UMA IMPLEMENTAÇÃO DE MODULA-2:

ANÁLISE E REPRESENTAÇÃO INTERMEDIÁRIA

AUTOR
José Pedro Junior

ORIENTADOR
Prof. Dr. Tomasz Kowaltowski

Dissertação apresentada no Instituto de Matemática, Estatística e Ciência da Computação, como requisito parcial para obtenção do título de Mestre em Ciência da Computação

NOVEMBRO - 1983

RESUMO

Implementou-se um compilador para a linguagem de programação MODULA-2. Esta redação descreve as análises léxica, sintática e de contexto, bem como a tradução do programa fonte para uma "representação intermediária". Tal representação é o ponto de partida para a geração de código (não descrita na redação).

A forma da representação intermediária é um grafo direcionado, cujos nós descrevem objetos da tabela de símbolos, comandos e expressões, sendo que o conteúdo da descrição é essencialmente independente da escolha da máquina alvo.

ABSTRACT

A compiler for the programming language MODULA-2 was implemented. This work describes the lexic, syntatic and context analysis, and also the source program translation to an "intermediate representation". This representation is the basis for code generation (not described here).

The intermediate representation is a direct graph, whose nodes describe symbol table objects, comands and expressions, in such a way that the description is virtually object-machine independent.

UMA IMPLEMENTAÇÃO DE MODULA-2:

ANÁLISE E REPRESENTAÇÃO INTERMEDIÁRIA

AGRADECIMENTO

A orientação do professor Tomasz Kowaltowski foi excelente. A ela se devem, principalmente, os méritos do trabalho.

Fernando Antônio Vanini e Maurício Breternitz Jr. foram os criadores da idéia de se desenvolver o compilador. Foram os principais impulsionadores do projeto e me auxiliaram em muitos aspectos no decorrer do trabalho.

Eduardo César Grizendi e Maria Cecília Rupp Blasi Mandel fizeram os testes iniciais.

A todos, meus sinceros agradecimentos.

Agradeço ainda ao CPqD - TELEBRÁS, pelos recursos computacionais com os quais foi realizado o trabalho.

à minha esposa
e aos meus filhos

Í N D I C E

1. INTRODUÇÃO

A linguagem MODULA-2 [1], sucessora de MODULA [2], foi definida por Niklaus Wirth.

O advento de MODULA-2 foi provocado pela necessidade de uma linguagem para programação de sistemas para minicomputadores, que fosse geral e eficientemente implementável.

Uma liguagem de programação de sistemas moderna deveria facilitar, em particular, a construção de programas grandes, projetados possivelmente por várias pessoas. Seria conveniente que as partes programadas pelas diversas pessoas tivessem interfaces bem especificadas, que pudessem ser declaradas indepententemente da sua implementação. MODULA-2 fornece facilidades para isso, através de módulos de definição e módulos de implementação.

O uso de módulos permite também que se isole efetivamente a programação dependente de máquina. É ainda conveniente para se especificar isoladamente tipos de dados e operações sobre eles.

MODULA-2 é assim, um complemento às linguagens de alto nível, no sentido de prover igualmente programação de "baixo nível" de modo simples, seguro e "limpo".

O presente trabalho é basicamente a descrição do primeiro passo ("front-end") de um compilador para a linguagem MODULA-2. O computador objeto é o processador INTEL-8088. A implementação tratada foi desenvolvida por ocasião de um convênio entre o Departamento de Ciência da Computação do IMECC - UNICAMP e a TELEBRÁS.

Tendo o compilador dois passos, foi o primeiro desenvolvido pelo autor desta redação, sob direção de seu orientador, o qual implementou o segundo passo.

2. A LINGUAGEM MODULA-2

Descendente do PASCAL, MODULA-2 herda-lhe a estrutura de dados e a de comandos, com pequenas alterações. A principal novidade em relação a PASCAL é o conceito de módulo. Já os havia, aliás em MODULA.

Apesar de ser linguagem voltada para programação de sistemas, MODULA-2 é essencialmente independente de máquina, sendo que as dependências podem ser isoladas pelo programador em módulos de "baixo nível". Em tais módulos, a possibilidade de se efetuar "programação insegura" é oferecida pela linguagem, sendo então suspensas as regras de compatibilidade de dados. Construções sintáticas especiais fazem com que tais ações sejam explicitamente anunciadas.

Particularmente importante é o módulo SYSTEM, obrigatório em toda implementação da linguagem e que encerra tipos como ADDRESS, WORD, e outros. Tipicamente, numa implementação de MODULA-2, o compilador trata de modo especial os objetos de tal módulo.

O conceito de processo e sua sincronização através de sinais como apareceu em MODULA, foi substituído em MODULA-2 por um conceito de nível mais baixo: corrotina. É possível, entretando, formular um módulo que implemente tais processos e sinais. A vantagem de não incluí-los na linguagem é que o programador pode escolher um algoritmo de escalonamento de processos adaptado às suas necessidades, programando um módulo apropriado. Tal escalonador pode ser até omitido nos casos simples (mas freqüentes), por exemplo, quando processos concorrentes ocorrem apenas como controladores de dispositivos.

Seria útil ao leitor, na seqüência, um prévio conhecimento mais detalhado da linguagem, que poderá ser haurido em [1], [3].

3. CONSIDERAÇÕES GERAIS SOBRE A IMPLEMENTAÇÃO

3.1. CARACTERÍSTICAS GERAIS DO COMPILADOR

A compilação é "cruzada": o compilador é executado no sistema DEC-10 e gera código em linguagem de montagem do processador INTEL-8086/8088. A linguagem de desenvolvimento foi PASCAL.

A divisão em dois passos, já citada, permitiu separar-se as funções de tradutor ("front-end") das de gerador de código ("back-end").

O tradutor (primeiro passo) é essencialmente independente de máquina (i.e. do computador objeto). Produz uma representação intermediária a partir do programa fonte, efetuando as verificações sintáticas e de contexto. A representação intermediária é uma estrutura ligada correspondente a uma árvore de derivação, mais informações da chamada tabela de símbolos. Usaremos, nesta redação, o nome impróprio "árvore do programa" para aquela estrutura.

Ainda no primeiro passo, é efetuado o tratamento das diretivas ao compilador.

O gerador de código (segundo passo) depende fortemente do computador objeto. É constituído basicamente, por um conjunto de rotinas que, percorrendo a árvore do programa, geram o código objeto (em linguagem de montagem, como vimos).

3.2. MODIFICAÇÕES E ADAPTAÇÕES DA LINGUAGEM

A fim de serem convenientemente aproveitadas as características do computador objeto, foram feitos alguns acréscimos à linguagem.

Criou-se, por exemplo o tipo BYTE, além do tipo WORD, previsto, pois o computador objeto provê mecanismos de manipulação com palavras (16 bits) e "bytes" (8 bits).

Analogamente, outros tipos foram criados em decorrência: SHORT_INTEGER e SHORT_CARDINAL. As "funções de transferência de tipos" foram convenientemente estendidas para aceitar objetos de tais tipos, e foram ainda criadas funções de conversão, para transformação de valores INTEGER para SHORT_INTEGER (e vice-versa) e CARDINAL para SHORT_CARDINAL (e vice-versa).

Outro fator de extensão da linguagem, veio das características de endereçamento no computador objeto. Assim, a noção de ADDRESS do MODULA-2 foi mantida em operações realizadas dentro de um mesmo segmento de memória. O nome LONG_ADDRESS ficou reservado para endereçamento mais genérico, em que se especifica a base de um segmento e o deslocamento dentro do mesmo.

Analogamente estendemos o tipo apontador para duas modalidades: POINTER e LONG_POINTER.

Foram também feitas restrições de implementação. Uma delas consistiu em omitir-se o tipo "REAL", já que o computador objeto não processa ponto flutuante.

Não foi também implementada a compilação separada, em sentido estrito. O efeito lógico da compilação separada é conseguido através da inclusão de todos os módulos de definição e de implementação necessários em toda compilação de um módulo de programa. Os módulos a serem incluídos na compilação são tratados pela diretiva USE (a ser ainda vista). (No capítulo 5 mencionamos trabalhos desenvolvidos após o escopo inicial de trabalho descrito nesta redação).

Entre modificações menores contam-se pequenas alterações da sintaxe. Por exemplo, a especificação de endereço pré-definido para variáveis passou de um para um ou dois parâmetros, a fim de se permitir o uso de endereço generalizado visto (i.e. base e deslocamento).

4. O PRIMEIRO PASSO ("FRONT-END")

4.1. ASPECTOS LÉXICOS

A análise léxica corresponde ao nível mais baixo de estrutura do programa fonte.

São funções do analisador léxico:

- ler o programa fonte
- escrever listagem do programa
- tratar comentários
- tratar diretivas ao compilador
- passar ao analisador sintático os itens léxicos encontrados

4.1.1. LEITURA DO PROGRAMA FONTE

O programa fonte é um texto lido seqüencialmente. O analisador léxico está implementado como uma rotina, chamada "scan" que a cada chamada devolve (através de variáveis globais) o próximo símbolo léxico do programa fonte. Na verdade, quando "scan" é chamada, o próximo caráter já foi lido e, complementarmente, quando do retorno, "scan" já se encarregou de ler o próximo caráter.

O texto do programa fonte pode estar fisicamente distribuído em vários arquivos. "scan" passa de um arquivo a outro, mediante as diretivas USE e INCLUDE, conforme veremos.

4.1.2. LISTAGEM DO PROGRAMA

À medida que o programa fonte é lido, uma listagem de saída é criada, na forma de um (único) arquivo. Em tal arquivo, o programa fonte é transcrito, sendo-lhe adicionado:

- numeração de linhas e páginas
- nível de encaixamento léxico

- mensagens de erro
- identificação de trechos ignorados pelo analisador sintático
- data e hora da compilação
- tempo de processamento empregado na compilação.

4.1.3. TRATAMENTO DE COMENTÁRIOS

Comentários são trechos do programa fonte ignorados do ponto de vista sintático. Em nossa implementação foram incluídos dois tipos de comentários:

- comentário de texto e
- comentário de linha.

O primeiro dos dois é previsto na definição da linguagem. Refere-se a trechos do programa situados entre "(*" e "*)". Tais delimitadores devem aparecer "balanceados", exatamente como os parênteses numa expressão aritmética.

O segundo tipo de comentário tem como início "--" e se estende até o fim da linha corrente.

O finalizador de um tipo de comentário não serve a outro tipo.

4.1.4. TRATAMENTO DE DIRETIVAS

Foi incluído no compilador, um certo número de diretivas. Todas elas começam com "$", seguido de um identificador.

De um modo geral, o uso de diretivas visa:

- ações do analisador léxico. Ex: especificação de arquivos onde o programa fonte está distribuído (USE, INCLUDE).

- ações do gerador de código. Ex: determinar se o código gerado incluirá verificação de limites permitidos para valores gerados durante a execução (CHECK).

Por motivos de enfoque, detalharemos apenas as diretivas com efeitos no primeiro passo. O leitor interessado poderá consultar [3] a respeito das outras diretivas.

4.1.4.1. DIRETIVA "UPPER-CASE"

Forma: "$UPPER_CASE = b" onde "b" é TRUE ou FALSE.

Significado: Permite ao usuário especificar se deseja ou não diferenciar letras maiúsculas e minúsculas em identificadores. Só pode ocorrer uma vez, no início do programa. Em caso de não especificação, o valor adotado é TRUE, significando não diferenciação.

4.1.4.2. DIRETIVA "LIST"

Forma: "$LIST = b" onde "b" é TRUE ou FALSE.

Significado: Ativa ou inibe a impressão do programa fonte, a partir do ponto em que ocorre. No início de cada unidade de compilação, a impressão é previamente ativada. A diretiva pode ocorrer em qualquer lugar do programa fonte.

4.1.4.3. DIRETIVA "INCLUDE"

Forma: "$INCLUDE [proj, progr] arq1, ... , arqn" onde os arqi's são nomes de arquivos e "[proj, progr]" indica área em disco no DEC-10.

Significado: Permite especificar arquivos a serem textualmente incluídos na compilação. A "área" é parâmetro opcional (se omitida, supõe-se a área que está sendo usada). As

inserções dão-se respeitando a ordem de ocorrência dos nomes de arquivos. Outros arquivos, porém, poderão ainda estar entremeados, já que um arquivo incluído pode também incluir. A diretiva pode ocorrer em qualquer ponto de programa fonte.

4.1.4.4. DIRETIVA "USE"

Forma: "$USE [proj, progr] ident1, ... identn".

Significado: Permite especificar módulos externos (definição e implementação) incluídos. Para cada identificador "identi", o compilador inclui os arquivos "identi.def" (contendo um módulo de definição) e "identi.imp" (contendo o módulo de implementação correspondente, se existir). Cada módulo é incluído uma única vez, mesmo que o arquivo correspondente apareça em mais de uma diretiva USE. Analogamente ao caso de INCLUDE, módulos incluídos por USE, podem também incluir. Assim, num dado momento, vários arquivos podem estar abertos para leitura. Restrições de implementação fazem com que fixemos em 10 o número máximo de tais arquivos simultaneamente abertos.

4.1.5. ERROS LÉXICOS

São de pequena variedade, geralmente causados por erros de digitação ou edição.

Exemplos:

- leitura de constante numérica muito grande em magnitude
- fim de programa fonte encontrado inesperadamente
- diretiva desconhecida ou incorreta
- leitura de caráter estranho.

4.2. ASPECTOS SINTÁTICOS

Segundo moldes já usados na descrição do ALGOL-60 [4], a definição da linguagem MODULA-2 foi feita [1] mediante duas

caracterizações:

- uma de enfoque sintático, dada por uma descrição tipo BNF [5], i.e., usando-se uma gramática livre de contexto [6].

- e uma caracterização semântica e de contexto, complementar.

Linguagens de programação em que a especificação de objetos acompanha seu uso são sensíveis ao contexto. A escolha de uma correspondente gramática sensível ao contexto, porém, dificilmente levaria a um analisador sintático eficiente. Esse é o motivo pelo qual desde ALGOL-60 usa-se, em linguagens de estrutura semelhante, uma definição híbrida como visto acima. Em tais casos, situações de confronto entre definição e uso de objetos (caso típico de análise de contexto) são resolvidos via "tabela de símbolos".

Optou-se por análise descendente por motivos de clareza e flexibilidade. A eliminação da necessidade de retrocesso deveu-se ao fato de a linguagem ser "quase" LL(1). (As poucas exceções são contornáveis via tabela de símbolos).

4.2.1. O ANALISADOR SINTÁTICO

Cada rotina sintática corresponde tipicamente a um símbolo não terminal da linguagem, sendo os símbolos terminais devolvidos pelo analisador léxico.

As rotinas sintáticas são mutuamente recursivas, sendo a estrutura de controle que as relaciona ditada pelas produções da gramática.

A dependência sintaxe-contexto é tal que se torna natural imbricar as análises sintáticas e de contexto. Assim, uma dada rotina "sintática" armazena e consulta informações de contexto. Para facilidade de exposição, porém, estaremos

descrevendo a análise sintática independentemente da de contexto (a ser tratada em item próprio).

O fato de o analisador ser descendente implica, a cada ponto do programa, saber-se a priori qual estrutura sintática está sendo tratada. Situações de decisão são resolvidas conhecendo-se previamente o próximo símbolo léxico (e talvez com uma consulta à tabela de símbolos). Assim, é conveniente que à entrada de um rotina sintática, "scan" já tenha sido chamada (e, correspondentemente, à saída, "scan" seja chamada uma vez a mais).

4.2.2. RECUPERAÇÃO DE ERROS SINTÁTICOS

O esquema de recuperação de erros sintáticos adotado foi basicamente o de Wirth [7]. Introduzimos porém modificações, visando melhores resultados dentro do espectro previsível de utilização do compilador. Usamos como hipótese de trabalho o público a que o compilador será especialmente destinado. Como MODULA-2 visa programação de sistemas, não suporemos seu uso por parte de principiantes em programação. Ao contrário, imaginamos um programador típico como:

- Sendo experiente e, portanto, cometendo relativamente poucos erros sintáticos.

