U M P R O C E S S A D O R E D I S O N

Luiz Carlos Zancanella

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE POS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSARIOS PARA OBTENÇÃO DO GRAU DE MESTRE EM CIENCIA (M.Sc.)

Aprovada por:

Edil Severiano Tavares Fernandes
(Presidente)

José Lucas Mourão Rangel Netto

Paulo Mário Bianchi França

RIO DE JANEIRO, RJ - BRASIL
DEZEMBRO DE 1983

ZANCANELLA, LUIZ CARLOS

Um Processador Edison [Rio de Janeiro] 1983.

V, 62p., 29,7cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e computação, 1983).

Tese - Univ. Fed. Rio de Janeiro, Fac. Engenharia.

1.Computação I.COPPE/UFRJ
II.Título(série).

A minha esposa Leonita

A minha filha Shelen

Aos meus pais

Domingos e

Maria

AGRADECIMENTOS

Ao professor Edil Severiano Tavares Fernandes, pelas idéias, conhecimentos e paciente orientação ministrada durante o desenvolvimento deste trabalho.

Ao professor Edil Severiano Tavares Fernandes, e a professora Lidia Segre pela concepção do projeto e das principais diretrizes de seu desenvolvimento.

A Miguel Argollo Jr., Sérgio de Mello Schneider e Nelson Quilula Vasconcelos, pelas críticas construtivas e idéias fornecidas.

A equipe do laboratório de Sistemas da COPPE/UFRJ, em especial ao Jean-Michel Nayrac, pela colaboração prestada durante a fase de implementação do projeto.

Aos colegas do Núcleo de Processamento de Dados da Universidade do Rio Grande (Rio Grande - RS), em especial ao professor Carlos Rodolfo Brandão Hartmann pelo apoio e incentivo constante.

A todos os meus familiares que colaboraram e estimularam na execução deste trabalho.

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

A Universidade do Rio Grande, a CAPES e ao CNPq, pelos recursos fornecidos durante meus estudos de mestrado.

SINOPSE

Este trabalho descreve o projeto e a criação de um meio ambiente destinado a execução de programas escritos na linguagem concorrente EDISON.

A criação desse meio ambiente consistiu na construção de um tradutor para a linguagem, bem como na implementação de uma máquina básica que interpreta o código virtual produzido pelo tradutor.

O compilador foi construido em estrutura multipasso, com o objetivo de permitir sua implementação em microcomputadores de memória reduzida. O método utilizado para reconhecer estruturas sintáticas do texto fonte é do tipo RRP LL(1). O compilador possui um algoritmo de correção de erros sintáticos, que produz como resultado um programa sintaticamente correto.

A máquina básica implementada é composta por um conjunto de procedimentos que realizam as funções necessárias para a execução do código virtual. Parte destes procedimentos estão concentrados no núcleo da máquina e são destinados a dar suporte as características de processamento paralelo e desempenham as funções de criação, destruição e escalonamento dos processos concorrentes.

Nossa intenção em produzir um processador para EDISON independente de máquina, foi alcançada graças a decisão de gerar uma forma intermediária como resultado da tradução.

ABSTRACT

This thesis describes the design and the implementation of an environment oriented to the concurrent language EDISON.

The environment consists of a language translator which generates a virtual code as output, and a basic machine responsible for the interpretation process of that code.

The compiler was constructed in a multipasses way in order to facilitate its implementation in microcomputers. The RRP LL(1) parser was adopted and the compiler also includes an algorithm to correct syntatic errors.

The Basic Machine includes a set of procedures realizing the functions specified by the Edison Programs. Some of trere procedures define the Basic Machine Kernel which is responsible for the parallel processing and the execution of the functions concerned to the creation, deletion and schedulling of concurrent processes.

Our desire in producing an EDISON processor that is machine independent was achieved due to the generation of on intermediate form as the result of the translation.

INDICE

CAPITULO IV

CAPITULO V

CAPITULO I

I N T R O D U Ç Ã O

A evolução dos primeiros sistemas operacionais do tipo lote para sistemas operacionais de tempo compartilhado, foi responsável pelo aparecimento de um novo conceito de processamento, visando utilizar de maneira mais efeciente os recursos do sistema, tais como processador, memória, dispositivos periféricos, etc.

Este novo conceito, denominado paralelismo, pode ser melhor descrito através do conceito de processo. Um processo é a ativação ou execução de um programa. Por exemplo, um compilador executando a compilação de programas fontes distintos: embora se tenha um só programa para todos os usuários, tem-se vários processos (um para cada compilação), que refletem as atividades paralelas de cada compilação.

Sistemas Operacionais são exemplos típicos de programas onde o conceito de processo é largamente utilizado para definir as diversas tarefas que o compõem. Existem situações em que os processos são independentes no sentido de que a atividade de um não depende ou interfere na atividade do outro. Eventualmente apesar de independentes eles podem competir pela aquisição de um recurso não compartilhável. Em outros casos, as atividades dos processos não são inteiramente independentes e dois ou mais processos eventualmente precisam se comunicar. Nesse caso dizemos que os processos são concorrentes (ou cooperantes) e é necessário introduzir um mecanismo de comunicação entre eles, também chamado de mecanismo de sincronização.

A competição entre processos concorrentes (ou não) em geral, envolve acesso a regiões da memória de forma compartilhada, regiões estas denominadas de regiões críticas. Devido a natureza dos recursos, este tipo de acesso pode introduzir erros dependentes do tempo e portanto, devem ser controlados para que o resultado final produzido pelo programa com ou sem paralelismo seja o mesmo. Assegurar que apenas um processo de cada vez execute os comandos dessas regiões, é, portanto, de importância fundamental no desenvolvimento de programas concorrentes e é usualmente conhecido como resolver o problema de exclusão mútua entre processos concorrentes.

O conceito clássico de programa constituído de execução estritamente seqüencial, apesar de utilizar elegantes primitivas de sincronização sobre semáforos, tornaram-se inadequados para descrição das atividades intrinsecamente paralelas, já que o simples esquecimento de uma primitiva de sincronização, poderia ocasionar erros em tempo de execução difíceis de serem detectados.

Esta dificuldade de se criar programas concorrentes confiáveis, levou ao desenvolvimento do conceito de monitor [Hoare,17] que concentrou dentro de uma única unidade do programa toda operação sincronizada sobre estrutura de dados compartilhada, com a vantagem de manter a característica de programação estruturada. Esta nova sistemática de desenvolvimento de "software", foi a responsável pelo aparecimento de uma nova geração de linguagens de programação, que suportam tanto modularidade como concorrência [Hoare,17 e Brinch Hansen,15]; linguagens como PASCAL CONCORRENTE [Brinch Hansen,12] e MODULA [Wirth,24], causaram um grande impacto sobre a capacidade de se desenvolver sistemas operacionais e sistemas de tempo-real para mini e microcomputadores.

A partir de então tornou-se comum o emprego de linguagens de alto nível ao invés de código "assembly" no projeto de sistemas complexos. Através dessas linguagens podem ser identificados processos concorrentes como parte de programas, e pode-se programar regiões críticas devidamente sincronizadas entre si. Além da vantagem óbvia de se utilizar linguagens de alto nível, há ainda a possibilidade de se realizar testes em tempo de compilação que auxiliam na manutenção da integridade dos recursos envolvidos.

A linguagem concorrente EDISON [Brinch Hansen,13] foi definida com o objetivo de combinar os significativos ganhos da recente tecnologia de "software", dentro de uma linguagem de programação simples e que suporta construções modulares, tanto para programação seqüencial como para programação concorrente.

A simplicidade de EDISON foi obtida, eliminando alguns conceitos das linguagens PASCAL CONCORRENTE e MODULA. Nestas linguagens, programação modular é suportada especialmente pelos complicados conceitos de processos (que combinam modularidade e execução concorrente) e por monitores (que combinam modularidade e execução sincronizada).

Em EDISON, as características de modularidade, concorrência e sincronização foram separadas e são expressas por construções distintas da linguagem. Processos e monitores podem contudo ser representados pela combinação de comandos concorrentes (isto é; comandos pertencentes a região delimitada pelos comandos "Cobegin" e "End") com módulos, procedimentos e o comando "When" (que é equivalente a região crítica condicional de PASCAL CONCORRENTE e MODULA). Apesar de compacta EDISON é uma linguagem sistemática, que pode ser uma ferramenta muito prática no ensino dos princípios de programação concorrente, bem como no desenvolvimento de "software" para sistemas multiprocessadores.

Essa tese consiste na criação de um meio ambiente, que suporta a linguagem concorrente EDISON. Considerando que o objetivo primário do projeto, está voltado para ensino e pesquisa em computação, ficou decidido então que os programas em EDISON seriam interpretados, ao invés de serem executados diretamente pela hospedeira. A adoção dessa estratégia de implementação, apesar de exigir tempos de UCP (Unidade Central de Processamento) mais longos do que a outra, permite exercer um controle mais fino na interpretação dos programas de usuários ainda inexperientes e/ou em fase de aprendizado da linguagem EDISON.

Para atingir esse objetivo, foi necessário inicialmente desenvolver um compilador para a linguagem EDISON, e depois construir uma máquina básica que interpretasse o código virtual gerado pelo tradutor. O capítulo II apresenta o compilador projetado e implementado neste projeto, suas características principais e técnicas utilizadas no seu desenvolvimento. O capítulo III apresenta o conjunto de instruções idealizadas para a linguagem concorrente EDISON , que corresponde a forma intermediária entre o compilador e a máquina básica apresentada no capítulo IV. No capítulo V apresentam-se os comentários finais deste trabalho.

CAPITULO II

C O M P I L A D O R E D I S O N

2.1 - INTRODUÇAO

Linguagens concorrentes, tais como EDISON são excelentes ferramentas para a construção de compiladores com estruturas em multipassos, pois durante a compilação, diversos passos do compilador podem estar ativos simultaneamente, concorrendo (ou cooperando) para tarefa da compilação. Infelizmente não foi possível construir nosso tradutor dessa forma, em virtude de não possuirmos, ainda, nenhum tradutor EDISON. Assim decidimos implementar uma primeira versão do tradutor no "Burroughs 6700" e a linguagem adotada foi o ALGOL extendido.

Dessa maneira, nosso tradutor foi fragmentado em quatro passos independentes, que são ativados, um de cada vez, por um programa de controle. Cada passo executa uma análise seqüencial do programa texto e produz como saída uma forma intermediária, que será usada como entrada pelo passo seguinte. O último passo realiza, ainda, uma passagem adicional em sua forma intermediária de modo a completar informações pendentes, tais como endereços de "jumps".

A figura 2.1 mostra a organização do compilador. Os três primeiros passos correspondem cada um a um processo de compilação: análise léxica, sintática e semântica; cabendo ao último a tarefa de geração do código virtual.

O programa de controle além de gerenciar a execução dos passos seqüencialmente, tem ainda a função de: criar e deletar os arquivos temporários usados como interface entre os passos e emitir os resultados da compilação.

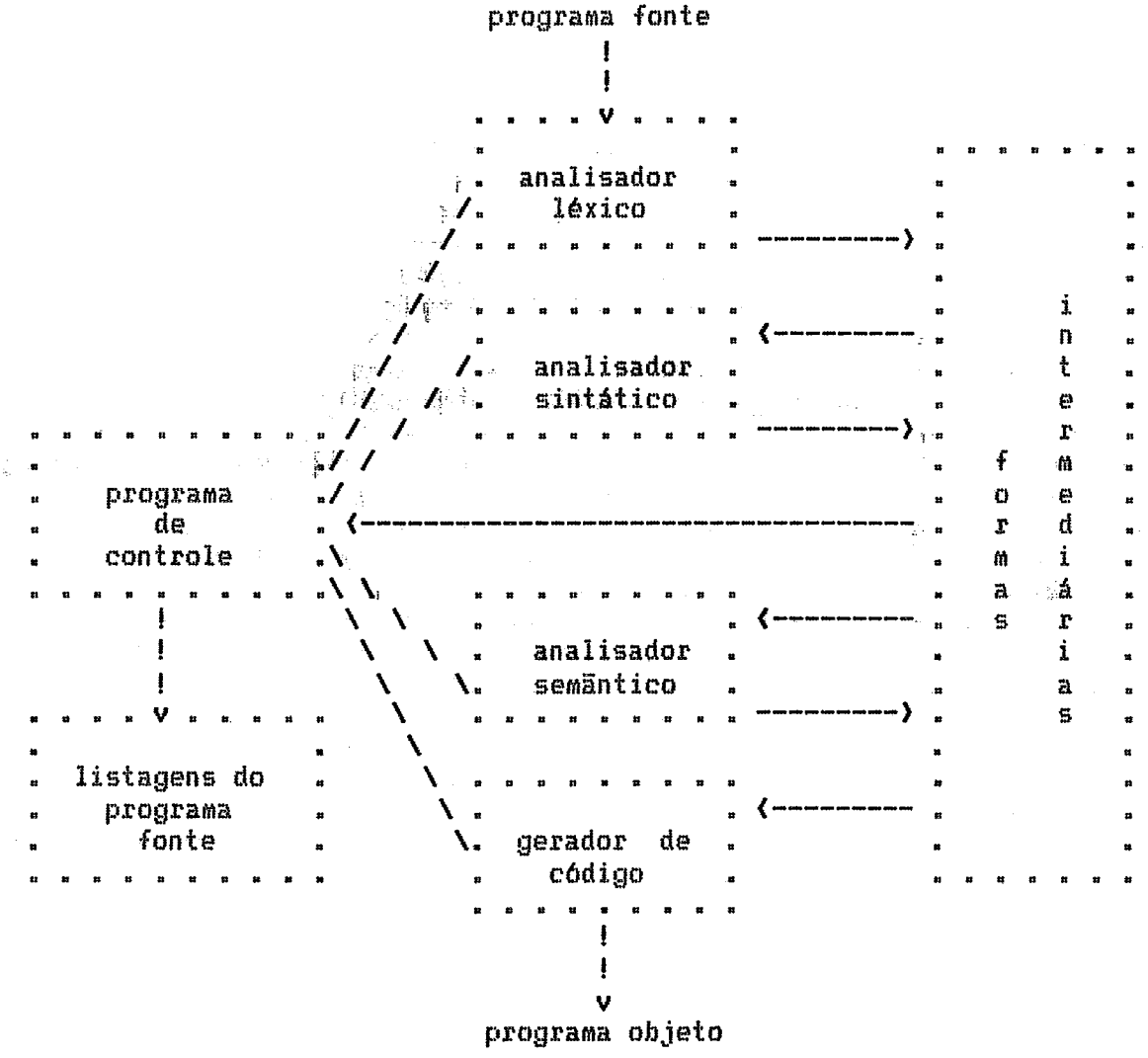


Fig. 2.1 - Organização do compilador

2.2 - ANALISE LEXICA (passo 1)

A análise léxica é a interface entre o programa fonte e o compilador. O analisador léxico lê o programa fonte um caracter de cada vez, e o converte em uma seqüência de inteiros chamados "tokens"; onde cada "token" corresponde a representação interna dos símbolos terminais da gramática (Tab 2.1). Podemos classificar os "tokens" em:

a) Simples:

São os terminais identificados por um inteiro único que representa o tipo do "token".

Por exemplo: IF é representado internamente pelo inteiro 43, DO é representado pelo inteiro 46, virgula por 7, e assim por diante.

b) Compostos:

São terminais identificados por um par de inteiros: tipo e valor do "token".

Por exemplo: VAR CHAVE : BOOL;
O identificador CHAVE da declaração acima poderia ser representado pelo par (2, 255), onde 2 corresponde a representação interna para identificador, e 255 representa o índice de entrada na tabela de símbolos.

