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.

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.

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.

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.

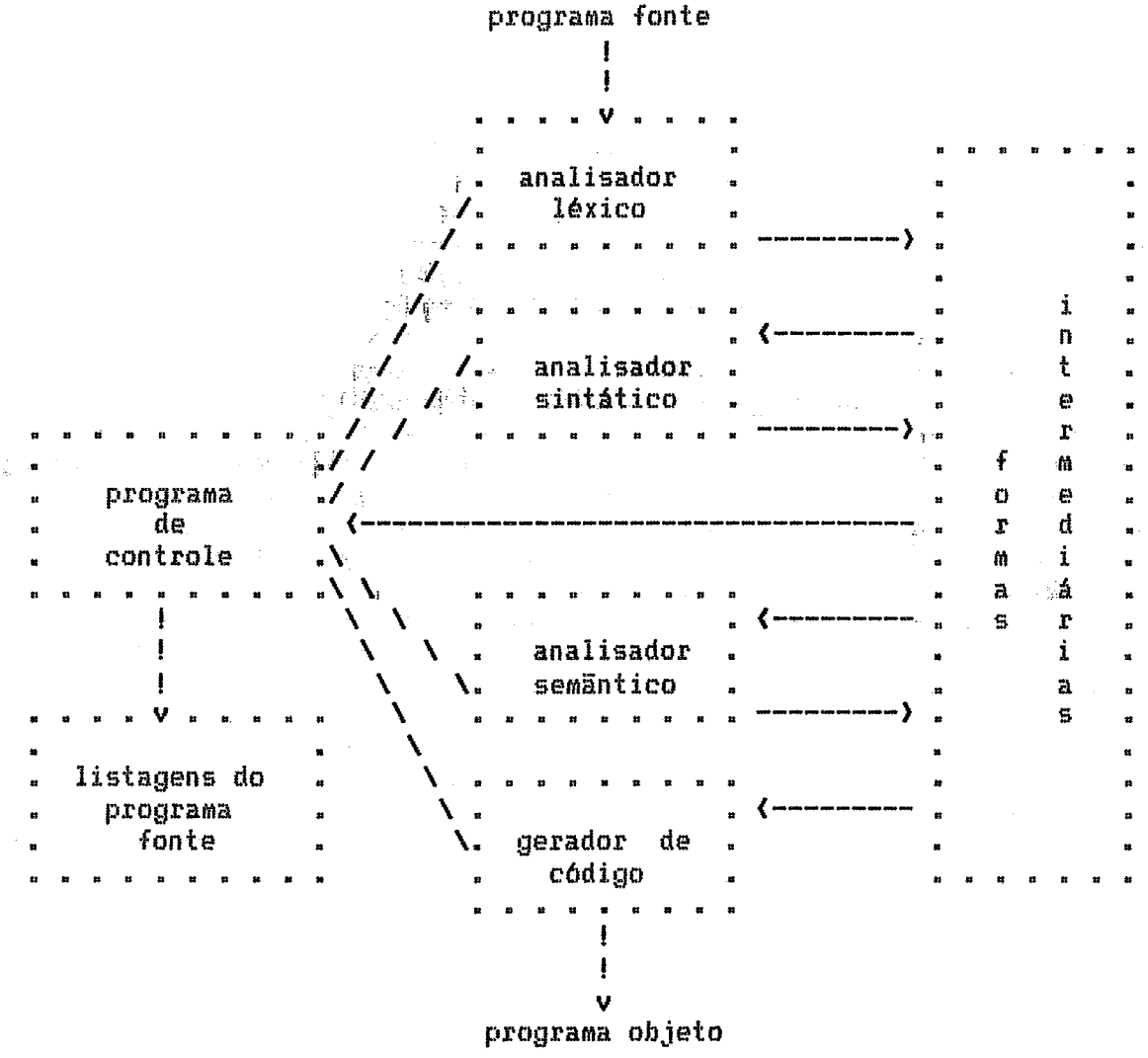


Fig. 2.1 - Organização do compilador

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

