(CodE2;instr(store,Endereco);CodE1;instr(OpCod,Endereco))
```

onde `CodE1` e `CodE2` correspondem aos códigos de `E1` e `E2` respectivamente e `OpCod` é a instrução correspondente ao operador `OP`. Este código significa que o resultado gerado pela execução de `CodE2` será armazenado em `Endereco`; a seguir será executado `CodE1` e o resultado será um elemento da operação, enquanto que `Endereco` será o outro.

```
cod_expressao(expr(OP,E1,E2),N,(CodE2;instr(store,Endereco);CodE1;
      instr(OpCod,Endereco)),Dicc) :-
   complexa(E2),
   procure(temp(N),Endereco,Dicc),
   cod_expressao(E2,N,CodE2,Dicc),
   N1 is N + 1,
   cod_expressao(E1,N1,CodE1,Dicc),
   operador_literal(OP,OpCod).

complexa(expr(_,_,_)).
```

O comando condicional if-then-else tem como saída do analisador sintático a estrutura

```
(condicao(Teste,ComEntao,ComSenao)).
```

Para compilar a estrutura, é necessário introduzir rótulos nos lugares para onde as instruções podem dar um salto — `JUMP`. Neste caso, precisa-se de um rótulo no início do

Senao e outro no final dele. Os rótulos tem a forma `label(N)` onde `N` é o endereço da instrução. O valor de `N` é preenchido durante a etapa de endereçamento, quando o rótulo é então removido. O esquema do código gerado é dado pelo terceiro argumento da seguinte cláusula de `codifique/3`:

```
codifique(condicao(Teste,ComEntao,ComSenao),(CodTeste;CodEntao;
        instr(jump,L2);label(L1);CodSenao;label(L2)),Dicc) :-
    codifique_teste(Teste,L1,CodTeste,Dicc),
    codifique(ComEntao,CodEntao,Dicc),
    codifique(ComSenao,CodSenao,Dicc).
```

Para comparar duas expressões aritméticas, subtrai-se a segunda da primeira e realiza-se a operação de JUMP apropriada com o operador de comparação testado. Por exemplo, caso o teste seja verificar se duas expressões são iguais, é gerado o JUMP quando o resultado da subtração for diferente de 0 (zero). Deve ser observado que o rótulo, ou seja, o endereço do código para onde será feito o desvio, é o segundo argumento de `codifique_teste/4`.

```
codifique_teste(teste(OP,Exp1,Exp2),Label,
        (Codigo;instr(CodOp,Label)),Dicc) :-
    cod_expressao(expr('-',Exp1,Exp2),0,Codigo,Dicc),
    operador_logico(OP,CodOp).

operador_logico('=',jumpne).
operador_logico('<>',jumpeq).
operador_logico('>',jumple).
operador_logico('>=',jumplt).
operador_logico('<',jumpge).
operador_logico('<=',jumpgt).
```

É analisada a seguir a compilação do comando *for-to-do*. A saída do analisador sintático para:

```
for i := <inic> to <fim> do <comandos>;
```

é a estrutura

```
(para(nome(I),Inicio,Fim,Comando))
```

onde `I` é uma variável, `Inicio` e `Fim` são expressões aritméticas e `Comando` é uma lista de um ou mais comandos. Numa pseudo-linguagem de máquina isto poderia ser traduzido como:

```
        Calcule Inicio
        I <- Inicio
L1 :    Calcule Fim
        Se I > Fim jump L2
        Execute Comando
        I <- I + 1
        jump L1
L2 :
```

Na linguagem da máquina virtual definida na seção 4.2, este comando é traduzido no terceiro parâmetro da seguinte cláusula do programa `codifique/3`:

```
codifique(para(nome(I),Inicio,Fim,Comando),(CodInic;label(L1);
        CodTeste;CodComando;instr(load,End_I);
        instr(add,1);instr(store,End_I);instr(jump,L1);
        label(L2)),Dicc) :-
    codifique(atribua(I,Inicio),CodInic,Dicc),
    procure(I,End_I,Dicc),
    codifique_teste(teste('<=',nome(I),Fim),L2,CodTeste,Dicc),
    codifique(Comando,CodComando,Dicc).
```

Deve ser observado que, como o teste não está explícito no comando *for*, o primeiro parâmetro da cláusula `codifique_teste/4` já deve estar instanciado. Além disso, como o desvio é realizado quando o teste falha na verificação de `I > Fim`, o operador de comparação utilizado é o menor ou igual.

De forma análoga são codificados os comandos *while-do* e *repeat-until*, enquanto a codificação do *read* e *write* é quase direta.