- Possivelmente acostumado a programar em PASCAL [8]. Isso, não significando que será penalizado o usuário não familiar com aquela linguagem.

4.2.2.1. O ALGORITMO DE WIRTH

Quando um erro sintático é detetado, uma mensagem apropriada deve ser emitida mas a análise sintática deve prosseguir para inclusive outros erros serem detetados.

No algoritmo de Wirth, quando um erro sintático é detetado, um trecho do programa fonte é ignorado pelo analisador

sintático até que "alguma análise plausível possa ser retomada" [7]. Para tal fim, Wirth recomenda a rotina "test" dada abaixo:

```
procedure test (s1, s2: symb_set;  n: integer);
begin
   if not (symb in s1)
   then begin
        error (n);  s1 := s1 + s2;
        while not (symb in s1) do scan
        end
end.
```

Nesta rotina:

- s1 é um conjunto de símbolos imediatamente admissíveis. Se o símbolo corrente não está entre eles, tem-se situação de erro.

- s2 é um conjunto de símbolos futuramente admissíveis; sua função é fazer o analisador sintático retomar atenção ao texto (que estava, em caso de erro, sendo ignorado).

- n é um número de identificação da mensagem de erro.

Assim, quando fosse ativada uma rotina sintática correspondente a um símbolo não terminal A, teríamos tipicamente:

- s1 com os símbolos que podem iniciar uma frase derivável de A (i.e. "first(A)" [7]).

- s2 com os símbolos terminais que podem seguir imediatamente A, na frase sentencial à qual A pertence no caso presente (i.e. "follow(A)").

4.2.2.2. MODIFICAÇÕES DO ALGORITMO DE WIRTH

A fim de diminuir o tamanho dos trechos do programa-fonte ignorados pelo analisador sintático, incluímos mais símbolos em s2; por exemplo, símbolos terminais pertencentes a frases sentenciais deriváveis do símbolo não terminal sendo tratado.

Exemplificando, suponhamos a análise de um trecho de programa correspondente a um não terminal A, que tenha a produção "A ::= BC". Incluímos então, via de regra, em s2, os símbolos pertencentes a "first(C)".

Como vimos, o aumento de s2 visava diminuir perdas de trechos do programa fonte; por outro lado, novos problemas poderão ser introduzidos, como é o caso da "recuperação prematura" [9]. No caso considerado, isso pode acontecer se first(C) follow(A) $\neq \emptyset$.

Uma solução de compromisso foi adotada, mediante exame particular de cada caso. Não temos, porém a presunção de que nossa solução não possa ser ainda bastante melhorada.

Há ainda outro tipo de "modificações locais" na aplicação do algoritmo de Wirth, visando erros sintáticos que julgamos bastante freqüentes. Os casos mais comuns dividem-se em três classes gerais:

- símbolos trocados
- símbolos omissos
- símbolos em demasia.

Na seqüência, exemplificamos tais situações.

Símbolos trocados:

- Uso intercambiado de ":", "=" e ":="

- Uso intercambiado de ",", ";" e "."

- Uso intercambiado de "(", "[" e "{"

- Uso intercambiado de ")", "]" e "}"

- Uso de "^" em vez de "POINTER TO"

- Uso de ";" em vez de "|" (separando variantes em registros e comando case).

Símbolos omissos:

- ausência de ";" entre comandos, declarações, etc.

- ausência de identificador seguindo end (ao fim de módulos e subprogramas).

Símbolos em demasia:

- begin ao início de seqüência de comandos

- "(" e ")" englobando lista de campos em declaração de registro

- ";" antes de end em comando case ou variante de registro.

Damos no Apêndice A, como ilustração, algumas transcrições de rotinas do compilador. Naquelas rotinas, o parâmetro REM indica o conjunto de símbolos que podem seguir a construção em exame.

4.3. ESTRUTURAS DE DADOS E REPRESENTAÇÃO INTERNA

Durante o primeiro passo, o programa fonte é lido e traduzido para uma representação interna (em memória). Tal representação corresponde a um grafo direcionado que, como já mencionamos, chamamos (abusivamente) de "árvore do programa".

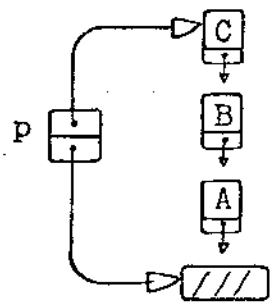
Cada nó da árvore representa, em geral, uma unidade sintática (símbolo terminal ou não terminal da linguagem). Ao mesmo tempo em que o programa fonte é lido, as análises sintática e de contexto vão sendo efetuadas. Assim, à medida em que a árvore é montada, é também percorrida. (Observe, por exemplo que a tabela de símbolos está espalhada na árvore). Certos trechos da árvore são manipulados segundo disciplina de pilha, outros de fila. Assim, localmente, o problema de manipulação de dados, se reduz a essas duas estruturas.

4.3.1. ESTRUTURAS DE DADOS BÁSICAS

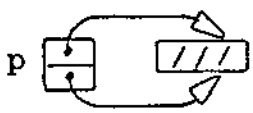
Os nós da árvore foram implementados como registros ligados por apontadores. Há na árvore, tipos diversos de nós, mas as pilhas e filas dos diferentes tipos de nós foram tratadas de modo análogo. As próximas figuras dão as linhas gerais de implementação das estruturas:

PILHAS:

pilha típica:

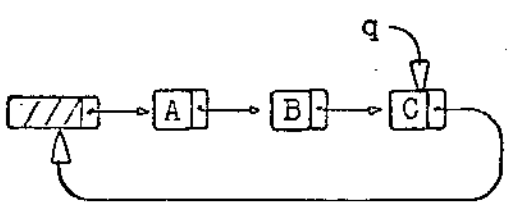


pilha vazia:

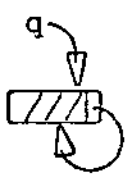


FILAS:

fila típica:



fila vazia:



Para inserção, remoção de elementos e outras operações, foram escritas rotinas específicas, de modo a separar o mais possível os diferentes níveis lógicos de acesso à árvore. Exemplificando, uma rotina que processa declarações, insere objetos numa fila, mas não precisa "saber" em detalhes a implementação de tal fila.

4.3.1.1. OPERAÇÕES SOBRE PILHAS

Descreveremos brevemente as diversas rotinas de manipulação de pilhas, quanto ao efeito visado. Lembrando que os nós de uma pilha podem ter tipos diferentes dos de outra pilha, vê-se que cada "rotina" abaixo corresponde na verdade, a várias rotinas do compilador (tipicamente uma para cada tipo de nó envolvido). Usamos no compilador, por razões de documentação, a convenção de o nome do tipo de um nó fazer parte (abreviadamente) do nome da rotina que o manipula.

Exemplos de tais nomes de rotinas são:

- PUSH_SYMT operação PUSH em nó do tipo SYMT_NODE
- ENQ_EXPR operação ENQUEUE em nó do tipo EXPR_NODE

Na descrição sumária dada a seguir, foram usadas algumas abreviações com significado óbvio. Não foram, por outro lado, explicitados os mecanismos de passagem de parâmetros (aliás, facilmente dedutíveis pelo leitor).

ROTINA:	FINALIDADE
function emptys (p): boolean	Verificar se p é pilha vazia.
procedure nulls (p)	Tornar vazia a pilha p.
procedure push (p, q)	Empilhar o nó q na pilha p.
procedure pop (p, q)	Desempilhar o nó q da pilha p.
procedure dels (p, q)	Remover de p o nó q (não necessariamente no topo).
procedure ssearch (p, value, q)	Buscar um nó de p, através do valor de um campo. Se o nó não for encontrado, q volta com NIL.
function sloc (p, n): ptr_node	Devolver um apontador para o n-ésimo nó de p.

4.3.1.2. OPERAÇÕES SOBRE FILAS

ROTINA:	FINALIDADE:
function emptyq (q): boolean	Verificar se q é fila vazia.
procedure nullq (q)	Tornar vazia a fila q.
procedure enq (q, p)	Inserir p como último elemento em q.
procedure deq (q, p)	Retirar o primeiro nó da fila q.
procedure delq (q, p)	Remover de q o nó p (não necessariamente no início).
procedure ssearch (q, value, p)	Buscar um nó de q, através do valor de um campo. Se o nó não for encontrado, p volta com valor NIL.
function qloc (q, n): ptr_node	Devolver um apontador para o n-ésimo nó de q.
function qcount (q): int	Calcular o número de nós de q.

4.3.2. A REPRESENTAÇÃO INTERMEDIÁRIA DO PROGRAMA

Vimos que a árvore do programa comporta vários tipos de nós. Os principais são três:

- nós da tabela de símbolos (SYMT_NODE)
- nós de comandos (STAT_NODE)
- nós de expressões (EXPR_NODE).

A cada um destes nós corresponde um tipo apontador (PTR_SYMT_NODE, PTR_STAT_NODE, PTR_EXPR_NODE).

Todo identificador usado no programa fonte aparece na constituição de um SYMT_NODE. Um SYMT_NODE relata o nome do objeto sendo descrito, sua categoria (i.e. se o objeto é constante, variável, subprograma, etc) e informações associadas a esta categoria.

Um STAT_NODE descreve o tipo de comando (atribuição, comando condicional, etc) e partes integrantes. Por exemplo, um nó que descreva um comando while tem um campo apontando para o EXPR_NODE que descreve a expressão condicional e outro para uma lista de STAT_NODE's, correspondente à seqüência de comandos repetidos no while em questão.

Um EXPR_NODE diz se a expressão é uma operação, indexação, derreferenciação, etc.