Desse modo o passo 1 do compilador, analisa um programa em EDISON e o converte para uma forma intermediária que é uma seqüência de inteiros.

O analisador léxico é composto por dois procedimentos: INICIALIZACAO e SCANNER. Além de atribuir os valores iniciais a algumas variáveis, cabe ainda ao procedimento de INICIALIZACAO, (utilizando um procedimento de pesquisa e inserção), incluir as palavras reservadas e as funções da linguagem na tabela de símbolos. Uma função "hash" [Wirth,25] é utilizada para esta finalidade e quando da ocorrência de colisões, o índice primário recebe um incremento quadrático, fazendo-se então uma pesquisa cíclica na tabela. Tal procedimento não é exclusividade da fase de inicialização já que ele é empregado também no tratamento dos identificadores do programa fonte.

A parte correspondente ao SCANNER [ALG 2.1] faz a conversão do texto fonte em uma forma intermediária que é armazenada em um arquivo temporário em disco. Essa forma intermendiária é descrita concomitantemente com a descrição dos procedimentos do analisador léxico.

```
PROC SCANNER ( PROC READ (VAR CH:CHAR) )
VAR ACABOUFONTE:BOOL; CH:CHAR
BEGIN
   ACABOUFONTE := FALSE;
   WHILE NOT ACABOUFONTE
   DO  READ ( CH );
       IF    CH  IN  'ABC,...,Z'  DO  ANALISE DE NOMES
       ELSE  CH  IN  '012,...,9'  DO  ANALISE DE DIGITOS
       ELSE  CH  IN  '+*(,...,)'  DO  CARACTER ESPECIAL
       ELSE  CH  IN     '(:)'     DO  SIMBOLO ESPECIAL
       ELSE  CH  IN      '''      DO  STRINGS
       ELSE  CH  IN      '"'      DO  COMENTARIO
       ELSE  CH  IN      ' '      DO  BRANCOS
       ELSE  CH  =  EOF           DO  ACABOUFONTE := TRUE
       ELSE  TRUE                 DO  SIMBOLO INVALIDO
       END "IF"
   END "SCANNER"
```

ALG. 2.1: Representação em EDISON do algoritmo básico do analisador léxico.

2.2.1 - PROCEDIMENTOS DO ANALISADOR LEXICO

A seguir descreveremos os vários procedimentos do analisador léxico; estes procedimentos são responsáveis pela transformação dos terminais da gramática em "tokens". Apartir deste ponto trataremos os terminais da gramática simplesmente por "tokens". isto é, pela sua representação interna.

ANALISE DE NOMES

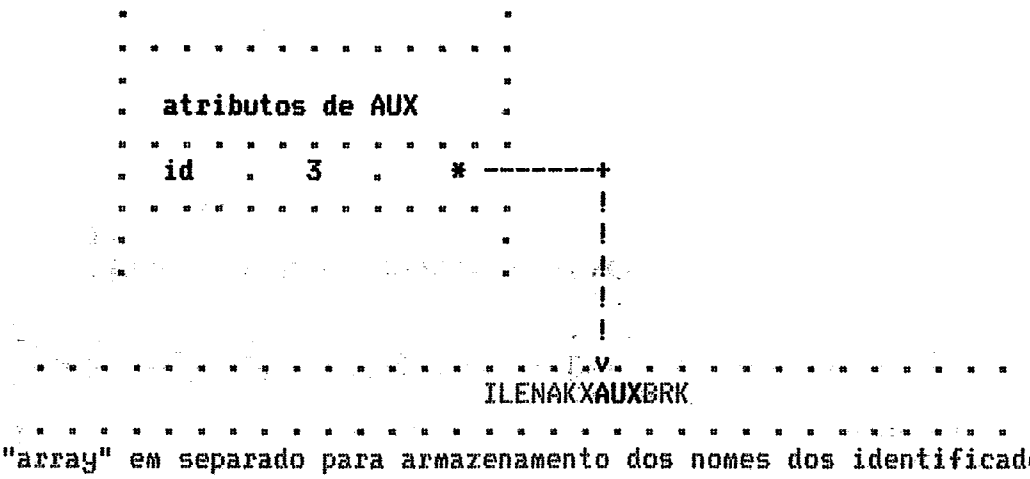
Esse procedimento é responsável pelo tratamento dos identificadores do texto fonte (isto é, palavras reservadas e nomes criados pelo programador). Inicialmente ele extrai a cadeia de caracteres que forma o identificador e avalia - através da função "hash",- o índice da tabela de símbolos correspondente ao identificador. O procedimento de pesquisa e inserção é aqui utilizado para incluir ou recuperar atributos do identificador.

Cada identificador é descrito na tabela de símbolos desse passo (fig.2.2) por três atributos:

1. O código numérico que está relacionado ao identificador.

2. O número de caracteres do identificador.

3. Um apontador para o início da cadeia de caracteres relativa ao identificador, armazenada em um "array" em separado.



"array" em separado para armazenamento dos nomes dos identificadores.

Fig. 2.2 Organização da tabela de símbolos utilizada pelo analisador léxico.

A forma intermediária gerada por este procedimento é formada pelo "token" de identificador e um segundo componente que especifica o índice de acesso a tabela de símbolos, caso este identificador seja uma variável do usuário.

ANALISE DE DIGITOS

As constantes numéricas do programa fonte são extraidas por este procedimento. Uma vez que em EDISON as constantes numéricas devem ser inteiras, precisamos garantir apenas que o resultado da conversão esteja dentro do intervalo [inteiro mínimo,inteiro máximo].

A forma intermediária gerada para constantes numéricas, é representada pelo par (código de constante, valor).
por exemplo:
A constante "1983" seria representada pelo par (15,1983).

CARACTERES e SIMBOLOS ESPECIAIS

Os caracteres especiais reconhecidos pelo procedimento seguem-se: ( ) * + - . ; = [ ]

Os símbolos da linguagem que são formados por dois caracteres são: <> <= >= :=

Tais símbolos ou caracteres são passados para o código intermediária sob a forma do "token" correspondente.

STRINGS

Este procedimento trata de cadeias de caracteres, que são armazenadas em uma tabela de "strings". Desse modo a forma intermediária gerada para representá-las é formada pelo par: (código de string, primeira posição na tabela).

Por exemplo: A cadeia "ISSO E UM EXEMPLO" poderia ser representada pelo par (16, 457), além da seqüência dos caracteres mantidos no "array" em separado.

OUTRAS ANALISES

Caracteres inválidos, brancos e comentários, são tratados respectivamente pelos procedimento: SIMBOLO INVALIDO, BRANCOS e COMENTARIO e não são passados para a forma intermediária.

A forma intermediária conterá ainda uma constante inteira que indicará o fim de uma linha do texto fonte, sendo que esta constante poderá conter códigos de erros que foram encontrados nesta linha.

Uma marca ("EOF") indicará na forma intermediária, fim do texto fonte.

1	CONST	31	IN
2	IDENTIFICADOR	32	+
3	=	33	-
4	;	34	OR
5	ENUM	35	DIV
6	(	36	MOD
7	,	37	AND
8	)	38	VAL
9	RECORD	39	.
10	ARRAY	40	NOT
11	[	41	SKIP
12	:	42	:=
13	]	43	IF
14	SET	44	WHILE
15	literal numérico	45	WHEN
16	'string'	46	DO
17	BEGIN	47	ELSE
18	END	48	COBEGIN
19	PROC	49	ALSO
20	VAR		
21	MODULE	71	CHAR
22	*	72	INT
23	PRE	73	BOOL
24	POST	74	FALSE
25	LIB	75	TRUE
26	<>	76	PLACE
27	<	77	SENSE
28	>	78	OBTAIN
29	<=	79	ADDR
30	>=	80	HALT

Tab 2.1 - Representação interna dos símbolos da gramática

2.3 - ANALISE SINTATICA (passo 2)

Cabe ao analisador sintático verificar se a seqÜência de "tokens" emitidos pelo analisador léxico, está obedecendo as regras estabelecidas pela linguagem fonte.

Por exemplo, se um programa EDISON contém a expressão " A + * B ", após a análise léxica esta expressão aparecerá como a seqÜência de "tokens" " id + * id ". O analisador sintático deve detectar essa situação de erro sobre a seqÜência, uma vez que a presença de dois operadores seguidos viola as regras formuladas pela linguagem EDISON para expressões.

Desse modo o passo 2 do compilador, tem a função de analisar sintaticamente a forma intermediária gerada pelo passo 1. Esse passo inclui também um algoritmo de correção de erros, que será descrito na seção 2.3.2.2.

2.3.1 - O ANALISADOR SINTATICO

Analisadores sintáticos dirigidos por tabelas, além da facilidade de implementação, apresentam a vantagem de serem facilmente adaptáveis a mudanças na gramática, uma vez que toda dependência da gramática está concentrada em tabelas. Além disso, o fato de possuirmos um gerador de analisadores [Teles,Simone,22], levou-nos a utilizar um analisador RRP LL(1), que trata-se de um método de análise sintática descendente, determinístico, em um passo e com um símbolo de "lookahead".

A partir de uma representação em RRP (fig.2.3) para a gramática EDISON [Brinch Hansen,13], o gerador mencionado anteriormente forneceu a tabela de controle para a gramática em forma de lista (fig 2.4), o que permitiu uma representação compacta para a linguagem EDISON (187 elementos).

Cada elemento da lista sintática é representado por três atributos:

1. O primeiro indica se o elemento é terminal ou não. Se o símbolo é terminal então ele é representado pelo respectivo "token", enquanto um não-terminal é representado por um ponteiro a um elemento da lista.

2. Um ponteiro ao elemento sucessor.

3. Uma condição booleana indicando se o próximo símbolo é aceito ou não pelo analisador sintático na configuração atual da análise.

A1 - Programa =
(Declaração de tipo)* Declaração completa de procedimento

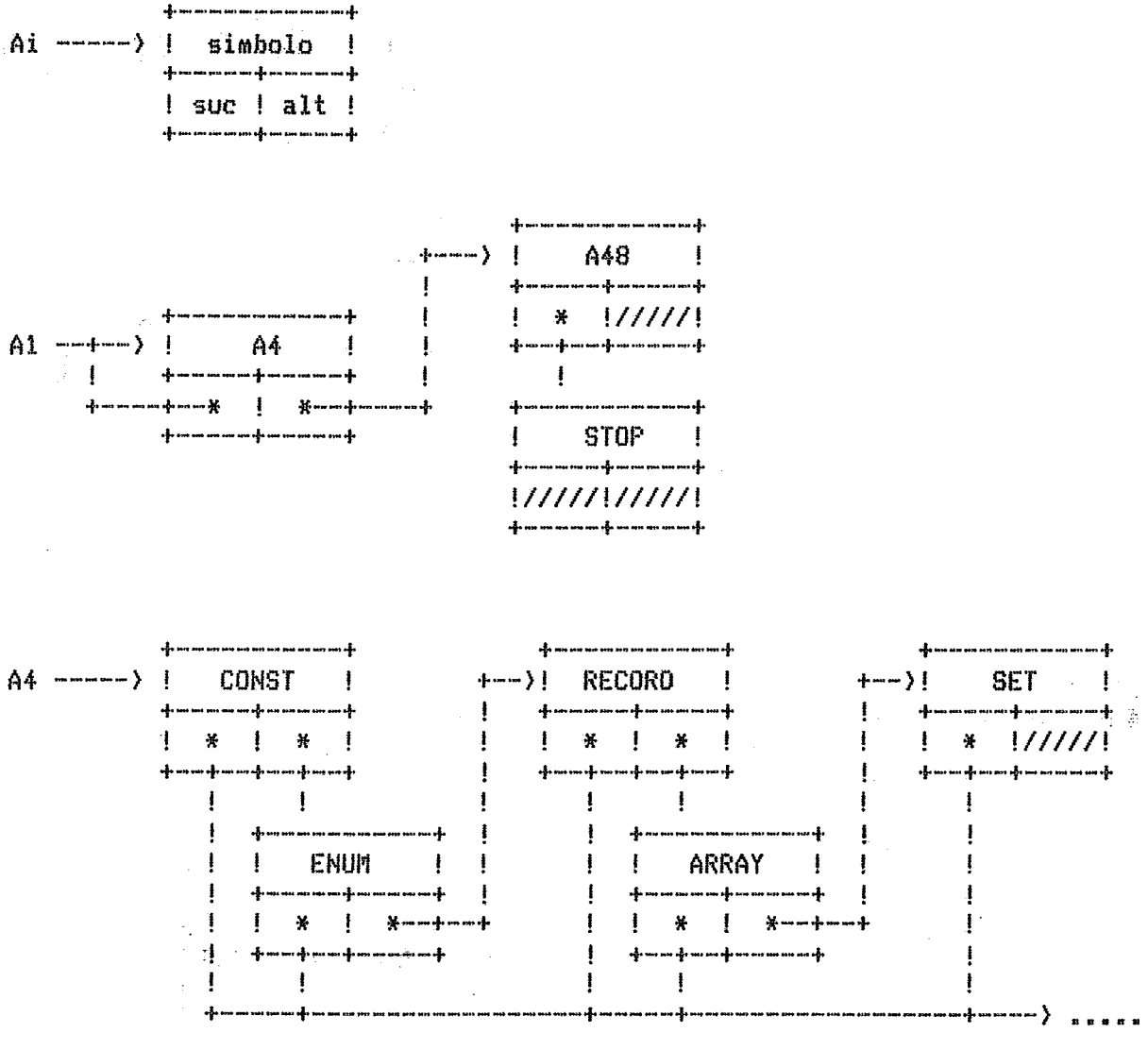
A4 - Declaração de tipo =
(**CONST** # **ENUM** # **RECORD** # **ARRAY** # **SET**) ...

A48 - Declaração completa de procedimento =
Declaração de procedimento (declaração) *
BEGIN (Comando & ;) **END**

A55 - Declaração de procedimento =
PROC ID ('(' Parâmetro ')')? ('*' ID)?

A151 - Comando =
(ID # **VAL** # **COBEGIN** # **IF** # **WHILE** # **WHEN** # **SKIP**).....

Fig 2.3 - Representação de parte da gramática EDISON na forma RRP (lados direitos regulares).