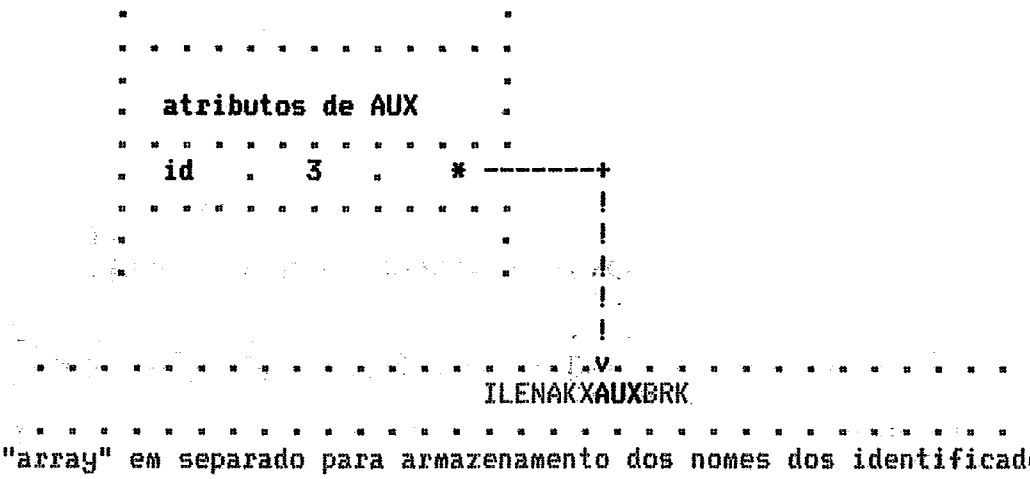
Esse procedimento é responsável pelo tratamento dos identificadores do texto fonte (isto é, palavras reservadas e nomes criados pelo programador). Inicialmente ele extrai a cadeia de caracteres que forma o identificador e avalia - através da função "hash",- o índice da tabela de símbolos correspondente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

