UM COMPILADOR PARA A LINGUAGEM DE PROGRAMAÇÃO PARALELA ACTUS

Luiz Eduardo Favre

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Claudio Luís Amorim, Ph. D.
(presidente)

Profa. Leila Maria Ripoll Eizirik, D.Sc.

Prof. José Lucas Mourão Rangel Netto, Ph. D.

RIO DE JANEIRO, RJ – BRASIL
MAIO DE 1992

FAVRE, LUIZ EDUARDO

Um Compilador para a Linguagem de Programação Paralela Actus [Rio de Janeiro] 1992

VI, 155 p., 29.7 cm, (COPPE/UFRJ, M. Sc., ENGENHARIA DE SISTEMAS E COMPUTAÇÃO, 1992)

TESE – Universidade Federal do Rio de Janeiro, COPPE

1 – Arquitetura de Computadores 2 – Processamento Paralelo

I. COPPE/UFRJ II. Título(Série).

À memória de meu pai André Louis Favre

Agradecimentos

Aos meus familiares, professores, colegas e amigos. Muitos foram os que em diversas oportunidades, através do estímulo, apoio, orientação e compreensão, me ajudaram na realização deste trabalho. Muito obrigado.

Resumo da Tese apresentada à COPPE como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M. Sc.)

Um Compilador para a Linguagem de Programação Paralela Actus

Luiz Eduardo Favre

Maio de 1992

Orientador: Claudio Luís de Amorim
Programa: Engenharia de Sistemas e Computação

Linguagens de programação para ambientes paralelos devem possuir uma notação adequada para expressar o paralelismo inerente ao problema. Seus compiladores devem gerar código objeto onde o paralelismo expresso no programa resulte em alto desempenho.

Este trabalho apresenta um compilador (front-end) para a linguagem Actus, orientada para processadores vetoriais e matriciais. Descrevemos o tratamento, pelo compilador, dos dados, expressões e comandos, com ênfase para as construções paralelas.

Cada construção apresentada inclui a sintaxe de Actus, as ações semânticas correspondentes, sua representação interna através de sub-árvores, e o código gerado na representação intermediária adotada.

Abstract of Thesis presented to COPPE as partial fulfillment of the requirements for the degree of Master of Science (M. Sc.)

A Compiler for the Parallel Programming Language Actus

Luiz Eduardo Favre

Maio de 1992

Thesis Supervisor: Claudio Luís de Amorim
Department: Programa de Engenharia de Sistemas e Computação

Programming languages for parallel environments, should have adequate notation to express the problem inherent parallelism. Their compilers should generate target code in a situation where the language parallel constructs result in high performance.

This work presents a (front-end) compiler for the programming language Actus, oriented to vector and array processors. We describe how the compiler deals with data structures, expressions and statements, in parallel constructions.

Each construction which is described includes Actus syntax, the semantic actions involved, the internal representation by sub-trees and the resulting code in the intermediate representation.

Índice

Lista de Figuras

Capítulo I

Introdução

Os supercomputadores têm sido utilizados em aplicações onde um grande número de dados e operações devem ser executados em um curto período de tempo. Busca-se alcançar este objetivo através do uso de componentes tecnologicamente avançados e novas arquiteturas, que necessitam de sistemas operacionais e linguagens de programação apropriados.

O advento de uma geração de computadores mais poderosos, atende parcialmente à comunidade de usuários que necessitam de computadores de alta performance – os supercomputadores – para suas aplicações. Aumenta porém a gama de aplicações onde o computador pode ser utilizado, tornando viável várias aplicações que necessitam de um poder computacional ainda maior.

O avanço tecnológico na área de circuitos integrados, tem sido o principal responsável pela criação destes computadores, através do contínuo desenvolvimento de componentes eletrônicos menores, mais baratos, mais rápidos e com uma maior taxa de integração.

Na área de arquitetura de computadores, esforços têm sido feitos no projeto de novas arquiteturas paralelas, que permitem utilizar os componentes de maneira eficaz, afastando-se do modelo VonNeumann, ressaltando-se os computadores vetoriais, os multiprocessadores e os multicomputadores, permitindo que dezenas a milhares de operações possam ser executadas em paralelo.

Para que a alta velocidade de processamento dos supercomputado-

res seja traduzida em alto desempenho, é necessário que o sistema operacional da máquina e a linguagem de programação utilizados possam explorar ao máximo o paralelismo oferecido.

É grande a variedade de arquiteturas de supercomputadores que estão sendo pesquisadas e industrializadas. Vários sistemas operacionais e linguagens de programação têm sido propostos para operar nestes ambientes, oferecendo ao programador uma máquina virtual com um maior ou menor nível de abstração dos detalhes característicos da arquitetura da máquina alvo.

Um maior nível de abstração torna o ambiente de programação mais amigavel, facilitando também a criação de programas transportáveis. Isto contudo dificulta a paralelização eficiente do programa, exigindo o uso de compiladores mais complexos. Um menor nível de abstração, onde detalhes da arquitetura não sejam transparentes ao programador, torna o ambiente de programação mais árido, mas permite que o programador possa explorar melhor as características da máquina alvo.

A linguagem Actus [1], baseada em Pascal, fornece um alto nível de abstração, permitindo ao programador escrever programas bem estruturados onde o paralelismo é expresso sem transparecer as características da máquina alvo. Por ter sido projetada visando uma família específica de supercomputadores, os computadores vetoriais e matriciais, Actus permite ao compilador mapear o paralelismo existente no programa fonte diretamente no paralelismo permitido pela máquina. Isto porque Actus expressa o paralelismo na estrutura de dados, descrevendo operações seqüenciais, onde cada operação pode ser executada simultâneamente sobre um conjunto de dados, normalmente contidos em matrizes.

Contrária a filosofia de Actus a linguagem FORTRAN, é utilizada com alta eficiência nos supercomputadores vetoriais, graças à ação de compiladores otimizadores. Embora estes compiladores consigam gerar códigos bastante eficientes, pretende-se verificar se um programa escrito em uma linguagem onde o paralelismo é explicitado pelo programador, pode gerar programas mais eficientes com compiladores mais simples.

Embora em Actus o programador expresse o paralelismo diretamente, é possível, ser incluiído no compilador, um otimizador, para aumentar ainda mais a eficiência do código, visando liberar o programador de pequenos detalhes que possam aumentar a performance.

O compilador Actus é composto de duas partes, um front-end que transforma o programa fonte em uma representação intermediária (ou linguagem intermediária que passaremos a chamar de LI) e um back-end, que utiliza o código em LI para gerar o código objeto. O assunto desta tese é o front-end, tendo sido o back-end tratado em [5]. Por simplicidade, quando nos referirmos ao compilador Actus, estaremos nos referindo ao front-end. Os detalhes da LI e do compilador back-end só são tratados quando estritamente necessário.

São discutidos aqui os aspectos mais relevantes da construção do compilador com relação às estruturas paralelas de Actus. Os aspectos do compilador onde são utilizadas técnicas de conhecimento geral, como os processos de análise léxica e sintática, as operações com a tabela de símbolos e as operações com a pilha de atributos, não serão tratados aqui.

O compilador foi escrito originalmente em C (Turbo C) em um PC, e posteriormente transportado para o sistema operacional Helios, uma versão de UNIX para transputers. O analisador sintático utiliza o método R*S simples [6] com as tabelas de redução e empilhamento geradas automaticamente [7].

A compilação é feita em dois passos. O programa fonte é transformado, no primeiro passo, em uma árvore sintática, com os nodos decorados com informações semânticas. No segundo passo a árvore é transversa para a geração do código LI. O apêndice A descreve como cada nodo é formado e as várias famílias de sub-árvores existentes na árvore sintática. Recomendamos a leitura deste apêndice antes da leitura do capítulo III e posteriores.

O apêndice B apresenta com mais detalhes a formação de cada sub-árvore para eventual consulta. A gramática utilizada nesta implementação é apresentada no apêndice C. A menos de algumas pequenas diferenças na representação dos meta-símbolos, esta gramática é a mesma utilizada na geração das tabelas R*S.

O capítulo II apresenta a linguagem Actus, descrevendo principalmente os aspectos paralelos. O capítulo III trata da declaração de dados escalares e paralelos, procurando sempre que possível associar para cada construção discutida as regras gramaticais com: as ações semânticas, as sub-árvores geradas e o código LI correspondente. Os aspectos seqüenciais são apresentados, quando importantes, para facilitar a discussão das construções paralelas. Os capítulos posteriores seguem a mesma filosofia.

O capítulo IV trata da manipulação de expressões, cuja eficiência na transformação em código de máquina é fundamental para o desempenho do compilador. O capítulo V descreve as operações com os conjuntos de índices que definem o paralelismo nas expressões e, as operações que estes índices podem sofrer. O capítulo VI descreve os comandos de Actus e a formação de máscaras que limitam o número de dados a serem manipulados em paralelo nas expressões.

O capítulo VII apresenta as considerações finais quanto a eficiência do código gerado, as caracteríisticas da linguagem Actus que julgamos possam ser alteradas, os resultados obtidos, as perspectivas de desenvolvimentos futuros e a sugestão de alterações na LI, que passaria a oferecer duas máquinas virtuais e um mecanismo de sincronismo entre ambas, uma paralela para tratamento exclusivo das expressões paralelas, das máscaras e índices paralelos, e outra seqüencial, para processar o restante do programa. Pretende-se que a alteração sugerida para Actus e na LI permitam construir um compilador mais simples gerando um código de máquina mais eficiente.

Capítulo II

A Linguagem Actus

Actus é uma linguagem de programação paralela baseada na linguagem Pascal, e foi projetada para operar em ambientes paralelos síncronos tais como processadores vetoriais e matriciais. Actus fornece uma estrutura de dados e comandos para declarar, acessar e manipular dados em paralelo, permitindo ao programador escrever algoritmos paralelos bem estruturados e independentes da máquina alvo.

Adotando uma posição diametralmente oposta àquela adotada pelos compiladores vetorizadores usados, por exemplo em FORTRAN, Actus fornece ao programador uma notação que lhe permite definir o paralelismo que julgar adequado à solução do problema. Ao compilador Actus cabe traduzir este paralelismo definido pelo usuário no paralelismo físico permitido pela máquina alvo.

II.1 O Paralelismo na Estrutura de Dados

Os dados que se pretende manipular em paralelo devem ser declarados na forma de arrays. Actus permite que possam ser manipuladas em paralelo uma ou duas dimensões desses arrays. Na declaração dos arrays paralelos, os índices que serão acessados em paralelo devem ser declarados como tal, os demais índices são declarados normalmente.

```
VAR
   SCALAR_A   :   array[ 1..P ] of INTEGER;
```

```
PAR_A      :  array[ 1:P ]  of INTEGER;
```

Neste exemplo a primeira variável SCALAR_A é declarada como um array unidimensional de P elementos que só podem ser acessados individualmente. A segunda variável PAR_A é declarada também como um array unidimensional de P elementos, mas que podem ser acessados, simultaneamente, em paralelo.

No máximo duas dimensões de um array podem ser declaradas como paralelas. Na declaração dos índices do array a seqüência de pontos '..' indica que apenas um elemento será processado por vez. Para introduzir o paralelismo na declaração do array o índice deve ser declarado, trocando-se a seqüência de pontos '..' por dois pontos ':'.

O paralelismo pode ser aplicado em qualquer tipo inteiro válido em Pascal, por exemplo:

```
VAR
   ARRAY_MIXTO : array [ '0':'9' ], FALSE:TRUE ] of BOOLEAN;
   MULTI_DIM   : array [ 1..20, 1:20, 1:50 ]     of REAL;
```

O primeiro caso define um array bidimensional com um total de 20 elementos, todos podendo ser acessados simultaneamente. O segundo caso define um array tridimensional com duas dimensões paralelas, num total de 20.000 elementos dos quais, no máximo, 1.000 podem ser acessados simultaneamente.

Também é possível declarar um tipo array paralelo e depois declarar variáveis com aquele tipo, por exemplo:

```
TYPE
   MATRIX   =  array [ 10:50, 30:50 ] of REAL;
VAR
   MAT1, MAT2 :  MATRIX;
```

II.2 A Extensão de Paralelismo

Na declaração de um array paralelo é definida a máxima "extensão de paralelismo" (**edp**) que pode ser usada associada àquele array. Esta **edp** ou **edps** menores podem ser usadas para a manipulação dos dados do array nas expressões e comandos do programa. No exemplo anterior as variáveis paralelas MAT1 e MAT2 podem ser manipuladas com a **edp** declarada ou com uma **edp** menor. Quando as variáveis paralelas são usadas em expressões a **edp** deve ser a mesma, para todas as variáveis envolvidas na operação.

II.3 O Conjunto de Índices

Para permitir o acesso seletivo ou total aos elementos das variáveis paralelas, Actus utiliza uma construção denominada INDEX, que permite definir um conjunto de valores. Este conjunto contém os índices dos elementos que se deseja acessar, e pode ser declarado de dois modos distintos, caracterizando os seguintes tipos para estes conjuntos:

1. Conjunto explícito de índices. Na declaração atribui-se à variável do tipo INDEX valores que permanecem inalterados durante todo o escopo daquela variável INDEX.

2. Conjunto redefínivel de índices. Na declaração a variável do tipo INDEX assume um tipo inteiro (INTEGER, CHAR, ...), seus valores serão definidos mais tarde, possivelmente em tempo de execução.

Os conjuntos de índices só podem receber valores inteiros, sua declaração é feita à semelhança da declaração das variáveis comuns, como no exemplo abaixo.

```
INDEX
   UM_A_VINTE :  1:20;     {conjunto explicito}
   IS         :  INTEGER;  {conjunto redefinivel}
```

Neste exemplo UM_A_VINTE é associado a uma faixa de valores específica, IS por outro lado pode assumir qualquer faixa de valores desde que compatível com o tipo a que foi associado na declaração, neste caso, o tipo INTEGER. A atribuição é feita pelos comandos que definem a **edp** de um bloco de comando (isto é feito pelo comando USING, visto mais adiante).

O identificador do conjunto de índices é usado para indexar as variáveis paralelas, como por exemplo na declaração,

```
VAR
   MA : array [ 1:25 ] of INTEGER;
```

o termo MA[UM_A_VINTE] acessa simultaneamente os 20 primeiros elementos do array MA.

Em Actus um conjunto de índices pode ser manipulado para acessar (em paralelo) todos os elementos de um array unidimensional, e uma linha, coluna ou diagonal de um array bidimensional. Para acessar em paralelo uma grade de elementos de um array bidimensional, dois conjuntos de índices serão necessários. Por exemplo:

```
VAR
   AA : array [ 1:20, 1:20 ] of REAL;
INDEX
   IS, JS : 1:20;
```

No escopo desta declaração AA[IS,1] acessará, simultaneamente, os vinte elementos da primeira coluna de AA. AA[6,IS] irá acessar os vinte elementos da sexta linha de AA, e AA[IS,JS] irá acessar os 400 elementos de AA simultaneamente.

Os conjuntos de índices podem ser regulares ou irregulares. Os conjuntos regulares são conjuntos cujos elementos estão espaçados de um valor constante (que chamaremos de *step*). O espaçamento é especificado entre chaves na declaração do conjunto de índices. Se nenhum espaçamento for especificado (como nos exemplos vistos até aqui) assume-se o espaçamento de 1.

```
INDEX
   STEPPED_INDEX  :   1:[3]16;
```

Para o exemplo acima, os valores atribuídos ao índice STEPPED_INDEX são: 1, 4, 7, 10, 13 e 16.

Para permitir maior flexibilidade na declaração de índices, os operadores de união (+), interseção (*) e diferença (−) podem ser aplicados na declaração dos conjuntos de índices. Por exemplo:

```
INDEX
   BROKEN_RANGE: 2:[2]7 + 14:[2]21;
   RANDOM_RANGE: 1:[2]11 + 4[2]8 - 5:7 + 20:20;
```

No primeiro caso são especificados os valores 2, 4, 6, 14, 16, 18 e 20. No segundo caso os valores especificados são: 1, 3, 4, 8, 9, 11 e 20.

II.4 As Constantes Paralelas

É possível declarar identificadores para representar uma seqüência de valores constantes que possam ser acessados em paralelo. A forma de declaração é a seguinte:

```
PARCONST
  P1 = 3:7;
  STEPPED_SEQ = 1:[4]82;
  BROKEN_SEQ  = 1:4, 60:64;
  RANDOM_SEQ  = 3:5, 6:[2]10, 15, -1, 20[10]50;
```

As constantes paralelas podem ser usadas para atribuir valores iniciais aos arrays paralelos ou serem usados como termos em operações paralelas. Como ocorre com os índices, as constantes paralelas só podem assumir valores inteiros.

```
MA[UM_A_VINTE] :=  STEPPED_SEQ;
```

No caso acima nota-se que o número de elementos do array MA indexados por UM_A_VINTE coincide com o número de valores da constante paralela STEPPED_SEQ. Isto é obrigatório em todas as operações envolvendo dados paralelos, isto é, todos os termos devem ter a mesma extensão de paralelismo (**edp**).

II.5 Os Comandos Paralelos

Os comandos paralelos fornecem os meios de manipulação das constantes e variáveis paralelas. Estes comandos têm associado uma **edp** unidimensional ou bidimensional que determina quantos e quais elementos sofrerão a ação dos comandos. Mantendo a estrutura modular da linguagem, cada **edp** é associada a um bloco que constitui o escopo daquela **edp**. Todos os comandos inclusos naquele bloco possuirão a mesma **edp**.

A associação da **edp** a um bloco de comandos é feita através do comando USING, que tem a seguinte forma

using *index-specification* **do** *statement*

onde *index-specification* define a **edp** válida para os comandos de *statement*. Todos os comandos paralelos devem estar contidos em um comando USING. A concentração dos comandos com a mesma extensão de paralelismo em um mesmo bloco tem por objetivo aumentar a clareza do programa além de facilitar sua implementação e torná-la mais eficiente.

A definição da **edp** na parte *index-specification* do comando USING é feita utilizando-se o identificador do tipo INDEX, já associado a um conjunto explícito de índices, ou utilizando um identificador associado a um conjunto de índices redefinível, que é então associado ao conjunto de índices a ser usado naquele comando USING. A parte de definição de índices do comando USING representa uma máscara unidimensional ou bidimensional associada a todos os comandos contidos em *statement*.

Os identificadores de índices explícitos têm sua **edp** associada no momento da declaração; isto facilita a definição dos conjuntos de índices que não serão alterados dentro do bloco de comandos. Por exemplo:

```
VAR
   AA : array [ 1:50, 1:500 ] of INTEGER;
INDEX
   PRIM_IND  :  1:10;
   SEG_IND   :  1:500;
  ...
BEGIN
   USING  PRIM_IND, SEG_IND  DO
      AA[PRIM_IND, SEG_IND] := 1
END;
```

No comando acima os 25.000 elementos de AA recebem simultaneamente o valor 1.

Um identificador de índices redefiníveis é declarado como sendo de um dos tipos inteiros e tem sua **edp** associada ao comando USING. O uso de índices redefiníveis facilita a programação nos casos em que a **edp** varia freqüentemente. Por exemplo:

```
VAR
  I   : INTEGER;
  AC  : array [ 1:50, 'A':'Z' ] of INTEGER;
  A   : array [ 1:100 ]          of INTEGER;
INDEX
  IS  : INTEGER;
  CH  : CHAR;
 ...
BEGIN
  {assume-se que o valor de I e' definido antes deste ponto}
  USING IS := I:[2]49, CH := \'A':'Z' DO
```

```
    AC[IS\c CH] := -1;
  USING IS := 1:10 + (I:I) DO
    A[IS] := 0;
END;
```

Aqui os identificadores IS e CH foram declarados como sendo do tipo INTEGER e CHAR respectivamente, e receberam valores atuais no comando USING. Estes identificadores podem receber qualquer conjunto de índices desde que seus valores sejam compatíveis com o tipo pré-declarado para aquele índice, e podem ser redefinidos, como no exemplo acima, por outros comandos USING.

Durante a manipulação das variáveis paralelas em um comando USING bidimensional, os índices devem aparecer na ordem em que foram declarados no cabeçalho do comando, assim, o primeiro índice deve corresponder a primeira dimensão paralela da variável e o segundo, a segunda dimensão paralela da variável. Além disso, todas as variáveis paralelas devem ser indexadas por ambos os índices, ou seja, não pode haver acesso a apenas uma dimensão de uma variável em um comando USING bidimensional.

Estas restrições se devem ao fato do comando USING ter sido projetado de forma a que sua **edp** defina uma grade ou vetor de processadores que estarão ativos durante a avaliação dos comandos daquele bloco. Este vetor ou matriz deve permanecer inalterada até o fim do bloco USING ou até a definição de um novo bloco USING.

É permitido aninhar comandos USING indefinidamente. Contudo somente os índices do comando USING externo mais próximo, isto é, do comando USING mais recentemente aberto e que ainda não foi fechado permanecem ativos definindo a **edp** em uso. Por exemplo:

```
USING IS1, IS2 DO
  BEGIN
    ...  {IS1 e IS2 ativos}
    USING IS2 DO
```

```
      BEGIN
       ... {apenas IS2 ativo}
       USING IS1, IS4 DO
          BEGIN
          ... {IS1 e IS4 ativos}
          END
        ... {apenas IS2 ativo}
    END
...  {IS1 e IS2 ativos}
END;
```

O acesso à diagonal de um array é possível dentro de um comando USING unidimensional, indexando ambos os índices do array com o mesmo identificador do conjunto de índices, como mostrado no exemplo abaixo, onde a diagonal principal de AA receberá o valor 0.

```
VAR
   AA: array [1:100, 1:100] of INTEGER;
INDEX
   IS: 1:100;
BEGIN
   USING IS DO
      AA[IS, IS] := 0
END;
```

Se em um comando USING um identificador de índices definir um conjunto vazio, então nenhum comando no seu escopo será executado. Por exemplo:

```
USING IS = J:K DO        {onde J > K}
```

II.6 O Comando de Atribuição

O comando de atribuição, apresentado informalmente na seção anterior, será paralelo quando a variável à esquerda do sinal de atribuição for paralela. Similarmente, o resultado de uma expressão será paralelo, se pelo menos um dos termos envolvidos for paralelo. O resultado será escalar, se ambos os termos forem escalares. Neste comando todos os elementos (variáveis e constantes) envolvidos devem ter a mesma **edp**, esta restrição é também feita para ambos os termos de uma expressão binária.

Para permitir que um valor escalar possa ser atribuído aos elementos de uma variável paralela e que também possa ser manipulado em expressões paralelas, os escalares adotam, implicitamente, o valor da edp vigente, quando utilizados em um comando de atribuição ou em expressões paralelas. Assim, o valor escalar será replicado em tantos elementos quantos forem o número de índices da **edp**. Os operadores permitidos nas expressões paralelas são os mesmos utilizados nas expressões escalares.

II.7 Os Comandos Condicionais Paralelos

Actus reconhece dois comandos condicionais, o comando IF e o comando CASE (que tem a forma definida abaixo) sendo que o comando IF pode ter a cláusula ELSE omitida.

if *bool_expr* **then** *statement*
if *bool_expr* **then** *statement* **else** *statement*

Se a expressão do comando IF for escalar a **edp** vigente não será alterada, e os comandos de *statement*, se forem executados, estarão sob a influência daquela **edp**. Se a expressão for paralela, a mesma será avaliada segundo a **edp** atual e seu resultado fornecerá um vetor ou uma grade de valores TRUE ou FALSE, que será usada como máscara para a **edp** atual, gerando uma nova **edp** a ser usada para os comandos do IF relativos a cláusula **then**. Para os comandos da cláusula **else** os

valores da máscara são complementados antes de gerar a nova **edp**.

```
VAR
   AA, BB: array [1:50, 1:50] of REAL;
INDEX
   IS, JS:  1:50;
BEGIN
   USING IS, JS DO
      IF AA[IS,JS] >= 00  THEN
         BB[IS,JS] := BB[IS,JS] - 0.5
      ELSE
         BB[IS,JS] := BB[IS,JS] + 0.5
END;
```

Neste exemplo os componentes de BB correspondentes aos valores positivos de AA são decrementados de 0.5, sendo que os demais valores de BB são incrementados de 0.5.