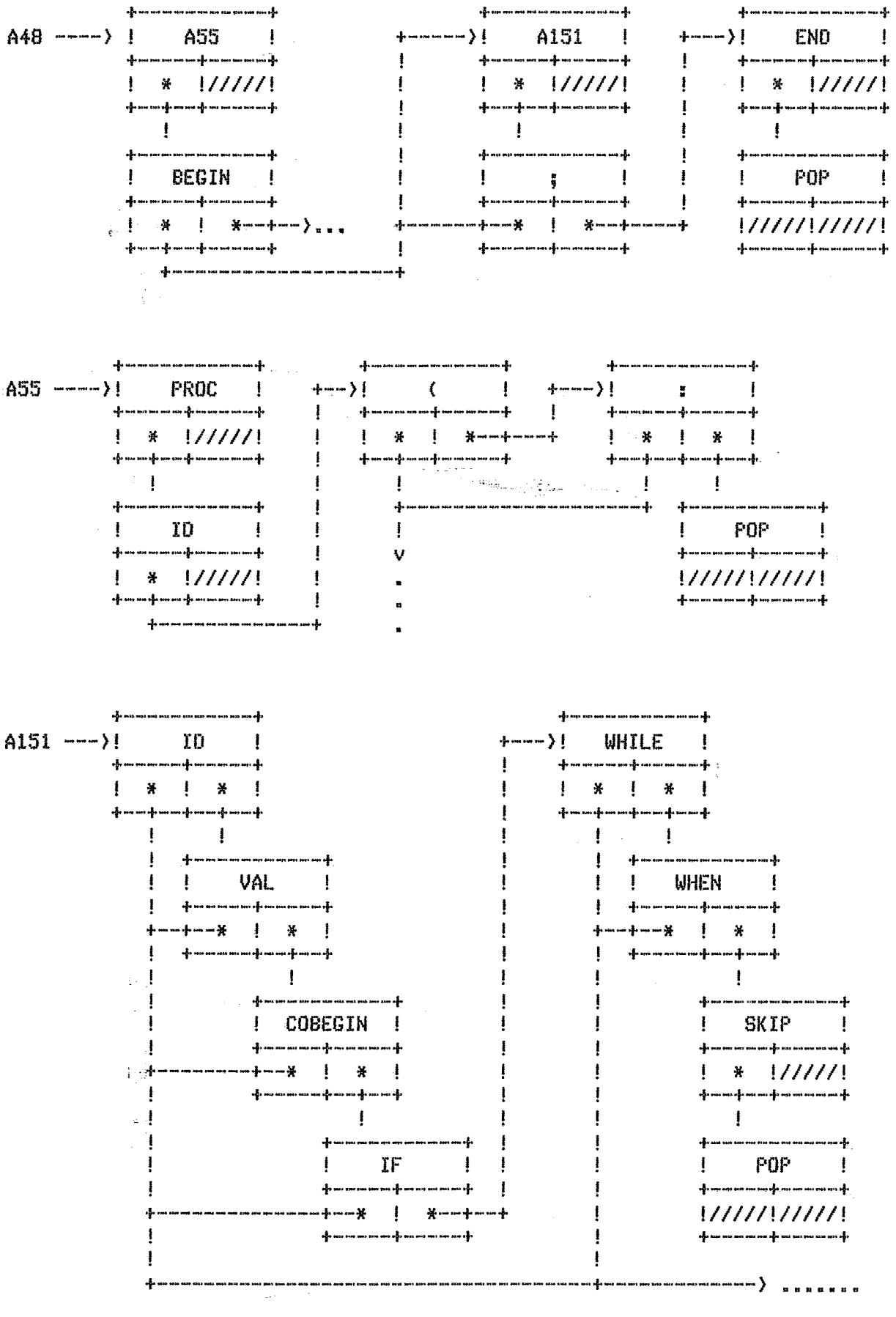


Fig 2.4 - Representação da figura 2.3 na forma de lista.

Analisadores dirigidos por tabela, além da tabela de controle, contém uma pilha explicita para registrar quais os não-terminais da gramática estão ativos correntemente. O analisador é controlado por um algoritmo [ALG.2.2] iterativo (não recursivo), que percorre a lista sintática, casando a entrada do analisador com a gramática EDISON.

```
RECORD ATRIBUTOS ( SIMBOLO,ALT,SUC : INT )
ARRAY TABELA [1:187] (ATRIBUTOS)
PROC PARSER
VAR NO,INPUT : INT ; LISTA : TABELA
BEGIN
    NO := 1;      " nó inicial da primeira produção"
    SCAN;         " coloca símbolo entrante em INPUT"
    WHILE LISTA[NO].SIMBOLO  IN  TERMINAL
          DO IF   LISTA[NO].SIMBOLO = INPUT
                  DO NO := LISTA[NO].SUC;   SCAN
             ELSE TRUE
                  DO  IF   LISTA[NO].ALT  DO  NO := NO + 1
                      ELSE TRUE  DO  ERROSINTATICO
                      END
             END
    ELSE LISTA[NO].SIMBOLO = Ai          "não terminal"
         DO IF   ECOMECO
                 DO PUSH (NO); NO := LISTA[NO].SIMBOLO
            ELSE TRUE
                 DO IF   LISTA[NO].ALT  DO  NO := NO + 1
                    ELSE TRUE  DO  ERROSINTATICO
                    END
            END
    ELSE LISTA[NO].SIMBOLO = POP   "redução de não terminal"
         DO  NO := LISTA[TOPO].SUC;   POP
    END "WHILE"
END "PARSER"
```

ALG.2.2 Algoritmo básico do analisador sintático

O procedimento ECOMECO, é uma função Lógica que verifica se o símbolo da entrada "INPUT", pertence ao conjunto de símbolos terminais iniciadores de uma determinada produção da gramática. Os procedimentos PUSH e POP, servem respectivamente para empilhar e desempilhar os não terminais correntemente ativos. O procedimento ERROSINTATICO, é ativado quando da ocorrência de um erro sintático e tem a função de corrigir o erro.

A seguir apresentamos um exemplo de análise sintática (Tab. 2.2), necessária para reconhecer um programa escrito em EDISON (ALG.2.3).

```
PROC  EXEMPLO
BEGIN
      SKIP
END
```

ALG.2.3 Exemplo de programa escrito em EDISON.

INPUT	NO	LISTA[NO].SIMBOLO	PILHA	PARSER
PROC	1	A4	----	2
"	2	A48	2	
"	48	A55	2,48	
"	55	PROC	2,48	1
EXEMPLO	56	ID	2,48	
BEGIN	57	(	2,48	
"	58	:	2,48	
"	59	POP	2	3
"	49	BEGIN	2	1
SKIP	51	A151	2,51	2
"	151	ID	2,51	1
"	152	VAL	2,51	
"	153	COBEGIN	2,51	
"	154	IF	2,51	
"	155	WHILE	2,51	
"	156	WHEN	2,51	
"	157	SKIP	2,51	
END	167	POP	2	3
"	52	;	2	2
"	53	END	2	
$	54	POP	------	3
$	3	STOP	------	fim

Tab. 2.2 Passos do PARSER para reconhecer a estrutura sintática do programa EDISON EXEMLO.

2.3.2 - CORREÇAO DE ERROS

A forma intermediária gerada por esse passo foi um dos pontos críticos do compilador. Visto que o analisador semântico (próximo passo) somente esperava receber textos fonte com estruturas sintáticamente corretas, foi necessário implementar um algoritmo de correção de erros.

Deve-se observar a diferença entre correção e recuperação de erros: enquanto a recuperação consiste na modificação da configuração do analisador, de modo a retornar a suas funções sem se preocupar em corrigir o erro; a estratégia de correção consiste na transformação de um programa sintáticamente incorreto em um programa correto, pela tranformação da cadeia de entrada a partir do ponto de erro.

Uma das preocupações que tivemos, ao escolher o método de correção de erros é que ele deveria ser confiável no sentido de conseguir tratar qualquer tipo de erro, além disso as correções deveriam ser efetivadas de forma a minimizar a produção de erros espúrios.

Desse modo a forma intermediária gerada por este passo sempre corresponderá a uma estrutura sintaticamente correta. As informações contidas para cada "token" dessa forma intermediária é a mesma gerada pelo passo 1, mais o elemento da lista sintática onde o "token" foi reconhecido.

2.3.2.1 - METODO BASICO DE CORREÇAO

Antes de entrarmos no método proposto, vamos apresentar a notação utilizada. Uma gramática livre de contexto é uma quádrupla G = (Vn,Vt,P,S) onde Vn é o alfabeto de não terminais, Vt é o alfabeto de terminais, V = Vn U Vt é o alfabeto da gramática, P é um sistema de reescrita com produções da forma X --> x, onde X pertence a Vn e x pertence a V*, e S é o símbolo inicial da gramática. A linguagem gerada por G é o conjunto L(G) formado por todo x pertencente a Vt* tal que S gera x. Se z = xy pertence a Vt* então x é um prefixo de z e y é um sufixo de z. Se z pertence a L(G) então x também é considerado como prefixo de L(G). Se z não pertence a L(G), o ponto de erro é o par ordenado (u,v) tal que z = uv e u é o maior prefixo comum entre z e L(G). Em outras palavras, o ponto de erro representa a divisão da cadeia de entrada na posição em que foi encontrado o primeiro símbolo que não permite a continuação da análise sintática.

Para cada ponto de erro (u,v), existe um conjunto W de cadeias tal que z = uw pertença a L(G) e w pertença a W. Tais cadeias são chamadas cadeias de continuação; entre essas cadeias exite ao menos uma de tamanho mínimo, chamada cadeia de continuação mínima. Para cada símbolo de uma cadeia aceito pelo analisador sintático existe um conjunto de símbolos terminais que também seriam válidos se apresentados nas mesmas condições, esses símbolos formam o conjunto de símbolos alternativos dos símbolos aceitos.

Qualquer cadeia de continuação w mostrada no parágrafo anterior resolve o problema de correção de erros sintáticos, porém nem sempre com resultados satisfatórios. O método proposto por Rohrich

[Rohrich,19 e 20] procura combinar um prefixo da menor cadeia de continuação com o maior sufixo possível da cadeia do texto não analisado. O problema de combinação entre as cadeias é resolvido da seguinte maneira: é formado um conjunto H de símbolos de sincronismo de modo que para cada símbolo da cadeia de continuação w são colocados em H seus símbolos alternativos. O trecho da cadeia de entrada não analisado é então examinado a procura de um dos símbolos de sincronismo; se algum for encontrado concatena-se o prefixo de w até o símbolo que provocou o sincronismo com o sufixo da cadeia de entrada apartir desse símbolo e continua-se a análise sintática com a nova cadeia. A figura 2.5 ilustra esse processo.

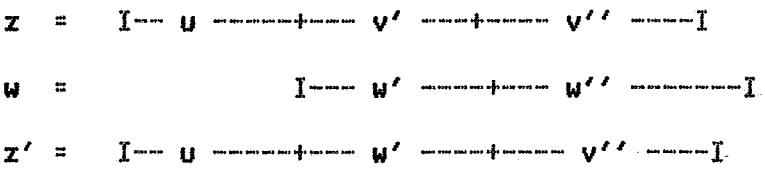


Fig 2.5 Correção do ponto de erro (u,v) da palavra z. Apartir da cadeia de continuação w.

Note que cada vez que o procedimento de correção for chamada pelo menos um novo símbolo da cadeia de entrada original é analisado, o que garante que a análise chegará eventualmente ao fim.

O Algoritmo 2.4 apresenta o método de correção do ponto de erro (u,v) proposto por Rohrich em [Rohrich,20].

1. Determine a menor cadeia de continuação w de u.

2. Calcule o conjunto de símbolos de sincronismo H, onde H é o conjunto formado por todo h pertencente a Vt tal que existe um prefixo w' de w, tal que uw'h é um prefixo de L(G).

3. Percorra v procurando um símbolo de sincronismo h, ou seja, encontre o menor prefixo v' de v tal que v = v'hv'' para algum h pertencente a H.

4. Se tal símbolo existir, insira o menor prefixo correspondente de w a esquerda do símbolo e obtenha a nova cadeia z' = uw'hv''.

5. Se o símbolo não existir, troque o sufixo v pela cadeia de continuação w.

ALG. 2.4 Algoritmo de correção do ponto de erro (u,v) em relação a linguagem L(G).

2.3.2.2 - IMPLEMENTAÇÃO DE ALGORITMO DE CORREÇÃO

O emprego da cadeia de continuação mínima como proposto por Rohrich, pode provocar em determinados casos o final do processo de análise sintática muito antes do final do texto fonte do programa.

Para solucionar esse problema propomos [Argolo e Zancanella,4] uma modificação na geração da cadeia de continuação w; enquanto Rohrich propõe a geração da cadeia de continuação mínima, nossa implementação utiliza o texto fonte não analizado para orientar-se, da seguinte maneira: se em um estado do analisador sintático houver transições com vários terminais e um desses terminais puder ser seguido pelo primeiro símbolo da cadeia de entrada, ele é o escolhido para a cadeia de continuação; caso contrário é escolhido o terminal que gera a menor seqüência de continuação como propõe Rohrich.

O conjunto de símbolos de sincronismo H é representado por um vetor que tem uma entrada para cada símbolo terminal da gramática. Ao se percorrer a lista do analisador sintático para formação da cadeia de continuação w, se um estado tiver transições com vários terminais todos esses são incluidos em H; se houver transições com algum não-terminal, seus símbolos iniciadores também são colocados em H. A inclusão de um símbolo no conjunto H equivale a se colocar em sua entrada correspondente um ponteiro para seus símbolos de sincronismo em w.

Após a geração da cadeia de continuação e do conjunto H, o sincronismo entre as cadeias é realizado e o processo de análise continua com a cadeia formada; a pilha sintática volta para a configuração em que se encontrava quando o último símbolo válido foi reconhecido.

No algoritmo 2.5 é apresentado o algoritmo básico do procedimento de correção de erros que foi implementado.

```
PROC ERROSINTATICO
BEGIN
   WHILE LISTA[NO].SIMBOLO  IN  TERMINAL
         DO  " inclui lista[no].simbolo em H "
             IF  LISTA[NO].ALT
                 DO  IF   LISTA[NO+1].SIMBOLO IN TERMINAL
                          DO IF  EFOLLOW
                                 DO  " inclui lista[no].simbolo em W "
                                     NO := NO + LISTA[NO].SUC
                             ELSE TRUE DO NO := NO + 1
                             END
                     ELSE LISTA[NO+1].SIMBOLO = Ai
                          DO  "inclui conjunto de first de Ai em H "
                              " inclui lista[no].simbolo em W "
                              NO := NO + LISTA[NO].SUC
                     END
             ELSE " inclui lista[no].simbolo em W "
                  NO := LISTA[NO].SUC
             END
   ELSE LISTA[NO].SIMBOLO = Ai              "não terminal"
        DO  PUSH ( NO );  NO := LISTA[NO].SIMBOLO
   ELSE LISTA[NO].SIMBOLO = POP    "redução de não terminal"
        DO  NO := LISTA[NO].SUC;    POP
   END "WHILE"
   " procura símbolo de sincronismo e determina cadeia minima "
END "ERROSINTATICO"
```

Alg. 2.5 Algoritmo básico de correção de erros.

A estrutura de dados utilizada por este algoritmo é a mesma utilizada pelo algoritmo de análise sintática. O procedimento EFOLLOW tem a função de verificar se o terminal corrente pode ser seguido pelo primeiro símbolo da cadeia de entrada.

Como vantagens do algoritmo implementado podemos citar:

a) Conceitualmente o método independe do método de análise sintática.

b) Pelo menos um símbolo da cadeia não analisada é analisado, o que garante que a análise chegará ao fim.

c) Não utiliza nenhuma área de dados adicional, pois gera a cadeia de continuação e o conjunto de símbolos de sincronismo utilizando somente a tabela de controle do analisador.

d) Não penalisa programas corretos.

e) Realiza em geral pequenas inserções e/ou deleções.

