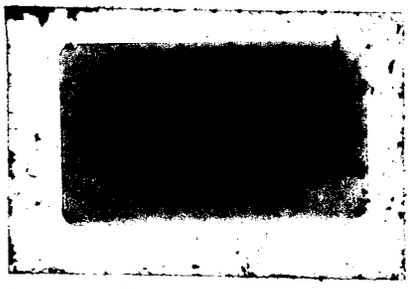
UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE FÍSICA E QUÍMICA DE SÃO CARLOS
DEPARTAMENTO DE FÍSICA E CIÊNCIA DOS MATERIAIS

Proposta e Construção de um Compilador PASCAL para Arquitetura RISC-LIE

ANTONIO FERNANDO TRAINA



ac

Dissertação apresentada ao Instituto de Física e Química de São Carlos, Universidade de São Paulo para a obtenção do título de Mestre em Ciências Física Aplicada.

Orientador: Prof. Dr. Carlos Antonio Ruggiero

São Carlos - São Paulo
1993

181

SERVIÇO DE BIBLIOTECA E INFORMAÇÃO - IFQSC
FÍSICA

UNIVERSIDADE DE SÃO PAULO
Instituto de Física e Química de São Carlos

Fone (0162) 72-6222
Fax (0162) 72-2218

Av. Dr. Carlos Botelho, 1465
Caixa Postal 369
CEP 13560.970 - São Carlos - SP
Brasil

MEMBROS DA COMISSÃO JULGADORA DA DISSERTAÇÃO DE MESTRADO DE **ANTONIO FERNANDO TRAINA** APRESENTADA AO INSTITUTO DE FISICA E QUIMICA DE SAO CARLOS, DA UNIVERSIDADE DE SAO PAULO, EM 13/09/1993

COMISSAO JULGADORA:

Prof.Dr.Carlos Antonio Ruggiero

Prof.Dr.Orion de Oliveira Silva

Prof.Dr.Edson dos Santos Moreira

**Aos meus pais Caetano e Aurora
e à minha esposa Maria do Carmo,
que tanto colaboraram e, pela
dedicação, compreensão e paciência,
eu dedico este trabalho.**

Gostaria de agradecer a todos que, de uma forma ou de outra contribuiram para realização deste trabalho. Em especial:

- *Ao Prof. Ruggiero pela luz constante de sua sabedoria,*
- *à Maria do Carmo, companheira paciente, pelo carinho e dedicação, e ainda pela revisão de meus textos*
- *ao Prof. Caetano, meu irmão, que me fez ver os primeiros rudimentos de computação, fazendo com que minha mudança de área fosse possível, pelo constante apoio técnico, e sugestões sempre decisivas,*
- *Ao Prof. Jan Slaets por sua inspiração onipresente,*
- *ao Prof. Gonzalo Travieso, que tantas vezes cedeu seu micro e pacientemente ajudou-me a ingressar na área da computação,*
- *ao Chú, com suas observações peculiares e sempre corretas,*
- *ao amigo Renato Vairo, que, com sua visão clara, muitas vezes me fez ver a luz no fim do túnel,*
- *aos amigos e colegas do laboratório de Instrumentação Eletrônica Claudio de Vincenzi, Lírio "Rock" de Almeida, Lia, Ivanilda e ao Barney,*
- *ao meu predecessor Fredy, pela base sólida que construiu,*
- *a Agma, pelo apoio constante e compreensão,*
- *Ao amigo Edgar Froese, pela inspiração,*
- *Ao meu tio Carlos, por ajudar na minha formação moral, e espiritual,*
- *e tantos outros ...*

Conteúdo

Resumo vi

Abstract. vii

1 Introdução à arquitetura RISC 1

1.1 Caminhos para uma nova arquitetura 1

1.2 Características da arquitetura RISC. 5

1.3 Arquiteturas RISC. 8

1.3.1 RISC I. 8

1.3.2 RISC II 9

1.3.3 IBM 801 11

1.3.4 Projeto MIPS X. 11

1.3.5 Projeto Spectrum. 12

1.3.6 Processador Am 29000. 13

1.3.7 RISC 6000 14

1.3.8 Outras implementações e tendências da arquitetura RISC . . 15

1.4 A Arquitetura RISC_LIE. 18

2 Construção de Compiladores 20

2.1 Linguagens de programação e compiladores. 20

2.1.1 Conceitos básicos. 20

2.1.2 Aspectos da construção de compiladores 24

2.2 Anatomia de um compilador 26

2.2.1 Analisador léxico. 27

2.2.2 Analisador sintático 30

2.2.3 Analisador semântico 32

2.2.4 Otimização global. 34

2.2.5 Geração de código. 35

2.2.6 Otimização local 35

2.2.7 Um exemplo de funcionamento de um compilador 35

2.3 Ferramentas de desenvolvimento de compiladores 38

2.3.1 Os interpretadores YACC e BISON. 38

3 Programação de um subconjunto da linguagem PASCAL 43

3.1 PASCAL Padrão. 43

3.1.1 Introdução. 43

3.1.2 Descrição geral 45

3.1.2 Noções básicas. 47
3.2 Subconjunto escolhido o compilador PARIS. 53
4 Geração e otimização de código. 56
4.1 Otimização de código 56
4.1.1 Conceitos básicos de otimização de código 56
4.1.2 Otimização de laços 58
4.1.3 Representação de blocos básicos D.A.G. 60
4.1.5 Análise de fluxo de dados global 63
4.2 Geração de código. 64
4.2.1 O programa objeto 65
4.2.2 O ambiente do gerador de código. 66
4.2.3 Problemas da geração de código 67
4.2.4 O modelo da máquina RISC-LIE. 68
4.2.5 Um gerador de códigos simples 70
4.2.6 Geração de códigos apartir de D.A.G.s 73
5 Resultados obtidos. 75
5.1 Considerações iniciais. 75
5.2 Etapas da implementação 76
5.2.1 O analisador léxico 79
5.2.1.1 Produção do módulo 84
5.2.2 O analisador sintático 84
5.2.2.1 O controle de ambiguidades 84
5.2.2.2 Conflitos na gramática do compilador PARIS 90
5.2.2.3 Produção do módulo 95
5.2.3 O analisador semântico 95
5.2.3.1 Interfaces em linguagem C 95
5.2.3.2 Chamadas das rotinas das ações semânticas 97
5.2.2.3 Produção das ações semânticas 99
5.2.3.4 Produção do módulo 100
5.2.4 Modulos de otimização 100
5.2.4.1 Produção do módulo 103
5.2.5 O módulo gerador de código 103
5.2.5.1 Produção do módulo 111
5.3 Análise de resultados e desempenho do compilador gerado 111

5.4 Considerações finais 113
6 Conclusões e perspectivas futuras 114
Bibliografia. 117
Apendice A Arquivos BNF e YACC a.1
A.1 Estrutura da BNF extendida para linguagem Pascal Padrão a.1
A.2 Regras sintáticas do compilador PARIS a.10
Apêndice B Manual do BISON/YACC b.1
B.1 Conceitos BISON b.1
B.1.1 Linguagens e gramática livre de contexto b.1
B.1.2 Das regras formais para entrada aceita em BISON b.3
B.1.3 Valores semânticos b.4
B.1.4 Ações semânticas b.5
B.2 Arquivos da gramática BISON b.5
B.2.1 Síntese da gramática BISON b.5
B.2.1.1 Secção das declarações em C b.6
B.2.1.2 Secção das declarações BISON b.6
B.2.1.3 Secção das regras gramaticais b.6
B.2.1.4 Secção de código C adicional b.6
B.2.2 Símbolos, terminais e não terminais b.7
B.2.3 Sintaxe das regras gramaticais b.8
B.2.4 Regras recursivas b.9
B.2.5 A semântica da linguagem b.10
B.2.5.1 Tipos de dados dos valores semânticos b.10
B.2.5.2 Mais de um tipo de dado para valores semânticos b.11
B.2.5.3 Ações b.11
B.2.5.4 Tipos de dados de valores em ações b.12
B.2.5.5 Ações de meia regra b.13
B.2.6 Declarações em BISON b.16
B.2.6.1 Declaração de nome do tipo *token* b.16
B.2.6.2 Declaração do operador de precedência b.17
B.2.6.3 Declaração de uma coleção de tipos de valores b.18
B.2.6.4 Declaração de tipos de valores de símbolos não terminais. b.18
B.2.6.5 Prevenção de conflitos (*warnings*) b.19

B.2.6.6 Declaração do símbolo inicial b.19
B.2.6.7 Requisitando um *parser* puro b.20
B.2.6.8 Sumário das declarações BISON b.20
B.2.7 Multiplos analisadores em um mesmo programa b.21
B.3 Interface em Linguagem C do analisador b.22
B.3.1 A função do analisador **yyparse** b.23
B.3.2 Convenção de chamada para **yylex** b.23
B.3.2.1 Convenção de chamada b.23
B.3.2.2 Retornando valores semânticos de *tokens* b.24
B.3.2.3 Relatando posições textuais de *tokens* b.25
B.3.2.4 Convenções de chamada para analisadores puros b.25
B.3.3 A função relatório de erro *yyerror* b.26
B.3.4 Características especiais para uso em ações b.27
Apêndice C Arquitetura RISC-LIE c.1
C.1 Micro arquitetura RISC-LIE c.1
C.2 Tabela das instruções da arquitetura RISC-LIE c.2
Apêndice D Listagem da saída dos módulos do compilador **PARIS** d.1
D.1 Analisador léxico d.1
D.2 Analisador sintático d.2
D.3 Analisador semântico d.19
D.4 Geração de código d.23

LISTA DE FIGURAS

Figura 1.1 *Pipeline* de três estágios 8
Figura 1.2 Formatos de instrução 9
Figura 1.3 Organização circular de registradores. 10
Figura 2.1 Esquema geral de um compilador 21
Figura 2.2 Esquema de execução de um programa objeto 22
Figura 2.3 Esquema do funcionamento básico de um compilador 27
Figura 4.1 Representação de um D.A.G. simples 61

SERVIÇO DE BIBLIOTECA E INFORMAÇÃO - IFQSC
FÍSICA

RESUMO

Este trabalho apresenta uma proposta para implementação de um subconjunto de instruções e comandos de uma linguagem Pascal Padrão ISO, aplicada a arquitetura RISC, tendo como base a arquitetura RISC-LIE [Vale91], proposta e desenvolvida no IFQSC.

Para a definição e construção de parte do código gerado foi utilizada a ferramenta de desenvolvimento de compiladores YACC, que definiu toda estrutura gramatical da linguagem, sendo que as demais estruturas foram desenvolvidas usando interfaces em linguagem C.

O código gerado pelo compilador utilizou trinta instruções de máquina que compõe o simulador da arquitetura RISC-LIE, gerando assim códigos compatíveis que podem ser interpretados por esse simulador.

ABSTRACT

This work presents a proposal for an implementation of a subset of instructions and commands of Standart Pascal ISO applied to RISC architectures. The work was developed using the RISC-LIE architecture as our target [Vale91]. The RISC-LIE has been proposed and developed at IFQSC.

Part of the code was defined and constructed using YACC, a tool for compilers development which defined the gramatical structure of language. The remainder routines were developed using the C language.

The code produced by the compiler used the thirty instructions of the RISC-LIE instruction set . These instructions are implemented in the RISC-LIE architecture simulator. Therefore, generates codes, that can be interpreted by this simulator.

Capítulo 1

Introdução à Arquitetura RISC

Neste capítulo discutiremos as tendências que levaram ao surgimento dos conceitos da arquitetura RISC (*Reduced Instruction Set Computer*), introduzindo as características básicas desta arquitetura. Serão descritas as principais arquiteturas RISC, como RISC I [PaSe81], RISC II [Kate84] e MIPS [ChHo87] e, finalmente, a arquitetura desenvolvida por Valente [Vale91] no Instituto de Física e Química de São Carlos (IFQSC-USP), chamada RISC-LIE[1].

1.1 - Caminhos para uma nova arquitetura

Com a diminuição do custo de *hardware* e a necessidade de processamento de informações cada vez mais complexas, a tendência geral, com relação à arquitetura e organização de computadores direcionou-se para uma crescente complexidade das Unidades Centrais de Processamento (UCP), com grandes conjuntos de instruções. Com isso era possível expressar algorítmos de forma cada vez mais concisa. A diferença entre as operações previstas por linguagens de alto nível e aquelas características da própria arquitetura da máquina geram um fato conhecido como *Gap* semântico. Para diminuir este *Gap*, os projetistas de arquiteturas começaram a incluir um grande número de instruções, cada vez mais complexas, no desenvolvimento de novas máquinas. Um exemplo é a arquitetura VAX onde podemos citar as instruções *case*, além da possibilidade de 25 tipos diferentes de operação de soma.

Com a implementação de instruções complexas em microcódigo, tornou-se possível produzir instruções de alto nível mais complexas, o que facilitava muito a escrita de compiladores para uma mesma arquitetura ou pelo menos, assim se acreditava. Contudo, conjuntos complexos para serem bem aproveitados, exigiam que o usuário tivesse um conhecimento profundo de todo o conjunto de instruções, característica limitada a um grupo reduzido de projetistas e programadores de computador. Com isso, muitas instruções complexas caíram em desuso. Também, pelo fato de se direcionarem a um uso específico, eram mais frequentemente utilizadas em situações particulares. Por este motivo, é evidente

[1]RISC-LIE- *Reduced Instruction Set Computer* do Laboratório de Instrumentação Eletrônica

que a frequência destas instruções não era muito alta. Devemos observar que as instruções complexas podem ser implementadas usando-se várias instruções de um conjunto reduzido e simples que, além de poderem ser conhecidas por todos os programadores e projetistas de computador, têm uma frequência de uso elevada.

Ao longo dos anos, estudos de medição sobre instruções de programas sugeriram que um conjunto reduzido e simples de instruções é mais eficiente [Kate84] em relação a conjuntos mais complexos. Para efetuar a medida de eficiência, foram utilizadas diversas técnicas. O rendimento a ser medido depende do compilador, da máquina e do programa que será utilizado para efetuar a medida. Katevenis identificou três áreas de interesse sobre as quais é importante obter-se informações: operandos, operadores e execução. Em relação aos operandos devemos conhecer seu tamanho, estrutura e uso. Entre outras informações, será importante reconhecer se os operandos são constantes ou variáveis; se são números inteiros, de ponto flutuante, caracteres ou ponteiros; se são escalares, vetores, cadeia de caracteres (*string*), ou registradores; se são globais, locais ou argumento de função. Também será necessário conhecer a frequência com que são utilizados. No que se refere às operações, devemos conhecer sua frequência relativa e, em relação à execução devemos conhecer as estruturas de controle, se são saltos condicionais, saltos incondicionais, chamada (*call*) ou retorno (*return*) de funções.

Existem vários métodos de efetuar-se medições sobre programas. Podemos realizar uma medição estática, que se refere ao texto do programa, uma análise do tipo "caixa branca", que verifica, por exemplo, a quantidade de memória necessária sem medir a eficiência do programa. Uma outra forma é a medição dinâmica, que se refere à execução do programa. Um método também interessante é o que realiza um estudo sobre partes do código onde o programa passa a maior parte do tempo em execução [Stall88] e [Colw85].

No projeto RISC-I de Berkeley, um estudo feito com Pascal e C determinou quais os tipos de dados mais utilizados [PaSe82]. Este resultado pode ser visto na Tabela 1.1 Podemos observar que constantes inteiras são quase tão frequentes quanto arranjos (*arrays*) ou estruturas. Entretanto, a tabela não mostra que mais de 80% dos elementos escalares eram variáveis locais, e que mais de 90% dos arranjos ou estruturas eram variáveis globais.

Em um estudo da eficiência de compiladores feito por Wulf [Wulf81] foram usadas versões de compiladores C e Pascal para o VAX, PDP-11 e 68000, sendo determinada a média de instruções e referências à memória por comando. Multiplicando-se a frequência da

ocorrência de cada declaração pelo correspondente número de instruções de máquina e referências à memória foi construída a Tabela 1.2.

Tabela 1.1 - Porcentagem dinâmica de operandos em Pascal e C

	P1	P2	P3	P4	C1	C2	C3	C4	média
constante inteira	14	18	11	20	25	11	29	28	20 07
escalar	63	68	46	54	37	45	66	62	55 11
array/struct	23	14	43	25	36	43	05	10	25 14

P1 COMP - compilador Pascal estilo código P

P2 MACRO - macro do sistema SCALD I

P3 PRINT - programa de impressão para Pascal

P4 DIFF - programa que procura as diferenças de dois arq.

C1 PCC - compilador portável de C para VAX

C2 CIFPLOT - programa que plota máscaras de VLSI em *ploters*

C3 NROFF - formatador de textos

C4 SORT - utilitário sort do UNIX

Tabela 1.2 - Frequência relativa de comandos de HLL ("high level language") (ordenada por referências à memória)

comandos de alto nível (HLL)	alto nível (ocorrência)		peso (instr.máqu.)		peso (ref.mem.)	
	P	C	P	C	P	C
call/return	15±1	12±5	31±3	33±14	44±4	45±19
loops	5±0	3±1	42±3	32±6	33±2	26±5
assign	45±5	38±15	13±2	13±5	14±2	15±6
if	29±8	43±17	11±3	21±8	7±2	13±5
with	5±5	-	1±0	-	1±0	-
case	1±1	<1±1	1±1	1±1	1±1	1±1
go to	-	3±1	-	0±0	-	0±0

Os dados da Tabela 1.2 indicam que procedimentos *CALL/RETURN* sao operações que consomem a maior parte do tempo em programas escritos em linguagens de alto nível

(HLL). As estatísticas apontadas na Tabela 1.1 enfatizam a importância de variáveis locais e constantes. Podemos observar na Tabela 1.3 outras medições importantes sobre operações. Elas foram realizadas sobre a frequência de operações em forma de tipo de sentenças para instruções a nível de fonte, ou de códigos de operações ao nível de máquina.

Tabela 1.3 - Outras medições sobre operações

Instruções executadas dinamicamente	Medição	Referência
Instruções entre registradores e memória	40 %	[Lund77]
Instruções de salto	30 %	
soma/resto em ponto fixo	12 %	
load, load adress	33 %	[AbWo75]
store	10 %	
branch	14 %	
compare	6 %	
Sentenças de HLL contadas estatísticamente		
associações	43 %	[AlWo75]
if	13 %	
call	13 %	

Em medições realizadas sobre 300 procedimentos usando o conceito de programação estruturada, Tanenbaum [Tane78] verificou que as chamadas de procedimento como uma porcentagem de sentenças de HLL executadas dinamicamente representavam 12 % das medições. Lunde analisou uma dezena de programas de cálculo numérico (escritos em FORTRAN, Basic, ALgol e Bliss) e constatou que a administração das chamadas a procedimentos como uma porcentagem de tempo de execução era de 25%. Estes resultados sugerem algumas observações:

- as operações mais simples são também as que são executadas mais frequentemente.

- chamadas e retornos de funções são importantes devido ao tempo que se gasta com eles.

- as estruturas das sequências de controle (comparações e saltos) são importantes

devido à frequência com que aparecem *ifs* e *loops*.

Estes estudos tornaram claro que as linguagens de alto nível podem ser implementadas mais eficientemente se as tarefas que ocupam maior tempo forem otimizadas. Chegou-se também a outras conclusões que mais tarde dariam origem aos conceitos RISC. Por exemplo, as de que a quantidade de referência por instruções junto com a localidade e a predominância de referências a escalares causavam um grande número de referências à memória. Isto poderia ser melhorado com o aumento do número de registradores. A alta proporção de saltos condicionais e instruções de chamadas a funções sugeriu o uso de "pipeline" de instruções.

1.2 - Características da arquitetura RISC

"RISC é uma sigla para *Reduced Instruction Set Computer*, sendo um estilo de arquitetura de computadores que enfatiza simplicidade e eficiência" [Vale91]. No presente ítem usaremos como referência a arquitetura RISC I. Sua proposta tem por objetivo reduzir a complexidade do conjunto de instruções, exigindo uma maior otimização dos compiladores, de forma a maximizar o uso de registradores (que devem ser em grande número, da ordem de centenas), e minimizar o número de referências à memória.

É muito comum nas arquiteturas mais recentes incluir-se alguma característica que venha da filosofia RISC. O fato de uma máquina possuir algumas características RISC não nos permite afirmar que ela seja uma máquina RISC. Tipicamente podemos relacionar as seguintes características como sendo pertencentes a uma arquitetura RISC:

. Operações de ciclo único.

. Projeto LOAD/STORE registro a registro.

. Controle por *hardware*.

. Reduzido número de instruções e modos de endereçamento.

. Formato fixo de instruções.

. *Pipeline*

. Maximização do uso de registradores.

Estas características serão discutidas rapidamente a seguir.

Operações de ciclo único

As instruções RISC devem ser rápidas e simples. Executar uma instrução completa que necessita de vários ciclos requer uma combinação de certa lógica e microcódigo. Normalmente, implementar esta instrução completa será mais rápido se a mesma for implementada como uma sequência de instruções primitivas. No caso de um conjunto de instruções mais complexo implementar uma instrução em 10 níveis e for necessário um nível extra, a velocidade da CPU terá sofrido uma degradação de 10%. Como exemplo dessa complexidade, podemos citar as instruções de multiplicação e divisão em ponto fixo e ponto flutuante. A maioria dos estudos mostraram que as primitivas *LOAD, STORE, BRANCH, COMPARE, ADD* são as mais frequentes.

Projeto LOAD/STORE, registrador a registrador

A maioria das instruções das máquinas RISC envolvem operações entre registradores, sendo permitido o acesso à memória somente às instruções de LOAD e STORE. Isto permite uma simplificação no conjunto de instruções e na lógica de controle. Este conceito já foi explorado há algum tempo por Seymor Cray, durante a elaboração do projeto do CDC 6600 [Pino89].

Controle por "*Hardware*"

A simplicidade do conjunto de intruções reflete-se diretamente na unidade central de processamento. Projetos baseados em conceitos RISC ocupam entre 6 a 10 % da área de um chip VLSI para a decodificação instruções, enquanto que processadores 68000 e Z8000 consomem entre 50 e 60 % da área do chip para a mesma tarefa. Além do controle por *hardware* ser mais rápido que o microprogramado, é reduzido o tempo de projeto e ainda é possível usar a área economizada para outras aplicações.

Instruções e modos de endereçamento

Os modos de endereçamento usados são os mais simples. Caso seja necessário, os mais complexos podem ser construídos usando-se os mais simples. Com esta técnica é possível simplificar o conjunto de instruções e, em consequência, a lógica de controle.

Formato fixo das instruções

Uma característica comum a todos os projetos RISC é justamente ter uma formato fixo de instruções. Esta característica ajuda a simplificar a implementação, permitindo uma

decodificação simples e rápida.

Maximização do uso de registradores

Os projetistas de máquinas RISC consideram a forma de maximizar o uso de registradores sob diversos pontos de vista . Algumas soluções encontram-se no compilador e outras no próprio "hardware". A solução por compilador normalmente considera um algorítmo de coloração de grafos, onde a coloração do grafo é feita com um número fixo de cores e cada cor representa um número de registradores. A solução por *hardware* consiste em contruir um processador com um banco de registradores. O processador associa um novo banco de registradores a cada chamada a um procedimento, evitando ter que armazenar os registadores na memória.

Pipeline

Um conceito importante em arquiteturas de computadores que foi bastante explorado em arquiteturas RISC, e que constitui-se numa forma de melhorar o rendimento de um computador é a execução em *pipeline*. Existem diferentes esquemas de *pipeline*. Por isso é importante diferenciar entre instruções que normalmente requerem duas fases para serem executadas e instruções que normalmente requerem três fases (RISC II). Entre as instruções que requerem duas fases incluem-se busca de instrução e operação aritmética simples (*add\sub*) ou lógica ou de deslocamento. No caso de instruções de três fases incluem-se busca da instrução, calculo do endereço necessário para acessar a memória e operação de memória a registrador e de registrador à memória. No caso das arquiteturas RISC pode ser feito apenas um acesso à memória, o que provoca uma interrupção do *pipeline* quando alguma operação *LOAD/STORE* esteja fazendo acesso à memória. Isto exige que haja instruções em estado de espera. Esta situação é resolvida com a introdução de um NOP (*no operation*). Na Figura 1.1 é mostrado o esquema do *pipeline* da arquitetura RISC II, que prevê três estágios.

SERVIÇO DE BIBLIOTECA E INFORMAÇÃO - IFGSC
FÍSICA

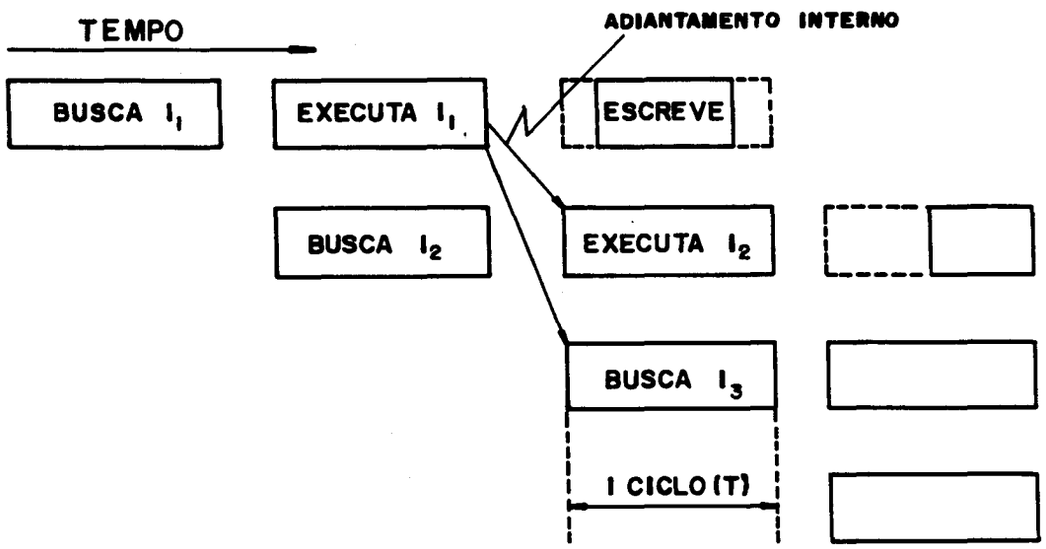


Figura 1.1 - *Pipeline* de três estágios (RISC II)

1.3 - Arquiteturas RISC

A arquitetura RISC é uma arquitetura do tipo formalizado por John von Neumann, ou seja sequencial com monoprocessador e com armazenamento interno de instruções, apesar existirem alguns estudos de aplicação destes conceitos em processamento paralelo. Neste ítem veremos algumas propostas da arquitetura RISC, e algumas implementações bem sucedidas dos conceitos da filosofia RISC, inclusive as que se utilizam de vários processadores.

1.3.1 - RISC I

A primeira arquitetura que sugeriu a utilização de um conjunto de instruções simples foi proposta em 1982 por Patterson e Séquin [PaSe82] em Berkeley. Apesar disso, como veremos adiante, arquiteturas anteriores já traziam alguns conceitos da filosofia RISC. O projeto do minicomputador IBM 801 [Radi83] indicou o caminho a ser seguido. Suas características principais já foram descritas anteriormente no ítem 1.2, sendo desnecessários maiores detalhes aqui.

1.3.2 - RISC II

A arquitetura RISC II [Patte85] foi proposta com base na RISC I, sendo uma arquitetura de 32 bits, com 32 bits para endereçamento e 8, 16 ou 32 bits para dados.

O conjunto de instruções é composto por 39 instruções, todas de tamanho de 32 bits, com campos em lugares fixos, como pode ser observado na Figura 1.2

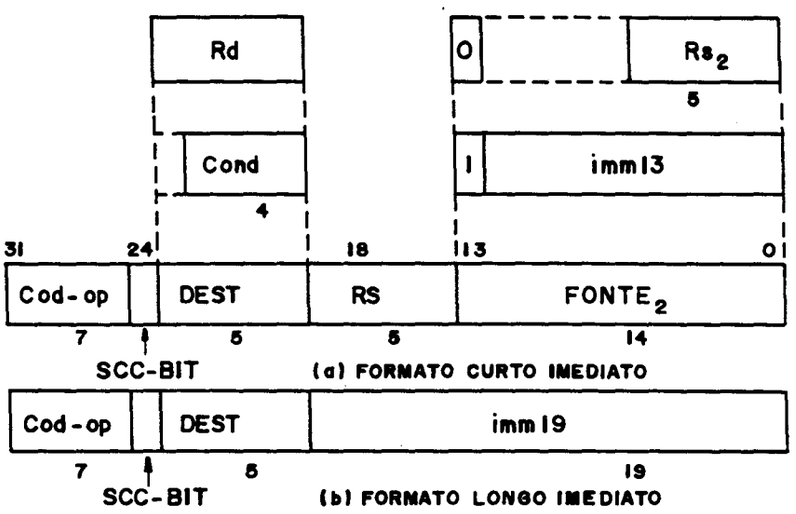


Figura 1.2 - Formatos de Instrução

O conjunto de instruções é formado por cinco grupos:

. 12 instruções que envolvem operações entre registradores (tais como soma de inteiro, resto de inteiro, operações booleanas).

. 10 instruções de LOAD (leitura de espaço virtual de endereços).

. 6 instruções de STORE (escrita de espaço virtual de endereço).

. 6 instruções de transferência de controle.

. 5 instruções diversas.

Na arquitetura RISC I o *pipeline* é constituído de duas etapas, superpondo a execução de uma instrução com a busca da instrução seguinte. Na arquitetura RISC II existe uma terceira etapa, que efetua a escrita no registrador destino (R_d) como mostrado na Figura 1.1.

Numa comparação entre as arquiteturas RISC I e RISC II notamos que existe uma grande diferença na via de dados, devido justamente a essa terceira etapa do *pipeline* no RISC II. Quando é feito um acesso à memória (LOAD/STORE) o *pipeline* é temporariamente suspenso, pois um acesso de busca não pode ser efetuado simultaneamente. RISC II possui uma única porta de acesso à memória e não possui memória *cache*. Outra característica da arquitetura RISC II são as janelas de registradores superpostas que estão organizadas em forma de um *buffer* circular, como mostrado na Figura 1.3. Na arquitetura RISC II temos 138 registradores de 32 bits, dos quais 10 registradores são globais. O outros estão organizados em 8 janelas de 6 registradores superpostos, 10 registradores locais e outros 6 superpostos. Para gerenciar as janelas ocupadas ou vazias são usados os ponteiros CWP (*current window pointer*), o qual aponta para a janela cujo procedimento está ativo, e SWP (*saved window pointer*) que identifica a última janela salva na memória.

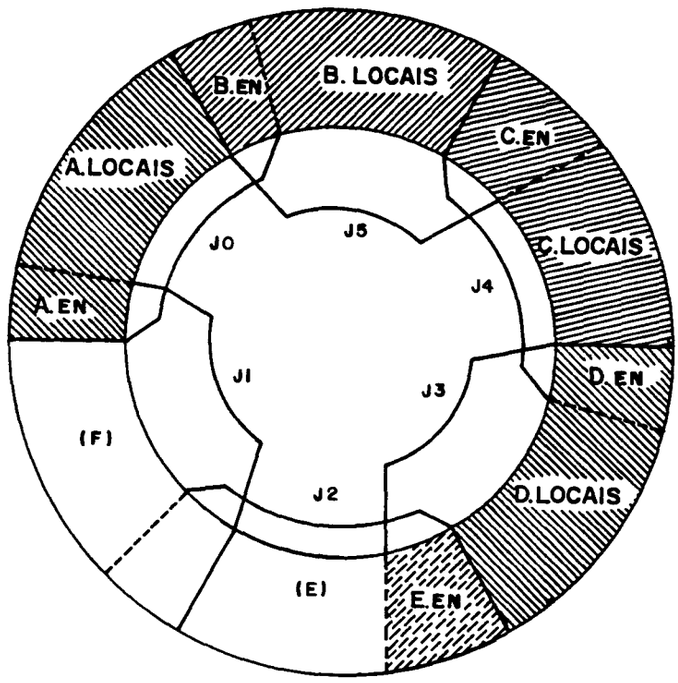


Figura 1.3 - Organização circular de janelas de registradores.

1.3.3 - IBM 801

Historicamente, uma das primeiras arquiteturas com algumas características RISC, surgiu na década de 70 como resultado do projeto da Watson Research Center. Este projeto, redescoberto em 1982, trata de um desenvolvimento de um controlador de comunicações de alta velocidade. Suas principais características envolviam execução de uma instrução por ciclo, acesso concorrente à memória, suporte de um compilador otimizador e um conjunto reduzido de 28 instruções com formato fixo de 32 bits, sendo que os operadores podiam ter meia palavra (16 bits) ou palavra inteira. Foram adotados 32 registradores na CPU especificados por 5 bits, resultando num esquema básico de 3 operandos, com dois de origem, e um de destino, todos distintos. A via de endereçamento também ocupa 32 bits, possibilitando endereçamento virtual.

Os operandos suportavam operações aritméticas de complemento de 2, operações lógicas tradicionais e de deslocamento. Os acessos à memória, como num processador RISC, realizavam-se por LOAD/STORE. O ciclo de busca e execução estava estruturado em um *pipeline* de 2 etapas, podendo ser acessado 3 registradores por ciclo de máquina. A primeira etapa decodifica a instrução, lê dois registradores e armazena em unidade aritmético-lógica e, finalmente, executa a operação. A segunda etapa armazena o resultado da operação, armazenando-o temporariamente em um registro destino e alterando os códigos de condição segundo haja correspondência.

1.3.4 - Projeto MIPS-X

Este projeto, desenvolvido na Universidade de Stanford nos primeiros anos da década de 80, foi o sucessor do projeto MIPS (*microprocessor without pipeline stalling*) [ChHo87]. Seu objetivo era o de melhorar seu antecessor, sendo uma arquitetura MIMD (*Multiple Instructions stream-Multiple data stream*), ou seja, tinha múltiplas vias de instruções e múltiplas vias de dados. Este projeto foi ainda tomado pelos projetistas de *hardware* como modelo no desenvolvimento de algumas tecnologias, tal como GaAs (arsenieto de gálio). O projeto gerou um processador com 32 bits, que utiliza bastante de *pipeline*, e de um compilador otimizador cujo objetivo é aproveitar o máximo das particularidades oferecidas pela arquitetura. A grande maioria das instruções são executadas em um ciclo de máquina de 50 ns. São também usadas algumas características não RISC, como a de ter uma cache interna ao processador de 2 Kbytes (Icache), organizada em um grupo associativo de 8 vias e 4 conjuntos, e 16 palavras por linha. Uma cache externa de 64 palavras (Ecache) é referenciada quando ocorre

falta de Icache, e esta por sua vez acessa a memória principal.

O *pipeline* é composto de cinco etapas: busca de instruções, decodificação de instruções e busca de registradores, operação na unidade lógica aritmética (ALU), espera/envio de um dado à memória e escrita do resultado em um registrador destino.

O endereçamento é feito de forma única, somando o conteúdo de um registrador a um deslocamento com sinal de 17 bits, produzindo um endereçamento de 32 bits. São utilizados um deslocador (*funnel-shifter* de 64-32 bits) e um registrador especial que armazena os resultados e operandos em execuções de multiplicação e divisão. Existem 31 registradores de propósito geral e um gravado com valor zero.

O processador permite a conexão de até 7 coprocessadores, sendo que a interface com os coprocessadores é feita através do conjunto de instruções. Suas operações podem ser de três tipos: operações com memória, operações de salto (*jump*) e operações de cálculo.

Este projeto foi finalizado em outubro de 1986, produzindo um integrado que era composto por cerca de 150.000 transítores e *clock* de 16MHz.

1.3.5 - Projeto SPECTRUN

Considerando as características da arquitetura RISC, este projeto seria o menos típico, possuindo um conjunto não reduzido de instruções. Foi desenvolvido para satisfazer necessidades diferentes, tais como a comercial, a científica e de engenharia, ao mesmo tempo que procurou melhorar a imagem dos computadores pessoais da Hewlett Packard [Maho86].

O sistema foi concebido com organização de memória hierárquica, com cache e memória principal, endereçada de forma virtual. O processador possui 8 registradores especiais utilizados para o endereçamento de memória virtual. Para o endereçamento são definidos três níveis. O nível 0, que não utiliza endereçamento virtual, tem tamanho 32 bits e é sempre endereçado fisicamente. O nível 1 trabalha com registradores especiais de 16 bits, de onde se obtém 48 bits de endereçamento lógico. O nível 2 tem registradores especiais de 32 bits. Existem 25 registradores de controle que armazenam o estado do sistema, bem como condições de proteção e gerenciamento de memória virtual. Os registradores de propósito geral são em número de 32 e não usam estruturas de janelas.

Apesar deste processador conter um conjunto complexo de instruções, apresentava

também particularidades importantes. Entre as mesmas pode-se destacar que este processador foi o primeiro, a nível comercial, a implementar manipulação de saltos de forma elaborada. A forma como esta implementação foi realizada inspirou posteriormente outros processadores.

1.3.6 - Processador Am29000

O Am29000 é um processador RISC disponível no mercado desde maio de 1988, fabricado pela Advanced Micro Devices, Inc. É um processador de propósito geral, sendo uma síntese de vários dos projetos anteriormente descritos.

A tecnologia empregada em seu desenvolvimento foi CMOS, para uma arquitetura de 32 bits e *clock* de 25 MHz, com 4 Gigabytes de endereçamento virtual. Apresenta ainda um co-processador numérico, 192 registradores de propósito geral, instruções de 3 operandos, endereçamento *pipeline*, acesso corrente de instruções e dados. Possui uma cache interna de 512 bytes para armazenamento de saltos (*jump*) e controlador de memória virtual interno ao processador com modo de usuário e de supervisor. O conjunto de instruções possui formato fixo de 32 bits, onde os 8 bits mais significativos são reservados para o código de operação e os demais para especificação do operando e do resultado.

São definidos 9 tipos de instruções para operações lógicas tradicionais e deslocamento. Existem três classes de registradores. Os de propósito geral, que constituem-se num conjunto de 192 registradores de 32 bits, podendo ser globais ou locais. Os registradores especiais são em número de 23, sendo 15 protegidos (acessados pelo supervisor) e 8 desprotegidos. Finalmente os registradores de tradução, são 128, agrupados em pares e constituem o centro da unidade de gerenciamento de memória.

É importante também observar que seu desenvolvimento foi direcionado para compiladores otimizadores, visto que o processador adota um modelo com grande número de registradores. Esta característica favorece a maioria dos compiladores que usam instruções de três operandos, e contam com a disponibilidade desses registradores em grande número. Outra importante característica para compiladores são as janelas de registradores, devido ao fato de permitir um bom gerenciamento de registradores locais e globais. Ambas as características foram exploradas pela arquitetura RISC-LIE.

1.3.7 - RISC/6000

O sistema RISC/6000 [Sach91], desenvolvido pela IBM, é uma evolução dos conceitos do projeto 801. A concepção desta arquitetura, que adota a tecnologia CMOS, tem duas características que distinguem o Sistema RISC/6000 de outros conjuntos de instruções reduzidas de microprocessadores: organização superescalar e alta performance, capacidade de representação em ponto flutuante.

Por superescalar pode-se entender uma máquina que, como uma simples máquina escalar, executa uma cadeia de instruções não-vetoriais, mas alcança elevada performance dividindo o trabalho entre unidades funcionais independentes, que operam em paralelo. Então, com uma unidade de execução em *pipeline* que permite execução de um ciclo, uma máquina escalar pode executar três instruções em três ciclos. Em comparação, uma máquina superescalar com três unidades independentes executam as mesmas três instruções em um ciclo, se elas são independentes. Os sistemas superescalares deste tipo podem melhorar substancialmente a quantidade de instruções por ciclo. Para atingir esta característica superescalar, o processador RISC/6000 foi dividido em três unidades de execução principais: ponto fixo, ponto flutuante e *branch*. Cada uma destas unidades opera em paralelo, com a unidade *branch* no controle global, e sendo responsável por garantir a integridade da execução do programa.

O registradores foram atribuídos para unidades funcionais, sendo que a maior parte das instruções são executadas inteiramente dentro de uma unidade. Quando o compilador requerer uma interação, por exemplo, se a unidade de ponto fixo gera códigos de condição que são usados pela unidade *branch*, ele faz o escalonamentro do código explícitamente para obtenção da máxima performance. Então, o processador expõe seu paralelismo latente, e espera que o compilador administre esse paralelismo.

A segunda caracterísitica que distingue o sistema é a ênfase no desempenho em ponto flutuante. No início do projeto foi concebido um projeto de unidade de ponto flutuante que permitiria que fosse completada, em cada ciclo, uma operação de multiplicação e adição de 64 bits. Para garantir que o acesso ao armazenamento não limitaria a performance em ponto flutuante, a máquina foi projetada para completar uma operação de carregamento ou armazenamento também em cada ciclo, em paralelo com as operações de ponto flutuante. A unidade de ponto fixo executa todas as operações de armazenamento, incluindo cálculos de endereçamento, o que permite completa sobreposição com as operações de ponto

flutuante. Para continuar melhorando o projeto 801, o conjunto de instruções deste foi revisado, e foram feitas mudanças. Em particular, foram incorporadas novas instruções de composição, como adição-multiplicação, *branch-on-count*, e *update-load*.

Estas duas características do sistema - organização superescalar e alta performance de operações de ponto flutuante -levam aos princípios básicos de projeto. Os princípios são oferecer a máxima sobreposição das três unidades funcionais, evitar ciclos mortos, e definir instruções que possam (na maior parte) ser completadas à taxa de uma por ciclo.

O Sistema RISC/6000 representa um avanço significativo sobre as idéias originais do 801: características de ponto flutuante foram incorporadas na arquitetura e concentradas nos aspectos paralelos da execução. Mas muitas das melhorias estão em detalhes que foram aprendidos no desenvolvimento de códigos de sistemas e compiladores para máquinas de estilo da 801

Ao contrário dos RISCs anteriores, a arquitetura do sistema RISC/6000 antecipa uma organização superescalar, particionando os registradores por função. O resultado permite o paralelismo nível-instrução, enquanto requer apenas coordenação limitada. Expondo as áreas de coordenação requerida, o sistema permite que o compilador gere código altamente otimizado que alcança, em alguns casos, a aproximação do máximo paralelismo teórico possível desta máquina.

1.3.8 - Outras arquiteturas e tendências da arquitetura RISC

No início do estudo da arquitetura RISC, algumas tendências se destacaram, sendo que algumas caminharam em direção de *hardware* (RISC I e RISC II), enquanto outras se desenvolveram orientadas para compiladores otimizadores (IBM 801 e MIPS). Mais recentemente estas duas tendências se combinaram, como no caso do processador Am 29000. Em busca de novas evoluções de *hardware* e de *software*, vemos a tecnologia RISC sendo aplicada em várias áreas, como descreveremos a seguir.

1.3.8.1 - Tendência de *hardware*

Atualmente, estações de trabalho da Intergraph utilizam o processador CLIPPER fabricado pela Fairchild. Este é um processador híbrido CISC-RISC, que possui duas caches de 4 Kbytes, uma para instruções e outra para dados. A estação de trabalho SUN, utiliza-se de um processador *SPARKS*, que implementa a arquitetura RISC II. Estes fatos refletem

uma tendência que vem se verificando ultimamente, onde é cada vez maior o uso de conceitos RISC, mesmo que não sejam utilizados num mesmo projeto todos os conceitos dessa arquitetura.

Podemos observar ainda a evolução no uso da tecnologia de Arsenieto de Gálio (GaAs), que apresenta alta estabilidade de temperatura e velocidade de comutação, permitindo frequências superiores a centenas de Mega Hertz, chegando até 10 GHz. Entretanto, seu custo ainda é alto devido a escassez do GaAS, e pelo fato dessa tecnologia não permitir um alto grau de compactação, apresentando assim uma baixa integração que não supera 25.000 a 30.000 transístores em uma mesma pastilha. Esta característica mostra-se ideal para RISC, no aspecto desta arquitetura utilizar-se de pastilhas menores do que as exigidas em arquiteturas menos reduzidas.

1.3.8.2 - Tendência de *software*

Serão considerados dois aspectos de *software*, para linguagens funcionais e programação de objetos, que serão discutidos a seguir.

1.3.8.2.1 - Linguagens funcionais

A arquitetura RISC também pode suportar outros paradigmas de programação, como por exemplo as linguagens funcionais, que ultimamente tem conquistado um grande número de adeptos. A arquitetura tradicional apresenta deficiências no suporte destas linguagens, que podem ser melhor suportadas pela arquitetura RISC. Podemos destacar que chamadas eficientes à funções podem ser bem suportadas na arquitetura RISC devido ao sistema de janelas característico. Este aspecto favorece também a manutenção do contexto, visto que toda chamada de função tem um contexto referencial associado que pode ser visto como um conjunto de valores (nome e valor).

Podemos também destacar a representação de listas e seu acesso, que deve provocar uma adaptação dos identificadores RISC para um tipo conhecido como *tagged architectures*, onde o dado passa a compor, junto com o identificador, um novo tipo de instrução. Finalmente, podemos citar o mecanismo de coleta (*garbage collector*) que detecta (onde são marcados os operadores referenciados) e recolhe os operandos (os operandos que não tem marca). Este mecanismo pode provocar mudanças substanciais na arquitetura para o processo de marcação que utiliza bits identificadores, criando um campo destinado ao recolhimento.

Considerando estes aspectos, por volta de 1986 em Berkeley, foi realizado o projeto SPUR (*symbolic processing under RISC*) que desenvolveu um suporte de linguagem funcional (*Common Lisp*) tomando como base a arquitetura RISC I, tendo ainda um coprocessador numérico. O projeto envolvia o uso de 6 a 12 processadores de memória hierárquica com três níveis. O primeiro nível com uma *cache* interna ao processador, o segundo nível com uma *cache* externa para dados e instruções. Finalmente, o terceiro nível com uma *cache* compartilhada por todos os processadores, tendo um endereçamento de até 256 Gigabytes, divididos em 256 segmentos de 16 bits por endereço, onde cada endereço possui 32 bits convertido a endereçamento global de 38 bits.

É importante observar que a proposta deste projeto provocou modificações arquitetônicas consideráveis. Manteve-se o uso de 8 janelas superpostas de 32 registradores com 6 registradores por janela, porém com a presença de 7 registradores especiais de 32 bits que armazenam a palavra de estado do sistema, dos usuários ponteiros de janelas e contadores de programas. Outra diferença se deu a nível de *pipeline* que envolve quatro etapas. São elas a busca da instrução, leitura da saída de ALU, acesso à cache e escrita. Modificações no conjunto de instruções também foram feitas [Hill86].

Na avaliação deste projeto foi usado um compilador *Splice* de Lisp, que, de modo similar ao projeto IBM 801, usou um algorítmo de coloração de grafos para otimização de código. Para isto foi usado um *benchmark* típico de estrutura funcional, conhecido por Gabriel.