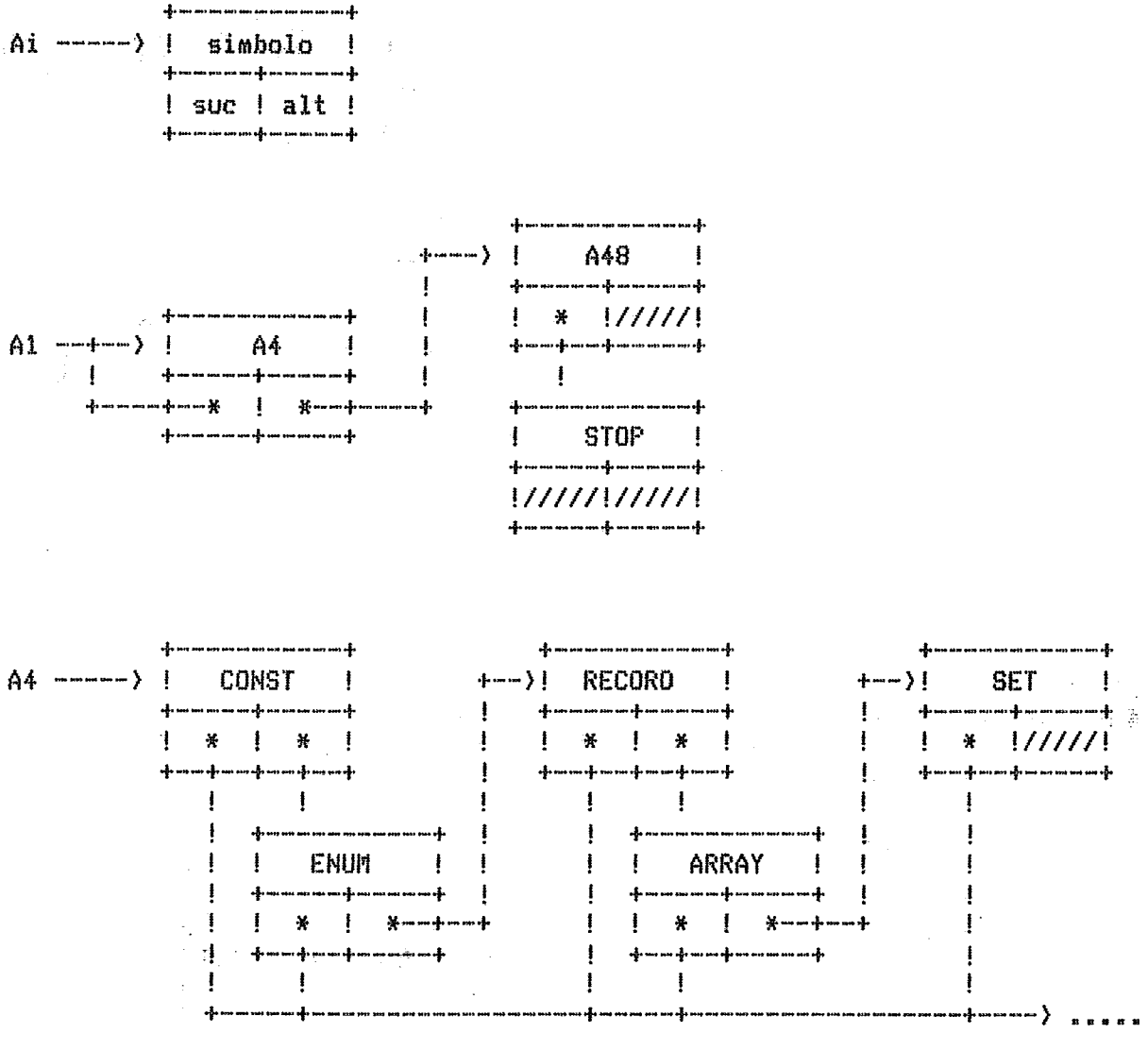
A4 - Declaração de tipo =
(**CONST** # **ENUM** # **RECORD** # **ARRAY** # **SET**) ...

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).