Aseguir são apresentados alguns exemplos de recuperação de erros sintáticos efetuados pelo algoritmo de correção implementado.

```
PROC  EX01            " troca ';' por ']' "
   VAR X : INT
   BEGIN
         X := 1 ]   SKIP
))))  CORRIGIDO PARA     X := 1 ; SKIP
   END


PROC  EX02            " troca '[' por ']' "
   ARRAY VET ] 1:5 ] (INT)
))))  CORRIGIDO PARA     ARRAY VET [ 1 : 5 ] ( INT )
   VAR A : VET; X : INT
   BEGIN
        X := A ] 5 ]
))))  CORRIGIDO PARA     X := A
   END


PROC  EX03            " troca 'DO' por 'THEN' "
   VAR B : BOOL
   BEGIN
         IF B THEN SKIP END
))))  CORRIGIDO PARA     IF B ( THEN ) DO SKIP END
   END


PROC  EX04            " falta 'END' "
   PROC INTERNA
      VAR X : INT; Y : BOOL
      BEGIN
            IF  Y  DO  X := X + 1
      END
   BEGIN
))))  CORRIGIDO PARA     ; SKIP END BEGIN
        SKIP
   END


PROC EX05              " falta 'BEGIN'  "
   VAR X : INT; Y : BOOL
         IF  Y  DO  X := X + 1  END
))))  CORRIGIDO PARA     BEGIN IF Y DO X := X + 1 END
   END


PROC EX06             " token extra VAR "
   VAR VAR, X : INT
))))  CORRIGIDO PARA     VAR ID : ID VAR ID, X : INT
   BEGIN
        X := X + 1
   END
```

2.4 - ANALISE SEMÂNTICA (passo 3)

O passo 3 analisa o código intermediário gerado pelo passo 2 e verifica se as regras semânticas da linguagem EDISON estão sendo observadas pelo programa texto. Esse passo é constituído de três tipos de análise:

1. das declarações;
2. de corpos de programas;
3. do alcance dos identificadores.

2.4.1 - ANALISE DE DECLARAÇÕES.

Durante a análise das declarações, o analisador semântico realiza as sequintes tarefas:

a) Verifica se as declarações são consistentes,
b) Avalia os tipos das variáveis e
c) Calcula os endereços das variáveis e parâmetros.

Essas informações estão distribuidas na tabela de símbolos, que nesse passo é totalmente construida pela análise de declarações, e consultada pelas análises de corpo e de alcance. Durante a análise semântica a tabela de símbolos é constituída por três estruturas de dados (fig. 2.6):

1. Estrutura HASH

"Array" cujo tamanho é o número máximo de identificadores que a tabela de símbolos suporta. O acesso a esta estrutura é feito através do índice calculado pela função "hash" no passo 1 e contém um único atributo que é um apontador para a tabela de entradas.

2. Estrutura de ENTRADAS

Pilha contendo cinco atributos das variáveis correntemente ativas:

a) Um apontador para a estrutura de nomes.
b) Um atributo de acesso que indica o estado da variável num dado momento.
c) Nível de aninhamento do bloco.
d) Deslocamento da variável dentro do bloco.
e) Apontador para uma possível ocorrência do mesmo identificador em um bloco mais externo.

As entradas pertencentes ao mesmo nível formam uma lista. Existe um apontador para o seu início, o que permite deletar todas as variáveis locais a um bloco, quando este deixar de existir.

4. Estrutura de NOMES

Pilha que contém as caracteríticas (isto é, os tipos das variáveis) declaradas, onde o primeiro campo identifica o tipo da entidade: CONSTANT, RECORD, ARRAY, SET , etc. Além disso, cada entidade tem um conjunto diferente de atributos:

a) Uma constante é descrita pelo seu valor e um apontador para o seu tipo

b) Um tipo"enumeration" é descrito pelo número de "enumeration symbols" que o compõem seguido pela descrição dos mesmos.

c) Um "enumeration symbols" é descrito pelo seu cardinal e um apontador para o seu tipo "enumeration".

d) Um "record" é descrito pelo somatório dos tamanhos de seus campos (em "bytes"), pelo número de campos de "records" que o compõem e pela descrição de cada um de seus campos.

e) Um campo é descrito por: um apontador para seu ancestral, pelo seu tamanho (em "bytes") e por um apontador para o seu tipo.

f) Um "array" é descrito por: duas constantes que definem seus índices; um apontador para o seu tipo; pelo tamanho de cada componente (em "bytes"); e pela dimensão do "array" em "bytes".

g) No caso da entidade ser um "set", existe um campo que aponta para a descrição do tipo associado ao "set".

h) Uma variável é descrita pelo seu tamanho (em "bytes") e por um apontador para o seu tipo.

i) Uma "procedure" é descrita por: um atributo com valor zero se ela for uma "procedure" propriamente dita, ou um apontador para um tipo se ela for uma função. Em ambos os casos segue-se o número de parâmetros formais, e a da descrição dos mesmos conforme o item j.

j) Parâmetros formais são descritos por: um apontador para a "procedure" onde eles foram declarados, pelo seu tamanho (em "bytes") e por um apontador para o seu tipo.

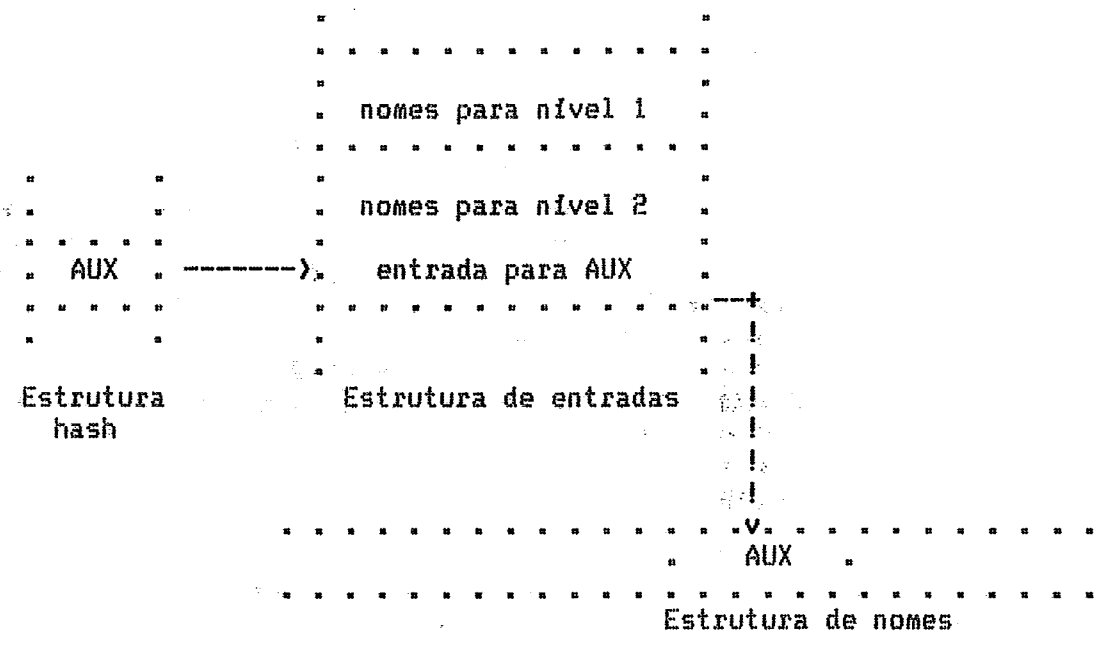


Fig. 2.6 Organização da tabela de símbolos para a análise semântica.

Quando um identificador é declarado, uma entrada é criada na estrutura de entradas e seu ponteiro para essa entrada é armazenada na tabela "hash". Subseqüentes referências ao identificador são processadas indiretamente através dessa tabela. Note que um ponteiro para a entrada ao invés da própria entrada está residente na tabela "hash", isto permite que entradas para a tabela de símbolos sejam alocadas a medida que são declaradas. O problema de declarar mesmos identificadores em diferentes níveis, diz respeito, as regras de alcance da linguagem que serão abordados durante a análise de alcance.

2.4.1.1 - ENDEREÇAMENTO DE VARIAVEIS

Classicamente linguagens com arquitetura em blocos, utilizam um endereçamento consistindo de uma base e um deslocamento, onde a base é o nível de aninhamento do bloco e o deslocamento a distância em relação a esta base.

Desse modo, sempre que um novo procedimento for declarado um novo nível é estabelecido, e empilhado numa tabela de níveis. Este nível deixará de existir, isto é, será descartado da tabela de níveis, quando o procedimento deixar de existir. Os deslocamentos dentro desse novo nível são associados seqüencialmente a parâmentros e variáveis a medida que suas declarações são analisadas, onde o deslocamento de uma variável é sempre o somatório dos tamanhos das variáveis anteriormente declaradas. O mesmo

método é utilizado para calcular o deslocamento de um campo dentro de um "record". Na próxima seção os identificadores das variáveis, constantes e procedimentos são traduzidos para o par (nível, deslocamento).

A análise de declarações, obtém todas as informações que as declarações de tipo contém, conseqüentemente as declarações de tipo não necessitam ser enviados para o próximo passo, já que todas as informações necessárias são transmitidas durante a análise de corpos de programas.

2.4.2 - ANALISE DE CORPOS DE PROGRAMAS

Durante a análise de corpos de programas, a compatibilidade entre operandos e operadores é verificada, as expressões são transformadas para a notação posfixada [Wirth,25] (ou polonesa) e passadas para a forma intermediária. Além disso são verificados os parâmetros reais utilizados na ativação de procedimentos, quanto ao seu número e tipo.

A análise de alcance necessária no corpo de programas, diz respeito a declarações de variáveis e utilização de variáveis exportadas, que será detalhada na próxima seção.

Existem dois tipos de compatibilidade que devem ser verificadas:

a) compatibilidade entre operandos;
b) compatibilidade entre operandos e operadores.

Por exemplo o operador adição (+) requer que os dois operandos sejam compatíveis entre si (a.), e que eles sejam operandos aritméticos (b.). A definição da linguagem EDISON estabelece as regras de compatibilidade que são as seguintes:

1. Os operadores aritméticos **+ - * DIV MOD**, são aplicados a dois operandos do tipo inteiro e resultam um tipo inteiro.

2. Os operadores lógicos **NOT AND OR**, são aplicados a dois operandos do tipo lógico e resultam um tipo lógico.

3. Os operadores relacionais = <> < >= > >=, são aplicados a dois operandos do mesmo tipo e resultam um tipo lógico.

4. O operador **IN** relaciona dois operandos v e x, onde x denota um tipo "set" e v denota um tipo elementar, retornando o valor lógico "true" se v pertencer a x.

A verificação das regras de compatibilidade e a transformação de uma expressão para a notação posfixada, são tarefas executadas simultaneamente, a medida que uma expressão evolui. Para verificar a compatibilidade entre operandos e operadores, foi necessário simular a ordem de execução das expressões. Para isso foi utilizada uma pilha de trabalho que simula as operações realizadas pela máquina básica na avaliação de expressões, que será detalhada no capítulo IV.

Como já foi mencionado anteriormente a forma intermendiária gerada por esse passo é totalmente produzida pela análise de corpos de programa e portanto consistirá de uma seqüência de corpos de programas. As informações contidas nessa forma intermediária é a mesma do passo 1 com algumas exceções:

1. Variáveis serão identificadas pelo seu endereço (nível, deslocamneto).

2. Procedimentos e chamadas de procedimentos serão identificadas por um rótulo.

3. Construtores serão identificados pelo tamanho da sua estrutura (em "bytes").

4. "Begin" (inicio de bloco) conterá o tamanho da área de dados necessária ao bloco iniciado.

5. "Arrays" serão identificados pelos limites inferior e superior e pelo seu tamanho.

2.4.3 - ANALISE DE ALCANCE

Compete a essa análise verificar se o acesso as entidades declaradas pelo programador estão obedecendo as regras de alcance estabelecidas pela gramática EDISON. A verificação dessas regras é realizada durante a análise de declarações e análise de corpos de programas.

EDISON permite, que entidades com mesmo nome, sejam usadas em blocos diferentes, a análise de declarações converte estas possíveis ambigüidades em um único índice para cada ocorrência da entidade. Um bloco é uma parte de um programa texto, que consiste de declarações de entidades e comandos que definem operações sobre estas entidades. Os tipos de blocos possiveis são: programas, módulos e procedimentos.

Geralmente um bloco Q pode ser parte de outro bloco P. O bloco Q é chamado interno de P, enquanto P é chamado circunvizinho de Q. Define-se alcance de um identificador x como sendo o trecho do programa fonte onde x pode ser usado para representar esta entidade.

Em geral o alcance de um identificador extende-se desde a sua declaração até o fim do bloco, com as seguintes regras adicionais:

1. Se a origem é um módulo M e se o alcance for extendido para o fim de um bloco B imediatamente circunvizinho, então esta entidade é dita exportada de M para B.

2. Se uma mesma entidade é declarada com tipos diferentes (isto é, com outro tipo em um bloco mais interno), então o alcance desta entidade não inclui este bloco.

Um certo identificador tem que ser declarado de um único modo em um bloco. E um identificador não pode ser exportado para um bloco circunvizinho no qual o identificador está declarado com outro tipo.

A maneira de usar um identificador em um determinado ponto de um bloco é dada pelas seguintes regras:

1. **Entidades locais**

Se o uso de um identificador é precedido pela declaração de entidade desse identificador no mesmo bloco, então o uso do identificador denota esta entidade.

2. **Entidades exportadas**

Se o uso de um identificador é precedido por um módulo no mesmo bloco a partir do qual a declaração de uma entidade desse identificador é exportada então o uso do identificador denota esta entidade.

3. **Entidades globais**

Se nenhuma das regras acima é aplicável então o identificador denota a entidade do bloco imediatamente circunvizinho.

Um módulo é um bloco que introduz dois tipos de entidades: locais e exportadas. As entidades locais podem somente ser usadas dentro do módulo. As entidades exportadas podem ser usadas tanto dentro do módulo, como no bloco imediatamente circunvizinho.

Para garantir que estas regras estão sendo observadas pelo programa texto, foram criados atributos de acesso a um identificador em um dado nível, associado com o índice do identificador correntemente ativo. Essa estrutura foi mostrada na análise de declarações. Os valores assumidos por esses campos são dinâmicos e o atributo de acesso pode ser de um dos seis tipos abaixo:

a) **acesso autorizado**

O identificador pode ser referenciado seja ele local ou global ao bloco onde o acesso está sendo feito.

b) **acesso indefinido**

Esse é o tipo de acesso que resulta quando um identificador ainda não declarado for referenciado. Desse modo o compilador somente emitirá uma mensagem de erro para cada identificador não declarado.

c) **acesso exportado**

Identificadores com esse tipo de acesso tanto podem ser referenciados no interior do bloco onde foram criados como no bloco imediatamente circunvizinho.

d) **acesso incompleto**

E esse acesso que vigora enguanto a declaração do identificador ainda não foi concluida. Por exemplo, nomes de procedimentos apresentam acesso incompleto até que apareça o "begin". Um tipo não pode fazer referência a si mesmo é outro exemplo.

e) **acesso não resolvido**

Identificadores com esse tipo de acesso somente podem ser referenciados nas listas de declarações. Dentro de uma seqÜência de declarações por exemplo o acesso é não resolvido, passando para autorizado no interior do bloco.

f) **acesso função**

Identificadores desse tipo, somente podem ser referenciados como variável destino no interior do corpo das funções correspondentes.

Como já mencionamos anteriormente, análise de alcance também é realizada durante a análise de declarações, portanto quando um identificador é introduzido as regras de alcance não poderão ser violadas.

Inicialmente todas as tabelas são inicializadas com zero; para o atributo de acesso na tabela de entradas, zero, significa sem atributo. Tentativas de serem introduzidos identificadores com acesso diferente de zero somente serão aceitos se tiverem atributo de acesso autorizado; neste caso a nova entrada conterá um apontador para a entrada antiga e o índice da tabela "hash" passa a apontar para a nova entrada. Recursividade inválida nas declarações de contantes e tipos de dados, são detectadas através do atributo de acesso incompleto, que são trocados para não resolvidos no fim das declarações.