1.3.8.2.2 - Programação com objetos

Em Berkeley, no início dos anos 80, foi também considerada a possibilidade da implementação de linguagem SMALLTALK 80 em RISC, num projeto conhecido como SOAR (*SMALLTALK on a RISC*). Devido ao alto custo de implemetação, o único projeto satisfatório foi o processador Dorado, com tecnologia ECL, desenvolvido pela Xerox. O projeto considerou que em 95 % dos casos os contextos de SMALLTALK necessitavam de mais de 8 registradores [Samp85]. Isto indicou o uso de dois grupos de janelas de registradores, registradores altos que contém as variáveis locais e os argumentos de contexto corrente, e o grupo de registradores baixos utilizados para preparar os argumentos de chamada.

Como na arquitetura RISC II, cada contexto dispõe de 32 registradores globais especiais (altos e baixos). O projeto usa a mesma solução SPUR para a verificação de tipos de dados com o uso de marcas (*tags*). O formato dos dados é composto pelo campo

marcador (*tag*), por um campo chamado *garbage* que perde o significado quando trata-se de um inteiro, e o campo para o próprio dado (28 bits). O *pipeline* é de três estágios (busca, operação e escrita), sendo que cada estágio está dividido em três etapas. No primeiro estágio são buscadas 3 instruções. No segundo estágio são feitas, em três fases, a precarga, leitura e o acesso a ALU. No terceiro estágio é realizado apenas uma fase de escrita.

1.4 - A arquitetura RISC-LIE

A arquitetura RISC-LIE inspira-se na proposta da arquitetura RISC-II de Berkeley, mas é uma arquitetura do tipo Harvard, com barramento e *cache* para dados e instruções separados. A micro arquitetura e a tabela de instruções da arquitetura proposta desenvolvida por Valente [Vale91] pode ser vista no Apêndice C. Para testar esta proposta, foi desenvolvido um simulador de eventos discretos usando linguagem C, apropriado para modelamento de sistemas de computação [Ston75]. O algorítmo básico deste simulador é fornecido abaixo.

Inicializa estrutura de dados
Incializa eventos iniciais
Enquanto (existem eventos)
    Seleciona próximo
    Atualiza próximo
    Executa evento (podendo gerar novo(s))
    Coleta estatística
Calcula estatísticas
Gera relatório

Neste tipo de simulação, o tempo é incrementado quando ocorre algum evento. No caso do simulador RISC-LIE, o término de um evento se dá quando todas as tarefas realizadas durante o período de um ciclo de máquina se completava. O simulador permitiu a obtenção da frequência em que cada instrução foi executada e do número de ciclos de máquina gastos por todas as instruções na execução de um programa. Permitiu também o uso opcional de "caches", que podiam ter seu tamanho alterado. Uma opção que pode ser usada é a variação da velocidade do processador, alterando-se assim a frequência (em MHz). O simulador desenvolvido apresentou um código bastante compacto e rápido, tornando-o somente 90 vezes mais lento que o computador usado para simulação [Vale91].

Para análise do desempenho, Valente usou um consagrado "benchmark" intitulado Dhrystone, desenvolvido por Reinhold P. Wecker [Werc84], de implementação simples. Este *benchmark* já foi aplicado a cerca de 300 máquinas diferentes, com resultados amplamente divulgados [Inte89]. O Dhrystone apresenta em suas declarações uma porcentagem de 51% de atribuições, 32.4% de declarações e 16.7% de chamadas de procedimento.

Capítulo 2

Construção de Compiladores

Neste capítulo descreveremos conceitos referentes à linguagem de programação e compiladores, ressaltando aspectos importantes na construção de compiladores. Destacaremos as funções internas de um compilador, descrevendo cada fase de sua anatomia. As fases de otimizações e geração de código serão melhor detalhadas no capítulo 4. Será abordado também o uso de ferramentas de *software* que apoiam o desenvolvimento de compiladores, tais como o **YACC** e **BISON** [Donn88].

2.1 - Linguagens de programação e compiladores

2.1.1 - Conceitos básicos

Os computadores operam manipulando correntes eletrônicas na forma de pulsos elétricos, que correspondem a dígitos binários. As pessoas expressam-se usando, no seu dia a dia, uma forma de comunicação própria, chamada **linguagem natural**, que é totalmente diversa da linguagem do computador. Esta última, denominada **linguagem de máquina**, é de difícil compreensão para o homem. Isto gera uma barreira natural que dificulta a comunicação entre homem e máquina. Para diminuir estas diferenças foram criadas as **linguagens de programação**. As linguagens evoluíram no sentido de abranger uma semântica cada vez mais próxima da linguagem natural, e são chamadas **linguagens de alto nível** (HLL - *high level language*). Elas permitem que o homem escreva textos próximos à sua linguagem, os quais através de dispositivos tradutores, podem ser entendidos pelos computadores. Como uma definição para linguagem de programação, podemos dizer que ela é uma coleção de cadeias de símbolos, de comprimento finito. Essas cadeias são chamadas sentenças da linguagem e são formadas pela justaposição de elementos individuais, que são **símbolos**, **átomos**, ou ***tokens*** da linguagem.

Uma linguagem pode ser representada de três formas. Uma delas é através da enumeração de todas as cadeias e símbolos possíveis que formam as sentenças. Outra forma é através das regras de formação das cadeias reunidas em um conjunto de regras

chamado **gramática**[1]. A forma efetivamente adotada entretanto, é a que se utiliza de um **reconhecedor**[2] que atua através de regras para aceitação de cadeias. As cadeias que serão reconhecidas pertencem a duas classes de substituição - os símbolos não substituíveis, chamados **terminais**, e símbolos que podem ser substituídos, chamados **não terminais**.

Os tradutores são programas que recebem como entrada um programa escrito numa determinada linguagem L1 e geram um programa numa linguagem L2. Se a linguagem L1 é de alto nível e a L2 é a linguagem de máquina, esse tradutor é chamado de **compilador**, conforme ilustrado no diagrama da Figura 2.1. O programa escrito na linguagem L1 é chamado de **programa fonte**, e o programa escrito na L2 é chamado de **programa objeto**.

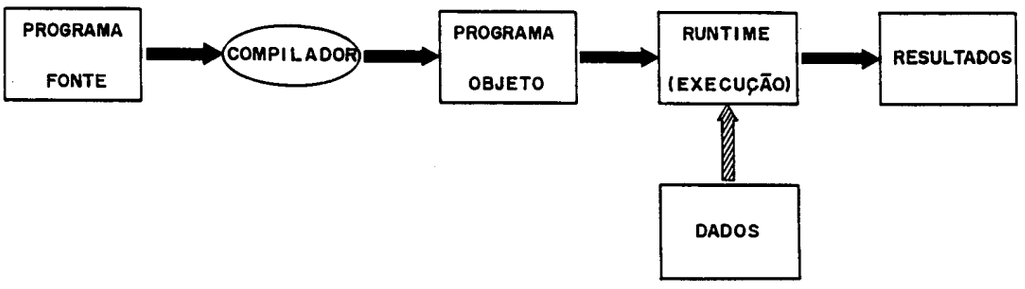


Figura 2.1 - Esquema de execução de um programa.

Tendo o programa objeto gerado pelo compilador, pode-se tomar seu código e submetê-lo à execução. O programa executável é um arquivo que contém o programa em linguagem de máquina. Para isso é usado um ***LINKER***, que incorpora no programa objeto, rotinas pré-definidas que estão armazenadas num **arquivo de biblioteca**. Para que

[1] Gramáticas são quádruplas ordenadas [Ahou77]. Elas são formadas na primeira ordem pelo vocabulário, e na segunda ordem pela representação dos elementos do vocabulário, que são símbolos ou átomos que formam as sentenças da linguagem. Na terceira ordem, são formadas pelo conjunto de leis de formação utilizadas para definir a linguagem. Finalmente, na última ordem são formadas pelo elemento não terminal que irá dar início ao processo de geração de sentenças, por isso chamada de símbolo inicial da gramática.

[2] Através de reconhecedores é possível submeter uma cadeia de símbolos a um teste de aceitação capaz de determinar se tal cadeia pertence ou não a linguagem em questão. O reconhecedor pode ser uma máquina de Turing, no caso da gramática pertencer ao tipo 0 (gramática irrestrita [Neto87]). Pode também ser uma máquina de Turing com memória limitada, podendo assim reconhecer gramáticas do tipo 1 (gramáticas sensíveis a contexto [Neto87]). Quando a gramática é do tipo 2 (livre de contexto [Neto87]), os reconhecedores são os autômatos finitos [Neto87]. Finalmente para o tipo de gramática que será desenvolvido em nosso compilador, que é uma gramática linear à esquerda (LR), do tipo 3, o reconhecedor é um autômato finito.

finalmente, o programa seja executado, utilizamos o **carregador** (*LOADER*), que irá colocar na memória o programa executável. Em certos casos existem ainda **gerenciadores de biblioteca,** que permitem que seja feita a inclusão ou exclusão rotinas pré-definidas na biblioteca. O diagrama mostrado na Figura 2.2 ilustra o esquema descrito.

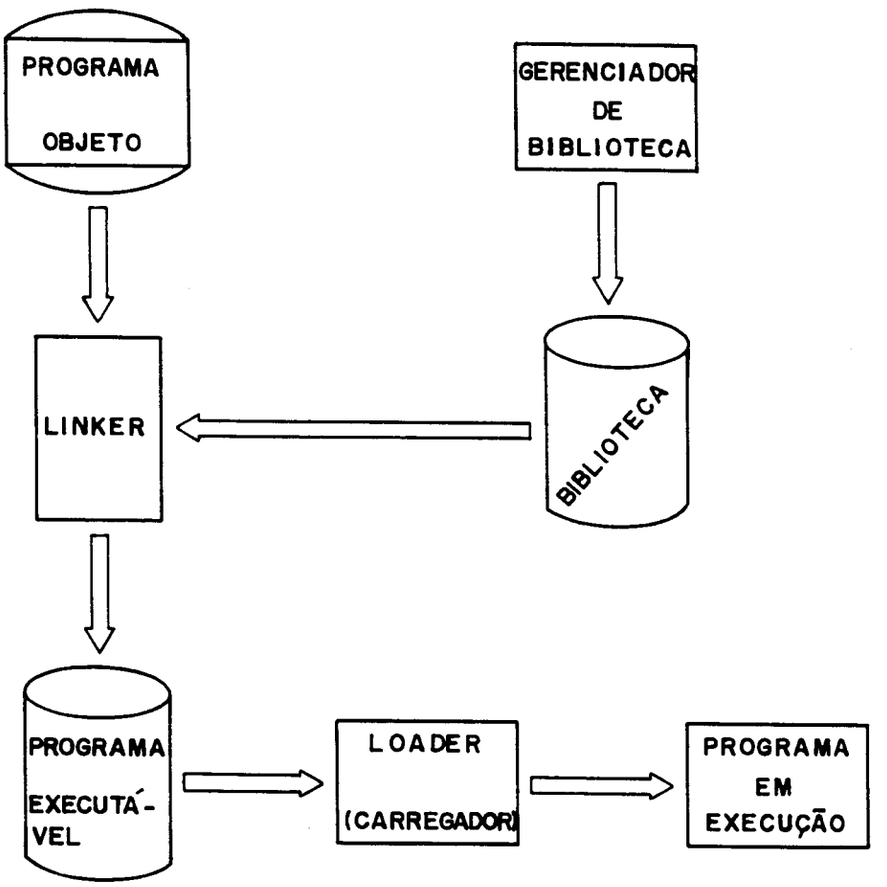


Figura 2.2 - Esquema de execução de um programa objeto.

A técnica de compiladores realizou notáveis avanços nos últimos trinta anos. Desde a década de 50, com o surgimento da linguagem FORTRAN (1954-57), desenvolvida por J. Backus visando fornecer apoio para a computação numérica, as linguagens buscam aumentar

o poder dos programadores. Para isso os projetos de linguagens de programação aumentam cada vez mais a capacidade semântica das mesmas. Com a crescente necessidade de processamento de informação de modo rápido e confiável, surgiram muitas outras linguagens. Como exemplo, podemos citar as linguagens LISP [GheJ87], direcionada para computação simbólica; ALGOL 68 [GheJ87], direcionada para programação de sistemas; Pascal [Coop83], uma linguagem de propósito geral e de ensino, suportando a programação estruturada, e também a linguagem C [KeRi88], desenvolvida em 1974 nos laboratórios da Bell por D. Richtie, que é muito utilizada na programação de sistemas.

A linguagem C será a linguagem utilizada na implementação do programa desenvolvido no presente trabalho. Os diversos motivos da escolha da linguagem C envolvem, entre outros, as qualidades da própria linguagem, tais como a sua portabilidade, que permite o desenvolvimento do compilador em ambiente IBM-PC, e posteriormente o seu transporte para o ambiente de uma *workstation* SUN, onde foi gerado o simulador RISC-LIE. Outro motivo refere-se à utilização da ferramenta *BISON*, como será visto adiante, que usa arquivos de entrada e saída em linguagem C.

Existem outras soluções para a implementação de linguagens de programação. Uma das formas é o uso de um tradutor que decodifica unidades básicas, executando-as imediatamente, ao invés de traduzir o programa fonte completo para só então executá-lo, como é feito pelo compilador. Este tipo de tradutor é chamado **Interpretador**. Algumas linguagens normalmente implementadas através de interpretadores são Basic [GheJ87], APL [GheJ87], LISP [GheJ87] e Smalltalk [GheJ87], além de diversas linguagens próprias para bancos de dados, como DBase III [Jone87] .

Existem também tradutores que atuam como **filtros**, fazendo a conversão entre duas linguagens de alto nível, quando a linguagem objeto é semelhante à linguagem fonte. Os filtros são bastante utilizados para resolver problemas de conversões de padrão entre diferentes implementações de linguagens de alto nível. Um exemplo disso é a conversão de um programa fonte em FORTRAN IV em nível G da IBM, para um programa fonte equivalente em FORTRAN IV padrão ANSI [Neto87].

Os compiladores proporcionam muitas facilidades para os usuários. Entre as mais importantes, podemos citar a de se escrever programas numa linguagem próxima à utilizada no dia a dia. Outra vantagem é a portabilidade, ou seja, a possibilidade de migrar um programa fonte para diversas máquinas diferentes, o que representa uma diminuição na

diferença entre máquinas. Com isso, uma linguagem de alto nível pode ser utilizada em diversos tipos de máquinas, o que evita a necessidade de se conhecer uma linguagem específica para determinada máquina. A facilidade na depuração e manutenção de programas e a melhoria na interface homem-máquina são outros fatores, bem como a redução de custos no treinamento de pessoal especializado, e a redução de custos de sistema devido á redução do tempo para desenvolvimento de programas. Finalmente, deve ser lembrado o aumento da confiabilidade do código gerado, permitindo a construção de algorítmos mais modulares e simples, e a eliminação de algorítmos mais complexos de difícil implementação e manutenção.

As linguagens de programação são influenciadas por estilos de programação e ferramentas de construção de programas, que formam metodologias de desenvolvimento de processos. Com isso, podem ser estabelecidos os requisitos que a linguagem deve ter. Já a arquitetura de computadores exerce influência na direção oposta, no sentido de restringir os projetos de linguagens de programação ao que as máquinas da época permitem implementar-se eficientemente. Em consequência, pode-se dizer que as linguagens têm sido restringidas pela arquitetura de Von Neumann, que ainda hoje é a arquitetura utilizada na maioria das máquinas disponíveis.

2.1.2 - Aspectos de construção de compiladores

Uma linguagem deve ter uma definição precisa antes de ser implementada. Em primeiro lugar, devem ser especificadas todas as sequências de símbolos que constituem programas válidos para uma linguagem. Somente estas sequências serão aceitas pelo compilador. Desta forma, estaremos realizando a especificação da sintaxe da linguagem. A segunda etapa é a definição do significado associado a cada construção da linguagem, que compõe a sua semântica.

Embora existam muitas notações para a definição de linguagens, a **Forma Normal de Backus**, ou **BNF** (*Backus-Naur Form*) [Ahou77], tem se destacado devido a sua grande facilidade de compreensão. A BNF é uma meta-linguagem, ou seja, é uma linguagem usada para especificar outras linguagens. Sendo uma notação recursiva de formalização da sintaxe de linguagens através de **produções gramaticais**[3], cada produção corresponde a uma regra

[3] Conjunto de regras de substituição de um símbolo por outro, permitindo assim a criação de dispositivos de geração de sentenças.

de substituição, em que a cada símbolo da metalinguagem é associado uma ou mais cadeias de símbolos, indicando as diversas possibilidades de substituição. No presente trabalho utilizaremos a notação BNF, não só pela razão apresentada, mas também por ser necessária como entrada para o **YACC** [Coop83], que servirá de base para o desenvolvimento do compilador deste trabalho.

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

1- Produzir um código objeto eficiente;

2- Produzir códigos objetos pequenos;

3- Minimizar o tempo necessário para compilar programas;

4- Manter o compilador tão pequeno quanto possível;

5- Produzir um compilador com boas capacidades de diagnóstico e recuperação de erro;

6- Produzir um compilador confiável.

Na verdade, alguns objetivos acima são conflitantes. Um compilador será mais lento se o objetivo principal for a produção de um código eficiente. Algumas técnicas de otimização, como a eliminação de código redundante e a remoção de sequências de códigos de ciclos, entre outras, podem consumir um tempo demasiadamente grande. Compiladores que façam uso de tais recursos devem também ter um tamanho maior do que outros que não possuam rotinas otimizadoras.

Uma decisão importante na construção de compiladores deve ser o número de passagens que ele executará. É contada uma passagem cada vez que o compilador lê o programa fonte completo. Do ponto de vista da simplicidade, os compiladores de uma única passagem são atraentes. No entanto, nem todas as linguagens permitem uma compilação deste tipo. Do ponto de vista estrutural, pode ser mais correto e simples realizar várias fases distintas de compilação em passagens separadas. O compilador desenvolvido no presente trabalho, que irá implementar um subconjunto da linguagem PASCAL, usa o recurso de compilação em uma única passagem.

No desenvolvimento de um compilador devem ser considerados alguns aspectos fundamentais, tais como a sintaxe e a semântica da linguagem, bem como o aspecto

referente à integração com a máquina hospedeira. No aspecto sintático, cuidamos de cada frase, que deve seguir uma gramática definida e que irá definir a linguagem de forma precisa antes dela ser implementada. Em primeiro lugar, são especificadas todas as seguências de símbolos que constituem os programas válidos para uma certa linguagem, que deverão ser aceitas pelo compilador. Estamos assim criando a especificação da sintaxe da linguagem. A seguir devemos cuidar da definição do significado associado a cada construção da linguagem, que é a especificação semântica.

No aspecto semântico devem ser traduzidas e refletidas as ações do programa fonte para a construção do programa objeto. As regras que determinam a semântica de uma linguagem são muito mais difíceis de serem determinadas do que as regras sintáticas. Devido à dificuldade de se definir regras semânticas, uma vez que não é encontrado um padrão que especifique as semânticas de linguagens, a construção de compiladores corretos é bastante dificultada. Cada linguagem tem uma particularidade, o que exige um grau de significação diferente, interferindo diretamente na semântica desta linguagem.

Finalmente, o último aspecto a ser considerado é o problema da integração do compilador com a máquina hospedeira. Devem ser avaliadas a influência da arquitetura básica, linguagens de máquina e sistemas operacionais, onde são tratados os problemas de gerenciamento de memória e manipulação de arquivos.

2.2 - Anatomia de um compilador

Os primeiros compiladores construídos não apresentavam uma estrutura definida e uniforme. Somente com a evolução de técnicas de planejamento e construção de compiladores surgiram estruturas definidas. Através do acúmulo de experiência, observação de resultados, e, principalmente pelo desenvolvimento de teorias relacionadas a tarefas da análise e síntese de programas, foi possível obter um acordo acerca da estrutura adequada para o desenvolvimento de compiladores.

Além de atuarem como tradutores, os compiladores ainda atuam de diversas outras formas, tais como na detecção de erros, inclusao de comentários, tratamento de macros e impressões de listagens. O funcionamento básico de um compilador está indicado de forma esquemática na Figura 2.3.

SERVIÇO DE BIBLIOTECA E INFORMAÇÃO - IFQSC
FÍSICA

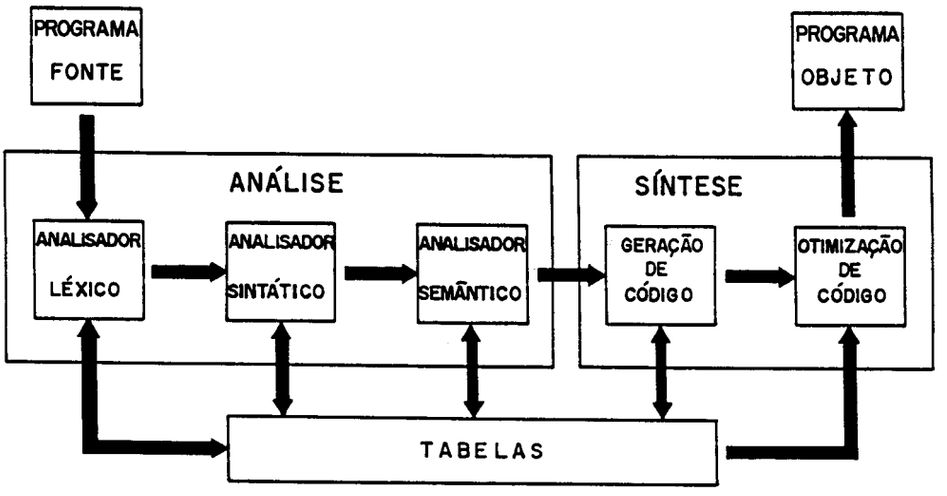


Figura 2.3 - Esquema do funcionamento básico de um compilador.

Podemos observar que existem duas fases. A primeira é de análise, que subdivide-se em outras três: análise léxica, análise sintática e análise semântica. A segunda fase é de síntese, e subdivide-se em otimização global, geração de código e otimização local. Apesar de, conceitualmente, a fase de análise ser anterior à fase de síntese, na prática elas ocorrem quase em paralelo. Podemos dividir um compilador (ou o trabalho de compilação?)em seis fases: análise léxica, análise sintática, análise semântica, otimização global, geração de código e otimização local [Kowa83]. As tabelas do compilador são consultadas durante todas as fases. A tabela de palavras reservadas, que contém os caracteres especiais e palavras reservadas da linguagem, é fixa. A tabela de símbolos, entretanto, é criada a partir do próprio programa fonte que está sendo analisado. A seguir analisaremos cada uma destas etapas em maiores detalhes.

2.2.1 - Analisador léxico

A análise léxica é a primeira das três fases que compõe a análise do programa-fonte (*source*). Um analisador léxico, também chamado *parser*, cumpre uma série de tarefas, não somente relacionadas à análise léxica. A principal função deste analisador é fragmentar o

programa fonte de entrada em trechos elementares completos, bem caracterizados e com identidade própria. Estes componentes básicos são chamados ***tokens***. Do ponto de vista da implementação do compilador, o analisador léxico atua como uma interface entre o programa fonte e o analisador sintático, convertendo a sequência de carcteres que constituem o programa na sequência de átomos que o analisador sintático consumirá. Os átomos constituem-se em símbolos terminais da gramática. O analisador léxico do compilador desenvolvido neste trabalho segue um processo extensamente estudado, sendo sua função varrer o programa fonte da esquerda para a direita, agrupando os símbolos de cada ítem léxico e determinando sua classe. Esta estrutura direciona o projeto para a construção de uma máquina abstrata, conhecida como **autômato finito**, de simples implementação, como será visto no Capítulo 5, no qual é descrito o compilador construído.

Algumas funções internas do analisador léxico são descritas a seguir:

. **Extração e classificação de átomos**: Esta é a função mais clara e imediata do analisador léxico, ou seja, mapear o programa fonte em outro texto, formado por átomos representados pelos símbolos que compõem o programa fonte. As classes de átomos mais encontradas em analisadores são: palavras reservadas, números inteiros sem sinal, números reais, cadeias de caracteres, sinais de pontuação, de operação, caracteres especiais, símbolos compostos e comentários, entre outros.

. **Eliminação de delimitadores e comentários**: Os comentários, caracteres brancos e delimitadores são de fundamental importância para os programadores, mas totalmente irrelevantes para a geração de código. Devem, portanto, ser eliminados pelo analisador léxico.

. **Conversão numérica**: Existe uma diversidade muito grande quanto à notação em que os números podem ser escritos, tais como: inteiros decimais, binários, octais, hexadecimais, inteiros de múltipla precisão, números reais em ponto fixo, em ponto flutuante, notação científica. Alguns compiladores tratam exclusivamente de números inteiros, decompondo as demais sequências de números inteiros e sinais de pontuação que os compõem, em sequencias inteiras que possam ser tradados internamente. Outros tratam todos os tipos, encarregando-se de interpretá-los adequadamente. Em qualquer caso, o analisador léxico tem uma atividade de conversão numérica, que mapeia internamente a representação que será manipulada pelo compilador.

. **Tratamento de identificadores:** O tratamento é semelhante ao dado para números, visto que os identificadores também são cadeias de tamanho variável. Sua representação é feita através de uma **tabela de símbolos**[4] que, por simplicidade, reserva para cada símbolo um comprimento constante.

. **Identificação de palavras reservadas:** Esta função encarrega o analisador de encontrar um identificador no programa-fonte. Neste caso, é preciso verificar se tal identificador pertence a um conjunto de identificadores especiais com significado pré-definido pela linguagem, que são as palavras chaves, ou palavras reservadas. Como, do ponto de vista formal, não existe nenhuma distinção entre identificadores e palavras reservadas, o compilador separa os identificadores em duas etapas: reconhece o átomo como identificador e depois verifica se o identificador é ou não palavra reservada. Em geral, a técnica utilizada é percorrer uma tabela, uma lista, ou uma árvore que represente o conjunto de palavras reservadas pela linguagem.

. **Recuperação de erros:** Quando ocorrem caracteres não identificados, ou sequências que não obedecem a lei de formação da linguagem na leitura do programa fonte pelo analisador, diz-se que existem erros léxicos neste texto. Para que o compilador possa prossequir na análise, apesar da perda de sincronismo no autômato que implementa o analisador léxico, são usados mecanismos de resincronização, denomindados **mecanismos de recuperação de erros.** Uma opção seria percorrer o texto em busca de um delimitador, que pode ser um sinal de pontuação ou palavra reservada, descartando os caracteres intermediários. Uma outra opçao é separar os átomos não reconhecidos e enviá-los para serem tratados pelo analisador sintático.

. **Listagens:** Pelo fato do analisador léxico estar efetuando a leitura física do programa-fonte, ele efetua a emissão da listagem, apesar dessa operação não se tratar de uma análise. Isto evita a transferência de fragmentos do programa fonte para os demais módulos do compilador.

. **Geração de tabelas de referências cruzadas (TCR):** O analisador léxico, tendo os átomos e as linhas do programa fonte a sua disposição, pode armazenar informações

[4] Uma vez extraídos do texto fonte pelo analisador léxico, os identificadores são colocados em uma tabela, sendo a cada um destes símbolos associado um código único. A cada ocorrência de um identificador a tabela é consultada. Se o identificador já constar da tabela, ela é apenas consultada, ou é registrada a ocorrência com finalidade de construir uma tabela de referências cruzadas. No caso de não constar o identificador, a tabela é alterada e ele é inserido.

acerca da ocorrência dos símbolos, coletando-as em uma tabela. Desta forma pode ser gerada uma listagem indicativa dos símbolos encontrados, com referência à localização de sua ocorrência.

. **Controle de listagens**: Existem, entre os comandos de controle do compilador, aqueles através dos quais o programador tem acesso á rotinas que efetuam a geração de listagens. Entre estas rotinas estão as que permitem ao programador ligar e desligar opções de listagem, mapeamentos de memória, coleta de símbolos em tabelas de referências cruzadas e formatação das saídas impressas.

. **Interação com sistema de arquivos**: Ao analisador léxico compete acessar o arquivo de leitura do programa fonte, via sistema operacional. No caso geral este procedimento é simples, porém pode existir a possibilidade do programador especificar, através de comando no compilador, o chaveamento de vários arquivos. O programa fonte é então construído pela justaposição destes arquivos. Neste caso, torna-se necessário construir uma interface mais elaborada com o Sistema Operacional.

. **Compilação condicional**: Fazendo uso de comandos de controle o programador pode parametrizar textos fonte. Isto torna possível a configuração de suas necessidades, selecionando, segundo condições, o texto a ser compilado ou a ser omitido. Desta forma o compilador atua como um pré-processador.

2.2.2 - Analisador sintático

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

Para a representação de notação da gramática de uma linguagem, uma forma que vem ganhando muitos adeptos, devido a legibilidade de seu aspecto gráfico, são os

diagramas sintáticos. Esses diagramas caracterizam-se por dois pontos diferentes: o ponto de partida e o de chegada. Os dois pontos são ligados entre si por um grafo orientado, cujos nódos representam pontos onde, eventualmente, podem ser feitas escolhas, e cujas arestas representam caminhos de percurso. Cada ramo pode incluir um terminal ou naõ terminal da gramática. Os diagramas da linguagem que são de interesse para este trabalho são mostrados no apêndice A.

A seguir serão relacionadas algumas das principais funções do analisador sintático.

. **Identificação de sentenças**: O analisador sintático pode ser visto como aceitador de cadeias. O conjunto dessas cadeias forma a linguagem que é reconhecida pelo analisador.

. **Detecção de erros de sintaxe**: No caso de ser detectada uma sentença que não pertence à linguagem, o analisador sintático deve identificar sua ocorrência, acusando erros de sintaxe. Uma forma de se fazer isto é 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 possível, sobre a causa deste erro.

. **Recuperação de erros**: Nos mecanismos de análise sintática de muitos compiladores são incorporados meios de ressincronização do reconhecedor, sempre que forem encontradas construções não pertencentes à linguagem. Isto é importante para que a análise não pare de ler o programa fonte, e, na continuação, possam ser encontrados todos os erros possíveis de serem detectados pelo compilador.

. **Montagem da árvore abstrata da sentença**: Apesar de dispensável, podendo ser substituída por outras atividades equivalentes menos onerosas, poderiam ser levantadas (ao menos conceitualmente. com base na gramática), as sequências de derivação da estrutura para a cadeia de entrada, compondo assim, a árvore abstrata de cada sentença.

. **Comando de ativação do analisador léxico**: Em função do progresso do reconhecimento do programa fonte, em muitas implementações o analisador sintático detecta a necessidade de novos átomos a serem reconhecidos. Desta forma é comandada a análise léxica do programa fonte em função da sintaxe.

. **Ativação de rotinas da análise referente às dependências de contexto da linguagem**: Muitas vezes denominadas de análise semântica estática, apesar de

serem atividades puramente sintáticas, estas rotinas cuidam da verificação de tipos de variáveis, da coerência de tipos de dados em expressões, do relacionamento entre as declarações e dos comandos executáveis.

. **Ativação de rotinas de análise semântica:** É o controle, feito em tempo de compilação, das rotinas responsáveis pelo controle dos objetos da linguagem, alocação da área para os mesmos, ligação para os objetos e comandos que os manipulam.

2.2.3 - Analisador semântico

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

A partir destas informações, podemos afirmar que a principal função da análise semântica é criar, a partir do programa fonte, uma interpretação deste texto, expressa em alguma notação adequada que é, geralmente, uma linguagem intermediária do compilador. Esta operação é realizada com base nas informações existentes nas tabelas construídas pelos outros analisadores, tabelas de palavras reservadas, mapas e saídas dos demais analisadores. Deve ser observada a fundamental importância de uma análise semântica bem realizada, visto que toda a síntese estará embasada na interpretação gerada por este analisador. Entre as ações semânticas encontramos tipicamente as seguintes:

. **Criar e manter a tabela de símbolos:** Em geral esta tarefa é executada pelo analisador léxico. Muitos compiladores passam a incorporar essas tarefas junto ao analisador semântico como uma forma de simplificar os analisadores léxico e semântico.

. **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. Devem ser indicadas todas as características que tal objeto exige e que sejam de interesse para o processo de geração de código.

. **Manter e verificar a informação sobre o escopo dos identificadores:** Esta função refere-se, principalmente, ao problema da localidade dos objetos, muitas vezes ligada à dinâmica do comportamento do programa em tempo de execução. Para que seja feita a associação dos identificadores aos escopos correspondentes, atribuindo-lhes o significado correto em todos pontos do programa fonte, a tabela de símbolos é organizada de forma hierárquica em relação aos escopos a que pertencem os identificadores nela registrados. Através da consulta à informação sobre o escopo em que um identificador está referenciado, o compilador deve executar procedimentos que garantam que os identificadores correspondam a objetos definidos nos pontos em que ocorreram.

. **Representar tipos de dados:** Um conceito bastante utilizado em linguagens mais recentes é o de oferecer um grande número de tipos, em geral representados por agregados homogêneos (tabelas), ou heterogêneos (estruturas). Alguns permitem que o programador crie novos tipos de dados. Neste caso caberá ao analisador semântico a tarefa de registrar as especificações dos diversos tipos de dados.

. **Analisar restrições quanto à utilização dos identificadores:** A coerência de utilização de cada identificador deve ser verificada, para cada situação específica, com relação aos atributos exibidos pelo mesmo.

. **Identificar declarações contextuais:** Refere-se aos mecanismos de interpretação adequada de identificadores que são declarados de modo implícito, e não através de construções sintáticas específicas. Um caso refere-se a identificadores pré-declarados, cuja semântica faz parte da própria linguagem de programação, outro caso refere-se a objetos com valores iniciais pré-estabelecidos (*default*).

. **Verificar a compatibilidade de tipos:** A verificação de tipos é um recurso auxiliar para as atividades de geração de código, uma vez que o uso de dados de tipos diferentes, embora possa ser coerente, impõe 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.

. **Efetuar o gerenciamento de memória:** Os requisitos de memória devem ser calculados, controlados e previstos em tempo de compilação. Tais requisitos podem ser estáticos, quando se referem às necessidades de memória que são fixas quando da execução do programa, e dinâmicos, quando se referem às necessidades de memória durante a execução do programa.

. **Representar o ambiente de execução dos procedimentos:** Esta função refere-se à interpretação do contexto de execução das rotinas. É importante, neste caso, o mecanismo de passagem de parâmetros (por valor, por endereço, por referência).

. **Implementação de um ambiente de programação:** Há necessidade de existir um suporte para a execução de um programa, que geralmente está implementado juntamente com a biblioteca de rotinas de execução e de uma interface com o sistema operacional. Em alguns casos, são incorporadas aos ambientes de execução ferramentas que permitem medidas de desempenho, para efeito de testes ou depuração.

. **Comunicação entre dois ambientes de execução:** São mecanismos que têm por objetivo permitir que procedimentos comuniquem-se entre si (através da passagem de parâmetros), e que trechos de programas pertencentes a escopos diferentes possam coexistir.

. **Geração de código:** Apesar de ser uma função de síntese, a geração de código, em primeira instância, ocorre já no analisador semântico. O código gerado, muitas vezes, é um código intermediário e relocável, que permite uma grande portabilidade. Este ítem terá um tratamento especial no Capítulo 4, onde será discutido juntamente com o ítem referente à otimização de código.

2.2.4 - Otimização global

Esta otimização, onde são efetuadas as manipulações da árvore sintática com a finalidade de reduzi-la, é independente de máquina. Nesta classe encontram-se as otimizações de expressões, a eliminação de subexpressões comuns, a fatoração do cálculo de subexpressões de uso frequente, as otimizações de construções iterativas. Este tópico será abordado em maiores detalhes no Capítulo 4.

2.2.5 - Geração de código

Podemos dizer que este é o objetivo final de qualquer compilador. Uma vez gerado e otimizado, o código estará pronto para ser executado. Devemos considerar que essa tradução implica na geração de um código que preserve a semântica do programa original. A tradução pode ser feita em um ou mais passos, criando códigos intermediários que podem ser concebidos em níveis diferentes. Os códigos intermediários são criados no caso de um compilador de múltiplos passos ou de um compilador cuja saída seja um programa objeto executável em uma máquina abstrata.

O uso de linguagens intermediárias facilita a otimização de código gerado e aumenta a portabilidade do programa-objeto, tornando o compilador menos dependente da máquina. Isso se deve ao fato de que o código intermediário gerado, pode ser lido por um montador, para cada máquina onde é executado, e codificado para a linguagem de máquina da mesma. Podemos ter diversas formas intermediárias para o código-objeto. Dentre as formas mais conhecidas, podemos citar a **notação polonesa**, que originalmente era apenas usada na compilação de expressões e que foi estendida para abranger construçoes não aritméticas. Outra forma intermediária corresponde à expressão do programa-objeto como uma sequência de **ênuplas**, cada qual representando uma operação completa: operandos e operadores. Em geral, fixa-se o número de elementos da ênupla em três ou quatro, obtendo-se, respectivamente, triplas e quádruplas. Podemos ainda usar a forma intermediária de **árvores**, que, por ser de fácil manipulação, mostra-se uma boa alternativa. Estas formas serão discutidas no Capítulo 4.

2.2.6 - Otimização local

Nesta fase final, a máquina na qual vai ser executado o programa influi na otimização a ser feita pelo compilador. É feita a otimizaçào do uso de registradores, do uso do conjunto de instruçoes de máquina, do tempo de execução do programa. Maiores detalhes serão apresentados no Capítulo 4.

2.3 - Um exemplo do funcionamento de um compilador

Como exemplo do funcionamento de um compilador, vamos analisar um simples cálculo do perímetro de um triângulo. A primeira etapa consiste na análise do problema pelo

programador. Se o problema é o calculo de uma fórmula matemática, deve ser feita uma abstração matemática desta situação, que é a seguinte:

Perímetro = Lado1 + Lado2 + Lado3

O próximo passo é converter os elementos da fórmula em variáveis associadas aos mesmos. Para Lado1 usaremos "b", para Lado2 usaremos "c", para Lado3 usaremos "d" e finalmente, para Perímetro usaremos "a". Em linguagem computacional (alto nível), a fórmula de perímetro é representada pela seguinte equação:

a = b + c + d

O próximo passo do compilador será ler e identificar os símbolos, separando cada elemento. O analisador acessa a tabela de símbolos e palavras reservadas, montando assim uma relação entre os elementos e *tokens* da linguagem. Os *tokens* para este caso são: a,=,b,+,c,+,d,fim_de_linha. O caracter 'fim _de_linha' é desprezado, assim como todos os espaços em branco que não interessam à compilação. A etapa seguinte será a análise sintática, que verificará se as variáveis a,b,c,d são reconhecidas. Para isto, o analisador sintático consulta a tabela interna de símbolos ou declarações explícitas criada pelo analisador sintático. Deve-se observar que os analisadores estão continuamente em contato entre si, sendo mutuamente ativados. O passo seguinte é a verificação das operações "=" e "+", que devem ser reconhecidas como pertencentes à gramática da linguagem. Posteriormente é necessário identificar o que deve ser construído. Neste exemplo, deve-se abstrair o sentido da operação que se deseja efetuar. Trata-se de uma análise semântica muito simples; o analisador semântico constrói uma linguagem intermediária, que deverá realizar a operação desejada e ser entendida pela máquina. Neste caso, o significado (semântica) representado pela equação do perímetro é a soma do Lado1 com o Lado2, o resultado desta soma é adicionado ao Lado3. Em uma linguagem intermediária isso seria representado da seguinte forma:

T1 = b + c
T2 = T1 + d
a = T2

Com esta etapa realizada, o compilador passa para a fase de síntese, que produzirá

a linguagem *assembly*[5] e, dependendo do compilador, efetuará duas otimizações, global e local. Na primeira fase da síntese, o compilador irá buscar uma forma de otimizar, se possível, as sentenças obtidas pela análise semântica. Esta fase é chamada otimizaçao global. A sentença acima ainda não foi otimizada. Uma possível otimização seria a eliminação das duas últimas sentenças em favor de uma que as substituísse, como pode ser visto abaixo:

T1 = b + c

a = T1 + d

O passo seguinte seria a produção de linguagem *assembly* para o texto acima. Caso a linguagem a ser produzida fosse para a arquitetura RISC_LIE[6], que pode ser vista em [Vale91], o código que seria produzido, a partir do texto não otimizado seria:

```
ldxw[7] b
add     c
stxw[8] t1
ldxw    t1
add     d
stxw    t2
ldwx    t2
stxw    a
```

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

```
ldxw b
add  c
add  d
stxw a
```

[5] código produzido pela linguagem de montagem (*assembler*)

[6] Apenas as palavras da máquina abstrata, e não seu formato, foram usados de forma simplificada.

[7] ldxw - carrega palavra (correspondendo a *load*)

[8] stxw - armazena palavra (correspondendo a *store*)

A linguagem *assembly* gerada é um **mneumônico** de código binário, que é entendido pela C.P.U. da máquina. Como as C.P.U.s são dispositivos eletrônicos que operam sob forma de tensões elétricas em dois níveis (0 se não há tensão e 1 se há tensão), a linguagem *assembly* corresponderia a uma sequência de 0's e 1's. Se a palavra tivesse um tamanho de 32 *bits*, teríamos as seguintes sentenças em linguagem de máquina:

0101 1000 0001 0000 0000 0010 1101 1001
0101 1010 0001 0000 0000 0000 1011 0000
0100 0100 1111 0010 0100 0000 0000 0000

Cada computador usa um **mneumônico** particular com padrão interno. Podemos imaginar uma **linguagem simbólica** para uma determinada máquina, com palavra de 4 *bits* da seguinte forma:

código binário	mneumônico
0000	load
0001	store
0010	mov
0011	add
.	.
.	.

2.3 - Ferramentas de desenvolvimento de compiladores

Para auxiliar a construção de compiladores, têm sido especialmente desenvolvido um conjunto de ferramentas apropriadas [Kern84] [Trem85]. São os geradores de compiladores (*compiler-generators*), os sistemas de tradução-escrita (*translator-writing systems*) e os compiladores de compiladores (*compiler-compiler*). A este último grupo pertencem, entre outros, o YACC e o Bison. O uso de geradores de *parser* aumenta a confiabilidade dos compiladores, visto que baseia-se em uma definição formal clara e sem ambiguidades.

2.3.1 - Os interpretadores YACC e BISON

O **YACC** (*Yet Another Compiler Compiler*) é um interpretador de linha de comando, que cria automaticamente um reconhecedor de linguagem de comando para uma dada

sintaxe. É importante destacar que o analisador léxico deverá ser escrito em separado, e incorporado ao programa de entrada junto ao conjunto de regras sintáticas estabelecidas. Embora tenha sido criado especialmente para ambientes UNIX, existem hoje versões para microcomputadores pessoais compatíveis com a linha IBM-PC.

No ano de 1988 foi apresentado um *software* semelhante e principalmente compatível ao YACC, o **BISON** [Coop83]. O **BISON** executa a mesma função, e é um *software* de domínio público, que pode ser usado livremente. Assim como o **YACC**, o **BISON** pode ser usado para o desenvolvimento de analisadores, desde simples calculadoras até complexas linguagens de programação. Qualquer especificação feita para **YACC** pode ser usada sem alterações para **BISON**. Existem, porém, algumas vantagens do **BISON** sobre o **YACC**, que se devem ao fato do **BISON** ser uma implementação mais recente e com recursos mais completos do que o **YACC**. Durante este trabalho foi usada uma versão do **BISON** de 1990, a qual nos referimos como **YACC**. Por esse motivo, a partir de agora quando citarmos o **YACC**, estamos nos referindo ao **BISON**. Sua semelhança com o **YACC** é muito grande, o que pode ser visto no Manual resumido, apresentado no Apêndice B.

A linguagem a ser descrita pelo **YACC** é do tipo livre de contexto, em um conjunto de tabelas para uma automação simples que executa um algorítmo LR(1), que permite a especificação de um ou mais grupos sintáticos, tais como expressões, identificadores e outros. Para a especificação desta linguagem livre de contexto é utilizada a forma **BNF**, que é a mais comum. Apesar da definição da gramática permitir a ocorrência de algumas ambiguidades, são utilizadas regras de precedência para eliminá-las.

O **YACC** então irá processar o arquivo **BNF** com as regras gramaticais de uma linguagem especificada, produzindo como saída uma implementação desta linguagem escrita em linguagem C, que deverá ser compilada para obtermos o reconhecedor da linguagem especificada. A linguagem especificada poderá ser, por exemplo, Pascal ou um subconjunto de outra linguagem de comandos. O código gerado poderá ser compilado em qualquer máquina que possua o compilador C, mesmo que não seja em ambiente UNIX, porque é independente da máquina, uma vez que o **YACC** fornece todas as rotinas de entrada e saída.

O YACC é considerado uma ferramenta de CASE (*Computer Aided Software Engineering*) bastante flexível, que permite que os arquitetos de linguagem concentrem-se no projeto da linguagem e não na sua implementação. A operação se resume à definição da linguagem usando **BNF**, obtendo o código de trabalho como saída. Esta ferramenta usa como

entrada um arquivo ("y_tab.h") onde são especificadas e definidas as cadeias (*tokens*) da gramática da linguagem. Na saída é gerado um arquivo em liguagem C ("y_tab.c"), que ao ser compilado irá gerar o analisador ("yparse"), sendo necessário serem carregados um programa de análise léxica ("yylex"), o programa principal e o programa ou rotina de tratamento de erros (yyerror). Pode ser usada ainda uma rotina de depuração ("YYDEBUG") opcional. Estas etapas serão apresentadas a seguir em maiores detalhes.

O YACC é assim, um gerador de "*parser*" (parte do compilador que faz a análise do código), ou seja, um analisador. Desta forma, o YACC converte uma especificaçao de uma gramática em um analisador que processará as declarações desta linguagem. Os etapas a serem seguidas no uso do YACC são as seguintes:

1- A gramática é escrita, de uma forma precisa, usando a forma de Backus-Naur (BNF). Nesta fase estamos especificando a sintaxe da gramática. O YACC pode ser usado para prevenir erros e ambiguidades da gramática. Por exemplo, se desejarmos construir um analisador de operaçoes matemáticas simples (quatro operações) deve ser escrita a seguinte BNF :

```
        lista     :  /* nada */
                   | lista '\n'
                   | lista expressão '\n'
          ;
expressão :   número
            | expr '+' expr
            | expr '-' expr
            | expr '*' expr
            | expr '/' expr
            | '(' expr ')'
            ;
número    : 0,1,2,3,4,5,6,7,8,9 ;
```

Em outras palavras, uma lista é uma sequência de expressões, cada uma seguida por outra linha. Uma expressão é um número, ou um par de expressões unidas por um operador ou colocadas entre parênteses. O número, por sua vez, está determinado pelos terminais 0,1...9. Esta formalização permite ao analisador reconhecer expressões como :

SERVIÇO DE BIBLIOTECA E INFORMAÇÃO - IFQSC
FÍSICA

1 + 2

4 - 3

3 / 5

(3 - 5) * (4 / 6)

e outras do mesmo tipo, e rejeitar expressões do tipo

-3 + 5

devido ao fato de não ter sido definido o operador unário menos três (-3).

2 - Cada regra de produção[9] deve ser acrescentada com a ação. Esta ação deve conter uma declaração do que deve ser feito quando for encontrada uma forma gramatical no programa que está sendo analisado. A parte que corresponde ao que fazer é escrita em linguagem C, sendo então definida a semântica da linguagem.