```
codifique(enquanto(Teste,Comando),(label(L1);CodTeste;CodComando;
        instr(jump,L1);label(L2)),Dicc) :-
    codifique_teste(Teste,L2,CodTeste,Dicc),
    codifique(Comando,CodComando,Dicc).

codifique(repita(Comando,Teste),(label(L1);CodComando;CodTeste),Dicc) :-
    codifique(Comando,CodComando,Dicc),
    codifique_teste(Teste,L1,CodTeste,Dicc).


codifique(leia(X),instr(read,Endereco),Dicc) :-
    procure(X,Endereco,Dicc).

codifique(escreva(Expressao),(CodExpr;instr(write,0)),Dicc) :-
    cod_expressao(Expressao,0,CodExpr,Dicc).
```

5.3.2 O Dicionário

Como já mencionado, a estrutura da tabela de símbolos utilizada é a estrutura de árvore binária de busca [Wirth 86] chamada dicionário. Nesse dicionário, o compilador implementado relaciona variáveis com endereços de memória e rótulos com endereços de instruções.

Quando a primeira variável é encontrada ela é armazenada na raiz do dicionário. Se a segunda variável for lexicograficamente menor que a primeira, ela será armazenada como descendente direto à esquerda da raiz e, se for maior, será armazenada à direita. Assim, para cada nó, a subárvore esquerda só contém variáveis lexicograficamente menores que ele próprio e a subárvore direita só variáveis maiores.

O dicionário é utilizado pelo programa `assembler/3` para encontrar endereços correspondentes a rótulos e variáveis. Ele é implementado em Prolog de uma forma muito simples e elegante, mediante a utilização de uma estrutura de dados incompleta. Este tipo de estrutura permite tanto inserir novas variáveis no dicionário como procurar por variáveis já armazenadas no dicionário [Sterling 87].

O dicionário é representado pela seguinte estrutura

```
dicc( Nome, Endereco, D_esq, D_dir)
```

onde `D_esq` e `D_dir` representam, respectivamente, os subdicionários à esquerda e à direita, enquanto `Nome` é uma variável armazenada no endereço `Endereco`. A figura abaixo mostra um diagrama correspondente a essa estrutura:

```
<Nome,Endereco>
   /       \
D_esq     D_dir
```

O dicionário é construído ao longo do processo de codificação, começando a partir de uma variável livre. Todos os nós terminais do dicionário, a cada momento, são também variáveis livres. Os valores dos endereços não são especificados até mais tarde (seus lugares são tomados por variáveis). Estes endereços só serão preenchidos durante a etapa de alocação de endereços do compilador.

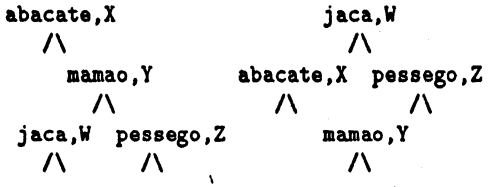
O programa `procure/3`, apresentado a seguir, acessa a estrutura `dicc/4` para recuperar o endereço `Endereco` da variável `Nome`, ou para inserir esta última no dicionário:

```
procure(Nome,Endereco,dicc(Nome,Endereco,_,_)) :- !.
procure(Nome,Endereco,dicc(N1,_,DiccE,_)) :-
    Nome @< N1,
    procure(Nome,Endereco,DiccE).
```

```
procure(Nome,Endereco,dicc(N1,_,_,DiccD)) :-
    Nome @> N1,
    procure(Nome,Endereco,DiccD).
```

Em cada passo, a chave `Nome` é comparada com a chave do nó corrente. Se ela for menor, o ramo esquerdo será verificado recursivamente; se for maior o mesmo será feito, só que no ramo direito. Se a busca chegar às folhas do dicionário e a chave `Nome` não for encontrada, essa chave será inserida como consequência da instanciação de variáveis da primeira cláusula de `procure/3`.

É importante analisar o significado e a razão do corte (`!`) na primeira cláusula do programa `procure/3`. A razão principal é que um dicionário ordenado para um determinado conjunto de pares de valores não é único; depende da ordem em que cada chave é inserida. Em princípio, o procedimento `procure/3` poderia construir dicionários com árvores diferentes, porém equivalentes, como por exemplo:



Isto é devido ao fato que uma meta como `procure(abacate,D,K)` com uma variável não instanciada como segundo argumento pode unificar não só com a primeira cláusula de `procure/3`, mas também com a segunda e/ou a terceira. Isto porque o átomo `abacate` pode ser igual, menor ou maior que qualquer variável não instanciada. Estas unificações alternativas permitiriam, em princípio, formas diferentes do dicionário que são equivalentes.

Em outras palavras, o corte está significando:

> se conseguir unificar com a primeira cláusula não tente com nenhuma das seguintes, pois, ou você achou a chave procurada ou é nesse nó que você deve inserí-la.

Assim, o corte assegura que `procure/3` construa um único dicionário ordenado, começando de uma variável inicialmente não instanciada. O dicionário é único exceto pelo fato de que os nós terminais são variáveis livres que representam subdicionários não especificados.

A seguir estão os códigos e o dicionário produzidos pelo gerador de código para as estruturas fornecidas pelo analisador sintático apresentadas na página 34.

MEDIA ARITMETICA