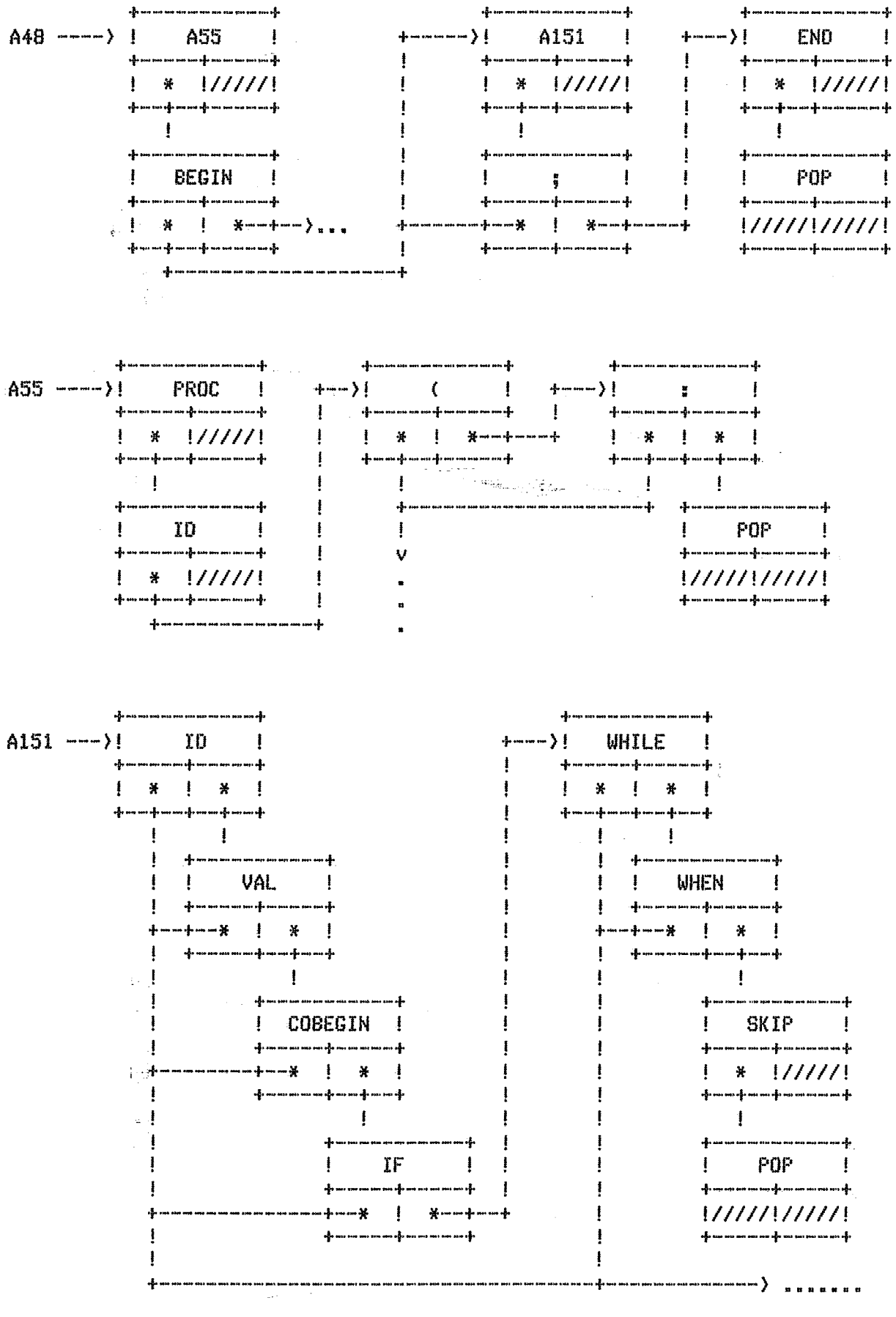


Fig 2.4 - Representação da figura 2.3 na forma de lista.

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.

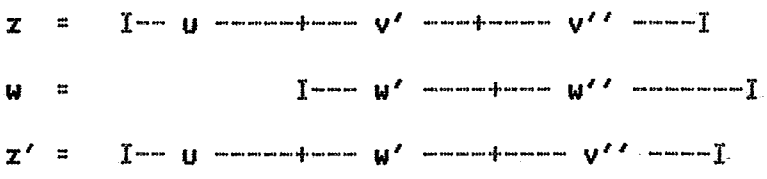


Fig 2.5 Correção do ponto de erro (u,v) da palavra z. Apartir da cadeia de continuação w.

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.