3 - Deve ser usado um analisador léxico, necessário para ler a entrada que está sendo analisada e levantar os ítens significativos para o analisador sintático. Por exemplo, um número é um ítem léxico (*token*), composto por vários caracteres numéricos. Outro exemplo são os operadores, formados por apenas um caracter, de soma (+), subtração (-) e outros.

4- É necessária uma rotina de controle para chamar o analisador sintático construído pelo **YACC**. O **YACC** processa a gramática e a ação semântica em uma função ("**YYPARSE**"), e escreve em um arquivo escrito em linguagem C. Caso não sejam detectados erros, o analisador léxico, o analisador sintático e a rotina de controle podem ser compilados e executados. Podem também, caso necessário, ser ligados a outras rotinas de linguagem C. A função deste programa é chamar diversas vezes o analisador léxico para extrair *tokens*, reconhecer a estrutura gramatical (sintaxe) na entrada e executar a ação semântica associada a cada regra gramatical reconhecida. A entrada para o analisador léxico é feita através da função "**YYPARSE**", e, quando esta função necessita de um novo *token* faz uma chamada recursiva.

A entrada para o **YACC** deve ter o seguinte formato:

---

[9] Regras de produção ou regras de aceitação - conjunto de estruturas a partir das quais pode-se verificar a derivação das sentenças de uma determinada gramática.

```
%{
        declarações em C (include,statement...) - Opcional
        }%
        declarações YACC (tokens, variáveis gramáticais, precedência, informações de associatividade)
        %%
        Regras gramáticais e ações
        %%
        outras declarações em linguagem C - opcional
        main () {...YYPARSE()...}
        YYLEX() {...}
        ...
```

Esta entrada é processada pelo **YACC** e o resultado é armazenado no arquivo "**Y_TAB.C**", que possui declarações em linguagem C entre %{ e }%, se existirem, declarações em linguagem C após o segundo %%, se existirem, e as rotinas de "**YYPARSE** e "**YYLEX**". Deve ser lembrado que o analisador léxico chama o "**YYLEX**". Podemos notar que o código gerado pelo **YACC** é em linguagem C.

Capítulo 3

Programação de um sub-conjunto da linguagem PASCAL

Neste capítulo serão apresentados os motivos da escolha da linguagem PASCAL Padrão na implementação do compilador desenvolvido para a arquitetura RISC, bem como os conceitos básicos desta linguagem e seus comandos essenciais. Será apresentado o sub-conjunto escolhido e implementado, fornecendo-se também alguns detalhes deste conjunto.

3.1 - Pascal padrão

3.1.1 - Introducão

Uma primeira discussão refere-se à escolha da linguagem que melhor se adequaria à proposta inicial deste trabalho - a construção de um compilador para uma máquina RISC. Foram consideradas diversas linguagens, tais como C, Basic, Fortran e Pascal entre outras.

A linguagem C, que sempre exerce grande atração devido ao seu poder de programação, oferecia dificuldades muito grandes no atual estágio do projeto RISC-LIE, devido a existência de algumas características internas de difícil construção. Como exemplos, pode-se citar a necessidade do desenvolvimento de um linker que possa construir e montar o código gerado em RISC com as funções de biblioteca da linguagem, e uma ferramenta indispensável para o funcionamento do compilador, que é o pré-processador C. Considerou-se que este desenvolvimento foge aos objetivos deste trabalho em sua proposta atual, descartando-se a utilização da linguagem C.

As demais linguagens, quando comparadas ao Pascal não apresentam nenhum atrativo ou vantagem significativa. A linguagem Basic por exemplo, por se tratar de um interpretador, não permite o desenvolvimento referente à parte de otimização de código, ponto de fundamental importância dentro do presente trabalho. O Fortran, a despeito do esforço de seus projetistas em atualizá-la, mantém ainda hoje, características de uma linguagem criada na década de 50 e não prima pela melhor interface e facilidade de utilização. Concluímos daí, ser a linguagem Pascal aquela que mais se adequa às nossas necessidades.

A linguagem Pascal foi desenvolvida em 1968 por Niklaus Wirth, pesquisador do Instituto de Informática da ETH (Eidgenossische Technische Hochschule) em Zurique [GheJ87]. O nome escolhido por Wirth para a sua linguagem foi dado em homenagem ao matemático e filósofo Blaise Pascal (século 17), que foi também o inventor de uma máquina de somar montada com engrenagens e rodas. A linguagem é considerada altamente estruturada. O objetivo da estruturação é dar apoio ao desenvolvimento de programas de

acordo com um conjunto de regras rígidas, de modo a reduzir os problemas de testes e aumentar a produtividade e a legibilidade dos programas resultantes [YOUR75]. Segundo Wirth [WIRT78], o desenvolvimento do compilador PASCAL surgiu da necessidade de se dispor de uma linguagem para o ensino de programação. As principais dificuldades encontradas nas demais linguagens são as características e estruturas frequentemente sem explicação. Além disso, os conceitos utilizados pela linguagem na qual o estudante aprende a expressar suas idéias influencia profundamente seus hábitos, raciocínio e criatividade.

A facilidade de aprendizagem do PASCAL é dada pela forma renovadora introduzida pela programação estruturada, onde esta linguagem é assumidamente uma das pioneiras, o que lhe conferiu grande popularidade. Em outras palavras, o Pascal apresenta várias características: é *fortemente tipado*, encoraja *projetos top-down*, é *estruturado*, é *orientado a procedimentos*, é *modular*.

Encontrando atualmente grande aceitação entre programadores de nível médio, a linguagem vem sofrendo alterações ao longo dos últimos anos, e muitas versões têm surgido para programadores de microinformática. Estas versões foram desenvolvidas pela Borland, e aceitas pela sua grande facilidade de uso com um ambiente integrado de ferramentas conhecido como "Ambiente Turbo Pascal". Este ambiente, posteriormente exportado para outras linguagens de programação, além de facilitar o uso da linguagem de forma mais rápida, forneceu um grande número de novas ferramentas e principalmente comandos de alto nível. Surgiram também bibliotecas associadas que trouxeram muitas vantagens à programação, evitando que o programador tenha que escrever uma grande quantidade de funções de tela, de estrutura de dados e estruturas básicas de bancos de dados. Entre essas bibliotecas podemos citar uma particularmente muito usada, a BGI (Borland Graphics Interface), destinada a usos na Computação Gráfica básica. Atualmente a linguagem Pascal vem recebendo alterações que visam extendê-la para a programação orientada para objetos. Toda essa vantagem inicial, entretanto, é mantida apenas quando o ambiente de execução dos programas é de micro informática. Estas versões do referido "Ambiente Turbo" não existem para outras plataformas. Com isso podemos concluir que a portabilidade desses programas fica reduzida à ambientes de microcomputadores pessoais, e, mais especificamente, aos da linha IBM PC.

Uma vez determidada a escolha da linguagem Pascal, restou ainda a escolha da versão que melhor responderia às exigências da proposta inicial. Optou-se pelo desenvolvimento de um compilador que atendesse ao maior número de máquinas possível, e onde, no futuro, pudéssemos efetuar uma comparação de desempenho. Para isso seria

necessária a escolha de uma versão que contivesse um conjunto mínimo de instruções fornecidos por todos os compiladores Pascal. Assim, a abordagem deste trabalho resumiu-se à linguagem Pascal Padrão. Por Pascal padrão (ou *Standard*), entendemos um conjunto reduzido e completo de instruções básicas, que permitam a construção de programas estruturados. Em linhas gerais, consideramos que todas as linguagens de programação devem permitir estruturas sequenciais, estruturas condicionais, e estruturas repetitivas. Além disso, deve existir a possibilidade de se criar módulos isolados em uma estrutura na forma de sub-programas. Outra característica importante a ser considerada é a criação de diversos tipos de dados, simples e estruturados.

A seguir serão descritos, de forma resumida, os conceitos essenciais encontrados em uma linguagem Pascal Padrão (**Standard Pascal - User Reference Manual** [CoopP83]), usada como referência no presente trabalho. Esta descrição será feita com o objetivo de caracterizar a linguagem adotada, em face às muitas versões disponíveis no mercado. Não é nossa intenção fornecer um manual detalhado, mas delimitar quais e como são os comandos implementados, e diferenciá-los dos possíveis comandos de outras versões. A gramática completa desta linguagem, descrita pela BNF, pode ser vista no Apendice A.

3.1.2 - Descrição geral

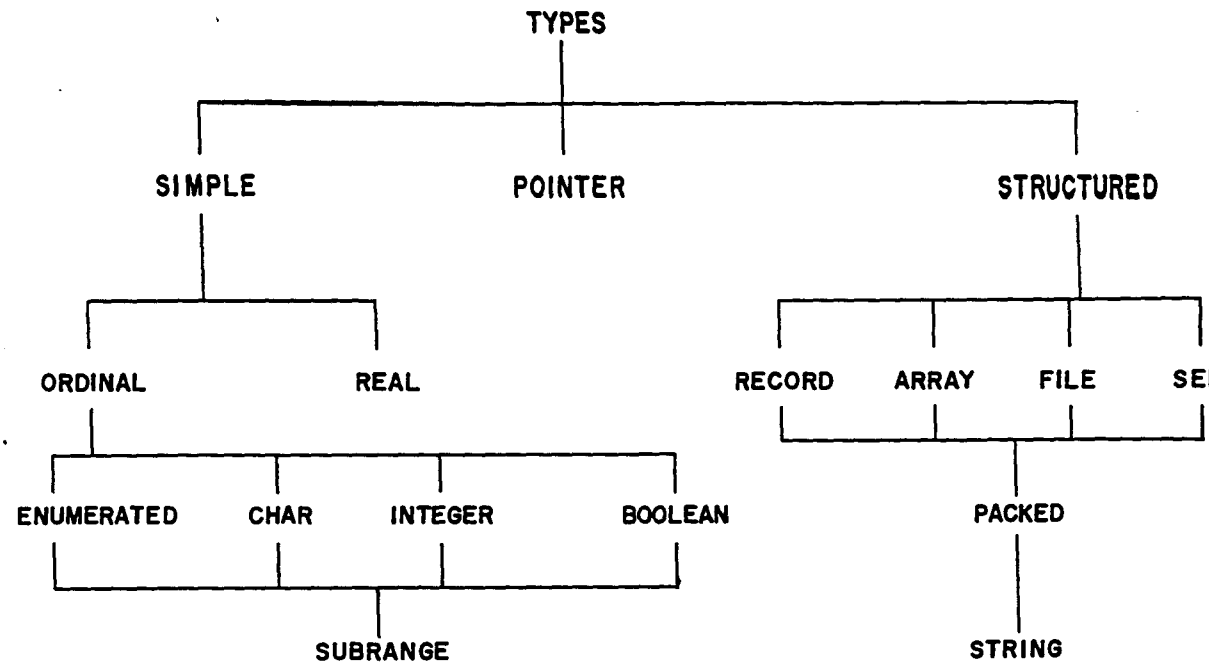
Quando nos referimos ao Pascal Padrão, somos logo levados a pensar no primeiro conjunto de comandos implementadas em 1974 por Niklaus Wirth. Porém, [Coop83] descreve um Pascal Padrão novo, padronizado pelas normas ISO nível 0. Esta normas são equivalentes às já aprovadas pelo ANSI (*American National Standards Institute*) e pelo IEEE770X3.97-1983 (*Institute of Electrical and Electronics Engineers*).

As palavras reservadas da linguagem são trinta e cinco, listadas abaixo:

AND	DOWNTO	IF	OR	THEN
ARRAY	ELSE	IN	PACKED	TO
BEGIN	END	LABEL	PROCEDURE	TYPE
CASE	FILE	MOD	PROGRAM	UNTIL
CONST	FOR	NIL	RECORD	VAR
DIV	FUNCTION	NOT	REPEAT	WITH
DO	GOTO	OF	SET	WHILE

Além destas palavras, fazem parte da linguagem os tipos de dados: real, char, integer, boolean, e text. Assim como todas as versões da linguagem Pascal, o Padrão permite uma série de tipos de variáveis, tais como simples (*simple*), ponteiros (*pointer*) e estruturados (*structuted*). Uma variável do tipo simples pode ser real (*real*) ou ordinal (*ordinal*). O tipo ordinal pode ser sub-limitado (*subranged*), enumerado (*enumerated*), caracter (*char*), inteiro

(*integer*) e lógico (*boolean*). Os tipos estruturados permitidos são registro (*record*), arranjo (*array*), arquivo (*file*) e conjunto (*set*), que podem formar ainda pacotes compactados (*packed*) de cadeias (*string*). Esta estrutura pode ser vista abaixo:



Também faz parte da linguagem um conjunto de funções pré-definidas (Abs(x), cos(x), trunc(x), ord(x), sqr(x), arqtan(x), round(x), chr(x), eoln(f), eof(f), sqrt(x), ln(x), succ(x), pred(x), sin(x), exp(x)), e um conjunto de procedimentos (*procedure* - read, readln, write, writeln, rewrite(f), reset(f), put(f), get(f), page(f), new(p), dispose(p), pack, unpack).

A linguagem Pascal Padrão permite que os comandos sejam estruturados de forma sequencial e em blocos, separados por comandos de início (begin) e fim (end) de bloco. Além deste formato, a linguagem dispõe ainda de comandos de estrutura condicional, onde são permitidas escolhas de acordo com condições específicas (comando *if then else*, e comando *case of*). Os blocos de repetição implementados na linguagem são de três tipos:

- com interrupção no início; onde uma condição é testada antes de qualquer comando interno ao corpo do laço do comando a ser executado (*while do*);
- com interrupção no fim; onde o corpo do laço é executado ao menos uma vez antes de ser interrompido (*repeat until*);

- com interrupção preemptiva; onde o número de interrupções é conhecido antes do início do bloco (*for to/downto do*).

3.1.3 Noções básicas

Após esta descrição geral, iremos apresentar alguns detalhes de construção da linguagem que são importantes na implementação desenvolvida neste trabalho. Vamos comentar rapidamente cada uma das trinta e cinco palavras reservadas. Não serão esgotados todos os assuntos referentes a cada ítem, mas apenas os de maior importância. Sempre que possível será apresentado um diagrama sintático do comando em questão. Maiores detalhes podem ser encontrados no Apêndice A.

Estrutura geral - Um programa Pascal começa com um cabeçalho que dá nome ao programa e faz algumas especificações sobre o seu ambiente. Dados como constantes, tipos, variáveis, sub-rotinas e funções são definidos e declarados quando necessário, sempre na ordem listada abaixo. As ações do programa são dadas por uma sequência de comandos, que podem incluir chamadas de sub-rotinas declaradas ou funções. Os programas Pascal seguem um roteiro básico do tipo:

program	cabeçalho do programa
label	rótulos de declarações goto
const	definições de constantes
type	definições de tipos simples e estruturados
var	declarações de variáveis
procedure	
or function	declarações de sub-programas
begin	comandos do programa
...	
end	

Todo bloco deverá conter delimitadores de início e fim. Este delimitadores são *begin* e *end*.

Cabeçalho do programa - O cabeçalho nomeia o programa e seus parâmetros. Dois parâmetros são definidos em todas as implementações Pascal como dispositivos padrão de entrada e saída, mas outros arquivos (e, potencialmente, outros dispositivos) também podem ser nomeados. Assim, o cabeçalho :

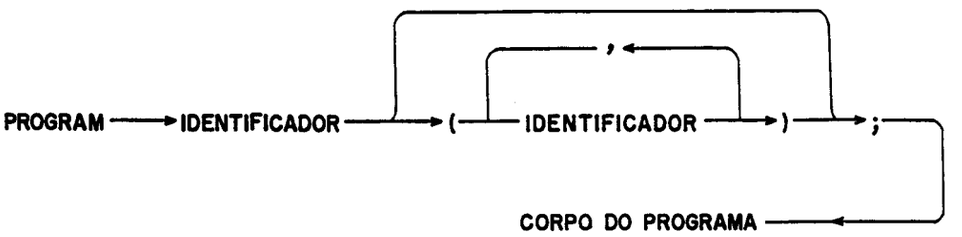
program Início (input, output);

nomeia um programa denominado 'Início', e informa ao processador os dispositivos padrão de entrada e saída que podem ser requeridos durante o tempo de processamento. O primeiro programa-exemplo canônico (que não requer nenhuma entrada) é:

SERVIÇO DE BIBLIOTECA E INFORMAÇÃO - IFQSC
FÍSICA

```
program     Primeiro (output);
begin
            writeln (Olá, mundo)
end.
```

O diagrama sintático para o início de todo programa em Pascal seria o seguinte:



Constantes e variáveis - Definições de constantes dão nomes simbólicos para valores - inteiros, reais, caracteres individuais, cadeias de caracteres, ou novos tipos de valores definidos pelo programador. O valor de uma constante não pode ser alterado durante a execução do programa. Ao lado dos benefícios que elas fornecem como auxíliares mnemônicos, as constantes são usadas frequentemente para documentar valores de implementação de forma clara e definida.

```
const pi          = 3.1416;
      Maxchar     = 255;
      Saudacao    = "Olá, mundo";
      Teste       = true;
```

As declarações de variáveis alocam e nomeiam locações de memória. Todas as variáveis devem ser declaradas, e podem denotar valores de um único tipo. Como será visto nos tipos definidos a seguir, podemos fazer declarações de variáveis tais como:

```
var i, baixo, alto : integer;
```

Também deve ser fornecido o tipo dos dados, que, em Pascal é um atributo de todo valor, variável e função. Assim, quando uma variável ou função é declarada (veja abaixo), o tipo dos valores que ela representa deve ser fornecido; isto permite que verificações de consistência sejam executadas no período de compilação. As *definições de tipo* permitem que o programador renomeie tipos existentes, e permitem uma grande variedade de novos tipos. Os tipos classificam-se em três categorias principais: simples, estruturados e ponteiros. Tipos *simples* são grupos de valores indivisíveis. Quatro tipos simples são pré-definidos em Pascal - real, inteiro, *booleano* e *char*. Tipos simples adicionais podem ser *enumerados* listando novos grupos de valores, ou *sub-limitados*, restringindo um tipo a uma sub-sequência dos valores

de outro tipo previamente definido. Por exemplo:

```
type  Ordem = 0..1000;              { Tipo sub-limitado  }
      Cor   = (azul, verde, lilás)  { Tipo enumerado     }
```

Um tipo estruturado define um conjunto de valores simples, ou um agregado de valores de tipos diferentes. Existem quatro tipos estruturados primitivos : *arrays, records, sets* e *files*. Um *array* é uma tabela de dimensão n, de valores de um único tipo, que é indexada por um ou mais tipos simples. Um *record* é uma união de valores, possivelmente de tipos diferentes, cujos campos podem ser acessados por nome. Um *set* é um grupo de variáveis simples, que dividem o mesmo tipo básico; operadores especializados são definidos por operandos do tipo *set*. Um *file* define uma sequência de valores de qualquer tipo, e é frequentemente associado com algum dispositivo externo. Novos tipos podem ser definidos, em praticamente qualquer combinação. Podem-se nomear tipos que são *arrays* de *records*, *files* destes *arrays*, etc. Por exemplo:

```
Pintura = array [Cor] of Ordem;      [tipo array]
Formula = record
            Maior, Menor: Cor;       [tipo record]
            MaiorPorcent, MenorPorcent: 0..100
            end;
Inventario = file of  Formula;       [tipo file]
```

Finalmente, os tipos *pointer* são úteis para a criação de estruturas de dados ligados - listas, árvores, gráficos, etc. Um *pointer* é uma variável nomeada que referencia uma variável de tipo qualquer alocada dinâmicamente. Na maioria das aplicações, a variável referenciada é definida como um registro que inclui um ponteiro para outro objeto do mesmo tipo que um de seus campos.

Sub-rotinas e funções - São sub-programas nomeados que podem ser chamados durante a execução do programa. Com exceção de seu cabeçalho, uma sub-rotina ou função é exatamente igual a um programa. Ela pode incluir definições e declarações de constantes novas e locais, tipos, variáveis e sub-programas, bem como comandos para serem executados.

Sub-rotinas e funções podem ter parâmetros formais, cujos parâmetros reais (argumentos) podem ser diferentes entre as chamadas das mesmas. Existem várias classes de parâmetros formais. Parâmetros de valor são como variáveis localmente declaradas, mas elas são inicializadas por valores passados durante a chamada. Parâmetros variáveis renomeiam variáveis globais, permitindo que elas sejam acessadas ou atribuídas para dentro do sub-programa.

Uma chamada de função aparece dentro de um programa como parte de uma

expressão. Ela calcula e representa um valor, e pode ter qualquer tipo simples. Muitas funções padrão são pré-definidas em todas as implementações do PASCAL. Um exemplo de uma função declarada pelo usuário é:

```
function MedidasValidas (Comprimento, Largura: real):boolean;
        {Retorna verdade se seus parâmetros são ambos valores positivos.}
        begin
        MedidasValidas:= (Comprimento > 0.0) and (Largura >0.0)
        end;
```

Uma chamada de sub-rotina aparece como um comando no programa. Uma amostra de declaração de sub-rotina que usa tanto parâmetros de valor como parâmetros variáveis é:

```
procedure CalcArea  (Comprimento, Largura : real; var Area: real);
{Calcula uma área, com os dados de comprimento e a largura.}
        begin
                Area:= Comprimento * Largura
        end;
```

Entrada e Saída - Em Pascal, normalmente o teclado e o terminal de vídeo são ajustados como dispositivos de entrada e saída padrão. Ambos têm estrutura de arquivos de texto, o que significa que eles processam caracteres e permitem uma estrutura de linha.

O Pascal se apoia em quatro sub-rotinas pré-definidas para a maior parte dos programas de entrada e saída. Os valores de quaisquer dos tipos podem ser retirados (com exceção dos booleanos) ou lidos e atribuídos a variáveis. Para a saída existem duas sub-rotinas, uma para um *buffer* de saída (*write*), outra para a produção de linhas de saída de forma distinta (*writeln*). Existem também duas sub-rotinas de entrada, uma (*read*) que obtém uma entrada de arquivo, e outra que possibilita a entrada linha a linha (*readln*). Por exemplo:

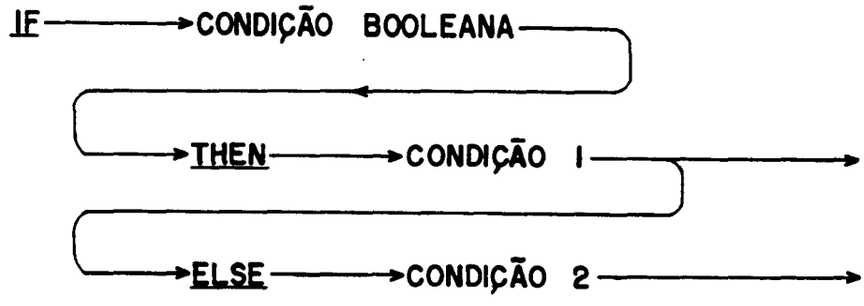
writeln (Entre com um número){Imprime: Entre com um número.}

readln (Número){Lê o valor de Número a partir da entrada padrão.}

Comandos - São as unidades básicas de ação em um programa Pascal. Um comando simples pode conter uma associação ou chamadas de rotinas. Este comando é a forma com que o Pascal implementa uma sequência de comandos sequenciais, onde cada comando deve ser executado sem interrupções ou *loops*.

Outra característica implementada são os comandos onde devem ser feitas escolhas. Neste caso, existe um único caminho a ser seguido. Um exemplo de comando estruturado condicional que está implementado em Pascal é o comando *if*, que pode incluir dois caminhos

diferentes nas seções:



Um cuidado a ser observado, no caso do comando *if*, refere-se à ambiguidade que pode ocorrer no caso de haver *ifs* aninhados. Em termos formais, isto significa que apenas a BNF normal não é suficiente, uma vez que na linguagem Pascal o *else* se refere sempre ao último *if* declarado, como no exemplo abaixo:

```
if Sonolento
  then if Bravo
       then escreva - Sonolento e bravo
       else escreva - Sonolento mas não bravo
  else escreva - Sem sono, mas quem sabe se bravo?
```

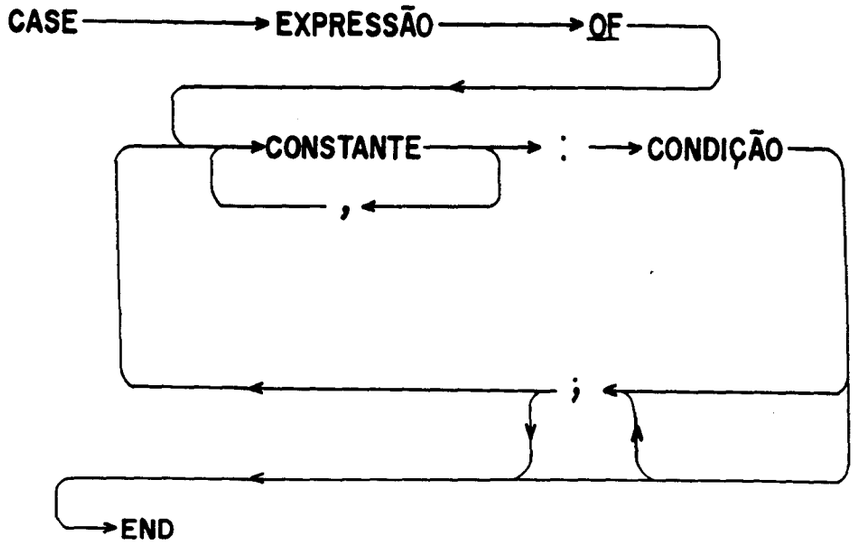
No caso de desejar-se associar o *else* ao primeiro *if*, será necessário o tratamento em forma de blocos bem determinados, usando os comandos *begin* e *end*, como no exemplo abaixo:

```
if Sonolento
  then begin
         if Bravo
           then escreva - Sonolento e bravo
         end
  else escreva - Sem sono, mas quem sabe se bravo ?
```

Quando existe um conjunto de condições que devem ser testadas, exigindo o uso de um conjunto de *ifs* aninhados, pode haver dificuldade de leitura e entedimento da estrutura; por isso o Pascal implementa uma estrutura que permite uma melhor legibilidade. Neste caso uma declaração *case* pode oferecer múltiplas seções:

```
case   Expressão   of
  valor1 : comando1;
    .
  valorN : comandoN;
end;
```

O diagrama sintático fornecido a seguir auxilia a visualização:



Outra estrutura de comandos importante é a que permite a repetição de comandos de forma condicional. Como a maioria das linguagens, o Pascal permite três formas de comandos repetitivos. A primeira é uma forma pre-emptiva, onde é conhecido o número de vezes que o bloco interno do laço será repetido. Esta forma é muito utilizada quando são usados arranjos de vetores e matrizes. Uma definição de comando pré-emptivo é o laço *for* como segue:

for i:= inicio *to* ou *downto* final
  do comando

Outra forma bastante usada é a que executa uma série de comandos enquanto uma determinada condição booleana for verdadeira. Se a condição for inicialmente falsa, o laço deste comando nunca será executado. O comando que implementa tal situação é mostrado abaixo:

while condição booleana
  do comando

A terceira forma implementada de interações condicionais é o comando *repeat*. A diferença entre este comando e o comando *while*, é que este irá executar, ao menos uma vez, o bloco interno ao laço, ou seja, irá se repetir até que a condição booleana seja falsa. Como o teste de parada é feito no final do comando, isto garantirá pelo menos uma execução do corpo do bloco. Estudos [Mizr90] apontam que estes laços, semelhantes ao comando *do while* da linguagem C, são necessários em apenas 5% dos casos de implementação de

comandos repetitivos. Sua forma geral é:

repeat
  declaração
until condição booleana

Demais palavras reservadas - Dentre as demais palavras reservadas do Pascal Padrão, destacam-se também os operadores *booleanos*, *and*, *or* e *not*, respectivamente conjuntivo, disjuntivo e de negação. Estas palavras são importantes para uma linguagem como o Pascal, que implementa tipos booleanos primitivos.

A palavra *goto* é um comando de desvio que irá transferir o controle do programa para um determinado rótulo, indicado pela palavra *label*. Por apresentar características que provocam a desestruturação do código, a linguagem Pascal desencoraja o uso deste comando.

A palavra *in* pode ter dois significados. O primeiro quando está associada a esquerda e constitui um operando booleano, que retorna valores de falso ou verdadeiro, no caso da existência ou não do valor testado em um conjunto de elementos ordinais. O segundo caso refere-se à associatividade a direita, e é utilizado em construções canônicas do tipo *set of*.

Quaisquer dos quatro tipos estruturados básicos do Pascal podem ser designados como sendo ***packed*** **(comprimidos)**, o que instrui o processador a economizar espaço de armazenamento para aquele tipo. Definir um tipo como **packed** frequentemente aumentará o tempo requerido para acessos de, ou operações com, variáveis daquele tipo. A operação de comprimir é transparente para o usuário, mas o programador deve lembrar que existem alguns detalhes a serem considerados, tais como o fato de que as sub-rotinas de transferência requeridas, *pack* e *unpack*, devem ser usadas apenas em conjunção com tipos *array* comprimidos.

O *token nil* é usado para inicializar variáveis ponteiros, onde o ponteiro-*nil* não se refere a nenhum endereço específico, sendo seu valor inexistente. Este raciocínio é aplicado na implementação de estruturas dinâmicas de alocação de pilhas e filas.

Finalmente, as palavras *div* e *mod* referem-se respectivamente a procedimentos matemáticos de retorno de parte inteira e resto de uma divisão inteira.

3.2 - Subconjunto escolhido - o compilador PARIS

O compilador desenvolvido neste trabalho foi denominado PARIS (PAscal para RISc) por usar um sub-conjunto do Pascal Padrão, descrito nos ítens anteriores deste capítulo. Buscou-se explorar ao máximo as características de uma linguagem Pascal Padrão, considerando-se que a otimização de código é fundamental para a arquitetura RISC. Este

fato, traduzido para o conceito de linguagem de programação significa otimização de código gerado global e localmente. Este aspecto será desenvolvido detalhadamente no próximo capítulo, que além de considerar as otimizações comuns, considera também as otimizações de características RISC.

O compilador PARIS possui trinta palavras reservadas, listadas a seguir:

AND	DOWNTO	IF	PROCEDURE	TO
ARRAY	ELSE	IN	PROGRAM	TYPE
BEGIN	END	NIL	RECORD	UNTIL
CASE	FILE	NOT	REPEAT	VAR
CONST	FOR	OF	SET	WITH
DO	FUNCTION	OR	THEN	WHILE

Estas palavras formam um conjunto básico e essencial, com o qual podemos explorar as características da linguagem padrão. Não foram implementadas palavras como **label** e **goto** pelo fato de seu uso ser altamente desencorajado pela linguagem. Estes comandos não são muito utilizados, por desestruturarem o código produzido pelo programador, e só permanecem devido a uma necessidade histórica, uma vez que usuários de linguagens tais como o Basic e o Fortran sentem sua falta ao passarem a trabalhar com Pascal. A palavra reservada *packed*, usada juntamente com vários tipos de dados foi implementada em Pascal Padrão para dar a possibilidade de compactação da informação. A linguagem PARIS não considerou este tipo de recurso. Os operadores *div* e *mod*, que correspondem respectivamente a divisão inteira e módulo também não foram considerados neste estágio, principalmente pelo fato de não acrescentarem nada de novo dentro do trabalho, apesar de serem de implementação simples.

Além das trinta palavras reservadas, fazem parte da linguagem PARIS os tipos de dados: *real, char, integer, boolean*. Como pode ser observado, não foi considerado nesta versão o tipo *text*. A linguagem permite ainda alguns tipos estruturados simples, tais como *record, array set* e *file*. Como exemplo prático da implementação de algumas funções internas, fazem parte do conjunto algumas funções básicas, tais como abs(x), cos(x), trunc(x), sin(x), arqtan(x), round(x), sqrt(x), log(x), log10(x), exp(x), pow(x). Não foi considerada nenhuma função de manipulação de arquivos tais como reset(f), ou put(f) e, no que se refere aos procedimentos, foi considerado um conjunto de dois procedimentos básicos *read*, e *write* para entrada e saída de dados.

Foram considerados os operações normais de operadores, tais como os operadores relacionais, operadores lógicos, e operadores algébricos, com a mesma precedência da linguagem Pascal Padrão, conforme relacionado a seguir:

not
* / and
+ - or
= <> < <= > >= in

Foi implementado também o operador unário negativo e positivo. Lembramos que isto é importante para linguagens que implementam recursos matemáticos, pois este operador difere da operação de subtração, tendo um nível conceitual diferente. A implementação desta ordem de precedência pode ser observada no arquivo PARIS.Y, apresentado no Apêndice C. Detalhes menores, porém importantes, não devem ser esquecidos, como é o caso da sensibilidade da linguagem quanto a letras maiúsculas e minúsculas. A linguagem Pascal não é sensível a este nível, considerando letras maiúsculas e minúsculas como iguais; o mesmo foi considerado na linguagem PARIS.

Finalmente, podemos dizer que a estrutura geral de cada comando foi a mais simples possível dentro das funções básicas da linguagem, e seguiram a estrutura gramatical do Pascal Padrão. Nada além dos recursos do Pascal Padrão foi considerado, e nem todos os recursos disponíveis foram implementados, ou por não acrescentarem nada de importante ao trabalho ou por dificuldades de implementação no atual estágio do projetos RISC-LIE. Entre estas dificuldades citamos, por exemplo, as manipulações de arquivos, que envolvem um tratamento com o ambiente (incluindo o próprio sistema operacional e carregadores) que ainda não foi desenvolvido, ou o caso dos tipos empacotados (*packed*).

Capítulo 4

Otimização e geração de código

Neste capítulo serão apresentados os algorítmos de otimização e geração de código, e também os métodos de otimização citados na literatura [Ahou77]. Serão consideradas as várias otimizações locais, dependentes da máquina, que se mostraram eficientes na melhoria da qualidade do código de saída. Serão assinaladas, ao longo do capítulo, particularidades quanto às otimizações adicionais exigidas pela arquitetura RISC-LIE.

4.1 - Otimização de código

4.1.1 - Conceitos básicos de otimização de código

As técnicas de otimização de código são geralmente aplicadas depois da análise sintática, usualmente antes e durante a geração de código. Essas técnicas consistem em detectar padrões no programa e substituí-los por construções equivalentes, porém mais eficientes. Estes padrões podem ser locais ou globais, e a estratégia de substituição pode ser dependente ou independente da máquina. Aqui serão feitas referências aos dois níveis de otimização, local e global.

Segundo Aho [Ahou77], o programa-fonte como entrada para um compilador é uma especificação de uma ação. O programa-objeto, a saída do compilador, é considerado como sendo outra especificação da mesma ação. Para cada programa-fonte, existem infinitos programas-objeto que implementam a mesma ação, ou seja, produzem a mesma saída para uma mesma entrada. Alguns desses programas-objeto podem ser melhores do que outros com relação a critérios tais como tamanho ou velocidade. O termo <u>otimização de código</u> refere-se às técnicas que um compilador pode empregar em uma tentativa para produzir um programa-objeto melhorado, para um dado programa-fonte. A qualidade de um programa-objeto pode ser medida por seu tamanho ou tempo de execução. Para programas grandes, o tempo de execução é particularmente importante. Para computadores de pequeno porte, o tamanho do código objeto pode ser tão importante quanto o tempo. Deve-se observar com cuidado o termo <u>otimização</u>, devido à dificuldade de se obter um compilador que produza o melhor programa-objeto possível, para qualquer programa-fonte, a um custo razoável. Assim,

um termo mais exato para otimização de código seria melhoria de código ("*code improvement*").

Deve-se porém, destacar que a mais importante fonte de melhoramento do tempo de execução de um programa frequentemente encontra-se na otimização do algorítmo, estando além do alcance do compilador. Por exemplo, no caso em que for necessária uma classificação simples e ordenada de elementos de uma lista, um algorítmo que poderia ser utilizado é *bubblesort*. Este algorítmo, apesar de ampla aceitação e simplicidade, não é otimizado e pode ser substituído por outro algorítmo muito mais eficiente e otimizado, tal como o *quicksort*. Neste caso, a otimização possível é escolher o algorítmo mais otimizado *(quicksort)*, e isto mostra a importância de se implementar o algorítmo adequado a nível de programa-fonte. Nenhuma otimização de código pode recuperar um algorítmo ineficiente.

Existem algumas questões importantes a serem consideradas no que se refere ao benefício de uma dada otimização e seu custo de implementação. Em algumas situações é desnecessário considerar uma otimização mais elaborada, sendo que uma tradução rápida e direta do programa-fonte é suficiente. Entretanto, na maioria dos casos, um programa não será executado indefinidamente sem ser modificado e recompilado. Assim, é importante dispor-se de um compilador com mecanismo de otimização, que faça apenas tentativas consagradas para melhorar o código produzido, sem com isso tomar muito tempo na compilação. O tempo de execução que esperamos economizar sobre o número de execuções esperado do programa-objeto otimizado deve exceder o tempo gasto pelo compilador fazendo a otimização. Uma opção é fazer com que os compiladores disponham internamente de várias opções alternativas, consumindo quantidades de tempo variáveis e permitindo ao usuário decidir quanto tempo ele deseja gastar otimizando seu programa.

A utilização de técnicas de otimização pode aumentar a velocidade de um programa-objeto por um fator de dois ou três em relação à velocidade de um programa-objeto produzido por um compilador sem este recurso [Ahou77]. Este fator é importante para programas executados com muita frequência. Segundo [Ahou77], os três principais critérios a serem considerados na otimização são:

1- A otimização captura a maior parte do potencial de melhoria com uma quantidade de esforço razoável do projetista do compilador ou do próprio compilador?

2- A otimização preserva o significado do programa-fonte, sem incorreções?

3- A otimização, em média, reduz o tempo ou espaço tomado pelo programa-objeto?

Interligada à otimização de código está a geração de código, um assunto que será abordado no próximo ítem. Não faz sentido fazer um bom trabalho de otimização de código sem um bom trabalho de geração de código. Entre as principais fontes de otimização intimamente relacionada à geração de código, encontra-se a utilização eficiente dos registradores e conjuntos de instruções da máquina.

Quanto às técnicas de otimização, uma classe importante refere-se ao manuseio de laços internos. Observou-se que a maior parte do tempo de execução é gasto em uma parte relativamente pequena do programa, ou seja, que 90% do tempo é gasto em 10% do código ("*rule 90-10*"). Então, a parte mais percorrida de um programa, os laços internos, são um alvo óbvio para a otimização. Otimizações de laços típicas são a remoção de computações invariantes de laço e a eliminação de variáveis de indução.

Uma fonte adicional de otimização é a identificação de sub-expressões comuns. Uma expressão única, como a sentença A[I+1] := B[I+1] é mais fácil de se compreender do que um par de sentenças como J:=I+1 e A[J]:=B[J]. É tarefa do compilador identificar a expressão comum I+1, fazendo com que o código objeto da primeira sentença pareça ter vindo da segunda. Outra fonte importante é a substituição de valores para nomes cujos valores são constantes (*constant folding*) em tempo de execução por computações em tempo de compilação.

A seguir, serão discutidas em maiores detalhes as técnicas de otimização citadas anteriomente.

4.1.2 - Otimização de laços

Os programas gastam a maior parte de seu tempo executando laços, que portanto, tornam-se fontes promissoras para o aumento da velocidade de um programa, sendo função de um compilador otimizador produzir códigos eficientes para laços. O processo de otimização de laços envolve desde técnicas para a detecção de sua existência até a implementação de suas possíveis otimizações. Uma vez determinado o trecho do código que

compõe um laço, o primeiro passo é quebrar esse trecho em blocos básicos. Considera-se um bloco básico como sendo uma sequência de comandos consecutivos, que são executados sem interrupção (*halt*) ou sem a possibilidade da ocorrência de desvios (*branch*) - exceto no final do bloco básico.

Um algorítmo útil que produz uma sequência de blocos básicos de fácil implementação, é aquele que considera como entrada uma sequência de comandos com três endereços. A saída deste algorítmo será a lista de blocos básicos, com cada comando de três endereços em um bloco. A primeira etapa do algorítmo é a determinação de componentes denominados líderes. O conjunto de líderes (***leaders***), a partir dos quais pode ser feita a determinação de um bloco básico, é especificado pelas seguintes regras: o primeiro comando é um líder; qualquer comando que é o alvo de um *goto* condicional ou incondicional é um líder; qualquer comando imediatamente seguinte a um *goto* condicional é um líder. Para cada líder deve ser construído o bloco básico correspondente. Cada bloco é constituído por um líder e por todos os comandos posteriores, sem incluir o próximo líder ou o final do programa. A etapa seguinte irá delinear os blocos básicos e suas relações com os blocos sucessores por um gráfico direcionado denominado *flow graph*[1]. Os nós do *flow graph* são os blocos básicos. Um nó é referenciado como inicial quando o líder é o primeiro comando do programa. Existe um direcionador do primeiro bloco para o seguinte, e assim sucessivamente.

Com esses conceitos, pode-se definir um laço como um conjunto de nós fortemente conectado. Ou seja, de qualquer nó situado dentro do *laço* para qualquer outro nó, existe um caminho de comprimento um ou mais, localizado internamente ao laço. Além disso, o conjunto de nós tem uma única entrada: um nó dentro do laço. Com isso, a única maneira de alcançar um nó dentro de um laço, a partir de um nó localizado fora do laço, será primeiro passando pela entrada deste laço onde se encontra o nó.

O tempo de execução de um programa pode ser melhorado se for diminuído o tamanho de seus laços, especialmente de um laço interno. Assim, mesmo que for aumentada a quantidade de código fora dos laços (admitindo-se que o laço em questão é executado, em média, pelo menos uma vez), é possível conseguir-se uma otimização reescrevendo-se este código. As modificações deste tipo são denominadas movimentação de código (***code***

[1]Uma tradução poderia sugerir o termo *gráfico de fluxo*, porém, para evitar confusões com outros termos já convencionados, optou-se pela não tradução deste termo, mantendo-se assim sua forma original.

motion). Na movimentação de código, uma ação que fornece sempre o mesmo resultado, independente do número de vezes que o *laço* é executado (ação denominada ***loop-invariant***), é colocada antes do laço.

Outra otimização importante, que pode ser aplicada ao *flow graph*, diminuindo o número total de instruções e aumentando a velocidade do laço, é o tratamento das variáveis de indução. Como exemplo cita-se uma variável I, cujo propósito é contar de 1 até 20 em um laço, e outra variável T_1, cujo propósito é ser o passo através de arranjos. O passo considerado é de quatro bytes de cada vez, ou seja $T_1 := 4 * I$. Os valores de I e T_1 são interdependentes, isto é, na atribuição, I assume os valores 1,2,...,20 cada vez que passa pelo início do laço, e T_1 assume os valores 4, 8,..., 80. Este comportamento das variáveis I e T_1, as caracteriza como variáveis de indução. Quando existirem duas ou mais variáveis de indução em um laço, e houver a oportunidade da eliminação de todas exceto uma, é aplicado o processo de eliminação de variáveis de indução. Neste exemplo, o passo da multiplicação $T_1 := 4 * I$ pode ser substituído por uma adição $T_1 := T_1 + 4$. Esta substituição aumentará a velocidade do código objeto se a adição tomar menos tempo do que a multiplicação, como é o caso em diversas máquinas. A substituição de uma operação dispendiosa por uma mais barata é denominada redução de força (***reduction in strength***). Certas instruções de máquina são consideravelmente mais baratas do que outras e, frequentemente, podem ser usadas como casos especiais de operadores mais caros. Por exemplo, o cálculo do quadrado de X é sempre mais barato de se implementar como X*X do que como uma chamada para uma rotina de exponenciação.

4.1.3 - A Representação de blocos básicos (DAG)

Uma estrutura útil para analisar automaticamente os blocos básicos é um gráfico acíclico direcionado[2] (***directed acyclic graph*** -DAG)[Ahou77]. Um DAG fornece uma visualização de como o valor executado por cada bloco é usado nos comandos subsequentes do bloco. A construção de um DAG a partir de comandos de três endereços é uma maneira de determinar sub-expressões comuns dentro de um bloco. É importante não confundir os **DAGs** com os *flow graphs*. A representação de cada nó de um *flow graph* pode ser feita por

[2]O DAG pode ser visualizado, de maneira aproximada, como uma estrutura em árvore, de forma hierárquica, com nós pais e filhos. Um nó que não tem filhos é denominado folha. Os nós que tem pais e filhos são denominados nós interiores. Finalmente, o nó que dá origem a árvore é denominado raiz.

um DAG, uma vez que há um bloco básico para cada nó do *flow graph*.

Para construir um DAG, toma-se um comando do bloco de cada vez. Em um comando expresso por A := B + C, olha-se para os nós que representam os valores correntes de B e C. Estes seriam folhas, ou nós interiores do DAG se B e/ou C tivessem sido usados por comandos anteriores do bloco. Cria-se então um nó rotulado + com dois filhos: o da esquerda é o nó para B e o da direita é o nó para C. Rotula-se este nó como A. Contudo, se ainda existir um nó denotando o mesmo valor de A + C, não é criado um nó; ao invés disto dá-se a este nó existente o rótulo adicional de A. Dois detalhes devem ser mencionados: primeiro, se A rotulou anteriormente algum outro nó, remove-se aquele rótulo, uma vez que o valor corrente de A é o nó que acabou de ser criado. Segundo, para uma atribuição tal como A := B, não é criado um novo nó; ao invés disto, acrescenta-se o rótulo A ao nó para o valor corrente de B. Na Figura 4.1 é mostrada a representação deste DAG.

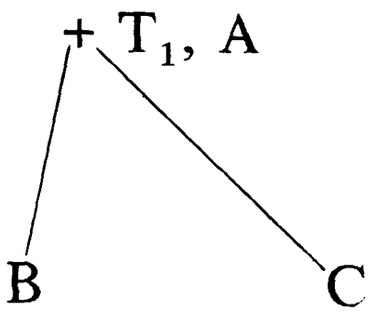


Figura 4.1 Representação de um D.A.G. simples

Um algorítmo para produção de um DAG tem como entrada um bloco básico e necessita das seguintes informações:

1. Um rótulo para cada nó. Para folhas é um identificador (são permitidas constantes) e para nós interiores, um operador símbolo.

2. Para cada nó, uma lista (possivelmente vazia) de identificadores atrelados (*attached identifiers*). Não são permitidas constantes.

Será obtido como saída um gráfico com as seguintes informações:

I. As folhas são rotuladas por identificadores únicos, ou nomes de variáveis ou constantes. As folhas representam valores iniciais de nomes, e devem ser subscritas com 0 para evitar confusão com rótulos que denotem valores correntes de nomes.

II. Nós interiores são rotulados por um operador símbolo.

III. Opcionalmente, são dados aos nós um conjunto extra de identificadores para rótulos. A intenção é de que os nós interiores representem valores computados, e identificadores rotulando um nó sejam considerados como tendo aquele valor.