```
instr(read,_559C) ; (instr(loadc,0) ; instr(store,_5BF0)) ;
(instr(loadc,0) ; instr(store,_5C58)) ; ((instr(loadc,1) ;
instr(store,_55D4)) ; label(_52C0) ; ((instr(load,_55D4) ;
instr(sub,_559C)) ; instr(jumpgt,_5340)) ; (instr(read,
_5994) ; ((instr(load,_5BF0) ; instr(addc,1)) ; instr(store,
_5BF0)) ; ((instr(load,_5C58) ; instr(add,_5994)) ; instr(
store,_5C58)) ; nop) ; instr(load,_55D4) ; instr(add,1) ;
instr(store,_55D4) ; instr(jump,_52C0) ; label(_5340)) ;
( (instr(load,_5C58) ; instr(div,_5BF0) ) ; instr(store,
_5D18)) ; (instr(load,_5D18) ; instr(write,0)) ; nop

dicc(n,_559C,dicc(cont,_5BF0,_5168,dicc(i,_55D4,_5484,
dicc(media,_5D18,_5B9C,_5BA0))),dicc(soma,_5C58,_5248,
dicc(valor,_5994,_5718,_571C)))
```

EXPRESSAO

```
instr(read,_99B0) ; instr(read,_993C) ; (((instr(loadc,5) ;
instr(sub , _99B0)) ; instr(store, _97C4) ; ((((instr(load,
_993C) ; instr(multc,7)) ; instr(store,_93B0) ; instr(load,
_99B0) ; instr(add, _93B0)) ; instr(subc,9)) ; instr(store,
_93B0) ; instr(loadc, 4) ; instr(mult, _93B0)) ; instr(div,
_97C4)) ; instr(store , _9A10)) ; ((((instr(loadc , 7) ;
instr(add,_993C)) ; instr(store,_97C4) ; instr(load,_99B0) ;
instr(mult,_97C4)) ; instr(store,_97C4) ; (instr(loadc,3) ;
instr(add,_9A10)) ; instr(sub,_97C4)) ; instr(write,0)) ;
nop

dicc(x,_99B0, dicc(a,_9A10,_8FB0,_8FB4),dicc(y,_993C,_8EDC,
dicc(temp(0),_97C4,_9114,dicc(temp(1),_93B0,_932C,_9330))))
```

5.4 Endereçamento

O estágio final a ser executado pelo compilador é transformar o código relocável em código objeto absoluto. O programa `assembler(Codigo,Dicc,Cod_Objeto)` tem como parâmetros de entrada `Codigo` e `Dicc`, gerados no estágio anterior, e como parâmetro de saída produz o código objeto `Cod_Objeto`.

Podem ser destacadas duas etapas na geração de endereços. Durante a primeira etapa, as instruções do código são contadas; ao mesmo tempo, são calculados os endereços dos rótulos gerados durante a geração de código, bem como são removidas as operações `no_op`. A esse código é adicionado a instrução `halt` — `instr(halt,0)` — e a instrução `block(n)`,

onde n é o número de locações de memória necessárias para alocar as variáveis do programa. Isto é feito pelo predicado `conta_linhas/4`. Na segunda etapa, o predicado `alocar/3` deverá criar os endereços para as variáveis do programa. O programa é:

```
assembler(Codigo,Codigo_Objeto,Dicc):-
    conta_linhas(Codigo,1,N,Codigo_Objeto/
                 (instr(halt,0);instr(block,NVar))),
    N1 is N + 2,
    alocar(Dicc,N1,N2),
    NVar is N2 - N1 + 1.
```

O predicado `conta_linhas/4` tem, como primeiro parâmetro de entrada, o código gerado pelo predicado `codifique/3` e, como último parâmetro, o código objeto, no qual os endereços dos rótulos foram preenchidos e as operações nulas — `no_op` — foram removidas. O segundo parâmetro é o endereço do início do código, enquanto o terceiro parâmetro é o endereço final do código incrementado em uma unidade.