Para ilustração, reproduziremos em seguida os três nós citados, precedidos de declarações auxiliares.

```
TYPE
        INT = INTEGER;  BOOL = BOOLEAN;  ASCII = CHAR;

        SYMB_SET=       SET OF SYMBOL;

        NAME=           PACKED ARRAY [MIN_IDENT..MAX_IDENT] OF ASCII;

        PTR_NAME_NODE=  ^NAME_NODE;
        NAME_QUEUE=     PTR_NAME_NODE;
        NAME_SHEAD=     RECORD
                           TOP, BOTTOM: PTR_NAME_NODE
                        END;
        NAME_STACK=     NAME_SHEAD;
        NAME_NODE=      RECORD
                        NEXT:           PTR_NAME_NODE;
                        ID_NAME:        NAME
                        END;


        PTR_SYMT_NODE=  ^SYMT_NODE;
        PTR_SYMT_ITEM=  ^SYMT_ITEM;
        SYMT_QUEUE=     PTR_SYMT_ITEM;
        SYMT_ITEM=      RECORD
                           NEXT:        PTR_SYMT_ITEM;
                           ITEM_DESCR:  PTR_SYMT_NODE
                        END;


        PTR_STAT_NODE=  ^STAT_NODE;
        PTR_STAT_ITEM=  ^STAT_ITEM;
        STAT_QUEUE=     PTR_STAT_ITEM;
        STAT_ITEM=      RECORD
                           NEXT:        PTR_STAT_ITEM;
                           ITEM_DESCR:  PTR_STAT_NODE
                        END;


        PTR_EXPR_NODE=  ^EXPR_NODE;
        PTR_EXPR_ITEM=  ^EXPR_ITEM;
        EXPR_QUEUE =    PTR_EXPR_ITEM;
        EXPR_ITEM=      RECORD
                           NEXT:        PTR_EXPR_ITEM;
                           ITEM_DESCR:  PTR_EXPR_NODE
                        END;
```

```
PTR_LBL_NODE=    ^LBL_NODE;
LBL_QUEUE=       PTR_LBL_NODE;
LBL_NODE=        RECORD
                 NEXT:              PTR_LBL_NODE;
                 CASE SIMPLE_LBL:BOOL OF
                    TRUE:           (LBL_VALUE:
                                    INT);
                    FALSE:          (LW_LBL, UP_LBL:
                                    INT);
                 END;


PTR_REC_ALT_NODE=
                 ^REC_ALT_NODE;
REC_ALT_QUEUE=   PTR_REC_ALT_NODE;
REC_ALT_NODE=    RECORD
                 NEXT:              PTR_REC_ALT_NODE;
                 LBL_LIST:          LBL_QUEUE;
                 DECL_LIST:         SYMT_QUEUE
                 END;

PTR_CLAUSE_NODE=^CLAUSE_NODE;
CLAUSE_QUEUE=    PTR_CLAUSE_NODE;
CLAUSE_NODE=     RECORD
                 NEXT:              PTR_CLAUSE_NODE;
                 LBL_LIST:          LBL_QUEUE;
                 STAT_LIST:         STAT_QUEUE
                 END;


SYMT_SHEAD=      RECORD
                    TOP,BOTTOM:     PTR_SYMT_ITEM
                 END;

SYMT_STACK=      SYMT_SHEAD;


EXPR_SHEAD=      RECORD
                    TOP,BOTTOM:     PTR_EXPR_ITEM
                 END;
EXPR_STACK=      EXPR_SHEAD;
```

```
PSEUDO_NAME= (
PD_ABS,          PD_ADDRESS,      PD_ADR,          PD_ASH,          PD_BITSET,       PD_BOOL,
PD_BYTE,         PD_BYTE_SIZE,    PD_CAP,          PD_CARD,         PD_CARD_TO_SH,   PD_CHAR,
PD_CHR,          PD_DEC,          PD_DISPOSE,      PD_ESCAPE,       PD_EXCL,         PD_FALSE,
PD_FIRST,        PD_HALT,         PD_HIGH,         PD_INC,          PD_INCL,         PD_INPUT,
PD_INT,          PD_INTERRUPT,    PD_INT_TO_SH,    PD_IOTRANSFER,   PD_LAST,         PD_LG_ADDRESS,
PD_LG_ADR,       PD_LISTEN,       PD_LOW,          PD_MAXSH_CARD,   PD_MAXSH_INT,    PD_MAX_CARD,
PD_MAX_INT,      PD_MINSH_CARD,   PD_MINSH_INT,    PD_MIN_CARD,     PD_MIN_INT,      PD_NEW,
PD_NEWPROCESS,   PD_NIL,          PD_NULL_NAME,    PD_ODD,          PD_ORD,          PD_OUTPUT,
PD_PRED,         PD_PROC,         PD_PROCESS,      PD_REGISTER,     PD_RELEASE,      PD_RESERVE,
PD_RT11CALL,     PD_SEMAPHORE,    PD_SH_BITSET,    PD_SH_CARD,      PD_SH_INT,       PD_SH_TO_CARD,
PD_SH_TO_INT,    PD_SIZE,         PD_SUCC,         PD_SYSRESET,     PD_SYSTEM,       PD_TRANSFER,
PD_TRUE,         PD_TSIZE,        PD_VAL,          PD_WAIT,         PD_WORD,         PD_WORD_SIZE);

IDENT_CLASS=
(PRIMIT_ID,      SYSTEM_ID,       USER_ID);

SYMT_CLASS=
(SYMT_CONST,     SYMT_MODULE,     SYMT_PROGR,      SYMT_SELEC,      SYMT_SUBPR,      SYMT_TYPE,
SYMT_UNDEF,      SYMT_VAR);

SYMT_TYPE_CLASS=
(ST_TY_DYN_ARR,  ST_TY_ENUM,      ST_TY_FORW,      ST_TY_FXD_ARR,   ST_TY_LG_PTR,    ST_TY_OPAQUE,
ST_TY_POINTER,   ST_TY_RECORD,    ST_TY_SET,       ST_TY_SUBPR,     ST_TY_SUBRAN,    ST_TY_S_OPAQUE,
ST_TY_TYNAM);

MODULE_CLASS=
(COMP_MOD,       DEF_MOD,         IMPL_MOD,        MAIN_MOD,        SIMPLE_MOD);

VAR_CLASS=
(CONST_VAR,      FIXED_VAR,       PROP_VAR,        REF_PAR,         VALUE_PAR);

SUBPR_CLASS=
(FUNC,           PROCED);

PTR_PAR_NODE=    ^PAR_NODE;
PAR_QUEUE=       PTR_PAR_NODE;
PAR_NODE=        RECORD
                    NEXT:        PTR_PAR_NODE;
                    PAR_TYPE:    PTR_SYMT_NODE;
                    PAR_MODE:    VAR_CLASS
                 END;
```

```
(*      N O'S   D A   T A B E L A     D E     S'I M B O L O S   *)

SYMT_NODE=      RECORD

                ID_NAME:          NAME;

                ID_CAT:           IDENT_CLASS;

                ID_PS_NAME:       PSEUDO_NAME;

                LEX_LEVEL:        INT;

                MOD_LEVEL:        INT;

                USED:             BOOL;

                INCOMPL:          BOOL;

                MUST_QUALIF:      BOOL;

                FROM_MODULE:      PTR_SYMT_NODE;

                BELONGS_TO:       PTR_SYMT_NODE;

                CASE CAT:         SYMT_CLASS OF

                   SYMT_CONST:    (CONST_DESCR:    PTR_EXPR_NODE);

                   SYMT_VAR:      (VAR_TYPE:       PTR_SYMT_NODE;
                                   VAR_CAT:        VAR_CLASS;
                                   VAR_OFF:        INT;
                                   VAR_BASE:       INT;
                                   VAR_VAL:        PTR_STR_CONST);

                   SYMT_SELEC:    (SELEC_OFF:      INT;
                                   PARENT_TY:      PTR_SYMT_NODE;
                                   SELEC_TY:       PTR_SYMT_NODE;
                                   CASE VARIANT_FLAG: BOOL OF
                                      TRUE:        (SUB_LIST:      REC_ALT_QUEUE);
                                      FALSE:       ());

                   SYMT_SUBPR:    (SUBPR_LN:                INT;
                                   SUBPR_TY:                PTR_SYMT_NODE;
                                   SUBPR_FPAR_LIST:         SYMT_QUEUE;
                                   SUBPR_DECL_LIST:         SYMT_QUEUE;
                                   SUBPR_STAT_LIST:         STAT_QUEUE;
                                   SUBPR_SEGLAB,
                                   SUBPR_INLAB,
                                   SUBPR_FINLAB:            INT_LABEL;
                                   SUBPR_RESULT:            PTR_SYMT_NODE);

                   SYMT_MODULE:   (MODULE_LN:               INT;
                                   MODULE_CAT:              MODULE_CLASS;
                                   PRIORITY:                INT;
                                   IMPORT_LIST:             SYMT_QUEUE;
                                   EXPORT_LIST:             SYMT_QUEUE;
                                   QUALIFLAG:               BOOL;
                                   MOD_DECL_LIST:           SYMT_QUEUE;
                                   MOD_STAT_LIST:           STAT_QUEUE;
                                   MOD_FINLAB:              INT_LABEL);
```

```
SYMT_PROGR:   (EXTERN_MOD_LIST:        PTR_SYMT_ITEM;
               INTERN_MOD:             PTR_SYMT_NODE);

SYMT_TYPE:    (TYPE_SIZE:              INT;

              CASE TYPE_CAT:           SYMT_TYPE_CLASS OF
                 ST_TY_TYNAM,

                 ST_TY_OPAQUE,
                 ST_TY_S_OPAQUE:       (TYNAM_DESCR:    PTR_SYMT_NODE);

                 ST_TY_ENUM:           (SCALAR_LIST:    SYMT_QUEUE;
                                        LAST_SCALAR:     INT);

                 ST_TY_SUBRAN:         (SUBRAN_TY:      PTR_SYMT_NODE;
                                        SUBRAN_LW_BD,SUBRAN_UP_BD:
                                                         INT);

                 ST_TY_POINTER,
                 ST_TY_LG_PTR:         (POINTED_TY:     PTR_SYMT_NODE);

                 ST_TY_SET:            (BASE_TY:        PTR_SYMT_NODE);

                 ST_TY_SUBPR:          (SUBPR_CAT:      SUBPR_CLASS;
                                        FR_TY_LIST:      PAR_QUEUE;
                                        RESULT_TY:       PTR_SYMT_NODE);

                 ST_TY_FXD_ARR:        (BOUNDS_TY:      PTR_SYMT_NODE;
                                        FXD_ELEM_TY:     PTR_SYMT_NODE);

                 ST_TY_DYN_ARR:        (DYN_ELEM_TY:    PTR_SYMT_NODE);

                 ST_TY_RECORD:         (LINEAR_FIELDS,
                                        FIELD_SEQ:       SYMT_QUEUE))

END; % RECORD SYMT_NODE \
```

```
                        (*      N Ó'S     D E     C O M A N D O S       *)

STAT_CLASS=
        (EMPTY_ST,      ASSIGN_ST,              PCALL_ST,       IF_ST,
        WHILE_ST,       REPEAT_ST,              LOOP_ST,        FOR_ST,
        CASE_ST,        WITH_ST,                EXIT_ST,        RETURN_ST,
        UNDEF_ST);

PTR_ALTERN=     ^IF_STAT_ALTERN_NODE;
ALTERN_QUEUE=   PTR_ALTERN;
IF_STAT_ALTERN_NODE=
        RECORD
           NEXT:        PTR_ALTERN;
           CONDITION:   PTR_EXPR_NODE;
           STAT_LIST:   STAT_QUEUE
        END;

STAT_NODE=      RECORD
                SOURCE_LN_NBR:  INT;

                CASE CAT:               STAT_CLASS OF

                   ASSIGN_ST:           (LEF_EXPR, RIG_EXPR:    PTR_EXPR_NODE);

                   PCALL_ST:            (PROC_DESCR:            PTR_EXPR_NODE;
                                        AC_PAR_LIST:            EXPR_QUEUE);

                   IF_ST:               (ALTERN_LIST:           ALTERN_QUEUE;
                                        ELSE_S_LIST:            STAT_QUEUE);

                   WHILE_ST:            (WHILE_COND:            PTR_EXPR_NODE;
                                        WHILE_S_LIST:           STAT_QUEUE);

                   REPEAT_ST:           (REPEAT_S_LIST:         STAT_QUEUE;
                                        REPEAT_COND:            PTR_EXPR_NODE);

                   LOOP_ST:             (LOOP_S_LIST:           STAT_QUEUE;
                                        LOOP_LAB:               INT_LABEL);

                   FOR_ST:              (FOR_CTL_VAR:           PTR_SYMT_NODE;
                                        FROM_EXPR, TO_EXPR, BY_EXPR:
                                                                PTR_EXPR_NODE;
                                        FOR_S_LIST:             STAT_QUEUE);

                   CASE_ST:             (SELEC_EXPR:            PTR_EXPR_NODE;
                                        CLAUSE_LIST:            CLAUSE_QUEUE;
                                        CLAUSE_LW_BD, CLAUSE_UP_BD:
                                                                INT;
                                        OTHERS_S_LIST:          STAT_QUEUE);

                   WITH_ST:             (WITH_SELEC:            PTR_EXPR_NODE;
                                        WITH_S_LIST:            STAT_QUEUE);

                   RETURN_ST:           (RETURN_EXPR:           PTR_EXPR_NODE;
                                        RETURN_UNIT:            PTR_SYMT_NODE);

                   EXIT_ST:             (FROM_LOOP:             PTR_STAT_NODE)

                END; % RECORD STAT_NODE \
```

```
                (*      N O'S     D E     E X P R E S S O E S   *)

EXPR_CLASS=     (CONST_EXPR,    VAR_EXPR,       SEL_FIELD_EXPR, INDXD_ELEM_EXPR,
                 DEREF_EXPR,    FUNC_EXPR,      UN_OP_EXPR,     BIN_OP_EXPR,
                 WITH_EXPR,     SUBPR_EXPR,     TYPE_EXPR,      UNDEF_EXPR);

EXPR_NODE=      RECORD

                TY:                     PTR_SYMT_NODE;

                ASSIGNABLE:             BOOL;

                DESCR:                  PTR_RSLT_DESCR;

                CASE CAT:               EXPR_CLASS OF

                   CONST_EXPR:          (VALUE:          INT);

                   VAR_EXPR:            (VAR_DESCR:      PTR_SYMT_NODE);

                   SEL_FIELD_EXPR:      (REC_DESIGN:     PTR_EXPR_NODE;
                                         SEL_FIELD:       PTR_SYMT_NODE);

                   INDXD_ELEM_EXPR:     (ARRAY_DESIGN:   PTR_EXPR_NODE;
                                         INDEX:           PTR_EXPR_NODE);

                   DEREF_EXPR:          (POINTING_EXPR:  PTR_EXPR_NODE);

                   FUNC_EXPR:           (FUNC_DESCR:     PTR_EXPR_NODE;
                                         AC_PAR_LIST:     EXPR_QUEUE);

                   UN_OP_EXPR:          (UN_OPERATOR:    SYMBOL;
                                         OPERAND:         PTR_EXPR_NODE);

                   BIN_OP_EXPR:         (BIN_OPERATOR:   SYMBOL;
                                         LEF_OPERAND,RIG_OPERAND:
                                                          PTR_EXPR_NODE);

                   WITH_EXPR:           (WITH_DESIGN:    PTR_EXPR_NODE;
                                         WITH_SELECTOR:   PTR_SYMT_NODE);

                   SUBPR_EXPR:          (SUBPR_DESCR:    PTR_SYMT_NODE);

                   TYPE_EXPR:           (TYPE_DESCR:     PTR_SYMT_NODE)

                END; % RECORD EXPR_NODE \
```

4.3.2.1. NÓS DA TABELA DE SÍMBOLOS

Daremos algumas explicações sobre os campos do nó SYMT_NODE. (Ver declarações de tipos em 4.3.2).

ID-NAME:

É o nome do objeto. Pode ser inicialmente conhecido pelo compilador (caso de objetos primitivos e do sistema) ou criado pelo programador. O nome pode também ser nulo. Isso ocorre quando na definição de um tipo faz-se referência a outros, sem especificação individual. Exemplo: na declaração "var V: array [1..10] of INTEGER" houve implicitamente a criação do tipo (anônimo) do índice do vetor.

ID-CAT:

Especifica a categoria do identificador, que pode ser:

- USER_ID se introduzido pelo programador,
- PRIMIT_ID se nome de objeto primitivo,
- SYSTEM_ID se nome de objeto do sistema.

ID-PS-NAME:

("Pseudo-name"). Para facilidade de referência, objetos primitivos e objetos do sistema (daqui em diante, ambos chamados "objetos iniciais") têm identificadores associados, que constam da enumeração PSEUDO_NAME.

LEVEL:

Nível léxico.

USED:

A não ser que a diretiva FORCE_GENERATION seja ativada, o segundo passo não gera código para objetos não referenciados. Para este fim, durante o primeiro passo, objetos são marcados como "usados", à medida em que são referenciados.

CAT:

É a "categoria" do objeto. Classifica o objeto como uma constante, um tipo, etc. Os identificadores auxiliares para a classificação constam da enumeração SYMT_CLASS. A categoria "SYMT_UNDEF" é reservada aos objetos indefinidos. Isso está associado à recuperação de erros de contexto.

SYMT-CONST:
O objeto é uma constante.

SYMT-VAR:
Variáveis e parâmetros formais.

SYMT-SELEC:
Campos de registro.

SYMT-SUBPR:
Procedimentos e funções.

SYMT-MODULE:
Módulos, sendo que MODULE-CAT especifica a categoria do módulo, de acordo com a enumeração MODULE-CLASS. Tais categorias referem-se respectivamente a:

módulo de definição,
módulo de implementação,
módulo principal,
módulo interno e
módulo composto.

Este último é criado pelo compilador para substituir uma dupla definição/implementação.

SYMT-PROGR:
É a unidade de compilação mais global: o "programa".

SYMT-TYPE:

Tipos.

TYPE-SIZE:

Tamanho (em número de "bytes") de um objeto do tipo em questão. Note que esta especificação é dependente de implementação.

TYPE-CAT:

Classifica um tipo, de acordo com a enumeração SYMT_TYPE_CLASS

4.3.2.2. NÓS DE COMANDOS

SOURCE-LN-NBR:

Número da linha do programa fonte onde se inicia o comando. A finalidade é controle de erros de execução.

CAT:

Categoria do comando. Classifica o comando segundo a enumeração STAT_CLASS. A categoria UNDEF_STAT tem função análoga à de SYMT_UNDEF num SYMT_NODE.

4.3.2.3. NÓS DE EXPRESSÕES

TY:

É o tipo da expressão (toda expressão tem um tipo).

ASSIGNABLE:

Diz se a expressão pode ocorrer à esquerda num comando de atribuição.

CAT:

Categoria da expressão. Classifica a expressão mediante a enumeração EXPR_CLASS. Sobre UNDEF_EXPR, vale a mesma explicação de UNDEF_STAT.

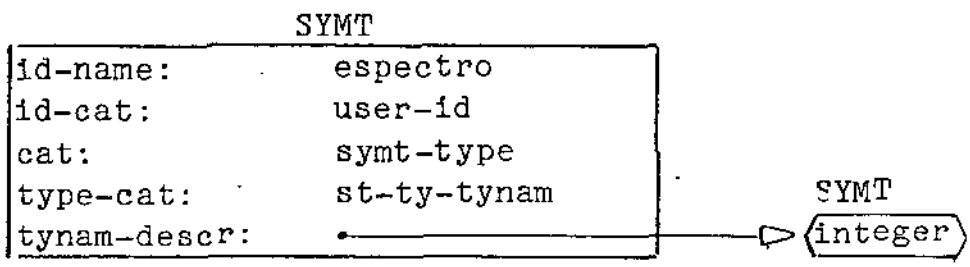
VALUE:

O valor de uma expressão constante é dado internamente por um número inteiro. A única exceção são cadeias de caracteres, que têm representação especial, conforme veremos.

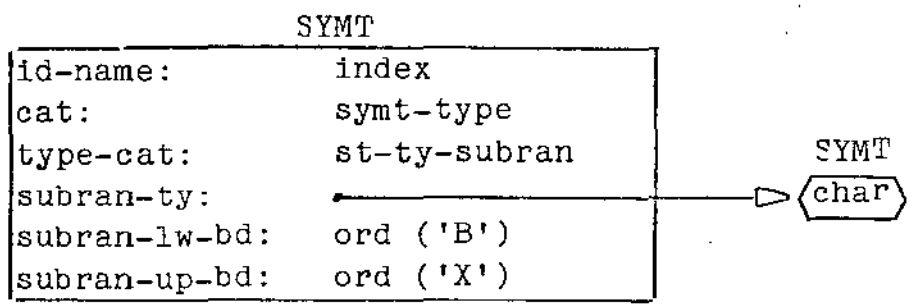
4.3.2.4. EXEMPLOS PICTÓRICOS

Para ilustração, serão dadas representações pictóricas de alguns exemplos de trechos da árvore. Retângulos designarão nós da árvore que tenham especificação razoavelmente completas. Hexágonos e nomes entre aspas são abreviaturas dos retângulos. Simplificações de ordem didática foram feitas na representação de listas.