Este método pode considerar ainda o uso de identidades algébricas. Um algorítmo conhecido como "algorítmo do número de valor" é útil na implementação de certas otimizações baseadas nas leis algébricas. Por exemplo, caso queira-se criar um nó com o filho esquerdo m e o filho direito n, com operador *, deve ser verificado se esse nó já existe. Como na maioria das máquinas a multiplicação é comutativa, deve-se procurar por um nó que tenha um operador *, filho esquerdo n e filho direito m. Este método, apesar de útil, pode ser restrito, uma vez que um programador dificilmente escreve A*B uma vez e B*A de forma alternada. Uma otimização similar pode ser obtida constatando-se que os operadores relacionais <=, >=, <, >, =, e <> podem ser implementados subtraindo-se os argumentos e efetuando-se um teste deste resultado. Somente precisa ser gerado um nó do DAG para, por exemplo, A-B e A <= B.

Existem diversas informações úteis que podem ser obtidas quando se está executando o algorítmo de geração de DAGs. Em primeiro lugar, observa-se que são detectadas automaticamente as sub-expressões comuns. Segundo, pode-se determinar quais os identificadores que tiveram seus valores usados no bloco. Terceiro, pode-se determinar quais comandos calculam valores que poderiam ser usados fora do bloco. Outro uso importante onde o DAG poderia ser aplicado é na reconstrução de uma lista simplificada de quádruplos, tirando vantagem de sub-expressões comuns e não efetuando atribuições do tipo A := B, a menos que seja absolutamente necessário. Apesar de todas as vantagens que os diagramas DAG possam apresentar, deve-se destacar que, no atual nível de implementação deste trabalho, não foi considerada esta forma de otimização.

4.1.4 - Análise de fluxo de dados global

Algumas otimizações só podem ser efetuadas quando são verificadas informações ao longo de todo o programa. Por exemplo, se uma variável A tem valor 3 toda vez que o controle alcança um determinado ponto p, então pode-se substituir 3 para cada uso de A em p. Saber que o valor de A é 3 em p pode exigir a verificação de todo o programa. A análise de fluxo global é uma técnica que pode ser usada para coletar estas informações. Uma destas formas, conhecidas como encadeamento-ud (*use-definition chaining*), descrita de modo geral, responde a questão a respeito do ponto de um programa em que o uso do identificador poderia ter seu valor definido. Para poder entender o encadeamento-ud, é necessária a definição de cada um dos termos da frase anterior. Assim, por uso do identificador A, entende-se a ocorrência do identificador como um operando. Por definição de identificador, referimo-nos a uma atribuição para o identificador ou a leitura de um valor para o mesmo. O caso mais difícil de identificação da definição de um identificador ocorre quando este é o nome de um arranjo. Por ponto de um programa, entende-se a posição, antes ou depois de um comando de linguagem intermediária. Assim, o controle atinge um ponto quando um comando está para ser executado, e o ponto seguinte quando este comando acabou de ser executado. A determinação de pontos em locais errados pode ser fatal para encadeamentos-ud. É, entretanto, uma situação comum de ocorrer em otimização de código, onde raramente as informações são exatas. Tomando-se, porém, alguns cuidados na elaboração desse algorítmo, a análise pode ser conduzida de tal forma que apresente poucas distorções em relação ao objetivo do programa.

Para determinar as definições de valor que podem alcançar um dado ponto em um programa, atribui-se primeiro um número diferente para cada definição de valor. A etapa seguinte do encadeamento-ud é calcular dois conjuntos para cada bloco básico. O primeiro é o conjunto de definições de valor geradas, as definições de valor dentro do bloco que atingem o seu final. O segundo conjunto necessário é o de definições de valor externas ao bloco, que definem identificadores que também têm definições de valor dentro deste bloco. O próximo passo, e o mais complexo, é computar, para todos os blocos, o conjunto de entrada, que consiste de todas as definições de valor que atingem o ponto antes do primeiro comando do bloco. Uma vez conhecido esse conjunto, pode-se determinar as definições de valor que alcançam um uso qualquer de identificador dentro deste bloco usando as regras

determinadas. Para auxiliar essa computação de entrada pode-se executar uma função de saída para todos os blocos, que obtém o conjunto de definições de valor que alcançam o ponto imediatamente depois do último comando do bloco. Os encadeamentos-ud são computados então a partir de informações sobre o alcance de definições de valor. Como os encadeamentos-ud ocupam muito espaço, é importante para um compilador otimizador compactá-las. Frequentemente, os vários usos do identificador terão encadeamentos-ud similares, ou mesmo idênticos.

Existe uma variedade de situações nas quais as informações obtidas por encadeamentos-ud podem ser aproveitadas. Por exemplo, se existir somente uma definição de nome A que alcança um ponto p, e essa definição for A := 3, então sabe-se que A tem valor 3 naquele ponto, e pode-se substituir 3 por A se existir um uso de A no ponto p. Uma segunda aplicação do encadeamento-ud pode determinar se uma definição particular alcança um determinado ponto, o que pode tornar-se importante para a reconstrução de um bloco básico mais eficiente a partir de um DAG. Um terceiro exemplo de utilização de encadeamentos-ud, não relacionada especificamente à otimização de código, é poder determinar se, para um uso particular de uma variável, é possível que esta seja indefinida naquele ponto. Isso pode ser conseguido com a introdução de definições falsas de todas as variáveis antes do bloco inicial. Se a definição falsa de variável puder alcançar um ponto que usa a variável, o compilador pode imprimir um aviso de que a variável pode estar indefinida.

Além da otimização, a etapa final de geração de código, que é objetivo de todo compilador, requer também uma série de cuidados. Na próxima seção, é feita uma breve revisão dos mais importantes aspectos a serem considerados.

4.2 - Geração de código

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

geração de um bom código.

4.2.1 - Programas objeto

Em geral, a entrada para uma rotina de geração de código é um programa em linguagem intermediária. Esta entrada pode ser uma sequência de quádruplos, uma sequência de triplos, uma árvore, ou uma cadeia em notação Polonesa Reversa. A saída do gerador de código é o programa-objeto. Esta saída pode apresentar-se numa variedade de formas de código: linguagem de máquina absoluto, linguagem de máquina relocável, linguagem montadora (*assembly*), ou então outra linguagem qualquer de programação. No caso particular do compilador desenvolvido neste trabalho, a saída será na linguagem montadora da máquina RISC-LIE.

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

A implementação do compilador **PARIS** envolve um modelo simplificado, onde os recursos são colocados de forma integrada dentro do compilador. Isto porque o projeto RISC-

SERVIÇO DE BIBLIOTECA E INFORMAÇÃO - IFQSC
FÍSICA

LIE, em fase de desenvolvimento, não apresenta todos os recursos necessários para dar suporte a todas fases de construção do código gerado. Por exemplo, podemos citar a inexistência de um *linker* para montar o código produzido com as rotinas pré-estabelecidas da linguagem (transformações de tipos permitidas, acesso a *buffer* de escrita e leitura, etc...). Apesar desta restrição, foi possível implementar todas as rotinas básicas necessárias para a geração de código.

4.2.2 - O ambiente do gerador de códigos

A natureza exata do código objeto é altamente dependente da máquina e do sistema operacional. Entretanto, no caso específico deste trabalho, em que o código objeto é linguagem de máquina RISC-LIE, o código é produzido a partir da forma de cada instrucão de alto nível da linguagem **PARIS**. Nesta linguagem, os símbolos internos correspondentes são gerados por si mesmos e inicializados quando requeridos. Neste caso, a tabela de símbolos deve ser consultada para determinar os nomes correspondentes à cada instrução PARIS. Sendo o código objeto uma linguagem de máquina, não surge nenhum problema com endereços, que deverão ser alocados no futuro pelo carregador (*loader*), a ser implementado em trabalho posterior.

Como discutido no ítem anterior, a entrada do gerador de código é um código intermediário, em uma das formas citadas. Nos algorítmos específicos deste trabalho, o primeiro passo foi gerar o código diretamente, sem formas intermediárias. Posteriormente, procurou-se lidar com quádruplos e com árvores sintáticas como formas intermediárias, que permitem a otimização do código gerado. Para isso, foi admitido que já teria sido feita toda a verificação semântica necessária, ou durante a geração de código intermediário (numa passagem separada), ou antes que cada quádruplo tenha sido examinado pelo gerador de código. Assim, o tipo de dado de cada operador e operando é conhecido, e toda as conversões de tipo necessárias são apresentadas no código intermediário. As áreas de dados e demais áreas também foram determinadas, e esta informação está presente na tabela de símbolos. Pretende-se com isso, dizer que a opção da geração de código (otimizado ou não) pode ser implementada sem prejuízo no conteúdo das sequências geradas, visto que toda informação necessária já está analisada.

Frequentemente, é útil imaginar que o código intermediário foi particionado em blocos

básicos, sempre com o cuidado adicional de observar que as chamadas de sub-rotinas e funções sejam, por si mesmas, isoladas em blocos. Este procedimento é necessário para a otimização de qualquer tipo de código. Mas, uma vez que existem algorítmos comuns que informam como reconhecer o início e fim de um bloco básico quando é lida uma lista de quádruplos, não é indispensável que seja apresentada uma estrutura de dados representando um *flow graph*. Consequentemente os algorítmos desenvolvidos para geração de códigos não exigem a existência de uma fase de otimização no compilador.

4.2.3 - Problemas na geração de códigos

Antes de nos aprofundarmos em detalhes, vamos considerar, a partir de um ponto de vista geral, o que está envolvido na geração de códigos. A entrada para o gerador de código RISC-LIE é uma sequência de comandos, visto que a linguagem foi analisada léxica e sintáticamente, e traduzida em uma linguagem intermediária de razoável baixo nível. A análise semântica necessária também foi feita, e os operadores de conversão de tipos foram tratados sempre que necessário. Erros semânticos óbvios, como por exemplo a tentativa de indexar por número de ponto flutuante, já foram detectados.

O próximo passo, a geração de códigos, pode parecer relativamente simples. Contudo, as dificuldades residem na tentativa de executar a computação representada pelo programa de linguagem intermediária eficientememte, usando as instruções disponíveis da máquina RISC-LIE. Surgem dificuldades para decidir que instruções de máquina gerar, em que ordem as computações devem ser feitas e que registradores utilizar. Estes problemas, citados na literatura [Ahou77], ficam mais evidentes ainda em uma máquina RISC, conforme relatado a seguir.

Sobre as instruções que devem ser geradas, pode ser observado no Apêndice C, que dispomos apenas de 30 instruções para implementar todos os comandos da linguagem **PARIS**. A decisão a respeito de qual sequência de código de máquina é melhor para uma dada construção requer aprofundamento extensivo sobre o contexto no qual a construção aparece. É de se supor que exista uma grande variedade de construções que, ainda permitindo as especificações originais do programa-fonte, gere códigos objetos com diferentes níveis de otimização. Com isso quer-se dizer que, de acordo com a melhor ou pior construção das rotinas básicas da geração de código, o compilador **PARIS** pode gerar código mais

otimizado ou não. Como não existe nenhuma ferramenta para atestar o uso ótimo das instruções, o método usado foi o de tentativas, para as diferentes construções possíveis. Portanto, a eficiência do compilador **PARIS**, segundo este ponto de vista, fica limitada à melhor escolha das construções determinadas para serem geradas. Um outro aspecto a ser considerado refere-se à ordem na qual cálculos devem ser executados, como será discutido no Capítulo 5.

Finalmente, decidir a atribuição ótima de registradores para variáveis é difícil, mesmo com a grande quantidades de registradores disponíveis na arquitetura RISC-LIE. Na verdade, aqui o problema torna-se ainda mais complicado, devido à existência de janelas de registradores, que são gerenciadas pela própria arquitetura. Esta avaliação só poderá ser feita após a construção de todo o ambiente operacional, que inclui o Sistema Operacional e o *linker*. Poderá então ser avaliada, através de diversos testes que usem os registradores de todas as formas previsíveis, a alocação destes. Neste sentido, poderiam ser sugeridos testes que envolvam, entre outros recursos, recursividade, referências a ponteiros, e manipulação de cálculo, de forma a exigirem um máximo do número de registradores disponíveis.

4.2.4 - A arquitetura RISC-LIE

Uma boa geração de código requer um conhecimento ótimo da máquina. A sequência de instruções na máquina RISC-LIE segue a forma usual de Von Neumann, exceto pelo esquema de *pipeline* de instrução. A arquitetura está construída de modo a executar as instruções simultâneamente, em *pipeline* de três instruções. Para cada instrução executada no *pipeline* é gasto um tempo igual. Este tipo de processamento favorece o "desvio atrasado" (*delayed branch)*, que, como será discutido, torna-se importante para compiladores RISC otimizados. Os aspectos referentes a *pipeline* foram discutidos em detalhes no Capítulo 1 deste trabalho.

Uma grande diferença entre as máquinas tradicionais e uma máquina de abordagem RISC é seu conjunto reduzido de instruções. Com um conjunto de apenas 30 instruções (ver Apêndice C), a máquina RISC-LIE pretende construir todas as operações necessárias para a realização das tarefas exigidas pelo processamento da informação. Esta tarefa, atrativa pelo fato de proporcionar significativa economia de tempo e memória, representa um ônus especial para o compilador, que deverá construir eficientemente as operações desejadas.

Entre os conceitos mais importantes da máquina RISC-LIE encontra-se a sua organização orientada a registradores. Este fato é importante se consideramos que um computador gasta 70% do seu tempo realizando acessos a operandos, que são armazenados físicamente em memória ou em registradores. Quando o acesso é feito através de registradores, torna-se muito mais rápido e eficiente. Os compiladores devem explorar então o fato de se dispor de um grande número de registradores. Esta operação refere-se tanto a registradores locais como globais. Com o objetivo de manter o máximo possível de operandos em registradores, a arquitetura RISC-LIE implementa um banco de registradores de formato fixo, entrelaçado. Este banco é formado por janelas de registradores adjacentes, organizado de forma circular, possuindo registradores comuns entre si. O número de janelas implementado foi 6, sendo que cada janela é formada por 10 registradores globais, 10 registradores locais e 12 registradores compartilhados, dos quais 6 registradores são para entrada e 6 para a saída de argumentos, como mostra a Figura 1.3.

Pelo fato das instruções envolvendo somente operandos de registradores serem mais curtas e mais rápidas do que aquelas envolvendo operandos de memória, a utilização eficiente de registradores é particularmente importante na geração de um bom código. Devido a esta característica ser altamente encorajada pela filosofia RISC, serão apresentadas algumas estratégias para decidir quais variáveis de um programa devem residir em quais registradores (problema frequentemente denominado como "alocação de registradores").

Um procedimento para alocação e atribuição de registradores é atribuir tipos específicos de valores em um programa-objeto para determinados registradores. Por exemplo, uma decisão pode ser tomada para atribuir blocos de sub-rotinas para um grupo de registradores, endereços de base para outros, cálculos aritméticos para outros, e assim por diante. Este procedimento tem a vantagem de simplificar o projeto de um compilador. A desvantagem é que, aplicado muito estritamente, ele usa registradores ineficientemente: certos registradores podem prosseguir sem uso através de porções substanciais do código, enquanto carregamentos e armazenamentos desnecessários são gerados. Porém, esta é uma tarefa bastante razoável quando tratamos de arquitetura RISC, que dispõe de um grande número de registradores à sua disposição.

Um procedimento que pode ser usado é aquele que tem como objetivo, fazer uso dos registradores para armazenar valores durante a duração de um único bloco básico. Para

economizar armazenamentos e correspondentes carregamentos, deve-se cuidar para atribuir registradores a variáveis frequentemente usadas, e manter esses registradores consistentes além dos limites de um bloco básico, de forma global. Já que os programas gastam a maior parte do seu tempo em laços internos, um procedimento natural na atribuição de registradores globais é tentar manter um nome frequentemente usado em um registrador fixado através de um laço. Como é preciso conhecer detalhes sobre os laços, deve-se admitir que a atribuição de registradores globais é executada depois que um diagrama de blocos foi construído para cada procedimento. Pode-se então utilizar a estratégia de gerenciamento de registradores globais. Neste caso, os registradores podem ser acessados a qualquer instante (ver Figura 1.3), com a hipótese de que um número fixo de registradores serão atribuídos para armazenar os nomes mais usados em cada laço interno. Os nomes selecionados podem diferir entre os diferentes laços. Os registradores ainda não usados podem armazenar valores locais a um bloco. O procedimento tem o incoveniente de que nenhum número dado de registradores será, universalmente, o número correto para tornar disponíveis para alocação de registradores globais, ainda que os estudos de [Kate84] demonstrem que um número considerado ótimo foi usado na arquitetura RISC. Mesmo para outras arquiteturas, este ainda é o método mais simples de ser implementado. É importante observar que, por meio da declaração "*register*", em linguagem C, o programador pode escolher os nomes a serem mantidos nos registradores para a duração de um procedimento. Foi observado que este recurso aumenta a velocidade de muitos programas por um fator de 2 [Ahou77].

Tendo atribuído registradores e gerado códigos para os laços mais internos, pode-se aplicar a mesma idéia para laços progressivamente maiores. Se um laço mais externo L1 contém um laço interno L2, os registradores alocados para nomes em L2 não precisam ser registradores alocados no escopo de L1 que não inclui L2. Contudo, se for alocado um nome A para um registrador no laço L1 mas não em L2, deve-se armazenar A no momento da entrada em L2, e tornar a carregá-lo no momento em que deixa-se L2 e entra-se em uma parte do bloco de L1 que não inclui L2. Similarmente, se A for alocada para um registrador em L2 mas não em L1, deve-se carregar A na entrada para L2 e armazenar A na saída de L2.

4.2.5 - Um gerador de códigos simples

Aqui será apresentado um breve resumo das técnicas de geração de código que

foram aplicadas no desenvolvimento deste trabalho. Primeiramente foi utilizada uma estratégia direta para gerar códigos de máquina a partir de quádruplos. Em seguida, com uma sequência de quádruplos montada, é gerado código para cada quádruplo, um por vez, sempre lembrando que os operandos do quádruplo estão correntemente nos registradores, e se possível, tirando vantagem deste fato. Os resultados calculados podem ser deixados em registradores pelo máximo tempo possível (fato altamente favorecido pela grande quantidade de registradores), e armazenados somente na raras situações em que os registradores serão necessários para outros cálculos. Não é necessário desalocar esses registradores quando ocorre uma chamada de procedimento, uma vez que quando isso ocorre, uma nova janela de registradores é usada com parâmetros comuns. Esta condição, que em compiladores comuns implica que tudo deve ser armazenado exatamente antes do final de um bloco básico, não é mais necessária.

A implementação da geração de código pode ser efetuada utilizando diferentes sequências de instruções equivalentes. A escolha adequada da sequência de instruções pode resultar na otimização do código a ser gerado. Pode-se produzir códigos melhores para um comando A := B + C se for gerada a instrução simples com custos menores. Por exemplo, em um código RISC-LIE, a instrução 'ADD r0,r1,r2' tem um custo baixo (por exemplo, custo = 1) se o resultado A for deixado em um registrador r2. Esta sequência é possível somente se o registrador r1 contiver B, r2 contiver C, e B e C não existirem depois do comando. Se r1 contém B mas C está em uma locação da memória (que denominaremos C), pode-se gerar a sequência 'ADD r0,#256,r21 e ADD r0,r21,r2' com um custo maior (por exemplo, custo = 2). Existem muitos casos a serem considerados, dependendo de onde B e C estão localizados correntemente, e se o valor corrente de B é utilizado subsequentemente. Devem ser considerados os casos em que um deles (B ou C) ou ambos, é uma constante. O número de casos que devem ser observados após esta consideração aumenta se admitirmos que o operador + é comutativo. Percebe-se então que a geração de códigos envolve o exame de um grande número de casos, e a escolha de quais casos devem prevalecer depende do contexto dentro do qual o quádruplo é visto.

Os compiladores RISC devem ter um tratamento especial no que se refere à busca da próxima instrução. Este fato está ligado ao *delayed-branch*, em que a transferência de controle de uma instrução, digamos I_2 do *pipeline*, para a instrução I_3 só ocorre após a execução da instrução I_2. Quer ocorra ou não um desvio, a instrução I_2 será executada.

Assim, uma instrução imediatamente após um *jump/cal/return* será executada antes da transferência de controle e o compilador coloca um NOP (não operação) neste lugar. O compilador, posteriormente, tentará mover para este local uma instrução, otimizando assim este NOP. Segundo [Kate84], os compiladores otimizados conseguem eliminar entre 40 e 90 % dos NOPs. Além disso, uma tarefa de compiladores tradicionais que também deve ser efetuada pelos compiladores RISC é o conhecimento da próxima instrução a ser usada. Neste sentido, o conhecimento da instrução refere-se à tomada de decisões sobre a alocação de registradores para calcular os próximos usos de cada nome em um quádruplo.

Tendo calculado as informações de próximo uso, pode-se executar uma passagem direta através do bloco, selecionando o código objeto. Para executar a alocação de registradores deve-se manter um "descritor" do registrador, que conserve a relação do que está correntemente em cada registrador. Consulta-se este descritor sempre que for necessário um novo registrador. Para cada nome no bloco deve ser mantido um descritor de endereço, que preserve a relação das localizações em um local onde o valor corrente do nome possa ser encontrado em tempo de execução. A localização deve ser um registrador, e esta informação pode ser armazenada na tabela de símbolos, sendo utilizada para determinar o método de acesso a um nome. Pode-se então delinear um algorítmo simples de geração de código. Tem-se uma sequência de quádruplos constituindo um bloco básico. Para cada quádruplo A:=B op C executa-se as seguintes ações:

1 - Chama-se uma função para determinar a localização L onde deveria ser executado o cálculo B op C . L poderá ser um registrador, ou poderá ser uma localização da memória, como será visto no Capítulo 5.

2 - Consulta-se o descritor de endereço para B a fim de determinar uma das n localizações correntes disponíveis de B. Se o valor de B não está em L, gera-se uma instrução para colocar uma cópia de B em L.

3 - Gera-se a instrução correspondente a localização corrente de C. Atualiza-se o descritor de endereço de A para indicar que A está na localização L. Sendo L um registrador, atualizamos seu descritor para indicar que, na hora da execução, ele contém o valor de A.

4 - Se os valores de B e/ou C não têm usos próximos, eles não estão presentes na saída do bloco, e estão nos registradores. Altera-se o descritor do registrador para indicar

que, depois da execução de A:= B op C, aqueles registradores não mais contêm B e/ou C.

4.2.6 - Geração de código a partir de DAG's

Além da geração de código simples, outra forma que pode ser aplicada, sendo bastante atraente para se efetuar a geração de código para um bloco básico, é aquela que efetua a geração de código a partir de uma representação DAG (secção 4.1.3). A vantagem de se fazer isto é que, a partir de um DAG, pode-se rearranjar a ordem da sequência de cálculos finais mais facilmente do que quando começamos a partir de uma sequência linear de quádruplos. O ponto central para a discussão é o caso onde o DAG é uma árvore. Para este caso, pode-se gerar um código onde há a possibilidade de se provar que este seja ótimo, sob critérios tais como comprimento do programa ou menor tempo utilizado. Este algorítmo para geração de código ótimo a partir de uma árvore é útil também quando o código intermediário é uma árvore sintática. O algorítmo, basicamente, reordena a sequência de quádruplos, buscando uma melhoria na ordem de execução de cálculos. Nem sempre é possível conseguir esta reordenação, e muitas vezes são usadas ordenações heurísticas para os DAG. Na seleção de uma ordem para os nós de um DAG, deve ser verificado se a ordem preserva as relações limites do DAG, visto que tais limites podem representar ou a relação operador-operando ou implicar restrições devido a possíveis interações entre atribuições de arranjos ou ponteiros.

Para uma máquina RISC, pode-se ter um algorítmo simples para determinar a ordem ótima na qual avaliar a sequência de quádruplos quando a representação DAG dos quádruplos é uma árvore. A ordem ótima, aqui, significa a ordem que fornece a sequência de instruções mais curta, sobre todas as sequências de instrução que avaliam a árvore. Este algorítmo foi utilizado em um grande número de compiladores não RISC, tais como ALGOL W, BLISS/11, e o compilador PDP-11 C, mas podem ser aplicados a arquiteturas como a RISC-LIE.

O algorítmo trabalha na representação árvore da sequência de quádruplos. Ele poderia também trabalhar na árvore sintática, se ela fosse a forma de código intermediária. O algorítmo tem 2 partes: a primeira rotula cada nó da árvore, de baixo para cima (*bottom-up*), com um inteiro que denota o menor número de registradores requeridos para avaliar a árvore com nenhum armazenamento de resultados intermediários. A segunda parte é uma

árvore transversal, cuja ordem é governada pelos rótulos dos nós calculados. O código de saída é gerado durante a árvore transversal.

É importante lembrar que quando exixtem sub-expressões comuns em uma sequência de quádruplos, o DAG correspondente não será mais uma árvore. As sub-expressões comuns corresponderão aos nós com mais de um pai, chamados nós compartilhados. Neste caso não é mais possível aplicar diretamente o algorítmo de rotulação. Pode-se, entretanto, particionar o DAG em um conjunto de árvores, encontrando para cada raiz e/ou nó compartilhado n, a sub-árvore máxima com n como raiz, que não inclui nenhum outro nó compartilhado, exceto como folhas.

Uma vez que o DAG é particionado em árvores desta maneira, pode-se ordenar a avaliação das árvores e usar um algorítmo similar ao discutido anteriomente para gerar código para cada árvore. A ordem das árvores deve ser tal que os valores compartilhados que são folhas de uma árvore devem estar disponíveis quando a árvore é avaliada. As quantidades compartilhadas podem ser calculadas e armazenadas na memória (ou mantidas nos registradores, sempre em grande número na arquitetura RISC-LIE). Mesmo que este procedimento não gere necessariamente um código ótimo, ele será, frequentemente, satisfatório.

Capítulo 5

Resultados obtidos

Neste capítulo será descrito detalhadamente o procedimento utilizado na construção do compilador *PARIS*. Serão também discutidas as principais dificuldades encontradas, já que as soluções adotadas poderão ser úteis para trabalhos futuros. Será apresentada uma análise dos resultados e do desempenho do compilador desenvolvido, e também será mostrada a sua compatibilidade com a linguagem interpretada pelo simulador RISC-LIE construído por Valente [Vale91].

5.1 - Considerações iniciais

O estudo inicial das regras gramaticais do Pascal Padrão revelou a existência de cento e sessenta regras [Coop83] da BNF estendida (ítem A.1 do Apêndice A). Estas regras, ao serem convertidas para o BNF do YACC foram reduzidas a cento e trinta, já que muitas delas servem apenas para manter a consistência lógica, sem influir na gramática propriamente dita. A seguir, na montagem do conjunto de regras, observou-se que a produção de palavras reservadas, ou reconhecimento de dígitos para formar constantes, deveriam ser efetuadas dentro da análise léxica, por se tratarem da construção de tipos de dados que compõe a formação de *tokens*. No exemplo a seguir, observa-se que a regra para o reconhecimento de palavras reservadas (*word_symbol*) deverá ser efetuada pelo analisador léxico, que construirá cada *token* e o enviará para o analisador sintático, eliminando assim uma regra de produção.

---

word_symbol = **PROGRAM | LABEL | CONST | TYPE | PROCEDURE | FUNCTION | VAR | BEGIN | END | DIV | MOD | AND | NOT | OR | IN | ARRAY | FILE | RECORD | SET | CASE | OF | FOR | TO | DOWNTO | DO | IF | THEN | ELSE | REPEAT | UNTIL | WHILE | WITH | GOTO | NIL**

Após uma análise detalhada do BNF [Coop83], concluiu-se que outras regras sintáticas também poderiam ser absorvidas pela analise léxica, sendo que um total de vinte e uma regras foram incorporadas pelo analisador léxico.

Um fato curioso observado na BNF original, é a seguinte regra de produção:

```
simple_type : ordinal_type
            | identifier
            ;
```

Esta regra jamais é referenciada em lugar algum da gramática. Ela apenas define um tipo simples, que nunca será usado por nenhuma outra regra. Por esse motivo, ela foi eliminada da gramática do compilador **PARIS**.

Além disto, determinados aspectos da linguagem foram propositalmente abandonados. Conforme discutido no ítem 3.2, não foram implementados os comandos *goto* e seu associado *label*, e o tipo de variável compactado (*packed*), o que eliminou mais cinco regras (três para o comando *goto*, uma para o *label*, e uma para *packed*). A interpretação de conteúdos semânticos, tais como a definição de tipos, fez com que outras dezenove regras fossem incorporadas pelo analisador semântico, restando para a BNF do compilador **PARIS** a construcao de oitenta e seis regras de produção, que foram definidas e apresentadas ao **YACC**.

5.2 - Etapas da implementação

A implementação do compilador **PARIS** foi efetuada segundo as etapas mostradas na Figura 2.3. Para o acompanhar e entender melhor o funcionamento do compilador RISC-LIE, foi adotado um esquema idêntico ao apresentado no ítem 2.2.7. Usaremos um programa-exemplo, denominado ***shell***[1] [Good77][Schl89], para demonstrar o funcionamento de cada módulo do compilador. Sempre que possível, será citado um exemplo da saída de cada módulo, demonstrando sua atuação. Cada exemplo será relacionado nos sub-ítens denominados '**produção do módulo**', e as listagens completas podem ser encontradas no Apêndice D. O programa **shell1.pas** é mostrado a seguir:

[1]Proposto por D. L. Shell, baseado em *diminishing increments*, para a ordenação de vetores.

```
PROGRAM shell (output, input);
    CONST max = 11;
        lim = 5;
    VAR vetor          : ARRAY [1..max] OF INTEGER;
      item            : ARRAY [1..lim] OF INTEGER;
      count, x        : INTEGER;
      i, j, k, s, w   : INTEGER;
    BEGIN
      vetor[1] := 9; vetor[2] := 5; vetor[3] := 3;
      vetor[4] := 3; vetor[5] := 1;
      item[1] := 9; item[2] := 5; item[3] := 3; item[4] := 9;
      item[5] := 5; item[6] := 3; item[7] := 3; item[8] := 1;
      item[9] := 3; item[9] := 1; item[10]:= 3; item[11]:= 1;
    FOR w := 1  TO w < 5   DO BEGIN
          k := vetor[w];
          s := -k;
          FOR  i := k TO  i < count  DO BEGIN
               x := item[i];
               j := i - k;
               IF s = 1  THEN BEGIN
                    s := -k;
                    s := s + 1;
                    item[s] := x;
                  END;
               WHILE ((x < item[j]) AND (i >= 0 ) AND (j <= count)) DO
                  BEGIN
                    item[j+k] := item[j];
                    j := k-1;
               END;
               item[j+k] := x;
          END;
      END;
  END.
```

A etapa da construção do compilador **PARIS** em que foi utilizado o **YACC** foi inicialmente implementada no arquivo **paris.y**, que compõe a entrada do **YACC**. Esta rotina incorpora uma secção de declarações em linguagem C, uma secção de declarações **YACC**, a definição de regras gramaticais, e outras rotinas adicionais em C. Entre estas rotinas adicionais encontra-se a rotina de ativação da leitura do programa pascal a ser processado pelo compilador **PARIS**, que é efetuada pela função (**main()**) mostrada a seguir. Esta função iniciará as chamadas das rotinas de análise sintática e o ambiente do compilador **PARIS**.

```
/*
  Funcao Principal do Programa PARIS
*/
void main(int argc, char *argv[]) {
int indx=0;
printf("
printf("  Compilador  RISC-LIE  PARIS  Versao  1.0    IFQSC-USP/Sao  Carlos  \n");
printf("                                                                        \n");
printf("                                                                        \n");
 if (argc < 2) {
 printf("                                                                      \n");
 printf("  Use   prompt> paris   [-] [s|l|o|m] <Arquivo_fonte.pas>             \n");
 printf("                                                                      \n");
 printf("  s : => rastreia o analisador sintático                              \n");
 printf("  l : => rastreia o analisador léxico                                 \n");
 printf("  o : => ativa o otimizador                                           \n");
 printf("  m : => rastreia o analisador semântico                              \n");
 printf("                                                                      "\n)
  exit(1);
 fin=0;

  for (indx=1; indx<argc; indx++) {
      if (argv[indx][0] == '-') {
          int i=1;
          while (argv[indx][i] != 0) {
              if (argv[indx][i] == 's') SINT = 1;
              if (argv[indx][i] == 'l') LEXI = 1;
              if (argv[indx][i] == 'o') OPTI = 1;
              if (argv[indx][i] == 'm') SEMA = 1;
              i++;
              }
          }
```

```
      else {
        if (fin != 0) {

printf("                                                    \n");
printf("  Use   prompt> paris [-] [s|l|o|m] <Arquivo_fonte.pas>  \n");
printf("                                                    \n");
printf("  s : => rastreia o analisador sintático            \n");
printf("  l : => rastreia o analisador léxico               \n");
printf("  o : => ativa o otimizador                         \n");
printf("  m : => rastreia o analisador semântico            \n");
printf("                                                    \n");
        exit (1);
      }
      else {
          fin=fopen (argv[indx], "r");
      }
    }
  }
  init_tabsimb ();
  init_arquivo();
  yyparse();
}
```

A seguir, as etapas da implementação serão discutidas em detalhes.

5.2.1 - O analisador léxico

Nesta etapa construiu-se um analisador léxico, de concepção mais simples possível, porém eficiente para realizar as seguintes operações básicas:

- reconhecimento e construção de palavras reservadas e de identificadores,
- reconhecimento de dígitos e construção de tipos inteiros e reais,
- reconhecimento de símbolos especiais, incluindo símbolos compostos (:=, >=, <= entre outros),
- identificação e eliminação de espaços em branco, tabulação e comentários (separados por símbolos "{" e "}").}

Para a implementação de todos os elementos do compilador, foram feitas as declarações dos símbolos terminais e não terminais, bem como a declaração dos tipos de

operadores adotados na linguagem, que serão usados na seção de declarações YACC. Esta secção é reproduzida a seguir.

```
/*
    Declaracao de Tokens para o Analisador Lexico
*/
%start program
%token <sym> TYPE PROCEDURE FUNCTION BEGIN END IN
%token <sym> ARRAY RECORD SET CASE OF FOR TO DOWNTO
%token <sym> DO IF THEN ELSE REPEAT UNTIL WHILe WITH NIL
%token <sym> C_INTEGER CONST C_REAL IDENTIFICADOR VAR C_CHAR
%token <sym> PROGRAM VARIABLE OPERATOR NADA
%token <sym> INTEGER REAL BOOLeAN CHAR RG
%type <sym> Idf
%type <sym> constant
%type <sym> id_list_arquivo
%type <sym> array_type
%type <sym> case_statement
%type <sym> compound_statement
%type <sym> consta
%type <sym> else_part
%type <sym> for_statement
%type <sym> function_heading
%type <sym> if_statement
%type <sym> formal_parameter_list
%type <sym> pfd
%type <sym> pf_part
%type <sym> assigment_statement
%type <sym> pd_part
%type <sym> program_heading
%type <sym> record_type
%type <sym> set_type
%type <sym> repeat_statement
%type <sym> type_definition
%type <sym> unpacked_conformant_array_schema
```

```
%type <sym> while_statement
%right IG
%right AT
%left OR
%left AND
%left GT GE LT Le EQ NE
%left '+' '-'   /* associatividade esquerda, mesma precedencia */
%left '*' '/'   /* associatividade esquerda, media precedencia */
%left UNARY NOT /* operador unario, alta precedencia          */
%expected 9
```

Os elementos que serão reconhecidos como *tokens* da linguagem são aqueles definidos como palavras reservadas no ítem 5.1. Os operadores implementados são os definidos pela BNF do Apêndice A (ítem A.1). Na declaração dos símbolos não terminais, deve ser observado quais devem ser os tipos a serem declarados para utilização na pilha de símbolos. Nem todos os não terminais devem ser declarados, pois o excesso de declarações pode ultrapassar a capacidade de armazenamento da pilha de símbolos. Entretanto, algumas regras exigem o retorno de algum valor que será utilizado por outra regra; neste caso, os tipos devem ser declarados, já que, caso contrário, o YACC retornará uma mensagem (*must return a value since LHS was a type*) exigindo a declaração. Para evitar quaisquer conflitos deste tipo, foi usado apenas um tipo de dado em toda a pilha de símbolos (o tipo *sym*, que é definido em **PARIS.H**[2]). Observa-se também o *token* ***program***, que dá início ao reconhecimento da gramática, indicado pelo símbolo YACC denominado ***%start***. O símbolo ***%expected*** 9 refere-se ao gerenciamento de ambiguidades e será discutido em maiores detalhes no ítem 5.2.2.1.

Depois da escolha e determinação de todos os tipos que devem ser declarados, a secção foi incorporada ao arquivo "paris.y", na secção de declarações YACC. Neste arquivo

[2] sym é definido como sendo do tipo estruturado st_Symbol:

```
typedef struct st_Symbol {
char nome[MAX_TAM_SIMBOLO+1] /* Nome do simbolo */
tipo_dado tipo;     /* Tipo do simbolo */
valor_dado valor;   /* Valor a ser armazenado na palavra */
long endereco;      /* Endereco da palavra risc-lle */
} st_Symbol;
```

existe ainda uma rotina usada para invocar os tokens descritos, denominada yylex(). Esta rotina, que representa um símbolo interno do YACC, foi colocada na secção de código adicional em linguagem C, e irá implementar o analisador léxico.

Dentro do analisador léxico, foi incorporada uma atividade não puramente léxica: uma ação semântica de início da marcação de escopo de variáveis[3] dentro de sub-rotinas. Isto foi feito devido à facilidade de ativar-se a rotina de reconhecimento de um escopo no exato momento em que o analisador léxico determina os *tokens program, procedure* e *function.* Desta forma, o início da atividade fica bem determinado, como pode ser observado na rotina parcialmente reproduzida abaixo:

```
if ((lstrcmp(mbuf, "PROGRAM"  )));  marca_escopo(); return PROGRAM;  }
if ((lstrcmp(mbuf, "PROCEDURE")));  marca_escopo(); return PROCEDURE;}
if ((lstrcmp(mbuf, "FUNCTION" )));  marca_escopo(); return FUNCTION;  }
```

Além desta atividade, também foi usada uma marcação de início de atividades tais como *while, for* e outros comandos, para determinar quando deveria ser efetuada a execução da pilha de operadores para, por exemplo, o teste condicional de um comando *while.* Esta atividade está ligada a geração de código propriamente dita.

Deve-se observar que o analisador aceita palavras reservadas escritas em letras maiúsculas ou minúsculas, mas a exemplo da linguagem C, diferencia as variáveis. Assim, um identificador qualquer, quando definido com letras maiúsculas deve ser sempre usado desta forma, e uma variável 'VARI', é diferente de outra 'vari', que por sua vez é diferente de 'Vari', ou 'vAri'. Esta característica permite ao programador a especificação de um grande número de variáveis. Deve-se observar que esta versão, por questão de simplicidade, não dispõe de um mecanismo de reconhecimento de constantes expressas em notação de potência de 10 (por exemplo, 1.0E+5).

A impressão das linhas lidas é uma tarefa efetuada pelo analisador léxico. Para

[3] O escopo de variáveis refere-se ao intervalo de existência em que a variáveis declaradas podem ser referenciadas

SERVIÇO DE BIBLIOTECA E INFORMAÇÃO - IFQSC
FÍSICA

realizá-la, foi necessário transformar as funções getc e ungetc[4] em outras rotinas pegac e unpegac. A rotina pegac lê o caracter de entrada do arquivo e armazena o mesmo em uma matriz unidimensional denominada linha, que é impressa sempre que for lido o caracter de fim de linha (\n), ou quando o limite de 80 caracteres for ultrapassado. A rotina unpegac é usada para devolver um caracter lido a mais, e neste caso decrementa a matriz linha e devolve esse caracter. Estas rotinas são reproduzidas abaixo:

```
/*
    Funcoes de Impressao de Linhas pelo Analisador Lexico
*/
static int ikkk=0;
char linha[120];

int unpegac (int c, FILE *arq) {
   ungetc (c, arq);
   --ikkk;
   return c;
   }

int pegac(FILE *arq) {
  int c;
  c=linha[ikkk++]=getc(arq);
  if (c=='\n' || ikkk>79) {
        linha[ikkk]=0;
        conta_linha++;
        if (LEXI) printf("\n$ %d>  %s", conta_linha, linha);
        ikkk=0;
        }
  return c;
  }
```

[4]Estas funções pertencem à biblioteca da linguagem C padrão, e podem ser incluídas usando-se o arquivo stdio.h.

5.2.1.1 - Produção do módulo

A saída deste módulo pode ser observada pela emissão do texto de entrada lido, que pode ser obtida digitando-se a diretiva de entrada -l junto ao nome do programa de entrada. Para o exemplo do programa **shell.pas**, a saída seria a impressão do arquivo de entrada que corresponde ao próprio arquivo **shell.pas**, e que pode ser vista no ítem D.1.

5.2.2 - O analisador sintático

Uma vez definidas as oitenta e seis regras a serem interpretadas pelo compilador **PARIS**, a etapa seguinte foi a construção das regras de produção usadas no arquivo **paris.y**, como entrada formal para o **YACC**. Entre os recursos oferecidos pelo **YACC**, podemos citar a opção da emissão de diversos tipos de saídas impressas, destacando-se aquela que emite o controle de ambiguidades da gramática construída como entrada. Este aspecto, de extrema importância, será dicutido a seguir.

5.2.2.1 - O controle de ambiguidades

O **YACC** fornece uma ferramenta de apoio que permite um bom controle para a eliminação ou diminuição de conflitos de ambiguidades. Dentro deste aspecto devemos considerar dois tipos de conflitos importantes:

1 - Os conflitos aceitáveis, de forma a tornar a grámatica razoavelmente consistente, usualmente denominados conflitos *Shift/Reduce*;

2- Os conflitos que não podem ser aceitos de maneira alguma, sob pena da gramática tornar-se ambígua, usualmente denominados conflitos *Reduce/Reduce*;

Para um melhor esclarecimento, mostraremos alguns exemplos destes conflitos e de que modo eles podem ser solucionados, conforme sugerido por [Donn88].

Conflitos *Shift/Reduce* : Suponha que esteja sendo feita a análise sintática de uma linguagem que tem comandos *if-then* e comandos *if-then-else*, com um par de regras como:

```
if_cmd   : IF expr THEN cmd              (1ª. regra)
         | IF expr THEN cmd ELSE cmd     (2ª. regra)
         ;
```

Quando o *token* ELSE é lido e torna-se o próximo *token* a ser interpretado

gramaticalmente, os conteúdos da pilha (assumindo que a entrada é válida) são adequados para redução[5] (*reduction*) pela primeira regra. Entretanto, também é válido substituir (*shift*) o *token* ELSE, porque isto levaria a uma eventual redução pela segunda regra. Esta situação, onde tanto um *shift* quanto um *reduction* seriam válidos, estabelece um conflito *shift/reduce*. O YACC é projetado para resolver estes conflitos escolhendo o caminho *shift*, a menos que existam declarações de precedência contrárias especificadas pelo operador. Para compreender a razão desta escolha, vamos compará-la com a outra alternativa possível. Se o analisador escolhe substituir (pelo caminho *shift)* o *ELSE*, conforme o exemplo abaixo:

```
IF x THEN IF y THEN vence; ELSE perde;
```

o resultado é anexar a declaração *else* ao comando *if* mais interno, tornando as duas entradas, de substituição pela primeira ou pela segunda regras equivalentes. Esta estrutura seria interpretada como:

```
IF x
THEN do;
┌─ IF y
└──────┬─ THEN vence;
       └─ ELSE perde; fim;
```

Porém, se o analisador sintático resolve reduzir quando possível, ao invés de substituir (*shift)*, o resultado seria anexar a declaração *else* ao comando *if* mais externo, e seria interpretado como:

```
┌─── IF x
│   ┌─ THEN do;
│   │ IF y
└───┼     THEN vence; fim;
    └─ ELSE perde;
```

O conflito existe porque a gramática escrita é ambígua. A convenção estabelecida é que estas ambiguidades são resolvidas anexando a declaração *else* ao comando *if* mais interno; isto é o que o YACC faz quando escolhe substituir ao invés de reduzir. Esta ambiguidade, em particular, foi primeiramente encontrada nas especificações de Algol 60 [Trem85], e foi denominada ambiguidade *dangling else*[6]. O correto seria escrever uma gramática sem ambiguidades. Entretanto isto é muito difícil de ser feito neste e em outros

[5] A redução é feita quando podemos, através de uma regra de produção, encontrar um símbolo terminal, e não restam elementos não terminais que possam ser substituídos.

[6] *dangling else: else* pendente ou oscilante seria uma tradução aproximada.

casos, razão pela qual elas algumas vezes são mantidas. Para eliminar as mensagens de erro de *Shift/Reduce* foi colocada a instrução **%expected 9**.

Conflitos *Reduce/Reduce*

Um conflito *reduce/reduce* ocorre se existirem duas ou mais regras que se apliquem à mesma sequência de entrada. Isto usualmente indica um erro grave na gramática. Como exemplo, é mostrada abaixo uma tentativa errônea para definir uma sequência de zero ou mais agrupamentos *palavra*, conforme citado por [Donn88].

```
sequencia : /* vazio */        { printf ("sequencia vazia\n");        }
          | palavra           { printf ("palavra simples %s", $1);    }
          | sequencia palavra { printf ("adiciona palavra %s\n", $2); }
          ;
```

Ocorre uma ambiguidade porque existe mais do que um modo de efetuar-se a análise sintática de um *token* **palavra**, isolado em uma sequência. Ela poderia ser reduzida diretamente através da segunda regra. Alternativamente, nada poderia ser reduzido para uma sequência através da primeira regra, e esta poderia ser combinada com a **palavra** usando a terceira regra. O YACC resolve um conflito *reduce/reduce* escolhendo usar a regra que aparece primeiro na gramática, mas este não é o critério mais seguro. Cada conflito *reduce/reduce* deve ser estudado e sempre eliminado. A seguir é apresentada a maneira adequada de definir-se uma **sequência**:

```
sequencia  : /* vazio */    { printf ("sequencia vazia\n");        }
           | sequencia palavra  { printf ("adiciona palavra%s\n", $2); }
           ;
```

Outro exemplo de erro comum, que resulta em conflito *reduce/reduce* é mostrado abaixo:

```
sequencia     : /* vazio */
              | sequencia palavra
              | sequencia redirecionam
              ;
```

```
palavras     : /* vazio */
             | palavras palavra
             ;

redirecionam : /* vazio */
             | redirecionam redireciona
             ;
```

A intenção aqui é definir uma sequência que contém ou os agrupamentos **palavra** ou os agrupamentos **redireciona**. As definições individuais de **sequencia**, *words* e **redirecionam** estão isentas de erro, mas as três juntas compõem uma ambiguidade: a mesma entrada **vazio** pode ser analisada de muitas maneiras. Abaixo são mostradas duas alternativas para corrigir estas regras. A primeira seria fazê-las em um único nível de sequência:

```
sequencia    : /* vazia */
             | sequencia palavra
             | sequencia redireciona
             ;
```

A segunda alternativa seria impedir que um **palavra** ou um **redirecionam**, estejam vazios:

```
sequencia    : /* vazio */
             | sequencia palavras
             | sequencia redirecionam
             ;

palavra      : palavra
             | palavras palavra
             ;

redireciona  : redireciona
             | redirecionam redireciona
             ;
```

Os problemas destes conflitos podem ser resolvidos de duas formas. A primeira seria percorrendo todo o conjunto de regras, analisando cada uma em particular, e depois

integrando cada regra ao conjunto maior que a envolve, identificando quais os conflitos que ocorrem. A segunda forma, seria solicitar ao próprio YACC a emissão de uma lista com todas as regras e os casos onde as reduções ocorrem ou não. A emissão da lista pode ser obtida com o uso da diretiva -v, juntamente com a solicitação do processamento do arquivo paris.y. Este procedimento produzirá, além do arquivo y_tab.c normalmente gerado, também o arquivo y.out. Este último indicará todos os estados, desvios, e reduções que foram encontrados, e pode ser usado na verificação dos locais onde ocorrem os casos de não redução e conflitos *shift/reduce* e *reduce/reduce*. Com isso, cabe ao projetista do compilador apenas buscar a solução para os conflitos, sem ter que se preocupar com a leitura extensiva de todas as regras. Para ilustrar este caso é mostrada seguir uma pequena parte inicial da gramática do compilador **PARIS**:

```
program          : program_heading ';' block '.'
                 ;

program_heading : PROGRAM idf
                 | PROGRAM IDENTIFICADOR '(' id_list_arquivo ')'
                 ;

id_list_arquivo : IDENTIFICADOR
                | IDENTIFICADOR ',' id_list
                ;
```

Observação: a regra para *block* não foi colocada.