```
conta_linhas(instr(X,Y),M,N,(instr(X,Y);Cod)/Cod) :-
    N is M + 1.

conta_linhas(label(N),N,N,Cod/Cod).

conta_linhas(nop,N,N,Cod/Cod).

conta_linhas((Cod;RestCod),M,N,Ass/RestAss) :-
    conta_linhas(Cod,M,M1,Ass/Resto),
    conta_linhas(RestCod,M1,N,Resto/RestAss).
```

No código ocorrem três tipos de primitivas: instruções, rótulos e `no_op`. No tratamento das instruções o contador de endereços do código é incrementado de um, e a instrução é inserida no código de saída.

As duas últimas cláusulas removem os rótulos e as operações nulas sem incrementar o contador de endereços.

O efeito do predicado `alocar/3` é atribuir os endereços reais de memória para as variáveis e preencher as referências a estes no programa.

```
alocar(X,N,N-1) :- var(X),!.

alocar(dicc(Nome,N,De,Dd),N,M) :-
    N1 is N + 1,
    alocar(De,N1,N2),
    N3 is N2 + 1,
    alocar(Dd,N3,M).
```

Os códigos objetos finais produzidos pelo endereçamento dos códigos apresentados na seção 5.3.2, página 41, estão listados a seguir, bem como os dicionários, agora com os endereços finais para cada variável.

MEDIA ARITMETICA

```
instr(read,29) ; instr(loadc,0)  ; instr(store,30) ;
instr(loadc,0) ; instr(store,33)  ; instr(loadc,1) ;
instr(store,31)  ; instr(load,31)  ; instr(sub,29) ;
instr(jumpgt,22) ; instr(read,34) ; instr(load,30) ;
instr(addc,1)  ; instr(store,30)  ; instr(load,33) ;
instr(add,34)  ; instr(store,33)  ; instr(load,31) ;
instr(add,1)  ;  instr(store,31)  ;  instr(jump,8) ;
instr(load,33)  ;  instr(div,30) ; instr(store,32) ;
instr(load,32)  ;  instr(write,0)  ; instr(halt,0) ;
instr(block,6)

dicc(n,29, dicc(cont,30,_5168 , dicc(i,31,_5484,
dicc(media,32,_5B9C,_5BA0))),dicc(soma,33,_5248,
dicc(valor,34,_5718,_571C)))
```

EXPRESSAO

```
instr(read,29) ; instr(read,31)  ; instr(loadc,5) ;
instr(sub,29) ; instr(store,32)  ; instr(load,31) ;
instr(multc,7) ; instr(store,33) ; instr(load,29) ;
instr(add,33)  ; instr(subc,9)  ; instr(store,33) ;
instr(loadc,4)  ; instr(mult,33)  ; instr(div,32) ;
instr(store,30)  ; instr(loadc,7) ; instr(add,31) ;
instr(store,32) ; instr(load,29) ; instr(mult,32) ;
instr(store,32)  ; instr(loadc,3) ; instr(add,30) ;
instr(sub,32)   ; instr(write,0)  ; instr(halt,0) ;
instr(block,5)

dicc(x , 29, dicc(a,30,_8FB0,_8FB4), dicc(y,31,_8EDC,
dicc(temp(0),32,_9114,dicc(temp(1),33,_932C,_9330))))
```

5.5 Impressão do Codigo

O programa apresentado a seguir é utilizado para produzir saídas semelhante às mostradas nas tabelas 2 e 3 nas páginas 17 e 18 respectivamente.

```
imprime_codigo(Assembly,Dicc) :-
   mostre_codigo(Assembly,1),
   mostre_var(Dicc).

mostre_codigo((Cod1;Cod2),N) :-
   mostre_codigo(Cod1,N),
   N1 is N + 1,
   mostre_codigo(Cod2,N1).

mostre_codigo((X),N) :- mostre(X,N).

mostre(instr(Codigo,Operando),N) :-
   tab(3),writedigit(N),
   tab(3),write(Codigo),
   tab(1),write(Operando),
   nl.

mostre_var(X) :- var(X),!.
mostre_var(dicc(Nome,N,De,Dd)) :-
   tab(3),writedigit(N),
   tab(3),write(Nome),nl,
   mostre_var(De),
   mostre_var(Dd).

writedigit(N) :- N < 10,write(0),write(N),!.
writedigit(N) :- write(N).
```

6 Listagem do Programa

A listagem completa da implementação do compilador CRISTAL encontra-se a seguir:

6.1 Módulo Principal

```
compile(Arq) :-
  abrir_arquivo(Arq,ArqE),
  lexico(ArqE,Tokens),
  sintatico(Tokens,Estrutura),
  codifique(Estrutura,Codigo,Dicc),
  assembler(Codigo,Assembly,Dicc),
  imprime_codigo(Assembly,Dicc).