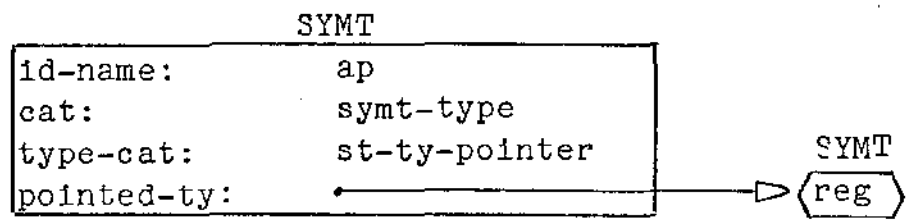
```
type espectro = integer;
```



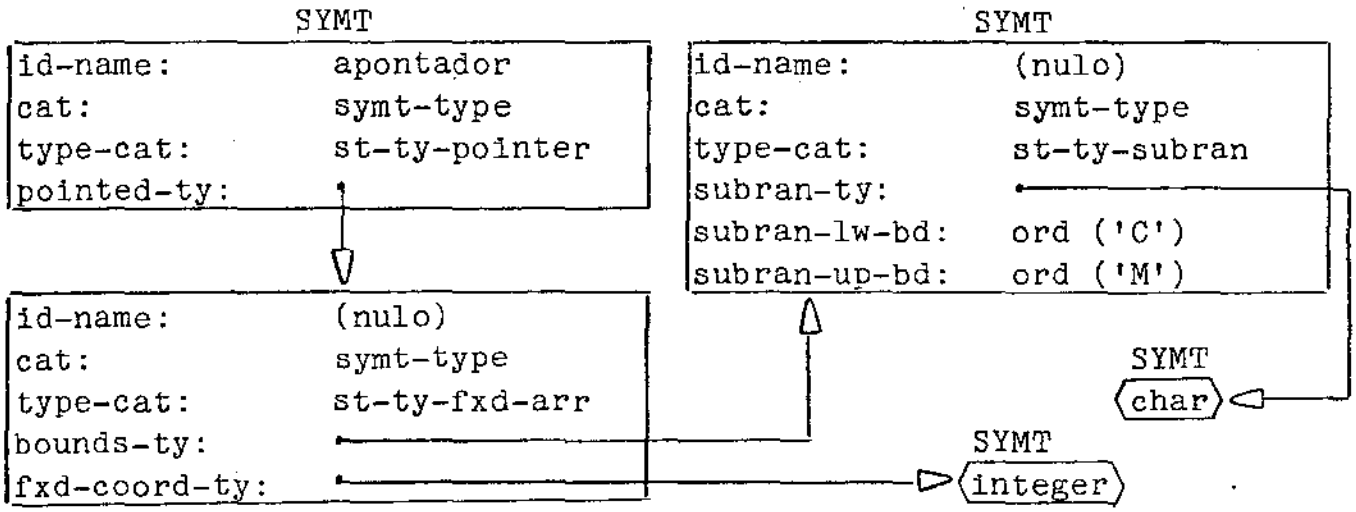

```
type index = ['B'..'X']
```



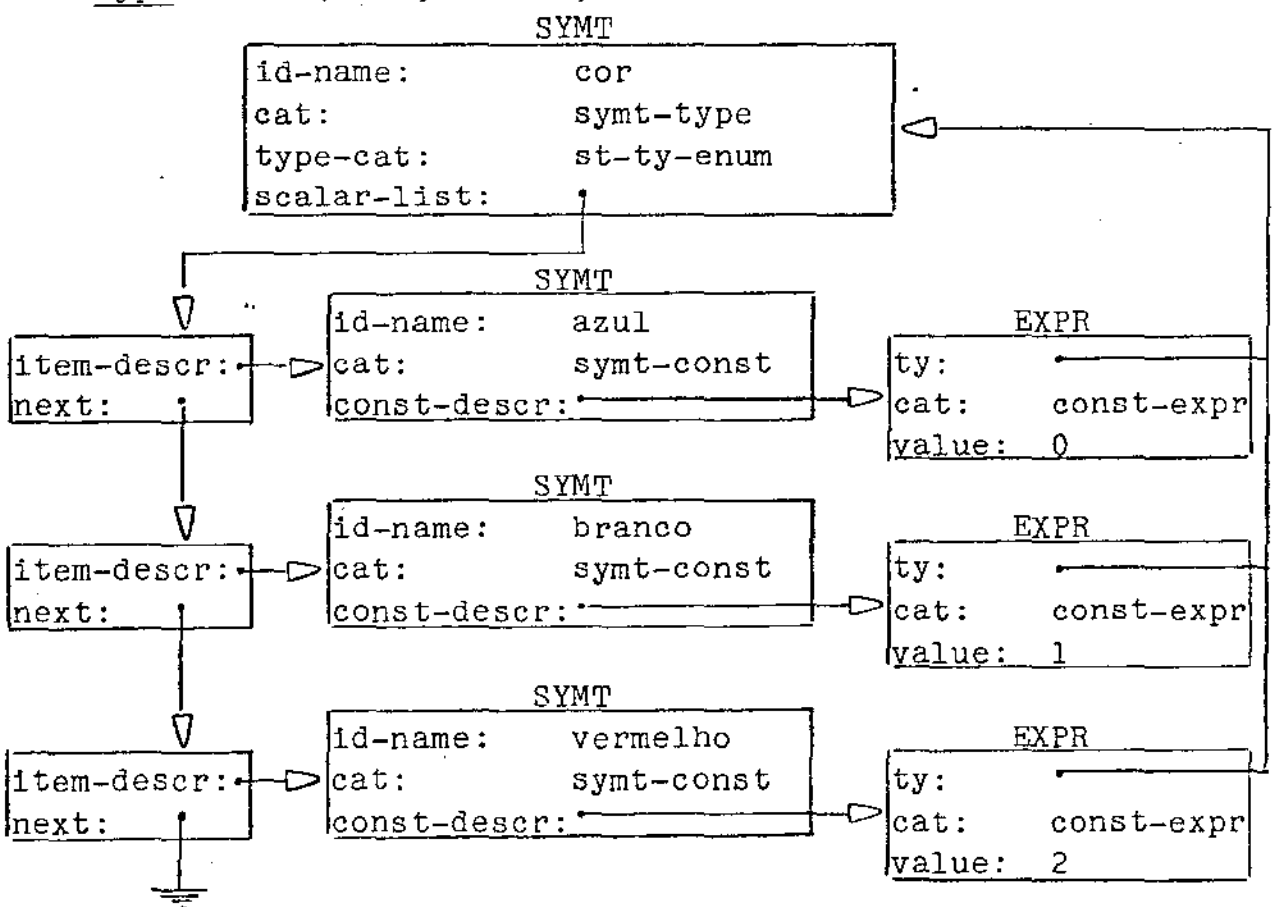

```
type ap = pointer to reg
```



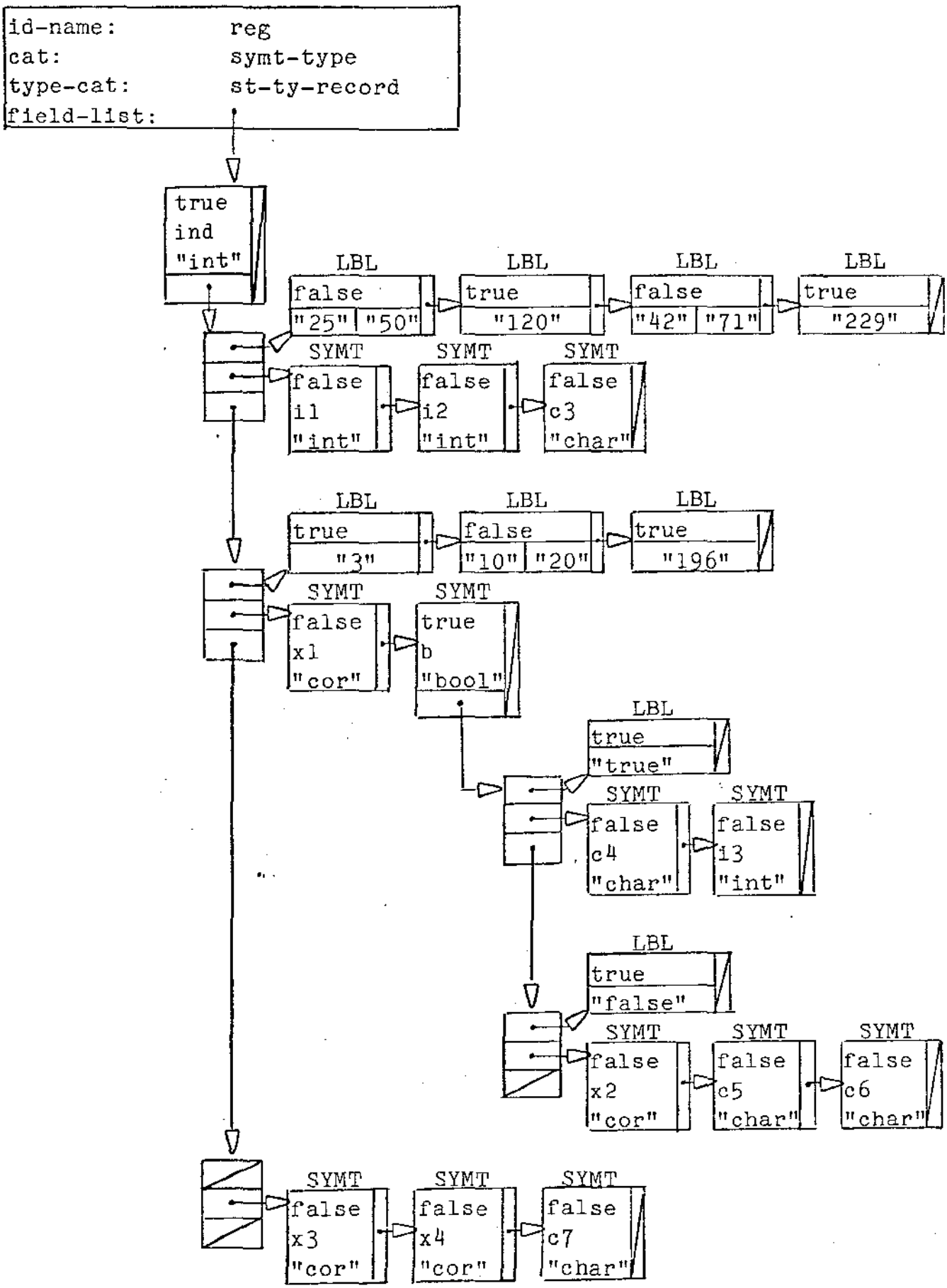
```
type apontador = pointer to
                     array ['C'..'M'] of
                         integer
```



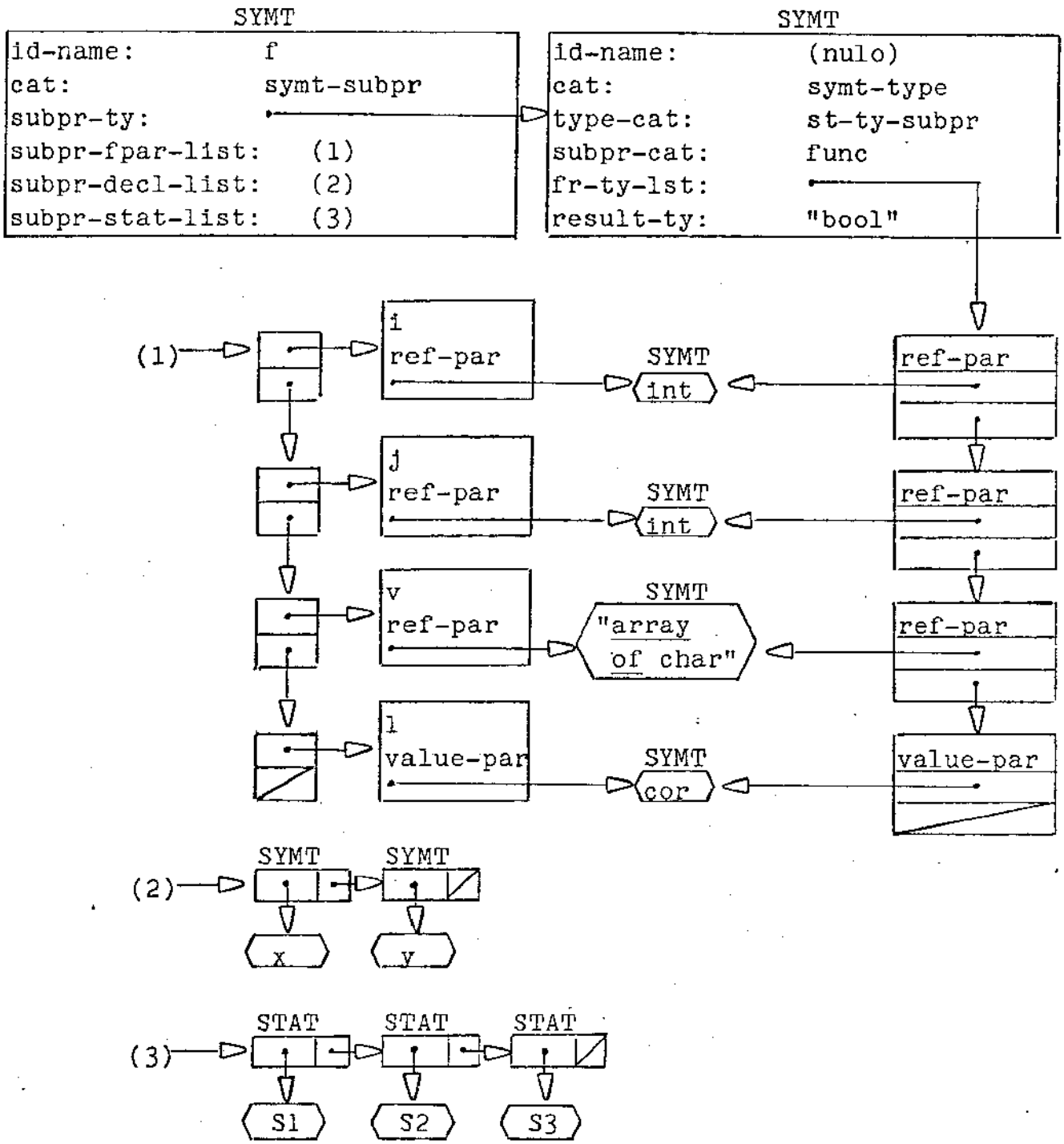

```
type cor = (azul, branco, vermelho)
```