Esta gramática gera o seguinte arquivo y.out:

```
state 0
        $accept : _program $end
        PROGRAM  shift 3
        . error
        program  goto 1
        program_heading  goto 2
```

```
state 1
        $accept : program_$end
        $end  accept
        . error
state 2
        program : program_heading_; block .
        ; shift 4
        . error
state 3
        program_heading : PROGRAM_idf
        program_heading : PROGRAM_IDENTIFICADOR ( id_list_arquivo )
        IDENTIFICADOR  shift 6
        . error
        idf  goto 5
state 4
        program : program_heading ;_block .
        nada : _    (3)
        CONST  shift 9
        . reduce 3
        block  goto 7
        cons_def_par  goto 8
        nada  goto 10
state 5
        program_heading : PROGRAM idf_    (103)
        . reduce 103
state 6
        program_heading : PROGRAM IDENTIFICADOR_( id_list_arquivo )
        idf : IDENTIFICADOR_    (197)
        (  shift 11
        . reduce 197
state 7
        program : program_heading ; block_.
        . shift 12
        . error
```

5.2.2.2 - Conflitos na gramática do compilador PARIS

Ao submeter-se a gramática do compilador **PARIS**, no formato obtido após a codificação para a BNF do Pascal Padrão [Coop83], foi constatada a ocorrência de noventa e quatro conflitos *shift/reduce* e trezentos e sete *reduce/reduce*. Além disso, foi relatada a existência de quarenta e seis regras que nunca eram reduzidas (*rules never reduced*). Através da reconstrução das regras originais, foi possível eliminar todos os trezentos e sete conflitos *reduce/reduce*, enquanto os conflitos *shift/reduce* foram reduzidos apenas para nove. Além disso, todas as regras passaram a ser reduzidas. O símbolo YACC ***%expected*** 9 foi usado para eliminar o aviso de ocorrência dos conflitos que não foram eliminados. Dos nove casos de conflitos *shift/reduce*, dois foram observados na estrutura de definição de ***record***, e quatro foram observados respectivamente nas estruturas de definição de ***with***, ***set***, ***if***, e na estrutura de definição de constantes (***const***). Os três casos restantes foram observados em algumas combinações de regras de produçao, que ocorriam em situações específicas.

Visando efetuar uma comparação com um exemplo de aplicação do YACC encontrado na literatura, foi analisado o programa **hoc6** (*hight order calculator*, versão seis) [Kern84]. Este programa adota uma linguagem simplificada, que implementa as quatro operações básicas de uma calculadora de alta ordem (*high order calculator*), a estrutura de escolha (*if then else*), um comando de repetição (*while*), e também as chamadas de sub-rotinas (*function* e *procedure*). A forma como são construídos estes comandos sugere uma pequena semelhança com a linguagem Pascal. Neste programa, o arquivo de entrada para o processamento do YACC apresenta apenas 14 regras de formação, que geram ao final do processamento 26 conflitos *shift/reduce*.

Para a gramática **PARIS**, um caso típico de ocorrência de conflito *shift/reduce* refere-se ao *if*, que foi declarado da seguinte forma:

```
if_statement    : IF expression THEN statement
                | IF expression THEN statement  else_part
                ;

else_part       : ELSE statement
                ;
```

Este caso poderia ser eliminado, se fosse acrescentado um ';' na primeira regra do

SERVIÇO DE BIBLIOTECA E INFORMAÇÃO - IFQSC
FÍSICA

if_statement e do **else_part**, conforme indicado abaixo:

```
if_statement  : IF exprelal  THEN statement ';'
              | IF exprelal  THEN statement  else_part
              ;

else_part  : ELSE statement ';'
           ;
```

Isso, entretanto, obrigaria o programador a colocar um ';' a mais sempre que usasse o caso do *if* simples (sem *else*), ou composto (com o *else*). Assim, a gramática teria dois ';' seguidos, o que pareceria estranho para qualquer programador. Optou-se então por deixar a solução do conflito (*shift/reduce*) para o YACC.

Entre os casos de conflitos que foram eliminados, pode-se citar o observado na estrutura de declaração de variáveis de acesso, inicialmente proposta como mostrado abaixo:

```
var_access          : idf
                  | component_variable
                  | var_access '^'
                  ;

component_variable     : var_access '[' ie  ']'
                     | var_access '.' idf
                     : idf
                     ;

ie                   : expression
                     | expression  ',' ie
                     ;
```

A estrutura das regras de produção, da forma como está construída provoca os seguintes conflitos: um *shift/reduce*, vinte e sete *reduce/reduce*, e uma regra que nunca é reduzida. A solucão adotada foi retirar o *idf* do *component_variable*, pois a substituição de *idf* é permitida pela regra de var_access, sem perda de caminhos percorridos e reconhecimento sintáticos. Com esta ação pode-se eliminar todos os conflitos.

Um exemplo de caso simples, que provoca um conflito *shift/reduce*, e apresenta uma regra sem nenhuma redução é apresentado abaixo:

```
case_list        : case_list_element ';'
                 | case_list_element ';' case_list
                 ;
```

A solução adotada para este caso foi simplesmente remover o *token* ';' da primeira linha da regra, como mostrado abaixo:

```
case_list        : case_list_element
                 | case_list_element ';' case_list
                 ;
```

O exemplo anterior, de solução relativamente simples, pode levar a crer que todos os casos são igualmente simples. Muitas vezes, porém, a simples inclusão de símbolos não terminais pode provocar um efeito em cascata, onde os conflitos se propagam de forma descontrolada. Por exemplo, a colocação do não terminal *function_designator* em *factor*, como mostrado no exemplo abaixo, provoca a ocorrência de quarenta e nove conflitos *shift/reduce* e vinte e um *reduce/reduce* :

```
factor                : var_access
                      | C_REAL
                      | C_INTEGER
                      | function_designator
                      | set_constructor
                      | '(' expression ')'
                      | NOT factor
                      ;

function_designator   : idf
                      | idf ie
                      ;
```

A ocorrência destes conflitos é explicada porque no não terminal *function_designator* existe o não terminal *idf*, que pode ser reconhecido por substituição do não terminal

var_access na regra de *factor*. A remoção de *function_designator* da regra poderia sugerir que a regra está sendo limitada, não substituindo a mesma para o caso de **factor** ser uma referência a chamada de função. Na verdade isso não ocorre, por que a chamada de uma função, a nível sintático, realmente não é nada além da determinação de um simples identificador. A tarefa de reconhecer **factor** como uma chamada de função é uma atividade semântica, e não sintática, como poderia indicar uma análise superficial. Pode-se, então, remover o não terminal *function_designator* da regra de *factor*, sem perda de uma ação do compilador, desde que esta ação seja implementada na semântica da chamada de uma função.

Um caso muito mais complexo poderia ocorrer se, na regra de *factor*, fosse incluído o não terminal *expression*. A inclusão deste não terminal seria desejável por que, desta forma, poderia-se realizar substituição de expressões (por exemplo soma e subtraçao), sem necessitar de uma ordem específica. Considerando uma expressão do tipo

a := a * b / c - 1 ,

pelo fato da precedência da multiplicação ser igual à da divisão, o **YACC**, não saberia escolher qual das operações efetuar primeiro. Esta análise pode ser estendida para operações de soma e subtração, e para os operadores relacionais. Sua inclusão, entretanto, provoca a ocorrência de cento e vinte e sete conflitos *shift/reduce* e setecentos e noventa e nove *reduce/reduce*, além de quatro regras nunca reduzirem. Optou-se então por não incluir o não terminal *expression* na regra, obrigando o programador a explicitar uma operação entre parêntesis sempre que possa haver igualdade de precedência entre dois operadores. Desta forma, a expressão anterior deveria ser implementada como:

a := (a * b)/ c - 1

Na versão atual do compilador, existe a possibilidade de se imprimir ou não a árvore sintática do programa que está sendo processado pelo compilador **PARIS**, com o uso da opção -s, como mostrado no ítem 5.2.2.3. Esta opção apresenta dois aspectos interessantes - pode ser útil para a compreensão do compilador **PARIS** do ponto de vista didático, além de ajudar o programador a detectar rapidamente os erros síntaticos.

É necessário associar a cada variável declarada em uma lista o seu tipo. Isso pode se feito somente ao final da lista, quando foi determinado qual o tipo denotado na lista. Para isto, um conjunto de rotinas adicionais foram implementadas visando o tratamento das listas

de variáveis. Estas rotinas foram inseridas em um arquivo particular denominado lista.c, reproduzido a seguir:

```
/* lista.c */

inicic_lista () {
  topo_lista_var = 0;
  return 0;
  }

reinic_lista () {
  topo_lista_var = 0;
  return 0;
  }

poe_lista_var (st_Symbol *indice) {
  lista_var[topo_lista_var++]=indice;
  return 0;
  }

inic_percorre_lista_var () {
  percorre_lista_var = 0;
  return 0;
  }

st_Symbol *le_lista_var() {
  if (percorre_lista_var < topo_lista_var) {
     return lista_var[percorre_lista_var++];}
    else
     return 0;
  }
```

Com este conjunto de rotinas foi possível implementar quaisquer tratamentos que

envolvessem declarações de objetos em uma lista, dentro da análise sintática (por exemplo, declaração de uma lista de constantes, ou declaração de uma lista de argumentos em passagens de parâmetros de funções ou procedimentos). Seguindo a metodologia adotada no desenvolvimento deste trabalho, as rotinas foram construídas de forma compacta e definidas de forma independente.

5.2.2.3 - Produção do módulo

A saída fornecida pelo compilador para este módulo pode ser visualizada quando de digita a opção -s, que irá gerar na tela, uma listagem que permite que seja montada a árvore sintática correspondente ao arquivo **shell.pas**. Esta listagem pode ser encontrada no ítem D.2.

5.2.3 - O analisador semântico

Completado o analisador léxico, a etapa seguinte consistiu em implementar, para cada uma das oitenta e seis regras de produção, suas correspondentes regras semânticas. Esta etapa envolveu duas atividades paralelas. A primeira foi o desenvolvimento de uma interface em linguagem C, com todas as rotinas semânticas necessárias para realizar o tratamento das operações das ações semânticas. Esta interface foi desenvolvida separadamente e inserida em um arquivo denominado **ROTINAS.C**. Ela implementa, entre outras rotinas, as estruturas de empilhamento e desempilhamento e demais atividades que geram um código intermediário. A segunda etapa foi a colocação de chamadas destas rotinas em pontos estratégicos, junto às regras de produção no arquivo **PARIS.Y**.

5.2.3.1 - Interfaces em linguagem C

O código intermediário gerado pode ser traduzido diretamente para o código RISC-LIE, ou ser submetido ao otimizador global. Este código intemediário foi gerado usando a estrutura da notação polonesa, onde os operandos são armazenados em uma estrutura de pilha e desempilhados quando ocorre a leitura de um operador. O motivo pelo qual estas rotinas foram implementadas em um arquivo separado faz parte da metodologia que norteou este trabalho, com a adoção de uma técnica de desenvolvimento de rotinas com estratégia orientada para modularização. Isto significa que todas as etapas que exigiam a criação de uma rotina foram desenvolvidas com um conjunto de rotinas independentes, que pudessem ser usadas a qualquer instante, por qualquer outra rotina da estrutura do compilador. Deve-se

ressaltar que, apesar de utilizar-se um conceito de desenvolvimento orientado para objetos, não foram utilizadas todas as técnicas que esta metodologia oferece. Como exemplo do trabalho efetuado, apresentamos a seguir as rotinas de manipulação de expressões em notação polonesa, que podem ser encontradas no arquivo **ROTINAS.C**:

```
/*
   Funcoes para Criar a Expressao em Forma Polonesa
*/
int vet_pol[100], pol_percorre, pol_topo;

void init_pol_topo() { /* inicia topo da pilha */
  pol_topo=pol_percorre=0;
  }

void init_pol_percorre() { /* percorre a pilha */
  pol_percorre=0;
  }

void pol_var(st_Symbol *indice) { /* obtem variavel da pilha */
  vet_pol[pol_topo++]=indice_tabela (indice);
  }
void pol_opr(operacao opr) {
  vet_pol[pol_topo++]=(int)opr;
  }
int le_pol () {
  if (pol_percorre<pol_topo)
      return vet_pol[pol_percorre++];
    else
      return -10000;
  }
void pol_print () {
  int i;
  printf ("\n=======Polonesa:\n");
  for (i=0; i<pol_topo; i++) printf ("<%d>", vet_pol[i]);
  }
```

Como pode ser observado, a implementação das funções para criar a expressao em forma polonesa são todas especializadas em pequenas atividades, sendo que o tamanho de cada rotina resumiu-se a poucas linhas. Com isso, foi possível criar uma especialidade para cada rotina, denotada por uma função específica. Esta metodologia foi utilizada ao longo de todo o desenvolvimento das interfaces em linguagem C. Em todo o arquivo **ROTINAS.C**, foram desenvolvidos conjuntos de rotinas, tais como as mostradas acima, para a manipulação da tabela de símbolos, estabelecimento do escopo de variáveis, e outras funções de usos gerais para a interface semântica.

Seguindo a mesma metodologia, foi criado o arquivo **PARIS.H**, onde foram declaradas todas as definições do pré-processamento, as variáveis globais (ou externas), e todos os protótipos das funções usadas por qualquer parte ou módulo do compilador **PARIS**. A definição deste arquivo acrescentou um controle centralizado sobre todas as definições, útil tanto no desenvolvimento do compilador, quanto em possíveis alterações futuras.

5.2.3.2 - Chamadas das rotinas das ações semânticas

Tendo sido definidas as estruturas das ações semânticas para cada conjunto gramatical, a próxima etapa do trabalho foi a de inserir estas rotinas nas localizações corretas em que elas deveriam ser ativadas. Esta etapa foi implementada quando foram criadas as regras semânticas dentro do arquivo **PARIS.Y**. Neste arquivo, ao lado de cada regra de produção, foram colocadas as chamadas, dentro de chaves. Como exemplo, na definição de associação de variáveis (**assigment_statement**) mostrada abaixo, a ação semântica da primeira regra é a de inserir na pilha da expressão polonesa o operador **AT** (atribuição). A segunda regra, além desta operação, deve ainda substituir o *token* **Idf** (identificador, ou seja uma o nome de uma variável) na tabela de símbolos.

```
assigment_statement    : var_access AT expression
                              { pol_opr(op_at); }
                     | Idf  AT expression
                              { pol_opr(op_at);
                                todovalor.c_integer=0;
                                substitui_simb ($1, tipo, todovalor); }
                     ;
```

Nem todas as regras, porém, têm alguma ação semântica, como no caso da definição

de ***expression***, que tem propósito apenas sintático, sem nenhuma ação semântica correspondente:

```
expression : simple_expression
           | exprelal
           ;
```

Completando o conjunto de ações que definem a avaliação de expressões, são relacionadas a seguir o conjunto de derivações sintáticas e semânticas referentes a este aspecto:

```
exprelal  : simple_expression GT simple_expression {
                    pol_opr (op_gt); }
          | simple_expression GE simple_expression {
                    pol_opr (op_ge); }
          | simple_expression LT simple_expression {
                    pol_opr (op_lt); }
          | simple_expression LE simple_expression {
                    pol_opr (op_le); }
          | simple_expression IG simple_expression {
                    pol_opr (op_eq); }
          | simple_expression NE simple_expression {
                    pol_opr (op_ne); }
          ;

mop  : factor '*' factor     { pol_opr (op_mul);  }
     | factor '*' factor mop { pol_opr (op_mul);  }
     | factor '/' factor     { pol_opr (op_div);  }
     | factor '/' factor mop { pol_opr (op_div);  }
     | factor AND factor     { pol_opr (op_and);  }
     | factor AND factor mop { pol_opr (op_and);  }
     ;
term  : factor
      | mop
      ;
```

```
simple_expression : '+' term '+' term { pol_opr (op_add);      }
                  | '+' term '-' term { pol_opr (op_unimenos);
                                        pol_opr (op_dif);      }
                  | '+' term OR term { pol_opr (op_or);        }
                  | '-' term '+' term { pol_opr (op_unimenos);
                                        pol_opr (op_add);      }
                  | '-' term '-' term { pol_opr (op_unimenos);
                                        pol_opr (op_dif);      }
                  | '-' term OR term { pol_opr (op_unimenos);
                                        pol_opr (op_or);       }
                  |     term '+' term { pol_opr(op_add);       }
                  |     term '-' term { pol_opr (op_dif);      }
                  |     term OR term { pol_opr (op_or);        }
                  |     term
                  | '+' term
                  | '-' term         { pol_opr (op_unimenos); }
                  ;


factor        : var_access
              | C_REAL { todovalor.c_real=0.0;
                         substitui_simb ($1, c_real, todovalor);  }
              | C_INTEGER { todovalor.c_integer=0;
                            substitui_simb ($1, c_integer, todovalor); }
              | idf '(' ie ')' { todovalor.c_integer=0;
                                 substitui_simb ($1, tipo, todovalor); }
              | set_constructor
              | '(' expression ')'
              | NOT factor
              ;
```

5.2.3.3 - Produção das ações semânticas

Um fato a ser observado é que, no momento em que uma ação ocorre (por exemplo o ato de empilhar e desempilhar operandos e operadores na pilha da expressão polonesa), já está sendo efetuada a produção de código. Por isso é dito que o analisador sintático está

no limite entre a fase de análise e a fase de síntese de um compilador.

As ações semânticas implementadas produzem então um código global em um arquivo intermediário e temporário, denominado **A.RIS**, criado pela rotina **Init_arquivo()** (reproduzida abaixo). Este arquivo pode ser lido pelo módulo gerador de código (descrito no ítem 5.2.5), ou, se o programador desejar que sejam efetuadas as otimizações tradicionais, pode ser usada a opção -o. Neste caso, o arquivo será lido pelo módulo de otimização de código descrito no ítem 5.2.4).

```
Init_arquivo() {
   Saida = fopen ("a.ris", "w");
   if (Saida == 0) {
       printf ("Erro na abertura do arquivo a.ris\n");
       exit (1);
   }
   return 0;
}
```

5.2.3.4 - Produção do módulo

A saída deste módulo pode ser visualizada de duas maneiras: pela impressão da tabela de símbolos, e pela impressão da pilha da expressão polonesa produzida. Esta saída é obtida utilizando-se a opção -m. Para uma melhor visualização, a opção -l foi também usada, que imprime a saída do analisador léxico. Com isso pode-se ter uma idéia da produção deste módulo, reproduzida no ítem D.3.

5.2.4 - Módulos de otimização

Primeiramente, destacaremos os dois tipos de otimização previstos neste trabalho. O primeiro se refere a otimização tradicional que, como já foi discutido no Capítulo 4, é um módulo fundamental na construção de compiladores, e torna-se ainda mais importante em compiladores de alta perfomance, como os das máquinas RISC. Algumas máquinas que utilizam a filosofia RISC - como por exemplo as que utilizam o processador SPARC (SUN) - consideram otimizações globais efetuadas independentemente da máquina em que se deseja gerar código, obtendo assim uma boa otimização sem considerar aspectos RISC. O outro tipo de otimização refere-se às exigidas pela arquitetura RISC, como por exemplo,

a otimização de instruções NOOP, que são produzidas em determinadas condições[7] quando o compilador executa o programa fonte. Deve-se observar que a existência da opção (-o), que permite o uso da otimização tradicional é interessante quando trabalha-se a nível de desenvolvimento de programas.

O processo que demanda maior esforço dos compiladores para máquinas RISC é o escalonamento de instruções, considerado por isso um fator delicado em otimizações dessas máquinas. A otimização do escalonamento de instruções está relacionada com a boa utilização do *pipeline*, o que assegura um fluxo estável de processamento das instruções. No caso de instuções que provoquem desvio (*branchs*), o *pipeline* é atrasado até que a instrução no endereço do desvio seja buscada (*fetch*), gerando uma bolha[8] neste *pipeline*. As técnicas usualmente utilizadas para a escolha da instrução apropriada após o desvio são complicadas[9]. Neste trabalho foi adotada a solução genérica proposta por Patterson [Patte85] para máquinas RISC. Este autor mostrou ser possível melhorar o desempenho do *pipeline* com um simples rearranjo automático das instruções dentro de um programa, de modo que as instruções de desvio ocorram depois do realmente desejado. Esta solução redefine os saltos dos desvios (*jump*) de modo que eles não tenham efeito até depois da instrução seguinte, técnica denominada *delayed branch* (desvio atrasado). O *delayed branch* faz com que sempre seja selecionada a próxima instrução durante a execução da instrução corrente. O código de linguagem de máquina é adequadamente arranjado de maneira que sejam obtidos os resultados desejados. Com isso, o programador não precisa preocupar-se em rearranjar suas instruções. O *delayed branch* elimina bolhas possivelmente existentes de forma bastante eficiente. No caso específico da arquitetura RISC-LIE, foi feito com que as operações de desvio reescrevam o código de forma a adiar o desvio em três passos (número de estágios do *pipeline*) para que as instruções de desvio pudessem estar prontas na sua vez de serem executadas.

Trabalhos publicados mostram que as soluções mais utilizadas para otimizações em *pipelines* não são triviais. Por exemplo, [Gonz93] cita um mecanismo para reduzir o custo de

---

[7]Um exemplo é quando da chamada (*call*) de sub-rotinas.

[8] Uma bolha (*bubble*) ocorre quando um dos estágios do *pipeline* fica vazio, provocando um baixo aproveitamento deste processador, com consequente redução de sua eficiência.

[9]Existem três possibilidades do tratamento em *hardware*: *multiple streams*, *prefetch branch target*, *branch prediction*

SERVIÇO DE BIBLIOTECA E INFORMAÇÃO - IFQSC
FISICA

desvios em *pipeline*, que é baseado no uso de pré-buscas (*prefetchs*) múltiplas e execução paralela de desvios. Neste trabalho é apresentado um modelo análitico no qual foi feita uma medição de eficiência, que permitiu uma análise do tamanho da memória cache necessária para minimizar o uso do processador.

Usualmente, um sistema RISC nao tem instruções construídas internamente para chamadas de procedimentos. Ao invés disso, é deixado para o projeto dos compiladores a criação de rotinas apropriadas. No caso de uma rotina "folha"[10], o projeto do compilador pode usar uma estrutura de chamada muito mais simplificada, que utiliza registradores para passar parâmetros, ao invés do sistema de pilhas. Com isso a responsabilidade pela economia de registradores será dividida entre o procedimento que faz a chamada e as rotinas que foram chamadas. Isso otimiza o tempo gasto, visto que normalmente só é permitido ao procedimento que faz a chamada economizar todos os registradores. Esta forma de otimização também foi considerada no presente trabalho.

Os módulos de otimização, reunidos no arquivo **OTIMIZA.C**, implementam o que denominamos otimização tradicional. As estruturas são básicas e bastante simples. Para o seu desenvolvimento, foram usadas as estruturas sugeridas por algumas rotinas básicas do compilador **GNU CC**[11]. Este pacote é um *free software*, ou seja, seu código fonte é livre para a inspeção por qualquer usuário. O compilador GNU CC, na versão de 1989, é um compilador de linguagem C completo e está disponível no IFQSC em qualquer terminal conectado ao VAX. Ele gera internamente um código intermediário denominado RTL, que servirá de entrada para um otimizador. O otimizador do C-GNU é composto por diversos arquivos independentes tais como os seguintes: otimizador de laços (*loop otimization*, encontrado no arquivo **loop.c**); otimizador de desvios (**jump.c**); otimizador de registradores globais (**global-a.c**[12]) e de registradores locais (**local-a.c**), entre outros. Neste trabalho foram implementados parcialmente os algorítmos básicos de otimização citados, uma vez que a aplicação de todos os algorítmos seria uma tarefa extensa, que foge aos nossos objetivos.

---

[10] rotinas que não chamam outras rotinas

[11] É importante destacar que foram usados apenas os algoritmos dessas rotinas, e as funções usadas módulo **OTIMIZA.C** foram reescritas.

[12] Os nomes dos arquivos apresentados aqui são os aqueles convertidos para o DOS, e estão truncados para oito caracteres que podem ser aceitos pelo DOS. O nome original, por exemplo do arquivo 'global-a.c' no ambiente do VAX é 'global-alloc.c'.

5.2.4.1 - Produção do módulo

Nenhuma saída pode ser obtida diretamente deste módulo, pois a opção é realizar ou não a otimização, e o arquivo temporário (a.ris) é destruído. A destruição deste arquivo é realizada em tempo de compilação do programa pascal, sendo usada apenas como suporte para a produção de código. No caso da opção -o ter sido escolhida, a otimização poderá ser vista na saída do módulo gerador de código, e comparada com uma saída não otimizada.

5.2.5 - O módulo gerador de código

Este é o módulo final, que vai gerar o código compatível com a arquitetura RISC-LIE. Uma forma, por exemplo, pela qual é ativada a geração de código está relacionada à chamada da rotina **executa-pol()**. Esta chamada é feita quando for identificada uma declaração simples. A seguir é mostrada a declaração sintática e a ação semântica correspondente da chamada descrita:

```
simple_statement : nada
                 | assigment_statement {
                    pol_print();      /* imprime a expressao polonesa final */
                    executa_pol();    /* executa a pilha da polonesa        */
                    init_pol_topo();  /* inicia leitura da polonesa         */ }
                 | procedure_statement
                 ;
```

A função **executa-pol()** por sua vez, lerá a pilha e efetuará o seu desmonte. Ela verifica se o símbolo lido é um operador ou um operando. Os valores colocados na pilha de dados da notação polonesa podem ser maiores que zero (neste caso trata-se de um operando), ou menores que zero (tratando-se neste caso de operadores). Caso a leitura tenha sido de um operando, o valor é retirado desta pilha, e colocado na pilha de execução. Caso a leitura tenha sido um operador, é verificada de qual operação se trata, sendo que o valor **operacao** é transferido para a rotina **aritmet_code(operacao)**. A rotina **aritmet_code(operacao)** efetua qualquer operação aritmética de soma, multiplicação, subtração ou divisão, sendo que a operação é explicitada e transferida como parâmetro através de um valor no momento de sua chamada. Os parâmetros definem qual operação

deve ser transcrita. Uma das primeiras ações da rotina **aritmet_code()** é a execução da função **pop()**, retirando os dois últimos valores da pilha da expressão polonesa, e atribuindo-os aos valores **d1** e **d2**, respectivamente, para cada execução. A rotina que efetua esta chamada, **executa_pol()**, é parcialmente reproduzida a seguir:

```
/*-----------------------------------------------------*
 |              Funcao  executa_pol()                  |
 *-----------------------------------------------------*/
 int operacao;
 st_Symbol S;
 int simb;
 init_pol_percorre();
 Topo_pilhainstr=0;
 while ( (simb=le_pol()) != -10000) {
   if (simb > 0) push(TabSimb[simb]);
        else {
          switch(-simb) {
           case 'add'  : operacao = 1; aritmet_code(operacao); break;
           case 'sub'  : operacao = 2; aritmet_code(operacao); break;
           case 'atr'  : atrib_code(operacao);                 break;
           case 'ge'   : logic_code(operacao);                 break;
                                  .
                                  .
           default : printf("\n  operacao PARIS impossivel"); break;  }
```

A etapa seguinte consistiu na verificação dos tipos de elementos que estão sendo operados (constante inteira ou real, variável inteira ou real). No caso da verificação não ter sucesso, o compilador indicará um erro. De acordo com o tipo verificado, o código a ser gerado será, obviamente, diferente. As rotinas **aritmet_code(operacao)**, e **pop()** podem ser vistas a seguir:

A próxima ação é a chamada da rotina **grava_instr ()**, que abre o arquivo onde será escrito o código objeto (**shell.obj** no presente exemplo), e chama a rotina **print_instr()**. Esta rotina recebe os mesmos parâmetros tranferidos para **grava_instr()**, e imprime os valores desejados. O primeiro parâmetro da rotina **grava_instr()** refere-se ao código de operação RISC-LIE desejado (por exemplo o número um corresponde à **add**, o número dois à **addc**, o número três à **sub**, conforme a tabela de instruções fornecida no Apêndice C). O valor de referência é recebido pela função **print_instr()** que, usando uma estrututra de escolha, imprimirá (na tela ou no arquivo objeto) a instrução RISC-LIE completa. Esta função é reproduzida parcialmente a seguir:

```
/*  ╔═══════════════════════════════════════╗
    ╚═══ Funcao  Para Imprimir o Codigo RISC Gerado ═══╝ ||

void print_instr (instrucao instr) {
    printf ("\n");
    switch (instr.cod) {
        case 1:
          printf ("add     R%d,", instr.opr1);
          print_S2 (instr.cod2, instr.opr2);
          printf (" ,R%d", instr.opr3);
          print_CC (instr.cond);
          break;
        case 2 :
          printf ("addc    R%d,", instr.opr1);
          print_S2 (instr.cod2, instr.opr2);
          printf (" ,R%d", instr.opr3);
          print_CC (instr.cond);
          break;
```

Os seis parâmetros desta rotina têm as seguintes funções: o primeiro refere-se ao número da instrução que se deseja executar. O segundo parâmetro refere-se a ativação (CC) ou não (NN) da opção que ativa o vigésimo-quarto bit da instrução, fazendo assim com que as CCs[13] sejam afetadas, no caso de testes lógicos. O terceiro parâmetro refere-se ao

[13] Ver [Vale91]

número do registrador que será usado, o quarto parâmetro indica se o valor passado trata-se de uma constante ou de uma variável. O quinto parâmetro da função **grava_instr()** refere-se a um código que deverá ser lido e convertido em um endereço calculado e determinado pelo **linker**. Nesta versão, este código corresponde à função que realiza a operação solicitada. As funções disponíveis encontram-se em uma tabela de funções internas, que provisóriamente substitui o mapa de funções. Em trabalhos futuros, o mapa deverá ser construído pelo **linker** e as funções serão ligadas por ele. Qualquer nova função que possa ser escrita ou acrescentada pelo usuário deverá ser inserida nesta tabela. Uma reprodução parcial desta tabela é mostrada a seguir:

```
tabela_fun tabela_func[MAX_FUNCS] =
        { { 'a','d','d','r','r',' ',' ',' ', 0L,  500L,    v_real },
          { 'a','d','d','r','l',' ',' ',' ', 0L,  400L,    v_real },
          { 's','u','b','r','r',' ',' ',' ', 0L, 1500L,    v_real },
          { 's','u','b','r','l',' ',' ',' ', 0L, 1000L,    v_real },
          { 'm','u','l','t','r','r',' ',' ', 0L, 1200L, v_integer },
          { 'm','u','l','t','r','l',' ',' ', 0L,  500L,    v_real },
          { 'm','u','l','t','l','l',' ',' ', 0L,  400L,    v_real },
          { 'd','l','v','r','r',' ',' ',' ', 0L, 2500L,    v_real },
          { 'd','l','v','r','l',' ',' ',' ', 0L, 2500L,    v_real },
          { 'd','l','v','l','l',' ',' ',' ', 0L, 1500L, v_integer },
          { 'w','r','l','t','e',' ',' ',' ', 0L, 5500L,    v_char },
          { 'r','e','a','d',' ',' ',' ',' ', 0L, 5700L,    v_char },
          {                    •
                               •
                                                                  },}
```

Finalmente, o sexto parâmetro refere-se ao registador para onde está sendo deslocado o resultado da instrução, podendo ainda indicar um tipo de operação lógica realizada em saltos.

Observa-se, na função **aritmet_code()**, o uso da rotina **aloca_temp()** que trata da manipulação dos valores temporários de memória. Para completar a manipulação destes espaços de memória são utilizadas também as rotinas **init_temp()**, e **desaloca_temp()**,

reproduzidas a seguir :

```
/* ------------------------------------------------------------
  | Funcoes Para Manipular o Espaco Temporario de Uso de Memoria |
  ------------------------------------------------------------ */
void init_temp () {
  int i;
  for (i=0; i<MAX_TEMP; i++) temp_livre[i]=0;
  }

void aloca_temp (st_Symbol d, tipo_dado tipo) {
  int i;
  for (i=0; i<MAX_TEMP; i++ ) if (temp_livre[i] == 0) break;
  if (i>MAX_TEMP) {
      printf ("\nERRO: espaco de variaveis temporarias exaurido!");
      exit(1);
      }
  temp_livre[i]=1;
  d.nome[0]='%';
  d.nome[1]='0'+ i/10;
  d.nome[2]='%'+ i%10;
  d.nome[3]='0';
  d.tipo = tipo;
  d.endereco = i+1;
  }

void desaloca_temp (st_Symbol d) {
  int i;
  if (d.nome[0] != '%') return;
  i=(d.nome[1]-'0')*10 + d.nome[2]-'0';
  temp_livre[i]=0;
  }
```

Após a execução das operações, o resultado deve ser novamente armazenado na

pilha de execução. Esta operação é efetuada pela rotina push(), reproduzida abaixo:

```
/* ------------------------------------------------------
  |                Função  push()                |
  ------------------------------------------------------ */
void push(st_Symbol d)  /* coloca na pilha de execucao */
{
  if (SEMC) printf("poe pilha");
  if (Topo_pilhainstr>= MAX_PILHAEXECUTA) execerror("stack overflow", (char *) 0);

  pilha_exec[Topo_pilhainstr++]=d;
}
```

A geração de código usando avaliação de expressões que é feita utilizando-se a rotina **executa_pol()**, é, sem dúvida, uma das tarefas mais importantes dentro da geração de código. Mas não se pode esquecer que não é esta a única maneira para produzir código. É importante destacar que nem todos os comandos produzem código de máquina. Por exemplo, o comando *begin*, que é insdispensável em certos momentos para delimitar o início e o término de um bloco, e de extrema importância para uma ação semântica, não gera uma sequência de código de máquina correspondente. Porém, comandos tais como *while* necessitam gerar sequências que correspondem a chamadas de rotinas para gerar comandos de testes, saltos e comparações. Como exemplo podemos ver abaixo as rotinas de tratamento deste comando:

```
/*
   Funcoes Para Manipular Pilhas da Geracao de While  */

void while_inicio_code() {
          cont_while++;
          codw++ ;
        contagew =conta_linha;
          empilha_endw(codw, contagew, cont_while);
          printf("\nf@%d@%d", codw, contagew);
          pol_print();
```

```
        executa_pol();
        init_pol_topo();
        grava_instr (22, NN, 0, CONSTVAL,0 , bool);
        cont_while++;
        codw++ ;
      contagew =conta_linha;
        empilha_endw(codw, contagew, cont_while);
        printf ("\b@%d@%d",codw,contagew);
        whi = FALSE;
  }

int temp_codw, temp_contagew;

void while_final_code() {
   temp_codw =desempilha_endw1(cont_while);
   temp_contagew =desempilha_endw2(cont_while);
   cont_while--;
   grava_instr (22, NN, 0, CONSTVAL,0 , alw);
   codw =desempilha_endw1(cont_while);
   contagew =desempilha_endw2(cont_while);
   cont_while--;
   printf ("\b@%d@%d",codw,contagew);
   printf ("\n@%d@%d",temp_codw,temp_contagew);
  }

 void empilha_endw(int co, int conta_a, int cont_while) {
   /*   ++cont_while;          */
      pilw[cont_while].cod = co;
      pilw[cont_while].conta = conta_a;
          }

 int desempilha_endw1(int prov) {
      return(pilw[prov].cod);
          }
```

```
int desempilha_endw2(int prov) {
    return(pilw[prov].conta);
        }
```

Para cada comando de repetição (*for*, *repeat*) ou escolha condicional (*case*, *if*), e demais comandos, também existe um conjunto de rotinas que geram código de máquina correspondente.

5.2.5.1 - Produção do módulo

A saída deste módulo, para o exemplo apresentado, é o arquivo **shell.rsc**, que contém o programa do arquivo **shell.pas** transformado em código RISC-LIE. O arquivo saída é mostrado no ítem D.4.

5.3 - Análise de resultados e desempenho do compilador gerado

Valente [Vale91] usou, para avaliação do desempenho da arquitetura RISC-LIE, o *benchmark* conhecido por *Dhrystone*. Como o autor não dispunha de um compilador na época em que definiu sua arquitetura, esse *benchmark* foi construído manualmente. Uma análise do código produzido, demonstra uma otimização ideal, que só pode ser obtida quando a escrita de cada instrução é feita de forma calculada e estudada. Com isso, é possível otimizar o código de uma maneira mais eficiente do que seria obtido usando qualquer dispositivo automático. Usando de recursos que tornam o programa auto-modificável, e alocando cada variável da melhor maneira, segundo um critério de rearranjo, o programa produzido por Valente não permite uma comparação direta com um código gerado e otimizado por compiladores.

Seguindo a mesma linha adotada por Valente, a geração de código para um programa semelhante ao **dhrystone**[14] foi uma das formas de avaliação utilizadas para o compilador **Paris**. A saída produzida pelo compilador **PARIS**, neste caso, é o arquivo **dhrystone.obj**. Deve-se observar que o programa final gerado pelo compilador **PARIS**, é apenas compatível com a arquitetura RISC-LIE, não podendo ser executado diretamente na entrada do simulador

[14] O *dhrystone* original usa a rotina malloc() em várias etapas de sua execução. O programa aqui considerado realiza todas operações com um tamanho fixo, sem uso de um dispositivo de gerenciamento de memória, tal como o usado pela rotina malloc(). O motivo deste procedimento é a não existencia ainda, de um conjunto de rotinas no compilador PARIS para realizar a operação efetuada pela rotina malloc().

RISC-LIE. Isso porque ainda não se dispõe de um *linker* para realizar a montagem do código gerado com as rotinas da linguagem. A solução adotada foi efetuar manualmente a montagem das rotinas do programa gerado, que deveriam ser montadas pelo *linker*. Em função das observações anteriores, uma comparação direta com o **dhrystone** produzido por Valente [Vale91] não seria aplicável.

Outros programas para teste foram elaborados para avaliar o funcionamento do compilador. Nestes programas procurou-se utilizar todo o conjunto de instruções disponíveis do compilador **PARIS**. Os programas de teste vão desde programas genéricos, que nada de prático realizam, até programas conhecidos, tais como o torre de hanoi[15], o *quicksort*, e o *bubblesort*. Não foi detectado nenhum erro no decorrer destes testes.

O atual estágio permite apenas algumas formas de avaliação. A única realmente confiável refere-se à comparação do código gerado com e sem otimização, apesar do compilador não dispor ainda de um otimizador completo. É importante lembrar que estamos executando o compilador **PARIS** em uma máquina hospedeira (*cross compiler*), mas a execução deste compilador no ambiente RISC-LIE deverá ser totalmente diferente. Isto porque a máquina hospedeira que está suportando o simulador **PARIS** diferencia-se bastante da arquitetura RISC-LIE. Assim, as comparações tradicionalmente efetuadas, como a velocidade com que o compilador gera um código RISC-LIE, não oferecem nenhum atrativo, uma vez que estaria sendo avaliada apenas a velocidade com que a máquina hospedeira estaria fazendo esta geração. Uma medida de desempenho comparativo entre o compilador **PARIS** e qualquer outro compilador pascal também não é representativa, pois a comparação estaria sendo feita entre as máquinas, e não entre os compiladores. As comparações entre arquiteturas já foram efetuadas no trabalho de desenvolvimento da arquitetura RISC-LIE [Vale91].

Finalmente, deve-se observar que, no atual estágio do trabalho, procurou-se estabelecer uma base sólida inicial sobre a qual pudéssemos trabalhar com segurança, sem a preocupação de resolver todos os problemas referentes ao desenvolvimento de um compilador completo. Uma vez dominadas todas as etapas de construção do compilador, qualquer alteração requerida poderá ser implementada. Os resultados encorajam refinamentos que fornecerão diversas respostas que ainda não puderam ser obtidas. Quando

[15] O programa (conhecido como *tower*) foi utilizado por Tamir [Tami83] para avaliação de sua proposta.

dispusermos de ferramentas tais como um **linker** e um Sistema Operacional, poderemos avaliar outros aspectos do compilador desenvolvido, tais como o uso de registradores, se o uso das instruções é o mais correto e se a utilização das instruções propostas pela arquitetura RISC-LIE é completa e ótima. Isso será de fundamental importância, tanto para a avaliação do compilador **PARIS**, como da própria arquitetura RISC-LIE. A avaliação feita até o momento, só demonstra que o código gerado pelo simulador **PARIS** é compatível com a arquitetura RISC-LIE.

5.4 - Considerações finais

Dentro do objetivo proposto, o compilador apresentou um desempenho bastante satisfastório. Pode-se destacar como muito satisfatórios o reduzido nível de ambiguidades alcançado, e a otimização das instruções obtida.

É importante destacar-se a influência do **YACC** dentro do trabalho de construção de um compilador. Este fato deve-se, não só a sua característica de poderosa ferramenta de apoio para a geração do analisador sintático de forma rápida, mas também pelo conjunto de recursos adicionais fornecidos para a geração de um analisador mais consistente.

No atual estágio do trabalho, muitas operações ainda não foram implementadas, pelo fato de não se dispor de um Sistema Operacional que efetue as diversas interfaces requeridas. Entre estas, podemos destacar o sistema de tratamento de arquivos (para o processamento das entradas e saídas do programa gerado), e o tratamento de alocação dinâmica (para gerenciamento de ponteiros).

As rotinas de otimização atuais são simples, mas seguindo a técnica aplicada, que define cada rotina como um objeto, pode-se trabalhar cada rotina de forma independente. Deve-se observar que, apesar do trabalho ter sido desenvolvido usando uma metodologia orientada para objetos, a linguagem de implementação utilizada foi a C padrão, e não a C^{++}. Esta escolha foi feita em função da necessidade de haver homogeneidade entre o código gerado pelo **YACC** com as interfaces em linguagem C.

Capítulo 6

Conclusões e perspectivas futuras

A arquitetura RISC apresenta características de *hardware* que podem ser implantadas em muitos processadores atuais, sendo um paradigma razoável para projetos de *hardware*. Sua principal característica é a de racionalizar o uso de instruções, usando com parcimônia somente as instruções estritamente necessárias e evitando o excesso e a complexidade. O mercado atualmente vem apresentando uma tendência cada vez maior em utilizar técnicas da filosofia RISC. Como exemplo, podemos citar o projeto Alfa da Digital, PA-RISC Série 800 da Hewlett Packard, entre outros.

Pelo fato de utilizar um grupo reduzido de instruções, a construção de um compilador deve prover um bom gerenciamento desse grupo de instruções, construindo através da composição de elementos simples, uma estrutura para suprir todas as necessidades de códigos dos programadores. Assim, as instruções utilizadas da arquitetura RISC-LIE foram as estritamente suficientes para suprir todas as necessidades da construção do compilador desenvolvido neste trabalho, sendo que foram aproveitadas a maioria das instruções propostas por [Vales91]. Isso gerou a necessidade de uma otimização adicional na geração de código, para explorar eficientemente os recursos da arquitetura RISC.

É importante observar que o código deve ser otimizado não por falhas no código fonte original, mas porque muitas tarefas são exercidas diretamente pelo compilador. Por exemplo, o manuseio de arranjos é uma tarefa importante para otimização, e deverá ser efetuada pelo compilador. É importante um compilador otimizador manusear adequadamente as referências a arranjos nos *loops*, sendo esta uma tarefa fundamental para sua eficiência na melhoria do código. Como observado no ítem 4.1.1, entre as fontes de otimização relacionadas à geração de código, encontra-se a utilização eficiente dos registradores e conjuntos de instruções da máquina. Procurou-se ao longo deste trabalho suprir estes aspectos.

A primeira etapa para a construção de um compilador RISC, foi o desenvolvimento do compilador PARIS. Ainda restam muitas etapas a serem realizadas até que tenhamos um compilador RISC completo. Nesta etapa construiu-se um compilador com muitas características RISC, sendo importante destacar que o compilador atualmente é um simulador de código para um simulador de arquitetura RISC-LIE. Em trabalhos futuros é importante que o próprio compilador seja escrito em RISC-LIE. Isto pode ser conseguido escrevendo-se um compilador em linguagem C para o RISC-LIE. Como todo o código PARIS gerado encontra-se

em linguagem C, bastaria a sua nova compilação usando a linguagem C para RISC para obter-se o compilador PARIS em RISC-LIE. A partir de então, poder-se-ia começar a desenvolver outras importantes ferramentas para o ambiente de programação, como o Sistema Operacional, *linker*, carregadores, editores de programa, interpretadores, e outras. Com o desenvolvimento de um Sistema operacional e com a definição de seus módulos básicos, seria possível realizar-se o gerenciamento de memória, essencial na comunicação com o compilador, e também a interface com o sistema de arquivos. Sem uma definição de quais e como são os módulos desse sistema, fica impossível requerer do atual compilador mais do que uma simples geração de código. Somente com a existência do Sistema Operacional e de um *linker* seria possível o carregamento do código gerado, de forma a calcular os endereços de memória, e demais requisitos para execução do código.

É importante também destacar o uso do YACC, como uma ferramento de apoio ao desenvolvimento do compilador. A utilização do YACC como ferramenta de apoio tem se limitado, na maioria dos casos, apenas à construção de pequenos compiladores, e interpretadores, como o *high order calculator*, ou HOC. Neste trabalho, procurou-se implementar um compilador de forma completa - não apenas o aspecto de geração de código na máquina hospedeira, mas a implementação de todo o conjunto de instruções de uma nova máquina (RISC-LIE). Não foi encontrada na literatura nenhuma referência de utilização do YACC semelhante à desenvolvida neste trabalho. Neste sentido, todas as dificuldades apresentadas no Capítulo 5 mostram que, tomando-se alguns cuidados é possível desenvolver-se um compilador otimizado e eficiente.