abrir_arquivo(Arq,Handle_E) :-
  concat(Arq,$.r$,Arquivo),
  open(Handle_E,Arquivo,r).
```

6.2 Analisador Léxico

```
lexico(Arquivo,[Palavra|Tokens]) :-
  le_caracter(Arquivo,Car),
  leia_palavra(Arquivo,Car,Palavra,UltCar),
  tokenize(Arquivo,Palavra,UltCar,Tokens).

tokenize(_,C,_,[]) :- fim_de_programa(C).

tokenize(Arquivo,_,Caracter,[Palavra|Tokens]) :-
  leia_palavra(Arquivo,Caracter,Palavra,UltCar),
  tokenize(Arquivo,Palavra,UltCar,Tokens).

le_caracter(Arquivo,T) :-
  get0(Arquivo,C),
  name(X,[C]),write(X),
  identifica_caracter(C,T),
  !.

identifica_caracter(C,identificador(letra,C)) :-
  C >= 'a,
  C =< 'z.
```

```
identifica_caracter(C,identificador(letra,L)) :-
  C >= 'A,
  C =< 'Z,
  L is C + 32.

identifica_caracter(C,identificador(numero,C)) :-
  C >= '0,
  C =< '9.

identifica_caracter(C,operador(C)) :-
  C >= '(,
  C =< '>.

identifica_caracter('{,comentario(inicio,tipo_abre_fecha)).
identifica_caracter('},comentario(termino,tipo_abre_fecha)).
identifica_caracter('%,comentario(inicio,tipo_linha)).
identifica_caracter(10,comentario(termino,tipo_linha)).
identifica_caracter(_,separador(_,_)).

leia_palavra(Arquivo,comentario(inicio,Tipo),Palavra,UltCar) :-
  le_caracter(Arquivo,Caracter),
  termina_documentacao(Arquivo,Tipo,Caracter,0),
  le_caracter(Arquivo,Caracter),
  leia_palavra(Arquivo,Caracter,Palavra,UltCar).

leia_palavra(Arquivo,operador(C),Simbolo,UltCar) :-
  pode_compor(C),!,
  le_caracter(Arquivo,Car),
  compoe_simbolo(Arquivo,C,Car,Simbolo,UltCar).

leia_palavra(Arquivo,operador(C),Simbolo,UltCar) :-
  name(Simbolo,[C]),
  le_caracter(Arquivo,UltCar).

leia_palavra(Arquivo,identificador(letra,C),Palavra,UltCar) :-
  le_caracter(Arquivo,Caracter),
  le_identificador(Arquivo,Caracter,Lista,UltCar),
  name(Palavra,[C|Lista]).

leia_palavra(Arquivo,identificador(numero,C),Numeral,UltCar) :-
  le_caracter(Arquivo,Caracter),
  le_numero(Arquivo,Caracter,Lista,UltCar),
  name(Numeral,[C|Lista]).
```

```
leia_palavra(Arquivo,separador,Palavra,UltCar) :-
  le_caracter(Arquivo,Caracter),
  leia_palavra(Arquivo,Caracter,Palavra,UltCar).

termina_documentacao(Arquivo,Tipo,comentario(termino,Tipo),0) :- !.

termina_documentacao(Arquivo,tipo_abre_fecha,
                     comentario(inicio,tipo_abre_fecha),N) :-
  N1 is N + 1,!,
  le_caracter(Arquivo,Caracter),
  termina_documentacao(Arquivo,tipo_abre_fecha,Caracter,N1).

termina_documentacao(Arquivo,Tipo,comentario(termino,Tipo),N) :-
  N1 is N - 1,!,
  le_caracter(Arquivo,Caracter),
  termina_documentacao(Arquivo,Tipo,Caracter,N1).

termina_documentacao(Arquivo,Tipo,_,N) :-
  le_caracter(Arquivo,C),
  termina_documentacao(Arquivo,Tipo,C,N).

le_identificador(Arquivo,identificador(_,Car),[Car|Lista],UltCar) :-
  le_caracter(Arquivo,Caracter),
  le_identificador(Arquivo,Caracter,Lista,UltCar).
le_identificador(_,Car,[],Car).

le_numero(Arquivo,identificador(numero,Car),[Car|Lista],UltCar) :-
  le_caracter(Arquivo,Caracter),

le_numero(Arquivo,Caracter,Lista,UltCar).
le_numero(_,Car,[],Car).

pode_compor(':).
pode_compor('<).
pode_compor('>).

compoe_simbolo(Arquivo,C,operador(aritmetico,Car),Simb,UltCar) :-
  pega_composicoes(C,Car),!,
  name(Simb,[C,Car]),
  le_caracter(Arquivo,UltCar).
compoe_simbolo(_,C,Car,Simb,Car) :- name(Simb,[C]).