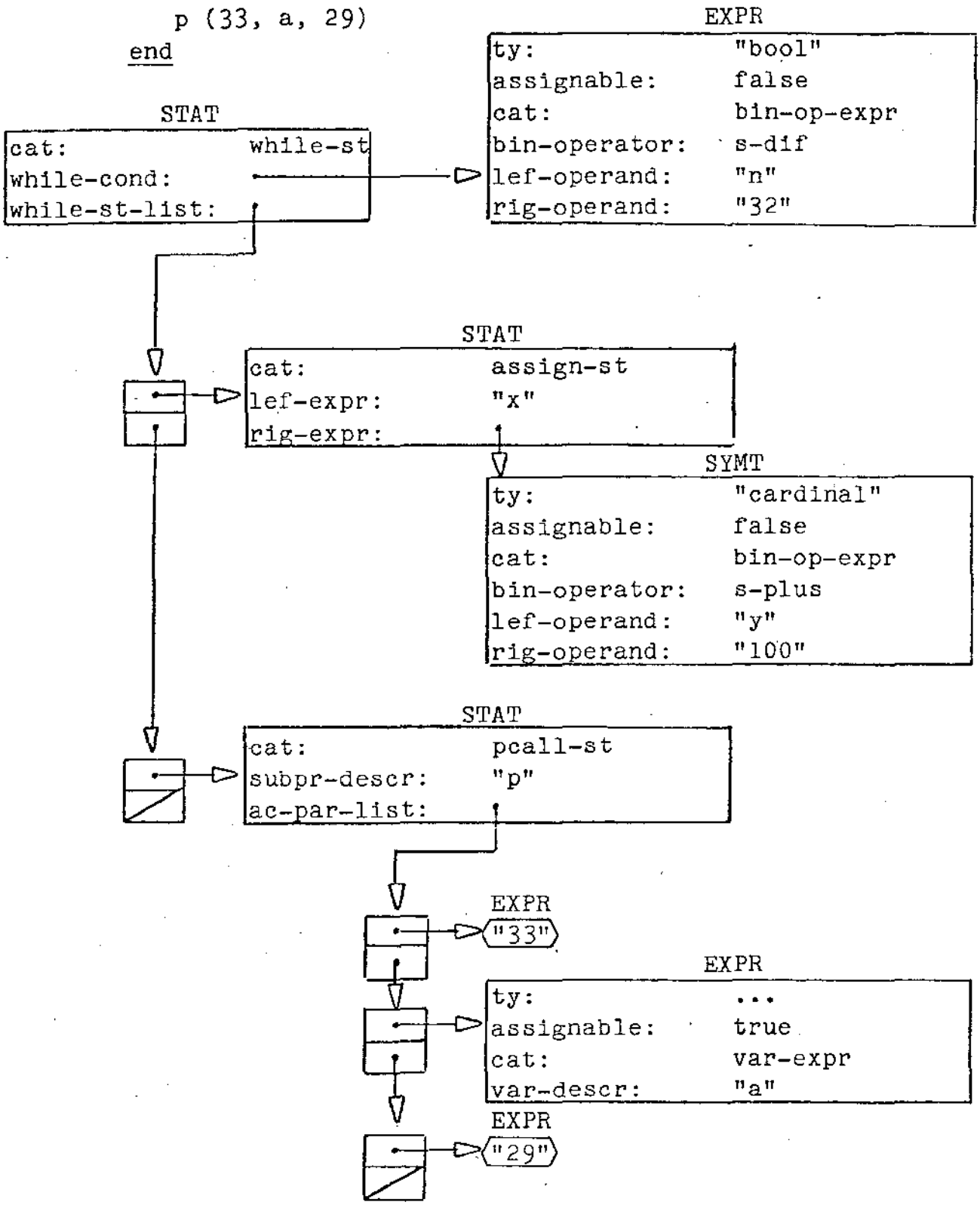
```
type reg =      record
                case ind: int of

                   25..50, 120, 42..71, 229:
                        i1, i2: int;
                        c3:     char

                   3, 10..20, 196:
                        x1:     cor;

                        case b: bool of
                           true:        c4:     char;
                                        i3:     int;

                           false:       x2:     cor;
                                        c5, c6: char
                        end   (* case *)
                   else
                        x3, x4: cor;
                        c7:     char

                end (* case *)

                end (* record *)
```


id-name: reg
cat: symt-type
type-cat: st-ty-record
field-list:
true
ind
"int"
LBL
false
"25" "50"
LBL
true
"120"
LBL
false
"42" "71"
LBL
true
"229"
SYMT
false
i1
"int"
SYMT
false
i2
"int"
SYMT
false
c3
"char"
LBL
true
"3"
LBL
false
"10" "20"
LBL
true
"196"
SYMT
false
x1
"cor"
SYMT
true
b
"bool"
LBL
true
"true"
SYMT
false
c4
"char"
SYMT
false
i3
"int"
LBL
true
"false"
SYMT
false
x2
"cor"
SYMT
false
c5
"char"
SYMT
false
c6
"char"
SYMT
false
x3
"cor"
SYMT
false
x4
"cor"
SYMT
false
c7
"char"

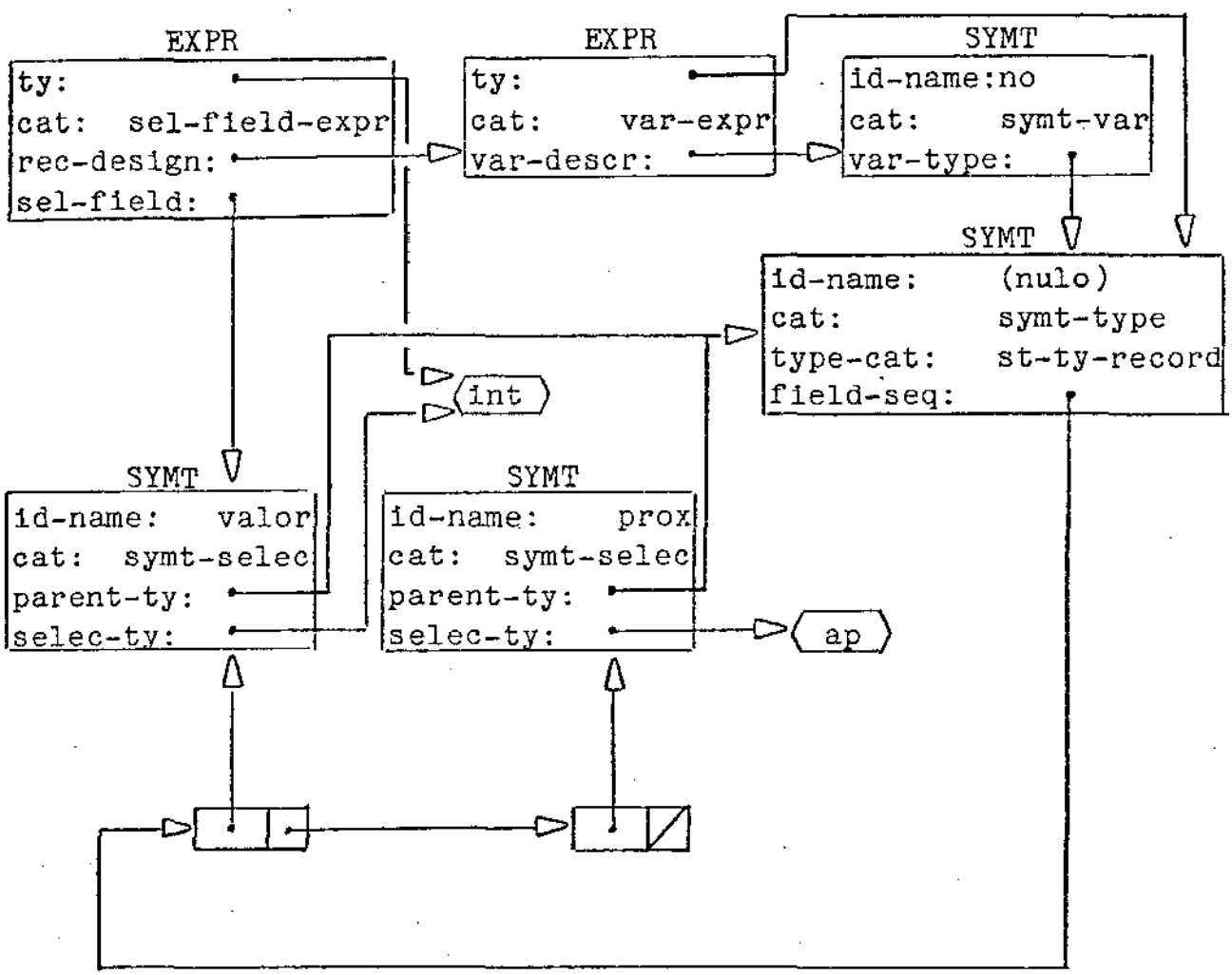
```
procedure f (var i,j: int; var v: array of char;
                  l: cor): bool;
   var x,y: bool;
begin
   S1; S2; S3
end f
```



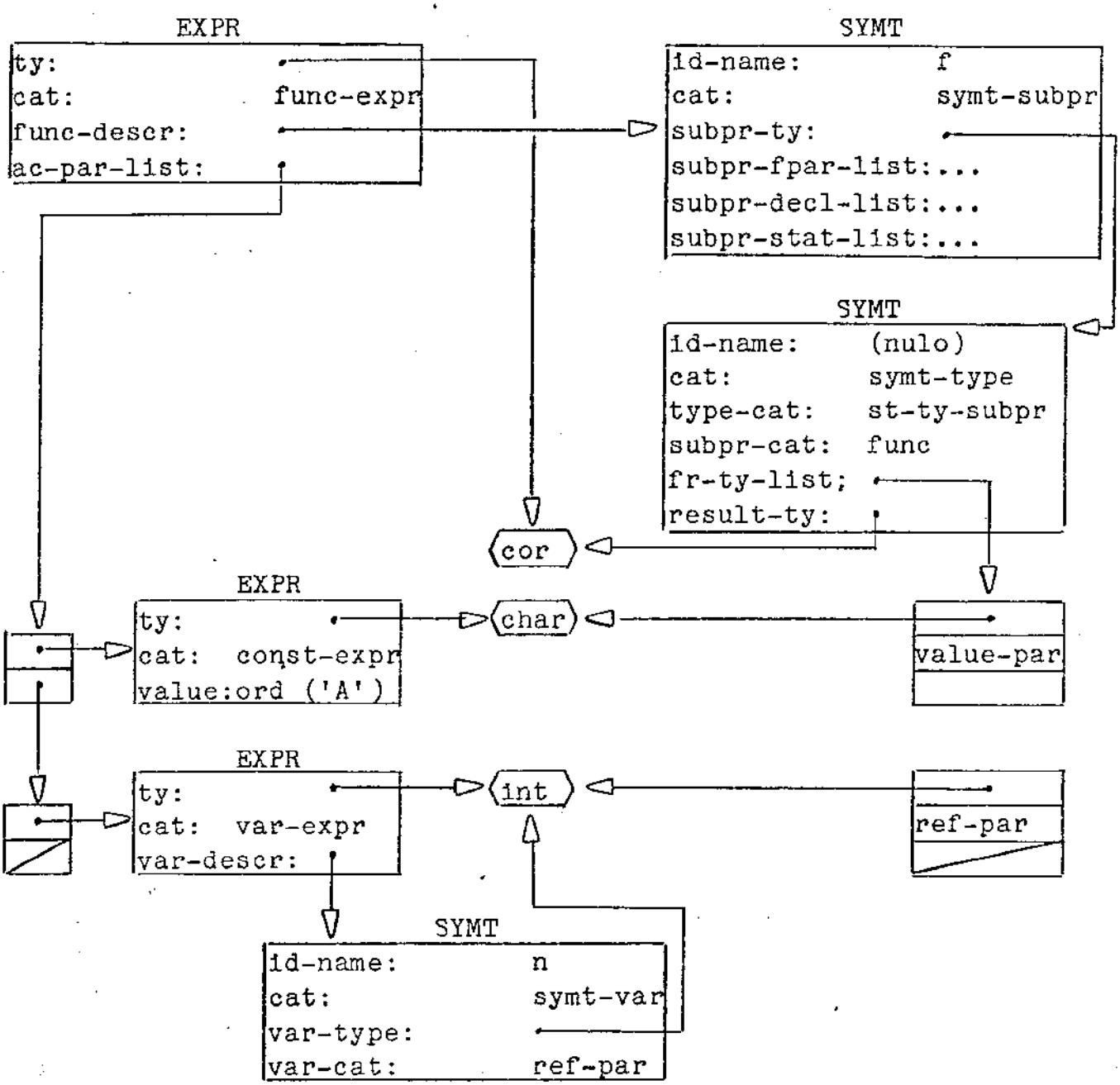
```
while n <> 32 do
   x := y + 100;
   p (33, a, 29)
end
```



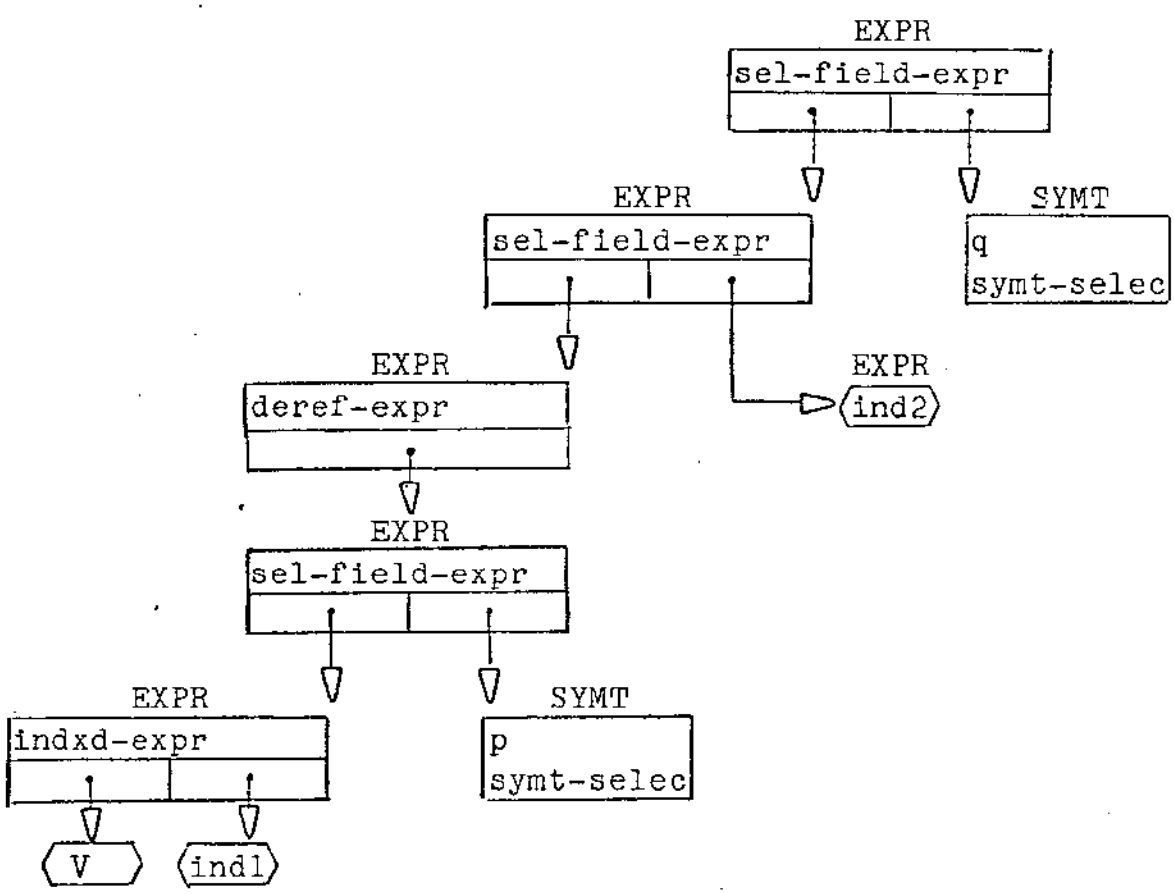
```
no . valor        (* Onde no:      record
                                   valor:  int;
                                   prox:   ap
                                   end              *)
```



```
f ('A', n)        (* Onde
                     procedure f (c: char; var i: int):
                     cor *)
```



V [ind1] . p ^ [ind2] . q



4.4. ANÁLISE DE CONTEXTO

Situações de confronto entre uso de um objeto e sua especificação são examinadas na chamada análise de contexto. Resulta que o problema de compatibilidade é crucial.

Como as ações do programador são secundadas pelas do compilador, podemos distinguir entre um ponto de vista "externo" e um "interno" durante a compilação.

Assim temos:

(i) especificação de objetos que é ação do programa (externa).

(ii) representação interna dos objetos declarados e fixação de mecanismos de acesso (ação interna correspondente).

(iii) referência a objetos em expressões, comandos, etc (ações externas).

(iv) verificação de compatibilidade (ação interna correspondente).

A separação entre (i) e (iii), bem como entre (ii) e (iv) é meramente de ordem expositiva. São, porém, ações interativas. Assim, por exemplo, uma referência a um objeto pode estar servindo à especificação de outro. Por outro lado, ao ser verificada compatibilidade, a situação típica é a de estar-se construindo novos objetos internamente e adequando-lhes mecanismos de acesso convenientes.

Associada à verificação de compatibilidade, há a recuperação de erros de contexto que, merecendo consideração especial, será tratada em item próprio. No Apêndice A constam algumas rotinas que ilustram o tratamento de erros de contexto.

4.4.1. ESPECIFICAÇÃO DE OBJETOS DO PROGRAMA

Num programa em MODULA-2 há objetos previamente conhecidos pelo compilador (objetos "iniciais") e objetos criados pelo programador.

Os objetos iniciais são ainda divididos em:

- Objetos primitivos e
- Objetos do sistema.

Objetos primitivos são aqueles definidos em MODULA-2, de uso livre e com significado quase totalmente independente da particular implementação. Objetos do sistema, ao contrário, são conhecidos pelo compilador, mas correspondem à implementação em questão. A finalidade destes últimos é prover comunicação com o computador objeto de um modo mais completo.