Sempre que o fim de um bloco for alcançado são removidas todas as entidades declaradas locais a este bloco, que são apontadas pela tabela de níveis. Se alguma entidade apontar para outra entrada, seu índice na tabela "hash" passa a conter esse apontador. Se alguma entidade contiver atributo de acesso exportado esta entidade passa para o bloco circunvizinho com acesso autorizado. Caso o procedimento desativado é do tipo função, seu atributo de acesso mudará de função para autorizado. A ocorrência da variável função VAL no corpo de um programa requer atributo função.

Procedimentos "SPLIT" da linguagem, são tratados da seguinte maneira:

a) Quando ocorrer a pré declaração do nome do procedimento, este é incluido na tabela de nomes como sendo um tipo SPLIT.

b) No momento em que for encontrada a declaração completa do procedimento SPLIT, esta deve referenciar um tipo SPLIT na tabela de nomes, que passará a partir de então, a ser um procedimento geral.

c) A ocorrência completa de um procedimento SPLIT em um mesmo bloco é controlada pela análise de alcance, consultando a tabela de níveis quando o bloco deixar de existir.

2.5 - GERAÇÃO DE CODIGO (passo 4)

O último passo do compilador, analisa a forma intermediária gerada pelo passo 3 e completa a transformação do programa texto para o código virtual idealizado [Vasconcelos,23] para este projeto. Na implementação atual o código gerado será interpretado. Contudo, modificando-se esse passo, será possível gerar código para uma hospedeira específica.

No presente gerador duas etapas distintas ocorrem. Na primeira etapa o código é gerado e os endereços de rótulos de procedimentos são definidos. Além disso, é construido uma tabela com endereços pendentes, isto é, que não puderam ser completados em tempo de geração do código. Na segunda etapa é realizada uma passagem no código gerado, preenchendo os endereços pendentes através de uma inspeção a tabela produzida na etapa precedente.

Os endereços pendentes que podem ocorrer são: endereços de procedimentos SPLIT e endereços de desvios; os procedimentos SPLIT têm seus endereços de início definidos quando o procedimento POST for alcançado, enquanto endereços de devios são definidos em tempo de geração do código virtual.

2.5.1 - ESTRUTURA DO CODIGO

Nessa seção decreveremos a maneira como é feita a transformação das estruturas da linguagem EDISON para o código virtual. Isto é, quais instruções de código são geradas para um determinado comando da linguagem EDISON. Como no capítulo seguinte será detalhada a função de cada instrução do código, nessa seção nos limitaremos apenas a mencioná-las sem nos preocupar com as funções das mesmas.

O resultado de um programa EDISON compilado, consiste de uma seqüência de instruções virtuais que representam o programa texto; estas instruções são endereçadas em "bytes" e tem sempre endereço par; sendo que o número de "bytes" necessários a cada instrução depende do tipo e do número de operandos imediatos que ela utiliza.

Os primeiros quatro "bytes" do código são utilizados para indicar o tamanho (em "bytes") do código gerado, e a última instrução do código sempre será uma instrução HALT que tem a função de parar a execução do programa.

Os programas EDISON são compostos por dois tipos de segmentos executáveis: segmento de inicialização de módulos e segmentos de procedimentos. O código de um procedimento começa com uma intrução PROC, que faz as inicializações e é seguida de uma instrução JP (isto é, "jump"), que desvia para o primeiro comando desse procedimento. Caso esse procedimento contenha módulos declarados locais a ele, o operando da instrução JP apontará para o início do segmento de inicialização do módulo, cuja última instrução será um JP para o próximo segmento de módulo ou ao segmento executável do procedimento. Uma instrução RET termina a execução do procedimento, restaurando o contexto do ambiente que fez a chamada ao procedimento, além de liberar o espaço de memória previamente alocado para as variáveis locais e parâmetros.

2.5.2 - REGRAS DE FORMAÇÃO DO CODIGO

Referëcias ou atribuição de valores a variáveis simples são implementadas utilizando-se respectivamente as intruções **PUSH** e **POP** elementares. Caso a variável faça parte de uma estrutura "array", é inicialmente calculado o valor do índice (colocando-o no topo da pilha), seguido da instrução **INDEX** e finalmente são utilizadas as instruções **PUSH** ou **POP** indexadas.

As intruções **PUSH** ou **POP** utilizadas podem ser do tipo local ou global de acordo com o fato da variável ter sido declarada local ou global ao bloco. Caso a variável tenha sido passada como parãmetro do tipo referëncia são utilizadas instruções com endereçamento indireto, sendo que a escolha do tipo local ou global fica, neste caso, condicionada pelo fato da variável ter sido passada como parãmetro tipo referëncia para o bloco local ou para um bloco global.

Referëncias ou atribuição de valores a variáveis estruturadas são implementadas utilizando-se respectivamente as instruções **PUSHS** e **POPS** ("push" e "pop" "string"), que manipulam um número variado de "bytes".

Chamadas de procedimentos são implementados da seguinte maneira:

a) E alocado espaço para seu resultado, se o procedimento for do tipo função (utilizando a instrução **ALOC**);

b) São calculados os valores dos parãmetros passados por valor e colocados no topo da pilha ou colocados os endereços dos parãmetros passados por referëncia utilizando-se as intruções **ADR**. Se um procedimento é passado como parãmetro então o seu endereço relativo é colocado na pilha.

c) Finalmente é realizada a chamada do procedimento utilizando-se uma instrução CALL, ou CALLI se o procedimento tiver sido passado como parãmetro.

Usaremos a forma extendida de Backus-Naur para descrever de maneira sucinta as regras de geração do código virtual para as estruturas da linguagem EDISON.

```
Programa :
        Tamanho    Declaração de procedimento    HALT
Declaração de procedimento :
        PROC   JP    [ Declaração ] *   Lista de comando
Declaração :
        Declaração de procedimento    RET    #
        Declaração de módulo
Declaração de módulo :
        [ Declaração ] *   Lista de comando    JP
Lista de comando :
        Comando   [ Comando ] *
Comando :
        Comando de atribuição  #
        Comando if  #
        Comando while  #
        Comando when  #
        Comando concorrente  #
        Chamada procedimento  #
        Vazio
Comando de atribuição :
        Variável    Expressão      POP
Variável:
        (  Expressão   INDEX  )  #
        Vazio
Comando if :
        Comando condicional  [ Comando condicional ] *
Comando while :
        Comando condicional  [ Comando condicional ] *
Comando when  :
        WHEN    Comando condicional
                    [ Comando condicional ] *    NOT   WAIT
Comando condicional  :
        Expressão    JPFALSE    Lista de comando    JP
Comando concorrente  :
        CREATE    Lista de comando
Chamada de procedimento  :
        [  Argumento  ] *    Procedimento jump  #
        Procedimento standard
Procedimento jump  :
        CALL    #    CALLI
Procedimento standard :
        PLACE    #    ADDR    #    HALT
Argumento  :
        Expressão  #
        Variável     ADR   #
        Procedimento argumento
Procedimento argumento  :
        CODAD
Expressão  :
        Expressão simples  [ Expressão simples  Operador relacional ]
Operador relacional  :
        EQL   #   NEQ   #   LSS   #   GTR   #   LEQ   #
        GEQ   #   IN   #   IS   #   ISNT
Expressão simples  :
        Termo  [  Termo   Operador de adição  ]
Operador de adição :
        ADD   #   SUB   #   OR   #   UNIAO   #   DIFER
```

```
Termo :
        Fator [ Fator  Operador de multiplicação ]
Operador de multiplicação :
        MTPY  #  DIV  #  MOD  #  AND  #  INTERCEP
Fator :
        Constante elementar  #  Constante string  #
        Expressão  #  Construtor  #  Variável PUSH  #
        Chamada de função  #  Fator  NOT
Constante elementar :
        CONSTW
Constante string :
        CONSTS
Chamada de função :
        ALOC  [ Argumento ] *  Procedimento jump  #
        Função standard
Função standard :
        SENSE  #  OBTAIN
Construtor :
        Construtor elementar  #  Construtor de record  #
        Construtor de array  #  Construtor de set
Construtor elementar :
        Expressão
Construtor de record :
        Expressão [ Expressão ] *
Construtor de array :
        [ Expressão [ CONSTS ] ]* ALOC  PACK
Construtor de set :
        EMPTYSET  [ Expressão ] *  [ INCLUDE ]
```

2.5.3 - OTIMIZAÇAO DE CODIGO

Esse passo iclui ainda um procedimento de otimização do código gerado. A otimização consiste numa tentativa de aperfeiçoamento em tempo e/ou espaço do programa objeto.

Nesta implementação a otimização é efetuada simplesmente pela eliminação de instruções redundantes. Sempre que uma instrução gerada puder ser combinada com a instrução precedente, o otimizador reduz estas duas instruções para uma única.

Aseguir apresentamos um exemplo de programa EDISON (Alg 2.6), o código virtual gerado para representá-lo (Fig 2.7), e o código otimizado (Fig 2.8).

```
ARRAY  AR [ 1:10 ] (CHAR)
PROC  EXEMPLO
     VAR A : AR
     BEGIN
          A := AR ( 'E','X','E','M','P','L','O' )
     END
```

Alg 2.6 Exemplo de programa EDISON

```
ADR      INST          OPERANDOS

 4       PROC           0     10     0
12         JP           4
16         CONSTS       2              ' E'
22         CONSTS       2              ' X'
28         CONSTS       2              ' E'
34         CONSTS       2              ' M'
40         CONSTS       2              ' P'
46         CONSTS       2              ' L'
52         CONSTS       2              ' O'
58         ALOC         6
62         PACK        10
66         POPSL       10     18
72       HALT
```

Fig 2.7 - Código gerado para o Alg 2.6

```
ADR      INST          OPERANDOS

 4       PROC           0     10     0
12         CONSTS      14              ' E X E M P L O'
30         ALOC         6
34         PACK        10
38         POPSL       10     18
44       HALT
```

Fig 2.8 - Código da figura 2.7 otimizado

Observe que a instrução JP da figura 2.7 está transferindo o controle para a instrução imediatamente seguinte, e conseqüentemente pode ser eliminada. Duas instruções CONSTS seguidas podem ser substituidas por uma única que manipula as "strings" de ambas.

2.6 - MANIPULAÇÃO DE ERROS

A manipulação de erros não se constitui propriamente em uma fase do compilador, porém é uma ferramenta que deve ser ressaltada. Além de informar ao programador o diagnóstico dos erros encontrados, é conveniente que as mensagens de erros procurem orientar sua correção.

Conforme mencionamos na seção 2.2, os erros diagnosticados por um passo do compilador, são inseridos na forma intermediária gerada por este passo juntamente com a indicação de fim de linha. As mensagens de erros são emitidas pelo programa de controle no final do processo de compilação.

Cada passo do compilador detecta uma classe diferente de erros, e portanto podemos classificar os erros em:

a. Erros dectados pelo analisador léxicos

Cadeia de caracteres muito grande:

O número total de caracteres de um identificador ou de uma "string" excede o limite estabelecido.

Constante inteira inválida:

Um literal numérico está fora do intervalo -32768 a 32767.

Caracter inválido:

Uso de um caracter não aceito pela gramática EDISON.

Excedeu o limite de identificadores:

O número de entidades declaradas excede o limite da tabela de símbolos.

b. Erros detectados pelo analisador sintático

Sintaxe inválida, sentença corrigida para:

A sintaxe da sentença está incorreta. Neste caso o compilador informa a ação tomada pelo algoritmo de correção de erros para corrigir a sentença.

Fim de arquivo não esperado:

O início de um programa não começa com uma declaração prefixada, ou o analisador sintático chegou ao fim sem encontrar fim do programa, ou faltando analisar trechos do programa.

c. Erros detectados pelo analisador semântico

Uso recursivo de um identificador inválido:

Um identificador introduzido por uma declaração de constante ou declaração de tipo está definindo ela própria.

Identificador não declarado

Um identificador antes de ser referenciado em um comando deve aparecer na lista de declarações.

Uso ambíguo de um identificador

Um identificador foi declarado com mais de um tipo em um mesmo bloco, ou uma entidade está sendo exportada para um bloco onde o mesmo identificador foi declarado.

Tipo inválido

O tipo de um identificador é inválido no contexto em que ele é usado.

Procedimento "split" inválido

Um bloco contém uma pré-declaração de um procedimento "split" e não contém a pós-declaração (ou vice versa).

Procedimento parâmetro inválido

Um procedimento declarado como parâmentro de outro procedimento não está dentro da especificação da gramática.

Intervalo de um "array" inválido

O limite inferior de um "array" é maior que o limite superior.

Chamada inválida de um procedimento

Os argumentos utilizados na chamada de um procedimento são diferentes em número e/ou natureza dos parâmentros declarados.

Operando inválido na expressão

Um operando da expressão não está de acordo com a sua declaração.

Chamada inválida de uma função

Os argumentos utilizados na chamada da função são diferentes em número e/ou natureza dos parâmetros da declaração.

Uso inválido de um "record"

O uso de um campo de um "record" não está declarado como tal.

Uso inválido de um "array"

Indíce de um "array" referenciado com tipo diferente do declarado.

Expressão lógica inválida

Neste ponto somente pode ser usado expressões lógicas.

Operador e/ou operando incompatível

Expressão contém operandos de tipos diferentes, ou operadores não podem ser aplicados a operandos deste tipo.

Construtor inválido

O número de expressões em um construtor de "record" (ou de "array") é diferente do especificado na declaração.

O algoritmo 2.7 apresenta um programa EDISON compilado, com alguns exemplos de erros detectados pelo compilador.

```
     C O M P I L A D O R    E D I S O N        ((((()))))      VERSAO  01.83

    1     CONST LAST = '.'; NL = CHAR (10); DATA = 121983
((((((((  1  ))))))))   CONSTANTE INTEIRA INVALIDA
    2
    3     PROC COPIER ( PROC READ (VAR C:CHAR); PROC WRITE (C:CHAR) )
    4
    5       MODULE
    6          VAR SLOT : CHAR;   FULL : BOOL
    7
    8         *PROC SEND (C : CHAR)
    9          BEGIN   WHEN NOT FULL DO SLOT := C; FULL := TRUE END   END
   10
   11         *PROC RECEIVE (VAR C : CHAR)
   12          BEGIN   WHEN FULL DO C := SLOT; FULL := FALSE END   END
   13
   14       BEGIN   FULL := FALSE   END
   15
   16       PROC PRODUCER
   17       VAR X : CHAR
   18       BEGIN
   19          READ ( X );
   20          WHILE  Z <> LAST  DO
((((((((  2  ))))))))   IDENTIFICADOR NAO DECLARADO
   21                  SEND ( X );   READ ( X );
   22          END;
   23          SEND ( X,X )
((((((((  3  ))))))))   CHAMADA INVALIDA DE UM PROCEDIMENTO
   24       END
   25
   26       PROC CONSUMER
   27       VAR Y : CHAR
   28       BEGIN
   29          RECEIVE ( Y );
   30          WHILE  Y <> LAST  DO
   31                 WRITE ( Y ));  RECEIVE ( Y )
((((((((  4  ))))))))   SINTAXE INVALIDA, SENTENCA CORRIGIDA PARA
))))   WRITE ( Y ) ; RECEIVE ( Y )
   32          END;
   33          WRITE ( Y );  WRITE ( NL )
   34       END
   35
   36    BEGIN
   37       COBEGIN  1  DO  CONSUMER
   38         ALSO  2  DO  PRODUCER
   39       END
   40    END


================================================================================
NUMERO DE ERROS DETECTADOS ..............................  =    4
TAMANHO DO PROGRAMA FONTE ...............................  =   40  REGISTROS
================================================================================
```

ALG 2.7 – Exemplo de listagem emitida pelo compilador.