Além dos operadores boleanos **and**, **or** e **not** os operadores **any** e **all** podem ser usados.

```
USING IS, JS DO
   IF  any( AA[IS,JS] >= 99.9 )  THEN
        AA[IS,JS]  00
```

O efeito do exemplo acima é de iniciar todos os elementos de AA com 0, se algum de seus elementos for maior ou igual a 99.9.

O comando CASE tem a seguinte forma:

case *expr* **of**
case_label_list1 : S1;
case_label_list2 : S2;
...

end

Assim como no comando IF, o comando CASE será tratado de forma diferente se *expr* for ou não paralela. Se *expr* for paralela, a **edp** vigente será distribuída entre os vários ramos do comando CASE, cada qual utilizando uma máscara própria, resultado da comparação da expressão com a lista de valores de cada ramo. Cada ramo é executado em seqüência até que todos os ramos do comando CASE tenham sido executados.

```
VAR
   XX : array [1:15, 1:20] of 1..9;
INDEX
   IS,JS : INTEGER;
BEGIN
   USING IS := 1:15, JS := 1:20 DO
      CASE  XX[IS,JS] OF
         1,2,4:  XX[IS,JS] := XX[IS,JS] -1;
         5..7 :  XX[IS,JS] := -1;
         3,8,9:
      END
END
```

O efeito do comando CASE, no exemplo acima, é o de acessar todos os elementos de XX, utilizando a **edp** vigente, i.e, aquela especificada pelo comando USING, e então usar estes valores de XX para determinar para cada ramo do comando CASE, quais os elementos que devem ser acessados, ou seja, definindo uma máscara para cada um desses ramos. Assim, os valores de XX iguais a 1, 2 ou 4 são decrementados de 1, os valores de XX entre 5 e 7 são transformados em -1, e os demais permanecem sem alteração. Não é semânticamente correto que um mesmo elemento possa ser acessado em mais de um ramo do comando CASE.

II.8 Os Comandos Paralelos de Repetição

Os comandos de repetição de Actus são REPEAT, WHILE e FOR. Para os comandos REPEAT e WHILE a expressão boleana de contrôle pode ser escalar ou paralela. Para o comando FOR a expressão deve ser sempre escalar.

O comando WHILE tem a seguinte forma:

while *boolean_expression* **do** *statement*

Se a expressão boleana for escalar os comandos em *statement* são executados repetidamente sem alterar o escopo da **edp** vingente, e portando sob sua influência. Estes comandos são executados sob influencia daquela **edp** enquanto o valor de *boolean_expression* for TRUE.

Se a expressão boleana for paralela, cada vez que esta expressão for avaliada antes de cada iteração, uma nova máscara será criada, alterando o valor da **edp** ativa para aquela iteração.

```
VAR
   AA, BB : array [1:15, 1:15] of INTEGER;
INDEX
   IS, JS : 1:15;
BEGIN
   USING IS, JS DO
      WHILE AA[IS,JS] >= 0 DO
        BEGIN
           BB[IS,JS] := BB[IS,JS] + BB[IS,JS] * AA[IS,JS];
           AA[IS,JS] := AA[IS,JS] - 4
        END
   END
END
```

Neste exemplo os comandos internos ao WHILE são executados repetidamente, en-

quanto pelo menos um dos elementos de AA for maior que 0. A **edp** original de 1:15, 1:15 não é válida para os comandos no corpo do loop, sendo seu valor alterado cada vez que a expressão

```
AA[IS,JS] >= 0
```

for avaliada. O elemento de AA menor ou igual a 0 tem seu índice correspondente retirado da **edp**. Quando a **edp** ficar vazia o comando WHILE termina.

O comando REPEAT tem a seguinte forma.

repeat *statements* **until** *boolean_expression*

Este comando é uma variação do comando WHILE e comporta-se exatamente como aquele com respeito à variação da **edp** válida dentro do comando, com a diferença de que a **edp** é alterada ao fim de cada iteração.

O comando FOR tem a seguinte forma.

for *var* := *expr*1 **to** *expr*2 **do** *statement*
for *var* := *expr*1 **downto** *expr*2 **do** *statement*

Neste comando *var* e *expr* são escalares e os comandos de *statement* sempre atuam sob a **edp** vigente antes do comando FOR. A expressão *expr*1 define o valor inicial de *var* e a expressão *expr*2 define o valor final de *var*. A cada iteração o valor da variável *var* é incrementado ou decrementado de 1, dependendo da direção TO ou DOWNTO definida no comando. O número de iterações é calculada antes da primeira iteração e não pode ser alterado pelos comandos executados a cada iteração.

II.9 O Aninhamento dos Comandos Paralelos

Quando um comando paralelo (excluindo-se o comando de atribuição) faz parte do corpo de outro comando paralelo, se não ocorrer redefinição da **edp** por um

comando USING, a **edp** vigente dentro do novo comando será a **edp** vigente antes da abertura deste novo comando, alterada pela máscara criada pela expressão de contrôle do comando. Quando o comando paralelo é fechado a **edp** vigente passa a ser a que estava em vigor imediatemente antes deste comando ser aberto.

II.10 Funções e Procedimentos

Actus segue as mesmas regras de Pascal quanto às declarações de funções e procedimentos, bem como de seus parâmetros. O mesmo ocorre para o modo como são chamados nos comandos. Esta implementação contudo não suporta que arrays conformantes sejam passados como parâmetro na chamada de funções e procedimentos.

II.11 Deslocamento de Índices

Para permitir uma maior versatilidade no acesso aos elementos de uma variável paralela, dois operadores de deslocamento são oferecidos: o operador SHIFT e o operador ROTATE. Estes operadores atuam sobre o conjunto de índices sem alterar o número de elementos que contêm, deslocando-os para a direita ou para a esquerda em um movimento circular (ROTATE) ou unidirecional (SHIFT). Cria-se assim uma nova **edp** que permanece contudo compatível com a anterior.

Para exemplificar o uso desses operadores, mostramos como atuam sobre dois pequenos conjuntos de índices.

```
  USING IS := 2:4, JS := 2:6 - 3:4 DO

IS  2,3,4     IS SHIFT 1   3,4,5     IS SHIFT -1   1,2,3
              IS ROTATE 1  3,4,2     IS ROTATE -1  4,2,3

JS  2,5,6     JS SHIFT 1   3,6,7     JS SHIFT -1   1,4,5
              JS ROTATE 1  5,6,3     JS ROTATE -1  6,2,5
```

O exemplo abaixo demonstra a iniciação de duas diagonais do array P.

```
VAR
   P : array [1:100,1:100] of INTEGER;
INDEX
   IS : INTEGER;
BEGIN
   USING IS := 1:99 DO
      P[IS,IS shift 1] := 0;
      P[IS shift 1,IS] := 0;
   END
END;
```

Os operadores de deslocamento podem ser usados em qualquer variável paralela, em ambos os lados do comando de atribuição. Por exemplo.

```
USING IS := 2:4, JS := 2:6 - 3:4 DO
   AA[IS shift -1, JS rotate -1] := BB[IS rotate 1, JS shift 1];
```

Este comando produz as seguintes atribuições:

```
AA[1,6] := BB[3,3];   AA[1,2] := BB[3,6];   AA[1,5] := BB[3,7];
AA[2,6] := BB[4,3];   AA[2,2] := BB[4,6];   AA[2,5] := BB[4,7];
AA[3,6] := BB[2,3];   AA[3,2] := BB[2,6];   AA[3,5] := BB[2,7];
```

Para permitir acesso aleatório aos elementos de um array, utiliza-se um outro array como índice, usando o que chamamos de indexação independente. O array usado como índice deve ser inicialmente carregado com os valores dos índices que se deseja acessar no primeiro array. Esta operação é mais importante quando o array a ser indexado tem dimensão igual ou maior que 2. Por exemplo.

```
CONST
   N = 50;
PARCONST
   VETOR = N:[-1]1;
```

```
VAR
   MATRIX :  array [1:N, 1:N] of REAL;
   INDEX_A : array [1:N]      of INTEGER;
INDEX
   IS :  1:N;

BEGIN
   USING  IS DO
     INDEX_A := VETOR;
     MATRIX[IS, INDEX_A[IS]] := 0
   END
END;
```

Neste exemplo INDEX_A é utilizado para permitir que os componentes da diagonal secundária do array MATRIX sejam acessados em paralelo.

Capítulo III

Declaração de Dados

Este capítulo trata dos aspectos mais importantes referentes à declaração de dados, tipos e procedimentos. Os casos específicos relacionados às construções paralelas são tratados com mais detalhes. Trechos em código LI são apresentados para ilustrar os casos mais importantes.

Inicialmente apresenta-se a estrutura da tabela de símbolos e as operações básicas de inserir, consultar, e retirar informações da tabela.

Descreve-se a seguir as sub-árvores da árvore sintática referentes às declarações dos tipos simples e estruturados (arrays e records), das variáveis, constantes, e temporárias, bem como as sub-árvores referentes às declarações das funções e procedimentos. Juntamente com os arrays discute-se a construção da tabela que descreve os arrays para LI.

Fechando o capítulo detalha-se as construções relativas às constantes paralelas, aos índices paralelos, e às temporárias paralelas. Vê-se também a tabela de índices que reune para uso da LI, todas as informações referentes às extensões de paralelismo utilizadas.

III.1 Tabela de Símbolos

A tabela de símbolos permite acessar as informações sobre um identificador a partir do seu nome. Estas informações, bem como o próprio nome do identificador, estão

contidas nas sub-árvores da árvore sintática. A única estrutura de dados específica da tabela de símbolos é uma tabela hash, onde cada entrada aponta para o nodo identificador da sub-árvore colocada mais recentemente na tabela de símbolos com aquele código hash. A entrada na tabela hash conterá o ponteiro vazio NULL se não existir nenhum identificador com o correspondente código hash. A Figura 1 ilustra duas entradas da tabela hash (thash), uma com três identificadores e outra vazia. Os ponteiros *prim*, *prox* e *tipo* são representados pelas letras **m**, **x** e **t** respectivamente.

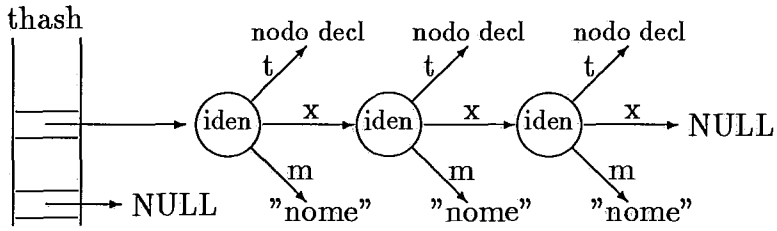


Figura III.1: Thash e a lista de nodos com o mesmo código hash

Os nodos identificadores na tabela de símbolos com o mesmo código hash estão interligados através do ponteiro *prox*. O ponteiro *tipo* liga o nodo identificador a raiz da sub-árvore de declaração da qual faz parte e que contém as informações sobre o identificador. O ponteiro *prim* liga o nodo identificador ao string com o seu nome.

O nodo declaração informa se o identificador corresponde a uma variável, a um tipo, a uma constante, a uma função, etc ... . As informações quanto ao tipo do identificador, se inteiro, real, etc ... , estão no nodo apontado pelo ponteiro *tipo* do nodo declaração. A Figura 2 apresenta as interligações do nodo identificador. Nesta figura o nodo identificador está com o nome *iden*, o nodo declaração com o nome *decl*.

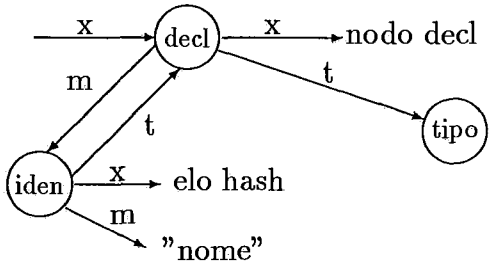


Figura III.2: Nodos que descrevem um identificador

O ponteiro *prox* do nodo declaração liga este nodo a outros nodos de declaração, formando o caminho a ser percorrido durante a geração de código LI. Este nodo será acessado através do ponteiro *prox* do nodo de declaração anterior. O ponteiro *prim* aponta para o nodo identificador, a fim de permitir acesso ao nome do identificador a partir do nodo de declaração. Do nodo **tipo** não é possível alcançar o nodo declaração.

III.1.1 Inserção de Dados

A operação de inserção na tabela é feita logo após a sub-árvore de declaração do identificador estar construída. É calculado o valor que define o código hash do nome ligado ao nodo identificador, o ponteiro *prox* do nodo identificador recebe o conteúdo da tabela hash indexada pelo código hash. Esta entrada da tabela hash recebe o endereço do nodo identificador.

O nodo identificador recebe como atributo o número que caracteriza último bloco aberto e que permanece aberto, definindo assim o escopo do identificador. Enquanto permanecer na tabela hash, o identificador estará na tabela de símbolos. É gerada uma situação de erro se existir na tabela outro nodo identificador com o mesmo nome e com o número do bloco atual, caracterizando dupla declaração do identificador no mesmo bloco.

III.1.2 Consulta

A consulta à tabela de símbolos é feita a partir do nome do identificador, que define uma entrada na tabela hash. A lista de nodos relativa àquela entrada é percorrida até encontrar-se um nodo com o mesmo nome ou chegar-se ao fim da lista, neste caso o identificador não foi declarado ou não é acessível por já ter sido retirado da tabela.

Se o identificador não for encontrado tem-se como resposta à consulta o ponteiro NULL, caso contrário recebe-se como resposta o nodo de declaração do identificador. A partir deste nodo é possível acessar às informações sobre o

identificador montadas durante a fase de declaração.

III.1.3 Retirada de Dados

Quando um bloco é fechado todos os identificadores declarados naquele bloco são retirados da tabela de símbolos. Esta operação é mais demorada que a inserção e a consulta, e é feita visitando-se todas as listas de nodos contidas na tabela hash.

Cada lista é percorrida retirando-se todos os nodos visitados até encontrar-se um nodo que não pertença ao bloco recém-fechado ou o fim da lista. Retira-se um nodo da lista passando o conteúdo do seu ponteiro *prox* para a tabela hash, o ponteiro então recebe o valor NULL. Os nodos retirados da tabela hash não podem mais ser acessados através da tabela de símbolos, mas continuam a fazer parte da árvore sintática.

III.2 Declaração de Tipos

A declaração de tipos em Actus permite, com a declaração de tipos simples (não estruturados), melhorar a inteligibilidade do programa fonte, e com a declaração de arrays e records (tipos estruturados) facilitar a descrição de estrutura de dados complexas, mais adequadas à representação do problema que se deseja tratar.

A declaração de tipos envolve duas operações distintas, ambas atuando na construção da árvore sintática.

1. A criação de uma nova sub-árvore ou busca na tabela de símbolos da sub-árvore que define o tipo que se deseja declarar. Esta sub-árvore passa a caracterizar o tipo.

2. A criação de uma sub-árvore *typedecl* que liga o nome do tipo à sub-árvore que caracteriza este tipo. Esta sub-árvore passa a identificar o tipo e permite que o mesmo seja acessado através da tabela de símbolos.

Nenhuma dessas duas sub-árvores produz código LI, contudo os tipos

estruturados interferem na geração do código LI durante a declaração das variáveis, porque a LI suporta apenas os tipos pré-definidos e os arrays de tipos pré-definidos.

Para simplificar a discussão da declaração de tipos, trataremos as duas operações separadamente. Será apresentada primeiro a operação de construção da sub-árvore que identifica o tipo, com a sub-árvore que caracteriza o tipo já montada. Logo após, trataremos da construção das sub-árvores que caracterizam os vários tipos aceitos pela linguagem, sem considerar a sua interligação com seu identificador.

III.2.1 Identificação dos Tipos

A regra gramatical responsável pela declaração de tipos é a seguinte:

typedecl $\longrightarrow$ *iden* '=' *type* ;

onde *iden* corresponde a um string ASCII com o nome do identificador e *type* corresponde a uma sub-árvore **tipo** que pode ser:

1. uma sub-árvore já existente, acessada através da tabela de símbolos.
2. uma sub-árvore construída especialmente para esta declaração.

Em ambos os casos a sistemática adotada é a mesma, uma vez que os procedimentos necessários ao tratamento desses casos já foram executados anteriormente (estes procedimentos serão discutidos na próxima seção), estaremos assim lidando com uma sub-árvore **tipo** não levando em consideração os passos adotados para formar esta sub-árvore, nem que informações ela contém.

A sistemática adotada para tratar a declaração de tipos é a seguinte, sendo estes três passos são sempre executados para todas as declarações:

1. Construir uma sub-árvore *iden* e ligá-la ao string com o nome declarado para o tipo.

2. Colocar a sub-árvore *iden* na tabela de símbolos, verificando se não está ocorrendo dupla declaração do mesmo identificador. Colocar no atributo de *iden* o número que caracteriza o bloco atual.

3. Criar uma sub-árvore *typedecl*, interligá-la ao nodo *iden* e à sub-árvore **tipo**.

A sub-árvore *typedecl* montada tem a configuração apresentada na Figura 3–a.

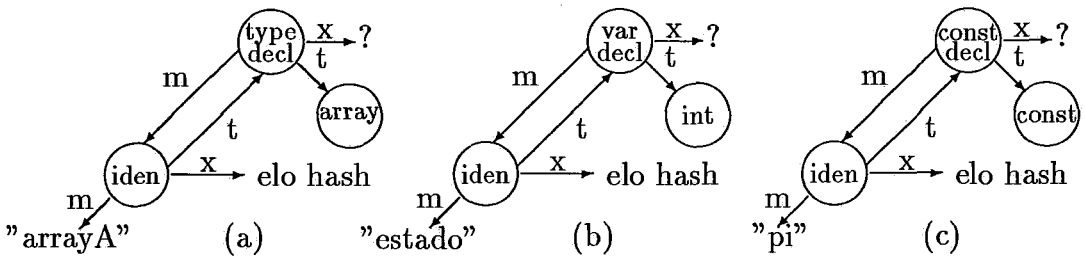


Figura III.3: Exemplos de sub-árvores de declaração

Decorre do acima exposto que as declarações de tipos como:

```
type
  tipo_a  :  tipo_x;
  tipo_b  :  tipo_a;
  tipo_c  :  tipo_b;
```

provocam a criação de três sub-árvores *typedecl* semelhantes, todas apontando para a mesma sub-árvore **tipo** que contém os dados que caracterizam o tipo '`tipo_x`'. A diferença entre as três sub-árvores estará nos nodos *iden*, um ligado ao string "tipo_a", outro ao string "tipo_b" e o último ao string "tipo_c".

III.3 Construção dos Tipos

Os tipos em Actus são construídos a partir de tipos já existentes na tabela de símbolos. Originalmente existem apenas os tipos primitivos, pré-definidos pelo compilador, antes de iniciar a compilação. As sub-árvores que caracterizam os tipos

pré-definidos são da classe *predef*. Para representar os tipos construídos pelo programa fonte são utilizadas sub-árvores da classe *struct*. Estes tipos são: subranges, enumerados, records e arrays.

É importante saber quantas variáveis escalares e matriciais devem ser declaradas na LI, em correspondência a cada variável declarada em Actus. Para isso os nodos que definem o tipo das variáveis (nodos *predef* e *struct*) têm estas informações como atributo. Os campos que contêm esta informação nos nodos da classe *struct* são *n_mtrz* e *n_esc*, definindo o número de matrizes e de escalares do tipo. Para os nodos da classe *predef* estes atributos são definidos implicitamente tendo *n_esc* = 1 e n_mtz = 0. Os nodos *subr* e *enum* têm sempre *n_esc* = 1 e *n_mtrz* = 0. Os nodos *array* têm *n_esc* = 0 e *n_mtrz* >= 1, e os nodos *record* têm *n_esc* >= 0 e *n_mtrz* >=0 (sendo *n_esc* + *n_mtrz* >= 1).

As informações contidas nestes atributos são redundantes, podendo ser adquiridas percorrendo os nodos descendentes do nodo **tipo** em questão. Contudo, para agilizar a compilação, estes atributos são registrados no nodo raiz.

III.3.1 Tipos Primitivos

O compilador possui 7 tipos pré-definidos que servem de base para os demais tipos simples ou estruturados que podem ser declarados no programa fonte. São eles: o tipo inteiro de 16, 34 e 64 bits, os tipos real de 32 e 64 bits, o tipo booleano e o tipo caracter, representados, respectivamente, pelos seguites identificadores: *integer*, *int32*, *int64*, *real*, *real64*, *boolean* e *char*, suas sub-árvores são criadas no início da compilação e permanecem na tabela de símbolos durante todo o processo de compilação.

A LI também reconhece estes, e somente estes, tipos onde são representados pelos números:

`30000` O tipo inteiro representado por 32 bits, corresponde ao identificador *int32*.

`30001` O tipo inteiro representado por 16 bits, corresponde ao identificador *integer*.

30003 O tipo inteiro representado por 64 bits, corresponde ao identificador *int64*

30004 O tipo real representado por 32 bits, 8 para o expoente e 23 para fração, Padrão ANSI-IEEE(754-1985), corresponde ao identificador *real*.

30005 O tipo real representado por 64 bits, 11 para o expoente e 52 para a fração, Padrão ANSI-IEEE(754-1985), corresponde ao identificador *real64*.

30006 O tipo inteiro representado por 8 bits, corresponde ao identificador *char*.

30007 O tipo booleano correspondente ao identificador *boolean*.

Ao ser traduzido para a LI os tipos enumerados são representados por 30006, o mesmo que representa o tipo *char*. Por isso, os tipos enumerados estão restritos a conter no máximo 256 elementos.

Os tipos não estruturados definidos pelo programa fonte são traduzidos diretamente em um dos tipos pré-definidos. A Figura 4 mostra a sub-árvore do tipo pré-definido *integer* e uma sub-árvore genérica relativa a um tipo não estruturado definido no programa fonte, como por exemplo:

```
type
  novotipo = integer;
```

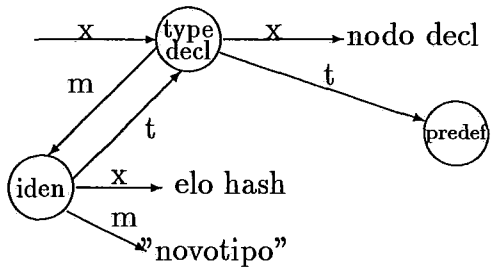


Figura III.4: Tipo pré-definido e tipo definido no programa fonte

III.3.2 Tipo Enumerado

A regra que constroi este tipo é a seguite:

simple_type ⟶ '('*lista_de_identificadores* ')'

onde *lista_de_identificadores* corresponde a uma lista enumerando todos os identificadores (na ordem desejada) que pertencerão ao tipo enumerado, como no exemplo abaixo.

```
type
  simple_type  ⟶  (seg, ter, qua, qui, sex, sab, dom);
```

Cada um dos identificadores da lista é colocado em uma sub-árvore *enumdecl* como mostra a Figura 5.

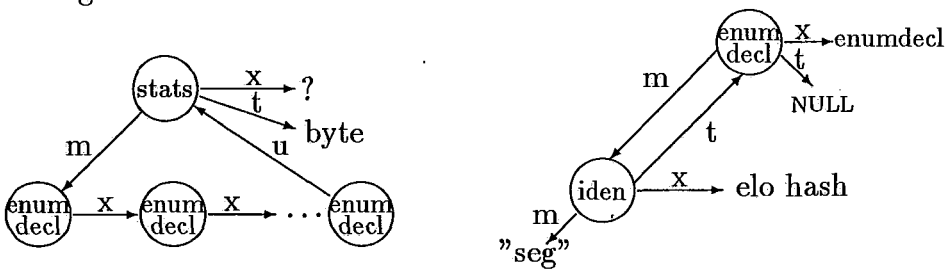


Figura III.5: Sub-árvores envolvidas na construção de tipos enumerados

Todos estes nodos são filhos do nodo *enum* que representará o tipo enumerado. Este nodo recebe por atributo $n\_mtrz = 0$ e $n\_esc = 1$.