Definições e declarações são maneiras de o programador criar objetos novos. Importações e exportações são mecanismos de controle da visibilidade e acesso a objetos; por meio delas, especifica-se uso de objetos pertencentes a módulos.

4.4.1.1. OBJETOS INICIAIS

Dentro de um módulo só se tem acesso a objetos locais e objetos importados. Assim, por conveniência, em MODULA-2 especifica-se que todo módulo importa implicitamente os objetos primitivos. Os objetos do sistema, porém, devem ser importados explicitamente. Isto é conseguido com "from SYSTEM import ...", i.e., através de importação qualificada de objetos do módulo SYSTEM. Tal módulo é um dos objetos primitivos.

Vimos que cada nó correspondente a um módulo tem um campo EXPORT_LIST que relata os objetos exportados pelo módulo em questão. Assim, os objetos do sistema figuram na EXPORT_LIST do módulo SYSTEM. O compilador trata, entretanto, de modo particular certos objetos do sistema. Há vários motivos para isso, por

exemplo:

- Há rotinas do sistema com número variável de parâmetros (ex: "INC" que pode ter 1 ou 2 parâmetros).

- Há rotinas do sistema cuja chamada envolve ações especiais (ex: NEW que gera uma chamada a um procedimento ALLOCATE do usuário).

4.4.1.2. DEFINIÇÕES E DECLARAÇÕES

Em MODULA-2 certos objetos podem ser especificados em duas etapas: uma descrição inicial num módulo de definição e uma descrição completa no módulo de implementação correspondente. Tal se dá com tipos e subprogramas. Um tipo pode ser "opaco" no módulo de definição (i.e., dá-se a conhecer ali apenas seu nome e tamanho). Um subprograma pode ter só seu cabeçalho conhecido num módulo de definição (i.e., lista dos tipos de parâmetros com respectivos mecanismos de passagem e tipo de valor devolvido, se função). Em qualquer destes casos, cabe ao compilador verificar a coerência entre as duas etapas da descrição.

Quando o compilador encontra um módulo de implementação, ele localiza (pelo nome) o módulo de definição correspondente (que já deveria ter sido declarado).

MODULA-2 especifica que objetos conhecidos num módulo de definição (i.e., objetos locais e objetos importados), também o sejam pelo módulo de implementação correspondente. Isto porque a finalidade de um módulo de implementação é de complementar especificações do módulo de definição correspondente. Outros objetos, porém, poderão ser declarados no módulo de implementação. Ao fim do módulo de implementação, todas as especificações parciais encontradas no módulo de definição correspondente, deverão ter sido completadas.

Tomou-se a decisão, na implementação do compilador, de

"fundir-se" um par definição/implementação num único "módulo composto". Criamos, para tal, uma quinta categoria de módulo: COMP_MOD ("compound module").

4.4.1.3. IMPORTAÇÕES E EXPORTAÇÕES

Conforme já foi visto, módulos controlam visibilidade a objetos, mas não criam novo escopo no sentido em que procedimentos o fazem. i.e., objetos locais a um módulo são locais (possivelmente de modo invisível) ao escopo no qual está declarado o módulo. A fronteira de visibilidade pode ser rompida de duas maneiras: "de fora" ou "de dentro". O primeiro caso corresponde a importações que fazem com que objetos do escopo circundante passem a ser acessíveis no interior do módulo. No segundo caso temos exportações efetuadas por módulos locais ao módulo em questão.

MODULA-2 especifica que exportações não podem ser efetuadas pelo módulo principal (mesmo porque, tal não faria sentido), nem por módulos de implementação, já que os módulos de definição têm o papel de interface com outros módulos. A restrição, no caso de módulos de definição é que as exportações sejam qualificadas. É uma medida de segurança para o caso de objetos com mesmo nome serem exportados de diferentes módulos de definição.

4.4.2. MECANISMOS DE ACESSO

O acesso aos objetos (por exemplo: consultas à "tabela de símbolos") é controlado, na nossa implementação, por "listas de acessibilidade", nas quais se mantém disciplina de pilha: o último objeto declarado é o primeiro encontrado numa busca. Isso resolve automaticamente conflito de objetos homônimos, de nível léxico diferente. A rotina que compila programa (progr) tem uma variável local GAL ("global access list") que é a lista de acessibilidade mais externa. Esta lista é passada como parâmetro para as rotinas DEF_MOD, IMPL_MOD e PROGRAM_MOD. As rotinas que compilam módulos (exceto o módulo principal) e subprogramas,

têm um parâmetro EAL ("external access list") e uma variável local IAL ("internal access list").

Módulos modificam EAL através de exportações. Subprogramas não modificam EAL.

À entrada das rotinas que compilam procedimentos, IAL é inicializada com o valor de EAL. Isto porque à entrada de um procedimento tem-se acesso a todos os objetos que no momento estejam na tabela de símbolos. A partir daí, IAL é incrementada pelos objetos declarados localmente ao procedimento em questão. Já no caso de módulos, IAL é inicializada com os objetos primitivos e incrementada através de importações e declarações locais. As importações são resolvidas consultando-se EAL. No caso de módulo de implementação, objetos conhecidos pelo módulo de definição correspondente são também acrescidos a IAL.

Rotinas que compilam declarações, comandos e expressões têm um parâmetro formal LAL ("local access list") que recebe como parâmetro efetivo IAL do módulo ou procedimento a que pertencem, ou ainda outra LAL, no caso de serem sub-declarações, sub-comandos ou sub-expressões. A finalidade de LAL em comandos e expressões é exclusivamente de consulta. Já em declarações, LAL é consultada e modificada.

Uma notável exceção ao que foi dito sobre comandos é o caso de with. Pelas características desse comando, um novo escopo de identificadores é aberto, acrescentando-se aos objetos até então conhecidos, os identificadores dos campos do registro referenciado pelo with. O que fizemos na compilação do comando with foi manter a dupla EAL/IAL. À entrada da compilação de with, incluímos na variável local IAL, os objetos de EAL e ainda os identificadores de campos mencionados. Quando WITH_STAT chama STAT_SEQ, passa-lhe IAL.

Existem ainda outras listas de objetos. Subprogramas têm SUBPR_DECL_LIST na forma de um campo do nó correspondente, na tabela de símbolos. Analogamente, nós de módulos têm o campo MOD_DECL_LIST. Tais listas relacionam os objetos locais em cada caso. Nós da tabela de símbolos correspondentes a módulos têm

os campos EXPORT_LIST e IMPORT_LIST, relatando, respectivamente, objetos exportados de modo qualificado e objetos importados pelo módulo. (Objetos exportados de maneira não qualificada, como vimos, são introduzidos diretamente no parâmetro EAL do módulo).

4.4.3. VERIFICAÇÃO DE COMPATIBILIDADE

Compatibilidade é um conceito que se aplica às várias situações em que objetos são confrontados. Tais situações podem ser agrupadas em 3 classes:

(i) Compatibilidade em operações.
(ii) Compatibilidade em atribuição.
(iii) Compatibilidade em passagem de parâmetro.

Na descrição da linguagem [1], (i) aparece simplesmente com o nome "compatibilidade" ("compatibility") e (ii) como "compatibilidade para atribuição" ("assignment compatibility"); (iii) não possui ali nome especial.

No Apêndice A constam as rotinas STRONG_COMPAT, WEAK_COMPAT e PAR_COMPAT, respectivamente relacionadas com (i), (ii) e (iii) acima.

4.4.3.1. COMPATIBILIDADE EM OPERAÇÕES

Numa operação, estamos particularmente interessados no tipo de seus operandos. Para conveniência de exposição, definiremos o conceito de equivalência de tipos, através de uma relação de equivalência "~", dada por:

(i) tipos sinônimos são equivalentes. (i.e., dada uma declaração "type t1 = t2", tem-se t1 ~ t2).

(ii) t1 ~ t2 se t1 e t2 são intervalos de tipos base equivalentes, com mesmos limites.

Definimos agora compatibilidade de tipos como uma relação reflexiva, simétrica, mas não transitiva, dada por: Dois tipos t1 e t2 são compatíveis se alguma das condições abaixo for satisfeita:

(i) t1 ~ t2.

(ii) t1 e t2 são sub-intervalos de um tipo t3.

(iii) t1 é endereço longo (curto) e t2 é apontador longo (curto) para algum tipo t3.

(iv) t1 é endereço curto e t2 é CARDINAL.

(v) - t1 e t2 são apontadores longos (curtos) para tipos compatíveis.

(vi) t1 e t2 são conjuntos com tipos base equivalentes.

(vii) t1 e t2 são vetores dinâmicos de elementos compatíveis.

(viii) t1 e t2 são vetores estáticos de índices equivalentes e elementos compatíveis.

(ix) t1 e t2 são subprogramas de mesma categoria (procedimento ou função), parâmetros concordando em número, de tipos respectivamente compatíveis, tendo respectivamente os mesmos mecanismos de passagem e tipos equivalentes de resultado, se função.

4.4.3.1.1. CONCEITUAÇÃO GERAL

Uma dada operação define os tipos de seus operandos. Por "operação", em MODULA-2 entende-se uma função de um ou dois argumentos dada por um particular símbolo léxico, tal como o

sinal "+", etc. Poderíamos chamar a tais funções operações explícitas, já que sua semântica está vinculada a uma construção sintática especial. (Note que em MODULA-2 não se pode redefinir o sentido dos operadores dados por símbolos léxicos.) Por operações implícitas entenderíamos, por outro lado, as funções definidas pelo programador.

Por motivos de ordem prática, usaremos o termo "operação" apenas no primeiro dos sentidos mencionados. Problemas relacionados ao segundo serão tratados na compatibilidade em passagem de parâmetros.

Usaremos a nomenclatura "tipos escalares" para tipos que são enumeração, intervalos, CHAR e tipos numéricos (INTEGER, SHORT_INTEGER, CARDINAL, SHORT_CARDINAL, seus intervalos e sinônimos). "Tipo base" de um conjunto ou enumeração designará o tipo de seus elementos. Dada uma declaração "type t1 = t2", o tipo raiz de t1 será o tipo raiz de t2. Se t1 não tiver sido declarado como sinônimo de outro tipo, t1 será seu próprio tipo raiz.

4.4.3.1.2. CASOS ESPECÍFICOS

(i) As operações binárias '=', '≠' admitem como operandos expressões escalares, conjuntos e apontadores, desde que os tipos dos dois objetos sejam compatíveis. O valor resultante é de tipo BOOLEAN.

(ii) As operações binárias '<', '<=', '>', '>=' admitem como operandos, objetos escalares de tipos compatíveis ou ainda conjuntos com tipos base equivalentes. O valor resultante é do tipo BOOLEAN.

(iii) As operações binárias and, or, e a unária not, são aplicáveis a objetos booleanos (i.e. compatíveis com BOOLEAN).

(iv) A operação unária "+" se aplica a objetos numéricos e é a função identidade.

(v) A operação unária "-" se aplica a objetos de tipo INTEGER, SHORT_INTEGER, seus intervalos e sinônimos.

(vi) As operações binárias div e mod aplicam-se a objetos numéricos de tipos compatíveis.

(vii) As operações binárias "+", "-", "*", "/" aplicam-se a argumentos numéricos de tipos compatíveis ou a conjuntos com tipos base equivalentes. Em qualquer caso, o resultado devolvido é do tipo raiz dos argumentos. Quando aplicadas a conjuntos, o significado de tais operações é, respectivamente, união, diferença, intersecção e diferença simétrica.

Na discussão de compatibilidade em operações até aqui efetuada, supusemos implicitamente que toda expressão tem um tipo. Há um problema quanto a isso no caso de constantes explícitas, tais como NIL, 15, etc. Tais expressões podem a priori pertencer a mais de um tipo (15, por ex. pode pertencer a qualquer dos quatro tipos numéricos básicos !). A decisão sobre o tipo de tais expressões faz-se pelo contexto em que elas ocorrem em operações, chamadas de subprogramas e comandos de atribuição. Isso, porém não cobre todos os casos, como se vê pelo seguinte exemplo:

"const m = 15".

Para resolver tais situações, o compilador provê alguns tipos especiais:

- UNIV_INT ("universal integer"): atribuído a toda expressão numérica de contexto não diferenciado.

- UNIV_PTR ("universal pointer"): atribuído ao valor NIL.

O usuário pode, porém fixar a priori qual tipo deseja atribuir a tais constantes, por meio das funções de transferência de tipos que são facilidades do sistema, previstas

pela linguagem. Exemplos de uso, seriam:

```
const m = SHORT_CARDINAL (15);

type intervalo = [CARDINAL (10) .. CARDINAL (20)].
```

4.4.3.2. COMPATIBILIDADE EM ATRIBUIÇÃO

Definiremos compatibilidade para atribuição como uma relação "~~" entre um tipo e uma expressão.

Dados um tipo t e uma expressão E, diremos que "t ~~ E" se pelo menos uma das condições abaixo for verificada:

(i) t é compatível com o tipo de E.

(ii) t é vetor dinâmico, E é vetor estático e os elementos de t e E são compatíveis.

(iii) t e o tipo de E são tipos numéricos de mesmo tamanho.

(iv) t e o tipo de E são cadeias de caracteres, sendo o tamanho de t não menor que o do tipo de E.

Restrição: No caso de E ser expressão constante e t tipo intervalo, o valor de E deverá estar compreendido entre os extremos permitidos a t.

Como o nome indica, a relação em estudo é considerada no tratamento de comandos de atribuição, caso em que deverá ser verificado se a expressão do lado esquerdo de ":=" pode receber valor. É ainda usada a citada relação, na análise de passagem de parâmetro por valor.

4.4.3.3. COMPATIBILIDADE EM PASSAGEM DE PARÂMETROS

(i) Um parâmetro formal tipo WORD (BYTE) admite como parâmetro efetivo qualquer expressão com tipo de igual tamanho.

(ii) Um parâmetro por referência tipo "array of WORD" ou "array of BYTE" admite como parâmetro efetivo, qualquer expressão.

(iii) Um parâmetro por referência que é vetor dinâmico (em que o tipo dos elementos não é WORD nem BYTE) admite como parâmetro efetivo, qualquer vetor com elementos de tipo equivalente ao dos seus.

Se as regras acima não são aplicáveis, tem-se:

(i) Se o mecanismo de passagem é por referência, os tipos dos parâmetros formal e efetivo devem ser compatíveis e o parâmetro efetivo deve poder receber valor.

(ii) Se o mecanismo de passagem é por valor, o tipo do parâmetro formal deve ser compatível para atribuição em relação ao parâmetro efetivo correspondente.

4.4.4. RECUPERAÇÃO DE ERROS DE CONTEXTO

Chamamos "erros de contexto" àqueles relacionados a descrições de objetos nas diversas listas, tais como tabela de símbolos, listas de acessibilidade, listas de declarações, listas de parâmetros, etc.

Podemos num primeiro plano diferenciar entre erros detetados:

(i) na inserção de um objeto e
(ii) na busca de um objeto.

A ocorrência do primeiro tipo de erro se dá quando já há objeto homônimo no escopo. O segundo tipo de erro pode ainda ser dividido em duas categorias, conforme a finalidade da busca:

(ii.1) A finalidade da busca era usar um objeto com especificação completa.

(ii.2) A finalidade da busca era completar a especificação de um objeto.

O caso ii.1 ocorre quando o objeto procurado não é encontrado, ou ainda, quando é encontrado mas fora das especificações desejadas. O caso ii.2 ocorre quando o objeto encontrado já tem especificação completa ou ainda, quando é encontrado objeto incompleto mas incompatível com as características "novas". Em qualquer caso, notifica-se o erro e realiza-se alguma ação corretora, dependente do tipo de erro. Uma das ações que se toma, consiste em "indefinir-se" certos objetos, i.e., marca-se o objeto como já havendo sido erroneamente referenciado, de modo que novos usos do objeto no escopo não redundem em desnecessárias repetições de mensagens de erro.