CAPITULO III

C O D I G O V I R T U A L

3.1 - INTRODUÇÃO

Nesse capítulo descreveremos o conjunto de instruções idealizadas em [Vasconcelos,23] para a linguagem de programação concorrente EDISON. O compilador descrito no capítulo II, transforma programas textos em EDISON para uma seqüência dessas instruções conforme descrito no ítem 2.4.

Uma das vantagens de se definir um conjunto de instruções virtuais na implementação de uma linguagem de programação, ao invés de utilizar o código real da máquina hospedeira, é o grau de portabilidade que o sistema adquire. Esta portabilidade pode ser caracterizada pelas seguintes facilidades:

1. O compilador pode ser escrito ignorando-se propriedades de computadores específicos, além de possuir uma geração de código simples.

2. Apenas o interpretador precisa ser reescrito para que o sistema funcione em um ambiente diferente.

3. Poderão ser usadas as características do novo computador para reescrever o interpretador, e deste modo o código intermediário derivado de programas em EDISON pode ser executado eficientemente.

4. O processo de interpretação, apesar de ser mais lento, permite exercer um controle mais fino nos programas de usuários, o que facilita a localização de erros em tempo de execução.

De modo a facilitar o entendimento do código virtual, as instruções foram divididas em grupos segundo as suas funções, conforme descreveremos no item 3.4.

3.2 - FORMATO DAS INSTRUÇÕES

Cada instrução do código virtual é formada pelo seu código de operação (que ocupa um "byte"), seguido ou não por operandos imediatos. A seguir são descritos estes operandos e suas funções.

1. **TAMANHO** (siz)

   Este operando destina-se a informar o número de "bytes" de uma cadeia ou região de memória. Ocupa dois "bytes".

2. **FATOR** (fat)

   Tal operando, é utilizado somente para calcular índices de "array", e destina-se a informar o número de "bytes" de um elemento do "array". Ocupa dois "bytes".

3. **DESLOCAMENTO** (dsp)

   Este operando destina-se a informar o número de "bytes" existente entre uma base e a posição de memória ocupada por uma determinada variável. No caso de tratar-se de uma instrução de desvio este operando informa o deslocamento (endereço relativo) existente entre um determinado endereço real e o ponto onde a instrução é gerada. Ocupa três "bytes".

4. **NIVEL** (niv)

   Indica qual a base que deve ser utilizada para a formação do endereço efetivo de uma variável. Ocupa dois "bytes".

3.3 - REGISTRADORES DA CPU

Durante a execução de um programa EDISON o código e as variáveis usadas são colocadas na memória. Além disto é utilizado o seguinte conjunto de registradores:

1. **Apontador de programa. (PC)**

   E um registrador de controle que destina-se a armazenar o endereço da próxima instrução a ser executada. Este registrador é atualizado somando-se o número de "bytes" utilizados pela instrução executada, ou por um endereço especificado numa operação de desvio.

2. **Apontador de pilha. (SP)**

Destina-se a permitir a implementação de uma pilha, na qual dados e endereços podem ser armazenados e recuperados. Este registrador aponta sempre para o topo da pilha.

3. **Base local. (BASE)**

Destina-se a armazenar o endereço de uma base, que corresponderá sempre ao início da área de dados do nível correntemente ativo.

4. **Endereço da tabela de base. (BT)**

Usado para referenciar a tabela de bases, que será utilizada para armazenar o valor das bases de dados dos procedimentos ativados (fig 4.6).

3.4 - CONJUNTO DE INSTRUÇOES

A seguir é apresentado uma descrição detalhada dos vários grupos que formam o conjunto de instruções utilizadas por este processador da linguagem EDISON

3.4.1 - GRUPO DE TRANSFERENCIA DE DADOS

As instruções deste grupo são utilizadas para transferir dados de uma variável para o topo da pilha através de instruções do tipo **PUSH**, ou do topo da pilha para a variável através de instruções do tipo **POP**. Estas instruções são endereçadas relativamente a: (L) base local, ou (G) uma base global ao nível correntemente ativo, podendo ainda este endereçamento ser: (I) indireto no caso da variável ter sido passada como parâmetro e/ou (X) indexado no caso desta variável manipular arranjos. Estas instruções sempre atualizam o apontador de programa. Podemos distinguir três variantes deste grupo de instruções:

a) Instruções que manipulam o conteúdo de um "byte".

PUSHBL	**POPBL**	⟨ dsp ⟩
PUSHBG	**POPBG**	⟨ dsp ⟩,⟨ niv ⟩
PUSHBLI	**POPBLI**	⟨ dsp ⟩
PUSHBGI	**POPBGI**	⟨ dsp ⟩,⟨ niv ⟩
PUSHBLX	**POPBLX**	⟨ dsp ⟩
PUSHBGX	**POPBGX**	⟨ dsp ⟩,⟨ niv ⟩
PUSHBLXI	**POPBLXI**	⟨ dsp ⟩
PUSHBGXI	**POPBGXI**	⟨ dsp ⟩,⟨ niv ⟩

b) Instruções que manipulam o conteúdo de dois "bytes".

PUSHWL	**POPWL**	⟨ dsp ⟩
PUSHWG	**POPWG**	⟨ dsp ⟩,⟨ niv ⟩
PUSHWLI	**POPWLI**	⟨ dsp ⟩
PUSHWGI	**POPWGI**	⟨ dsp ⟩,⟨ niv ⟩
PUSHWLX	**POPWLX**	⟨ dsp ⟩
PUSHWGX	**POPWGX**	⟨ dsp ⟩,⟨ niv ⟩
PUSHWLXI	**POPWLXI**	⟨ dsp ⟩
PUSHWGXI	**POPWGXI**	⟨ dsp ⟩,⟨ niv ⟩

c) Instruções que manipulam o conteúdo de um número arbitrário de "bytes". (isto é. "strings")

PUSHSL	**POPSL**	⟨ siz ⟩,⟨ dsp ⟩
PUSHSG	**POPSG**	⟨ siz ⟩,⟨ dsp ⟩,⟨ niv ⟩
PUSHSLI	**POPSLI**	⟨ siz ⟩,⟨ dsp ⟩
PUSHSGI	**POPSGI**	⟨ siz ⟩,⟨ dsp ⟩,⟨ niv ⟩
PUSHSLX	**POPSLX**	⟨ siz ⟩,⟨ dsp ⟩
PUSHSGX	**POPSGX**	⟨ siz ⟩,⟨ dsp ⟩,⟨ niv ⟩
PUSHSLXI	**POPSLXI**	⟨ siz ⟩,⟨ dsp ⟩
PUSHSGXI	**POPSGXL**	⟨ siz ⟩,⟨ dsp ⟩,⟨ niv ⟩

Além disto estão previstas duas instruções que permitem atribuir valores constantes ao topo da pilha. A primeira atribui um valor de dois "bytes", enquanto a segunda permite mover cadeias pré-definidas de tamanho arbitrário para o topo da pilha:

```
CONSTW      ( valor da constante )
CONSTS      ( siz )
```

3.4.2 - GRUPO PARA OBTENÇÃO DE ENDEREÇOS

São instruções destinadas a criar facilidades para obtenção e colocação de endereços absolutos de variáveis no topo da pilha. As considerações sobre endereçamento de instruções feitas ao grupo anterior são válidas também neste grupo. Estas instruções são as seguintes:

```
ADRL      ( dsp )
ADRG      ( dsp ),( niv )
ADRLI     ( dsp )
ADRGI     ( dsp ),( niv )
ADRLX     ( dsp )
ADRGX     ( dsp ),( niv )
ADRLXI    ( dsp )
ADRGXI    ( dsp ),( niv )
```

Além disso está prevista uma instrução cuja função é obter um endereço absoluto apartir de um endereço relativo, isto é, a distância que existe até o endereço absoluto que se quer atingir. Está instrução é utilizada particularmente na chamada de procedimentos que tenham como parâmetros outros procedimentos.

```
CODAD     ( dsp )
```

3.4.3 - GRUPO DE OPERADORES

Este grupo de instruções substitui no topo da pilha os valores de um ou dois operandos, pelo resultado da operação realizada sobre estes mesmos operandos.

A maioria destas instruções são realizadas sobre o conteúdo de um número fixo de "bytes", e portanto não necessitam de nenhuma informação adicional.

3.4.3.1 - OPERADORES ARITMETICOS

As instruções aritméticas são aplicadas a dois operandos do tipo, resultando também é do tipo inteiro. Estes operadores executam as operações de soma, subtração, multiplicação, quociente e resto da divisão.

Os mnemônicos utilizados para este grupo de instruções são respectivamente: **ADD**, **SUB**, **MTPY**, **DIV** e **MOD**.

3.4.3.2 - OPERADORES LOGICOS

As instruções lógicas **NOT**, **AND**, e **OR** executam as operações de negação, disjunção e conjunção respectivamente. Tais operadores são aplicados a um ou dois operandos do tipo lógico e produzem um resultado do tipo lógico.

3.4.3.3 - OPERADORES RELACIONAIS

As instruções do código virtual **EQL**, **NEQ**, **LSS**, **LEQ**, **GTR** e **GEQ** comparam dois operandos verificando se eles são iguais, diferentes, etc.

Estes operadores são aplicados a dois operandos de um mesmo tipo elementar (inteiro, lógico, caracter ou tipo enumeração) e produzem sempre um resultado do tipo lógico.

3.4.3.4 - OPERADORES SOBRE CONJUNTOS

As instruções **UNION**, **INTERCEP** e **DIFFER** executam as operações de união, diferença e interseção respectivamente.

Estes operadores são aplicados a dois conjuntos do mesmo tipo, e produzem um conjunto deste tipo.

Uma instrução relacional **IN** é prevista para ser aplicada a dois operandos v e x, onde x representa um conjunto de um dado tipo T e v representa um tipo elementar. A relação v **IN** x é verdadeira se v é membro do conjunto x, e falso caso contrário.

Além disso estão previstas duas instruções que permitem executar as operações de igualdade e diferença sobre cadeias contendo um número arbritário de "bytes".

IS ‹ siz ›
ISNT ‹ siz ›

3.4.4 - GRUPO DE DESVIO

As instruções deste grupo destinam-se a alterar a seqüência normal do fluxo do programa; estes desvios podem ser de dois tipos: condicional e incondicional.

3.4.4.1 - DESVIO INCONDICIONAL

Estes grupo de instruções simplesmente modifica o conteúdo do registrador de controle (PC).

a) **JP** ‹ dsp ›

Essa instrução é utilizada para provocar desvios, sendo o controle transferido para a instrução cujo endereço é dado pela soma do valor atual do registrador de controle (PC) com o deslocamento especificado pelo operando.

b) **CALL** ‹ dsp ›

Esta instrução é utilizada para chamar de forma direta os procedimentos; isto é, o controle é transferido diretamente para a instrução cujo endereço é dado pela soma do valor atual do registrador de controle com o valor especificado pelo operando. Além disto, ela coloca no topo da pilha o endereço de retorno.

c) **CALLI** ‹ dsp ›,‹ niv ›

Cabe a tal instrução chamar procedimentos de forma indireta; Isto é, o endereço de transferência de controle é obtido através de um endereço relativo dado pelos operandos. Esta instrução é utilizada sempre que se deseja chamar procedimentos passados como parâmentro de outro procedimento. A colocação do endereço de retorno no topo da pilha também é uma tarefa dessa instrução.

d) **RET** ‹ dsp ›,‹ niv ›

Esta instrução é utilizada para retornar de procedimentos. O controle é transferido para o endereço de retorno colocado na pilha pela instrução que chamou o procedimento. Além disso ela serve para recuperar o valor da base local, atualizar a tabela de bases e liberar o espaço da memória.

3.4.4.2 - DESVIO CONDICIONAL

Este tipo de operação altera o conteúdo do registrador de controle (PC) conforme o valor contido no topo da pilha.

a) **JPFALSE** (dsp)

Se o valor contido no topo da pilha for o valor lógico falso, o controle é transferido para a instrução cujo endereço é dado pela soma do valor atual do registrador de controle com o deslocamento especificado pelo operando, caso contrário, o fluxo do programa segue normalmente.

3.4.5 - GRUPO DE MANUTENÇÃO DA PILHA

Tal grupo de instruções tem a função de realizar exclusivamente modificações na pilha e nos registradores.

a) **PROC** (siz),(dsp),(niv)

Esta instrução é utilizada no início do código de um procedimento e destina-se a informar o nível de aninhamento deste procedimento e a quantidade de memória necessária aos seus parâmetros e as suas variáveis locais. Tal instrução atualiza a posição da tabela de bases relativa ao nível de aninhamento do procedimento, salvando seu valor anterior no topo da pilha, juntamente com o nível atualizado. Além disto são atualizados os valores da base local e do endereço da tabela de bases, sendo que seus valores anteriores são, também, salvos no topo da pilha.

b) **ALOC** (siz)

Essa instrução do código virtual reserva uma região no topo da pilha contendo brancos. Ela é empregada também para reservar espaço no topo da pilha, destinada a armazenar o resultado da avaliação de um procedimento do tipo função.

c) **EMPTYSET**

Cria um conjunto vazio no topo da pilha, e o registrador que aponta para o topo da pilha é atualizado de modo a reservar espaço para o conjunto, cujo o conteúdo é inicializado com zeros.

d) **PACK** (siz)

Esta instrução serve para compactar arranjos de caracteres ou valores lógicos.

e) **INDEX** (siz),(fat),(dsp)

Tal instrução serve para avaliar índices absolutos de arranjos. Além disto é verificado se o índice calculado está dentro do intervalo estabelecido pelos limites inferior e superior do arranjo.

3.4.6 - GRUPO DE PROCESSOS

Compete a estas instruções a tarefa de criação, destruição e sincronização de processos. Esse grupo de instruções é executado pelo núcleo da máquina básica, enquanto as instruções precedentes são realizadas pelo interpretador.

a) **CREATE** (sz),(dsp)

Esta instrução é gerada pela tradução de um comando "cobegin", e é utilizada para criar processos e alocar espaço na pilha para área de trabalho do processo. Além disso ocorre a transferência do controle para o endereço que é dado pelo soma do valor atual do registrador de controle (PC) com o deslocamento especificado pelo operando (dsp).

Conforme mencionado em 4.2.1 depois da execução de uma instrução de criação de processos, as únicas possíveis instruções a serem executadas pelo processo pai serão as de criação de processos. No momento que o processo pai tenta executar uma instrução que não seja de criação de processo ele ficará bloqueado e iniciar-se-á a execução dos processos filhos. O processo pai será reativado somente quando acabar a execução de todos os processos filhos. Caso um processo filho tente utilizar uma instrução de criação de processos, ele será abortado, conforme exigido pela especificação da linguagem.

b) **HALT**

A instrução Halt serve para informar fim de execução de um processo. Quando o núcleo encontra esse comando, o processo que o executou e' destruido. Caso o processo que está sendo destruido seja o último processo filho, o processo pai será ativado.

c) **WHEN e WAIT**

Estas duas instruções estão associadas ao bloqueio de processos. A primeira coloca um valor falso no topo da pilha e é usada para informar que o processo precisa entrar em uma região crítica. A segunda testa se o valor contido no topo da pilha é verdadeiro, permitindo, neste caso que o processo continue executando ou fique bloqueado caso contrário.