O nodo *enumdecl* tem como atributo a posição relativa do seu identificador na lista, sendo que a numeração das posições inicia em 0. Todos os identificadores que formam o tipo *enum* são colocados na tabela de símbolos, onde permanecem, enquanto o bloco onde foram declarados não for fechado.

III.3.3 Tipo Subrange

A sub-árvore que caracteriza este tipo está representada na Figura 6. Esta sub-árvore *subrange* é montada a partir de uma das seguites regras gramaticais.

$$\begin{aligned} simple\_type \longrightarrow\ & const_1\ \text{“..”}\ const_2 \\ \mid\ & const_1\ \text{“:”}\ const_2 \end{aligned}$$

onde $const_1$ é o limite inferior e $const_2$ é o limite superior da faixa de valores compreendida pelo tipo.

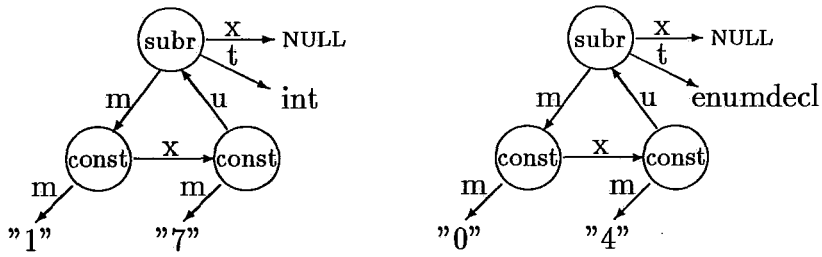


Figura III.6: Exemplos de sub-árvores do tipo subrange

A segunda regra gramatical difere da primeira por definir um tipo subrange paralelo. As sub-árvores são formadas do mesmo modo, com a exceção do nodo raiz que passa a ser *psubr*.

Se $const_1$ e $const_2$ corresponderem a elementos de um tipo enumerado o nodo raiz *subr* terá como tipo a sub-árvore daquele tipo enumerado, e os nodos *const* apontarão para strings com os valores correspondentes à ordem dos elementos enumerados dentro de seu tipo. A sub-árvore (b) da Figura 6 apresenta este caso com referência ao exemplo abaixo.

```
type
  simple_type  ⟶  1 .. 7
  simple_type  ⟶  seg .. sex
```

O atributo dos nodos *enum* é sempre $n\_mtrz = 0$ e $n\_esc = 1$.

III.3.4 Tipo Record

As regras envolvidas na construção deste tipo são:

(1) *type* ⟶ RECORD *lista_de_fields* END

(2) *lista_de_fields* ⟶ *lista_de_fields* "," *field*

| ε

(3) *field* ⟶ *iden* ":" *type*

onde *lista_de_fields* é uma lista composta dos campos que formam o record.

A definição do tipo record é recursiva, possibilitando a construção de tipos bastante complexos, contudo, devido às características da estrutura das sub-árvores do tipo *record* e da sistemática adotada para a sua construção é possível tratar estas regras com algoritmos bastante simples. A Figura 7 ilustra uma sub-árvore *record* com três campos.

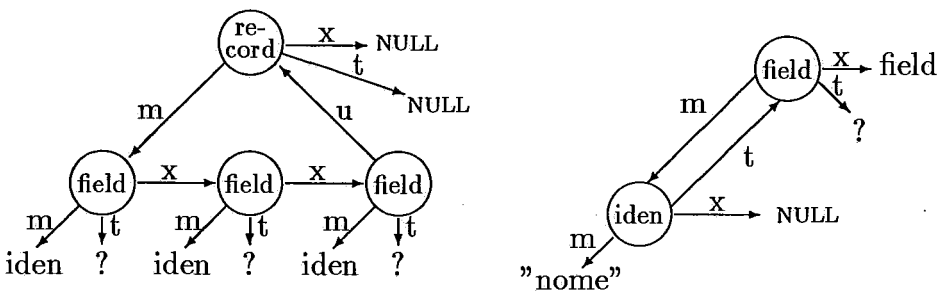


Figura III.7: Sub-árvores envolvidas na construção do tipo record

Devido às características da gramática de Actus as sub-árvores que representam os campos do record já estão montadas (pelas ações relativas as regras 2 e 3), quando a regra número 1 é ativada para comandar a criação da sub-árvore *record*.

O algoritmo utilizado correspondente à cada uma das regras gramaticais acima, é o seguinte:

(regra 3) Montar a sub-árvore que forma um campo do record.

1. Construir uma sub-árvore *iden* e ligá-la ao nome do identificador deste campo do record (o identificador não é colocado na tabela de símbolos).
2. Criar uma sub-árvore *field* e interligá-la ao nodo *iden* e a sub-árvore que define o tipo deste campo.

(regra 2) Acrescentar mais um campo a lista.

1. Acrescentar ao fim da lista de sub-árvores *field* a sub-árvore recém criada pela regra número 3.

(regra 1) Criar a sub-árvore que define o tipo record.

1. Criar um nodo *record* e ligá-lo aos seus filhos. Estes estão contidos na lista criada pela regra número 2.

2. Verificar a não existência de mais de um campo com o mesmo identificador.

3. Percorrer todos os nodos filhos, fazendo o somatório dos atributos *n_mtrz* e *n_esc* de seus tipos.

4. Colocar como atributo do nodo *record* em *n_mtrz* o resultado total do somatório de *n_mtrz*, e em *n_esc* o somatório de *n_esc*.

III.3.5 Tipo Array

A sub-árvore que caracteriza o tipo array está representada na Figura 8 como um array tri-dimensional. As regras envolvidas na construção deste tipo são:

(1) *type* ⟶ `ARRAY` *lista_de_indices* `of` *type*

(2) *lista_de_indices* ⟶ *lista_de_indices* "," *indice*

| *indice*

(3) *indice* ⟶ *simple_type*

Assim como o tipo record, o tipo array é definido recursivamente, podendo gerar tipos (e interligações de sub-árvores) bastante complexos. Também como no tratamento dos records, o algoritmo de tratamento das regras gramaticais acima são simples.

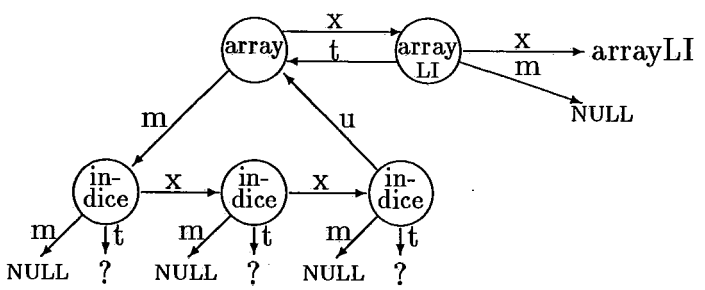


Figura III.8: Sub-árvores envolvidas na construção do tipo array

A regra número 2 é resolvida do pelo mesmo processo utilizado para atender à regra 2 do tipo record não sendo repetida aqui. Esta regra une em uma

lista os índices do array que, se tiver dimensão N, terá N nodos na lista. O algoritmo para as demais regras é o seguite:

(regra 3) Criar o nodo que representa o índice e ligá-lo ao seu tipo.

1. Criar um nodo *índice*, ligá-lo ao seu nodo **tipo** construído (pelos processos que tratam os tipos enumerados e subranges) para representar *simple_type*.

(regra 1) Criar a sub-árvore que define o tipo array.

1. Criar um nodo *array*, interligando-o à lista de nodos *índice* que passam a ser filhos daquele nodo *array*.
2. Ligar o nodo *array* ao seu nodo tipo (relativo a *type*).
3. Definir os atributos do nodo *array*, fazendo *n_esc* = 0 e colocar em *n_mtrz* a soma dos valores de *n_esc* e *n_mtrz* do seu nodo **tipo**.

O nodo *array* é o único nodo da classe *struct* que utiliza o ponteiro *prox*. Este ponteiro é utilizado para ligar o tipo array ao nodo *arrayLI*, que determina a identificação do array para a LI. Para evitar que os arrays que não forem associados a variáveis sejam colocados na relação dos arrays reconhecidos pela LI, a interligação dos nodos *array* com os nodos *arrayLI* só é feita durante a declaração de variáveis.

Quando uma variável é declarada como sendo do tipo array, se o nodo *array* não estiver ligado a um nodo *arrayLI*, um nodo *arrayLI* será criado e interligado neste momento ao nodo *array*. Este processo será discutido na próxima seção porque, mesmo ocorrendo durante a fase de declaração de variáveis, está logicamente relacionado a declaração do tipo array.

III.4 O Array Para a Linguagem Intermediária

O único tipo estruturado reconhecido pela LI é o tipo array, que deve contudo ser formado por elementos de um dos tipos primitivos. Por isso arrays de records devem

ser decompostos em tantos arrays quantos forem os campos do record (supondo que os tipos dos campos sejam primitivos).

A LI permite que as variáveis e as temporárias sejam declaradas localmente, dentro do bloco (sub-programa) onde são usadas, utilizando as mesmas regras de escopo de Actus. Contudo, a LI requer que o escopo dos tipos arrays seja global, e que sejam declarados em separado das demais declarações e comandos da LI. em uma tabela de arrays que concentra as informações sobre todos os tipos de arrays utilizados.

A tabela de array é organizada de forma que cada entrada corresponda a um tipo array. As informações contidas em cada entrada são: o tipo dos elementos que compõem o array, a dimensão do array, e os dados referentes a cada uma das dimensões. Para cada dimensão indica-se se a mesma pode ser acessada em paralelo ou somente de modo seqüêncial, e quantos elementos compõem a dimensão.

Uma variável do tipo array é declarada na LI, fazendo referência à entrada na tabela de arrays que contém os dados sobre o seu tipo. As entradas na tabela de array são numeradas em sequência, iniciando em 0.

```
type
  TA = array [1:50, 1:250, 1..500];
var
  array_a : TA;
```

Para o array acima a entrada na tabela de arrays da LI será a seguinte.

```
:30004; 3; 1;50; 1;250; 0;500;
```

onde

`30004;` – indica que o array é formado por elementos do tipo real (ver codificação LI para os tipos primitivos).

`3;` – indica que o array tem três dimensões.

`1;50;` – indica primeira dimensão paralela com 50 elementos.

`1;250;` – indica segunda dimensão paralela com 250 elementos.

`0;500;` – indica terceira dimensão escalar com 500 elementos.

Os nodos *arrayLI* são utilizados para formar na árvore sintática uma representação da tabela de arrays da LI. Esta tabela só será gerada no segundo passo da compilação, após a árvore sintática estar toda construída.

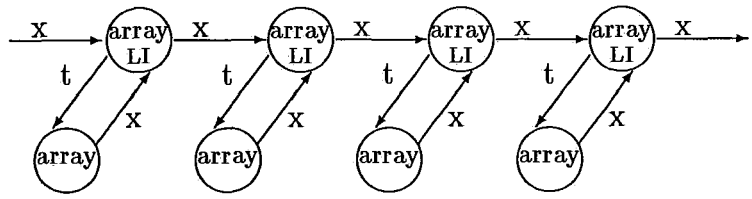


Figura III.9: A interligação de nodos *arrayLI*

A Figura 9 ilustra a formação de um nodo *arrayLI* e como é formada a lista destes nodos. Cada nodo *array* pode ser formado de tipos primitivos ou estruturados, de modo a gerar uma ou mais entradas na tabela de arrays, assim o nodo *arrayLI* correspondente deve reservar tantas entradas na tabela quantos forem os arrays definidos pelo tipo.

O nodo *arrayLI* tem como atributo o número da entrada na tabela de arrays onde serão colocadas as informações do seu nodo *array*. Se ao nodo *array* corresponder mais de uma entrada na tabela de arrays, o nodo *arrayLI* guardará como atributo, o número da primeira entrada. As demais entradas serão subseqüentes a esta.

Nas seções que se seguem apresentamos os algoritmos utilizados para:

1. Colocar um nodo *array* na lista de nodos *arrayLI*.
2. Gerar a tabela de arrays a partir da lista de nodos *arrayLI*.
3. Identificar a entrada na tabela correspondente a uma variável array durante o tratamento de expressões.

III.5 Criação da Lista de Arrays

Os nodos *arrayLI* formam uma lista que permite ao compilador localizar e acessar dentro da árvore sintática todos os tipos arrays utilizados pelo programa fonte.

Estão associadas a esta lista um contador de linhas (o número da linha caracteriza a entrada na tabela de arrays) inicializado com 0, um ponteiro que aponta para o primeiro nodo da lista inicializado com NULL, e um ponteiro que aponta para o último nodo da lista tambem inicializado com NULL. A lista de nodos é construída fazendo cada nodo *arrayLI* apontar para o próximo nodo da lista através do ponteiro *prox*.

Quando uma variável é declarada o seguite algoritmo é executado a partir do nodo *vardecl*:

1. Visitar o nodo **tipo**, se o nodo for *enum*, *subr*, *psubr*, ou corresponder a um dos tipos primitivos, interromper o processo. Se o nodo **tipo** for *array* executar o passo 2. Se o nodo **tipo** for *record*, para cada nodo *field* do record tornar a executar o passo 1.

2. Se o nodo *array* já estiver ligado a um nodo *arrayLI* interromper o processo. Caso contrário executar os seguites sub-passos:

   2.1 Criar um nodo *arrayLI* que recebe como atributo o valor atual do contador de linhas.

   2.2 Incrementar o contador de linhas com o valor do atributo *n_mtrz* do nodo *array*.

   2.3 Interligar os nodos *arrayLI* e *array* através de seus ponteiros *tipo* e *prox*.

   2.4 Colocar o nodo *arrayLI* na lista, se o ponteiro para o início da lista contiver NULL, fazer ambos os ponteiros apontarem para o nodo *arrayLI*. Caso contrário colocar este nodo *arrayLI* no fim da lista, fazendo com que o nodo que estava no fim da lista e o ponteiro de fim de lista apontem para este novo nodo.

III.5.1 Geração da Tabela de Arrays

Na fase de geração de código a tabela de arrays é a primeira a ser criada, sendo cada entrada na tabela transformada em uma linha de código LI. Para a construção desta tabela o algoritmo abaixo é executado, partindo do nodo *arrayLI* no início da lista.

1. Partindo do início da lista executar o passo 2 para cada nodo da lista.
2. Gerar uma lista de índices vazia (lista_ind) e acessar o nodo **tipo** deste nodo, executar o passo 3 com estes parâmetros (lista_ind, nodo_tipo).
3. Parâmetros (lista_ind, nodo_tipo).

   Se nodo_tipo for *array*.

   - Concatenar à lista_ind às informações sobre os índices deste nodo.
   - Tornar a executar o passo 3, passando como parâmetros a nova lista_ind e o nodo **tipo** deste nodo *array*

   Se nodo_tipo for *record*, repetir os sub-passos abaixo para cada nodo *field*.

   - Criar uma cópia da lista_ind.
   - Acessar o nodo **tipo** deste nodo *field*.
   - Executar o passo 3 passando, como parâmetros, a cópia da lista_ind e o nodo **tipo** deste campo.

   Caso contrário (tipo é primitivo).

   - Gerar uma linha de código LI com os dados sobre os índices do array lidos na lista_ind e do tipo primitivo do array definido por nodo_tipo.

III.6 Declaração de Variáveis do Tipo Array

Após gerar as tabelas de arrays e de edp's da LI, a árvore sintática é percorrida para gerar o código LI propriamente dito, compreendento as declarações e comandos.

O algoritmo utilizado na declaração das variáveis do tipo array é utilizado no tratamento de todos os tipos de variáveis (incluindo records e tipos primitivos), por isso sua apresentação é feita na seção que trata da declaração de variáveis. Trataremos aqui apenas das características específicas no tratamento dos arrays.

A declaração de variáveis do tipo array são feitas na LI fazendo referência á linha na tabela de arrays que contém os dados daquele array. O número da linha é encontrado no nodo *arrayII* ligado ao nodo *array* que define o tipo da variável. Se o array for composto de records, então, à váriavel Actus corresponderá à várias variáveis na LI, e possivelmente ao tipo array corresponderão várias entradas na tabela de arrays. Não importando como o array é decomposto, apenas um nodo *arrayLI* é utilizado, contendo o número da primeira entrada na tabela de arrays. Cabe ao algoritmo de declaração das variáveis, analisando o tipo do array, declarar o número de variáveis definidas pelo tipo e referenciá-las à linha correspondente na tabela de arrays.

A declaração na LI de uma variável do tipo array tem a seguinte forma:

```
DEC; arrayA; 37;
```

onde:

`DEC;` Palavra chave para a LI indicando tratar-se de uma declaração de variável.

`arrayA;` Nome com o qual a variável é conhecida na LI[1].

`37;` Número da linha na tabela de arrays que contém os dados do array.

III.7 Declaração de Constantes

A declaração das constantes tem a única finalidade de melhorar a inteligibilidade do programa Actus. Para a LI, que não possui construções para a declaração de

[1] As variáveis são renomeadas ao serem traduzidas para a LI

constantes, estas declarações são transparentes, porque quando um identificador de constante é encontrado, o compilador o substitui pelo seu valor.

A regra gramatical responsável pela declaração de constantes é a seguite:

constdecl ⟶ *iden* "=" *expr*

onde *iden* corresponde a um string ASCII com o nome do identificador e *expr* deve ser um string ASCII com o valor da constante inteira, real, ou corresponder a uma operação aritmética entre inteiros.

Os passos dados para tratar esta regra são:

1. Se *expr* for uma operação aritmética entre constantes inteiras, calcular o resultado e usá-lo em substituição a *expr*.
2. Construir uma sub-árvore *iden* e ligá-la ao string com o nome.
3. Colocar a sub-árvore *iden* na tabela de símbolos.
4. Criar um nodo *const* e ligá-lo ao string ASCII com o resultado da expressão e à sub-árvore que define o tipo da constante.
5. Criar um nodo *constdecl* e ligá-lo aos nodos *iden* e *const* (esta sub-árvore é ilustrada na Figura 3–c).

III.8 Declaração de Variáveis

A declaração de variáveis é feita em Actus observando-se a seguite regra gramatical,

vardecl ⟶ *iden* ":" *type* ";"

onde *iden* corresponde a um string ASCII com o nome do identificador, e *type* corresponde a uma sub-árvore **tipo**, podendo ser da classe *struct* ou *predef*. O

algoritmo que trata esta regra é idêntico ao que trata a declaração de tipos, não sendo repetida aqui.

A construção da sub-árvore *vardecl* tem a mesma construção das sub-árvores *constdecl* e *typedecl*, como pode ser observado na Figura 3–b. A sub-árvore *vardecl*, contudo, tem seu atributo *offset* carregado com o valor atual de offset, o valor de offset é em seguida incrementado do resultado da soma dos atributos *n_mtrz* e *n_esc* relativos ao nodo **tipo** correspondente a *type*. Assim é reservado no espaço de nomes da LI tantos nomes quantas forem as variáveis a serem declaradas na LI, para aquele tipo.

A declaração de variáveis na LI tem a seguite forma:

```
DEC; nomeLI; 30000;
```

onde:

`DEC;` Palavra reservada LI indicando declaração de variável.

`nomeLI;` Nome gerado pelo compilador para a variável.

`30000;` Tipo da variável (neste exemplo INT32).

A declaração das variáveis para a LI é feita no segundo passo da compilação, logo após a geração das tabelas de arrays e edp's, durante a visita aos nodos da árvore sintática, que é então percorrida a partir da raiz, sendo gerado o código LI correspondente a cada nodo visitado.

O algoritmo, utilizado quando um nodo *vardecl* é encontrado, é descrito a seguir.

1. Ler no nodo *vardecl* as seguites informações que devem ser passadas como parâmetros ao passo 2.

   - O nodo **tipo** de *vardecl* (nodo_tipo).

- O atributo *offset* de *vardecl*.
- O número da linha na tabela de arrays (inicializado com -1).

2. Parâmetros (nodo_tipo, offset, linha).
   - Se nodo_tipo é array executar passo 4.
   - Se nodo_tipo é record executar passo 5.
   - Caso contrário executar passo 3 (tipo primitivo).

3. Tipo primitivo (nodo_tipo, offset, linha).
   Se linha = -1: gerar nome de variável simples, utilizando os dados de nodo_tipo e offset.
   Caso contrário: gerar nome de variável matricial, utilizando os dados de nodo_tipo, offset e linha.

4. Tipo array (nodo_tipo, offset, linha).
   Se linha = -1: fazer linha receber o atributo do nodo *arrayLI* ligado ao nodo_tipo.
   Utilizar como novo nodo_tipo o nodo que define o tipo do nodo_tipo atual, executar o passo 2 com nodo_tipo e linha atualizados.

5. Tipo record (nodo_tipo, offset, linha).
   Visitar cada um dos campos (nodos *field*) do record, para cada nodo *field* utilizar como novo nodo_tipo o nodo **tipo** deste campo, executar o passo 2 com nodo_tipo, offset e linha atualizados. Incrementar offset de 1 logo após executar o passo 2.

III.9 Declaração de Temporárias

As expressões são transcritas para a LI usando uma notação de três endereços, tornando necessário o uso de temporárias para reter o resultado parcial das expressões. O compilador monta uma estrutura de dados na qual registra o número de temporárias de cada tipo utilizadas em cada sub-rotina (funções e procedimentos), num

processo que será discutido com mais detalhes, posteriormente, junto com o tratamento das expressões. Estes dados são utilizados para declarar o número exato de temporárias utilizadas no bloco. As temporárias são declaradas dentro do escopo do bloco em que são utilizadas, isto é, são sempre variáveis locais.

Um contador é mantido para cada tipo primitivo. Quando o bloco de comando de uma sub-rotina é fechado, estes contadores são consultados para a geração das sub-árvores *tempdecl*. Estas sub-árvores são então inseridas entre as sub-árvores que correspondem à declaração de dados da sub-rotina.

Para cada tipo primitivo o seguinte processo é executado, formando uma lista com as temporárias de cada tipo.

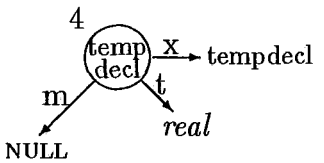
1. Se o valor do contador for zero, interromper o processo e passar para o próximo tipo.

2. Criar um nodo *tempdecl* cujo atributo recebe o valor do contador, ligá-lo à sub-árvore que define seu tipo (sempre um tipo primitivo).

3. Acrescentar à lista de nodos *tempdecl* o nodo criado no passo 2. Repetir o passo 1 para o próximo tipo.

Ao fim do processo a lista de nodos *tempdecl*, se não estiver vazia, é inserida entre as sub-árvores *vardecl* da sub-rotina em questão.

A declaração das temporárias na LI é feita exatamente como para as demais variáveis não estruturadas, com a diferença de que o atributo *offset* determina quantas temporárias serão declaradas para cada nodo *tempdecl*. O trecho de código LI da Figura 10 corresponde ao código gerado para a árvore da Figura 10, supondo que o valor de seu *offset* seja 4.




```
DEC; TR001; 30004;
DEC; TR002; 30004;
DEC; TR003; 30004;
DEC; TR004; 30004;
```

Figura III.10: Exemplo de sub-árvore *tempdecl* e seu código LI

III.10 Acesso aos Dados Paralelos

O acesso aos dados paralelos é feito em Actus por um dos dois modos relacionados abaixo:

1. Pela indexação de arrays paralelos por identificadores do tipo INDEX com os valores definidos quando os identificadores são declarados, ou no comando USING que os utilizam[2]

2. Pelo uso de constantes paralelas em expressões e comandos de atribuição paralelos.

A LI trata ambos os casos de modo semelhante, utilizando o que denomina "tabela de edp's" para concentrar todas as informações sobre estes índices e constantes. Isto é possível porque ambos consistem num conjunto de valores, com a diferença de que os índices formam um conjunto não ordenado e as constantes formam um conjunto ordenado.