4.4.4.1. OBJETOS INDEFINIDOS

Duas perguntas são ineludíveis consequências da decisão de se indefinir objetos:

- O que fazer quando mais tarde o objeto indefinido for encontrado ?
- Se um objeto indefinido faz parte de outro, este também deverá ser indefinido ?

A primeira das perguntas é mais fácil de se responder: encontrado um objeto indefinido, agimos como se ele tivesse as propriedades desejadas. A resposta à segunda pergunta depende da espécie do objeto mais global. Não seria prudente indefinir sem necessidade tal objeto, pois isso poderia encobrir erros de referências posteriores a tal objeto. Suponhamos que um objeto indefinido A seja integrante de B. Se, pelas características de B houver possibilidade de o compilador voltar a selecionar o objeto A, então indefinimos B. Caso contrário, não o fazemos. Por exemplo, se houve erro na especificação do índice de um vetor, indefinimos o próprio vetor para que mais tarde não tenhamos problemas ou pelo menos ineficiência ao se verificar a compatibilidade de tal índice (digamos, na compilação de uma expressão envolvendo o vetor). Outro caso: um dos comandos componentes de um comando while foi indefinido. Não há necessidade de se indefinir todo o while, uma vez que, compilado um comando, o compilador (no primeiro passo) não mais entrará no mérito de suas partes integrantes.

Damos a seguir um quadro sinótico da recuperação dos erros de contexto.

Situações de deteção de erros de contexto	Na inserção de um objeto		ERRO: Existência prévia de homônimo no escopo. AÇÃO CORRETIVA: Indefine-se o objeto encontrado e não se insere o novo objeto.
	Na busca de um objeto		ERRO1: Objeto não encontrado. AÇÃO CORRETIVA: Insere-se, no escopo, um objeto fictício, com o nome procurado e campo cat = symt_undef.
		Era desejado objeto completo	ERRO2: Objeto encontrado, mas fora das especificações esperadas. AÇÃO CORRETIVA: Indefine-se o objeto, se encontrado no escopo atual. Caso contrário, cria-se um objeto fictício, com o nome procurado, e campo cat = symt_undef.
		Era desejado objeto incompleto	ERRO: O objeto encontrado é completo ou tem características diferentes das esperadas. AÇÃO CORRETIVA: Indefine-se o objeto encontrado.

4.4.4.2. OBJETOS DESCRITOS EM DUAS ETAPAS

Há em MODULA-2 várias situações em que objetos têm uma especificação parcial e posteriormente uma especificação complementar. São os casos seguintes:

(i) Um tipo pode aparecer como opaco num módulo de definição e com descrição completa no módulo de implementação correspondente.

(ii) Um subprograma pode ser mencionado num módulo de definição através de seu cabeçalho e ter declaração completa no módulo de definição correspondente.

(iii) Um módulo de definição é complementado pelo módulo de implementação correspondente.

(iv) O nome de um tipo apontado pode aparecer antes de sua especificação.

Nossa implementação acrescenta ainda o seguinte caso:

(v) Um subprograma pode ser especificado pelo seu cabeçalho precedido da símbolo forward, sendo que sua descrição completa aparece mais tarde, no mesmo escopo. Isso pode ocorrer no interior de qualquer bloco.

O caso (iii) foi visto quando tratávamos das diversas categorias de módulos; em todos os outros casos, as duas partes da descrição devem pertencer a um mesmo escopo.

Para diferenciar objetos com especificação completa dos demais, há nos nós da tabela de símbolos, um campo INCOMPL, de valor booleano. Na primeira parte da descrição, INCOMPL fica valendo TRUE e passa a FALSE na segunda parte.

Quando se compila comandos, as declarações do bloco correspondente já devem estar todas completas. Para que isso

seja verificado, a rotina BLOCK percorre, ao fim das declarações e antes de chamar STAT_SEQ, a lista de declarações dos objetos locais. Se algum objeto é então encontrado com INCOMPL = TRUE, tem-se situação de erro.

4.4.4.3. OBJETOS SEMI-OPACOS

Vimos que tipos opacos são complementados num módulo de implementação, sendo aí conhecidos de modo completo; i.e., eles não são realmente opacos dentro do módulo de implementação. O caso é que dentro de um módulo de implementação, partes integrantes de um objeto opaco (que lhe pertença), podem ser referenciadas.

Para isso criou-se a categoria de tipo semi-opaco (ST_TY_S_OPAQUE). Quando numa declaração de tipo encontra-se a complementação de um tipo opaco, muda-se sua categoria para semi-opaco. Ao fim do módulo de implementação, porém, o tipo retorna à sua categoria de opaco.

4.5. MISCELÂNEA

4.5.1. RETURN'S e EXIT'S

A compilação de um comando return necessita de um apontador para o módulo ou a rotina a que o return se refere. Tal apontador constituirá o campo RETURN_UNIT do STAT_NODE correspondente. Analogamente, um comando exit refere-se a um loop, que será apontado pelo FROM_LOOP do seu STAT_NODE. Para esse fim, muitas rotinas terão os parâmetros:

- RET_UNIT : PTR_SYMT_NODE e
- FR_LOOP : PTR_STAT_NODE.

As rotinas que compilam módulos (exceto DEF_MOD) e subprogramas, ao chamarem BLOCK, passam-lhe RET_UNIT apontando para si mesmas e FR_LOOP com o valor NIL. BLOCK, ao chamar STAT_SEQ e esta ao chamar STATEMENT, passam os dois parâmetros

inalterados. STATEMENT passa os dois parâmetros inalterados nas chamadas de rotinas que compilam if, while, repeat, loop, for, case e with. RETURN_STAT não tem FR_LOOP. EXIT_STAT não tem FR_LOOP. ASSIGN_STAT e PCALL não têm nenhum dos dois parâmetros. As rotinas que compilam if, while, repeat, for e case chamam STAT_SEQ com os dois parâmetros inalterados. LOOP_STAT, ao chamar STAT_SEQ, transmite-lhe RET_UNIT inalterado, mas FR_LOOP é passado apontando para o nó correspondente ao comando sendo criado.

4.5.2. COMANDO "WITH" E SELEÇÃO IMPLÍCITA

Posto que é possível haver with's encaixados e que as expressões seletoras, sendo construídas uma vez podem ser usadas muitas vezes, adotamos a solução de guardar uma pilha de expressões seletoras, através de uma variável global:

- WITH_STACK : EXPR_STACK.

O comando with, então, ao construir uma expressão seletora, a empilha em WITH_STACK e faz o campo WITH_SELEC apontar para ela. Ao fim da compilação do with a expressão é desempilhada. (Relembre o que foi dito sobre with em 4.4.2, para completar a idéia). Quando, na compilação de uma expressão, encontrar-se um nó SYMT_SELEC, ter-se-á que localizar o registro correspondente, a fim de se preencher o campo WITH_DESIGN da expressão em questão. (O campo restante WITH_SELECTOR é o nó já encontrado). Para tal, faz-se o seguinte:

(i) através do nó SYMT_SELEC localiza-se o tipo t do registro requerido (isso é conseguido mediante o campo PARENT_TYPE).

(ii) procura-se em WITH_STACK ocorrência de uma expressão de tipo t.

(iii) o campo WITH_DESIGN é então preenchido com a expressão assim encontrada.

Observe que com esse esquema fica solucionado inclusive o caso em que dois ou mais with's encaixados têm mesmo tipo de registro associado.

4.5.3. O PROBLEMA DO "FORWARD IMPLÍCITO" EM TIPO APONTADOR

Já vimos que o nome de um tipo apontado pode ocorrer antes da declaração correspondente.

Quando isso acontece, cria-se provisoriamente um tipo (incompleto) com o nome referenciado e categoria ST_TY_FORW. A especificação completa de tal tipo ocorrerá mais tarde no mesmo escopo (salvo erro). O novo nó deverá então substituir o antigo. Há em princípio duas maneiras de se conseguir isso:

(i) modificando as referências ao nó antigo, de modo que passem a apontar para o novo nó, ou:

(ii) copiar os campos do novo nó para o velho.

A primeira das soluções é difícil de se implementar porque haveria necessidade de saber todas as referências até então feitas ao primeiro nó. A segunda solução é a que adotamos. Necessita ela, porém uma correção: se simplesmente copiarmos os campos, teremos problemas no caso de o novo tipo ser um registro ou enumeração. Isto porque, do ponto de vista de tabela de símbolos, tais objetos se auto-referenciam. O que se faz para salvar a situação é percorrer, no caso em questão os componentes de tais objetos, alterando as "retro-referências", de modo a apontarem para o nó antigo.

4.5.4. REPRESENTAÇÃO DE CADEIAS DE CARACTERES

MODULA-2 especifica que uma constante explícita constituída por apenas um caráter (ex. 'x') é do tipo CHAR; cadeias de caracteres explícitas com tamanho maior que um são de tipo STRING. Em nossa implementação representamos as cadeias de caracteres explícitas de tamanho dois diferentemente de cadeias maiores. Uma razão é que cadeias de tamanho dois cabem numa palavra de memória do computador objeto. Assim, as guardamos como constantes com tipo "array [0..1] of CHAR" e com valor que é a própria representação de caracteres na palavra do computador objeto.

Cadeias explícitas maiores, representamos como pseudo variáveis. Assim, a expressão correspondente tem categoria VAR_EXPR e o campo ASSIGNABLE recebe o valor FALSE, indicando a condição de constante. O campo VAR_DESCR, ainda na mesma expressão, aponta para um nó anônimo da tabela de símbolos, de categoria SYMT_VAR e VAR_CAT = CONST_VAR (i.e., "variável constante"). Em tal nó, o campo VAR_VAL ("valor da variável") aponta para a representação da cadeia.

Razões de conveniência aos eventuais usuários do compilador, levaram-nos a relaxar as regras de compatibilidade em passagem de parâmetro, no caso de cadeias de caracteres: uma cadeia constante pode ser passada por referência (!). O compilador limita-se a dar uma advertência quando isso ocorre.

4.5.5. TAMANHOS DOS TIPOS

O número de "bytes" necessários à representação de um objeto no computador para o qual é gerado código é o que chamamos "tamanho do tipo" deste objeto. O tamanho do tipo é armazenado no campo TYPE_SIZE do nó da tabela de símbolos que descreve o tipo.

Existem:

(i) tipos de tamanho fixo.

(ii) tipos de tamanho variável (mas conhecido em tempo de compilação.

(iii) tipos sem tamanho estabelecido.

Entre os primeiros estão as enumerações (tamanho um), intervalos (tamanho um ou dois), conjuntos (tamanho um ou dois), apontadores longos (tamanho quatro) e curtos (tamanho dois) e subprogramas (tamanho quatro).

Entre os do tipo (ii) estão:

- vetores estáticos:
  tamanho = tamanho da dimensão * tamanho do tipo dos elementos.

-registros:
  tamanho = soma dos tamanhos das componentes;

-componente variável de registro:
  tamanho = máximo dos tamanhos das variantes.

Finalmente, (iii) corresponde aos vetores dinâmicos.

5. CONCLUSÃO

O princípio fundamental a que me ative durante o desenvolvimento do programa foi o de segurança (em detrimento inclusive da eficiência). Assim, o uso de tratamentos similares a problemas similares foi extensivamente adotado.

O compilador tem aproximadamente vinte mil linhas (quinze mil das quais referentes ao primeiro passo); compila cerca de 115 linhas por segundo, numa mesma máquina em que o compilador PASCAL [10] compila 260 linhas por segundo. Cerca de um terço do tempo é consumido pelo analisador léxico e um terço pelo segundo passo.

Em termos de espaço o compilador gasta 53 Kw (≅ 238 Kby) de código, sendo dois terços relativos ao primeiro passo. A área de dados fixa é de 12 Kw (≅ 54 Kby), da qual quatro quintos relativos ao primeiro passo. A área variável cresce cerca de 47 w (≅ 233 by) por linha compilada. O elevado gasto se deve em parte ao fato de a árvore de programa ficar toda em memória.

Conforme citado na Introdução, o trabalho aqui descrito foi realizado dentro de um convênio entre o Departamento de Ciência da Computação do IMECC - UNICAMP e a TELEBRÁS. Após o encerramento do trabalho, o compilador continuou a ser modificado, nas dependências do CPqD-TELEBRÁS. Implementou-se então compilação separada, e testou-se o compilador com 26.000 linhas fonte, de módulos de programas escritos na Universidade de Zurique. Tive oportunidade de participar de tais trabalhos.