pega_composicoes(':,'=).
pega_composicoes('>,'=).
pega_composicoes('<,'=).
```

```
pega_composicoes('<','>').

fim_de_programa('.').
```

6.3 Analisador Sintático

```
sintatico(Tokens,Estrutura) :- programa(Estrutura,Tokens,[]).

programa(E) --> [program],id(X),[';'],comando(E),['.'].

comando((C ; Cresto)) --> [begin],comando(C),
                          resto_comando(Cresto).
comando(atribua(X,Expr)) --> id(X),[':='],expressao(Expr).
comando(condicao(Teste,C1,C2)) --> [if],teste(Teste),
                                   [then],comando(C1),
                                   [else],comando(C2).
comando(enquanto(Teste,C)) --> [while],teste(Teste),
                               [do],comando(C).
comando(repita(C,Teste)) --> [repeat],nucleo_repeat(C),
                             [until],teste(Teste).
comando(para(nome(J),Inicio,Fim,C)) --> [for],id(J),
                                        [':='],expressao(Inicio),
                                        [to],expressao(Fim),
                                        [do],comando(C).
comando(leia(X)) --> [read],['('],id(X),[')'].
comando(escreva(X)) --> [write],['('],expressao(X),[')'].

resto_comando(fim) --> [end];[';',end].
resto_comando((C;Cresto)) --> [';'],comando(C),
                              resto_comando(Cresto).

nucleo_repeat(C) --> comando(C).
nucleo_repeat((C;Cresto)) --> comando(C),[';'],
                              nucleo_repeat(Cresto).

teste(teste(OP,E1,E2)) --> expressao(E1),
                           comparacao(OP),
                           expressao(E2).

expressao(E) --> termo(T),resto_expressao(E,T).
resto_expressao(E,E1) --> op1(Op),termo(T),
                          resto_expressao(E,expr(Op,E1,T)).
resto_expressao(E,E) --> [].
```

```
termo(T) --> fator(F),resto_termo(T,F).
resto_termo(T,T1) --> op2(Op),fator(F),
                      resto_termo(T,expr(Op,T1,F)).
resto_termo(T,T) --> [].

fator(E) --> ['('],expressao(E),[')'].
fator(F) --> identificador(F).

identificador(nome(X)) --> id(X).
identificador(numero(X)) --> inteiro(X).

id(X) --> [X],{atom(X)}.
inteiro(X) --> [X],{integer(X)}.

op1('-') --> ['-'].
op1('+') --> ['+'].
op2('*') --> ['*'].
op2('/') --> ['/'].

comparacao('=') --> ['='].
comparacao('<') --> ['<'].
comparacao('>') --> ['>'].
comparacao('>=') --> ['>='].
comparacao('<=') --> ['<='].
comparacao('<>') --> ['<>'].
```

6.4 Gerador de Código

```
codifique((E;Resto),(E1;Resto1),Dicc) :-
   codifique(E,E1,Dicc),!,
   codifique(Resto,Resto1,Dicc).

codifique(atribua(Nome,Expressao),
          (CodExpr;instr(store,Endereco)),Dicc):-
   procure(Nome,Endereco,Dicc),
   cod_expressao(Expressao,0,CodExpr,Dicc).

codifique(leia(X),instr(read,Endereco),Dicc) :-
   procure(X,Endereco,Dicc).

codifique(escreva(Expressao),(CodExpr;instr(write,0)),Dicc) :-
   cod_expressao(Expressao,0,CodExpr,Dicc).
```

```
codifique(condicao(Teste,ComEntao,ComSenao),(CodTeste;CodEntao;
          instr(jump,L2);label(L1);CodSenao;label(L2)),Dicc) :-
   codifique_teste(Teste,L1,CodTeste,Dicc),
   codifique(ComEntao,CodEntao,Dicc),
   codifique(ComSenao,CodSenao,Dicc).

codifique(enquanto(Teste,Comando),(label(L1);CodTeste;
          CodComando;instr(jump,L1);label(L2)),Dicc) :-
   codifique_teste(Teste,L2,CodTeste,Dicc),
   codifique(Comando,CodComando,Dicc).

codifique(repita(Comando,Teste),(label(L1);
          CodComando;CodTeste),Dicc) :-
   codifique(Comando,CodComando,Dicc),
   codifique_teste(Teste,L1,CodTeste,Dicc).

codifique(para(nome(I),Inicio,Fim,Comando),
         (CodInic;label(L1);CodTeste;CodComando;
         instr(load,End_I);instr(add,1);
         instr(store,End_I);instr(jump,L1);
         label(L2)),Dicc) :-
   codifique(atribua(I,Inicio),CodInic,Dicc),
   procure(I,End_I,Dicc),
   codifique_teste(teste('<=',nome(I),Fim),L2,CodTeste,Dicc),
   codifique(Comando,CodComando,Dicc).