3.4.7 - GRUPO ESPECIAL

Tais instruções permitem modificar, ler ou comparar o conteúdo de uma posição de memória endereçada de forma absoluta. Elas foram introduzidas na linguagem de forma a permitir a execução de operações de entrada e saída, bem como para formar facilidades de se interrogar o estado de um periférico.

a) **PLACE**

Esta instrução é utilizada para transferir o conteúdo dos dois "bytes" do topo da pilha para a posição de memória apontada pelo conteúdo dos quatro "bytes" imediatamente abaixo destes na pilha.

b) **OBTAIN**

Serve para obter, no topo da pilha, o conteúdo da posição de memória cujo endereço absoluto é dado pelos quatro primeiros "bytes" do topo da pilha.

c) **SENSE**

Compara ("bit" a "bit") o conteúdo dos dois "bytes" do topo da pilha com o conteúdo da posição de memória cujo endereço é dado pelos quatro "bytes" imediatamente abaixo deste, na pilha. Caso algum dos "bits" correspondentes nos dois valores comparados sejam, iguais a um, um valor lógico verdadeiro será colocado no topo da pilha, caso contrário será colocado um valor falso.

d) **ADDR**

Permite obter o endereço absoluto de uma variável.

C A P I T U L O IV

M A Q U I N A B A S I C A

4.1 - INTRUDUÇAO

A adoção da estratégia de interpretação dos programas EDISON, conforme mencionado no capítulo I, permite que seja definida uma máquina básica adequada a execução de programas codificados nesta linguagem.

Este capítulo é a descrição geral da implementação desta máquina básica, desenvolvida na linguagem LPS para o equipamento COBRA-300, que utiliza, como unidade central de processamento o microprocessador 8080 da Intel, e cuja configuração atual da memória é de 48 "kbytes" de RAM, sendo 42 "kbytes" livres para programas de aplicação.

Esta máquina básica está implementada e possui um processador virtual que manipula palavras de 16 "bits" organizadas numa pilha. A figura 4.1 apresenta a organização geral da máquina. Nela podem ser vistos o interpretador, o núcleo, a memória e a unidade de entrada e saída de dados.

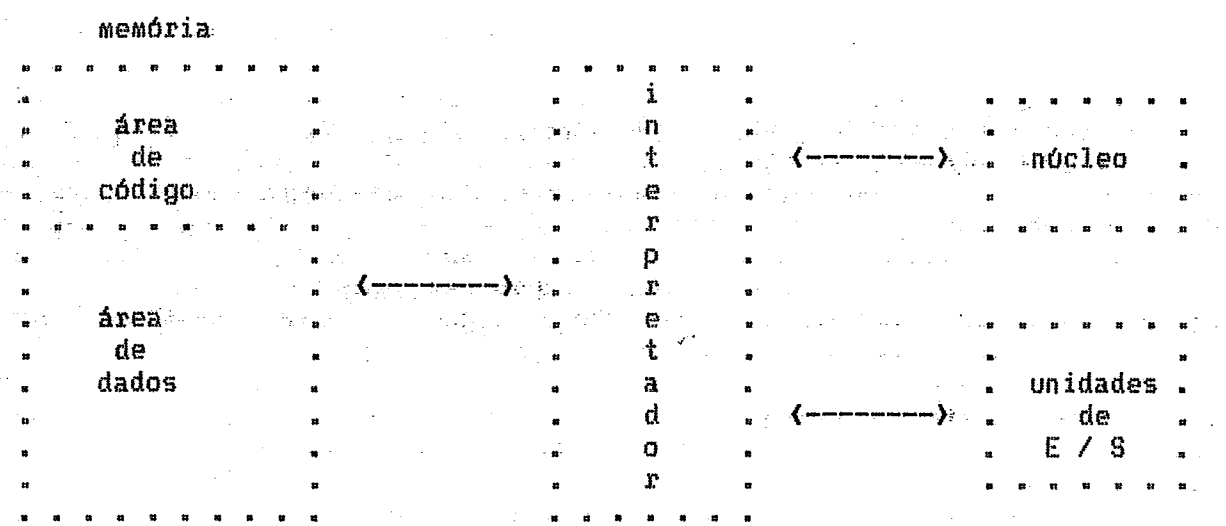


Fig. 4.1 - Organização geral da máquina virtual.

4.2 - O NÚCLEO

Conforme foi mencionado no capítulo I, a introdução do processamento paralelo em sistemas de computação visou essencialmente aumentar sua eficiência e otimizar a utilização de seus recursos.

Muitas têm sido as formas de paralelismo adotadas em sistemas modernos. Essas englobam tipicamente a divisão de tarefas de um processador entre sub-unidades de processamento, a divisão de tarefas do sistema entre processadores e a multiprogramação de um processador.

O núcleo é formado por um conjunto de procedimentos destinados a dar suporte as características de processamento paralelo introduzidas pelas linguagens de programação concorrente. Esses procedimentos desempenham as funções de criação, destruição e escalonamento dos processos concorrentes. Além do escalonamento para execução, o núcleo também escalona os processos para acesso a recursos compartilhados.

A implementação aqui descrita, destina-se a arquitetura monoprocessador, porém a sua utilização em arquiteturas multiprocessadores ficaria garantida com a criação de uma política de escalonamento de processadores para execução dos processos. Deste modo, nesta implementação a tarefa do núcleo consiste em multiprogramar o processador, alocando-o a um processo diferente de tempos em tempos. Essa alocação temporária do processador a um dado processo, é modelada, como se a cada processo fosse dado um processador permanente, virtual, de velocidade inferior a do processador real.

O projeto de sistemas concorrentes envolve dois problemas fundamentais: o da especificação da concorrência entre os processos e o do gerenciamento do acesso a recursos compartilhados. O que está se chamando de especificação da concorrência é algum método que realiza as relações de precedência entre as execuções de determinados processos. Esse problema é tratado no item 4.2.1, onde é apresentada a política de escalonamento de processos implementada. O problema de gerenciamento do acesso a recursos compartilhados está tratado no item 4.2.2, onde é apresentado a ferramenta para delimitar regiões críticas e executá-las com exclusão mútua.

4.2.1 - ESPECIFICAÇÃO DA CONCORRENCIA ENTRE PROCESSOS

As formas de especificar a concorrência entre processos segue uma clasificação geral que as divide no que se pode chamar de formas estáticas e dinâmicas.

Especificar a concorrência entre processos de maneira estática é fazê-lo independentemente de sua execução, por exemplo, através da ativação simultânea de todos eles, fazendo-os existir indefinidamente como entidades ativas; conforme ocorre com os processos em Sistemas Operacionais. As formas dinâmicas de especificação de concorrência são realizadas através de comandos de uma linguagem de programação [Barbosa,5].

Em EDISON, a especificação de concorrência entre processos é feita dinamicamente através de um comando denominado "cobegin". O uso deste comando causa a ativação simultânea de um certo número de processos e o bloqueio do processo que o executou (processo pai) até que todos os processos ativados terminem. A figura 4.2 ilustra a ativação dos processos B, C e D pelo processo A; o grafo da figura mostra o bloqueio de A até o termino de B, C e D.

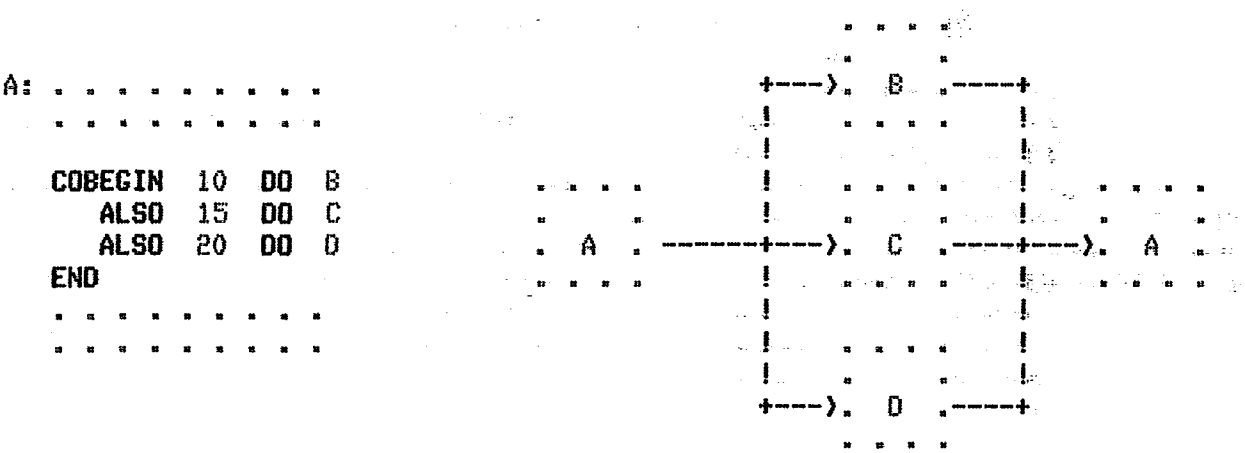


FIG 4.2 Uso do comando "cobegin" para sincronização de processos

Deste modo na implementação aqui descrita, um programa EDISON, inicialmente é executado como um processo único, que ocupa toda a área de dados da memória principal. Quando este processo executar uma instrução do núcleo denominada CREATE, ele é bloqueado pelo núcleo, que inicia a ativação dos processos concorrentes, usualmente denominados de processos filhos.

Durante a ativação dos processos, a área de dados destinada a execução do processo pai, é alocada seqüencialmente aos processos filhos, a medida que os mesmos vão sendo identificados. O tamanho da área de dados destinada a cada um dos processos filhos é determinada pelo usuário através da constante de processo no comando "cobegin". Além disso todos os processos são colocados numa lista circular (anel), através de um apontador para o mesmo na memória.

Um ponteiro para a essa lista, denominado processo atual (PA), é utilizado para indicar qual processo está sendo executado pelo processador, deste modo quando todos os processos tiverem sido criados, este ponteiro, que anteriormente apontava para o processo pai, passa agora a apontar para um dos processo filhos (primeiro processo criado no anel), iniciando-se a execução dos processos concorrentemente.

Para destruir processos é utilizada uma instrução específica, denominda HALT. Ao executar tal procedimento o núcleo verifica, inicialmente se o processo desativado é o processo pai. Caso isto ocorra a execução de todo programa EDISON é cancelada; se o processo desativado é um processo filho, o mesmo é retirado da lista circular, e é ativado o processo seguinte da lista. Se esse processo filho desativado for o único processo da lista o controle do processador é dado ao processo pai, isto é: o ponteiro (PA) volta a apontar para o processo pai.

A figura 4.3 mostra a estrutura de dados usada para representar processos no sistema.

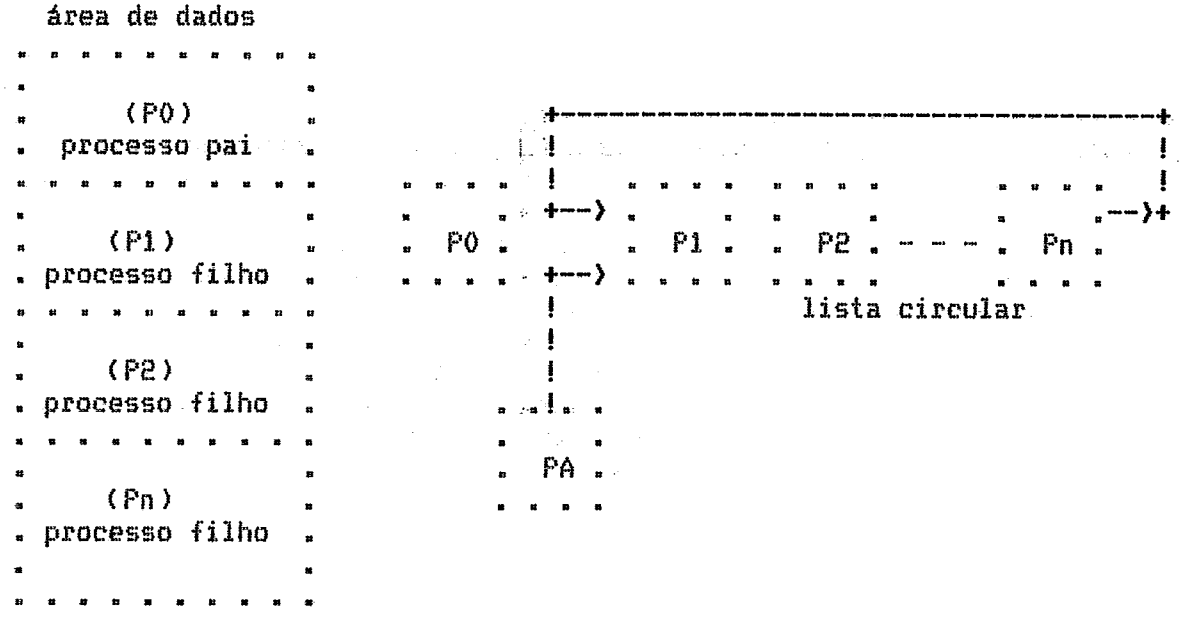


FIG. 4.3 - Representação de processos no sistema

Cada processo é constituido pelo seu descritor e por sua área de dados própria. O descritor de processos contém campos destinados ao armazenamento do contexto do processo. isto é; o conteúdo dos registradores descritos no item 3.3, a tabela de bases ativadas pelo processo e um registrador que indica o tamanho da área de dados do processo.

A área de dados do processo é a região da memória destinada a execução do código virtual, ao armazenamento de variáveis, e ao armazenamento temporário de dados utilizados como operandos na avaliação de expressões. O topo da área de dados é apontada pelo registrador SP conforme mostra a figura 4.4. A utilização de uma área de dados especifica a cada processo, garante que este processo terá acesso exclusivo aos dados declarados locais a ele.

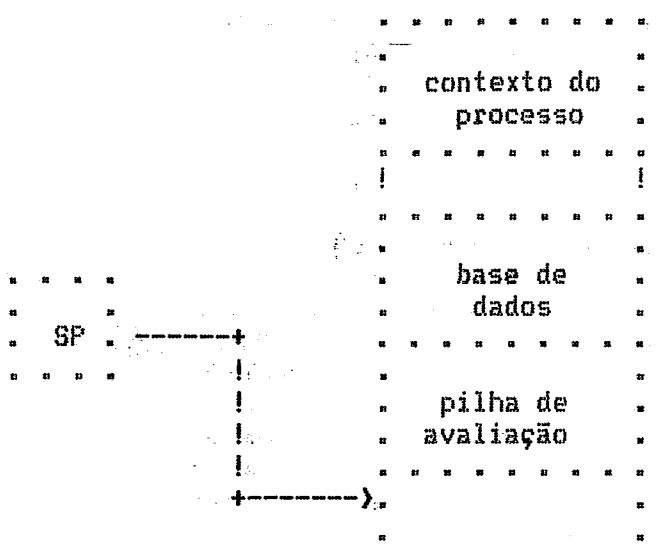


Fig 4.4 - Area de um processo único

4.3.2 - GERENCIAMENTO DE ACESSO A RECURSOS COMPARTILHADOS

A competição entre processos concorrentes (ou não) em geral, envolve acesso a regiões da memória de forma compartilhada, regiões estas denominadas de regiões críticas. O problema de exclusão mútua consiste em garantir que uma região crítica seja executada de maneira exclusiva e seqüencial no tempo por processos concorrentes.

Várias ferramentas já foram sugeridas para a exclusão mútua de processos concorrentes durante a execução de regiões críticas [Andrews, Schneider,3]. De um modo geral, essas propostas buscam criar um conjunto de primitivas que alcancem os requisitos desejáveis para uma solução geral do

problema de exclusão mútua, apresentados por Dijkstra em [Dijkstra,9]:

Em EDISON, toda operação sincronizada sobre estrutura de dados compartilhada, está concentrada em módulos, que possuem entidades exportáveis, acessíveis ao bloco que o envolve. As rotinas exportadas podem ser chamadas por processos e, não se excluem no tempo. A exclusão mútua é assegurada por uma forma simplificada de região crítica condicional (Fig 4.5), que intruduz, não determinismo na ordem em que as chamadas as rotinas do módulo são atendidas.