Pode-se dizer que o compilador construído comportou-se de forma satisfatória quando solicitado nas diferentes situações de teste a que foi submetido. O compilador mostrou-se capaz de gerar código otimizado para os *benchmarks* testados, comunicando-se bem com o simulador RISC-LIE. As otimizações, tanto tradicionais, como de características RISC (o *delayed branch*, otimizações em registradores, construção otimizada de chamada de rotinas, etc...) foram satisfatórias e preencheram as espectativas.

Pode-se afirmar que os resultados alcançados encorajam a construção de arquitetutas RISC e também a construção de compiladores otimizados para essa arquitetura. As duas etapas de trabalho desenvolvidas até o presente - construção do simulador RISC-LIE e do compilador PARIS - sem dúvida estimulam a continuação do projeto, com o objetivo de torná-lo completo para uma análise de desempenho global e abrangente.

Finalmente, podemos afirmar que os resultados alcançados confirmam as conclusões obtidas por Valente [Vale91]. Entre as principais, podemos citar a comprovação de que um conjunto reduzido e bem escolhido de instruções, combinado com o restante das características da arquitetura, pode resultar em arquiteturas de alto desempenho, e na redução do tempo de execução das instruções.

SERVIÇO DE BIBLIOTECA E INFORMAÇÃO - IFQSC
FÍSICA

BIBLIOGRAFIA

[AcBa87] Ackerman, M.; Baum, G. - ***The Fairchild Clipper, Byte, april,*** 1987.

[Ahou77] Aho, V. A.; Ullman, J.D. - ***Principles of Compiler Design, Addison-Wesley Publishing Company, 1 th edition,*** 1977.

[Adva87] *Devices, Advanced Micro,-* ***Am29000 Users's Manual, Advanced Micro Devices,*** 1987.

[Alma89] Almasi, G. S.; Gottieb. A. - ***Highly Parallel Computing, the Benjamin/Cummings Publishing Company,*** Inc, 1989.

[Bell86] Bell, C. G. - ***RISC: Back to the Future?, Datamation, june,*** 1986.

[Blom83] Blomseth, R. - ***A Big RISC, Report UCB/CSD 83/143, Computer Science Division, University of California, Berkeley,*** 1983.

[Chai82] Chaitin ,G. J. et al. - ***Register Allocations via Coloring, Proceedings of SIGPLAN Symposium on Compiler Construction, june,*** 1982.

[ChHe84] Chow, F.; Hennessy, J. - ***Register Allocation by Priority-Based Coloring, Proceedings of the SIGPLAN Symposium on Compiler Construction, june,*** 1984.

[ChHo87] Chow, P.; Horowitz M. - ***Architectural Tradeoffs in Design of MIPS-X, The 14th Annual International Symposium on Computer Architecture.***

[Chow86] Chow, F et al.- ***Engineering a RISC Compiler System, Proceedings of COMPCON Spring 86, march,*** 1986.

[Clip86] *Fairchild, CLIPPER Module,* ***32-bit Microprocessor Product Description,*** *Prentice Hall, Inc.* 1986.

[Clip87] *Fairchild, CLIPPER Module,* ***32-bit Microprocessor Product Description,*** *Prentice Hall, Inc.* 1987.

[CoHi85] Colwell,R. P.; Hitchcock III,C. Y.; Jensen, E. D. Sprunt,H. M. B.; Kollin, C. P.- ***More Controversy about "Computers, Complexity, and***

Controversy", IEEE Computer, december, 1985.

[Colw85] Colwell,R. P.; Hitchcock III,C. Y.; Jensen, E. D. Sprunt,H. M. B.; Kollin, C. P.- ***Instruction Set and Beyond: Computers, Complexity, and Controversy,*** *IEEE Computer, september*, 1985.

[Coop83] Cooper, D. - ***Standard Pascal - User Reference Manual,*** *W. W. Norton & Company - New York*, 1983.

[Croo86] Crookes, J. G.; Balmer, D. W., - ***A Three-Phase Simulation System Writem in Pascal,*** *Journal of the Operational Research Society* Vol 37, No. 6, 1986.

[Donn88] Donnelly, C.; Stallman, R. -***BISON- The YACC-compatible Parser Generator,*** *Bison Reference Manual, Free Software Foundation*, 1988.

[Farr85] Farrer, H. et alli - **Pascal Estruturado**, Guanabara Dois, 1985.

[Flyn87] Flynn, M. J.; Mitchell C. L.; Mulder, J. M. - ***And Now a Case for More Complex Instructions Sets,*** *IEEE Computer, september* 1987.

[Foxe86] Fox, E. et al. - ***Reduced Instruction Set Architecture for a GaAs Microprocessor System,*** *IEEE Computer, october*, 1986.

[Ghez87] Ghezzi C. E.; Jazayeri V. M. - ***A Survey of RISC Processors and Computers of the Mid-1980s,*** *Computer, september* 1987.

[GheJ87] Ghezzi C. E.; Jazayeri V. M. - **Conceitos de Linguagens de Programação**, Editora Campus, 1987.

[Gima87] Gimarc C. E.; Milutinovic, V. M. - ***A Survey of RISC Processors and Computers of the Mid-1980s,*** *Computer, september* 1987.

[Gonz93] Gonzalez, A.M.; Llaberia J.M. - ***Reducing Branch Delay to zero in Pipelined Processors,*** *Computer, march* 1993.

[Good77] Goodman, S.E.; Hedetniemi S.T. - ***Introduction to Design and Analisys of Algorithms,*** *MacGraw-Hill International Editions - Computer Science Series* 1977.

[Gott93] Gottfried B. S. - **Programando em C**, Schaum McGraw-Hill, 1993.

[Heat84] Heath, J.L. - ***Re-evaluation of the RISC I**, Computer Architecture News, march*, 1984.

[Henn84] Hennessy, J. L. - ***VISI Processor Architecture**, IEEE Computer, december*, 1884.

[Hill86] Hill, Mark et al. - ***Design Decisions on SPUR**, IEEE Computer, november*, 1986.

[Hitc85] Hitchcoock, C. y.; Sprunt, H. M. B. - ***Analyzing Multiple Register Sets**, Proceedings of the 12th Annual Interancional Symposium on Computer Architecture, july*, 1985.

[Holli89] Hollingsworth,W. Sachs, H.: Smith A. J. - ***The Clipper Processor: Instruction Set Architecture and Implementation**, Communications of the ACM*, february 1989.

[Hwan84] Hwang, K.; Briggs F. A. - ***Computer Architecture and Parallel Processing**, MacGraw-Hill Book Company*, 1984.

[John87] Johnson, T. - ***The RISC/CISC Melting Pot**, Byte, april*, 1987.

[Inte89] ***A Performance Report on the 386 Family of High Perfomance 32-bit Microprocessors** Intel Corporation, april*, 1989.

[Kate84] Katevenis, M.; ***Reduced Instruction Set Computer for VLSI**, - ACM Doctoral Dissetation Award 1984, The MIT Press.*

[KeRi88] Kernighan, B. W.; Ritchie, D.M - ***The C Programming Language**, 2nd Edition, Prentice Hall*, 1988.

[Kern84] Kernighan, B. W.; Pike, R. - ***The UNIX Programming Environment**, Prentice-Hall*, 1984.

[Koop87] Koopman, P. - ***The WISC Concept**, Byte, april*, 1987.

[Kowa83] Kowaltowski, Thomasz - **Implementação de Linguagens de Programação**, Editora Guanabara Dois- I edição - 1983.

[Leej67] Lee, J. A. N.- ***The Anatomy of a Compiler***, *Van Nostrand Reinhold Company*, 1967.

[Lewi76] Lewis, P. M.: Rosenkrantz, D. J.; Stearns, R. E. - ***Compiler Design Theory***, *Addison-Wesley Publishing Company*, 1976.

[Lund77] Lunde, A. - ***Empirical Evaluation of Some Features of Instruction Set Processor Architectures*** Communications of the ACM, march, 1977.

[Magi72] Maginnis, J. B. - ***Elements of Compiler Construction***, *Appleton-Century-Crofts - Meridith Corporation*, 1972.

[Maho86] Mahon, M et al. - ***Hewlett-Packard Precision Architecture: The Processor***, *Hewlett-Packard Journal*, Agosto 1986.

[McDo87] McDonald, J. et al. - ***Wafer Scale Interconnections for GaAs Packing-Applications to RISC Architecture***, *IEEE Computer, april*, 1987.

[Mizr90] Mizrahi, V. V. - **Treinamento em Linguagem C - Curso completo - Módulo 1 e Módulo 2**, Editora McGraw-Hill Ltda, 1990.

[Mokh86] Mokhoff, N. - ***New RISC Machines Appear as Hybrids with boths RISC and CISC Features***, *Computer Design, april*, 1986.

[Neto87] Neto, J. J. - **Introdução à Compilação**, Livros Técnicos e Científicos Editora S.A., 1987.

[Nich75] Nicholls, J. E. - ***The Structure and Design of Programming Languages***, *Addison Wesley Publishing Company*, 1975.

[McNe87] McNeley, K. J.; Milutinovic - ***Emulating a Complex Instruction Set Computer with a Reduced Instruction Set Computer***, *IEEE Micro, february*, 1987.

[PaHe85] Patterson, D. A.; Hennessy, J. - ***Response to Computers, Complexity and Controversy***, *IEEE Computer, november*, 1985.

[PaPi82] Patterson, D. A.; Piepho, R. S. - ***Acessing RISCs in High-Level Language Suport***, *IEEE Micro, november*, 1982.

[PaSe81] Patterson, D. A.; Séquin, C. - ***RISC I: A Reduced Instruction Set VLSI Computer***, *Proc of the 8th Symposium on Computer Architecture, ACM SIGARCH CAN 9.3, may*, 1981.

[PaSe82] Patterson, D. A.; Séquin, C. - ***A VLSI RISC***, *IEEE Computer, september*, 1982.

[Patt84] Patterson, D. A. - ***RISC Watch***, *Computer Architecture News, march*, 1984.

[Patt85] Patterson, David A.- ***Reduced Instruction Set Computers***, *Communictions of the ACM, january* 1985.

[Patt87] Patterson, D. A. - ***A Progress Report on SPUR: February 1, 1987***, *Computer Architecture, march*, 1983.

[Pino89] Pino, G. A del; Marrone, L. A.- **Arquiteturas RISC**, Publicação do IV EBAI - Escola Brasil Argentina de Informatica- 1989.

[Plau91] Plauger, P. J. ; Brodie, J. - ***Standard* C - Guia de Referência Básica - *Quick Reference***, Editora McGraw-Hill Ltda, 1991.

[Radf86] Radford, L. E.; Haigh, R. W. - **Turbo Pascal para IBM PC e compatíveis**, Guanabara Dois, 1986.

[Radi83] Radin G.; Cocke, J. - ***The 801 Minicomputer***, *IBM Journal of Research and Development, may*, 1986. .

[Rass86] Rasset, T. et al. - ***A 32-bit RISC Implementation in Enhancement-Mode JFET GaAs***, *IEEE Computer, october*, 1986.

[Robi87] Robinson, P. - ***How Much of a RISC ?***, *Byte, april*, 1987

[Samp85] Samples, A.D. et al. - *SOAR Architecture, Report 226, Computer Science Division, University of California, Berkeley*, 1982.

[Schi89] Schildt, H. - C Avançado - Guia do Usuário - Editora McGraw-Hill, segunda edição, 1989.

[Setz86] Setzer, W. W.; Melo, I. H. - A construção de um Compilador Editora Campus, terceira edição, 1986.

[Stal88] Stallings, W. - *Reduced Instruction Set Computer Architecture, IEEE Transactions on Computer, vol.76, n.1, january* 1988.

[StHe88] Steenkiste, J.; Hennessy J. - *Lisp on a Reduced Instruction Set-Processor: Characterization and Optimization, IEEE Computers, july*,1988.

[Ston75] Stone, H. S. et alii, *Introduction to Computer Architecture, Science Research Associates, Inc*, 1975.

[TaSe83] Tamir, Y.; Séquin C. - *Strategies for Managing the Register in RISC, IEEE Transactions on Computers, vol C-32, november*, 1983.

[Tane78] Tanenbaum, A. S - *Implications of Structured Programming for Machine Architecture, Communications of the ACM, march*, 1978.

[Tane84] Tanenbaum, A. S - *Opeting Systems - Design and Implementations, Prentice-Hall, Inc*, 1984.

[Tayl86] Taylor, G. at al. - *Evaluation of the SPUR Lisp Architecture,*

[Trel82] Treleaven, P. et al. - *VLSI Processo Architectures, IEEE Computer, june*, 1982.

[Trem85] Tremblay, J. P.; Sorenson, P. G. - *The Theory and Practice on Compiler Writing, MacGraw-Hill Book Company*, 1985.

[UnPa87] Ungar, S.; Patterson, D. A. - *What Price Smaltalk, IEEE Computer, january*, 1987.

[Vale91] Valente, F. J. - **Proposta e e Simulação de uma Arquitetura RISC**, Dissertação de Mestrado - IFQSC USP, 1991.

[Vegd84] Vegdahl, S. - ***A Survey of Proposed Architectures for the Execution of Funcional Languages***, *IEEE Transactions on Computers, december*, 1984.

[Your75] Yourdon, E. - ***Techniques of Program Scruture and Design***, Englewood Cliffs, N. J.: Prentice-Hall, Inc, 1975 p 144.

[Warr80] Warren, D. H. D. -***Logic Programming and Compiler Writing Software Pratice and Experience***, *Vol 10 (pp 97-125)*, 1980.

[Weic84] Wiecker, R. P. - ***Dhrystone: A Synthetic Systems Programming Benchmark***, *Communications of the ACM, october*, 1984.

[Weis87] Weiss, R.- ***RISC Processors: the New Wave in Computer Systems*** *Computer Design, pp 53-10, may* 1985.

[Wins90] Winston, A. -***Fast Company***, *UnixWorld, september*, p. 78-84, 1990.

[Wirt78] Wirth, N.; Jensen K. - ***Pascal User Manual an Report***, *New York Springer-Verlag*, 1978, p. 133.

[Wirt86] Wirth, N. - ***Algorithms and Data Structures***, *Prentice-Hall*, 1986.

[Wulf 75] Wulf, W. et al. - ***The Design of an Optmizing Compiler***, *American Elsevier Publishing Company*, 1975.

[Wulf81] Wulf, W. A. - ***Compilers and Computer Architecture***, *IEEE Computer, july*, 1981.

APÊNDICE A

A.1 Estrutura da BNF estendida para linguagem Pascal Padrão

actual_parameter = expression | variable_access | procedure_identifier | function_identifier.

actual_parameter_list = '(' actual_parameter { ',' actual_parameter } ')'.

adding_operator = '+' | '-' | 'OR'.

apostrophe_image = "'".

array_type = 'ARRAY' '[' index_type { ',' index_type } ']' 'OF'component_type.

array_variable = variable_access.

assignment_statement = (variable_access | function_identifier) ':=' expression.

base_type = ordinal_type.

block = label_declaration_part constant_definition_part type_definition_part variable_declaration_part procedure_and_function_declaration_part statement_part.

boolean_expression = expression.

bound_identifier = identifier.

buffer_variable = file_variable '^'.

case_constant = constant.

case_constant_list = case_constant { ',' case_constant }.

case_index = expression.

case_list_element = case_constant_list ':' statement.

case_statement = 'CASE' case_index 'OF' case_list_element { ';' case_list_element } [';'] 'END'.

SERVIÇO DE BIBLIOTECA E INFORMAÇÃO - IFQSC

character_string = "' string_element {string_element } "'.

component_type = type_denoter.

component_variable = indexed_variable | field_designator.

compound_statement = 'BEGIN' statement_sequence 'END'.

conditional_statement = if_statement | case_statement.

conformant_array_parameter_specification

= value_conformant_array_specification | variable_conformant_array_specification.

conformant_array_schema = packed_conformant_array_schema | unpacked_conformant_array_schema.

constant = [sign] (unsigned_number | constant_identifier) | character_string.

constant_definition = identifier '=' constant.

constant_definition_part = ['CONST' constant_definition ';' { constant_definition ';' }].

constant_identifier = identifier.

control_variable = entire_variable.

digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'.

digit_sequence = digit { digit }.

directive = letter { letter | digit }.

domain_type = type_identifier.

else_part = 'ELSE' statement.

empty_statement = .

entire_variable = variable_identifier.

enumerated_type = '(' identifier_list ')'.

expression = simple_expression [relational_operator simple_expression].

factor > variable_access | unsigned_constant | function_designator | set_constructor | '(' expression ')' | 'NOT' factor.

factor > bound_identifier.

field_designator = record_variable '.' field_specifier | field_designator_identifier.

field_designator_identifier

= identifier.

field_identifier = identifier.

field_list = [(fixed_part [';' variant_part] | variant_part) [';']].

field_specifier = field_identifier.

file_type = 'FILE' 'OF' component_type.

file_variable = variable_access.

final_value = expression.

fixed_part = record_section { ';' record_section }.

for_statement = 'FOR' control_variable ':=' inicial_value ('TO' | 'DOWNTO') final_value 'DO' statement.

formal_parameter_list

= '(' formal_parameter_section { ';' formal_parameter_section } ')'.

formal_parameter_section

> value_parameter_specification |
variable_parameter_specification |
procedural_parameter_specification |
functional_parameter_specification.

formal_parameter_section

```
                    > conformant_array_parameter_specification.

fractional_part     = digit_sequence.

function_block      = block.

function_declarations = function_heading ';' directive | function_identification ';'
                    function_block | function_heading ';' function_block.

function_designator = function_identifier [ actual_parameter_list ].

function_heading    = 'FUNCTION' identifier [ formal_parameter_list ] ':' result_type.

function_identification = 'FUNCTION' function_identifier.

function_identifier = identifier.

functional_parameter_specification

                    = function_heading.


goto_statement      = 'GOTO' label.

identified_variable = pointer_variable '^'.

identifier          = letter { letter| digit }.

identifier_list     = identifier { ',' identifier }.

if_statement        = 'IF' boolean_expression 'THEN' statement [ else_part ].

index_expression    = expression.

index_type          = ordinal_type.

index_type_specification = identifier '..' identifier ':' ordinal_type_identifier.

indexed_variable    = array_variable '[' index_expression { ',' index_expression } ']'.

inicial_value       = expression.

label               = digit_sequence.

label_declaration_part = [ 'LABEL' label { ',' label } ';' ].
```

letter = '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'.

member_designator = expression { '..' expression }.

multiplying_operator = '*' | '/' | 'DIV' | 'MOD' | 'AND'.

new_ordinal_type = enumerated_type | subrange_type.

new_pointer_type = '^' domain_type.

new_structured_type = ['PACKED'] unpacked_structured_type.

new_type = new_ordinal_type | new_structured_type | new_pointer_type.

ordinal_type = new_ordinal_type | ordinal_type_identifier.

ordinal_type_identifier = type_identifier.

packed_conformant_array_schema

= 'PACKED' 'ARRAY' '[' index_type_specification ']' 'OF' type_identifier.

pointer_type = new_pointer_type | pointer_type_identifier.

pointer_type_identifier = type_identifier.

pointer_variable = variable_access.

procedural_parameter_specification

= procedure_heading.

procedure_and_function_declaration_part

= { (procedure_declaration | function_declaration) ';' }.

procedure_block = block.

procedure_declaration = procedure_heading ';' directive | procedure_identification ';' procedure_block | procedure_heading ';' procedure_block.

procedure_heading = 'PROCEDURE' identifier [formal_parameter_list].

procedure_identification = 'PROCEDURE' procedure_identifier.

procedure_identifier = identifier.

procedure_statement = procedure_identifier ([actual_parameter_list] | read_parameter_list |readln_parameter_list | write_parameter_list | writeln_parameter_list).

program = program heading ';' program_block '.'.

program_block = block.

program_heading = 'PROGRAM' identifier ['(' program_parameters ')'].

program_parameters = identifier_list.

read_parameter_list = '(' [file_variable ','] variable_access { ',' variable_access } ')'.

readln_parameter_list = ['(' (file_variable | variable_access) {',' variable_access } ')'].

real_type_identifier = type_identifier.

record_section = identifier_list ':' type_denoter.

record_type = 'RECORD' field_list 'END'.

record_variable = variable_access.

record_variable_list = record_variable { ',' record_variable }.

relacional_operator = '=' | '<>' | '<' | '>' | '<=' | '>=' | 'IN'.

repeat_statement = 'REPEAT' statement_sequence 'UNTIL' boolean_expression.

repetitive_statement = repeat_statement | while_statement | for_statement.

result_type = simple_type_identifier | pointer_type_identifier.

scale_factor = signed_integer.

set_constructor = '[' [member_designator { ',' member_designator }] ']'.

set_type = 'SET' 'OF' base_type.

sign = '+' | '-'.

signed_integer = [sign] unsigned_integer.

signed_number = signed_integer | signed_real.

signed_real = [sign] unsigned_real.

simple_expression = [sign] term { adding_operator_term }.

simple_statement = empty_statement | assigment_statement | procedure_statement |goto_statement.

simple_type = ordinal_type | real_type_identifier.

simple_type_identifier = type_identifier.

special_symbol = '+' | '-' | '*' | '/' | '=' | '<' | '>' | '[' | ']' | '.'| ',' | ':' | ';' |'^' | '(' | ')' | '<>' | '<=' | '>=' | ':=' | '..' | word_symbol.

statement = [label ':'] (simple_statement | structured_statement).

statement_part = compound_statement.

statement_sequence = statement { ';' statement }.

string_character = one_of_a_set_of_implementation_defined_characters.

string_element = apostrophe_image | string_character.

structured_statement = compound_statement | conditional_statement | repetitive_statement |with_statement.

structured_type = new_structured_type | structured_type_identifier.

structured_type_identifier= type_identifier.

subrange_type = constant '..' constant.

tag_field = identifier.

tag_type = ordinal_type_identifier.

term = factor { multiplying_operator_factor }.

type_definition = identifier '=' type_denoter.

type_definition_part = ['TYPE' type_definition ';' { type_definition ';' }].

type_denoter = type_identifier | new_type.

type_identifier = identifier.

unpacked_conformant_array_schema

= 'ARRAY' '[' index_type_specification { ';' index_type_specification } ']' 'OF' (type_identifier |conformant_array_schema).

unpacked_structured_type = array_type | record_type | set_type | file_type.

unsigned_constant = unsigned_number | character_string | constant_identifier | 'NIL'.

unsigned_integer = digit_sequence.

unsigned_number = unsigned_integer | unsigned_real.

unsigned_real = unsigned_integer '.' fractional_part ['E' scale_fator] | unsigned_integer 'E' scale_factor.

value_conformant_array_specification

= identifier_list ':' conformant_array_schema.

value_parameter_specification

= identifier_list ':' type_identifier.

variable_access = entire_variable | component_variable | identified_variable |buffer_variable.

variable_conformant_array_specification

= 'VAR' identifier_list ':' conformant_array_schema.

variable_declaration = identifier_list ':' type_denoter.

variable_declaration_part = ['VAR' variable_declaration ';' { variable_declaration ';' }].

variable_identifier = identifier.

variable_parameter_specification

= 'VAR' identifier_list ':' type_identifier.

variant = case_constant_list ':' '(' field_list ')'.

variant_part = 'CASE' variant_selector 'OF' variant { ';' variant }.

variant_selector = [tag_field ':'] tag_type.

while_statement = 'WHILE' boolean_expression 'DO' statement.

with_statement = 'WITH' record_variable_list 'DO' statement.

word_symbol = 'PROGRAM' | 'LABEL' | 'CONST' | 'TYPE' | 'PROCEDURE' | 'FUNCTION' | 'VAR' | 'BEGIN' | 'END' | 'DIV' | 'MOD' | 'AND' | 'NOT' |'OR' | 'IN' | 'ARRAY' | 'FILE' | 'RECORD' | 'SET' | 'PACKED' | 'CASE' | 'OF' | 'FOR' | 'TO' | 'DOWNTO' | 'DO' | 'IF' | 'THEN' | 'ELSE' | 'REPEAT' | 'UNTIL' | 'WHILE' | 'WITH' | 'GOTO' | 'NIL'.

write_parameter = expression [':' expression [':' expresssion]].

write_parameter_list = '(' [file_variable ','] write_parameter { ',' write_parameter } ')'.

writeln_parameter_list = ['(' (file_variable | write_parameter) {',' write_parameter } ')'].

A.2 Regras Sintáticas do compilador PARIS

```
%%
/*
    86 Regras Sintáticas do compilador PARIS
                                                    /*

program : program_heading ';' block '.' { retorna_escopo();
        if (SINT) { printf("\n                    ");
                    printf("\n  FIM DE PROGRAMA  ");
                    printf("\n                    ");} }
        ;

block : cons_def_par type_definition_part variable_declaration_part
        procedure_and_function_declaration
        compound_statement {
        if (SINT) printf("\n=> fim de bloco"); }
      ;

nada : { if (SINT) printf("\ncadeia vazia"); }
     ;

cons_def_par  : CONST consta  {
        if (SINT) printf("\ndefinicao de constante: %s",$2);    }
             | nada  {
        if (SINT) printf("\nsem constante");    }
             ;

consta : idf IG constant ';'
             {
        if (SINT) printf("\nidentificador de constante = %s", nome_simbolo($1));
             todovalor.c_integer=0;
             substitui_simb ($1, c_integer, todovalor);
             }
      | idf IG constant ';' consta
             {
        if (SINT) printf("\nidentificador de varias constante ; %s", nome_simbolo($1));
             todovalor.c_integer=0;
             inic_percorre_lista_var ();
                while ( le_lista_var() != 0 )
                    substitui_simb ($1,  c_integer, todovalor);
             }
      ;

enumerated_type     : '(' id_list ')' {
        if (SINT) printf("\ntipo enumerado com lista");    }
                    ;

new_ordinal_type    : enumerated_type {
        if (SINT) printf("\nnovo tipo enumerado: "); }
                    | subrange_type {
        if (SINT) printf("\nnovo tipo subrangeado: "); }
                    ;
```

```
new_type     : new_ordinal_type {
        if (SINT) printf("\nnovo tipo ordinal: ");    }
            | '^' idf {
        if (SINT) printf("\nponteiro para novo tipo: %s", nome_simbolo($2));
                todovalor.c_integer=0;
                substitui_simb ($2,  label, todovalor); }
            |  unpacked_structured_type { /*nao feito */  }
            ;

type_definition_part :  nada  {
        if (SINT) printf("\nsem definicao de tipo");    }
                    |  TYPE type {
        if (SINT) printf("\nfim de definicao de tipo");    }
                    ;

type  : type_definition ';'        {
        if (SINT) printf("\ndefinicao de tipo");    }
     | type_definition ';'  type  {
        if (SINT) printf("\ndefinicoes de tipo");   }
     ;

type_definition : idf IG type_denoter {
                todovalor.c_integer=0;
                substitui_simb ($1,  tipo, todovalor);
        if (SINT) printf("\ndefinicao de  tipo  = tipo denotado");  }
                ;

program_heading: PROGRAM idf
                {
        if (SINT) printf("\nNome do programa sem lista: %s", nome_simbolo($2));
                  todovalor.c_integer=0;
                  substitui_simb ($2,  label, todovalor);
                }
            | PROGRAM idf '(' id_list_arquivo ')'
                { st_Symbol *indice;
                  todovalor.c_integer=0;
                  substitui_simb ($2,  label, todovalor);
        if (SINT) printf("\nNome do programa com lista: %s", nome_simbolo($2));
                  inic_percorre_lista_var ();
                  while ((indice=le_lista_var()) != 0 ){
        if (SINT) printf("\nlendo %s", indice);
                        todovalor.c_integer=0;
                        substitui_simb (indice,  arquivo, todovalor);
                    }
                  reinic_lista();
            }
           ;

id_list_arquivo: idf     {
            poe_lista_var($1);
        if (SINT) printf("\nVariavel_arquivo: %s", nome_simbolo($1));  }
      | idf ',' id_list_arquivo
            {
            poe_lista_var($1);
        if (SINT) printf("\nVariaveis de arquivo: %s", nome_simbolo($1));
            }
      ;
```

```
subrange_type  : constant  RG constant {
        if (SINT) printf("\nsub-range com constantes"); }
            ;

type_denoter : INTEGER {
        if (SINT) printf("\ntipo denotado INTEGER");
                tipo= v_integer;
                }
            | REAL    {
        if (SINT) printf("\ntipo denotado REAL");
                tipo = v_real;
                }
            | BOOLEAN {
        if (SINT) printf("\ntipo denotado BOOLEAN");
                tipo = v_boolean;
                }
            | CHAR    {
        if (SINT) printf("\ntipo denotado CHAR");
                tipo = v_char;
                }
            | new_type {
        if (SINT) printf("\ntipo denotado novo tipo");
            /* a ser determinado em definicoes dectetadas adiante */
                }
            ;

variable_declaration  : id_list ':' type_denoter
                      { st_Symbol *indice;
        if (SINT) printf("\ndeclaracao de lista de variaveis: ");
                        inic_percorre_lista_var ();
                        while ( (indice=le_lista_var()) != 0 ){
                          substitui_simb (indice,  tipo, todovalor);}
                       reinic_lista();
                      }
                    ;

vary : variable_declaration ';'   {
        if (SINT) printf("\numa declaracao de VAR");   }
    | variable_declaration ';' vary {
        if (SINT) printf("\nvarias VAR");   }
    ;

idf : IDENTIFICADOR {
        if (SINT) printf("\n~IDENTIFICADOR %s",$1); }
    ;

variable_declaration_part :  nada  {
        if (SINT) printf("\nsem declaracao de VAR");  }
            | VAR vary  {
        if (SINT) printf("\nDeclaracao de Variavel -> VAR");   }
            ;

id_list: idf  {    poe_lista_var($1); }
      | idf ',' id_list {
                poe_lista_var($1);
        if (SINT) printf("\nVariavel -> %s", nome_simbolo($1));
                        }
      ;
```

```
constant :    C_INTEGER { tipo = c_integer;
        if (SINT) printf("\nconstante inteira"); }
         |    C_REAL    {   pol_var($1);
                        tipo = c_real;
                        todovalor.c_real= 0.0;
                        substitui_simb ($1, tipo, todovalor);
        if (SINT) printf("\nconstante real:  "); }
         | '+' C_INTEGER {   pol_var($2);
                        tipo = c_real;
                        todovalor.c_real= 0.0;
                        substitui_simb ($2, tipo, todovalor);
        if (SINT) printf("\nconstante inteira com sinal (+)");  }
         | '-' C_REAL    { tipo = c_real;
        if (SINT) printf("\nconstante real com sinal (-)"); }
         | '-' C_INTEGER { tipo = c_integer;
        if (SINT) printf("\nconstante inteira com sinal (-)"); }
         | '+' C_REAL    {   pol_var($2);
                        tipo = c_real;
                        todovalor.c_real= 0.0;
                        substitui_simb ($2, tipo, todovalor);
        if (SINT) printf("\nconstante real com sinal (+)");  }
         | idf      {
        if (SINT) printf("\nNome da constante: %s", nome_simbolo($1));
                        todovalor.c_integer=0;
                        substitui_simb ($1,  tipo, todovalor); }
         ;

procedure_and_function_declaration : nada  {
        if (SINT) printf("\nsem funcao nem procedimento");  }
                                | pfd procedure_and_function_declaration
                                  {
        if (SINT) printf( "com procedimento ou funcao"); }
                                ;

pfd  : FUNCTION  pf_part  { retorna_escopo();
        if (SINT) printf("\ndefinicao de funcao");    }
    | PROCEDURE pd_part  { retorna_escopo();
        if (SINT) printf("\ndefinicao de procedimento");   }
    ;

pf_part        : function_declaration  {
        if (SINT) printf("\ndeclaracao de funcao");   }
             | function_declaration ';' pf_part  {
        if (SINT) printf("\ndeclaracao de <funcoes>|<procedimentos>");   }
             ;

pd_part        : procedure_declaration   {
        if (SINT) printf("\ndeclaracao de procedure"); }
             | procedure_declaration ';' pf_part ';'{
        if (SINT) printf("\nvarias declaracoes de procedure"); }
             ;

function_declaration  : function_heading  ';' idf ';' {
        if (SINT) printf("\nNome da funcao com cabecalho: %s", nome_simbolo($3));
                        todovalor.c_integer=0;
                        substitui_simb ($3,  funcao, todovalor); }
                    |  idf ';' block ';'
                        {
```

```
        if (SINT) printf("\nNome do programa com lista: %s", nome_simbolo($1));
                        todovalor.c_integer=0;
                        substitui_simb ($1,  funcao, todovalor); }
                  | function_heading  ';' block ';' {
        if (SINT) printf("\ncabecalho de funcao e bloco"); }
                  ;

function_heading      : idf ':' idf {
        if (SINT) printf("\ncabecalho %s : %s", nome_simbolo($1), nome_simbolo ($3));
                        todovalor.c_integer=0;
                        substitui_simb ($1, funcao, todovalor);
                        todovalor.c_integer=0;
                        substitui_simb ($3,  funcao, todovalor); }
                  | formal_parameter_list ':' idf {
        if (SINT) printf("\nlista de parametros: %s", nome_simbolo($3));
                        todovalor.c_integer=0;
                        substitui_simb ($3,  funcao, todovalor); }
                  ;

formal_parameter_list   : '(' fpl  ')' {
        if (SINT) printf("\nlista de parametros formais" );    }
                        ;

fpl  : formal_parameter_section {
        if (SINT) printf("\nsecao de parametro formal");   }
    | formal_parameter_section  ';' fpl {
        if (SINT) printf("\nsecao de parametros formais");    }
    ;

formal_parameter_section : v_p_s {
        if (SINT) printf("\nsecao de parametros");   }
                        | variable_parameter_specification {
        if (SINT) printf("\nespecificacao parametro de variavel");   }
                        | procedure_heading {
        if (SINT) printf("\nespecificacao de cabecalho de procedimento");   }
                        | function_heading  {
        if (SINT) printf("\nespecificacao de cabecalho de funcao");    }
                        | c_a_p_s {
        if (SINT) printf("\nespecificao de parametro de arranjo");    }
                        ;

variable_parameter_specification : VAR id_list ':' idf {
                                st_Symbol *indice;
                                inic_percorre_lista_var ();
                                while ( (indice=le_lista_var()) != 0 ){
                                   substitui_simb (indice,  tipo, todovalor);}
                                reinic_lista();
                                todovalor.c_integer=0;
                                substitui_simb ($4, tipo, todovalor);
        if (SINT) printf("\nespecificacao de  parametros de VAR");    }
                                ;
```

```
procedure_declaration    : procedure_heading ';' idf ';' {
        if (SINT) printf("\ndeclaracao de procedimento: %s", nome_simbolo($3));
                        todovalor.c_integer=0;
                        substitui_simb ($3,  procede, todovalor); }
                       | idf ';' block  ';'
                       {
        if (SINT) printf("\nprocedimento: %s bloco", nome_simbolo($1));
                        todovalor.c_integer=0;
                        substitui_simb ($1,  procede, todovalor); }
                       | procedure_heading ';' block ';' {
        if (SINT) printf("\nproc_head block"); }
                       ;

procedure_heading   : idf {
        if (SINT) printf("\nNome cabecalho do procedimento: %s", nome_simbolo($1));
                        todovalor.c_integer=0;
                        substitui_simb ($1,  procede, todovalor); }
                  | idf  formal_parameter_list {
        if (SINT) printf("\nnome cabecalho de procedimento %s lista de parametros",
nome_simbolo($1));
                        todovalor.c_integer=0;
                        substitui_simb ($1,  procede, todovalor); }
                  ;

v_p_s : id_list ':' idf  {
                        st_Symbol *indice;
                        if (SINT) printf("\nlista de variaveis");
                        inic_percorre_lista_var ();
                        while ( (indice=le_lista_var()) != 0 )
                                substitui_simb (indice,  tipo, todovalor);
                  }
      ;

c_a_p_s : v_c_a_s {
        if (SINT) printf("\nespecificacao de valor de arranjo");   }
        | var_c_a_s {
        if (SINT) printf("\nvaraiavel de arranjo estruturado" ); }
        ;

v_c_a_s : id_list ':' conformant_array_schema {
        if (SINT) printf("\nvalor de arranjo"); }
        ;

var_c_a_s : VAR id_list ':' conformant_array_schema {
                        st_Symbol *indice;
                        inic_percorre_lista_var ();
                        while ( (indice=le_lista_var()) != 0 ){
                                substitui_simb (indice,  tipo, todovalor);}
                        reinic_lista();
        if (SINT) printf("\ndefinicao de lista de VAR" );   }
          ;

conformant_array_schema  : unpacked_conformant_array_schema
                        {
        if (SINT) printf("\nesquema de arranjo"); }
                        ;
```

```
unpacked_conformant_array_schema : ARRAY '[' its  ']' OF idf {
                                   todovalor.c_integer=0;
                                   substitui_simb ($6,  tipo, todovalor);
        if (SINT) printf("arranjo de tipo");  }
                                 | ARRAY '[' its  ']' OF  conformant_array_schema { if (SINT)
printf("\narray de conformant array sch");
        if (SINT) printf("arranjo de arranjo");          }
                                 ;

unpacked_structured_type : array_type  {
        if (SINT) printf("\tipo arranjo");  }
                         | record_type {
        if (SINT) printf("\ntipo registro"); }
                         | set_type    {
        if (SINT) printf("\ntipo conjunto");   }
                         ;

record_type    : RECORD field_list END {
        if (SINT) printf("\ndefinicao de RECORD");  }
             ;

field_list : nada          {
        if (SINT) printf("\nsem lista de registro");  }
           | fixed_part ';' variant_part {
        if (SINT) printf("\nparte fixa ; parte variante");  }
           | fixed_part  {
        if (SINT) printf("\nparte fixa");  }
           | variant_part  ';'              {
        if (SINT) printf("\nparte variante ;");  }
           | variant_part {
        if (SINT) printf("\nparte variante");   }
           ;

fixed_part     : record_section              {
        if (SINT) printf("\nsecao de registro");  }
             | record_section ';' fixed_part {
        if (SINT) printf("\nsecao de registro ; parte fixa");  }
             ;

record_section  : id_list ':' type_denoter      {
        if (SINT) printf("\nsecao de registro : tipo denotado");  }
                ;

variant_part : CASE variant_selector OF vart {
        if (SINT) printf("\nseletor de CASE");  }
             ;

variant_selector : idf ':'  idf {
        if (SINT) printf("\nseletor nome : nome");
                  todovalor.c_integer=0;
                  substitui_simb ($1, tipo, todovalor);
                  todovalor.c_integer=0;
                  substitui_simb ($3,  tipo, todovalor);  }
                 | idf {
                   todovalor.c_integer=0;
                   substitui_simb ($1, tipo, todovalor);
        if (SINT) printf("\nseletor");   }
                 ;
```

```
vart : variant {
        if (SINT) printf("\nvariante");   }
    | variant ';' vart {
        if (SINT) printf("\nvariantes");   }
    ;

variant : case_constant_list ':' '(' field_list ')' {
        if (SINT) printf("\nvariante de lista de CASE");   }
        ;

array_type : ARRAY '[' index ']' OF type_denoter {
        if (SINT) printf("\ndefinicao de arranjo de tipo");  }
          ;

set_type    : SET OF ordinal_type {
        if (SINT) printf("\ndefinicao de conjunto");  }
          ;

ordinal_type   : new_ordinal_type  {
        if (SINT) printf("\nnovo tipo ordinal");  }
              | idf {
                  todovalor.c_integer=0;
                  substitui_simb ($1,  tipo, todovalor);
        if (SINT) printf("\nnome de tipo ordinal");  }
              ;

index     : ordinal_type   {
        if (SINT) printf("tipo ordinal");    }
          | ordinal_type  ',' index {
        if (SINT) printf("tipos ordinais");   }
          ;

its  : index_type_specification {
        if (SINT) printf("\nespecificacao de tipo de indice");  }
    | index_type_specification  ';' its {
        if (SINT) printf("\nespecificacoes de tipos de indices");    }
    ;

index_type_specification : idf  RG idf ':'idf  { todovalor.c_integer=0;
                              substitui_simb ($1,  tipo, todovalor);
                              todovalor.c_integer=0;
                              substitui_simb ($3, tipo, todovalor);
                              todovalor.c_integer=0;
                              substitui_simb ($5, tipo, todovalor);
        if (SINT) printf("\nespecificacao de tipo de indice: idf ..idf :idf");   }
                          ;

compound_statement  : BEGIN  statement_sequence END
                  {
        if (SINT) printf("\ncomponente BEGIN-END");
                  }
                      ;

statement_sequence  : statement    {
        if (SINT) printf("\numa sequencia de declaracoes");   }
                    | statement ';' statement_sequence  {
        if (SINT) printf("\nvarias sequencias de declaracoes");   }
                      ;
```

```
statement : simple_statement  {
        if (SINT) printf("\ndeclaracao simples");   }
          | structured_statement {
        if (SINT) printf("\ndeclaracao estruturada");   }
          ;

structured_statement : compound_statement {
        if (SINT) printf("\n componente com BEGIN-END");   }
                  | conditional_statement {
        if (SINT) printf("\ndeclaracao condicional");   }
                  | repetitive_statement {
        if (SINT) printf("\n declaracao de laco repetitivo");   }
                  | with_statement {
        if (SINT) printf("\n declaracao com WITH");   }
                  ;

simple_statement : nada {
        if (SINT) printf("\nsem declaracao simples");  }
                | assigment_statement  {
                  if ((Isel)|| (I whi)|| (I rof)) {
                    pol_print();
                    executa_pol();
                    init_pol_topo(); }
        if (SINT) printf("\ndeclaracao simples com associacao");    }
                | procedure_statement {
        if (SINT) printf("\ndeclaracao simples com declaracao de procedimento");   }
                ;

procedure_statement : idf  ';'  {
                    todovalor.c_integer=0;
                    substitui_simb ($1, procede, todovalor);
        if (SINT) printf("\nnome de declaracao de procedimento");
                  }
                  | idf '(' ie ')' ';' {
                    todovalor.c_integer=0;
                    substitui_simb ($1,  procede, todovalor);
        if (SINT) printf("\nnome de declaracao de procedimento com parametros");   }
                  ;

assigment_statement : var_access AT expression  { pol_opr(op_at);
        if (SINT) printf("\nassociacao de variavel = expressao"); }
                  | idf  AT expression { pol_var($1);
                                         pol_opr(op_at);
        if (SINT) printf("\nidentficador = expressao"); }
                  ;

expression : simple_expression     {
        if (SINT) printf("\nexpressao simples");   }
          | exprelal              {
        if (SINT) printf("\nexpressao relacional");  }
          ;
```

```
exprelal  : simple_expression GT simple_expression {
                    pol_opr (op_gt);
                    if (sel) if_code();
                    if (whi) while_code();
     if ((rep)||(!whi) ||(!sel)||(!rof)) repeatf_code();
       if (SINT) printf("\nGT"); }
            | simple_expression GE simple_expression {
                    pol_opr (op_ge);
                    if (sel) if_code();
                    if (whi) while_code();
     if ((rep)||(!whi) ||(!sel)||(!rof)) repeatf_code();
       if (SINT) printf("\nGE"); }
            | simple_expression LT simple_expression {
                    pol_opr (op_lt);
                    if (sel) if_code();
                    if (whi) while_code();
     if ((rep)||(!whi) ||(!sel)||(!rof)) repeatf_code();
       if (SINT) printf("\nLT"); }
            | simple_expression LE simple_expression {
                    pol_opr (op_le);
                    if (sel) if_code();
                    if (whi) while_code();
     if ((rep)||(!whi) ||(!sel)||(!rof)) repeatf_code();
       if (SINT) printf("\nLE"); }
            | simple_expression IG simple_expression {
                    pol_opr (op_eq);
                    if (sel) if_code();
                    if (whi) while_code();
     if ((rep)||(!whi) ||(!sel)||(!rof)) repeatf_code();
       if (SINT) printf("\nEQ"); }
            | simple_expression NE simple_expression {
                    pol_opr (op_ne);
                    if (sel) if_code();
                    if (whi) while_code();
     if ((rep)||(!whi) ||(!sel)||(!rof)) repeatf_code();
       if (SINT) printf("\nNE"); }
            ;

term  : factor {
       if (SINT) printf("\nfator");  }
     | mop   {
       if (SINT) printf("\nmultiplos fatores");   }
     ;

mop  : factor '*' factor { pol_opr (op_mul);
       if (SINT) printf("\nfator * fator"); }
     | factor '*' factor mop { pol_opr (op_mul);
       if (SINT) printf("\nfator * fator  multiplos fatores"); }
     | factor '/' factor { pol_opr (op_div);
       if (SINT) printf("\nfator /fator"); }
     | factor '/' factor mop { pol_opr (op_div);
       if (SINT) printf("\nfator / fator  multiplos fatores"); }
     | factor AND factor { pol_opr (op_and);
       if (SINT) printf("\nfator AND fator"); }
     | factor AND factor mop { pol_opr (op_and);
       if (SINT) printf("\nfator AND fator  multiplos fatores"); }
     ;
```

```
simple_expression : '+'  term  '+' term {
                        pol_opr (op_add);
        if (SINT) printf("\n+termo + termo");
                        }
                | '+'  term  '-' term {
                    pol_opr (op_unimenos);
                    pol_opr (op_dif);
        if (SINT) printf("\n+termo - termo"); }
                | '+'  term  OR  term {
                        pol_opr (op_or);
        if (SINT) printf("\n+termo OR termo"); }
                | '-'  term  '+' term {
                    pol_opr (op_unimenos);
                    pol_opr (op_add);
        if (SINT) printf("\n-termo + termo"); }
                | '-'  term  '-' term {
                        pol_opr (op_unimenos);
                        pol_opr (op_dif);
        if (SINT) printf("\n-termo - termo"); }
                | '-'  term  OR  term {
                    pol_opr (op_unimenos);
                    pol_opr (op_or);
        if (SINT) printf("\n-termo OR termo"); }
                |      term  '+' term {
                    pol_opr(op_add);
        if (SINT) printf("\ntermo + termo"); }
                |      term  '-' term {
                    pol_opr (op_dif);
        if (SINT) printf("\ntermo - termo"); }
                |      term  OR  term {
                    pol_opr (op_or);
        if (SINT) printf("\ntermo OR termo"); }
                |      term          {
        if (SINT) printf("\ntermo"); }
                | '+'  term          {
        if (SINT) printf("\n+termo"); }
                | '-'  term          {
                    pol_opr (op_unimenos);
        if (SINT) printf("\n-termo"); }
                ;

factor          : var_access {
        if (SINT) printf("\nvariavel associada: fator");    }
                | C_REAL
                        {  pol_var($1);
        if (SINT) printf("\nvariavel associada: constante real");  }
                | C_INTEGER
                        { pol_var($1);
        if (SINT) printf("\nvariavel associada: constante inteira");    }
                | idf '('  ie  ')' {
        if (SINT) printf("\nnome de fator ( expressoes)"); }
                | set_constructor {
        if (SINT) printf("\nfator de contrucao de conjunto");  }
                | '(' expression ')' {
        if (SINT) printf("\ ( expressao )");  }
                | NOT factor {
        if (SINT) printf("\n NOT fator");  }
                ;
```

```
mesignator : nada  {
        if (SINT) printf("\nsem membro"); }
          | member_designator {
        if (SINT) printf("\ndesignador de membro");  }
          | member_designator  ',' mesignator {
        if (SINT) printf("\ndesignador de membros ,");  }
          ;

set_constructor : '[' mesignator  ']' ';' {
        if (SINT) printf("\construtor de conjunto");  }
                ;

member_designator   : expression {
        if (SINT) printf("\designador de membro expressao");  }
                  | expression  RG expression {
        if (SINT) printf("\nmembro expressao..expressao");   }
                  ;

var_access : idf {
        if (SINT) printf("\nVariavel(L): %s", nome_simbolo($1));
                  pol_var ($1); }
          | component_variable {
        if (SINT) printf("\ncomponente de variavel");  }
          | var_access '^' {
        if (SINT) printf("\nacesso a variavel^");   }
          ;

component_variable  : var_access '[' ie  ']' {
        if (SINT) printf("tipo de componente -> indice");  }
                  | var_access '.' idf {
                    pol_var($3);
                    todovalor.c_integer=0;
                    substitui_simb ($3, tipo, todovalor);
        if (SINT) printf("\ntipo de componente -> variavel.nome");  }
                  ;

ie  : expression {
        if (SINT) printf("\nexpressao");  }
    | expression  ',' ie {
        if (SINT) printf("\nexpressoes");   }
    ;

conditional_statement   : if_statement {
        if (SINT) printf("condicao if");   }
                        | case_statement {
        if (SINT) printf("condicao case");   }
                        ;

if_statement   : IF exprelal THEN statement  {
                                iffim_code();
        if (SINT) printf("\nIF sem ELSE");  }
              | IF exprelal THEN statement ';' else_part  {
                                iffim_code();
        if (SINT) printf("\nIF com ELSE");  }
              ;
```

```
else_part   : ELSE statement  {
        if (SINT) printf("\nparte de ELSE");  }
            ;

case_statement : CASE exprelal OF case_list  END  {
        if (SINT) printf("\nCASE OF");  }
              ;

case_list          : case_list_element ';' {
        if (SINT) printf("elemento CASE");   }
                    | case_list_element ';' case_list {
        if (SINT) printf("lista CASE");   }
                    ;

case_list_element   : case_constant_list ':' statement  {
        if (SINT) printf("\ncelemento de lista CASE");  }
                    ;

case_constant_list  : constant   {
        if (SINT) printf("\nlista de constantes CASE");  }
                    | constant ',' case_constant_list {
        if (SINT) printf("\nlista de constantes CASE ,");  }
                    ;

repetitive_statement : repeat_statement {
        if (SINT) printf("\ndeclaracao de REPEAT");  }
                    | while_statement  {
        if (SINT) printf("\ndeclaracao WHILE");    }
                    | for_statement    {
        if (SINT) printf("\ndeclaracao FOR");    }
                    ;

repeat_statement : REPEAT statement_sequence UNTIL exprelal {
        if (SINT) printf("\ndeclaracao de REPEAT - UNTIL");  }
                ;

while_statement : WHILE exprelal DO statement  {
                    whi_code();
        if (SINT) printf("\ndeclaracao de WHILE - DO");
                    }
                ;

for_statement : FOR for_exp TO exp_for DO statement {
                    forf_code();
        if (SINT) printf("\nFOR com TO");  }
            | FOR for_exp  DOWNTO  exp_for DO statement {
                    forf_code();
        if (SINT) printf("\nFOR com DOWNTO");   }
            ;

for_exp  : idf AT expression {
                    pol_var($1);
                    pol_opr(op_at);
                    for_code();
                      }

exp_for   : expression { for_cond_code(); }
```

```
with_statement : WITH record_variable_list DO statement ';' {
        if (SINT) printf("\ndeclaracao de WITH DO"); }
            ;

record_variable_list : var_access {
        if (SINT) printf("\nvariavel de registro"); }
                    | var_access ',' record_variable_list {
        if (SINT) printf("\nlista de variaveis de registro"); }
                    ;

%%
/*
    FIM DA GRAMATICA
*/
```

Apêndice B

B.1 - Conceitos de BISON

B.1.1 - Linguagens e Gramáticas de Livre-Contexto

Para que o *BISON* possa analisar uma linguagem, ela deve ser definida e descrita por uma gramática de livre-contexto. Isto significa que é especificado um ou mais agrupamento sintático, e devem ser fornecidas regras para construí-los a partir de suas partes componentes. Por exemplo, em linguagem C, um tipo de agrupamento é chamado uma **expressão**. Uma regra para fazer uma **expressão** poderia ser: "Uma **expressão** pode ser feita de um **sinal menos** e outra **expressão**". Outra seria, "Uma **expressão** pode ser um **inteiro**". Como pode servisto, as regras são frequentemente recursivas, mas deve existir pelo menos uma regra que saia fora da recursão.