As regras gramaticais de Actus envolvidas são:

(1) *parconst_decl* $\longrightarrow$ *iden* "=" *exprs* ";"
(2) *index_decl* $\longrightarrow$ *iden* ":" *expr* ";"
(3) *edp* $\longrightarrow$ *iden* ":=" *expr* ";"

A regra número 1 é responsável pela declaração das constantes paralelas, a regra número 2 pelos conjuntos de índices explícitos, e a regra número 3 pelos conjuntos de índices redefiníveis. O trecho do programa em Actus que se segue apresenta um exemplo correspondente a cada uma das regras acima. A Figura 11 apresenta as sub-árvores correspondentes onde as figura (a), (b) e (c) correspondem às regras 1, 2 e 3, respectivamente.

PARCONST

[2]Ver item 1.3 – O Conjunto de Índices e item 1.5 – Os Comandos Paralelos

```
   BROKEN_STEP = 1:4, 60:64;          {regra 1}
INDEX
   BROKEN_RANGE : 2:[2]7 + 14:[2]21;  {regra 2}
BEGIN
  USING IS := 1:[2]49 DO ...          {regra 3}
```

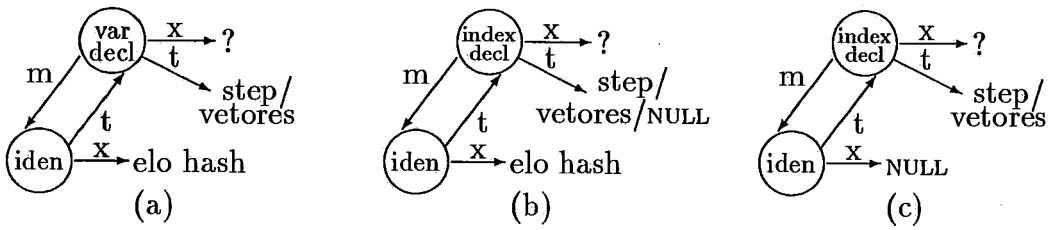


Figura III.11: Exemplos de sub-árvores de declaração de conjuntos paralelos

O conjunto regular é representado por uma sub-árvore *step*, e o irregular por uma sub-árvore *vetores*, a Figura 12 apresenta estas sub-árvores.

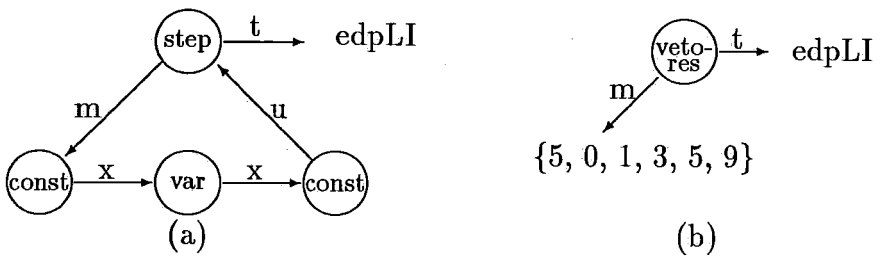


Figura III.12: Sub-árvores que definem os conjuntos paralelos

A LI concentra as informações sobre os conjuntos regulares em uma lista chamada "tabela de edp's", de construção semelhante à tabela de arrays. Cada linha da tabela é uma entrada na mesma, sendo acessada pelo número da linha. A contagem inicia em 0.

Os valores irregulares não podem ser representados na tabela de edp's, sendo representados nesta tabela, indiretamente, quando se tratar de conjunto de índices. Os conjuntos de constantes irregulares não são representados de nenhuma forma na tabela.

III.10.1 Conjuntos Regulares

Estes conjuntos são representados pela sub-árvore *step* mostrada na Figura 13. Esta sub-árvore pode ter variações, dependendo do valor do espaçamento (step) entre os

valores do conjunto.

1. O espaçamento é conhecido em tempo de compilação.

2. O espaçamento deve ser calculado em tempo de execução.

No primeiro caso a sub-árvore *step* terá apenas dois filhos, o primeiro indicando o valor inicial do conjunto e o segundo o valor final. O valor do espaçamento é guardado pelo próprio nodo *step*, sendo permitidos espaçamentos entre -32 767 a +32 767.

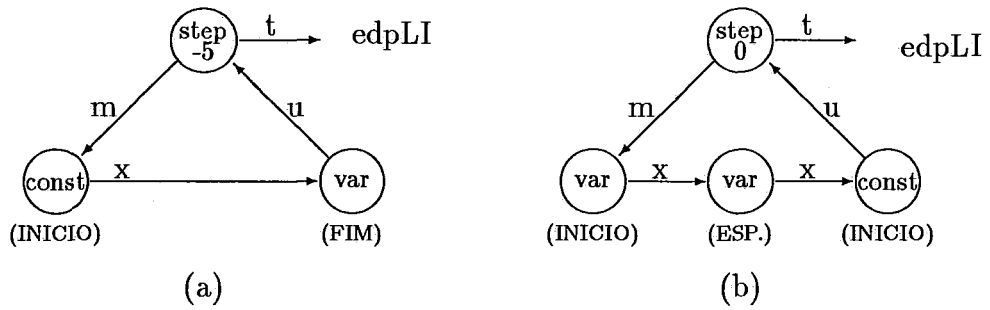


Figura III.13: Exemplos de construções de sub-árvores *step*'s

No segundo caso a sub-árvore *step* terá três filhos, onde o primeiro e o terceiro filhos indicam respectivamente o valor inicial e final do conjunto. O valor do espaçamento é guardado pelo segundo nodo. Os nodos filhos da sub-árvore *step* podem ser *const* e *var*, sendo que os nodos que representam o espaçamento, se existirem, serão sempre nodos *var*.

O ponteiro *tipo* liga o nodo step ao nodo *edpLI*, que indica qual a entrada na tabela de edp's conterá os dados deste conjunto de valores para uso da LI.

III.10.2 Conjuntos Irregulares

Os conjuntos irregulares são representados pela sub-árvore *vetores* apresentada na Figura 12–b. Este nodo aponta para uma lista de inteiros que contém todos os nodos do conjunto. O primeiro elemento da lista contém o tamanho da lista, um conjunto com n elementos será representado por uma lista com $n+1$ posições. Esta

representação não é muito eficiente, ocupando muito espaço de memória e tempo de máquina para ser processado, agravado pelo fato de que a LI só pode representar operações entre dois ou mais conjuntos regulares como um conjunto irregular. Outra limitação é a de que operações entre conjuntos de índices regulares só são possíveis se os valores inicial, final e espaçamento de todos os conjuntos envolvidos forem conhecidos em tempo de compilação.

Embora os conjuntos irregulares não sejam representados na tabela de edp's, os nodos *vetores* usados como índices são ligados a um nodo *edpLI*, porque ocupam indiretamente uma entrada nesta tabela, como veremos mais adiante.

III.11 Os Conjuntos na Representação da LI

Os conjuntos de valores reconhecidos pela LI são concentrados na tabela de edp's, onde cada entrada ou linha da tabela representa um destes conjuntos ou edp's, sendo que a cada linha corresponde um nodo *step* ou nodo *vetores*.

Assim como os nodos *arrayLI*, os nodos *edpLI* são interligados, formando uma lista no primeiro passo da compilação, para permitir a posterior criação da tabela de edp's. Os dados para as entrada da tabela são lidos nos nodos *step* e *vetores* interligados aos nodos *edpLI*.

A Figura 14 apresenta algumas das várias formações possíveis para os nodos *edpLI*. O atributo desses nodos é o número da linha que ocuparão na tabela de edp's.

O algoritmo de criação da lista de nodos *edpLI* é semelhante a da lista de nodos *arrayLI*, e não será apresentado.

Quando a tabela de edp's é montada cada entrada pode ter uma das seguintes construções:

1– *valor inicial;* *valor do incremento;* *tamanho;*

2– *valor nulo;* *vetor de valores;* *tamanho;*

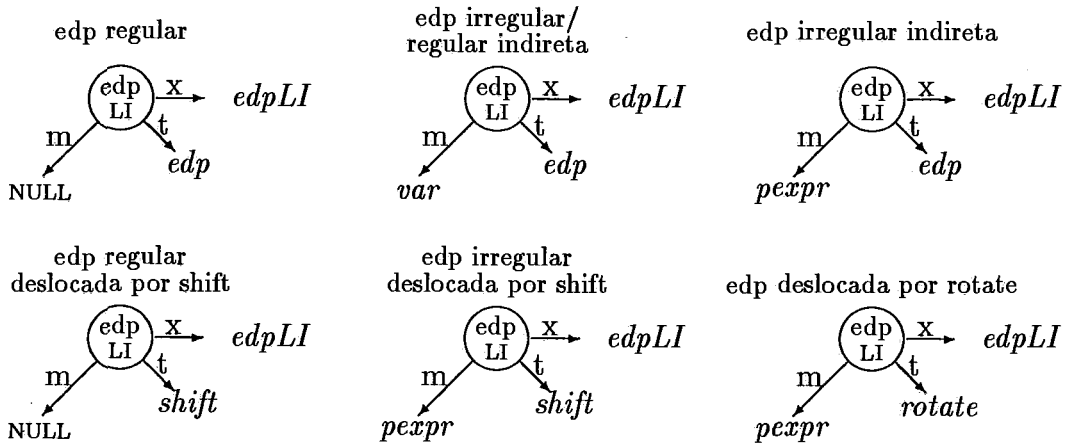


Figura III.14: Construções possíveis para sub-árvores *edpLI*

O campo com valor nulo é representado pelos valores `1; 0`, os demais podem assumir uma das duas construções:

1. Valor conhecido em tempo de execução:
   `0;` *valor inteiro*

2. Valor a ser computado em tempo de execução:
   `1;` *nome da variável*

Essas entradas ainda podem receber mais um campo adicional necessário a LI, contendo o número de outra entrada na tabela, indicando que os dados da entrada em questão foram criados a partir de deslocamentos nos valores daquela outra entrada.

Abaixo apresentamos exemplos de algumas entradas possíveis na tabela de edp's.

`:0;0; 0;1; 0;15;` {início 0, incremento 1, tamanho 15}

`:1;0; 1;A; 0;15;` {array A contém o vetor de tamanho 15}

`:0;3; 0;1; 0;25; 4;` {início 3, incrementol, tamanho 25, dados gerados a partir de deslocamento da edp da entrada 4}

`:0;B; 1;C; 1;D;` {início em B, incremento em C, tamanho em D}

No último exemplo a variável D recebe o resultado da expressão (supondo que o conjunto tenha sido declarado como B: [C] E, onde a variável E determine o valor final):

```
D := ((E - B) / C) + 1;
```

Note que se a notação adotada pela LI fosse a mesma de Actus, isto é, utilizando o valor final e não o tamanho do conjunto, o código gerado seria mais simples, agilizando a compilação e a execução do programa, uma vez que a expressão acima, se necessário, poderia ainda ser executada automaticamente pela LI.

As entradas 1 e 4 do exemplo acima são criadas diretamente a partir do nodo *step*, sendo que para o caso 1 o tamanho pôde ser calculado em tempo de compilação e no caso 4 o compilador deve gerar uma expressão que calcule o tamanho posteriormente (em tempo de execução).

O segundo caso foi gerado a partir de um nodo *vetores*, e o terceiro caso a partir de deslocamentos nos valores de um nodo *step*, supostamente na entrada 4 da tabela.

Estes casos serão vistos com mais detalhes posteriormente.

III.12 Declaração de Constantes Paralelas

As constantes paralelas são declaradas de acordo com a seguinte regra gramatical:

parconst_decl $\longrightarrow$ *iden* "=' *exprs* ";"

onde *iden* corresponde a um string ASCII com o nome do identificador e *exprs* corresponde a um nodo *step* ou *vetores*, formado quando as regras gramaticais de *exprs* foram tratadas.

Como já foi dito, apenas as constantes paralelas regulares são colocadas na tabela de edp's, tornando assim diferenciados os processos utilizados pela LI para tratar as duas construções permitidas para as constantes paralelas.

A Figura 15 apresenta duas sub-árvores *parcdecl*, uma regular outra irregular.

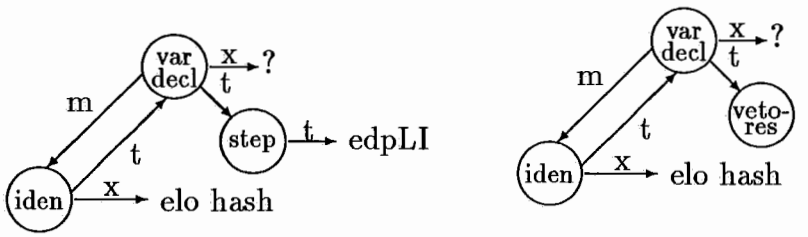


Figura III.15: Exemplos de sub-árvores *pardecl*

O algoritmo correspondente a esta regra é similar aos já apresentados para a declaração de constantes, tipos e variáveis, não sendo visto aqui. A diferença desta declaração é que após a construção da sub-árvore mostrada na Figura 13, se o nodo **tipo** for *step*, um nodo *edpLI* será criado, interligado ao nodo *step* e colocado no fim da lista de nodos *edpLI*.

Assim como as constantes escalares, a declaração de constantes paralelas não gera código LI. A seguir mostramos alguns exemplos de declarações de constantes paralelas em Actus.

```
PARCONST
  P1 = 1:15;
  P2 = 1:[PULO]FIM;
  P3 = 3:5, 6:[2]10;
```

Neste exemplo P1 e P2 terão como tipo nodos *step*'s sendo que para P1 *step* terá dois filhos e seu atributo será 1. O nodo *step* de P2 terá três filhos e seu atributo será 0. Ambos os nodos *step* serão interligados a nodos *edpLI*.

III.13 Declaração dos Conjuntos de Índices

O conjunto de índices é uma estrutura de dados em torno da qual são executadas todas as operações paralelas de Actus. Toda a notação de Actus, para exprimir paralelismo, está associada aos conjuntos de índices.

Na seção Acesso aos Dados Paralelos foram discutidos alguns aspectos referentes ao conjunto de índices e apresentada a regra gramatical que define a declaração dos mesmos. As sub-árvores construídas são mostradas na Figura 12.

As sub-árvores *indexdecl* construídas a partir da regra 1 podem ter como nodo **tipo** nodos *step* ou *vetores*, sendo o índice colocado imediatamente na lista de nodos *edpLI*, se seu tipo for *step*. As declarações de constantes paralelas não geram código LI, apenas ocupam entradas na tabela de edp's (se forem regulares).

As sub-árvores com *int* como nodo **tipo** não são colocadas na lista de nodos *edpLI*, isto só ocorrenrá quando o índice receber os seus valores em um comando USING. Cada uma das demais sub-árvores (com tipo *step* e *vetores*) ocupa uma entrada na tabela de edp's, sendo que se o tipo for *vetores* um procedimento mais complexo deve ser executado para que o índice possa ser utilizado na LI.

Como os índices irregulares serão usados para indexar arrays paralelos, e a LI só aceita indexar estes arrays através da tabela de edp's, e como esta tabela não pode representar os conjuntos irregulares a seguinte solução alternativa é adotada:

1. Criar uma variável paralela uni-dimensional com tantos elementos quantos forem os do conjunto de índices irregular em questão.

2. Criar uma sub-árvore que atribua os valores do índice irregular à variável recém criada. A atribuição será feita em tempo de execução.

3. Criar uma entrada na tabela de edp's (através de um nodo *edpLI*) com um conjunto de índices regular fictício, com espaçamento 1 e com tantos elementos quantos os que estão contidos no índice irregular.

4. Substituir posteriormente nos comandos, todas as indexações feitas utilizando o índice irregular, por uma indexação indireta, utilizando a variável criada para este fim.

O trecho de código abaixo exemplifica como é oneroso o tratamento de índices irregulares. O primeiro trecho apresenta o código como realmente foi

escrito, o segundo mostra como este código deve ser transformado pelo compilador para que o mesmo possa ser traduzido em código LI.

```
{A - trecho do programa como foi escrito}
  USING IS := 1:3 + 5:7 DO
    A[IS] := B[IS];

{B - trecho trasformado para poder ser traduzido para a LI}
  VAR
    Vx : array[1:6] of INTEGER;
  PARCONST
    Px = 1,2,3,5,6,7;
  USING Ix := 1:6 DO
    Vx[Ix] := Px;
    A[Vx[Ix]] := B[Vx[Ix]];
```

A transformação acima é feita pelo compilador através da geração de sub-árvores reproduzem o trecho de programa B, embora tenha lido o trecho A. Isto implica em um programa mais lento pelo número de dados a declarar, mais entradas na tabela de edp's, e mais operações e acessos à memória.

Parte dos comandos acrecentados em B já foi visto neste capítulo, os demais serão vistos nos capítulos que tratam de comandos e expressões.

III.14 Declaração de Temporárias Paralelas

As temporárias paralelas são necessárias pela mesma razão que torna necessário o uso das temporárias escalares. A mesma sub-árvore *typedecl* usada para declarar as temporárias escalares é usada na declaração das temporárias paralelas, sendo que estas têm como **tipo** um nodo *array*, devidamente colocado na lista de nodos *arrayLI*.

Outra diferença é que o escopo de cada temporária paralela é restrito a um comando USING. Como foi visto, as expressões paralelas (onde as temporárias

são usadas) só podem existir dentro dos blocos dos comandos USING, e cada comando USING define uma edp que rege todas as operações paralelas dentro do bloco. Assim, as temporárias paralelas têm a edp do comando USING onde são usadas, e não podem ser usadas em outro comando USING (a menos que ambos tenham a mesma edp, mas por simplicidade cada comando USING utiliza suas próprias temporárias).

Assim as sub-árvores *tempdecl* paralelas são declaradas junto ao comando USING a que pertencem (ao contrário das escalares que são declaradas junto com as demais variáveis). Como a LI permite declarações de variáveis em qualquer bloco, ficando o escopo da variável limitado àquele bloco, a declaração das temporárias paralelas junto ao seu comando USING é correta e adequada.

Juntamente com a discussão sobre as expressões paralelas é mostrada a estrutura de dados e o algoritmo utilizado para determinar quantas temporárias paralelas por tipo devem ser declaradas para cada comando USING.

III.15 Funções e Procedimentos

As regras da gramática Actus que definem a declaração de sub-rotinas (Funções e Procedimentos) são as seguites:

(1)	*proc_part*	⟶	*proc_head bloc* ";"
(2)	*proc_head*	⟶	PROCEDURE *iden formal_part* ";"
(3)		|	FUNCTION it iden *formal_part* ":" *iden* ";"
(4)	*formal_part*	⟶	"(" *param_groups* ")"
(5)	*param_groups*	⟶	*param_groups param_group*
(6)		|	*param_group*
(7)	*param_group*	⟶	VAR *idens* ":" *type*
(8)		|	*idens* ":" *iden*

As regras que formam o não terminal *bloc*, não são mostradas porque delas derivam todas as demais regras da gramática (ver apêndice 1). As demais

regras comandam principalmente a verificação semântica das construções e para interligar as sub-árvores que representam todo o corpo da sub-rotina. Estas sub-árvores estão definidas no apêndice 3 e são somente mencionadas aqui.

pfbody Sub-árvore que une as sub-árvores que definem o nome da sub-rotina, seus parâmetros e seus comandos.

procdecl Sub-árvore de declaração que contém o nome do procedimento.

fundecl Sub-árvore de declaração que contém o nome da função e o seu tipo.

params Sub-árvore que concentra os parâmetros da sub-rotina.

paramref Sub-árvore de declaração do parâmetro passado por referência.

paramval Sub-árvore de declaração do parâmetro passado por valor.

bloco Sub-árvore que contém toda a parte de declarações e de comandos da sub-rotina.

A Figura 16 apresenta alguns desses nodos.

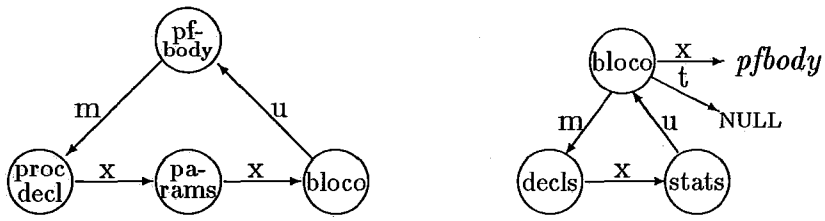


Figura III.16: Sub-árvores envolvidas na declaração de sub-rotinas

Note que os parâmetros são declarados no mesmo escopo que as demais declarações contidas no bloco. A regra número 1 acima, retira da tabela de símbolos todas as variáveis declaradas em seu bloco assim como seus parâmetros.

Abaixo apresentamos um trecho do código LI relativo à declaração de um procedimento (`rotinaA`) com três parâmetros (`Pa, Pb, Pc`. Este trecho é mostrado para que se possa ter uma noção da aparência do código na LI. A LI é descrita em detalhes em [2].

```
PROC; rotinaA; 0; 3; 20; {3 parametros, 20 linhas de codigo}
  PAR; Pa; 30003; 1;     {valor}
  PAR; Pb; 30004; 0;     {referencia}
  PAR; Pc; 30005; 1;     {valor}
  BLOCK; 16;
    ...                  {16 linhas de codigo}
```

Capítulo IV

Expressões e Atribuições

Este capítulo trata da manipulação das expressões e comandos de atribuição paralelos e escalares. Para cada tópico discutido, quando for conveniente, serão apresentados:

- A regra gramatical envolvida e trechos do programa em Actus.
- A representação correspondente na árvore sintática.
- As estruturas de dados utilizadas.
- Os algoritmos de controle.
- O código LI gerado.

O comando de atribuição não é tratado junto com os demais comandos no próximo capítulo, porque o tratamento das expressões e dos comandos de atribuição têm muito em comum. Embora os aspectos ligados ao paralelismo sejam tratados com mais detalhes, algumas questões relacionadas a atribuições e expressões escalares são discutidas para gerar uma base para a discussão das operações paralelas. A menos que seja mencionado em contrário, as observações feitas para as expressões são válidas para os comandos de atribuição.

Em última instância, todo o paralelismo de Actus está voltado para o tratamento de expressões e atribuições, e a eficiência do programa dependerá fundamentalmente da eficiência com que são tratadas as expressões. Assim, é importante

que as expressões sejam tratadas eficientemente, levando-se em consideração tanto a movimentação dos dados entre a memória e a CPU, quanto as operações lógicas e aritméticas executadas.

Como será visto no próximo capítulo, todas as operações paralelas são feitas dentro de um comando USING. Este comando define o ambiente onde as expressões serão avaliadas, permitindo que as operações de "set-up" da máquina alvo sejam executadas apenas uma vez para todo um grupo de comandos.

A seguir veremos o tratamento de comandos de atribuição e expressões escalares, as operações unárias e binárias, como são tratados os termos da expressão quando forem constantes, variáveis simples, variáveis indexadas, variáveis do tipo record ou ainda o resultado parcial de outra expressão. Para as variáveis indexadas discute-se a correção dos índices que não iniciam em 1.

Será visto como são utilizadas as variáveis criadas pelo compilador para receber os resultados parciais das expressões (temporárias), como são requisitadas e liberadas, e como é feita a contagem de linhas de código LI, que serão necessárias ao tratamento das expressões.

No próximo capítulo, fechando o tratamento de expressões, são vistos alguns detalhes diretamente ligados a definição do paralelismo nas expressões. As expressões paralelas são avaliadas dentro de um ambiente paralelo pré-definido, regido por uma edp uni ou bidimensional. A edp é representada por um ou dois conjuntos de índices, regulares ou irregulares. No próximo capítulo, veremos como estes conjuntos são formados, como podem ser manipulados (operações de shift e rotate), as atribuições com constantes paralelas, o ajuste necessário para os índices paralelos que não iniciem em 0 e as temporárias paralelas.