codifique(fim,nop,_).

codifique_teste(teste(OP,Exp1,Exp2),Label,
                (Codigo;instr(CodOp,Label)),Dicc) :-
   cod_expressao(expr('-',Exp1,Exp2),0,Codigo,Dicc),
   operador_logico(OP,CodOp).

cod_expressao(numero(X),_,instr(loadc,X),_).
cod_expressao(nome(X),_,instr(load,End),Dicc) :-
   procure(X,End,Dicc).
cod_expressao(expr(OP,E1,E2),N,(CodE1;Instr),Dicc) :-
   instr_simples(OP,E2,Instr,Dicc),
   cod_expressao(E1,N,CodE1,Dicc).
```

```
cod_expressao(expr(OP,E1,E2),N,(CodE2;instr(store,Endereco);
              CodE1;instr(OpCod,Endereco)),Dicc) :-
   complexa(E2),
   procure(temp(N),Endereco,Dicc),
   cod_expressao(E2,N,CodE2,Dicc),
   N1 is N + 1,
   cod_expressao(E1,N1,CodE1,Dicc),
   operador_literal(OP,OpCod).

complexa(expr(_,_,_)).

instr_simples(OP,numero(N),instr(OpCod,N),Dicc) :-
   operador_memoria(OP,OpCod).
instr_simples(OP,nome(N),instr(OpCod,Endereco),Dicc) :-
   procure(N,Endereco,Dicc),
   operador_literal(OP,OpCod).

operador_memoria('+',addc).
operador_memoria('-',subc).
operador_memoria('*',multc).
operador_memoria('/',divc).

operador_literal('+',add).
operador_literal('-',sub).
operador_literal('*',mult).
operador_literal('/',div).

operador_logico('=',jumpne).
operador_logico('<>',jumpeq).
operador_logico('>',jumple).
operador_logico('>=',jumplt).
operador_logico('<',jumpge).
operador_logico('<=',jumpgt).


procure(Nome,Endereco,dicc(Nome,Endereco,_,_)) :- !.
procure(Nome,Endereco,dicc(N1,_,DiccE,_)) :-
   Nome @< N1,
   procure(Nome,Endereco,DiccE).
procure(Nome,Endereco,dicc(N1,_,_,DiccD)) :-
   Nome @> N1,
   procure(Nome,Endereco,DiccD).
```

6.5 Endereçamento

```
assembler(Codigo,Codigo_Objeto,Dicc):-
  conta_linhas(Codigo,1,N,Codigo_Objeto/
               (instr(halt,0);instr(block,NVar))),
  N1 is N + 2,
  alocar(Dicc,N1,N2),
  NVar is N2 - N1 + 1.

conta_linhas(instr(X,Y),M,N,(instr(X,Y);Cod)/Cod) :-
  N is M + 1.
conta_linhas(label(N),N,N,Cod/Cod).
conta_linhas(nop,N,N,Cod/Cod).
conta_linhas((Cod;RestCod),M,N,Ass/RestAss) :-
  conta_linhas(Cod,M,M1,Ass/Resto),
  conta_linhas(RestCod,M1,N,Resto/RestAss).

alocar(X,N,N-1) :- var(X),!.
alocar(dicc(Nome,N,De,Dd),N,M) :-
  N1 is N + 1,
  alocar(De,N1,N2),
  N3 is N2 + 1,
  alocar(Dd,N3,M).
```

6.6 Impressão do Codigo

```
imprime_codigo(Assembly,Dicc) :-
  mostre_codigo(Assembly,1),
  mostre_var(Dicc).

mostre_codigo((Cod1;Cod2),N) :-
  mostre_codigo(Cod1,N),
  N1 is N + 1,
  mostre_codigo(Cod2,N1).

mostre_codigo((X),N) :- mostre(X,N).

mostre(instr(Codigo,Operando),N) :-
  tab(3),writedigit(N),
  tab(3),write(Codigo),
  tab(1),write(Operando),
  nl.
```

```
mostre_var(X) :- var(X),!.
mostre_var(dicc(Nome,N,De,Dd)) :-
  tab(3),writedigit(N),
  tab(3),write(Nome),nl,
  mostre_var(De),
  mostre_var(Dd).