O sistema formal mais comum para apresentar estas regras para os usuários lerem é **Backus Naur Form (BNF)**, que foi desenvolvido para especificar a linguagem Algol 60. Qualquer gramática expressa em **BNF** é uma gramática de livre-contexto. A entrada para o **BISON** é essencialmente uma **BNF** que pode ser aceita pela máquina.

Nas regras gramaticais formais para uma linguagem, cada tipo de unidade sintática ou agrupamento é chamada por um símbolo. Aqueles que são construídos através de construções de agrupamentos menores, de acordo com as regras gramaticais são chamados **símbolos não-terminais**; aqueles que não podem ser sub-divididos são chamados **símbolos terminais** ou tipo *token*. Chamamos uma entrada correspondente a um único símbolo terminal de um ***token***, e uma entrada correspondente a um único símbolo terminal de um **agrupamento**.

Podemos usar a linguagem C como um exemplo do que significam os símbolos, terminais e não-terminais. Os *tokens* da linguagem C são identificadores, constantes (numéricas e caracter), e as várias palavras-chave, operadores aritméticos e sinais de pontuação. Então os símbolos terminais de uma gramática para linguagem C incluem **identificador**, **digito**, **caracter literal**, mais um símbolo para cada palavra chave, operador e sinal de pontuação: ***if***, ***return***, ***const***, ***static***, ***int***, ***char***, **sinal de mais**, **abre-colchete**, **fecha-colchete**, **vírgula** e muito mais. (Estes *tokens* podem ser sub-divididos em caracteres, mas isto é um problema de lexicografia, não de gramática.)

Aqui está uma função em linguagem C simples sub-dividida em *tokens*:

```
        int                 /* palavra chave 'int'                    */
        square (x)          /* identifcador, abre parenteses
                                   identificador, fecha parenteses    */
        int x;          /* palavra chave 'int', identificador,
                                   ponto e virgula                    */
{                           /* abre chaves                            */
        return x * x;   /* palavra chave 'return', identificador
                                   asteristico, identificador, ponto e
                                   vírgula                            */
}                           /* fecha chaves e                         */
```

Os agrupamentos sintáticos da linguagem C incluem a expressão, o comando, a declaração e a definição da função. Estes são representados na gramática de C por símbolos não-terminais "expressão", "comando", "declaração" e "definição de função". A gramática completa usa dúzias de construções de linguagem adicionais, cada uma com seus próprios símbolos não-terminais, para expressar os significados destes quatro. O exemplo acima é uma função de definição; ela contém uma declaração, e um comando. No comando, cada 'x' é uma expressão, assim como 'x * x'.

Cada símbolo não-terminal deve ter regras gramaticais mostrando como ele é construído a partir de construções mais simples. por exemplo, um tipo de comando C é o comando **return**; este seria descrito com uma regra gramatical que lê informalmente como segue:

Um 'comando' pode ser feito com um tecla 'return', uma 'expressão' e um 'ponto e vírgula'.

Existiriam muitas outras regras para 'comando', uma para cada tipo de comando em C.

Um símbolo não-terminal pode ser diferenciado como um símbolo especial que define uma declaração completa na linguagem. Ele é chamado "***start symbol***" **(ou símbolo inicial)**. Em um compilador, isto significa um programa de entrada completo. Na linguagem C, o símbolo não-terminal "sequência de definições e declarações" representa este papel.

Por exemplo, **1 + 2** é uma expressão em linguagem C válida - uma parte válida de um program emlinguagem C - mas não é válida como um programa em linguagem C inteiro. Na gramática livre de contexto da linguagem C, isto vem do fato de que **expressão** não é o

símbolo inicial.

O analisador **BISON** lê uma sequência de *tokens* como entrada, e grupos de *tokens* usando as regras de gramática. Se a entrada é válida, o resultado final é que toda a sequência de *tokens* reduz-se a um único agrupamento cujo símbolo é o símbolo inicial da gramática. Se usamos uma gramática para linguagem C, toda a entrada deve ser uma sequência de definições e declarações. Caso contrário, o analisador relaciona um erro sintático.

B.1.2 - Das Regras Formais para a Entrada aceita Bison

Uma gramática formal é uma construção matemática. Para definir a linguagem para o **BISON**, deve-se escrever um arquivo expressando a gramática em sintaxe **BISON**: um arquivo de gramática **BISON**. Ver o ítem B.2.

Um símbolo não-terminal na gramática formal é representada na entrada **BISON** como um identificador, do mesmo modo que um identificador em linguagem C. Por convenção, estes identificadores devem ser representados em letras minúsculas, tais como **expr, stmt** ou **declaration**.

A representação **BISON** para um símbolo não-terminal é também chamado um tipo **token**. Por convenção, estes identificadores devem estar em letras maiúsculas, para distingui-los dos símbolos não-terminais: por exemplo, **INTEGER, IDENTIFIER, IF** ou **RETURN**. Um símbolo terminal que existe para um determinado caracterer de teclado na linguagem deverá ser especificado depois do caracter ser convertido para letras maiúsculas. O símbolo terminal **error** é reservado para recuperação de erros.

Um símbolo terminal também pode ser representado como um caracter literal, exatamente como um caracter de constante em linguagem C. Isto deve ser feito sempre que um *token* é apenas um caracter isolado (parênteses, sinal de mais, etc.): use esse mesmo caracter em um literal como o símbolo terminal para esse *token*.

As regras de gramática também tem uma expressão na sintaxe **BISON**. Por exemplo, será mostrada a regra Bison para um comando **return** em linguagem C. O ponto e vírgula entre aspas é um *token* caracter literal , representando parte da sintaxe C para o comando, o ponto e vírgula sozinho, e a vírgula. são as pontuações **BISON** usadas em todas as regras.

```
stmt:  RETURN expr ';'
       ;
```

Ver a seção B.2.3.

B.1.3 - Valores Semânticos

Uma gramática formal seleciona *tokens* apenas pela sua classificações: por exemplo, se uma regra menciona o símbolo terminal **constante inteira**, significa que qualquer constante inteira é gramaticamente válida nessa posição. O valor preciso da constante é irrelevante para como analisar a entrada: se 'x + 4' é gramaticalmente correta, então 'x + 1' ou 'x + 3989' é igualmente correta.

Mas o valor preciso é muito importante para o significado da da entrada depois que ela foi analisada. Um compilador seria inútil se não conseguisse distinguir entre 4, 1 e 3989 como constantes no programa. Assim, cada **token** na gramática **BISON** tem um tipo *token* e um valor **semântico**. Ver seção B.2.5.

O tipo *token* é um símbolo terminal definido na gramática, tais como **INTEGER-CONSTANT, IDENTIFIER,** ou ',' . Ele diz tudo que precisa-se saber para decidir onde o *token* pode ser válido, e como agupá-lo com outros *tokens*. As regras gramaticais não sabem mais nada sobre os *tokens* além de seus tipos.

O valor semântico tem todas as outras informações sobre o significado do *token*, tais como o valor de um inteiro, ou o nome de um identificador. (Um *token* como ',' que é apenas pontuação não precisa ter um valor semântico.)

Por exemplo, um *token* de entrada poderia ser classificado como um tipo *token* **INTEGER** e ter o valor semântico 4. Outro "token" de entrada poderia ter o mesmo tipo *token* **INTEGER**, com o valor 3989. Quando uma regra de gramática diz que é permitido um **INTEIRO**, qualquer um destes *tokens* é aceitável, porque cada um é um **INTEIRO**. Quando o analisador aceita o *token*, ele guarda o seu valor semântico.

Cada agrupamento pode também ter um valor semântico, assim como seu símbolo não-terminal. Por exemplo, em uma calculadora, uma expressão tem tipicamente um valor semântico que é um número. Em um compilador para uma linguagem de programação, uma expressão tem tipicamente um valor semântico que é uma estrutura de árvore descrevendo o significado da expressão.

B.1.4 - Ações Semânticas

Para ter utilidade, um programa deve fazer mais do que analisar a entrada; ele deve também produzir uma saída baseada na entrada. Em uma gramática **BISON**, uma regra gramatical pode ter uma ação feita de comandos em linguagem C. A cada vez que o analisador reconhece uma combinaçao para essa regra, a ação é executada. Ver a seção B.2.5.3.

A maior parte do tempo, o propósito de uma ação é computar o valor semântico da construção global a partir dos valores semânticos de suas partes. Por exemplo, suponha que temos uma regra que diz que uma expressão pode ser a soma de duas expressões. Quando o analisador reconhece essa soma, cada uma das sub-expressões tem uma valor semântico que descreve como ela foi construída. A ação para esta regra deveria criar uma espécie de valor similar para a expressão maior que acaba de ser reconhecida.

Por exemplo, aqui está uma regra que diz que uma expressão pode ser a soma de duas sub-expressões:

```
expr:  expr '+' expr   {$$ = $1 + $3;  }
       ;
```

A ação diz como produzir o valor semântico da expressão de soma dos valores das duas sub-expressões.

B.2 - Arquivos de Gramática BISON

O **BISON** manipula arquivos de entrada especificadas em gramáticas livres de contexto, produzindo como saida funções em linguagem C.

Os arquivos de entrada para gramática **BISON**, por convenção, tem em seu nome a extensão '.y'.

B.2.1 - Síntese da Gramática BISON

O arquivo da gramática **BISON** é dividido em quatro secções, definidas abaixo com os delimitadores apropriados:

```
%{
  Declarações em C
%}
  Declarações do BISON
%%
  Regras Gramaticais
%%
  Códigos Adicionais em C
```

Comentários incluidos entre '/* ... */' podem aparecer em qualquer uma dessas secções.

B.2.1.1 - Secção das Declarações em C

As declarações em linguagem C contém macro definições, declarações das funções e variáveis que serão usadas nas ações das regras gramaticais. Estes dados são copiados para início do arquivo do analisador de modo que eles precedem a definição do **yylex**. Pode ser usado o comando "*#include*" para associação com declarações de outros arquivos *header*. Se não forem necessárias declarações em linguagem C os delimitadores %{ e %} podem ser omitidos.

B.2.1.2 - Secção das Declarações BISON

Esta secção contém declarações que definem símbolos terminais e não-terminais, especifica a precedência de operadores e outras atividades. Em gramáticas simples pode ser necessária nenhuma declaração.

B.2.1.3 - Secção das Regras Gramaticais

Esta secção contém apenas mais regras da gramática **BISON**. Deve haver ao menos uma regra gramatical e não se pode omitir o simbolo %% inicial.

B.2.1.4 - Secção de Código C adicional

Esta secção é copiada palavra por palavra no final do arquivo do analisador, como são copiadas as declarações C no começo. Este é o local apropriado para colocar as funções que podem ser necessárias para o funcionamento do programa mas não fazem parte do **yylex**. Neste espaço pode ser colocado, por exemplo o **yyerror**, e outras funções que fazem interface com os demais modulos do programa.

Se não houver nenhuma declaração a ser feita pode ser omitidos os separadores %%.

O analisador do **BISON** contém variaveis estáticas que começam por **YY**, tendo ainda macros com **YY** no nome. Deve ser evitado pelo usuário nomes de variaveis e macros com essas letras.

B.2.2 - Símbolos, terminais e não-terminais

Símbolos na gramática **BISON** representam as classificações gramáticais da linguagem.

Um símbolo terminal (também conhecido como *token*) representa uma classe sintaticamente equivalente. O uso do símbolo na gramática significa que o *token* dessa classe é permitido. O símbolo é representado na gramática **BISON** por um código numérico conhecido, e a função **yylex** retorna um código do tipo do *token* que indica sua especie que esta sendo lida. Não é necessário que o usuário conheça este valor, bastando-lhe o uso do símbolo.

Um símbolo não-terminal é usado para uma classe de grupos sintaticamente equivalentes. O nome do símbolo é escrito nas regras gramaticais. Por convenção devem ser escritos em letras minúsculas.

Os nomes de símbolos podem conter letras, números (não devendo apenas começar por número), pontos, ou *underscore*. Pontos tem sentido apenas em não-terminais.

Existem duas formas de escrevermos um símbolo terminal na gramática:

1- Como um identificador, exatamente como em linguagem C. Por convenção é escrito em letra maiúscula. Cada um desses identificadores deve ser definido nas declarações **BISON** como ***%token***.

2- Como um *token* do tipo caracter (ou *token* literal) é escrito na gramática usando a mesma sintaxe em C para constantes, por exemplo '+' é um caracter do tipo *token* literal. O *token* literal não precisa ser declarado, devendo apenas ser especificada sua semântica, associatividade e precedência.

Por convenção um tipo *token* literal é usado somente para representar um *token* que consiste em um caracter particular. Então o literal *token* '+' é usado para representar o caracter '+' como um *token*. Nada garante esta conveção, mas esta deve ser respeitada

pelo usuário, para que o programa não confunda outras letras.

Qualquer sequência usada em C pode ser usada em **BISON** satisfatoriamente, mas não deve ser usado o caracter nulo ASCII, pois este é o código retornado pelas funções ddo **yylex**.

O valor retornado pelo **yylex** é sempre um símbolo terminal (ou 0 para fim de entrada), sendo que um codigo numérico para o caracter é seu código ASCII.

B.2.3 - Sintaxe das regras gramaticais

A sintaxe **BISON** seguem a forma geral:

Resultados : Componentes...

Resultados são símbolos não terminais que essa regra descreve,

Componentes são terminais e não-terminais que são colocados juntos na regra.

Por exemplo:

exp : exp '+' exp

mostra que dois grupos do tipo **exp** com um '+' entre ele, que podem ser combinados em um grupo maior do tipo **exp**.

Os espaços em branco nas regras são separadores de símbolos, podendo ser adicionado um espaço extra se necessário.

Distribuido entre os componentes encontram-se as ações que determinam a semântica das regras. As ações são representadas entre parentese como segue:

{ expressões em linguagem C }

Regras múltiplas são separadas por uma barra vertical '|', sendo consideradas regras distintas, como por exemplo:

```
Resultado   :   componente_regra1
            | componente_regra2
```

Se o componente da regra for vazio, isso significa que o resultado podera ser

igualmente uma cadeia vazia. Por exemplo como é definido o separador 'virgula' por zero ou mais grupos **exp**:

```
expseq      :     /* vazio */
            |expeseq1
            ;
expeseq1    :     exp
            | expseq1 ',' exp
            ;
```

É usual escrever o comentário /* vazio */ nas regras sem componentes.

B.2.4 - Regras Recursivas

A regra é chamada recursiva quando em **Resultado** aparece um não terminal igual ao lado direito. Praticamente todas as gramáticas **BISON** necessitam usar a recursão, porque essa é a única forma de definir uma sequência de uma série de coisas. Considerando a definição do separador '**virgula**' como uma ou mais expressões:

```
expseq1 :   exp
            | expseq1 ',' exp
            ;
```

Como o uso recursivo de **expseq1**, que é um simbolo a esquerda no lado direito da regra, nos chamamos de recursão à esquerda. Em contrapartida podemriamos ter a mesma construção definida como recursão à direita como segue:

```
expeseq1    : exp
            | exp ',' expseq1
            ;
```

Qualquer tipo de sequência pode ser definida usando-se recursão à direita ou à esquerda, mas deve-se optar pela à esquerda, por que isto pode analisar uma sequência de quaisquer números de elementos com varios espaçamentos. Recursão à direita usa até o fim o espaço da pilha do **BISON** na proporção do números de elementos na sequência, por que todos os elementos devem ser deslocados para a pilha, antes que a regra possa ser aplicada mesmo uma vez.

A recursão indireta ou mutua ocorre quando o resultado da regra não aparece

diretamente no lado direito da regra, mas aparece na regra para outros não-terminais os quais aparecem do lado direito da regra. Por exemplo:

```
expr        :   primary
            | primary '+' primary
            ;
primary     :   constant
            | '(' expr ')'
            ;
```

define dois não-terminais mutuamente recursivos, desde que cada um se refere ao outro.

B.2.5 - A semântica da linguagem

As regras gramaticais para a linguagem determinam somente a sintaxe. A semântica é determinada pelos valores semânticos associados com varios *tokens* e grupos, e pelas ações tomadas quando varios agrupamentos são reconhecidos.

Por exemplo, uma calculadora executa corretamente um cálculo porque o valor associado com cada expressão é o próprio número, a soma é feita adequadamente por que a ação para o agrupamento 'x + y'é a adição dos número associados com x e y.

B.2.5.1 - Tipos de Dados dos Valores Semânticos

Em um programa simples pode ser suficiente o uso do mesmo tipo de dado para valores semânticos de todos construtores da linguagem. Isto pode ser verdade na construção de uma calculadora que use notação polonesa reversa (RPN), ou mesmo uma simples calcudora infixa, como pode ser visto em exemplos comumente encontrados na literatura a respeito de geradores de compiladores.

O padrão (*default*) do **BISON** é o uso do tipo **int** para valores semânticos. Para especificar outros tipos , é definido **YYSTYPE** como uma macro, tal como:

```
#define YYSTYPE double
```

Esta definição de macro deve estar entre secção das declarações da gramática (ver secção B.2.1).

B.2.5.2 - Mais de um tipo de Dado para Valores Semânticos

Na maioria dos programas, pode ser necessário diferentes tipos de dados para diferentes tipos de *tokens* e agrupamentos. Por exemplo, uma constante numérica pode necessitar de um tipo **int** ou **long**, enquanto que uma cadeia de caracteres (*string*) necessita do tipo **char** *, e um indentificador pode precisar de um ponteiro para referenciar a tabela de símbolos.

Para usar mais doque um tipo de dado para valores semânticos em um analisador, o **BISON** requer duas ações:

. Especificar a coleção completa dos possiveis tipos de dados, com a declaração **%union** do **BISON** (secção B.6.3).

. Escolher um dos tipos de dados para cada símbolo (terminal ou não-terminal) para cada valor semântico que será usado. Isto é feito para *tokens* com a declaração **%token** do **BISON** (ver secção B.2.6.1), e para agrupamentos com a declaração **%type** do **BISON** (ver secção B.2.6.4).

B.2.5.3 - Ações

Uma ação acompanha a regra sintática e contém código em linguagem C para ser executado cada vez que uma parte da regra é reconhecida. A tarefa da maioria das ações é computar um valor semântico para um agrupamento construido pela regra através dos valores semânticos associados com *tokens* ou agrupamentos menores.

Uma ação consiste de uma declaração em linguagem C limitado pelo caracter chaves, bastante parecido com declarações em limguagem C. Isto pode ser colocado em qualquer posição na regra; sendo executado nesta posição. A maioria das regra tem apenas uma ação no final de cada regra, seguindo todos os componentes. Ações de meia regra são truques e são usadas apenas em casos especiais. (ver secção B.2.5.5).

O código em linguagem C em uma ação pode referir-se a valores semânticos unidos pela regra com a construção **$n**, com padrões para o valor do enéssimo componente. O valor A semântico para o grupo é iniciado pelo símbulo **$$** (O **BISON** traduz ambas as construções em um arranjo de elementos de referência quando copia as ações para o arquivo do analisador).

Eis um exemplo típico

```
exp     :     ...
        | exp '+' exp
        { $$ = $1 + $3; }
```

Esta regra constroe um agrupamento **exp** através de dois grupos menores **exp** conectados por um símbolo de soma. Nesta ação, **$1** e **$3** refere-se a valores semânticos de dois componentes de grupo **exp**, que são o primeiro e terceiro simbolos do lado direito da regra. A soma é armazenada em **$$**, sendo que este torna-se o valor semântico para a expressão de adição, reconhecido pela regra. Se for necessário um uso semântico para o *token* '+', este pode ser referenciado por **$2**.

O símbolo **$n** com n zero ou negativo é permitido por referência aos *tokens* e grupos em pilha antes destes unidos pela regra corrente. Isto é uma prática de grande risco, e o uso com segurança deve-se ter certeza docontexto em que a regra é aplicada. Aqui esta um caso em que você pode usar com segurança:

```
foo   : expr bar    '+' expr { ... }
      | expr bar    '+' expr { ... }
      ;
bar   : /* empty */
      | previous_expr = $0; }
      ;
```

Enquanto **bar** é usado apenas da forma acima mostrada **$0** sempre se refere à **expr** que precede **bar** na definição de **foo**.

B.2.5.4 - Tipos de dados de valores em ações

Se foi escolhido um tipo de dados simples para os valores semânticos, as construções **$$** e **$n** sempre devem ter este tipo de dado.

Se foi usado **%union** para especificar uma variedade de tipos de dados, então deve ser declarada a escolha entre os tipos para cada símbolo terminal e não-terminal que tem um valor semântico. Então cada vez que usado **$$** ou **$n**, este tipo de dado é determinado por cada símbolo referido a regra. Por exemplo:

```
exp    : ...
       | exp '+' exp
       { $$ = $1 + $3; }
```

$3 e $$ refere-se a exemplos de exp, tendo apenas declarado o tipo de dado para o símbolo não-terminal exp. Se $2 for usado, deve ser declarado o tipo de dado para o símbolo terminal '+', qualquer tipo que possa ser.

Alternativamente, pode ser especificado o tipo de dado quando for feita a referência ao valor, por inserção de '<type>' depois de '$' no início da referência. Por exemplo como mostrado a seguir:

```
%union {
        int itype;
        double dtype;
    }
```

então pode ser escrito $<itype>1 para referir a primeira sub-unidade da regra como do tipo integer, ou $<itype>1 para referir a este como do tipo *double*.

B.2.5.5 - Açoes de meia de regra.

Ocasionalmente, é útil colocar uma ação no meio de uma regra. Estas ações são escritas exatamente como as ações de fim-de-regra usuais, mas são executadas antes mesmo que o analisador reconheça os componentes seguintes.

Uma ação de meio-de-regra pode referir-se aos componentes que o precedem pelo uso de $n, mas ele não pode referir-se aos componentes subsequentes, porque ele é executado antes que seja analisado.

A própria ação de meio de regra conta como um dos componentes da regra. Isto faz diferença quando existe outra ação posterior na mesma regra (e usualmente existe outra no final): deve-se considerar as ações em conjunto com os símbolos quando resolve-se que número n usar com $n.

As ações de meio de regra podem também ter um valor semântico. Este pode ser ajustado dentro da ação por uma atribuição para $$, e pode ser referida por ações posteriores usando $n. Uma vez que não existe nenhum símbolo para nomear a ação, não existe nenhuma maneira para declarar um tipo de dados para o valor adiantado, deve-se

então usar a construção '$<...>' para especificar um tipo de dados a cada vez que se referir a este valor.

Aqui está um exemplo de um compilador hipotético, manuseanndo um comando *let*, que aparece como **let (variable) statement**, e serve para criar um variável denominada **variable** temporariamente, druante a duração da declaração. Para analisar esta construção, deve-se colocar **variable** na tabela de símbolos enquanto a declaração é analisada, e posteriormente removê-lo. Isto é feito do seguinte modo:

```
stmt:   LET '(' var ')'
      { $<context>$ = push_context();
        declare_variable ($3);}
        stmt { $$ = $6;
        pop_context ($<context>5);}
```

Assim que ' let (variable)' foi reconhecida, a primeira ação é run. Ela salva uma cópia do contexto semântico corrente (a lista de variáveis acessíveis) como seu valor semântico, usando a alternativa context na união de tipo de dados. Então ela chama declare-variable para adicionar a nova variável à lista. Uma vez que a primeira ação está terminada, o comando delimitado stmt pode ser parseado. Observe que a ação de meio de regra é o componente número 5, então o 'stmt' é o componente número 6.

Depois que o comamdo delimitado é analisado, seu valor semântico torna-se o valor da declaração **let** inteiro. Então o valor semântico da ação anterior é usado para recuperar a lista anterior de variáveis. Isto remove a **let-variable** temporária da lista, de modo que ela não parece existir quando o resto do programa é analisado.

Tomar uma ação antes que uma regra seja completamente reconhecida, com frequência leva a conflitos, uma vez que o analisador deve ser submetido a outro analisador a fim de executar a ação. Por exemplo, as duas regras seguintes, sem ações de meio de regra, podem coexistir em um analisador operante porque o outro analisador pode trocar o *token* **chave aberta** e verificar o que segue antes de decidir se existe uma declaração ou não:

```
compound   : '{' declarations statements '}'
           | '{' statements '}'
           ;
```

SERVIÇO DE BIBLIOTECA E INFORMAÇÃO - IFQSC
FÍSICA

Mas quando acrescenta-se uma ação de meio de regra, como segue, a regra torna-se não funcional:

```
compound:  { prepare_for_local_variables(); }
           '{' declarations statements '}'
              | '{' statements '}'
              ;
```

Agora o analisador é forçado a decidir se executa a ação de meio de regra quando ele não lê nada mais além de um símbolo **chave aberta**. Em outras palavras, ele deve decidir usar uma regra ou outra, sem informações suficientes para fazê-lo corretamente. (O token colchete aberto é chamado o próximo *token* desta vez, uma vez que o analisador ainda está decidindo o que fazer com ele.

Poderia-se pensar em solucionar este problema colocando ações idênticas nas duas regras, como esta:

```
compound:  { prepare_for_local_variables(); }
           '{' declarations statements '}'
              | { prepare_for_local_variables(); }
              '{' statements '}'
              ;
```

Mas isto não funciona, porque o **BISON** não percebe que as duas ações são idênticas. (O **BISON** nunca tenta compreender o código C em uma ação.)

Se a gramática é tal que uma declaração pode ser diferenciada de um comando pelo primeiro *token* (que é verdade em linguagem C), então uma solução que realmente funciona, é colocar a ação depois do símbolo **chave aberta**, como a seguir:

```
compound:  '{'{ prepare_for_local_variables(); }
            declarations statements '}'
            | '{' statements '}'
            ;
```

Agora, o primeiro *token* da declaração ou comando seguinte, que poderia em qualquer caso dizer ao **BISON** que regra usar, ainda pode fazê-lo.

Outra solução é encobrir a ação dentro de um símbolo não terminal que serve como

uma sub-rotina:

```
subroutine:  /* empty*/
             { prepare_for_local_variables(); }
             ;
compound:    subroutine
             '{' declarations statements '}'
             | subroutine
             '{' statements '}'
             ;
```

Agora, o **BISON** pode executar a ação na regra para **subroutine**, sem decidir que regra para **compound** ele eventualmente usará. Observe que a ação está agora no final desta regra. Qualquer ação de meio de regra pode ser convertida em uma ação de fim de regra desta maneira, e isto é o que o **BISON** na verdade faz para implementar as ações de meio de regra.

B.2.6 - Declarações em BISON

A secção de declarações **BISON** da gramática **BISON** define os símbolos usados na formulação da gramática e dos tipos de dados dos valores semânticos. Ver secção B.2.2.

Todos os nomes dos tipo *tokens* (mas não carácter simples do tipo literal, como '+' e '*') devem ser declarados. Os Símbolos não-terminais devem ser declarados se for necessária uma especificação com tipo de dados para o uso do valor semântico. Ver secção B.2.5.2.

A primeira regra do arquivo, por convenção também especifica o símbolo inicial (*start*). Se for necessário algum outro símbolo como inicial, deve ser explicitamente declarado. Ver secção *1.1

B.2.6.1 - Declaração de nome do tipo *token*

A forma básica de declaracção do nome do tipo *token* (símbolo terminal) é como segue:

%*token* nome

O **BISON** irá converter esta declaração em uma diretiva ***#define*** no arquivo *parser*,

que apenas a função **yylex** (se esta estiver no arquivo) pode usar **nome** para suportar este código do tipo *token*.

Alternativamente pode ser usado ***%left***, ***%right***, ou ***%nonassoc*** ao invez de ***%token***, no caso de ser necessário especificar a precêdencia. Ver secção B.2.6.2.

Pode ser explicitado uma especificação de um código numérico para o tipo *token* por adição de um inteiro no campo imediatamente seguinte ao nome do *token*:

%token **NUM 300**

O melhor a se fazer, porém, é deixar o **BISON** fazer a escolha dos códigos numéricos para todos os tipos *tokens*. O **BISON** irá fazer automaticamente a seleção de código que não terá conflito com outros códigos ou caracteres ASCII.

No caso em que o tipo pilha é uma união, deve ser acrescentada a declaração de ***%token*** ou outras declarações para incluir o tipo de dado alternativo delimitado por "<" e ">". Ver secção B.2.5.2. Por exemplo:

```
%union { /* define a pilha de tipo */
            double val;
            symrec *tptr;
        }
        % token <val> NUM        /* define o tipo token e seu tipo */
```

B.2.6.2 - Declaração do Operador de Precedência

É usado de ***%left***, ***%right***, ou ***%nonassoc*** para declarar um *token* e especificar a precedência e associatividade. Isto é chamado declarações de precêdencia. Ver secção 5.2.3, para informações gerais sobre o operador precedencia.

A sintaxe da declaração de precedência é a mesma que a ***%token***: ou **%left símbolos**

ou **%left <val> símbolos**

E realmente algumas destas declarações servem ao propósito de ***%token***. Mas em adição, deve ser especificado a associatividade e precedência relativa para todos os símbolos:

. A associatividade de um operador **op** determina como uso repetido do operador esta aninhado: se 'x op y op z' é analisado por agrupamento de x com y primeiro, ou por agrupamento de y com z primeiro. A declaração **%left** especifica a associatividade à esquerda (agrupando x com y primeiro), e **%right** determina a associatividade à direita (agrupando y e z primeiro). A declaração **%nonassoc** especifica que não há associatividade, o que significa que 'x op y op z' é considerado um erro de sintaxe.

. A precedência de um operador determina como este está enlaçado com outros operadores. Todos os *tokens* relacionados em uma declaração simples de precedência tem a mesma igual, e serão aninhados juntos, de acordo com suas associatividades. Quando dois *tokens* declarados em diferentes precedências são asssociados, o que é declarado depois tem precedência mais alta e é agrupado primeiro.

B.2.6.3 - Declaração de uma Coleção de Tipos de Valores

A declaração **%union** especifica uma coleção inteira de possibilidades de tipos de dados para valores semânticos. A palavra chave **%union** é seguido por um par de chaves contendo o mesmo tipo de comando que vai dentro do comando **union** na linguagem C. Por exemplo:

```
%union {
        double val;
        symrec *tptr;
    }
```

Isto significa que existem dois tipos alternativos que são **double** e **symrec ***. Os nomes das variáveis **val** e **tptr** são os nomes usados na declaração **%token** e **%type** para escolher um dos tipos para um terminal ou não-terminal (secção B.6.4).

Nota-se que, diferente da linguagem C, não se deve colocar o símbolo de ponto e virgula após o símbolo de chaves que fecha o comando.

B.2.6.4 - Declaração de Tipos de Valores de Símbolos não Terminais

Quando é usado a definição **%union** para especificar multiplos tipos de dados, deve ser declarado o valor do tipo para cada símbolo não-terminal para o qual o valor devera ser usado. Isto é feito com a declaração **%type**, tal como:

%type <tipo> não-terminal

Onde **não-terminal** é o nome do símbolo não terminal, e **tipo** é o nome dado em **%union** para a alternativa desejada (secção B.2.6.3). Pode ser dado um número para o símbolo não-terminal na mesma declaração **%type**, se há o mesmo valor do tipo. Deve ser usado um espaço para separar os nomes do símbolos.

B.2.6.5 - Prevenção (*Warnings*) sobre conflitos

Normalmente o **BISON** avisa se existe algum conflito na gramática (secção*5.2), mas a maior parte das gramáticas reais possuem conflitos **shift/reduce** perigosos que são resolvidos de uma maneira previsivel e são dificieis de serem eliminados. É desejavel suprimir o aviso sobre estes conflitos, a menos que o número de conflitos se altere. Isto pode ser feito com o uso da declaração **%expect**. Esta declaração tem o seguinte aspecto:

%expect n

Onde **n** é um inteiro decimal. A declaração diz que não deve existir nenhum aviso se existirem conflitos **shiht/reduce**. O aviso usual é dado se existirem mais ou menos conflitos, ou se existir algum conflito **reduce/reduce**.

Em geral, o uso da declaração **%expect** envolve os seguintes passos:

. Deve ser compilada a gramática sem **%expect**. O uso da opção '-v' irá fornecer uma lista completa de onde os conflitos ocorrem. O **BISON** imprimirá também o número de conflitos que ocorrem.

. Deve ser verificado cada um dos conflitos para certificar-se que a resolução padrão (*default*) do **BISON** é oque realmente se deseja. Senão deve-se reescrever a gramática e retornar ao inicio.

. Deve ser adicionada uma declaração **%expect**, copiando o número **n** apartir do número que o **BISON** imprimir.

Agora o **BISON** não irá de causar problemas sobre os conflitos checados, mas voltará a avisar sobre os conflitos se alterações na gramática resultarem em conflitos adicionais.

B.2.6.6 - Declaração do símbolo inicial da gramática

O **BISON** assume, por conveção, que o símbolo inicial para a gramática é o primeiro

não terminal especificado na secção de especificação de gramática. O programador pode não levar em conta esta restrição e usar a declaração *%start* como segue:

%start símbolo

B.2.6.7 - Requesitando um *Parser* Puro (Reentrante)

Um programa reentrante é um programa que não se altera no curso da execução; em outras palavras, consiste inteiramente de código puro (somente leitura). A reentrância é importante sempre que é possível execução assíncrona; por exemplo, um programa não-reentrante pode não ser seguro para chamar de um operador de sinal. Em sistemas com múltiplas linhas de controle, um programa não-reentrante deve ser chamado apenas dentro de blocos (*interlocks*).

O *parser* **BISON** não é normalmente um programa reentrante, porque ele usa variáveis estatisticamente alocadas para comunicação com **yylex**. Estas variáveis incluem **yylval** e **yylloc**.

A declaração **BISON %pure-parser** diz que há o desejo que o *parser* seja reentrante. Ele aparece como:

%pure-parser

O efeito é que as duas variáveis de comunicação tornam-se variáveis locais em **yyparse**, e uma convenção de chamada diferente é usada para a função do analisador léxico **yylex**. Ver secção B.3.2.4, para maiores detalhes. A variável **yynerrs** também torna-se local em **yyparse** . Ver secção B.3.3. A convenção para a chamada **yyparse** não é alterada.

B.2.6.8 - Sumário das Declarações BISON

Nesta secção esta um sumário das declarações do **BISON**:

%union Declara uma coleção de tipos de dados com valores semânticos - seção B.2.6.3.

%token Declara um simbolo terminal (um nome para um tipo *token*) - secção B.2.6.1.

%right Declara um símbolo terminal (nome para tipo *token*) que esta associado à direita - secção B.2.6.2.

%left Declara um símbolo terminal (nome para tipo *token*) que esta associado à esquerda - secção B.2.6.2.

%nonassoc Declara um símbolo terminal (um nome para um tipo *token*) que é não associativo (é usado de maneira que se fosse associativo seria um erro sintático) - secção B.2.6.2.

%type Declara um tipo de valor semÂntico para um símbolo não terminal - secçào B.2.6.4.

%start Especifica o símbolo que inicia a gramática - secção B.2.6.6.

%expect Declara um número esperado de conflitos *shift-reduce* - secção B.2.6.5.

%pure_parser

Reqisita um programa analisador puro (reeentrante) - secção B.2.6.7.

B.2.7 - Múltiplos analisadores em um mesmo programa

Muitos programas que usam o *parser* **BISON** para uma linguagem somente por essa razão contém apenas um **parser BISON**. Mas pode ser necessário um *parser* para mais de uma linguagem em um mesmo programa. Então o procedimento que deve ser seguido é o seguinte:

. Fazer cada analisador como sendo um analisador puro (secção B.2.6.7). Isto elimina as variáveis globais tais como **yyval**, caso contrario haverá conflito entre os varios analisadores, porém isso requer que se convencione uma regra para efetuar a chamada alternada para a função **yylex**. (ver secção B.3.2.4).

. Em cada arquivo da gramática, definir **yyparse** como uma macro, com o nome do analisador expandido. Colocar esta definição na secção em linguagem C (ver a secção B.2.1.1). Por exemplo:

```
%{
define yyparse parse_algol
%}
```

Deve ser usado então o nome **parse_algol** nos outros arquivos fonte para chamar este analisador.

. Se for analisadores léxicos diferentes para cada gramática, poderá ser definido **yylex** como uma macro, como foi descrito para o **yyparse**. O usando o nome expandido quando for definido o **yylex** em outro arquivo. Se for definido **yylex** no próprio arquivo da gramática, deve ser feito como uma função do tipo **static**, como no exemplo:

```
%{
static int yylex();
%}
... regras gramaticais ...
%%
static int
yylex (yyvalp, yyllocp)
        YYSTYPE *yylvalp;
        YYLTYPE *yyllocp;
{ ... }
```

. Se for necessário uma função **yyerror** diferente para cada gramática, deve ser usado um metodo igual ao descrito para a função **yylex**.

B.3 - Interface em Linguagem C do Analisador

O analisador é, na verdade, uma função C denominada **yyparse**. Será descrito aqui as convenções da interface de **yyparse** e as outras funções que ele precisa usar.

Deve-se ter em mente que o analisador usa muitos identificadores C, começando com "**yy**" e "**YY**" para propósitos internos. Se for usado identificador (além daqueles neste manual) em uma ação ou em um código C adicional no arquivo de gramática, provalvelmente ocorrerão problemas.

B.3.1 - A Função do Analisador yyparse

A função **yyparse** é chamada para provocar a ocorrência de análise. Esta função lê "**tokens**", executa ações, e por último retorna quando encontra fim de entrada (*end-of-input*) ou um erro de sintaxe irrecuperável. Pode-se também escrever uma ação que direciona **yyparse** a retornar imediatamente sem leitura posterior.

O valor retornado por **yyparse** é 0 se a análise foi bem sucedido (o retorno é devido a fim de entrada.

O valor é 1 se a análise falhou (o retorno é devido a um erro de sintaxe).

Em uma ação, pode-se provocar o retorno imediato de **yyparse** pelo uso dos seguintes macros:

YYACCEPT retorna imediatamente com valor 0 (para relatar sucesso)

YYABORT retorna imediatamente com valor 1 (para relatar falha).

B.3.2 - A Função do Analisador Léxico yylex

A função do analisador léxico, **yylex**, reconhece *tokens* da corrente de entrada e retorna-os para o analisador. O **BISON** não cria esta função automaticamente; deve-se escrevê-la de modo que **yyparse** possa chamá-la. A função é algumas vezes denominada como um analisador lexico.

Em programas simples, **yylex** é frequentemente definido no final do arquivo de gramática **BISON**. Se **yylex** é definido em um arquivo fonte separado, é necessário preparar para que as definições macro tipo *token* estejam disponíveis. Para fazer isto, use a opção '**-d**' quando executar o **BISON**, de modo que ele escreverá estas definições macro em um topo de arquivo separado '**name.tab.h**' que pode ser incluído em outros arquivos fonte que precisem dele.

B.3.2.1 - Convenção de Chamada para yylex

O valor que **yylex** retorna deve ser o código numérico para o tipo de *token* que ele acabou de encontrar, ou 0 para fim de entrada.

Quando um *token* é mencionado nas regras de gramática por um nome, este nome no arquivo analisador torna-se um macro em linguagem C cuja definição é o próprio código numérico para aquele tipo de "token". Então, **yylex** pode usar o nome para indicar aquele tipo. Vera secção B.2.2.

Quando um *token* é mencionado nas regras de gramática por um caracter literal, o código numérico para aquele caracter é também o código para o tipo do *token*. Então o **yylex** pode simplesmente retornar aquele código de caracter. O caracter nulo não deve ser usado deste modo, porque seu código é zero, que significa fim de entrada.

A seguir é dado um exemplo mostrando o que foi descrito:

```
yylex()
{
 ...
 if (c == EOF)          /* Detecta fim de4 arquivo. */
   return 0;
 ...
if (c == '+' // c == '-')
  return c;    /* Assume tipo token type para  '+' é '+'. /*
 ...
  return INT; /* Retorna a tipoe do token.          */
 ...
}
```

Esta interface foi projetada de modo que a saída da utilidade **lex** possa ser usada sem alteração como a definição de **yylex**.

B.3.2.2 - Retornando Valores Semânticos de *Tokens*

Em um analisador comun (não-reentrante), o valor semântico do *token* deve ser armazenado na variável global **yyval**. Quando utiliza-se apenas um tipo de dado para valores semânticos, **yyval** tem aquele tipo. Então, se o tipo é **int** (que é o *default*), deve-se escrever em **yylex**:

```
...
yyval = value;          /* Coloca o valor dentro da pilha do BISON. */
return INT;    /* Retorna o tipo do token. */
...
```

Quando estiverem sendo usados múltiplos tipos de dados, o tipo de **yyval** é uma união feita de **%union declaration** (veja secção B.2.6.3). Então quando armazena-se um valor de *token*, deve-se usar o membro apropriado da união. Se a declaração **%union** tem o seguinte aspecto:

```
%union {
        int intval;
        double val;
        symrec *tptr;
}
```

então o código em **yylex** seria:

```
...
yyval.intval = value;          /* Coloca o valor dentro da pilha do BISON. */
return INT;    /* Retorna o tipo de token. */
```

B.3.2.3 - Relatando Posições Textuais de *Tokens*

Se estiver sendo usada a característica 'n' (ver secção B.3.4) em ações para manter a pista das localizações textuais de *tokens* e agrupamentos, então deve-se fornecer esta informação em **yylex**. A função **yyparse** espera encontrar a localização textual de um *token* que acabou de ser analisado na variável global **yylloc**. Então **yylex** deve armazenar o dado apropriado naquela variável. O valor de **yylloc** é uma estrutura e é necessário apenas inicializar os membros que serão usados pelas ações. Os quatro membros são chamados **first-line, first-column, last-line e last column.** Observe que o uso desta característica torna o analisador consideravelmente mais lento.

O tipo de dado de **yyloc** foi chamado **YYLTYPE.**

B.3.2.4 - Convenções de Chamada para Analisadores Puros.