IV.1 Tratamento de Expressões

As regras gramaticais de Actus envolvendo expressões são utilizadas também, para definir as regras de precedência entre os vários operadores lógicos, relacionais e aritméticos. Se nos abstrairmos desta relação entre os operadores, as regras grama-

ticais envolvendo expressões e atribuições serão:

(1)	*expr*	⟶	$termo_1$ *op_bin* $termo_2$
(2)		|	*(op_un termo)*
(3)	*termo*	⟶	*var*
(4)		|	*const*
(5)		|	*expr*
(6)	*var*	⟶	*iden*
(7)		|	*var* '[' *exprs* ']'
(8)		|	*var* '.' *iden*

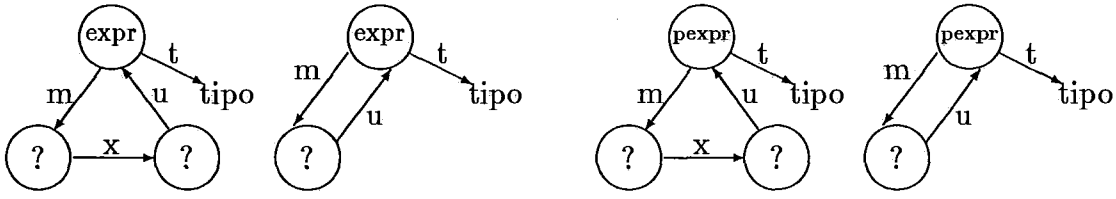


Figura IV.1: Expressões binárias e unárias, escalares e paralelas

As sub-árvores que representam as expressões, podem ter como raiz um nodo *expr* ou *pexpr* indicando tratar-se de uma operação escalar ou paralela. A Figura 1 apresenta as construções possíveis destas sub-árvores, onde o nodo **tipo** representa o tipo do resultado parcial da expressão. Os nodos com o sinal "?" representam os termos da expressão e podem ser nodos *var*, *const*, *expr* e *pexpr*. O operador é um dos atributos do nodo raiz, operadores unários e binários corresponderão a sub-árvores com um ou dois filhos (termos). O outro atributo do nodo raiz identifica a temporária que receberá o resultado parcial da operação.

As regras gramaticais envolvendo o comando de atribuição é apresentada abaixo, a sub-árvore correspondente a este comando é mostrada na Figura 2.

atrib_stat ⟶ *var* ':=' *expr*

O nodo *atrib* não possui atributos e pode ser usado em operações escalares e paralelas. Seu nodo **tipo** é normalmente nulo, exceto para atribuição de

constantes paralelas regulares.

Figura IV.2: Exemplo de sub-árvores de comandos de atribuição escalar e paralelo

A LI adota para o tratamento de expressões uma codificação de três endereços, com a seguinte construção:

`EXP`; *operação*; *resultado*; *operando1*; *operando2*;
`EXPP`; *operação*; *resultado*; *operando1*; *operando2*; *mascara*;

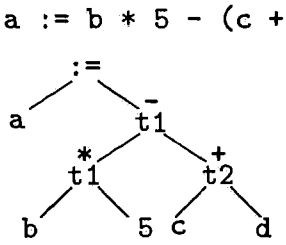
onde `EXP` indica uma operação escalar e `EXPP` indica uma operação paralela. O campo *máscara* só pode ser aplicado às operações paralelas e é opcional, como veremos mais adiante. O campo *operação* indica qual a operação lógica, aritmética ou relacional será executada aos operandos 1 e 2, e o campo *resultado* contém o nome da variável que receberá o resultado da operação.

Os campos *operando1* e *operando2* podem conter constantes inteiras ou reais, variáveis simples, variáveis indexadas e variáveis paralelas. Assim como o campo *resultado*, estes campos estão divididos em dois sub-campos, o primeiro determinando o tipo da informação contido no segundo. A regra de formação destes sub-campos é a seguinte:

`0`; *constante*
`1`; *variavel*
`2`; *variavel indexada*
`3`; *variavel indexada paralela vii*
`4`; *variavel indexada paralela vdi*
`5`; *constante paralela*

A Figura 3 apresenta um comando de atribuição em Actus, as sub-árvores construídas e o código LI correspondente. Nesta figura o nodo marcado com

':=' corresponde a um nodo *atrib*, e os nodos com os sinais '+', '-' e '*' correspondem a nodos *expr*'s.




```
(1) EXP;MULT; 1;t1; 1;B; 0;5;
(2) EXP;ADD; 1;t2; 1;C; 1;D;
(3) EXP;SUB; 1;t1; 1;t1; 1;t2;
(4) EXP;COPY; 1;A; 1;t1; 1;t2;

(5) EXP;SUB; 1;A; 1;t1; 1;t2;
```

Figura IV.3: Representação de uma expressão em Actus, na árvore e na LI.

Nesta figura o código LI gerado diretamente da sub-árvore corresponde à sequência 1,2,3,4. Contudo, a operação da linha 3 é eliminada durante a montagem do nodo *atrib*, quando a temporária de seu filho à direita é anulada (seu campo recebe o valor 0) e durante a geração de código a linha 5 é gerada substituindo as linhas 3 e 4.

Quando um nodo*expr* é criado pelos algoritmos que atendem às regras gramaticais 1 e 2, as temporárias (se) utilizadas nos nodos filhos são liberadas, sendo então requisitada a temporária do nodo raiz (as temporárias são reaproveitadas).

IV.2 Os Termos das Expressões

Para poderem ser usados nas expressões os termos devem ser tratados, isto é, transformados em sub-árvores que contenham as informações necessárias à posterior geração do código LI. Como pode ser visto pela construção das regras gramaticais 3 a 8, os termos podem ser:

1. O resultado parcial de outra operação.
2. Uma constante (inteira, real ou booleana).
3. Um identificador (definindo uma variável ou constante).
4. Uma variável indexada.

5. Uma variável do tipo record.

Quando o termo é outra expressão, sua sub-árvore já estará construída. Se o termo for uma constante, será representado por um nodo *const*, ligado a um string ASCII com o valor correspondente.

Se o termo for um identificador, sua definição é procurada na tabela de símbolos, onde deve corresponder a uma constante ou variável. No primeiro caso utiliza-se como termo um nodo *const* como o descrito anteriormente. No segundo caso constrói-se um nodo *var*.

O nodo *var* pode representar uma variável simples, uma variável indexada escalar, uma variável indexada paralela, ou uma variável definida por um dos campos de um record. Os records podem conter variáveis indexadas e variáveis indexadas podem ser formadas por records.

Para poder ser usado em uma expressão, o nodo *var* deve corresponder a uma variável simples que, se for indexada ou for do tipo record, deve ter sido completamente tratada pelas regras 7 e 8, de forma a estar definindo apenas um elemento da estrutura. A sub-árvore *var* é apresentada na Figura 4.

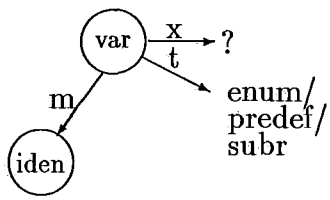


Figura IV.4: Formação básica de uma sub-árvore *var*

Na construção desta sub-árvore faz-se o nodo *var* apontar para o nodo *iden* da sub-árvore *vardecl* criado anteriormente durante a declaração da variável. O nodo *vardecl* é acessado através da tabela de símbolos. O ponteiro *tipo* do nodo *var* aponta para o tipo da variável e o ponteiro *prox* é usado na formação da sub-árvore *expr* a que pertence.

O atributo do nodo *var* é o mesmo do nodo *vardecl* e define o *offset* da variável na LI, ou seja, sua posição entre as variáveis declaradas dentro do seu

bloco. O bloco onde foi declarada a variável é caracterizado pelo atributo do nodo *iden*.

As variáveis são declaradas na LI utilizando-se um algoritmo bastante simples que gera os nomes a partir do nível do bloco, do offset da variável e do seu tipo. Por isso estas informações devem estar presentes (ou poderem ser acessadas através) no nodo *var*.

IV.3 Variáveis do Tipo Array e Record

O tipo record não é reconhecido pela LI, por isto as variáveis Actus do tipo record são decompostas em tantas variáveis simples ou indexadas quantas forem as que estiverem contidas no tipo record. Na formação do nodo *var*, a identificação de qual das variáveis do tipo record estamos tratando é feita simplesmente pelo seu atributo *offset*.

As regras gramaticais 6, 7 e 8, responsáveis pela construção das variáveis indexadas e do tipo record estão repetidas abaixo,

(6)	*var*	$\longrightarrow$	*iden*
(7)		|	*var* '[' *exprs* ']'
(8)		|	*var* '.' *iden*

e comandam os seguintes processos.

(regra 6) Procurar identificador na tabela de símbolos.

1. Procurar na tabela de símbolos a sub-árvore de declaração do identificador (um nodo *iden* que representando o não terminal *iden*). Se o identificador não for encontrado ou se a sub-árvore acessada não corresponder a uma declaração de tipo será gerada uma situação de erro.

2. Se a sub-árvore for *vardecl* (os demais casos são tratados em outras seções) criar uma sub-árvore *var*, copiar o *offset* do nodo *vardecl* e o conteúdo de seus

ponteiros *prim* e *tipo*.

Se o tipo da variável for primitivo, subrange ou enumerado, o nodo *var* estará pronto para ser utilizado como termo de uma expressão. Se o tipo for array ou record deverá ser posteriormente processado pelas regras 7 e 8.

(regra 7) Tratar variável indexada.

1. Verificar se o nodo *var* é do tipo array.
2. Verificar se os índices atuais definidos pelo não terminal *exprs* estão de acordo com a dimensão definida para aquele tipo array[1].
3. Para cada índice, se necessário, criar uma sub-árvore *expr* para ajustar o índice.
4. Ligar o nodo *var* aos seus índices, fazer com que o novo tipo de *var* seja o tipo do array. A Figura 5 mostra como fica o nodo *var* correspondente a uma variável indexada. Se o tipo for array de array, os índices do novo array são concatenados aos índices já existentes na sub-árvore *exprs*.

(regra 8) Tratar variável do tipo record.

1. Verificar se o nodo *var* é do tipo record.
2. Percorrer os nodos *field* do tipo record até encontrar um cujo nome coincida com o nome do nodo *iden*. Para cada nodo visitado sem encontrar o nome desejado incrementar o *offset* da variável.
3. Encontrado o nodo *field*, fazer com que o tipo do nodo *var* passe a ser o mesmo do nodo *field*.

IV.3.1 Variáveis Indexadas

Quando a sub-árvore *var* é finalmente utilizada como termo em uma expressão, se for uma variável indexada (Figura 5–a) a sua configuração será alterada para que

[1]Cada índice é representado por uma sub-árvore e deve ser um termo válido

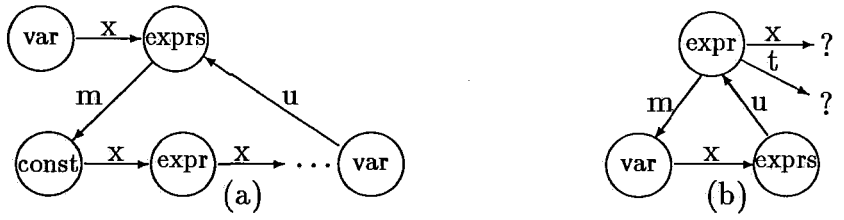


Figura IV.5: Formação da sub-árvore de uma variável indexada

possa ser mais facilmente interligada ao outro termo da expressão ou ao nodo pai. Esta transformação consiste em criar um nodo *expr* e ligá-lo aos nodos *var* e *exprs*, note que esta operação só será feita se o tipo da variável for um tipo simples. A Figura 5b mostra a sub-árvore tranformada. As temporárias utilizadas nos índices (nodos *exprs*) são liberadas no momento em que a transformação é feita.

Como na LI todos os arrays são sempre indexados a partir de 0, é necessário fazer-se uma correção em cada índice da variável indexada.

```
VAR
  A : array [5:10, 1:15, -2:7] of REAL;
...
  B := A[7, D, E*F];              {B := A[2, D-1, E*F+2];}
```

O trecho de programa Actus acima apresenta um comando de atribuição utilizando uma variável indexada, o comentário mostra como o comando é representado na árvore sintática e posteriormente traduzido para a LI. A seguir apresentamos o código LI gerado para este comando de atribuição. Note que o ajuste da primeira dimensão foi feita diretamente pelo compilador porque os valores eram conhecidos.

```
EXP; MULT; 1;T1; 1;E;  1;F;                          -- {temp1 := e * f}
EXP; SUB;  1;T2; 1;D;  0;1;                          -- {temp2 := d - 1}
EXP; ADD;  1;T1; 1;T1; 0;2;                          -- {temp1 := temp1 + 2}
EXP; COPY; 1;B;  2;A;3; 0;2; 1;T2; 1;T1; 0;0; -- { b := a[2,t2,t1]}
```

A notação da LI para as variáveis indexadas é a seguinte:

3; *nome_var*; *n_dim*; *índice1*; *índice2*; ...

onde *n_dim* indica a dimensão do array (que no exemplo acima é 3) e *índice_n* é formado por dois sub-campos similar aos utilizados para os *operando1* e *operando2*, mas onde não é permitido que possam aparecer novas variáveis indexadas. Quando ocorre que uma variável indexada possui como índice outra variável indexada, o valor da variável é atribuído à uma temporária que a substitui. Assim o trecho

```
B := A[10, C[E*D], C[4]];
```

é substituído pelo compilador para o seguinte trecho (sem considerar o ajuste de índices):

```
T1 := C[E*D];
T2 := C[4];
B  := A[10, T1, T2];
```

Os nodos *expr* quando representam variáveis indexadas (como na Figura 5) usados simplesmente para fazer a união dos nodos *var* e *exprs*, terão seus atributos $temp = 0$. Quando utilizados para também atribuir o valor da variável a uma temporária, o atributo *temp* terá o número que define a temporária.

IV.3.2 O Uso das Temporárias

Para controlar o uso das temporárias define-se dois contadores por tipo primitivo, o primeiro com o número de temporárias ocupadas no momento (*uso*) e o segundo com o número máximo de temporárias já utilizadas simultaneamente (*total*). Ao iniciar um bloco de subrotina, ambos os contadores devem estar com o valor 0.

As primitivas oferecidas para a operação com temporárias são:

1. Solicitar uma temporária.

2. Liberar um temporária.

em ambos os casos o tipo da variável deve ser fornecido como parâmetro.

O procedimento é o seguinte:

1. *uso* := *uso* + 1;
   Se *uso* > *total* então *total* := *uso*.
   Retornar com valor de *uso*[2].

2. *uso* := *uso* − 1;

Quando o bloco é fechado todos os contadores *uso* devem ter o valor 0, sendo o valor de cada *total* usado para definir quantas variáveis devem ser declaradas para o tipo correspondente (a declaração das temporárias é feita por nodos *tempdecl*).

A função que atende a solicitação de temporárias é utilizada também para determinar o número de linhas de código LI utlizados nas expressões, isto pode ser feito porque cada temporária requisitada corresponde sempre a uma e apenas uma linha de código LI. Para os demais nodos a contagem é feita durante a criação dos mesmos.

IV.3.3 A Extensão de Paralelismo

As expressões e atribuições envolvendo dados paralelos só são possíveis dentro de um bloco (USING) para o qual é definida a extensão de paralelismo (edp) válida para todas as operações internas ao bloco.

A edp vigente no momento em que uma expressão é avaliada está no topo da pilha de edp's. Esta pilha é controlada pelo comando USING e será vista posteriormente.

Conceitualmente a edp de um comando pode ser unidimensional ou bidimensional, mas na representação adotada para a árvore sintática as edp's unidimensionais correspondem à uma sub-árvore *edp* e as bidimensionais correspondem

[2]O valor retornado é usado como atributo do nodo *expr*, sendo utilizado para dar nome a temporária no código LI

à duas dessas sub-árvores. A edp é colocada no topo da pilha na forma de uma sub-árvore *config*, que terá um ou dois nodos *edp*'s como filhos. Estas sub-árvores são mostradas na Figura 6.

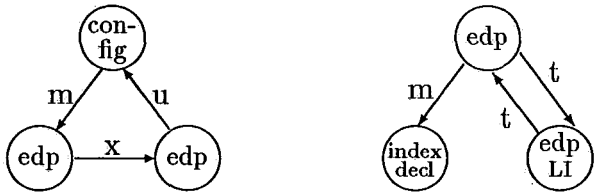


Figura IV.6: Sub-árvores que definem a edp dos comandos paralelos

Como vimos no capítulo anterior, os conjuntos de índices são estruturas de dados utilizadas para declarar os índices que poderão ser usados para definir a edp dos comandos USING, conjuntos estes que podem ser regulares ou irregulares. As sub-árvores *edp*'s são ligadas aos nodos *indexdecl* que são utilizados para definir os conjuntos de índices.

Todas as expressões paralelas devem estar coerentes com a edp definida pelo nodo *config* que estiver no topo da pilha. As temporárias utilizadas assumem implicitamente esta edp.

Na LI as edp's serão todas concentradas em uma tabela, representada na árvore sintática pela lista de nodos *edpLI*. Quando um conjunto de índices é declarado, um nodo *edpLI* é criado e interligado a este conjunto. Posteriormente quando os nodos *edp*'s são criados a interligação é desfeita, sendo o nodo *edpLI* interligado ao nodo *edp*.

A referência às edp's na LI é feita através do número da linha na tabela de edp's onde a edp (isto é o conjunto de índices) foi declarado, o número desta linha é atributo do nodo *edpLI*.

Para isso os conjuntos de índices, quando declarados, são ligados a nodos *edpLI*. Quando estes índices forem usados na definicão da edp de um bloco de comandos, os nodos *edp*'s são utilizados para determinar quais os conjuntos de índices serão usados, e a correspondente entrada na tabela de edp's.

IV.3.4 Variáveis Paralelas

Quando uma variável paralela é utilizada em uma expressão ou comando paralelo, o paralelismo é explicitamente definido através do identificador do conjunto de índices que indexa uma ou mais dimensões da váriavel.

A semântica de Actus pede que os conjuntos de índices utilizados na indexação sejam os mesmos definidos na sub-árvore *config* ativa, e que os índices estejam na mesma ordem e número dos nodos *edp*'s. A seguir, apresentamos trechos de um programa Actus com um comando de atribuição paralelo e o código LI correspondente:

```
A[IS] := B[IS] + 10;
EXPP;ADD; 3;A;1;1;5; 3;B;1;1;5; 0;10; 0;0;
```

onde

`3;A;1;1;5` representa a variável `A[IS]`
`3;B;1;1;5;` representa a variável `B[IS]` e
`0;10;` representa a constante 10.

O valor 3 que antecede `A` e `B` indica que se trata de uma variável paralela, e o índice `1;5;` determina que a dimensão é indexada pela entrada número 5 da tabela de edp's. A expressão em LI termina por `0;0;` representando a máscara utilizada para restringir a edp. Neste exemplo a máscara é nula, representando uma máscara com todos os elementos TRUE.

As regras gramaticais envolvidas são as mesmas para o caso sequencial, sendo que o identificador de um (ou mais) índice corresponde a um nodo *indexdecl*. A sub-árvore criada para representar uma variável paralela é mostrada na Figura 7. Considera-se o índice regular.

Quando, entre os índices de uma variável indexada é encontrado um identificador, que a tabela de símbolos informa corresponder a um nodo *indexdecl*, o seguinte procedimento é adotado:

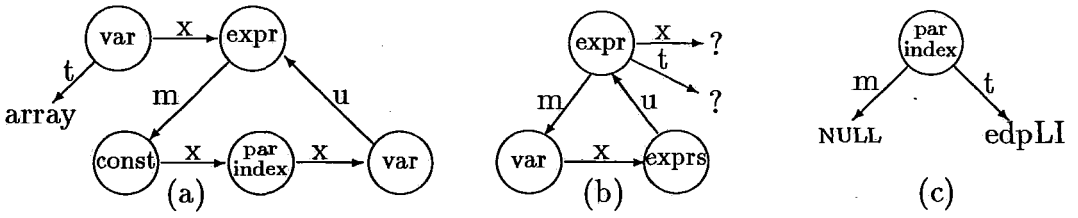


Figura IV.7: Sub-árvores de uma variável indexada e seu índice paralelo

1. Verificar se o nodo *indexdecl* pode ser utilizado, procurando entre os filhos da sub-árvore *config* um nodo *edp* que corresponda a este índice.

2. Substituir o índice por um nodo *parindex*.

3. Ligar o nodo *parindex* ao nodo *edpLI* do nodo *edp* correspondente.

IV.3.5 As Temporárias Paralelas

As expressões paralelas são formadas do mesmo modo como as escalares, tendo contudo algumas pequenas diferenças como pode ser observado na Figura 1, onde as sub-árvores escalares têm por raiz o nodo *expr* e as paralelas um nodo *pexpr*.

Se todas as variáveis paralelas envolvidas em uma expressão estiverem semanticamente corretas com relação a edp, os demais termos da expressão também estarão. Todas as temporárias utilizadas assumem automaticamente a edp vigente.

Como se trata de uma expressão paralela, as temporárias devem ser declaradas como sendo do tipo array, tendo a dimensão e número de elementos definidos pela sub-árvore *config*. Para cada tipo de temporária paralela utilizada sob a influência desta sub-árvore, será declarado um array com os dados da sub-árvore *config*.

As primitivas utilizadas para requisitar e liberar as temporárias paralelas são idênticas as das temporárias escalares, com a diferença de que: nas temporárias escalares uma única estrutura de dados é suficiente para atender completamente ao programa, não importando quantas sub-rotinas existem nem como estão aninhadas.

Para as temporárias paralelas é necessário criar uma pilha, onde sempre que um comando USING é aberto uma nova estrutura é criada e colocada no topo da pilha. Quando o comando USING for fechado a estrutura no topo da pilha será retirada. A cada tipo primitivo corresponde um ponteiro *p_array* que deve indicar o array a ser declarado para definir o tipo da temporária. Quando a estrutura de dados é colocada no topo da pilha, todos os campos *uso* e *total* são inicializados com 0 e os ponteiros *p_array* com o valor nulo.

Quando a estrutura é retirada do topo da pilha, todos os campos *uso* devem estar com 0 e os campos *total*, que não estiverem com 0, são utilizados para declarar tantas temporárias quantas forem o valor do campo, com o tipo apontado pelo ponteiro *p_array/* correspondente. Todos os campos *total* com 0 terão o ponteiro *p_array* correspondente com o valor nulo.

Quando uma temporária paralela é solicitada, se o ponteiro correspondente ao seu tipo for nulo, será criada uma sub-árvore *array* de acordo com a sub-árvore *using*, tendo por tipo o mesmo da temporária. O nodo *array* será inserido na lista de nodos *arrayLI* e o ponteiro *p_array* apontará para o nodo *array*. Assim um mesmo nodo *array* serve como tipo para todas as temporárias paralelas relativas ao mesmo tipo primitivo. A Figura 8 apresenta uma sub-árvore *tempdecl* montada a partir dos dados lidos nos camps *total* e *p_array*. O atributo do nodo *tempdecl* é o valor do campo *total* e o nodo *array* que serve de tipo é aquele apontado pelo ponteiro *p_array*.

Figura IV.8: Sub-árvore *tempdecl* paralela

Outra particularidade no uso das temporárias paralelas é a necessidade de requisitar uma temporária que não tenha sido usada anteriormente. Para explicar melhor este caso é necessário ver como funcionam as máscaras para as expressões paralelas.

IV.3.6 O Uso de Temporárias em Máscaras

Como foi mencionado na discussão sobre variáveis paralelas, para a LI as expressões paralelas são executadas segundo um edp global (definida pelo comando USING) que pode ser restringida por um vetor ou grade com valores TRUE ou FALSE, e que surgem como uma quarta variável nas expressões da LI.

Se o campo correspondente a máscara contiver os valores `0;0;` a LI não utiliza nenhuma máscara na execução da expressão. Caso contrário o campo deve ter a seguinte formação:

```
3;mascara;1; 1;n;          {unidimensional ou}
3;mascara;2; 1;k; 1;m;     {bidimensional}
```

onde `mascara` corresponde ao nome de uma variável do tipo array booleano, provavelmente uma temporária paralela, e `n`, `k` e `m` são as entradas na tabela de edp's onde os valores daquela dimensão estão descritos.

As máscaras são usadas na implementação dos comandos seletivos e de repetição, para definir os elementos que sofrerão a ação dos comandos internos ao comando seletivo ou de repetição. A variável a ser utilzada como máscara é definida durante a avaliação da expressão de controle dos comandos paralelos.