writedigit(N) :- N < 10,write(0),write(N),!.
writedigit(N) :- write(N).
```

7 Sugestões para Estender a Implementação

São várias as sugestões para estender a implementação Prolog do compilador apresentado neste trabalho, entre as quais se destacam as seguintes:

1. **Tratamento de Erros:** deve ser observado que se houver um erro na sintaxe do programa fonte, o analisador sintático, do modo como foi implementado, pára e não indica o tipo de erro detectado. O analisador sintático pode ser estendido para informar ao usuário qual a cadeia não reconhecida, indicando o tipo de erro e, eventualmente, a cadeia esperada. Além disso, é possível estender o analisador sintático para que, após reconhecer um erro, ele seja capaz de continuar a análise do restante do texto fonte. Isso pode ser implementado de uma forma simples fazendo com que, após detectar um erro, o analisador sintático continue consumindo símbolos até encontrar um delimitador — por exemplo um ponto e vírgula — ou uma palavra chave — por exemplo *begin, while, for, repeat* — que o leve a um estado de reconhecimento inicial. Em seguida, ele poderá continuar a analisar normalmente o texto de entrada.

2. **Tipos de Dados:** a implementação considera um único tipo primitivo de dados — inteiro. Ela pode ser estendida para admitir outros tipos primitivos, tais como real, caracter, booleano.

3. **Procedimentos e Funções:** é interessante ampliar a linguagem CALIAL para manipular procedimentos e funções com diversas formas de passagem de parâmetros — por valor, referência, etc.

4. **Otimização:** o código gerado não é otimizado. Assim, uma outra sugestão para estender o compilador é a inclusão de um otimizador de código para gerar código objeto otimizado.

5. **Geração do Código Objeto Final:** é interessante implementar um montador para gerar o código objeto para uma máquina real, a partir do p-code da máquina virtual gerado pelo compilador. Deste modo, o código pode ser efetivamente executado por essa máquina.

8 Conclusões

O uso da linguagem Prolog para implementar compiladores proporciona algumas vantagens para o programador, dentre as quais podem ser destacadas:

1. A implementação é mais legível e pode ser auto-documentada. Pode-se até dizer que a implementação é a própria especificação.
2. A corretude da implementação fica mais visível e o escopo dos erros é grandemente reduzido.
3. As modificações no compilador, bem como as extensões da linguagem fonte, são incorporadas mais facilmente em Prolog, pois a implementação do compilador consiste de pequenas unidades independentes que estão relacionadas diretamente à estrutura da linguagem fonte.
4. A implementação de um compilador usando a linguagem Prolog deixa o programador livre da preocupação com detalhes característicos das linguagens procedimentais, tais como: atribuições, referências (apontadores), seleção detalhada de estruturas de dados, etc.

Resumindo, Prolog apresenta várias vantagens como ferramenta para desenvolver compiladores, entre elas:

- menos tempo e esforço do implementador são requeridos;
- há menor probabilidade de erros;
- a implementação resultante é mais fácil de ser mantida e modificada.

Entretanto, para desenvolver um compilador de maior porte em Prolog, é imprescindível conhecer bem os dois aspectos desta linguagem, que são:

- o *aspecto declarativo*, que descreve a lógica do problema;
- o *aspecto procedimental*, que descreve como o compilador resolve o problema.

Conseguir conciliar estes dois aspectos nem sempre é uma tarefa fácil. O entendimento do aspecto procedimental de Prolog permite desenvolver programas eficientes que também não usam muita memória.

Referências

[Aho 86] ALFRED V. AHO , RAVI SETHI & JEFFREY D. ULLMAN. *Compilers - Principles, Techniques and Tools.* Addison-Wesley, 1986

[Arity 86] ARITY CORPORATION. *The Arity/Prolog Programming Language.* CSA Press, Massachussets, 1986

[Bratko 86] IVAN BRATKO. *Prolog Programming for A.I.* Addison-Wesley, 1986

[Clocksin 87] WILLIAM F. CLOCKSIN & CHRISTOPHER S. MELLISH. *Programming in Prolog.* Springer-Verlag, 3a. Ed. , Germany, 1987

[Marcus 86] CLAUDIA MARCUS. *Prolog Programming - Applications for Database Systems, Expert Systems and Natural Language Systems.* Addison-Wesley Publishing Company, 1986

[Neto 87] JOÃO JOSÉ NETO. *Introdução à Compilação.* Livros Técnicos e Científicos Editora S. A., 1987

[Sterling 87] LEON STERLING & EHUD SHAPIRO. *The Art of Prolog.* The Mit Press, 1987

[Warren 80] DAVID H. D. WARREN. *Logic Programming and Compiler Writing.* Software Practice and Experience, Vol. 10, pp. 97-125, 1980

[Wirth 82] NIKLAUS WIRTH. *Programação Sistemática em Pascal.* Ed. Campus, 1982

[Wirth 86] NIKLAUS WIRTH. *Algorithms and Data Structures.* Prentice-Hall, 1986