Quando é utilizada a declaração **BISON %pure-parser** para requisitar um analisador puro, reentrante, as variáveis de comunicação global **yylval** e **yylloc** não podem ser usadas. (Ver B.2.6.7). Nesses analisadores, as variáveis globais são substituídas por ponteiros passados como argumentos para **yylex**. Eles devem ser declarados como mostrado aqui, e passar a informação de volta armazenando-a atravás desses ponteiros.

```
yylex (lvalp, llocp)
        YYSTYPE  *lvalp;
        YYLTYPE  *llocp;
{
 ...
 *lvalp = value;        /* Coloca o valor na pilha  do BISON. */
 return INT;   /* Retorna o tipo de token. */
 ...
}
```

B.3.3 - A Função Relatório de Erro yyerror

O analisador **BISON** detecta um erro de análise ou erro sintático sempre que lê um *token* que não pode satisfazer alguma regra sintática. Uma ação na gramática também pode explicitamente proclamar um erro, usando o macro **YYERROR** (ver secção B.3.4).

O analisador **BISON** espera relatar o erro chamando a função de relatório de erro, denominada **yyerror**, que deve ser fornecida. Ela é chamada por **yyparse** sempre que um erro sintático é encontrado, e recebe um argumento. Para um erro de análise, a cadeia é sempre erro de análise.

O analisador pode detectar um outro tipo de erro: pilha cheia (*overflow*). Isto acontece quando a entrada contém construções que são muito profundamente aninhadas. Não é provável que isto seja encontrado, uma vez que o analisador **BISON** estende sua pilha automaticamente até um limite bastante grande. Mas se ocorrer *overflow*, **yyparse** chama **yyerror** da maneira usual, exceto pelo fato de que a cadeia de argumentos é "**parser stack overflow**".

A seguinte definição basta em programas simples:

```
yyerror (s)
        char   *s;
               {
               fprintf  (stderr, "%s\n",s);
               }
```

Depois que **yyerror** retorna para **yyparse**, este último tentará a recuperação do erro, se tiverem sido escritas regras de gramáticas adequadas para recuperação de erros. Se a

SERVIÇO DE BIBLIOTECA E INFORMAÇÃO - IFQSC
FISICA

recuperação é impossível, **yyparse** imediatamente retornará 1.

A variável **yynerrs** contém o número de erros sintáticos encontrados até aí. Normalmente, esta variável é global; mas se for requisitado um analisador puro (ver seção B.2.6.7), então ela é uma variável local que apenas as ações podem acessar.

B.3.4 - Características Especiais para Uso em Ações

Aqui está uma tabela de construções Bison, variáveis e macros que são úteis em ações.

'$$' Age como uma variável que contém o valor semântico para o agrupamento feito pela regra corrente. Ver seção B.2.5.3.

'$n' Age como uma variável que contém o valor semântico para o n-ésimo componente da regra corrente. Ver seção B.2.5.3.

'$<typealt>'

Idêntico a **$$** mas especifica **typealt** alternativo na união especificada pela declaração **%union**. Ver secção B.2.5.4.

'$<typealt>n'

Idêntico a **$n** mas especifica **typealt** alternativo na união especificada pela declaração **%union**. Ver secção B.2.5.4

'YYABORT;'

Retorna imediatamente de **yyparse**, indicando falha. Ver secção B.3.1.

'YYACCEPT;'

Retorna imediatamente de **yyparse**, indicando sucesso. Ver secção B.3.1.

'YYEMPTY' Valor armazenado em **yychar** quando não existe nenhum *token* a ser analisado (**token look-ahead**).

'YYERROR;'

Provoca um erro de sintaxe imediato. Isto faz com que **yyerror** seja chamado,

começando então a recuperação de erro.

'yychar' Variável contendo o a ser analisado corrente. (Em um analisador puro, ela é na verdade uma variável local dentro de **yyparse**). Quando não existe nenhum a ser analisado, o valor **YYERROR** é armazenado aqui.

'yyclearin;'

Descarta o "token look-ahead" corrente. É útil primariamente em regras de erros.

'yyerrok;'

Reassume imediatamente a geração de mensagens de erro para erros sintáticos subsequentes.É útil primariamente em regras de erros.

'n' Age como uma variável estruturada contendo informações sobre o número de linhas e colunas do n-ésimo componente da regra corrente. A estrutura tem quatro membros, como esta:

```
struct {
        int first-line, last-line;
        int first-column, last-column;
};
```

Então, para obter o número de linha inicial do terceiro componente, use **'3.first-line'**. Para que os membros desta estrutura contenham informações válidas **yylex** deve suprir estas informações sobre cada *token*. Se forem necessários apenas alguns membros, então **yylex** precisa completar apenas esses membros. O uso desta característica torna o analisador consideravelmente mais lento.

Apêndice C

C.1 - Micro arquitetura RISC-LIE[1]

. **Banco de registradores:** 138 registradores de 32 bits, com decodificador de porta dupla e *latches* RA, RB, RD para manuseio dos endereços de uma instrução. R_0 sempre possui zero (via *hardware).*

.**PWS:** 13 bits de estado do processador (*processor status word*). Inclui CWP, SWP, condições de código CCs, e bits de controle de interrupção.

.**DST:** *latch* destino, serve também como *latch* temporária para o *pipeline.* O resultado de cada operação é mantido nela até que seu valor seja escrito em um registrador apropriado, ou usado durante o próximo ciclo.

.**SRC:** *latch* de entrada para o *Shifter.* DST ou BI são usados como *latch* de saída do *Shifter.*

.**Shifter:** *Shifter* do tipo *cross-bar* de 32 bits. A quantia a ser deslocada (0-31) é especificada pelo conteúdo da *latch* quant.SHIT e decodifica por S.DEC. Um deslocamento da direita para a esquerda flui na direção busR busL, e ao contráreio quando o deslocamento é da esquerda para direita.

.**AI,BI:** *latches* de entrada da ULA. A ULA não possui *latches* de saída, ela usa DST ou um dos dois barramnetos de saída.

.**ULA:** unidade lógico aritmética para inteiros de 32 bits. Realiza adição, subtração, operações bit-a-bit lógicas (*and, or, xor*), ou passa o conteúdo de BI para saída.

.**BAR:** registrador de endereçamento de *byte,* que computa e manuseia os dois bits menos significativos da soma de AI e BI. Quando a ULA está computando um endereço efetivo, BAR irá conter parte do endereço para alinhamento de dado dentro de uma palavra.

.**NXTPC:** registrador do próximo contador do programa (*next program counter*), que possui o endereço da próxima instrução a ser buscada durante o ciclo corrente.

.**INC:** um incrementador que computa NXTPC + 4.

.**PC:** contador de programa, possui o endereço da instrução que está sendo executada durante o ciclo corrente.

.**LSTPC:** último PC (*last* PC) possui o endereço da última instrução executada, ou que se tentou executar. Quando uma interrupção ocorre, LSTPC conterá o endereço da instrução interrompida, durante o primeiro ciclo após a interrupção, pois já no

---

[1] **Para maiores detalhes, ver [Vale 91].**

segundo ciclo, LSTPC conterá o endereço da primeira instrução da rotina de interrupção.

.IMM: *latch* que manuseia os 19 bits menos significativos da instrução, caso ela contenha uma constante imediata.

.DIMM: *latch* combinado com um circuito que extende o sinal ou preenche com zeros. Manuseia dados vindos da cacheD ou operandos imediatos vindos de IMM.

.OP: manuseia o código da operação, o bit de operando imediato.

.busA, busB: barramento do banco de registradores.

.busD: barramento usado para alimentar AI e DST.

.busR, busL: barramentos do *Shifter*, busR é também utilizado para alimentar BI, enquanto busL é usado também para introduzir dados vindos de DIMM na rota dos dados.

.busDADO: usado para buscar/enviar dados e endereços à cache de dados.

.BUSInstr: usado para endereçar instruções na cache de instruções, e para alimentar RA, RB, RD, IMM e OP.

.cacheD: cache de dados.

.mmuD: unidade de gerenciamento de memória de daos (interna ã cacheD).

.cacheI: cache de instruções.

.mmuI: unidade de gerenciamento de memória de instruções (interna a Icache).

C.2 Tabela de instruções da arquitetura RISC-LIE [2]

Tabela C.1: Conjunto de Instruções RISC-LIE

Instrução	operandos	comentário	nome
add	Rs,S2,Rd	Rd ← Rs+S2	adição inteira
addc	Rs,S2,Rd	Rd ← Rs+S2+*carry*	adição com *carry*
sub	Rs,S2,Rd	Rd ← Rs-S2	subtração inteira
subc	Rs,S2,Rd	Rd ← Rs-S2-*carry*	subtração c/ *carry*
subi	Rs,S2,Rd	Rd ← S2-Rs	subtração inteira
subci	Rs,S2,Rd	Rd ← S2-Rs-*carry*	subtração c/ *carry*
and	Rs,S2,Rd	Rd ← Rs & S2	"E" lógico
or	Rs,S2,Rd	Rd ← Rs / S2	"OU" lógico
xor	Rs,S2,Rd	Rd ← Rs xor S2	"OU exclus." lóg.
sll	Rs,S2,Rd	Rd ← Rs shift S2	desloc. à esq.
srl	Rs,S2,Rd	Rd ← Rs shift S2	desloc.lóg. à dir.
sra	Rs,S2,Rd	Rd ← Rs shift S2	desloc.aritm.àdir.
ldxw	(Rx)S2,Rd	Rd ← M [Rx+S2]	carr. palavra
ldxhu	(Rx)S2,Rd	Rd ← M [Rx+S2]	carr.meiap.s/sinal
ldxhs	(Rx)S2,Rd	Rd ← M [Rx+S2]	carr.meiap.c/sinal
ldxbu	(Rx)S2,Rd	Rd ← M [Rx+S2]	carr.*byte* s/sinal
ldxbs	(Rx)S2,Rd	Rd ← M [Rx+S2]	carr.*byte* c/sinal
stxw	Rm,(Rx)S2	M [Rx+S2] ← Rm	armaz. palavra
stxh	Rm,(Rx)S2	M [Rx+S2] ← Rm	armaz. meiap.
stxb	Rm,(Rx)S2	M [Rx+S2] ← Rm	armaz. *byte*
jmpx	COND,(Rx)S2	pc ← Rx+S2	desvio condicional
jmpr	COND,Y	pc ← pc+Y	desvio relativo
callx	Rd,(Rx)S2	Rd ← pc,next	*call* e troca
		pc ← Rx+S2,CWP←CWP-1	janela
callr	Rd,Y	Rd ← pc,next	*call* relat.e troca
		pc ← pc+Y,CWP←CWP-1	janela
ret	Rm,S2	pc ← Rm+S2,CWP←CWP+1	ret.e troca janela
ldhi	Rd,Y	Rd<31:13>←Y;Rd<12:0>←0	car. *high* imediato
getlpc	Rd	Rd ← last pc	lê últ.cont. prog.
getpsw	Rd	Rd ← PSW	car.*prog.stat.word*
putpsw	Rm	PSW ← Rm	car.*prog.stat.word*
time[3]	Rs,S2,Rd	Rd ← nciclos-(Rs+S2)	lê relógio

[2] proposta desenvolvida por Valente [Vale91]

[3] Não utilizada pelo compilador PARIS

SERVIÇO DE BIBLIOTECA E INFORMAÇÃO - IFQSC

Apêndice D

D.1 - Listagem do analisador léxico

A saída do analisador léxico é observada pela emissão do texto de entrada lido, que pode ser obtida digitando-se a diretiva de entrada **-l** junto ao nome do programa de entrada (**shell.pas**), que corresponde ao próprio arquivo **shell.pas**, e é mostrada abaixo:

```
1>  PROGRAM shell (output, input);
2>        CONST max = 11;
3>              lim = 5;
4>        VAR vetor          : ARRAY [1..max] OF INTEGER;
5>           item            : ARRAY [1..lim] OF INTEGER;
6>           count, x        : INTEGER;
7>           i, j, k, s, w   : INTEGER;
8>         BEGIN
9>           vetor[1] := 9; vetor[2] := 5; vetor[3] := 3; vetor[4] := 3; vetor[5] := 1;
10>          { vetor a ser ordenado }
11>            item[1] := 9; item[2] := 5; item[3] := 3; item[4] := 9;
12>            item[5] := 5; item[6] := 3; item[7] := 3; item[8] := 1;
13>            item[9] := 3; item[9] := 1; item[10]:= 3; item[11]:= 1;
14>          FOR w := 1  TO w < lim  DO
15>              BEGIN
16>                k := vetor[w];
17>                s := -k;
18>                FOR  i := k TO  i < count  DO
19>                   BEGIN
20>                      x := item[i];
21>                      j := i - 1;
22>                      IF s = 1  THEN
23>                         BEGIN
24>                            s := -k;
25>                            s := s + 1;
26>                            item[s] := x;
27>                         END;
28>                END;
29>                WHILE x < item[j] DO
30>                  WHILE  i >= 0  DO
31>                    WHILE j <= count DO
32>                      BEGIN
33>                        item[j+k] := item[j];
34>                        j := k-x;
35>                      END;
36>               item[j+k] := x;
37>        END;
38>   END.
```

D.2 - Listagem do analisador sintático

A saída para o analisador sintático é visualizada quando se digita a opção **-s**, que irá gerar na tela, uma listagem que permite que seja montada a árvore sintática correspondente ao arquivo **shell.pas**. A árvore sintática deve ser montada na ordem inversa em que surgem as mensagens de reconhecimento. Esta listagem é mostrada abaixo:

```
1> iniciou lista
2> -->yylval.sym {shell}(0)
3> ~IDENTIFICADOR shell
4> -->yylval.sym {output}(1)
5> ~IDENTIFICADOR output
6> -->yylval.sym {input}(2)
7> ~IDENTIFICADOR input
8> pos na lista input
9> Variavel_arquivo: input
10> pos na lista output
11> Variaveis de arquivo: output
12> Nome do programa com lista: shell
13> inic percorre lista
14> lendo a lista => ...input
15> lendo input
16> lendo a lista => ...output
17> lendo output
18> re-iniciou lista
19> -->yylval.sym {max}(3)
20> ~IDENTIFICADOR max
21> constante inteira
22> -->yylval.sym {lim}(5)
23> ~IDENTIFICADOR lim
24> constante inteira
25> identificador de constante = lim
26> identificador de varias constante ; max
27> inic percorre lista
28> definicao de constante: max
29> cadeia vazia
30> sem definicao de tipo
31> -->yylval.sym {vetor}(7)
32> ~IDENTIFICADOR vetor
33> pos na lista vetor
34> -->yylval.sym {max}(3)
35> ~IDENTIFICADOR max
36> nome de tipo ordinal tipo ordinal
```

```
37> tipo denotado INTEGER
38> definicao de arranjo de tipo
39> tipo arranjo
40> tipo denotado novo tipo
41> declaracao de lista de variaveis:
42> inic percorre lista
43> lendo a lista => ...vetor
44> re-iniciou lista
45> -->yylval.sym {item}(8)
46> ~IDENTIFICADOR item
47> pos na lista item
48> -->yylval.sym {lim}(5)
49> ~IDENTIFICADOR lim
50> nome de tipo ordinal tipo ordinal
51> tipo denotado INTEGER
52> definicao de arranjo de tipo
53> tipo arranjo
54> tipo denotado novo tipo
55> declaracao de lista de variaveis:
56> inic percorre lista
57> lendo a lista => ...item
58> re-iniciou lista
59> -->yylval.sym {count}(9)
60> ~IDENTIFICADOR count
61> -->yylval.sym {x}(10)
62> ~IDENTIFICADOR x
63> pos na lista x
64> pos na lista count
65> Variavel -> count
66> tipo denotado INTEGER
67> declaracao de lista de variaveis:
68> inic percorre lista
69> lendo a lista => ...x
70> lendo a lista => ...count
71> re-iniciou lista
72> -->yylval.sym {i}(11)
73> ~IDENTIFICADOR i
74> -->yylval.sym {j}(12)
75> ~IDENTIFICADOR j
76> -->yylval.sym {k}(13)
77> ~IDENTIFICADOR k
78> -->yylval.sym {s}(14)
79> ~IDENTIFICADOR s
80> -->yylval.sym {w}(15)
81> ~IDENTIFICADOR w
82> pos na lista w
83> pos na lista s
```

```
84> Variavel -> s
85> pos na lista k
86> Variavel -> k
87> pos na lista j
88> Variavel -> j
89> pos na lista i
90> Variavel -> i
91> tipo denotado INTEGER
92> declaracao de lista de variaveis:
93> inic percorre lista
94> lendo a lista => ...w
95> lendo a lista => ...s
96> lendo a lista => ...k
97> lendo a lista => ...j
98> lendo a lista => ...i
99> re-iniciou lista
100> uma declaracao de VAR
101> varias VAR
102> varias VAR
103> varias VAR
104> Declaracao de Variavel -> VAR
105> cadeia vazia
106> sem funcao nem procedimento
107> -->yylval.sym {vetor}(7)
108> ~IDENTIFICADOR vetor
109> Variavel(L): vetor
110> variavel associada: constante inteira
111> fator
112> termo
113> expressao simples
114> expressaotipo de componente -> indice
115> componente de variavel
116> variavel associada: constante inteira
117> fator
118> termo
119> expressao simples
120> associacao de variavel = expressao
121> declaracao simples com associacao
122> declaracao simples
123> -->yylval.sym {vetor}(7)
124> ~IDENTIFICADOR vetor
125> Variavel(L): vetor
126> variavel associada: constante inteira
127> fator
128> termo
129> expressao simples
130> expressaotipo de componente -> indice
```

```
131> componente de variavel
132> variavel associada: constante inteira
133> fator
134> termo
135> expressao simples
136> associacao de variavel = expressao
137> declaracao simples com associacao
138> declaracao simples
139> -->yylval.sym {vetor}(7)
140> ~IDENTIFICADOR vetor
141> Variavel(L): vetor
142> variavel associada: constante inteira
143> fator
144> termo
145> expressao simples
146> expressaotipo de componente -> indice
147> componente de variavel
148> variavel associada: constante inteira
149> fator
150> termo
151> expressao simples
152> associacao de variavel = expressao
153> declaracao simples com associacao
154> declaracao simples
155> -->yylval.sym {vetor}(7)
156> ~IDENTIFICADOR vetor
157> Variavel(L): vetor
158> variavel associada: constante inteira
159> fator
160> termo
161> expressao simples
162> expressaotipo de componente -> indice
163> componente de variavel
164> variavel associada: constante inteira
165> fator
166> termo
167> expressao simples
168> associacao de variavel = expressao
169> declaracao simples com associacao
170> declaracao simples
171> -->yylval.sym {vetor}(7)
172> ~IDENTIFICADOR vetor
173> Variavel(L): vetor
174> variavel associada: constante inteira
175> fator
176> termo
177> expressao simples
```

```
178> expressaotipo de componente -> indice
179> componente de variavel
180> variavel associada: constante inteira
181> fator
182> termo
183> expressao simples
184> associacao de variavel = expressao
185> declaracao simples com associacao
186> declaracao simples
187> -->yylval.sym {item}(8)
188> ~IDENTIFICADOR item
189> Variavel(L): item
190> variavel associada: constante inteira
191> fator
192> termo
193> expressao simples
194> expressaotipo de componente -> indice
195> componente de variavel
196> variavel associada: constante inteira
197> fator
198> termo
199> expressao simples
200> associacao de variavel = expressao
201> declaracao simples com associacao
202> declaracao simples
203> -->yylval.sym {item}(8)
204> ~IDENTIFICADOR item
205> Variavel(L): item
206> variavel associada: constante inteira
207> fator
208> termo
209> expressao simples
210> expressaotipo de componente -> indice
211> componente de variavel
212> variavel associada: constante inteira
213> fator
214> termo
215> expressao simples
216> associacao de variavel = expressao
217> declaracao simples com associacao
218> declaracao simples
219> -->yylval.sym {item}(8)
220> ~IDENTIFICADOR item
221> Variavel(L): item
222> variavel associada: constante inteira
223> fator
224> termo
```

225> expressao simples
226> expressaotipo de componente -> indice
227> componente de variavel
228> variavel associada: constante inteira
229> fator
230> termo
231> expressao simples
232> associacao de variavel = expressao
233> declaracao simples com associacao
234> declaracao simples
235> -->yylval.sym {item}(8)
236> ~IDENTIFICADOR item
237> Variavel(L): item
238> variavel associada: constante inteira
239> fator
240> termo
241> expressao simples
242> expressaotipo de componente -> indice
243> componente de variavel
244> variavel associada: constante inteira
245> fator
246> termo
247> expressao simples
248> associacao de variavel = expressao
249> declaracao simples com associacao
250> declaracao simples
251> -->yylval.sym {item}(8)
252> ~IDENTIFICADOR item
253> Variavel(L): item
254> variavel associada: constante inteira
255> fator
256> termo
257> expressao simples
258> expressaotipo de componente -> indice
259> componente de variavel
260> variavel associada: constante inteira
261> fator
262> termo
263> expressao simples
264> associacao de variavel = expressao
265> declaracao simples com associacao
266> declaracao simples
267> -->yylval.sym {item}(8)
268> ~IDENTIFICADOR item
269> Variavel(L): item
270> variavel associada: constante inteira
271> fator

```
272> termo
273> expressao simples
274> expressaotipo de componente -> indice
275> componente de variavel
276> variavel associada: constante inteira
277> fator
278> termo
279> expressao simples
280> associacao de variavel = expressao
281> declaracao simples com associacao
282> declaracao simples
283> -->yylval.sym {item}(8)
284> ~IDENTIFICADOR item
285> Variavel(L): item
286> variavel associada: constante inteira
287> fator
288> termo
289> expressao simples
290> expressaotipo de componente -> indice
291> componente de variavel
292> variavel associada: constante inteira
293> fator
294> termo
295> expressao simples
296> associacao de variavel = expressao
297> declaracao simples com associacao
298> declaracao simples
299> -->yylval.sym {item}(8)
300> ~IDENTIFICADOR item
301> Variavel(L): item
302> variavel associada: constante inteira
303> fator
304> termo
305> expressao simples
306> expressaotipo de componente -> indice
307> componente de variavel
308> variavel associada: constante inteira
309> fator
310> termo
311> expressao simples
312> associacao de variavel = expressao
313> declaracao simples com associacao
314> declaracao simples
315> -->yylval.sym {item}(8)
316> ~IDENTIFICADOR item
317> Variavel(L): item
318> variavel associada: constante inteira
```

```
319> fator
320> termo
321> expressao simples
322> expressaotipo de componente -> indice
323> componente de variavel
324> variavel associada: constante inteira
325> fator
326> termo
327> expressao simples
328> associacao de variavel = expressao
329> declaracao simples com associacao
330> declaracao simples
331> -->yylval.sym {item}(8)
332> ~IDENTIFICADOR item
333> Variavel(L): item
334> variavel associada: constante inteira
335> fator
336> termo
337> expressao simples
338> expressaotipo de componente -> indice
339> componente de variavel
340> variavel associada: constante inteira
341> fator
342> termo
343> expressao simples
344> associacao de variavel = expressao
345> declaracao simples com associacao
346> declaracao simples
347> -->yylval.sym {item}(8)
348> ~IDENTIFICADOR item
349> Variavel(L): item
350> variavel associada: constante inteira
351> fator
352> termo
353> expressao simples
354> expressaotipo de componente -> indice
355> componente de variavel
356> variavel associada: constante inteira
357> fator
358> termo
359> expressao simples
360> associacao de variavel = expressao
361> declaracao simples com associacao
362> declaracao simples
363> -->yylval.sym {item}(8)
364> ~IDENTIFICADOR item
365> Variavel(L): item
```

```
366> variavel associada: constante inteira
367> fator
368> termo
369> expressao simples
370> expressaotipo de componente -> indice
371> componente de variavel
372> variavel associada: constante inteira
373> fator
374> termo
375> expressao simples
376> associacao de variavel = expressao
377> declaracao simples com associacao
378> declaracao simples
379> -->yylval.sym {w}(15)
380> ~IDENTIFICADOR w
381> variavel associada: constante inteira
382> fator
383> termo
384> expressao simples
385> -->yylval.sym {w}(15)
386> ~IDENTIFICADOR w
387> Variavel(L): w
388> variavel associada: fator
389> fator
390> termo
391> -->yylval.sym {lim}(5)
392> ~IDENTIFICADOR lim
393> Variavel(L): lim
394> variavel associada: fator
395> fator
396> termo
397> LT
398> expressao relacional
399> -->yylval.sym {k}(13)
400> ~IDENTIFICADOR k
401> -->yylval.sym {vetor}(7)
402> ~IDENTIFICADOR vetor
403> Variavel(L): vetor
404> -->yylval.sym {w}(15)
405> ~IDENTIFICADOR w
406> Variavel(L): w
407> variavel associada: fator
408> fator
409> termo
410> expressao simples
411> expressaotipo de componente -> indice
412> componente de variavel
```

SERVIÇO DE BIBLIOTECA E INFORMAÇÃO - IFQSC
FÍSICA

```
413> variavel associada: fator
414> fator
415> termo
416> expressao simples
417> identficador = expressao
418> declaracao simples com associacao
419> declaracao simples
420> -->yylval.sym {s}(14)
421> ~IDENTIFICADOR s
422> -->yylval.sym {k}(13)
423> ~IDENTIFICADOR k
424> Variavel(L): k
425> variavel associada: fator
426> fator
427> -termo
428> expressao simples
429> identficador = expressao
430> declaracao simples com associacao
431> declaracao simples
432> -->yylval.sym {i}(11)
433> ~IDENTIFICADOR i
434> -->yylval.sym {k}(13)
435> ~IDENTIFICADOR k
436> Variavel(L): k
437> variavel associada: fator
438> fator
439> termo
440> expressao simples
441> -->yylval.sym {i}(11)
442> ~IDENTIFICADOR i
443> Variavel(L): i
444> variavel associada: fator
445> fator
446> termo
447> -->yylval.sym {count}(9)
448> ~IDENTIFICADOR count
449> Variavel(L): count
450> variavel associada: fator
451> fator
452> termo
453> LT
454> expressao relacional
455> -->yylval.sym {x}(10)
456> ~IDENTIFICADOR x
457> -->yylval.sym {item}(8)
458> ~IDENTIFICADOR item
459> Variavel(L): item
```

```
460> -->yylval.sym {i}(11)
461> ~IDENTIFICADOR i
462> Variavel(L): i
463> variavel associada: fator
464> fator
465> termo
466> expressao simples
467> expressaotipo de componente -> indice
468> componente de variavel
469> variavel associada: fator
470> fator
471> termo
472> expressao simples
473> identficador = expressao
474> declaracao simples com associacao
475> declaracao simples
476> -->yylval.sym {j}(12)
477> ~IDENTIFICADOR j
478> -->yylval.sym {i}(11)
479> ~IDENTIFICADOR i
480> Variavel(L): i
481> variavel associada: fator
482> fator
483> variavel associada: constante inteira
484> fator
485> termo - termo
486> expressao simples
487> identficador = expressao
488> declaracao simples com associacao
489> declaracao simples
490> -->yylval.sym {s}(14)
491> ~IDENTIFICADOR s
492> Variavel(L): s
493> variavel associada: fator
494> fator
495> termo
496> variavel associada: constante inteira
497> fator
498> termo
499> EQ
500> -->yylval.sym {s}(14)
501> ~IDENTIFICADOR s
502> -->yylval.sym {k}(13)
503> ~IDENTIFICADOR k
504> Variavel(L): k
505> variavel associada: fator
506> fator
```

```
507> -termo
508> expressao simples
509> identficador = expressao
510> declaracao simples com associacao
511> declaracao simples
512> -->yylval.sym {s}(14)
513> ~IDENTIFICADOR s
514> -->yylval.sym {s}(14)
515> ~IDENTIFICADOR s
516> Variavel(L): s
517> variavel associada: fator
518> fator
519> variavel associada: constante inteira
520> fator
521> termo + termo
522> expressao simples
523> identficador = expressao
524> declaracao simples com associacao
525> declaracao simples
526> -->yylval.sym {item}(8)
527> ~IDENTIFICADOR item
528> Variavel(L): item
529> -->yylval.sym {s}(14)
530> ~IDENTIFICADOR s
531> Variavel(L): s
532> variavel associada: fator
533> fator
534> termo
535> expressao simples
536> expressaotipo de componente -> indice
537> componente de variavel
538> -->yylval.sym {x}(10)
539> ~IDENTIFICADOR x
540> Variavel(L): x
541> variavel associada: fator
542> fator
543> termo
544> expressao simples
545> associacao de variavel = expressao
546> declaracao simples com associacao
547> declaracao simples
548> cadeia vazia
549> sem declaracao simples
550> declaracao simples
551> uma sequencia de declaracoes
552> varias sequencias de declaracoes
553> varias sequencias de declaracoes
```

```
554> varias sequencias de declaracoes
555> componente BEGIN-END
556> componente com BEGIN-END
557> declaracao estruturada
558> IF sem ELSE
559> condicao if
560> declaracao condicional
561> declaracao estruturada
562> uma sequencia de declaracoes
563> varias sequencias de declaracoes
564> varias sequencias de declaracoes
565> componente BEGIN-END
566> componente com BEGIN-END
567> declaracao estruturada
568> FOR com TO
569> declaracao FOR
570> declaracao de laco repetitivo
571> declaracao estruturada
572> -->yylval.sym {x}(10)
573> ~IDENTIFICADOR x
574> Variavel(L): x
575> variavel associada: fator
576> fator
577> termo
578> -->yylval.sym {item}(8)
579> ~IDENTIFICADOR item
580> Variavel(L): item
581> -->yylval.sym {j}(12)
582> ~IDENTIFICADOR j
583> Variavel(L): j
584> variavel associada: fator
585> fator
586> termo
587> expressao simples
588> expressaotipo de componente -> indice
589> componente de variavel
590> variavel associada: fator
591> fator
592> termo
593> LT
594> -->yylval.sym {i}(11)
595> ~IDENTIFICADOR i
596> Variavel(L): i
597> variavel associada: fator
598> fator
599> termo
600> variavel associada: constante inteira
```

```
601> fator
602> termo
603> GE
604> -->yylval.sym {j}(12)
605> ~IDENTIFICADOR j
606> Variavel(L): j
607> variavel associada: fator
608> fator
609> termo
610> -->yylval.sym {count}(9)
611> ~IDENTIFICADOR count
612> Variavel(L): count
613> variavel associada: fator
614> fator
615> termo
616> LE
617> -->yylval.sym {item}(8)
618> ~IDENTIFICADOR item
619> Variavel(L): item
620> -->yylval.sym {j}(12)
621> ~IDENTIFICADOR j
622> Variavel(L): j
623> variavel associada: fator
624> fator
625> -->yylval.sym {k}(13)
626> ~IDENTIFICADOR k
627> Variavel(L): k
628> variavel associada: fator
629> fator
630> termo + termo
631> expressao simples
632> expressao
633> tipo de componente -> indice
634> componente de variavel
635> -->yylval.sym {item}(8)
636> ~IDENTIFICADOR item
637> Variavel(L): item
638> -->yylval.sym {j}(12)
639> ~IDENTIFICADOR j
640> Variavel(L): j
641> variavel associada: fator
642> fator
643> termo
644> expressao simples
645> expressao
646> tipo de componente -> indice
647> componente de variavel
```

```
648> variavel associada: fator
649> fator
650> termo
651> expressao simples
652> associacao de variavel = expressao
653> declaracao simples com associacao
654> declaracao simples
655> -->yylval.sym {j}(12)
656> ~IDENTIFICADOR j
657> -->yylval.sym {k}(13)
658> ~IDENTIFICADOR k
659> Variavel(L): k
660> variavel associada: fator
661> fator
662> -->yylval.sym {x}(10)
663> ~IDENTIFICADOR x
664> Variavel(L): x
665> variavel associada: fator
666> fator
667> termo - termo
668> expressao simples
669> identficador = expressao
670> declaracao simples com associacao
671> declaracao simples
672> cadeia vazia
673> sem declaracao simples
674> declaracao simples
675> uma sequencia de declaracoes
676> varias sequencias de declaracoes
677> varias sequencias de declaracoes
678> componente BEGIN-END
679> componente com BEGIN-END
680> declaracao estruturada
681> declaracao de WHILE - DO
682> declaracao WHILE
683> declaracao de laco repetitivo
684> declaracao estruturada
685> declaracao de WHILE - DO
686> declaracao WHILE
687> declaracao de laco repetitivo
688> declaracao estruturada
689> declaracao de WHILE - DO
690> declaracao WHILE
691> declaracao de laco repetitivo
692> declaracao estruturada
693> -->yylval.sym {item}(8)
694> ~IDENTIFICADOR item
```

695> Variavel(L): item
696> -->yylval.sym {j}(12)
697> ~IDENTIFICADOR j
698> Variavel(L): j
699> variavel associada: fator
700> fator
701> -->yylval.sym {k}(13)
702> ~IDENTIFICADOR k
703> Variavel(L): k
704> variavel associada: fator
705> fator
706> termo + termo
707> expressao simples
708> expressaotipo de componente -> indice
709> componente de variavel
710> -->yylval.sym {x}(10)
711> ~IDENTIFICADOR x
712> Variavel(L): x
713> variavel associada: fator
714> fator
715> termo
716> expressao simples
717> associacao de variavel = expressao
718> declaracao simples com associacao
719> declaracao simples
720> cadeia vazia
721> sem declaracao simples
722> declaracao simples
723> uma sequencia de declaracoes
724> varias sequencias de declaracoes
725> varias sequencias de declaracoes
726> varias sequencias de declaracoes
727> varias sequencias de declaracoes
728> varias sequencias de declaracoes
729> componente BEGIN-END
730> componente com BEGIN-END
731> declaracao estruturada
732> FOR com TO
733> declaracao FOR
734> declaracao de laco repetitivo
735> declaracao estruturada
736> cadeia vazia
737> sem declaracao simples
738> declaracao simples
739> uma sequencia de declaracoes
740> varias sequencias de declaracoes
741> varias sequencias de declaracoes

```
742> varias sequencias de declaracoes
743> varias sequencias de declaracoes
744> varias sequencias de declaracoes
745> varias sequencias de declaracoes
746> varias sequencias de declaracoes
747> varias sequencias de declaracoes
748> varias sequencias de declaracoes
749> varias sequencias de declaracoes
750> varias sequencias de declaracoes
751> varias sequencias de declaracoes
752> varias sequencias de declaracoes
753> varias sequencias de declaracoes
754> varias sequencias de declaracoes
755> varias sequencias de declaracoes
756> varias sequencias de declaracoes
757> varias sequencias de declaracoes
758> componente BEGIN-END
759> => fim de bloco
760> ╔═══════════════════════╗
761> ‖ FIM DE PROGRAMA ‖
762> ╚═══════════════════════╝
```

D.3 - Listagem do analisador semântico

A saída do analisador semântico é visualizada pela impressão da tabela de símbolos, e pela impressão da pilha da expressão polonesa produzida. Esta saída é obtida utilizando-se a opção -m. Para uma melhor visualização, a opção -l foi também usada, que imprime a saída do analisador léxico. Esta listagem é mostrada abaixo:

```
1> Marcou escopo(0):0
2> $ 1> PROGRAM shell (output, input);
3> $ 2>        CONST max = 11;
4> $ 3>              lim = 5;
5> $ 4>        VAR vetor          : ARRAY [1..max] OF INTEGER;
6> $ 5>            item           : ARRAY [1..lim] OF INTEGER;
7> $ 6>            count, x       : INTEGER;
8> $ 7>            i, j, k, s, w  : INTEGER;
9> $ 8>        BEGIN
10> $ 9>
11> =======Polonesa:
12> <7><16><-6><17><-17>
13> =======Polonesa:
14> <7><18><-6><19><-17>
15> =======Polonesa:
16> <7><20><-6><21><-17>
17> $ 10>          vetor[1] := 9; vetor[2] := 5; vetor[3] := 3;
18> =======Polonesa:
19> <7><22><-6><23><-17>
20> =======Polonesa:
21> <7><24><-6><25><-17>
22> $ 11>          vetor[4] := 3; vetor[5] := 1;
23> $ 12>         { vetor a ser ordenado }
24> =======Polonesa:
25> <8><26><-6><27><-17>
26> =======Polonesa:
27> <8><28><-6><29><-17>
28> =======Polonesa:
29> <8><30><-6><31><-17>
30> =======Polonesa:
31> <8><32><-6><33><-17>
32> $ 13>          item[1] := 9; item[2] := 5; item[3] := 3; item[4] := 9;
33> =======Polonesa:
34> <8><34><-6><35><-17>
35> =======Polonesa:
36> <8><36><-6><37><-17>
37> =======Polonesa:
```

SERVIÇO DE BIBLIOTECA E INFORMAÇÃO - IFGSC
FÍSICA

```
38> <8><38><-6><39><-17>
39> =======Polonesa:
40> <8><40><-6><41><-17>
41> $ 14>            item[5] := 5; item[6] := 3; item[7] := 3; item[8] := 1;
42> =======Polonesa:
43> <8><42><-6><43><-17>
44> =======Polonesa:
45> <8><44><-6><45><-17>
46> =======Polonesa:
47> <8><46><-6><47><-17>
48> =======Polonesa:
49> <8><48><-6><49><-17>
50> $ 15>            item[9] := 3; item[9] := 1; item[10]:= 3; item[11]:= 1;
51> =======Polonesa:
52> <50><15><-17>
53> =======Polonesa:
54> <15><5><-12>
55> $ 17>
56> $ 18>            BEGIN
57> $ 19>
58> =======Polonesa:
59> <7><15><-6><13><-17>
60> $ 20>            k := vetor[w];
61> =======Polonesa:
62> <13><-8><14><-17>
63> $ 21>            s := -k;
64> =======Polonesa:
65> <13><11><-17>
66> =======Polonesa:
67> <11><9><-12>
68> $ 22>            FOR  i := k TO  i < count  DO
69> $ 23>               BEGIN
70> $ 24>
71> =======Polonesa:
72> <8><11><-6><10><-17>
73> $ 25>                  x := item[i];
74> =======Polonesa:
75> <11><51><-2><12><-17>tipo de dado desconhecido
76> $ 26>                  j := i - 1;
77> $ 27>                  IF s = 1  THEN
78> =======Polonesa:
79> <14><52><-16>tipo de dado desconhecido
80> =======Polonesa:
81> $ 28>
82> $ 29>                     BEGIN
83> =======Polonesa:
84> <13><-8><14><-17>
```

```
85>  $ 30>                          s := -k;
86>  =======Polonesa:
87>  <14><53><-1><14><-17>
88>  $ 31>                          s := s + 1;
89>  =======Polonesa:
90>  <8><14><-6><10><-17>
91>  $ 32>                          item[s] := x;
92>  $ 33>                        END;
93>  $ 34>                END;
94>  =======Polonesa:
95>  <10><8><12><-6><-12>
96>  =======Polonesa:
97>  $ 35>                WHILE x < item[j] DO
98>  $ 36>                  WHILE  i >= 0  DO
99>  =======Polonesa:
100> <11><54><-13>
101> =======Polonesa:
102> $ 37>
103> $ 38>                    WHILE j <= count DO
104> =======Polonesa:
105> <12><9><-14>
106> =======Polonesa:
107> $ 39>
108> $ 40>                      BEGIN
109> =======Polonesa:
110> <8><12><13><-1><-6><8><12><-6><-17>
111> $ 41>                        item[j+k] := item[j];
112> =======Polonesa:
113> <13><10><-2><12><-17>
114> $ 42>                        j := k-x;
115> $ 43>                      END;
116> =======Polonesa:
117> <8><12><13><-1><-6><10><-17>
118> $ 44>                item[j+k] := x;
119> $ 45>        END;
120> $ 46>   END.
121>
122> ----------------------------------------
123> [0]shell   LABEL, 0  21
124> [1]output   ARQUIVO, 0  23
125> [2]input   ARQUIVO, 0  22
126> [3]max   C_INTEGER, 0  0
127> [4]     24
128> [5]lim   C_INTEGER, 0  0
129> [6]     25
130> [7]vetor   V_INTEGER, 0  26
131> [8]item   V_INTEGER, 0  27
```

```
132> [9]count   V_INTEGER, 0  29
133> [10]x   V_INTEGER, 0  28
134> [11]i   V_INTEGER, 0  34
135> [12]j   V_INTEGER, 0  33
136> [13]k   V_INTEGER, 0  32
137> [14]s   V_INTEGER, 0  31
138> [15]w   V_INTEGER, 0  30
139> [16]    35
140> [17]    36
141> [18]    37
142> [19]    38
143> [20]    39
144> [21]    40
145> [22]    41
146> [23]    42
147> [24]    43
148> [25]    44
149> [26]    45
150> [27]    46
151> [28]    47
152> [29]    48
153> [30]    49
154> [31]    50
155> [32]    51
156> [33]    52
157> [34]    53
158> [35]    54
159> [36]    55
160> [37]    56
161> [38]    57
162> [39]    58
163> [40]    59
164> [41]    60
165> [42]    61
166> [43]    62
167> [44]    63
168> [45]    64
169> [46]    65
170> [47]    66
171> [48]    67
172> [49]    68
173> [50]    69
174> [51]    70
175> [52]    71
176> [53]    72
177> [54]    73
178> ----------------------------------------Novo MaxSimbolo(0):0
```

D.4 - Listagem do código RISC-LIE produzido (geração de código)

A saída para o exemplo apresentado, é o arquivo **shell.rsc**, que contém o programa do arquivo **shell.pas** transformado em código RISC-LIE. O arquivo saída é mostrado abaixo:

```
1>  add   R0,#1 ,R16
2>  add   R0,#0 ,R17
3>  ldxw  (R0)#35 ,R16
4>  ldxw  (R0)#1 ,R16
5>  stxw  (R18)R36 ,R0
6>  add   R0,#2 ,R17
7>  ldxw  (R0)#26 ,R16
8>  stxw  (R18)R37 ,R0
9>  ldxw  (R0)#2 ,R16
10> stxw  (R18)R38 ,R0
11> add   R0,#3 ,R17
12> ldxw  (R0)#26 ,R16
13> stxw  (R18)R39 ,R0
14> ldxw  (R0)#3 ,R16
15> stxw  (R18)R40 ,R0
16> add   R0,#4 ,R17
17> ldxw  (R0)#26 ,R16
18> stxw  (R18)R41 ,R0
19> ldxw  (R0)#4 ,R16
20> stxw  (R18)R42 ,R0
21> add   R0,#5 ,R17
22> ldxw  (R0)#26 ,R16
23> stxw  (R18)R43 ,R0
24> ldxw  (R0)#5 ,R16
25> stxw  (R18)R44 ,R0
26> add   R0,#1 ,R17
27> ldxw  (R0)#27 ,R16
28> stxw  (R18)R45 ,R0
29> ldxw  (R0)#1 ,R16
30> stxw  (R18)R46 ,R0
31> add   R0,#2 ,R17
32> ldxw  (R0)#27 ,R16
33> stxw  (R18)R47 ,R0
34> ldxw  (R0)#2 ,R16
35> stxw  (R18)R48 ,R0
36> add   R0,#3 ,R17
37> ldxw  (R0)#27 ,R16
38> stxw  (R18)R49 ,R0
39> ldxw  (R0)#3 ,R16
40> stxw  (R18)R50 ,R0
```

```
41>  add    R0,#4 ,R17
42>  ldxw   (R0)#27 ,R16
43>  stxw   (R18)R51 ,R0
44>  ldxw   (R0)#4 ,R16
45>  stxw   (R18)R52 ,R0
46>  add    R0,#5 ,R17
47>  ldxw   (R0)#27 ,R16
48>  stxw   (R18)R53 ,R0
49>  ldxw   (R0)#5 ,R16
50>  stxw   (R18)R54 ,R0
51>  add    R0,#6 ,R17
52>  ldxw   (R0)#27 ,R16
53>  stxw   (R18)R55 ,R0
54>  ldxw   (R0)#6 ,R16
55>  stxw   (R18)R56 ,R0
56>  add    R0,#7 ,R17
57>  ldxw   (R0)#27 ,R16
58>  stxw   (R18)R57 ,R0
59>  ldxw   (R0)#7 ,R16
60>  stxw   (R18)R58 ,R0
61>  add    R0,#8 ,R17
62>  ldxw   (R0)#27 ,R16
63>  stxw   (R18)R59 ,R0
64>  ldxw   (R0)#8 ,R16
65>  stxw   (R18)R60 ,R0
66>  add    R0,#9 ,R17
67>  ldxw   (R0)#27 ,R16
68>  stxw   (R18)R61 ,R0
69>  ldxw   (R0)#9 ,R16
70>  stxw   (R18)R62 ,R0
71>  add    R0,#9 ,R17
72>  ldxw   (R0)#27 ,R16
73>  stxw   (R18)R63 ,R0
74>  ldxw   (R0)#9 ,R16
75>  stxw   (R18)R64 ,R0
76>  add    R0,#1 ,R17
77>  ldxw   (R0)#27 ,R16
78>  stxw   (R18)R65 ,R0
79>  ldxw   (R0)#1 ,R16
80>  stxw   (R18)R66 ,R0
81>  add    R0,#1 ,R17
82>  ldxw   (R0)#27 ,R16
83>  stxw   (R18)R67 ,R0
84>  ldxw   (R0)#1 ,R16
85>  stxw   (R18)R68 ,R0
86>  add    R0,#0 ,R16
87>  ldxw   (R0)#30 ,R17
```

```
88>  subi   R0,#0 ,R16{C}
89>  jmpr   le, #@2@16
90>  @1@15
91>  add    R0,#0 ,R17
92>  ldxw   (R0)#26 ,R16
93>  stxw   (R18)R30 ,R0
94>  ldxw   (R0)#0 ,R16
95>  stxw   (R18)R32 ,R0
96>  add    R0,#0 ,R16
97>  add    R0,#0 ,R16
98>  callx  R31,(R0)subri
99>  ldxw   (R0)#0 ,R16
100> stxw   (R18)R31 ,R0
101> ldxw   (R0)#0 ,R16
102> stxw   (R18)R34 ,R0
103> ldxw   (R0)#29 ,R16
104> ldxw   (R0)#34 ,R17
105> subi   R0,#0 ,R16{C}
106> jmpr   le, #@3@21
107> @2@21
108> add    R0,#0 ,R17
109> ldxw   (R0)#27 ,R16
110> stxw   (R18)R34 ,R0
111> ldxw   (R0)#0 ,R16
112> stxw   (R18)R28 ,R0
113> ldxw   (R0)#0 ,R16
114> callx  R31,(R0)conri
115> stxw   (R18)R1 ,R0
116> subi   R0,#1 ,R16{C}
117> jmpr   ne, #@1@27
118> add    R0,#0 ,R16
119> add    R0,#0 ,R16
120> callx  R31,(R0)subri
121> ldxw   (R0)#0 ,R16
122> stxw   (R18)R31 ,R0
123> ldxw   (R0)#0 ,R16
124> callx  R31,(R0)conri
125> stxw   (R18)R1 ,R0
126> add    R0,#0 ,R17
127> ldxw   (R0)#27 ,R16
128> stxw   (R18)R31 ,R0
129> ldxw   (R0)#0 ,R16
130> stxw   (R18)R28 ,R0
131> @1@27
132> @3@21
133> jmpr   alw, #@2@21
134> @1@34
```

```
182> stxw    (R18)R28 ,R0
183> @2@16
184> jmpr    alw, #@1@15
185> ret     R26, #0
```