Na geração de código relativo a um comando seletivo (ou de repetição) o compilador coloca no topo da pilha de máscaras o ponteiro para o nodo *pexpr*, que define a temporária ou variável a ser utilizada. Esta máscara será utilizada para todas as expressões e comandos de atribuição contidos no comando. Seu valor deve ser o definido durante a avaliação da expressão de controle e permanecer inalterado até que a expressão de controle seja eventualmente reavaliada, ou um novo comando seja aberto, com sua máscara passando a assumir o topo da pilha, ou ainda até que o comando original termine retirando a máscara do topo da pilha.

Quando utilizada em um comando REPEAT como no caso abaixo pode ocorrer um problema se não forem tomadas as devidas precauções:

```
REPEAT
   ...
UNTIL A[IS] > 1;
```

Como as temporárias utilizadas dentro do comando REPEAT já foram todas liberadas quando a expressão `A[IS] > 1` é avaliada, é possivel que a temporária paralela a receber o resultado da expressão tenha sido utilizada dentro do comando REPEAT. Se isto ocorrer, sempre que a cada nova iteração os comandos internos ao REPEAT forem executados, e a expressão que utiliza a mesma temporária de controle for reavaliada, o valor da máscara será corrompido.

Por isto, a temporária usada como máscara não deve ter sido utilizada anteriormente, sendo fornecida pela primitiva "solicitar temporária virgem" executada pelo seguinte procedimento:

- *uso* := *uso* + 1;
  total := *total* + 1;
  Retornar com valor de *total*.

A liberação da temporária é feita pelo modo usual.

Capítulo V

Índices Paralelos

Foi visto nos capítulos anteriores como os índices paralelos (conjuntos de índices) são declarados, como são construídas suas sub-árvores, como são representados na tabela de edp's e como são utilizados na indexação das variáveis e temporárias nas expressões e comandos de atribuição paralelos. Neste capítulo veremos:

- Como os índices são manipulados para construir outros conjuntos.
- As peculiaridades dos índices irregulares e porque devem ser representados por constantes paralelas.
- A utilização dos índices indefinidos.
- As operações de alinhamento dos índices através dos operadores shift e rotate.
- Como são feitos os ajustes dos índices paralelos quando o array foi declarado com o índice iniciando em um valor diferente de zero.

V.1 Operações com Conjuntos

Quando um conjunto é formado da união, interseção ou diferença de outros dois conjuntos, este conjunto será considerado irregular porque para a LI existem apenas dois tipos de conjuntos de índice: regulares e irregulares. Por isto o novo conjunto será sempre irregular, sendo representado por um nodo *vetores*.

O algoritmo utilizado para as operações de uniao, interseção e diferença é apresentado a seguir. Note que os conjuntos são sempre representados em ordem crescente e na forma de uma lista de inteiros, onde a primeira posição (posição 0) indica quantos elementos estão contidos na lista.

V.1.1 União

A operação de união é feita em Actus como apresentada no exemplo abaixo:

```
INDEX
  IS: 1:33 + 3:[4]55;
```

O algoritmo que trata desta operação recebe os dois conjuntos como parâmetros, e transforma-os em uma lista representando o novo conjunto. Este algoritmo forma a lista copiando os elementos dos conjuntos 1 e 2, descartando um dos elementos quando houver repetição. Esta é uma operação comutativa.

```
UNIÃO(c1, c2)
    i:=1; j:=1; k:=1;
    repetir para todos os elementos de c1 e c2
        lista[k] := c1[i];
        se c1[i] = c2[j]
            j:=j+1; {descartar este elemento de c2}
        i:=i+1; k:=k+1;
        se c1[i] > c2[j]
            swap(c1, c2);
            swap(i, j);
    repetir enquanto c1[0] >= i
        lista[k]:=c1[i];
        k:=k+1; i:=i+1;
    repetir enquanto c2[0] >= j
        lista[k]:=c2[j];
        k:=k+1; j:=j+1;
```

```
lista[0]:=k-1;
```

V.1.2 Diferença

A operação de diferença é feita em Actus como apresentada no exemplo abaixo.

```
INDEX
  IS: 1:33 - 3:[4]55;
```

O algoritmo que trata desta operação recebe como parâmetros os dois conjuntos e transforma-os em uma lista representando o novo conjunto. Este algoritmo forma a lista com os elementos do conjunto 1 que não se repetirem no conjunto 2. Esta não é uma operação comutativa.

```
DIFERENÇA(c1, c2)
    i:=1; j:=1; k:=1;
    repetir para todos os elementos de c1 e c2
        se c1[i] = c2[j]
            i:=i+1; j:=j+1; {nao copiar este elemento}
        caso contrario
            lista[k] := c1[i];
            i:=i+1; k:=k+1;
        repetir enquanto c1[i] > c2[j]
            j:=j+1;
    repetir enquanto c1[0] >= i
        lista[k]:=c1[i];
        k:=k+1; i:=i+1;
    lista[0]:=k-1;
```

V.1.3 Interseção

A operação de interseção é feita em Actus como apresentada no exemplo abaixo.

```
INDEX
  IS: 1:33 * 3:[4]55;
```

O algoritmo que trata desta operação recebe como parâmetros os dois conjuntos e transforma-os em uma lista representando o novo conjunto. Este algoritmo forma a lista com os elementos do conjunto 1 que se repetirem no conjunto 2. Esta é uma operação comutativa.

```
INTERSEÇÃO(c1, c2)
    i:=1; j:=1; k:=1;
    repetir para todos os elementos de c1 e c2
        se c1[i] = c2[j]
            lista[k]:=c1[i];
            i:=i+1; j:=j+1; k:=k+1;
        caso contrario
            i:=i+1;
        se c1[i] > c2[j]
            swap(c1, c2);
            swap(i, j);
    lista[0]:=k-1;
```

V.2 Constantes Paralelas

As constantes paralelas não podem ser usadas em expressões, sendo seu uso permitido apenas na atribuição de seus valores a variáveis paralelas. A edp vigente deve ser unidimensional porque todas as constantes paralelas também o são. As constantes paralelas regulares são colocadas na tabela de edp's e as irregulares devem ser tratadas por um comando (da LI) específico.

O mesmo tratamento será utilizado para os índices irregulares, porque a LI só pode tratar destes índices de forma indireta, como constantes paralelas irregulares atribuídas a um array que será utilizado no lugar do índice original.

Abaixo apresentamos o exemplo de um trecho de programa Actus, onde são feitas atribuições com os dois tipos de constantes. As sub-árvores construídas para representar este trecho de programa na árvore sintática e o código LI correspondente são também apresentados.

```
PARCONST
  REG := 1:[3]1000;                {regular}
  IRG := 1:[3]1014 - 445:455;      {irregular}
INDEX
  IS := 1:334;
  ...
  A[IS] := REG;
  B[IS] := IRG;
```

Para o exemplo acima REG será representada por um nodo *step*, que será colocado na lista de nodos *edpLI* e ocupará uma entrada na tabela de edp's. A constante IRG será representada por um nodo *vetores*.

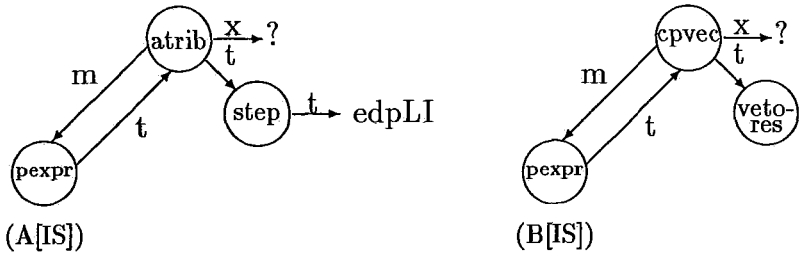


Figura V.1: Constantes paralelas sendo atribuídas à variáveis

Os comandos de atribuição são representados pelas sub-árvores mostradas na Figura 1, onde o nodo *cpvec* faz a atribuição da constante irregular e os nodos *pexpr*, que representam as variáveis, são formados como mostrada na Figura 7 do capítulo anterior. O código LI correspondente, supondo que o índice IS ocupa a entrada 'k' na tabela de edp's e a constante REG ocupa a entrada 'n', é mostrado abaixo:

```
EXPP;COPY; 3;A;1;1;k; 5;n; 0;0;          --{A[IS]:=REG}
CPVEC;B;k;18;                            --{B[IS]:=IRG}
```

```
CPVAL;25; 1;4; ... ;73;76;
  ...
CPVAL;9; 988;991; ... ;1009;1012;
```

onde a atribuição da constante regular REG ocupa apenas uma linha de código, e a atribuição constante irregular IRG ocupa 19 linhas.

A instrução CPVEC informa que as 18 instruções CPVAL que se seguem devem atribuir seus valores à variável B indexada pela edp da entrada 'k'. Cada instrução CPVAL indica quantos valores contém, listando-os em seguida. Neste exemplo cada CPVAL contém no máximo 25 elementos.

V.3 Índices Irregulares

Os índices irregulares, como as constantes irregulares, não podem ser colocados na tabela de edp's, porque a notação adotada pela LI só reconhece os índices e constantes regulares.

Por esta razão, sempre que um índice irregular for usado em um comando USING, o compilador deve gerar sub-árvores para um programa fictício, que possa ser traduzido para a LI. Assim o trecho abaixo:

```
USING  IR := 1:[3]1014 - 445:455 DO
  A[IR] := B[IR];
```

será convertido no sequinte trecho

```
PARCONST
  IR := 1:[3]1014 - 445:455;                {1}
INDEX
  IF := 1:334;                              {2}
VAR
  AF : array [1:334] of INTEGER;            {4}
USING IF DO                                 {3}
```

```
A[IF] := IR;                                    {5}
A[AF[IF]] := B[AF[IF]];                         {6}
```

As alterações consistem em:

1. Tratar o índice irregular como uma constante irregular.
2. Declarar um novo índice regular com o mesmo número de elementos do anterior.
3. Usar este novo índice no comando USING substituindo o índice irregular.
4. Declarar uma variável do tipo array paralelo unidimensional, com tantos elementos quanto o índice irregular.
5. Inserir como primeiro comando do USING a atribuição dos valores do índice irregular à variável recém criada.
6. Alterar todas as indexações feitas originalmente com o índice irregular por indexações indiretas, utilizando como índice a variável criada em 4, indexada pelo índice regular criado em 2.

As construções que representam estas operações na árvore sintática são mais complexas, com interligações entre as várias sub-árvores envolvidas representando as relações existentes entre elas, como foi descrito acima.

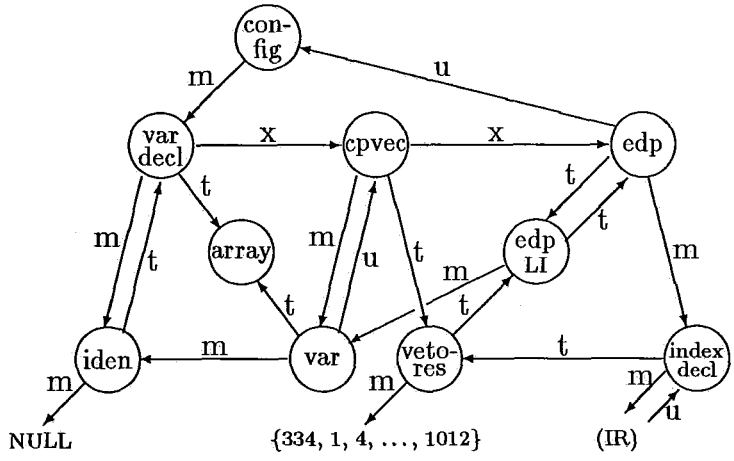


Figura V.2: Sub-árvores que permitem o uso de índices irregulares

A Figura 2 apresenta uma parte das sub-árvores construídas pelo compilador para transformar uma edp irregular em regular. A declaração da variável é feita junto com as temporárias paralelas do comando USING e a instrução CPVEC será a primeira a ser executada dentro do comando USING.

A sub-árvore *config* tem a forma descrita na Figura 2, somente durante a avaliação dos comandos internos ao USING. Quando a sub-árvore que representa o USING na árvore sintática for formada, após a avaliação de todos seus comandos internos, os nodos *vardecl* e *cpvec* são retirados da sub-árvore *config* e colocados na posição correta junto com as declarações de temporárias, a frente do primeiro comando do USING.

A LI exige ainda, que cada indexação indireta possua uma entrada própria na tabela de edp's. Assim o nodo *edpLI* da Figura 2 representa duas entradas na tabela, a primeira correspondendo ao conjunto regular criado pelo compilador, e a segunda correspondendo à variável que será utilizada como índice indireto. Abaixo apresentamos as duas entradas na tabela de edp's e o código LI relativo ao trecho em Actus acima.

```
  (n)  0;1;0;1;0;334;
(n+1)  1;0;1;AF;0;334;

  CONFIG;1;n;
    CPVEC;AF;n;18;
      CPVAL; ...
      ...
    EXPP;COPY; 3;1;1;n+1; 3;B;1;1;n+1; 0;0; 0;0;
```

No código LI acima não é mostrada a declaração da variável AF nem a entrada na tabela de arrays relativa ao tipo de **AF**.

As linhas n e $n+1$ são geradas quando o nodo *arrayLI* é visitado, porque este nodo estará ligado (indiretamente) aos nodos *vetores* e *var*.

A linha de código LI responsável pela atribuição

```
A[AF[IF]] := B[AF[IF]];
```

é gerada utilizando a entrada $n+1$ para indexar os arrays A e B, porque os nodos *parindex* destas variáveis estarão ligados ao nodo *edpLI*, que gera as linhas n e $n+1$ (ver Figura 7 do capítulo anterior), e neste caso sempre será utilizada a segunda entrada reservada pelo nodo *arrayLI*.

V.4 Indexação Indireta

A indexação indireta (paralela) pode também ser feita por intervenção direta do programador ao utilizar um array paralelo como índice na indexação de outro array paralelo.

A indexação indireta é tratada de modo semelhante ao descrito na seção anterior. A variável ou expressão paralela utilizada como índice deve ser representada na tabela de edp's com o número desta entrada substituindo o índice original, de maneira similar ao utilizado para indexar as variáveis A e B.

Este modo de tratar a indexação indireta, impede em comandos USING bidimensionais, o uso deste tipo de indexação. Abaixo apresentamos variáveis paralelas indexadas de modo ilegal.

```
A[B[IS,JS]]
C[D[IS],JS]
```

As indexações indiretas são representadas na árvore sintática pelas sub-arvores mostradas na Figura 3–a. A Figura 3–b mostra como é formada a sub-árvore no caso de indexação direta para contrastar com o caso indireto.

Quando o nodo *edpLI* estiver ligado, através de seu ponteiro *prim*, a um nodo nulo, a entrada a ser gerada na tabela de edp's corresponderá a uma indexação direta. Quando *prim* estiver apontando para um nodo *var* ou *pexpr*, a entrada na tabela corresponderá uma indexação indireta. Assim a atribuição abaixo é válida,

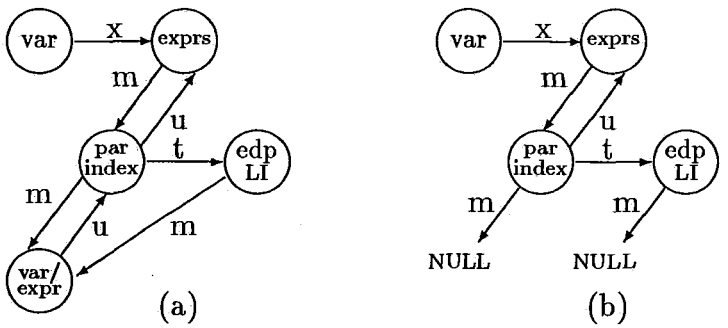


Figura V.3: Indexações de variáveis paralelas

```
USING IS DO
  A[C[IS ]+B[IS]+7, D*E] := 1;
```

gerando o sequinte código LI (máscara não é mostrada),

```
EXPP;ADD; 3;PTEMP1;1;1;n; 3;C;1;1;n; 3;B;1;1;n;    {pt1[]:=c[]+b[]}
EXPP;ADD;3;PTEMP1;1;1;n; 3;PTEMP;1;1;n; 0;7;       {pt1[]:=pt1[]+7}
EXP;MULT;1;TEMP1; 1;D; 1;E;                        {t1:=d*e}
EXPP;COPY;3;A;2;1;n+1;1;TEMP1; 0;1; 0;0;           {a[pt1[],t1]:=1}
```

onde supomos que a entrada n da tabela de edp's contenha a descrição do índice `IS` e a entrada $n+1$ tenha sido criada para indicar que `PTEMP1` será usada como índice indireto.

Para não complicar desnecessáriamente as operações feitas pelo compilador, não é permitido nesta versão o aninhamento de índexações indiretas paralelas. Assim o seguinte comando não é válido:

```
A[B[C[IS]]] := 1;
```

embora seja aceito o mesmo comando escrito da seguinte forma:

```
D := B[C[IS]];
A[D[IS]] := 1;
```

V.5 Índices Indefinidos

Os índices irregulares devem ter seus valores conhecidos em tempo de compilação, não sendo aceitos pelo compilador se tal não acontecer. A mesma limitação não é feita aos índices regulares porque a LI permite que os parâmetros utilizados na sua definição possam ser calculados em tempo de execução.

Os parâmetros utilizados pela LI na definição dos conjuntos regulares são:

início; espaçamento; tamanho;

qualquer um podendo ser representado por uma constante ou variável.

Como em Actus os parâmetros que definem estes conjuntos (ver Figura 13 no capítulo anterior) são:

início; espaçamento; fim;

é necessário que o compilador calcule o tamanho do conjunto através da expressão:

tamanho := ((**fim** − **início**) / **espaçamento**) + **fim**;

Isto pode ser feito em tempo de compilação, se os três valores forem conhecidos (representados por constantes). Caso contrário, uma expressão será gerada e o resultado atribuído a uma temporária. A sub-árvore *expr* que representar a operação acima, será coloca imediatamente antes de cada comando USING que utilizar o conjunto.

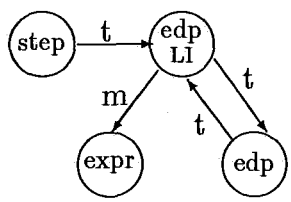


Figura V.4: Sub-árvore de edp a ser definida em tempo de execução

Para que isto seja possível esta sub-árvore é ligada ao nodo *edpLI* como mostrado na Figura 4. Esta solução é conveniente, porque o ponteiro *prim* não é utilizado (como nos casos de conjuntos irregulares) para outra função e o nodo *edpLI* é facilmente alcançado, a partir da sub-árvore *using* a que pertence. Caso o índice indefinido seja utilizado em vários comandos USING, todas as sub-árvores *edpLI* desses comandos apontarão para a mesma sub-árvore *expr*, gerando assim, onde necessário, as mesmas linhas de código LI, para cálculo dos parâmetros do índice.

O cálculo da expressão é necessário antes de cada um dos comandos USING, porque o valor de um (ou mais) dos termos da expressão (os parâmetros do índice) pode ter sido alterado após a última avaliação.

V.6 Shift Regular

O operador shif pode ser aplicado em conjuntos de índices regulares e irregulares, e tem a finalidade de deslocar seus valores para a direita ou para a esquerda. O efeito desejado é conseguido somando-se a cada elemento do conjunto o valor especificado pelo shift. Por exemplo:

```
A[IS shift 5]
```

Se o shift é aplicado a um conjunto regular, o deslocamento é feito criando-se uma nova entrada na tabela de edp's. Nesta nova entrada o valor de início do conjunto estará somado ao valor especificado para o shift (o valor pode ser positivo ou negativo).

Se o valor início não for uma constante, será necessário criar uma sub-árvore *expr* para calcular o novo valor de início, e usar a temporária definida no nodo *expr* para representar o parâmetro **início** na nova edp.

A Figura 5 mostra como é construída a sub-árvore relativa a um índice regular deslocado por um shift. Um nodo *edpLI* é criado para representar a nova edp criada pelo deslocamento. Quando acessado através da lista de nodos

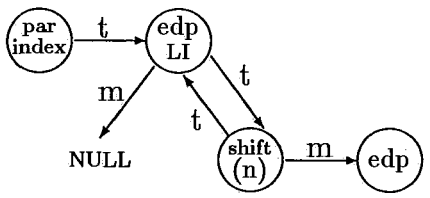


Figura V.5: Shift sobre um conjunto de índices regular

edpLI, a entrada na tabela de edp's será gerada aplicando à edp original o deslocamento definido pela sub-árvore *shift*.

A seguir apresentamos um trecho de programa em Actus, onde é definido um conjunto de índices e uma operação de atribuição envolvendo um índice deslocado .

```
USING  IS := 1:[3]1000 DO
  A[IS shift 5] := B[IS shift-4] - C[IS, 8];
```

As entradas geradas na tabela de edp's relativas a este trecho são definidas abaixo, onde a entrada n descreve os valores do índice IS. As entradas $n+k$ e $n+k+1$ especificam os valores de IS após os deslocamentos. Note que ambas as entradas apontam para a entrada original n da qual evoluiram.

n	`0;1; 0;3; 0;334;`	{ *definição de IS* }
$n+k$	`0;6; 0;3; 0;334;n;`	{ *IS shift 5* }
$n+k+1$	`0;-3; 0;3; 0;334;n;`	{ *IS shift-4* }

O código LI, correspondente ao comando de atribuição acima, é apresentado a seguir. Não se observa claramente os efeitos dos deslocamentos no código LI, porque (os deslocamentos) já foram executados na tabela de edp's, bastando agora fazer referência às entradas correspondentes.

```
EXPP;SUB; 3;A;1;1;n+k; 3;B;1;1;n+k+1; 3;C;2;1;n;0;8; 0;0;
```

Se o novo ínicio for calculado em tempo de execução, a expressão será definida pela sub-árvore *expr* ligada ao próprio nodo *edpLI*. A expressão será

transcrita para o código LI, quando a variável indexada que sofrer o shift for visitada antes de gerar o código LI relativo a própria variável.

Como foi visto na seção anterior, os nodos *edpLI* dos conjuntos indefinidos também estão ligados a uma sub-árvore *expr* através do ponteiro *prim*. Contudo, os dois caso são diferenciados, porque no caso anterior o nodo *edpLI* está ligado diretamente ao nodo *edp* e no caso discutido nesta seção um nodo *shift* é usado para intermediar os dois, permitindo que o gerador de código LI saiba como proceder quando estiver percorrendo a árvore sintática.

V.7 Shift Irregular

O operador shift, quando aplicado a um índice irregular, não pode ser tratado de modo semelhante ao visto na seção anterior, porque os índices irregulares não são definidos na tabela de edp's, sendo representados de forma indireta.

Quando um índice irregular é deslocado por um shift, é gerada uma nova sub-árvore *vetores* com uma cópia da lista de inteiros da sub-árvore original, mas somando-se a cada elemento o valor do shift. Esta é uma operação simples, mas aumenta o tempo de compilação e execução do programa, e do uso de memória, se a lista for muito grande.

A Figura 6 mostra a interligação entre o índice deslocado – sub-árvore *parindex*, o nodo que define a entrada do novo índice na tabela de edp's – *edpLI*, e a edp original – nodo *edp*.

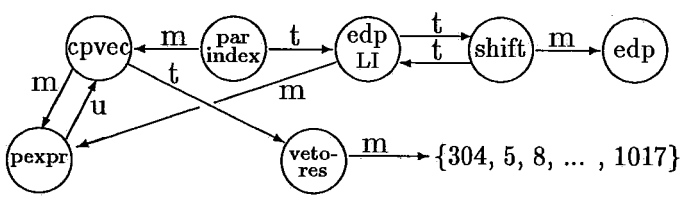


Figura V.6: Representação de um índice irregular deslocado por um shift

Alem da criação da sub-árvore *vetores* com os valores do novo índice, é necessário criar uma sub-árvore *cpvec* para atribuir estes valores a variável que será usada como índice indireto (neste caso uma temporária paralela), e criar o

nodo *edpLI* que reserva uma entrada na tabela de edp's.