APÊNDICE A: Algumas rotinas do compilador.

```
   PROCEDURE PROGR;
(                                                  (REM:                SYMB_SET;
                                                   VAR P:               PTR_SYMT_NODE);
)

      VAR
         GAL:          SYMT_STACK;
         P1, P2:       PTR_SYMT_ITEM;

   BEGIN

      PRIMIT_INIT (GAL);
      NEW(P);  WITH P^ DO
         BEGIN
         ID_NAME :=              NULL_NAME;
         ID_CAT :=               USER_ID;
         LEX_LEVEL :=            LEXICAL_LEVEL;
         MOD_LEVEL :=            MODULAR_LEVEL;
         USED :=                 FORCE_FLAG;
         INCOMPL :=              FALSE;
         FROM_MODULE :=          NIL;
         CAT :=                  SYMT_PROGR;
         NULLQ_SYMT              (EXTERN_MOD_LIST);

         WHILE SYMB IN [S_DEFINITION, S_IMPLEMENT] DO
              IF SYMB = S_DEFINITION
              THEN
                 BEGIN
                 SCAN;
                 DEF_MODULE (REM + [S_DEFINITION, S_IMPLEMENT, S_MODULE],
                             GAL, EXTERN_MOD_LIST)
                 END
              ELSE
                 BEGIN
                 SCAN;
                 IMPL_MODULE (REM + [S_DEFINITION, S_IMPLEMENT, S_MODULE],
                              GAL, EXTERN_MOD_LIST)
                 END;

         IF SYMB <> S_MODULE
         THEN
              INTERN_MOD := NIL
         ELSE
              BEGIN
              SCAN;
              PROGR_MODULE (REM, GAL, INTERN_MOD)
              END
         END;

         P1:= PRIMIT_OBJ.TOP;  P2:= PRIMIT_OBJ.BOTTOM;
         WHILE P1 <> P2 DO
              BEGIN
              P1^.ITEM_DESCR^.BELONGS_TO := P;
              P1 := P1^.NEXT
              END;

         PUT_ERROR

   END; (* PROGR *)
```

```
   PROCEDURE BLOCK;
(                                                  (REM:                SYMB_SET;
                                                   VAR LAL:             SYMT_STACK;
                                                   VAR DECL_LIST:       SYMT_QUEUE;
                                                   VAR STAT_LIST:       STAT_QUEUE;
                                                   RET_UNIT:            PTR_SYMT_NODE;
                                                   FR_LOOP:             PTR_STAT_NODE);
)

      VAR
         OLD_DECL_LIST:       SYMT_QUEUE;

   BEGIN

      WHILE SYMB IN FIRST_DECL DO
         DECL (REM + [S_BEGIN] + FIRST_STAT + [S_END], LAL, DECL_LIST);

      CHECK_INCOMPL (DECL_LIST);

      OLD_DECL_LIST :=        GLB_DECL_LIST;
      GLB_DECL_LIST :=        DECL_LIST;

      IF SYMB = S_BEGIN
         THEN BEGIN
              SCAN;
              STAT_SEQ (REM + [S_END],
                        LAL, STAT_LIST, RET_UNIT, FR_LOOP)
              END;

      GLB_DECL_LIST :=        OLD_DECL_LIST;

      TEST (25, [S_END], REM);  IF SYMB = S_END THEN SCAN;

   END; (* BLOCK *)
```

```
   PROCEDURE CONST_DECL;                                    (REM:              SYMB_SET;
{                                                           VAR LAL:           SYMT_STACK;
                                                            VAR DECL_LIST:     SYMT_QUEUE);

}

      VAR
         OBJ_NAME:     NAME;
         P, C:         PTR_SYMT_NODE;
         AUX_EXPR:     PTR_EXPR_NODE;
         BUG:          BOOL;

   BEGIN

      SCAN;  TEST (26, [S_IDENT], REM);
      WHILE SYMB = S_IDENT DO
         BEGIN
         OBJ_NAME:=ID_VAL;  SCAN;  BUG:=FALSE;

         SSEARCH_SYMT (LAL, OBJ_NAME, P);
         IF P <> NIL
         THEN IF P^.LEX_LEVEL = LEXICAL_LEVEL
                 THEN BEGIN
                      CONTEXT_ERROR (24);  BUG:=TRUE;  P^.CAT:=SYMT_UNDEF
                      END;

         TEST (27, [S_EQUAL], REM + [S_COLON, S_BECOMES] + FIRST_EXPR + [S_SEMICOLON]);
         IF SYMB IN [S_EQUAL, S_COLON, S_BECOMES]
         THEN SCAN;

         EXPR (REM + [S_SEMICOLON, S_IDENT], LAL, AUX_EXPR);

         IF AUX_EXPR^.CAT = UNDEF_EXPR
         THEN
              BUG := TRUE
         ELSE
              IF NOT CONSTANT_EXPR (AUX_EXPR)
              THEN
                BEGIN
                CONTEXT_ERROR (25);  BUG:= TRUE
                END;
```

```
            IF BUG
            THEN
                 UND_SYMT (C)
            ELSE
                 IF AUX_EXPR^.CAT = VAR_EXPR
                 THEN
                    BEGIN
                    C :=                    AUX_EXPR^.VAR_DESCR;
                    C^.ID_NAME :=           OBJ_NAME
                    END
                 ELSE
                    BEGIN
                    NEW (C);  WITH C^ DO
                         BEGIN
                         ID_NAME :=              OBJ_NAME;
                         ID_CAT :=               USER_ID;
                         ID_PS_NAME :=           PD_NULL_NAME;
                         LEX_LEVEL :=            LEXICAL_LEVEL;
                         MOD_LEVEL :=            MODULAR_LEVEL;
                         USED :=                 FORCE_FLAG;
                         INCOMPL :=              FALSE;
                         CAT :=                  SYMT_CONST;
                         CONST_DESCR :=          AUX_EXPR
                         END
                 END;

            PUSH_SYMT (LAL, C);  ENQ_SYMT (DECL_LIST, C);

            TEST (28, [S_SEMICOLON], REM+[S_IDENT]);
            IF SYMB = S_SEMICOLON THEN SCAN
            END

       END; (* CONST_DECL *)
```

```
   PROCEDURE IF_STAT;
(                                                    (REM:                SYMB_SET;
                                                     LAL:                 SYMT_STACK;
                                                     VAR STAT:            PTR_STAT_NODE;
                                                     RET_UNIT:            PTR_SYMT_NODE;
                                                     FR_LOOP:             PTR_STAT_NODE);
)

      VAR
         AUX_EXPR:            PTR_EXPR_NODE;
         AUX_STAT_LIST:       STAT_QUEUE;
         AUX_ALT:             PTR_ALTERN;
         BUG:                 BOOL;

   BEGIN

      BUG := FALSE;  SCAN;
      NEW(STAT);  WITH STAT^ DO
         BEGIN
         CAT :=               IF_ST;
         NULLQ_ALTERN         (ALTERN_LIST);
         ELSE_S_LIST :=       NIL
         END;

      LOOP
         EXPR (REM, LAL, AUX_EXPR);
         IF NOT COMPAT (AUX_EXPR^.TY, BOOL_TY)
         THEN BUG:= TRUE;

         TEST (89, [S_THEN], REM + FIRST_STAT + [S_ELSIF]);
         IF SYMB = S_THEN THEN SCAN;

         STAT_SEQ (REM, LAL, AUX_STAT_LIST, RET_UNIT, FR_LOOP);

         NEW(AUX_ALT);  WITH AUX_ALT^ DO
              BEGIN
              CONDITION :=    AUX_EXPR;
              STAT_LIST :=    AUX_STAT_LIST
              END;

         IF NOT BUG THEN ENQ_ALTERN (STAT^.ALTERN_LIST, AUX_ALT)

      EXIT IF SYMB <> S_ELSIF;

         SCAN

      END; % LOOP\

      IF SYMB = S_ELSE
         THEN BEGIN
              SCAN;  STAT_SEQ (REM, LAL, STAT^.ELSE_S_LIST, RET_UNIT, FR_LOOP)
              END;

      TEST (90, [S_END], REM);  IF SYMB = S_END THEN SCAN;
      IF BUG THEN UNB_STAT(STAT)

   END; (* IF_STATEMENT *)
```

```
PROCEDURE TERM;
(                                                   (REM:                   SYMB_SET;
                                                    LAL:                    SYMT_STACK;
                                                    VAR T:                  PTR_EXPR_NODE);
)

      VAR
         AUX_SYMB:              SYMBOL;
         AUX_SYMB_SET:          SYMB_SET;
         F1, F2:                PTR_EXPR_NODE;
         T1, T2:                PTR_SYMT_NODE;
         BUG:                   BOOL;

BEGIN

      BUG:= FALSE;
      AUX_SYMB_SET := [S_TIMES, S_DIV, S_MOD, S_SLASH, S_AND];
      FACTOR (REM + AUX_SYMB_SET, LAL, T);
      IF T^.CAT = UNDEF_EXPR THEN BUG:= TRUE;

      WHILE SYMB IN AUX_SYMB_SET DO
         BEGIN
         AUX_SYMB:=SYMB;  SCAN;  F1:=T;  FACTOR (REM, LAL, F2);
         T1:= AUX_BASE_TY (F1^.TY);  T2:= AUX_BASE_TY (F2^.TY);
         IF F2^.CAT = UNDEF_EXPR
         THEN BUG:= TRUE;
         IF NOT BUG
         THEN
              BEGIN
              CASE AUX_SYMB OF

                 S_AND:                 BUG := NOT (BOOL_TYPE(T1) AND BOOL_TYPE(T2));

                 S_DIV, S_MOD:          BUG := NOT (NUM_TYPE (T1) AND NUM_TYPE(T2));

                 S_TIMES:               BUG := NOT (NUM_TYPE(T1) AND NUM_TYPE(T2) OR
                                                    SET_TYPE(T1) AND SET_TYPE(T2));

                 S_SLASH:               BUG := NOT (SET_TYPE(T1) AND SET_TYPE(T2))
              END; % CASE \

              IF BUG THEN CONTEXT_ERROR (85)
              END;

         IF BUG
         THEN UND_EXPR (T)
         ELSE
              MAKE_BIN_EXPR (AUX_SYMB, T, F1, F2)
         END

END; (* TERM *)
```

```
FUNCTION STRONG_COMPAT;
(                                       (T1, T2:            PTR_SYMT_NODE): BOOL;
)

    VAR
       BUG: BOOL;
BEGIN

    BUG:= FALSE;  T1:= AUX_BASE_TY (T1);  T2:= AUX_BASE_TY (T2);
    IF T1 <> T2
    THEN
       IF (T1^.CAT <> SYMT_UNDEF) AND (T2^.CAT <> SYMT_UNDEF)
       THEN
            IF NOT (NUM_TYPE(T1) AND (T2=UNIV_INT_TY) OR
                    NUM_TYPE(T2) AND (T1=UNIV_INT_TY))
            THEN
               IF NOT (PTR_TYPE(T1) AND (T2 = UNIV_PTR_TY) OR
                       PTR_TYPE(T2) AND (T1 = UNIV_PTR_TY))
               THEN
                       IF NOT ((T1=ADDR_TY) AND (T2^.TYPE_CAT = ST_TY_POINTER) OR
                               (T2=ADDR_TY) AND (T1^.TYPE_CAT = ST_TY_POINTER))
                       THEN
                            IF NOT ((T1=LG_ADDR_TY) AND (T2^.TYPE_CAT = ST_TY_LG_PTR) OR
                                    (T2=LG_ADDR_TY) AND (T1^.TYPE_CAT = ST_TY_LG_PTR))
                            THEN
                               IF NOT ((T1=ADDR_TY) AND (T2=CARD_TY) OR
                                       (T2=ADDR_TY) AND (T1=CARD_TY))
                               THEN
                                    IF NOT ((T1^.TYPE_CAT = T2^.TYPE_CAT) OR
                                            (T1^.TYPE_CAT = ST_TY_DYN_ARR) AND
                                            (T2^.TYPE_CAT = ST_TY_FXD_ARR))
                                    THEN BUG:= TRUE
                                    ELSE
                                            CASE T1^.TYPE_CAT OF

                                            ST_TY_POINTER, ST_TY_LG_PTR:
                                                          BUG:= NOT STRONG_COMPAT (T1^.POINTED_TY, T2^.POINTED_TY);

                                            ST_TY_SET:    BUG:= NOT INDEX_COMPAT (T1^.BASE_TY, T2^.BASE_TY);

                                            ST_TY_FXD_ARR: IF NOT INDEX_COMPAT (T1^.BOUNDS_TY, T2^.BOUNDS_TY)
                                                          THEN BUG:= TRUE
                                                          ELSE BUG:= NOT STRONG_COMPAT (T1^.FXD_ELEM_TY, T2^.FXD_ELEM_TY);

                                            ST_TY_DYN_ARR: IF T2^.TYPE_CAT = ST_TY_FXD_ARR
                                                          THEN BUG:= NOT STRONG_COMPAT (T1^.DYN_ELEM_TY, T2^.FXD_ELEM_TY)
                                                          ELSE BUG:= NOT STRONG_COMPAT (T1^.DYN_ELEM_TY, T2^.DYN_ELEM_TY);

                                            ST_TY_SUBPR:  BUG:= NOT SUBPR_COMPAT (T1, T2);

                                            OTHERS:       BUG:= TRUE

                                            END; % CASE \

    STRONG_COMPAT := NOT BUG

END; (* STRONG_COMPAT *)
```

```
FUNCTION WEAK_COMPAT;
{                                                (T:                    PTR_SYMT_NODE;
                                                 E:                     PTR_EXPR_NODE):        BOOL;
}

     VAR
        BUG:          BOOL;
        T1, T2:       PTR_SYMT_NODE;

BEGIN

     T1:= AUX_BASE_TY (T);  T2:= AUX_BASE_TY (E^.TY);  BUG:= FALSE;

     IF (T1^.CAT <> SYMT_UNDEF) AND (T2^.CAT <> SYMT_UNDEF)
     THEN
        IF (T1^.TYPE_CAT = ST_TY_DYN_ARR) AND (T2^.TYPE_CAT = ST_TY_FXD_ARR)
        THEN BUG:= NOT STRONG_COMPAT (T1^.DYN_ELEM_TY, T2^.FXD_ELEM_TY)
        ELSE
             IF (T1^.TYPE_CAT = ST_TY_DYN_ARR)
             THEN BUG:= TRUE
             ELSE
             IF NOT STRONG_COMPAT (T1, T2)
                     THEN
                     IF NOT (NUM_TYPE(T1) AND NUM_TYPE(T2) AND (T1^.TYPE_SIZE = T2^.TYPE_SIZE))
                     THEN
                        IF NOT (STRING_TYPE(T1) AND STRING_TYPE(T2))
                        THEN BUG:= TRUE
                        ELSE BUG:= T1^.TYPE_SIZE < T2^.TYPE_SIZE;

     IF NOT BUG
     THEN
        IF SCAL_TYPE(T1) AND CONSTANT_EXPR(E)
        THEN
             BUG :=  (E^.VALUE < MIN_VAL(T)) OR
                     (E^.VALUE > MAX_VAL(T));

     WEAK_COMPAT := NOT BUG

END; (* WEAK_COMPAT *)
```

```
FUNCTION PAR_COMPAT;
(                                          (FRPAR:          PTR_PAR_NODE;
                                           ACPAR:           PTR_EXPR_NODE):     BOOL;
)

     VAR
        FR_TY, AC_TY:        PTR_SYMT_NODE;
        FR_MODE:             VAR_CLASS;
        BUG:                 BOOL;

BEGIN

     BUG := FALSE;
     FR_TY := DISPL_TYNAM (FRPAR^.PAR_TYPE);
     AC_TY := DISPL_TYNAM (ACPAR^.TY);
     FR_MODE := FRPAR^.PAR_MODE;

     IF NOT (((FR_TY = WORD_TY) OR ( FR_TY = BYTE_TY)) AND
             (FR_TY^.TYPE_SIZE = AC_TY^.TYPE_SIZE))
     THEN
        IF NOT (UNIV_TYPE (FR_TY) AND (FR_MODE = REF_PAR))
        THEN
             IF (FR_TY^.TYPE_CAT = ST_TY_DYN_ARR)
             THEN
                IF FR_MODE <> REF_PAR
                THEN
                     BEGIN
                     BUG := TRUE;
                     CONTEXT_ERROR (204)
                     END
                ELSE
                     IF NOT ARR_TYPE (AC_TY)
                     THEN
                        BEGIN
                        CONTEXT_ERROR (205);  BUG := TRUE
                        END
                     ELSE
                        BUG := NOT COMPAT (ELEM_TY (FR_TY), ELEM_TY (AC_TY))
             ELSE
```

```
                IF FR_TY^.TYPE_CAT = ST_TY_SUBPR
                THEN
                     IF ACPAR^.TY^.TYPE_CAT <> ST_TY_SUBPR
                     THEN
                        BEGIN
                        CONTEXT_ERROR (206);  BUG := TRUE
                        END
                     ELSE
                        BEGIN
                        IF ACPAR^.CAT = SUBPR_EXPR
                        THEN
                             IF ACPAR^.SUBPR_DESCR^.LEX_LEVEL <> GLOBAL_LEVEL
                             THEN
                                BEGIN
                                CONTEXT_ERROR (207);  BUG := TRUE
                                END
                             ELSE
                                IF (FR_MODE = REF_PAR) AND (NOT ACPAR^.ASSIGNABLE)
                                THEN
                                     BEGIN
                                     CONTEXT_ERROR (208);  BUG := TRUE
                                     END;

                        IF NOT BUG
                        THEN
                             BUG := NOT SUBPR_COMPAT (FR_TY, AC_TY)
                        END
                ELSE
                     IF FR_MODE = VALUE_PAR
                     THEN
                        BUG := NOT ASS_COMPAT (FR_TY, ACPAR)
                     ELSE
                        IF NOT ACPAR^.ASSIGNABLE
                        THEN
                             BEGIN
                             CONTEXT_ERROR (209);  BUG := TRUE
                             END
                        ELSE
                             BUG := NOT COMPAT (FR_TY, AC_TY);

     PAR_COMPAT := NOT BUG

END; (* PAR_COMPAT *)
```

BIBLIOGRAFIA

[1] N. Wirth:
"Programming in MODULA-2"
Ed. Springer-Verlag (1982).

[2] N. Wirth:
"Modula: a language for modular multiprogramming"
Software - Practice and Experience 7, 3-35 (1977).

[3] N. Wirth:
"A Linguagem de Programação MODULA-2"
Tradução e adaptações de Tomasz Kowaltowski.
Arqvo no DEC-10 (1983).

[4] P. Naur et al.:
"Revised report on the algorithmic language ALGOL-60"
Comm. ACM, 6, No. 1 (Jan 1971), 1-17.

[5] T. Kowaltowski:
"Implementação de linguagens de programação"
Ed. Guanabara Dois, RJ (1983).

[6] Hopcroft, Ullman:
"Formal languages and their relation to automata"
Ed. Addison-Wesley (1969).

[7] N. Wirth:
"Algorithms + data structures = programs"
Ed. Prentice-Hall (1976).

[8] N. Wirth:
"The programming language PASCAL"
Acta Informatica 1, 35-63 (1971).

[9] Heloísa V.R.C. Silva:
"Recuperação de erros em analisadores sintáticos descendentes"
Tese de Mestrado - Depto. de C. Computação, IMECC-UNICAMP (1981).

[10] NAGEL/AMMANN/KISICK/HEDRICK:
Compilador para a linguagem PASCAL / PDP-10.