WHEN Comando Sincronizado **END**

Comando Sincronizado :
Comando Condicional [**ELSE** Comando Condicional] *
Comando Condicional :
Expressão Lógica **DO** Lista de Comandos

Fig 4.5 - Uso do comando "when" como região crítica condicional.

Um processo executa um comando "when" em duas fases:

1. **Fase de sincronização:**

   O processo é bloqueado até que nenhum outro processo esteja executando a fase crítica de um comando "when" qualquer.

2. **Fase crítica:**

   O comando sincronizado é executado da seguinte maneira: se todas as expressões lógicas do comando "when" tiverem valor falso, o processo retorna a fase de sincronização; caso contrário a lista de comandos (região crítica) é executada até o fim.

Para implementar este mecanismo que possibilita aos processos concorrentes acesso exclusivo a recursos compartilhados, foram previstas duas instruções. A primeira, denominada WHEN, é utilizada para implementar a fase de sincronização. Quando um processo executar esta instrução, ele é bloqueado até que nenhum outro processo esteja executando uma instrução WHEN. Se todas as expressões lógicas do comando "when" tiverem um valor falso, então é executada a segunda instrução do núcleo denominada WAIT; caso contrário é executada a região crítica.

A instrução WAIT do núcleo é utilizada para evitar a espera ocupada; isto é, o processo que executou a intrução é bloqueado, e um outro processo da lista circular é selecionado para ser executado.

O processador executa os processos numa ordem cíclica, e cada processo é executado até que uma das instruções HALT ou WAIT sejam encontradas; isto é, os processos não são interrompidos enquanto estiverem em execução. Conseqüentemente a exclusão mútua é automaticamente garantida se tivermos um único processador. Desse modo em nossa implementação a instrução WHEN não precisa garantir a exclusão mútua entre processos. Na eventualidade de ser implementado algum tipo de interrupção (por exemplo, fatia de tempo), a instrução WHEN deverá implementar a fase de sincronização mencionada.

4.3 - O INTERPRETADOR

O interpretador é a Unidade Central de Processamento, sendo encarregada da execução do código virtual. Para tanto, utiliza o conjunto de registradores descritos no item 3.3. Este processador manipula palavras de 16 "bits" organizadas em pilha (área de dados), uma para cada processo. A pilha e os registradores podem ser descritos em EDISON da seguinte maneira:

```
ARRAY  MEMORIA [ ... ] (INT)
VAR    PILHA : MEMORIA;  PC, SP, BT, BASE : INT
```

O interpretador basicamente se compõe de um procedimento inicializador que carrega o programa objeto na memória (área de código), e de um conjunto de procedimentos (um procedimento para cada instrução do código) que realizam as funções necessárias para a execução do código virtual. O interpretador pode ser resumido segundo o algoritmo 4.1.

```
PROC INTERPRETADOR
VAR  INSTRUCAO : INT
BEGIN

      " carrega código objeto na memória "
      PC := endereço inicial de execução
      WHILE   TRUE
              DO   INSTRUCAO := "byte endereçado por PC";
                   PC := PC + 1;
                   " interpreta a instrução "
              END

END " INTERPRETADOR"
```

Alg. 4.1 - Funcionamento do interpretador

O registrador PC aponta para a instrução na área de código que será a próxima a ser interpretada. Ao mesmo tempo em que o "byte" apontado pelo registrador PC é transferido para o registrador de instruções, o valor de PC é incrementado, e passará a apontar para o primeiro operando da instrução.

O mecanismo responsável pela avaliação do endereço efetivo dos operandos está incluído na interpretação de cada instrução virtual, cabendo a esse mecanismo portanto a busca (ou modificação) dos operandos referenciados pelo código. A seleção do procedimento adequado a interpretação de cada instrução é feita através de um comando "case" da linguagem LPS.

Durante a chamada de um procedimento o interpretador usa um bloco denominado base de dados (fig 4.4) que contém as seguintes informações:

1. Se o procedimento é do tipo função então há um campo destinado para guardar o resultado da função.

2. Espaço destinado aos parâmetros do procedimento (se houver algum);

3. O conjunto de valores relativos ao meio ambiente que chamou o procedimento, Com esses valores será possível restaurar o contexto da máquina quando do retorno.

4. As variáveis do procedimento e dos módulos declarados locais a ele.

O endereço da base de dados é conhecido como endereço da base de um procedimento ativado. Este endereço está residente na tabela de bases implementada pelo registrador BT (fig 4.6). Assim quando um processo chamar um procedimento, a base de dados é estabelecida na pilha, e o endereço de sua base de dados é incluido no topo da tabela de bases, sendo removido quando o procedimento terminar de ser executado. Considerando, por exemplo, um processo descrito pelo algoritmo 4.2.

```
PROC  P1
   PROC  P2
   VAR X2
      PROC  P3
      VAR X3
      BEGIN . . . . END

      PROC  P4
      BEGIN . . . P3 . . . END
   BEGIN . . . P4 . . . END
BEGIN . . . P2 . . . END
```

Alg. 4.2 - Exemplo de processo ativo

A figura 4.6 mostra a configuração da pilha e dos registradores (contexto do processo), quando o processo já chamou os procedimentos na ordem P1, P2, P4 e P3; e está executando P3.

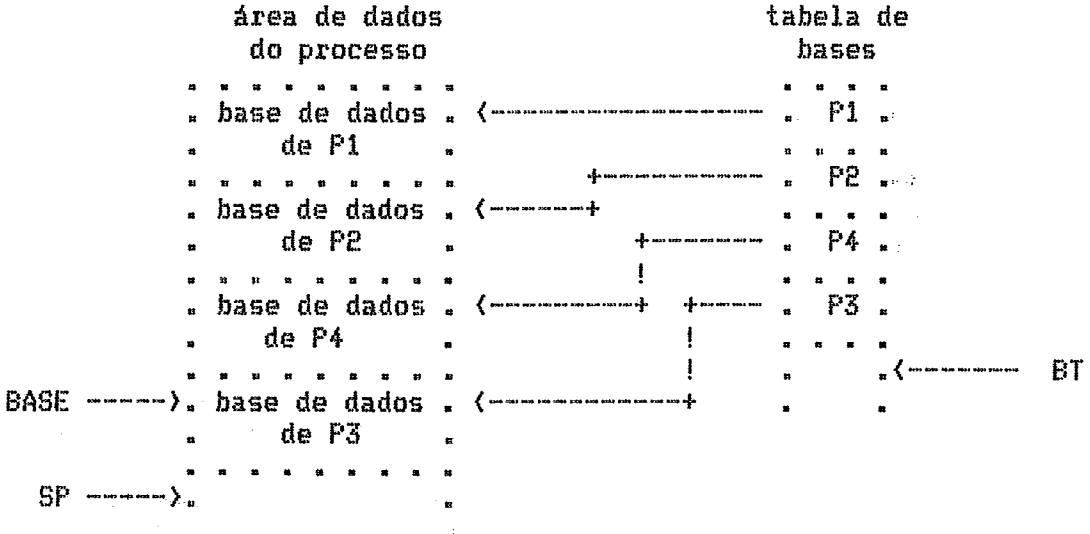


Fig. 4.6 - Contexto do processo descrito no Alg 4.2

O registrador BASE contém o endereço da base de dados do procedimento correntemente ativo (no caso P3); enquanto o topo da pilha é apontado pelo registrador SP, e o topo da tabela de bases pelo registrador BT.

A tabela de bases é utilizada particularmente para obtenção do endereço de variáveis globais ao procedimento ativo. Conforme mencionamos no item 2.4.1.1, os endereços das variáveis são constituidos de uma base e um deslocamento. A base de uma variável local ao procedimento ativo (P3 referencia X3), é obtida diretamente do registrador BASE; enquanto a base de variáveis globais (P3 referencia X2) é obtida através da tabela de bases.

A pilha de avaliação representada na figura 4.4 é utilizada para o armazenamento temporário de dados utilizados como operandos na avaliação de expressões. Esta pilha se desenvolve dinamicamente durante a execução de um procedimento e é manipulada pelo registrador SP.

4.4 - UNIDADES DE ENTRADA E SAIDA

Nesta implementação a máquina básica possue unidades virtuais para entrada e saida de dados, mapeadas em dispositivos periféricos reais pelo interpretador.

Estas unidades são vistas simplesmente como fonte e destino de uma posição de memória endereçada de forma absoluta, e são manipuladas por procedimentos pré definidos da seguinte forma:

```
PROC  PLACE  ( X, Y, Z : INT )
PROC  SENSE  ( X, Y, Z : INT ) : BOOL
PROC  OBTAIN ( X, Y : INT ) : INT
PROC  ADDR   ( VAR Z : QUALQUER; VAR X, Y : INT )
```

A operação PLACE atribui o valor inteiro de Z, a unidade mapeada pelo endereço (X,Y) onde X representa os endereços até 64k, e Y endereços acima de 64k. A operação OBTAIN atribui ao topo da pilha o valor da unidade mapeada pelo endereço (X,Y). Enquanto a operação SENSE compara o valor inteiro de Z com os "bytes" endereçados por (X,Y). A operação ADDR coloca o endereço de Z em (X,Y).

Além disso a operação PLACE causa a impressão do valor Z em dispositivo periférico real do equipamento, enquanto a operação OBTAIN lê um valor inteiro de um dispositivo periférico real, colocando-o no topo da pilha.

4.5 - ERROS DE EXECUÇAO

Durante a execução de um programa EDISON, poderão ocorrer situações em que a execução do programa poderá ser interrompida; nestes casos a máquina básica emitirá uma mensagem de advertência ao usuário e interromperá a execução do programa. Estas mensagens poderão ser uma das seguintes:

Final de execução:

O procedimento HALT foi executado.

Programa muito grande, execução cancelada:

O tamanho do programa excede a capacidade da memória, ou a área de dados é insuficiente para a criação dos processos.

Indice inválido, execução cancelada:

Os limites de um arranjo foram ultrapassados.

Proteção de memória, execução cancelada:

Houve uma tentativa de ser acessada uma posição de memória fora dos limites atribuidos ao processo.

C A P I T U L O IV

C O M E N T A R I O S F I N A I S

A meta inicial de contrução de um meio ambiente que suportase a linguagem concorrente EDISON foi atingida com a implementação e validação do compilador e da máquina básica.

A escolha de uma estrutura multipasso no projeto do compilador EDISON, foi adotada com a finalidade de que o sistema pudesse ser implementado em minicomputadores. Graças a essa estrutura, durante a fase de testes, o compilador lista o texto fonte e as formas intermediárias geradas por cada passo. Dessa forma a comparação entre a entrada e a saída de cada passo permitiu que eles fossem testados independentemente.

Os resultados produzidos pelo compilador são bastante satisfatórios tanto na qualidade dos erros corrigidos, como no desempenho apresentado. A incorporação nas mensagens de erros da ação tomada pelo compilador na corrreção dos erros sintáticos trouxe legibilidade aos resultados, facilitando portanto o trabalho dos usuarios do compilador EDISON.

Após um tempo inicial de 8.0 segundos o compilador EDISON analisa cerca de 10 linhas por segundo de texto fonte.

O tamanho do sistema é o seguinte:

Programa de controle	300	linhas
Análise léxica (passo 1)	400	"
" sintática (" 2)	400	"
" semântica (" 3)	1500	"
Gerador de código (" 4)	800	"
Compilador	3400	linhas
Máquina Básica	700	linhas
Processador EDISON	4100	linhas

A construção de um compilador EDISON, escrito em EDISON já factível agora, permitiria que todo o sistema ficasse concentrado em uma única máquina. Este trabalho será desenvolvido numa etapa posterior como prosseguimento deste projeto.

Atualmente encontra-se em desenvolvimento na COPPE/UFRJ projetos que visam aperfeiçoar o processo de interpretação do código virtual.

REFERENCIAS BIBLIOGRAFICAS

01- AHO, A.V., ULLMAN,J.D., "Principles of compiler design", Addison-Wesley, April 1979.

02- ALGOL B 7000 / B 8000, "Reference manual", Burroughs Corporation.

03- ANDREWS, R.G., SCHNEIDER B.F., "Concepts and notation for concurrent programming", Computing Surveys, Vol 15, 1, march 1983.

04- ARGOLLO JR. M., ZANCANELLA, L.C., "Uma experiência em correção de erros sintáticos", Anais do X Semish, Campinas, SP, 1983.

05- BARBOSA, V.C., "Uma proposta para estender o sitema Pascal USCD a processamento concorrente", Tese M.Sc. COPPE/UFRJ, 1982.

06- BARRON, D.W., "Pascal- The language and its implementation", John Wiley, Chichester, 1981.

07- BARROS, L., SCHNEIDER, S.M., "Recuperação de Erros", Rel. Tec., COPPE/UFRJ (a ser publicado).

08- CIESINGER, J., "A bibliography of Erros Handling", Sigplan Notices, June 1974.

09- DIJKSTRA, E.W., " Hierarchical ordering of sequencial process", Acta Informatica, 1,2, pp. 115-138, 1971.

10- FERNANDES,E.S.T., FRANÇA, F.M.G., OBRASCA, K., SANTOS, M.S., "Em direção a uma máquina EDISON LSI", Anais do I SBCCI, Porto Alegre, 1983.

11- HANSEN, P.B., "Programming a personal computer", Prentice-Hall, Englewood, NJ, 1982.

12- HANSEN, P.B., "The Architecture of Concurrent Programs", Prentice-Hall, Englewood Cliffs, NJ, July 1977.

13- HANSEN, P.B., "Edison: a multiprocessor language", Software: practice and Experience, 11, pp. 325-361, August 1981.

14- HANSEN, P.B., "Edison Programs", Software: Practice and Experience, 11, pp. 397-414, August 1981.

15- HANSEN, P.B., "Operating System Principles", Prentice-Hall, Englewood Cliffs, NJ, July 1973.

16- HARTMANN, A.C., "A concurrente Pascal compiler for minicomputers", Lecture Notes in Computer Science, 50, Springer-Verlag, Berlin, 1977.

17- HOARE, C.A.R., "Monitors: an operating system structuring concept", Comm. ACM 14, 10, pp. 549-570, October 1974.

18- LPS, "Manual de programação LPS", Cobra Computadores e Sistemas Brasileiros S/A.

19- ROHRICH, J., "Automatic Construction of Error Correcting Parsers", Bericht 8, Facultat fur Informatic, Universitat Karlsruhe,1978.

20- ROHRICH, J., "Methods for the Automatic Construction of Error Correcting Parsers", Acta Informatica, vol. 13, 1980.

21- SOM, "Manual de referência do sistema operacional", Cobra computadores e sistemas Brasileiros S/A.

22- TELES, A.A.S., SIMONE, E.G., "Gerador de Analisadores Sintáticos RRP LL(1)", Anais do VIII Semish, Florianopolis, SC, 1981.

23- VASCONCELOS, N.Q., "Uma máquina Edison", Anais do X Semish, Campinas, SP, 1983.

24- WIRTH, N., "Modula: a language for modular multiprogramming", Software: Practice and Experience 7, 2, pp. 3-35, March-April 1977.

25- WIRTH, N., "Algorithms + Data Structures = Programs", Prentice-Hall, Englewood Cliffs, NJ, 1976a.

26- WIRTH, N., "The programming language Pascal", Acta Informatica 1, pp. 35-63, 1971.