A interligação entre eles é mostrada na Figura 6, e permite que na geração da tabela de edp's os dados possam ser acessados através do nodo *edpLI*. Na geração de código, quando o nodo *parindex* for visitado (ver Figura 7 do capítulo anterior), a instrução CPVEC será gerada antes da própria variável indexada.

A seguir, apresentamos um trecho de programa em Actus, onde é definido um conjunto de índices irregulares e uma operação de atribuição, onde o índice é deslocado.

```
USING IS := 1:[3]1014 - 445:455 DO
  A[IS shift 5] := B[IS shift-4] - C[IS, 8];
```

Como se trata de um conjunto irregular, o compilador deve gerar as sub-árvores correspondentes traduzindo não o trecho de programa acima, mas sim o trecho fictício apresentado a seguir (ver a seção Índices Irregulares para mais detalhes).

```
USING IF DO
  BEGIN
    AF[IF]   := IR;
    PT1[IF] := IR + 5;
    PT2[IF] := IR - 4;
    A[PT1[IF]] := B[PT2[IF]] - C[AF[IF], 8];
  END
```

As entradas na tabela de edp's são apresentadas abaixo, onde as entradas n e $n+1$ são geradas pelo comando USING, e as entradas $n+k$ e $n+k+1$ são geradas para indexar os arrays A e B através das temporárias paralelas PT1 e PT2. Estas entradas devem fazer referência a linha n para informar à LI a partir de quem foram geradas (ou de quem são dependentes na terminologia da LI).

```
n          0;1; 0;1; 0;334;          { definição de IF }
```

$n+1$	1;0; 1;AF; 0;334;	{ *definição de AF* }
$n+k$	1;0; 0;PT1; 0;334;n;	{ *IS shift 5* }
$n+k+1$	1;0; 0;PT2; 0;334;n;	{ *IS shift-4* }

O código LI gerado para a operação de atribuição é mostrado abaixo. Note que apenas as três últimas operações de atruibução do código fictício estão sendo mostradas, sendo que, duas dessas atribuições são realizadas por meio de instruções CPVEC, cada uma relacionando todos os elementos do conjunto.

```
CPVEC;PT1;n;18;
  CPVAL; ...
  ...
CPVEC;PT2;N;18;
  CPVAL; ...
  ...
EXPP;SUB; 3;A;1;1;n+k; 3;B;1;1;n+k+1; 3;C;2;1;n+1;0;8; 0;0;
```

V.8 Rotate

O operador rotate provoca um deslocamento circular no conjunto de índices para a direita, ou para a esquerda, como mostrado no exemplo abaixo.

```
IS := 1:[3]38;     --{10,13,16,19,22,25,28,31,34,47}
IS rotate 3        --{31,34,37,10,13,16,19,22,25,28}
IS rotate-2        --{16,19,22,25,28,31,34,37,10,13}
```

Ocorre que um índice regular após a operação de rotate, não pode mais ser representado na tabela de edp's. Por isso, o novo índice passa a ser considerado como um conjunto irregular, sendo representado por uma sub-árvore *vetores* e só pode ser utilizado como um índice indireto, sendo substituído pela variável que receber seus valores. A operação que atribui os valores do índice após o deslocamento, é representada por uma sub-árvore *cpvec* numa operação similar ao shift de índices irregulares.

Assim os conjuntos regulares e irregulares, quando deslocados por um rotate serão representados pelas mesmas estruturas na árvore sintática, e irão gerar o mesmo código LI, similar ao gerado para o shift irregular. A Figura 7 apresenta a sub-árvore do índice de uma variável paralela sob a ação de um rotate.

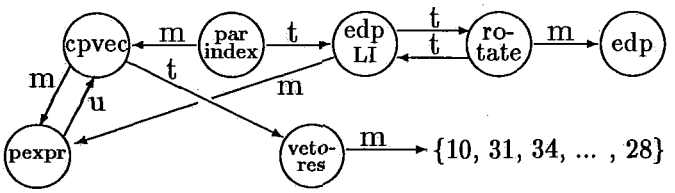


Figura V.7: Representação do índice deslocado por rotate

Apresentamos abaixo, trechos de um programa Actus, onde o índice regular é deslocado (o processo é o mesmo para índices irregulares), seguido do código fictício gerado pelo compilador, das entradas correspondentes na tabela de edp's e do código LI gerado. O processo é semelhante ao adotado para tratar os shifts irregulares.

```
USING IS := 10:[3]38 DO
  A[IS rotate 3] := B[IS rotate -2] - C[IS, 8];

{inicio do codigo ficticio}
USING IS := 10:[3]38 DO
    PT1[IS] := IS ->  3;        {IS rotate 3}
    PT2[IS] := IS -> -2;        {IS rotate-2}
    A[PT1[IS]] := B[PT2[IS]] - C[IS, 8];
END
{fim do codigo ficticio}
```

n	0;10; 0;3; 0;10;	{ *definição de IS* }
$n+k$	1;0; 0;PT1; 0;10;n;	{ *IS rotate 3* }
$n+k+1$	1;0; 0;PT2; 0;10;n;	{ *IS rotate-2* }

```
CPVEC;PT1;n;1;
  CPVAL;10; 31;34;37;10;13;16;19;22;25;28;
```

```
CPVEC;PT2;N;1;
  CPVAL;10; 16;19;22;25;28;31;34;37;10;13;
EXPP;SUB; 3;A;1;1;n+k; 3;B;1;1;n+k+1; 3;C;2;1;n;0;8; 0;0;
```

V.9 Ajuste de Índices Paralelos

Actus permite na declaração dos arrays, a definição do valor inicial e final dos índices. Para a LI os índices iniciam sempre em 0, gerando a necessidade de ajustar os índices de todas as variáveis indexadas, a menos que um determinado índice em particular tenha sido declarado iniciando em 0. O ajuste dos índices paralelos é feito de modo semelhante ao ajuste feito para os não paralelos, vistos anteriormente. Utilizaremos o trecho de programa abaixo para exemplificar o tratamento necessário ao ajuste dos índices paralelos:

```
VAR
  A: array [0:500] of REAL;
  B: array [100:100] of REAL;
  C: array [200:500] of REAL;
USING IS := 300:400 DO
  A[IS] := B[IS] + C[IS];
```

Para a LI as variáveis A, B e C são consideradas como se tivessem sido declaradas em Actus como:

```
VAR
  A': array [0:500] of REAL;
  B': array [0:400] of REAL;
  C': array [0:300] of REAL;
```

Deste modo o índice `IS` deve ser alterado para que a expressão envolvendo `B` e `C` seja executada como pretendido pelo programa original, como por exemplo:

```
A'[IS] := B'[IS'] + C'[IS"]
```

onde

```
IS' := 200:300;
IS" := 100:200;
```

o que pode ser feito com a operação de deslocamento shift. Assim o compilador irá gerar código LI para o seguinte comando de atribuição:

```
A[IS] := B[IS shift-100] + C[IS shift-200];
```

deslocando para a esquerda o índice de cada variável, tantas posições quanto for o valor inicial deste índice. Se o valor inicial for negativo o deslocamento será para a direita.

Este procedimento é válido também para índices irregulares. Assim, para o sequinte índice irregular:

```
USING IS := 300:320 - 305:315;
```

as variáveis A, B e C serão indexadas pelos seguintes valores:

```
(A) IS           = {300,301,302,303,304,316,317,318,319,320}
(B) IS shift-100 = {200,201,201,203,204,216,217,218,219,220}
(C) IS shift-200 = {100,101,101,103,104,116,117,118,119,120}
```

Quando o índice já está sofrendo um deslocamento por um shift, o compilador utiliza para o shift o valor original subtraído do valor inicial do índice. Para as operações de rotate, que representam seus índices através de uma lista de inteiros da sub-árvore *vetores*, o ajuste do índice é feito subtraindo-se o valor inicial do índice de cada um dos elementos da lista.

Capítulo VI

Comandos

Os comandos de Actus compreendem os comandos de Pascal, expandidos para expressar operações com estruturas de dados paralelas, mais especificamente, matrizes e vetores. A definição dos elementos da estrutura que serão acessados é feita por um ou dois conjuntos de índices, que denominamos a extensão de paralelismo (edp) do comando.

Actus não suporta paralelismo na estrutura de controle, sendo os comandos executados na sequência em que foram declarados. Como o paralelismo é expresso apenas nos dados, as operações paralelas que permitem aumentar a performance (na execução) do programa estão restritas às expressões e às atribuições.

Os comandos paralelos podem definir a edp a ser utilizada em um bloco de comandos, ou restringir (mascarar) a edp vigente através de um vetor booleano. A máscara terá tantos elementos quantos os definidos pela edp, sendo que somente serão acessados os elementos da edp que corresponderem, na máscara, ao valor TRUE.

O comando de atribuição já foi tratado nos capítulos ateriores, e o comando USING já foi visto parcialmente. Os demais comandos são os comandos seletivos, os comandos de repetição e as chamadas de procedimentos. Os comandos seletivos compreendem os comandos IF com e sem a cláusula ELSE, e o comando CASE. Os comandos de repetição compreendem o comando REPEAT, o comando WHILE e o comando FOR.

Os comandos de Actus podem assumir, em geral, a forma paralela ou a forma seqüencial. O comando USING não tem seu dual seqüencial, e o comando FOR e a chamada de funções e procedimentos não têm o dual paralelo.

Os comandos seletivos e repetitivos possuem uma variável de controle que determina como o comando será executado. Se esta variável[1] for paralela, o comando será paralelo, e se a variável for escalar, o comando será seqüencial.

A seguir discutiremos os comandos paralelos nos seus aspectos principais, sem nos determos no tratamento dos comandos seqüenciais.

VI.1 O Comando Using

O termo "comando" não é bem aplicado ao USING, que pode ser considerado mais como uma declaração, pois é usado para definir qual edp será usada no seu bloco de comandos. A edp vigente anteriormente a abertura do comando USING volta a ter efeito quando o bloco é fechado.

A principal razão para a existência do comando USING é a de incentivar o programador a manter estruturados os comandos que manipulam os dados paralelos. O comando USING pede que comandos com a mesma extensão de paralelismo sejam agrupados em um mesmo bloco, para o qual define a edp. Além de melhorar a inteligibilidade do programa, pretende-se aumentar a eficiência do código executável, ao permitir que uma mesma operação de set-up da máquina alvo possa seu utilizada para um conjunto maior de comandos.

As regras gramaticais deste comando são:

(1)	*using_stat*	$\longrightarrow$	*edps* 'DO' *stat*
(2)	*edps*	$\longrightarrow$	*edp* ',' *edp*
(3)		|	*edp*
(4)	*edp*	$\longrightarrow$	*iden* ':=' *expr*
(5)		|	*iden*

[1]Normalmente uma temporária com o resultado da expressão

As regras 4 e 5 são utilizadas para definir o conjunto de índices a ser usado no comando USING. Se o conjunto é explícito, isto é, já foi declarado anteriormente com seus valores definidos, a regra utilizada é a número 5. Neste caso o conjunto de valores é buscado na tabela de símbolos e ulilizado no lugar do identificador.

A regra número 4 é utilizada para conjuntos de índices redefiníveis, e o identificador deve ter sido declarado anteriormente, apenas como sendo um índice do tipo inteiro, de forma a poder assumir posteriormente qualquer conjunto de valores, desde que compátivel com o tipo original. A tabela de símbolos neste caso é consultada simplesmente para se verificar se o identificador corresponde a um índice redefinível e se o tipo está correto.

A regra número 3 determina um comando USING unidimensional. A regra número 2 associa dois conjuntos de índice, formando a edp de um comando USING bidimensional. A sub-árvore criada nesta ocasião é mostrada na Figura 1. O nodo raiz *config* tem o mesmo nome do comando correspondente na LI.

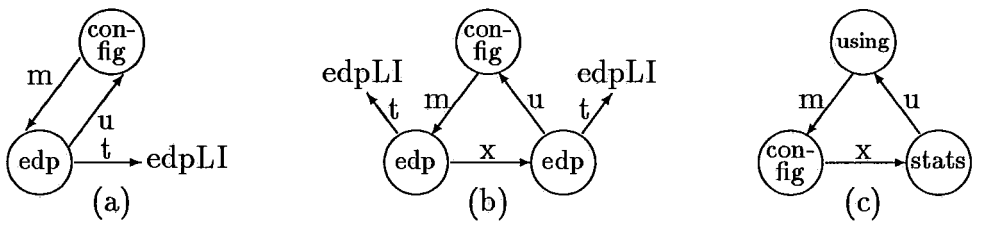


Figura VI.1: Sub-árvores relativas á definição do ambiente paralelo

Como foi visto no capítulo anterior, o índice irregular deve ser transformado em indireto, alterando a sub-árvore *config*, incluindo entre seus filhos, nodos *vardecl* e *cpvec*. Contudo a regra número 1, quando avaliada, recupera a formação original da sub-árvore *config*. Os nodos estranhos a esta sub-árvore são retirados e colocados na posição correta.

Se o índice for explícito, já terá sido colocado na tabela de edp's, possuindo um nodo *edp*. Se o índice for redefinível, será então necessário criar-se um nodo *edp* e um nodo *edpLI* especialmente para ser usado neste comando USING.

As sub-árvores *config* definem a expressão de paralelismo de todos

os comandos avaliados pelo compilador. Para que a sub-árvore *config* correta seja encontrada durante a avaliação dos comandos paralelos, é mantida uma pilha de edp's com ponteiros para esta sub-árvore. A sub-árvore que estiver no topo da pilha será a que define a edp do comando avaliado.

A sub-árvore *config* é colocada no topo da pilha quando da sua criação pelas regras número 2 ou 3. Assim uma nova edp passa a reger os comandos paralelos, desativando a edp que anteriormente ocupava o topo da pilha. A regra número 1 remove a sub-árvore que estiver no topo da pilha, tornando ativa a edp anterior.

No início do bloco de comandos do programa fonte, ou do bloco de comandos de suas funções e procedimentos, o topo da pilha de edp's recebe o ponteiro nulo, definindo o ambiente como seqüencial. Quando o bloco é fechado o ponteiro é removido do topo da pilha.

A regra número 1 tem a função de unir as edp's definidas pela sub-árvore *config* ao comandos (representados pelo não terminal *stats*) que serão executados sob a sua influência. A sub-árvore *using* (ver Figura 1–c) é utilizada para este fim. Quando a regra número 1 produz a sub-árvore *using*, são também geradas as sub-árvores *tempdecl* para declaração das temporárias utilizadas na sub-árvore *stats*.

A seguir apresentamos o trecho de um programa Actus regido pelas regras gramaticais acima e o código LI correspondente. Neste exemplo consideramos que `IS` é descrito pela entrada n da tabela de edp's.

```
USING  IS  DO
  BEGIN
    ...       {stats}
  END

CONFIG;1;n;
    ...       {comandos LI relativos a stats}
```

VI.2 Máscaras

Os comandos USING definem a edp vigente para os comandos contidos em seu bloco. Esta edp não sofre nenhuma influência do ambiente onde o USING foi declarado, ou seja, os comandos externos ao comando USING não interferem na extensão de paralelismo definida pelo USING. Mesmo para comandos USING aninhados, a edp definida por um comando USING mais interno não é influenciado pelo comandos USING mais externos.

Contudo pela semântica de Actus, a edp é mascarada pela variável (ou expressão) de controle dos comandos paralelos seletivos ou de repetição. Assim em uma expressão ou comando de atribuição paralelos, os elementos manipulados são determinados pela interseção entre a edp e os valores TRUE da máscara.

Como a LI exige que a máscara seja explicitada para cada expressão[2], o compilador utiliza durante a geração de código, uma pilha onde coloca a máscara gerada pelos comandos, utilizando a máscara na geração do código LI das expressões paralelas. Na geração de código, quando uma sub-árvore correspondendo a uma atribuição ou expressão paralela é visitada, é gerada a linha de código LI para expressões paralelas (EXPP;), a qual deve ser concatenada a variável utilizada como máscara.

Para cada expressão paralela transcrita para a LI, o compilador consulta o topo da pilha de máscaras, onde encontra a variável a ser utilizada para mascarar a expressão. Se o topo da pilha contiver o valor NULL, não existirá nenhuma máscara atuante, e o string `"0;0;"` é colocado no fim da expressão, indicando à LI que a edp deve ser usada em sua plenitude, sem sofrer nenhuma restrição.

Caso contrário o topo da pilha conterá uma variável (ou temporária) booleana paralela, que será acrescida a cada linha de código LI referente às expressões paralelas. A máscara é transcrita para a LI utilizando a mesma notação adotada para as demais variáveis.

[2]Uma solução mais eficiente seria a LI definir a máscara apenas uma vez, logo no início cada bloco de comandos

A pilha de máscaras é alterada sempre que um comando paralelo é aberto ou fechado. O comando é aberto quando o nodo raiz da sub-àrvore que descreve o comando é visitado. O comando é fechado quando o nodo é abandonado após todos os seus filhos terem sido visitados.

Sempre que um comando paralelo é fechado, a máscara no topo da pilha é removida. Quando o bloco do comando é aberto, se o comando não for um USING, a variável de controle é colocada no topo da pilha. Se o comando for um USING, como a nova edp sempre entra em vigor sem sofrer a ação de nenhuma máscara, coloca-se no topo da pilha o valor NULL.

VI.3 Os Comandos Seletivos

Os comandos seletivos de Actus são: IF THEN, IF THEN ELSE, e CASE. A variável de controle desses comandos determina se o comando será executado na versão paralela ou na versão escalar. Ou seja, variáveis de controle paralelas definem comandos paralelos, e variáveis de controle escalares definem comandos escalares.

Os comandos seqüenciais não atuam na pilha de máscaras, mas se estiverem contidos em um comando USING, poderão conter comandos paralelos. Os comandos paralelos atuam na pilha de máscaras e só podem ser utilizados dentro de comandos USING, sendo que a variável de controle do comando deve ter a mesma edp do USING.

Os comandos (seletivos) paralelos podem ser aninhados, ocasionando que os elementos definidos pela edp fiquem cada vez mais limitados pela máscara. Isto porque a própria variável de controle do comando interno é avaliada sob a influência da máscara do comando externo, definindo uma nova máscara com tantos ou, possivelmente, com menos elementos que a anterior.

VI.3.1 O Comando If

As regras gramaticais relativas a este comando são:

(1) *if_stat* ⟶ *expr* 'THEN' *stat*

(2) | IF' *expr* 'THEN' *stat* *stat*

podendo gerar comandos escalares e paralelos, dependendo da sub-árvore que representar o não terminal *expr*. Esta sub-árvore define a variável de controle do comando. Se a variável de controle for escalar o comando será executado da maneira usual. Apenas o comando paralelo será descrito a seguir.

Os comandos internos ao IF correspondem ao não terminal *stat*, e já terão sido tratados e reunidos em uma sub-árvore *stats* quando a regra número 1 ou 2 for avaliada. A Figura 2 apresenta as interligações da sub-árvore *ifstat* de um comando paralelo.

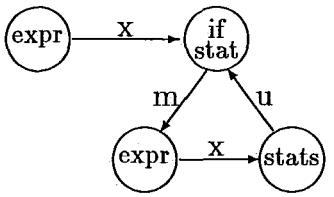


Figura VI.2: Representação de um comando IF paralelo

A semântica deste comando determina que os comandos de *stats* sejam executados se pelo menos um dos elementos da variável de controle tiver o valor TRUE. Os comandos paralelos contidos em *stats* só atuarão nos elementos que corresponderem aos elementos da variável de controle com valor TRUE.

Para determinar se pelo menos um dos elementos da variável de controle tem valor TRUE, a LI possui uma função pré-definida *ifany* que retorna com TRUE se este for o caso, e com FALSE caso contrário. O resultado da função é colocado na temporária representada pelo primeiro nodo *expr* da Figura 2 (o segundo nodo *expr* utiliza a mesma temporária).

A sub-árvore que representa a chamada da função não aparece na Figura 2, estando indicada pelo nome 'ifany' apontado pelo primeiro nodo *expr*. Nesta figura, *pexpr* representa a variável de controle que será utilizada como máscara.

```
IF  A[IS] > B[IS] THEN
```

```
BEGIN
  ...
END
```

O trecho de código Actus acima gera o seguinte código LI:

```
(1) EXPP;GT; 3;PT5;1;1;n; 3;A;1;1;n; 3;B;1;1;n; 0;0;
(2) CALLFUNC; ifany; 1;T1;1;
(3)   CPAR; 2;PT5;
(4) IF; 1;T1;1;k;
(5)   ...   {k linhas de codigo LI utilizando PT5 como mascara}
```

onde pode ser observado que a expressão `A[IS] > B[IS]` (linha 1) é avaliada antes do comando IF ser aberto para a LI, e seu resultado é atribuído a temporária paralela PT5[3]. As linhas 2 e 3 são utilizadas para chamar a função `ifany`, com a temporária `PT5` sendo passada como parâmetro, e o resultado recebido pela temporária T1.

A quarta linha apresenta o comando IF, utilizando a temporária (escalar) T1, que define se o comando será ou não executado. A linha 5 representa todos os comandos internos ao IF. Estes comandos serão realizados utilizando a temporária PT5 como máscara, desde que outro comando não crie um novo contexto e uma nova máscara, isto sendo possível porque os comandos podem ser aninhados.

Note que o comando LI da linha 4 poderia aparentemente ser eliminado, sem que o programa sofresse nenhuma alteração. Isto porque esta linha de código serve apenas para evitar que, com a máscara contendo apenas elementos FALSE, as `k` linhas de código LI sejam avaliadas sem que nenhum elemento da edp possa ser acessado. Assim a linha 4 simplesmente aumentaria a eficiência do código para o caso especial em que todos os valores da máscara forem FALSE. Contudo o código da linha 4 é necessário, porque o comando IF pode conter um comando USING ou comandos escalares, uma vez que os comandos escalares e o comando USING são executados sem sofrer a ação da máscara.

[3]Para esta aplicação é sempre utilizada uma temporária virgem

A máscara a ser usada durante a geração de código, é definida colocando-se no topo da pilha de máscaras a variável de controle do comando. Todas as expressões paralelas da LI utilizam como máscara a variável no topo desta pilha.

A regra número 2, não pode ser traduzida diretamente para a LI, que não reconhece o comando IF THEN ELSE paralelo, só aceitando esta construção para o comando seqüencial. Por isto o comando IF THEN ELSE paralelo deve ser transformado em dois comandos IF THEN como mostrado abaixo. Esta transformação é feita na própria sub-árvore.

```
IF A[IS] > B[IS] THEN
  ...
ELSE
  ...
{comando modificado}
PT5 := A[IS] > B[IS]
IF PT5 THEN
  ...
PT5 := NOT PT5;
IF PT5 THEN
  ...
```

VI.3.2 O Comando Case

As regras gramaticais de Actus que definem a construção do comando CASE são:

(1)	*case_stat*	⟶ OF'	'CASE' *expr cases*
(2)	*cases*	⟶	*cases* ';' *case*
(3)		|	*case*
(4)	*case*	⟶	*const* '..' *const* ':' *stat*
(5)		|	*constseq* ':' *stat*
(6)	*constseq*	⟶	*constseq* ',' *const*
(7)		|	*const*

Assim como para os demais comandos seletivos e de repetição, a variável de controle representada por *expr* na regra número 1 determina, se o comando será ou não paralelo.

Cada ramo do comando, representados pelas regras 4 e 5, deve definir um conjunto de valores exclusivos, isto é, não deve haver interseção entre os conjuntos dos vários ramos. A Figura 3 mostra como cada ramo é representado na árvore sintática, e como a sub-árvore do comando case reune os vários ramos.