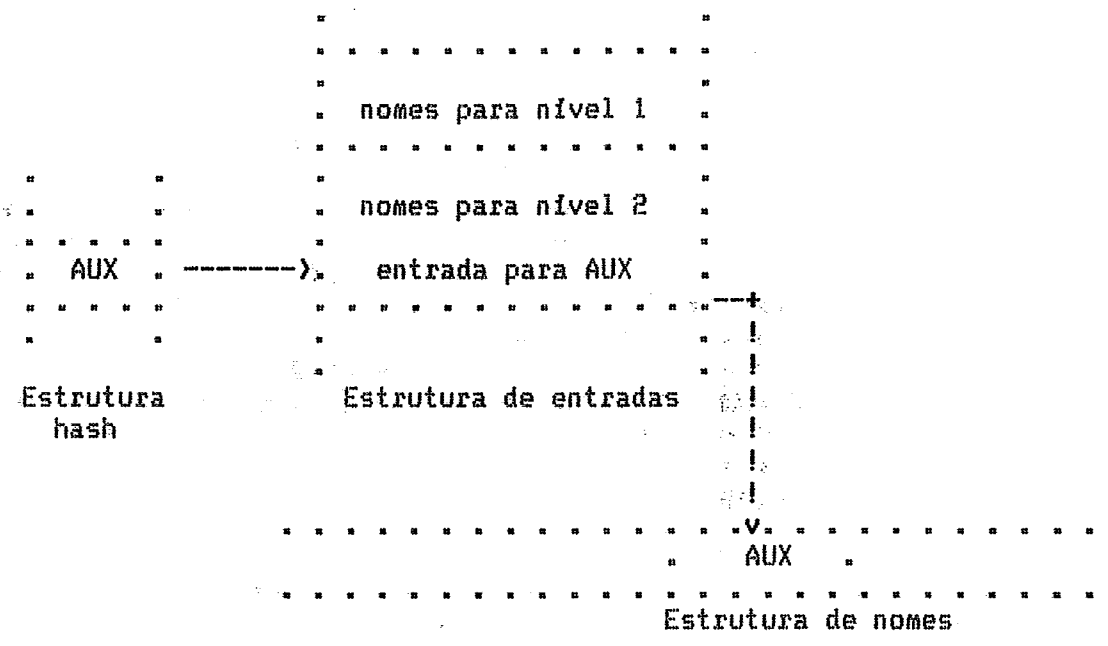


Fig. 2.6 Organização da tabela de símbolos para a análise semântica.

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 capitulo 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.

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.

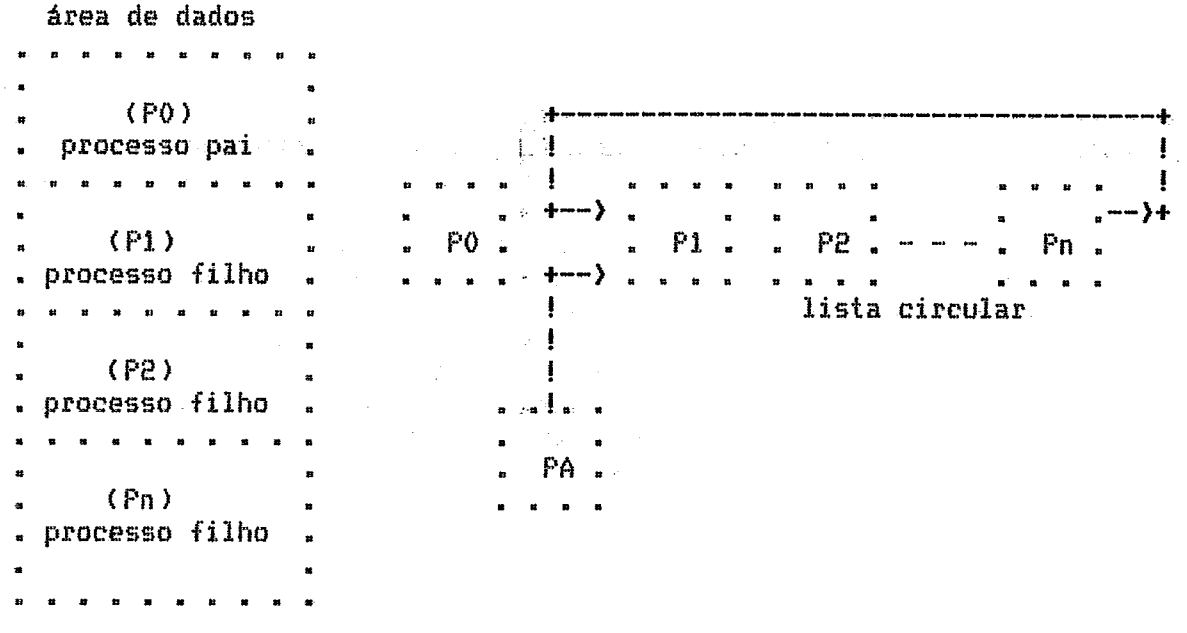


FIG. 4.3 - Representação de processos no sistema