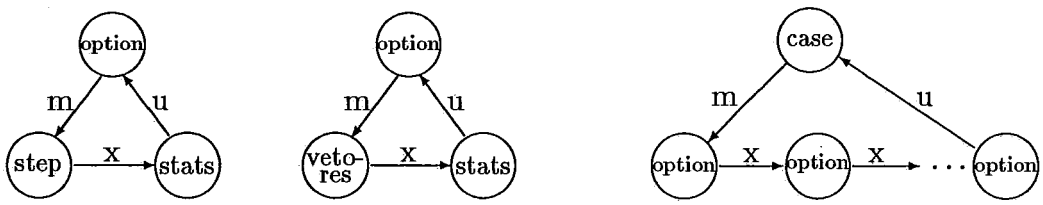


Figura VI.3: Sub-árvores que representam o comando CASE

Como pode ser observado nesta figura, as cláusulas formadas pela regra número 4 tem os valores representados por um nodo *step*, e as formadas pela regra número 5 têm seus valores representados pelo nodo *vetores*. O nodo *expr* da sub-árvore *case* aglutina as várias sub-árvores *option* que compõem o comando.

Contudo a LI só fornece primitivas para o comando CASE seqüencial, devendo o comando paralelo ser transformado em uma sequência de comandos IF. Isto aumenta o número de expressões a serem avaliadas, diminuindo a eficiência do código gerado, principalmente para as cláusulas definidas pela regra 5. Assim o trecho de programa

```
CASE A[IS] + B[IS] OF
  3..9:          ...  {stat1} ;
  10,12,15,33:   ...  {stat2} ;
END
```

será implicitamente transformado pelo compilador nos seguintes comandos:

```
PT1[IS] := A[IS] + B[IS];
IF (PT1[IS] >= 3 AND PT1[IS] <= 9) THEN
```

```
    stat1;
IF (PT1[IS]=10 OR  PT1[IS]=12 OR PT1[IS]=15 OR PT1[IS]=33) THEN
    stats2;
```

A transformação sofrida pelo programa é feita na árvore sintática, onde a sub-árvore *case* é substituída por várias sub-árvores *ifstat*.

VI.4 Os Comandos de Repetição

Os comandos de repetição de Actus são os comandos REPEAT, WHILE e FOR, sendo que o comando FOR não será tratado aqui por não poder ser executado no modo paralelo. As regras da gramática de Actus para os comandos REPEAT e WHILE são:

(1) *repeat_stat* ⟶ 'REPEAT' *stats expr*

(2) *while_stat* ⟶ 'WHILE' *expr* 'DO' *stat*

Estes comandos são semelhantes aos comandos seletivos quanto a formação e manipulação da máscara que controla os comandos contidos em *stat*. As sub-árvores correspondentes a estes comandos são apresentadas na Figura 4.

Figura VI.4: Sub-árvores do comando WHILE e REPEAT

A variável de controle do comando REPEAT só é avaliada no fim do primeiro loop, deixando o valor da máscara indefinido para todos estes comandos. Como a semântica de Actus exige para este comando, que os comandos do primeiro loops sejam avaliados sem sofrer nenhuma restrição da máscara, antes do comando REPEAT ser aberto, uma expresão LI é gerada atribuindo o valor TRUE para todos os elementos da variável de controle.

```
REPEAT
   ...
UNTIL A[IS] > B[IS];
```

```
(1) EXPP;COPY; 3;PT5;1;1;N; 0;1; 0;0; 0;0;
(2) REPEAT;1;T1;k;
(3)    ...  {comandos com a mascara 3;P5;1;1;3;}
(4) EXPP;GT; 3;PT5;1;1;n; 3;A;1;1;n; 3;B;1;1;n; 0;0;
(5) CALLFUNC; ifany; 1;T1;1;
(6)   CPAR;2;PT5;
```

No código LI da linha 1, a variável de controle PT5 tem seus elementos iniciados com o valor 1 (TRUE), antes de iniciar o comando REPEAT (linhas 2 a 6). Os comandos contidos no REPEAT estão contidos em k −3 linhas de código LI (linha 3), sob a influência da temporária PT5, que é utilizada como máscara. Ao fim de cada loop a temporária PT5 é atualizada (linha 4) e a função "ifany" é chamada para avaliar se existe pelo menos um elemento TRUE na máscara. O valor retornado pela função é atribuído à variável T1.

Neste exemplo supomos que o comando REPEAT não está contido em outro comando seletivo ou de repetição, de modo a que o topo da pilha de máscaras contenha o valor NULL. Assim as expressões paralelas nas linhas 1 e 4 terminam com o string "0;0;", indicando que devem ser executadas sem sofrer a ação de máscaras. Se o topo da pilha contivesse, por exemplo, a temporária PT4, as expressões das linhas 1 e 4 terminariam com o string "3;PT4;1;1;n;" substituindo o string "0;0;".

O comando WHILE tem construção semelhante, como mostrado no trecho em Actus e LI, abaixo. Note que não é mais necessário iniciar os elementos da variável de controle, como no comando REPEAT.

```
WHILE A[IS] > B[IS]
   ...
END;
```

```
(1) EXPP;GT; 3;PT5;1;1;n; 3;A;1;1;n; 3;B;1;1;n; 0;0;
(2) CALLFUNC; ifany; 1;T1;1;
(3)   CPAR;2;PT5;
(4) WHILE;1;T1;k;
(5)   ...  {comandos com a mascara 3;P5;1;1;3;}
```

Capítulo VII

Conclusões

Neste trabalho apresentamos a construção de um compilador para a linguagem de programação paralela Actus. As construções paralelas da linguagem s ao apresentadas mais detalhadamente que as seqüenciais (baseadas em Pascal). Foi visto o significado semântico de cada construção paralela, como são representadas pelo compilador e o código LI correspondente.

A compilação é feita em dois passos, construindo-se um árvore sintática no primeiro passo, árvore essa que será utilizada no segundo passo para a geração do còdigo LI. Sistematicamente, a cada produção, o não terminal à esquerda é representado por uma sub-árvore, formada a partir dos terminais, não terminais e ações semânticas à direita da produção.

Declarações e comandos, paralelos e seqüenciais, são naturalmente incorporados à árvore sintática. A árvore sintática mostrou-se adequada e autosuficiente para representar todas as construções paralelas de Actus. A geração de código LI a partir da árvore sintática, completamente construída torna-se uma tarefa simples, com cada nodo sendo transformado em um string (possivelmente vazio) prédeterminado.

Os arrays paralelos e os conjuntos de índices, regulares e irregulares, constituem o aspecto básico na expressão do paralelismo de Actus. A extensão do paralelismo (edp), constituída por conjuntos de índices, e seu escopo são definidos por comandos USING. No escopo de uma extensão de paralelismo, todos os coman-

dos e expressões paralelos têm definidos quantos e quais elementos serão tratados simultaneamente a cada operação.

As expressões e atribuições que operam com dados paralelos, sofrem a ação da edp vigente, e de máscaras criadas pelos comandos seletivos e de repetição paralelos.

A construção deste compilador tem por objetivo permitir a utilização de Actus na programação de aplicações onde o paralelismo possa se explorado, fornecendo mais um recurso à pesquisa na área de arquiteturas, linguagens e processamentos paralelos.

A avaliação do desempenho de Actus nestas aplicações fornecerá dados que permitirão medir sua adequação na expressão eficaz do paralelismo. Alterações e extensões à linguagem poderão ser sugeridas com base nestes dados. Algumas dessas possibilidades são discutidas a seguir.

VII.1 Simplificações na Linguagem Actus

Algumas construções de Actus, herdadas de Pascal, poderiam ser eliminadas, visando simplificar o compilador e gerar códigos mais eficazes, seguindo o exemplo de N. Wirth no projeto da linguagem Oberon [8]. No caso de Actus, as alterações propostas para simplificação consistem em retirar da linguagem os tipos enumerados e subranges.

Para tornar a linguagem mais eficiente, deve ser tambem eliminada a noção de arrays com tipo de índices definíveis. Todos os índices assumindo automaticamente o tipo INTEGER. Além disso, o valor inferior de cada índice será sempre 0, na declaração dos arrays sendo definido o número de elementos de cada dimensão, em vez do par de limites atuais.

As mesmas justificativas, que motivaram Wirth são válidas para Actus, principalmente na indexação de arrays paralelos. Como foi visto no tratamento de expressões, os ajustes feitos para a edp em cada array paralelo geram

uma grande quantidade de cálculo e movimentação de dados na memória, prejudicando a eficiência exatamente das operações onde se pretende conseguir um bom desempenho.

Outra alteração, esta justificada para o caso específico desta implementação, consiste em limitar a edp para uma única dimensão. Como apenas um dos índices poderá ser tratado em paralelo, a edp bidimensional passa uma idéia falsa do paralelismo que pode ser alcançado. Os programas podem ser escritos de maneira mais eficientes se o programador não utilizar as construções paralelas que não podem ser implementadas como tal. Outro benefício alcançado é tornar o compilador mais simples, permitindo que os índices dos arrays paralelos, de dimensão maior que 1, sejam rearrumados pelo compilador, fazendo com que os elementos a serem acessados em paralelo estejam sempre em posições adjacentes.

VII.2 Limitações da Versão Atual

A versão atual sofre de algumas deficiências, notadamente no tratamento de erros sintáticos. À primeira ocorrência de um erro sintático, o compilador interrompe o processo de compilação, o que torna o ambiente de programação bastante hostil durante a fase de depuração do programa.

O tratamento de índices paralelos é feito de maneira pouco eficiente, com o índice de cada elemento da expressão sendo indicado explicitamente. O mesmo ocorrendo com a máscara utilizada para restringir a edp, que é concatenada a cada expressão paralela. Um método mais eficiente seria declarar a edp e a máscara apenas uma vez no início do bloco, por exemplo junto ao comando `CONFIG`. Este procedimento estaria também mais de acordo com o que foi sugerido por P. H. Perrot em [1]. O processo descrito na próxima seção utiliza este princípio. Estas duas operações além de diminuirem o código LI gerado, e eliminarem a tabela de edp's, ainda permitem que o compilador back-end prepare o ambiente onde as operações serão executadas, apenas no início do bloco, e válida para todas as expressões do bloco, não sendo necessário mencioná-las a cada termo paralelo.

A representação para a LI dos conjuntos de índices deveria permitir que operações entre conjuntos fossem representadas como tal, além das representações de conjuntos regulares e irregulares, de modo que conjuntos como

```
IS := 1:20000 + 30000;
```

possam ser representados sem ocupar 20001 posições transcritas uma a uma como atualmente.

O tratamento de funções e procedimentos também não está completo, não sendo possível transferir para a função (ou procedimento) chamada a edp ativa no escopo onde a função foi chamada. Além disso o número de funções pré-definidas é limitado, limitando a aplicação da linguagem. Notadamente as operações com arquivos, que sofrem as restrições impostas por OCAMM. Uma possibilidade seria o compilador gerar código em C, que além de permitir usar a biblioteca já existente para esta linguagem, permitiria ainda que o procedimento descrito na próxima seção pudesse ser gerado diretamente em C.

VII.3 Uma Versão Simplificada do Compilador

Este compilador (front-end e back-end) foi projetado para operar em um dos nós do processador hiper-cúbico baseado em transputer, NCP1, desenvolvido no Laboratório de Computação Paralela da COPPE/UFRJ. Cada nó de processamento contém dois processadores, um transputer T800 e um **intel** *i*860, comunicando-se através de memória compartilhada. O T800 desempenha o papel de processador central com as funções escalares e de controle, além de controlar a comunicação com os demais nós. O *i*860 funciona como um processador vetorial, na verdade um super-escalar, utilizando pipelines aritméticos que lhe permitem atingir até 60 MFLOPS com precisão dupla.

Para esta arquitetura, seria conveniente limitar a expressão de paralelismo a apenas uma dimensão, simplificando ainda mais o compilador, sem limitar o paralelismo que realmente pode ser explorado pela máquina.

Um processo simples de transferir o processamento vetorial para o *i*860, deixando as demais funções para o T800 é apresentado informalmente a seguir, consistindo de alterações na LI, criando uma representação mais simples e concisa para as máscaras e edps, e uma biblioteca de funções básicas para o *i*860. O T800 passa as operações paralelas (sob orientação do compilador) para o *i*860 através de um buffer circular de tamanho limitado. O sincronismo entre os dois processadores é feito através de semáforos.

Os dados paralelos são passados do T800 para o *i*860 através de ponteiros, de modo que o *i*860 manipule os dados diretamente da memória, sem intermediação do T800.

O T800 funciona como uma máquina de pilha, processando as expressões em notação polonesa reversa, acrescida de poucas instruções para permitir o uso das edp's, máscaras, loops e ações de sincronismo, constituindo uma máquina virtual simples e de implementação razoavelmente fácil.

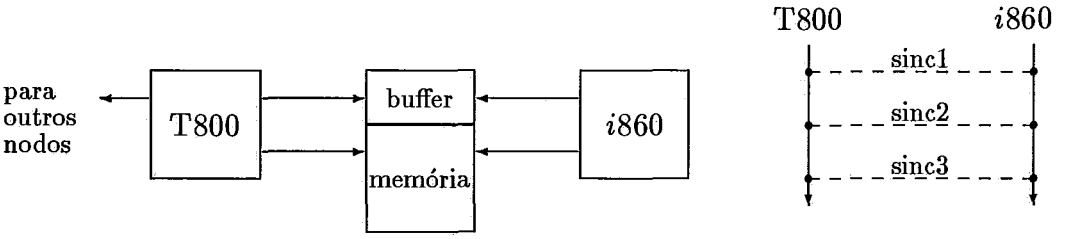


Figura VII.1: Interligação entre processadores de um mesmo nó, e fluxo de controle do programa com sinais de sincronismo entre o T800 e o *i*860

São utilizadas 3 pilhas, uma para os operandos, outra para as edp's e a terceira para as máscaras. O *i*860 executa as operações definidas no buffer, utilizando os operandos no topo da pilha de operando, colocando o resultado no topo. Os operandos são representados por ponteiros, acessados utilizando os índices no topo da pilha de edp's e a máscara no topo da pilha de máscaras.

As operações colocadas no buffer são de tamanho reduzido, consistindo de uma ou duas palavras, e são as seguintes, onde pushO, popO, pushE, popE, pushM e popM são executadas nas pilhas de operandos, edp's e máscaras respectivamente.

P(var), S(var) Operações de sincronismo P e V sobre a variável.

ADD, SUB, ... Operações lógicas, aritméticas e relacionais a serem operadas na pilha de operandos. A edp e máscara vigente são utilizadas para acessar os dados a partir dos ponteiros. Sua semântica é a seguinte (supor operação ADD):

```
op1 := popO; op2 := popO; pushO(op1 ADD op2);
```

Se os operandos forem unários a semântica é a seguinte (supor operação NOT):

```
(op1 := popO; pushO(NOT op1);)
```

COPY Transfere os dados entre operando no topo da pilha. Sua semântica é a seguinte:

```
op1 := popO; op2 := popO; op2 := op1;
```

EDP Transfere para o topo da pilha de edp's o ponteiro no topo da pilha de operadores, coloca o ponteiro nulo no topo da pilha de máscaras. A semântica é a sequinte:

```
(pushE( popO ); pushM( NULL );)
```

ROTATE(k), SHIFT(k) Altera a edp para acessar os elementos do operando no topo da pilha.

endEDP Remove a edp e máscara do topo das respectivas pilhas.

```
(popE; popM; )
```

MASC Coloca no topo da pilha de máscaras o ponteiro no topo da pilha de operandos.

```
(pushM( popO );)
```

PAR(var) Coloca no topo da pilha de operandos o ponteiro para a variável.

ESC(var) Espandir o valor da variável segundo a edp vigente. Colocar no topo da pilha de operandos o ponteiro para a posição dos dados espandidos.

IF Mascarar a edp, proseguir se pelo menos um dos elementos da máscara for TRUE, caso contrário pular para o fim do bloco (RET ou endBLOCO).

LOOP Marca o início de um loop.

RET Retorna para o início do loop correspondente se a máscara vigente conter pelo menos um valor TRUE. Caso contrário remover a máscara no topo da pilha e passar para a próxima instrução do buffer.

endBLOCO Retirar a máscara no topo da pilha de máscaras. Esta instrução e a anterior marcam o fim de um bloco na lista de instruções contidas no buffer.

No código gerado para o T800, este processador passa as primitivas acima diretamente para o buffer. As demais instruções da LI continuam sendo executadas pelo T800. O trecho de programa Actus e o código LI correspondente ilustra o processo.

{programa em Actus}

```
(1) USING IS DO
(2)   BEGIN
(3)     A[IS] := B[IS] - 5;
(4)     WHILE A[IS] > 0
(5)       BEGIN
(6)         C[IS] := A[IS shift-5] * B[IS rotate e];
(7)         IF C[IS] < B[IS] THEN
(8)           A[IS] := A[IS] + e * 5;
(9)         D[IS] := A[IS] + B[IS] * C[IS];
(10)        cont := cont + 1;
(11)      END
(12) END;
```

{programa na LI}

```
(1)  PAR(IS); EDP;
(3)    PAR(A); PAR(B); ESC(50; SUB; COPY;
(4)    LOOP;
```

```
(4a)      PAR(A); ESC(0); GREAT; MASC;
(6)       PAR(C); PAR(A); SHIFT(-5); PAR(B); ROTATE(e); MULT; COPY;
(7)       PAR(C); PAR(B); LESS; MASC;
(7a)      IF;
(8)         EXP;MULT;1;t1; 1;e; 0;5;
(8a)        PAR(A); PAR(A); ESC(t1); MULT; COPY;
(8b)      endBLOCO;
(9)       PAR(D); PAR(A); PAR(B); PAR(C); MULT; ADD; COPY; SINC(1);
(10)      EXP;ADD; 1;cont; 1;cont; 0;1;
(11)    RET; SINC(2);
(12)  endEDP; SINC(3);
```

As instruções das linhas (8) e (10) são executadas pelo T800, as demais serão transferidas diretamente para o buffer. As instruções de sincronismo servem para manter um processador esperando o outro quando necessário. As ações executadas pelo T800, seriam as sequintes:

```
transferir linhas (1) a (7a) para o buffer;
t1 := e * 5;
transferir as linhas (8) a (9) para o buffer;
repetir
  se( sinc1 )
    cont := cont + 1;
    se ultima linha tranferida foi linha (9)
      transferir as linhas (11) e (12) para o buffer;
enquanto( NOT sinc2 );
```

Um otimizador pode ser utilizado para colocar as instruções em uma seqüência mais adequada, de forma a manter as ações de sincronismo no mínimo necessário. As instruções do buffer a serem executadas pelo *i*860 são simples e em número limitado, ficando a maior complexidade em acessar a memória a partir dos índices definidos pela edp e máscara.

Se às instruções de Actus forem acrescentadas instruções de troca

de mensagens entre nodos, Actus torna-se-á uma linguagem adequada para programação paralela também em multicomputadores. É possível também construir-se uma bliblioteca de funções mais sofisticadas para o *i*860, como multiplicação de matrizes, matriz inversa, e outras. As subrotinas seriam chamadas com o T800 colocando o código correspondente direto no buffer, com os parâmetros lidos na pilha de operandos. Este processo, como os anteriores, seria totalmente definido pelo compilador, que simplesmente mapeia o nome da subrotina chamada pelo programador em Actus, no código correspondente a ser colocado no buffer.

Referências Bibliográficas

[1] Perrot, R.H., Lyttle, R.W., e Dhillon., P.S., "The Design and Implementation of a Pascal-Based Language for Array Processor Architetures", *Journal of Parallel and Distrituted Computing*, 4 (1987), 266-287.

[2] Perrot, R.H., "Parallel Programming", *Addison-Wesley*, 1987.

[3] Perrot, R.H., Crooks, D. e Milligan, P., "The Programming Language Actus", *Software – Practice and Experience*, Vol. 13 (1983), 305-322.

[4] Perrot, R.H., "Actus, A Language for Array and Vector Processors – User Manual" , CS023, *Department of Computer Science, The Queen's University*, Fevereiro, 1983.

[5] Sales, C.L, "Projeto e Implementação de uma Linguagem Intermediária para *Transputer*" , *Tese M.Sc. Engenharia de Sistemas e Computação, COPPE / UFRJ*, 1990.

[6] Schneider, S.M., "Gramáticas e Analisadores R*S", *Tese de Doutorado, Engenharia de Sistemas de Computação, COPPE UFRJ*, Setembro, 1987.

[7] Rangel, J.L., "Manual de Operação do Sistema de Geração de Analisadores Sintáticos R*S simples", *Departamento de Informática, PUC–RJ*, Setembro, 1988.

[8] Wirth, N., "From Modula to Oberon", *Software – Practice and Experience*, Vol. 18 (1988), 661-670.

[9] Aho, A.V., Sethi, R, e Ullman, J.D., "Compilers – Principles, Techniques and Tools", *Addison Wesley*, 1986.

[10] Ghezzi, C. e Jazayeri, M., "Conceitos e Linguagens de Programação", *Campus*, 1985.

[11] Tanenbaum, A.S., Bal, H.E. e Steiner, J.G. "Programming Languages for Distributed Computing Systems", *ACM Computing Surveys*, Vol. 21, Setembro, 1989.

Apêndice A

Formação das Sub-árvores

A árvore sintática reflete a estrutura do programa fonte, contendo as informações necessárias e suficientes para a posterior geração do código LI. Na construção da árvore sintática três elementos básicos são utilizados: nodos, strings e o nodo nulo.

Os nodos pertencentes ao conjunto de nodos disponíveis para a costrução da árvore sintática possuem características que são comuns a todos os nodos do conjunto, e outras que são comuns a sub-conjuntos de nodos. Estes sub-conjuntos (que passaremos a chamar de classes) formam as seguintes classes de nodos: *iden*, *predef*, *def*, *grupo*, *struct* e *expr*.

Os nodos têm em comum a estrutura interna, composta de um código, três ponteiros e um atributo.

A.1 Os Ponteiros

Os ponteiros são conhecidos por: *tipo*, *prim* e *prox*, permitindo criar ligações unidirecionais entre o nodo e os outros elementos da sub-árvore, ou seja: um outro nodo, um string ou o nodo nulo. Por extenção estes ponteiros também interligam as sub-árvores, formando assim a árvore sintática. O ponteiro *prim* liga o nodo ao seu **primeiro** filho. O ponteiro *tipo* liga o nodo ao nodo que lhe define o **tipo** que possivelmente será a raiz de outra sub-árvore. O ponteiro *prox* está associado a uma variável booleana que lhe permite ter duplo significado, ligando um nodo filho ao seu **próximo** irmão ou, se for o **último** da lista de filhos, ao seu nodo pai. Onde

conveniente for chamaremos este ponteiro de *último*. Deteta-se se um ponteiro está apontando para o nodo nulo se este contiver o endereço NULL. A simples análise do conteúdo do ponteiro não determina se o mesmo está apontando para um string ou para outro nodo, isto é definido pelo sub-conjunto a que pertence o nodo e de qual dos três ponteiros estamos tratando.

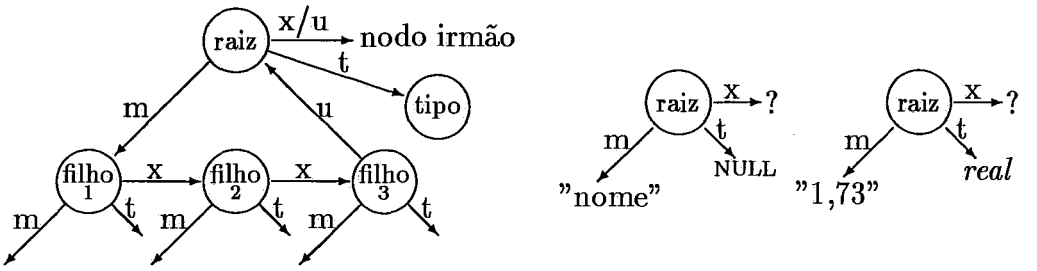


Figura A.1: Exemplo de sub-árvores genéricas

A Figura 1 ilustra o uso dos ponteiros em uma sub-árvore fictícia. Os ponteiros são identificados pelas letras:

t ponteiro *tipo*

m ponteiro *prim*

x ponteiro *prox* se o nodo não for o último da lista

u ponteiro *prox* se o nodo for o último.

Conceitualmente todo nodo (com exceção do nodo nulo) é raiz de uma sub-árvore. O nodo será também um nodo folha, se seu filho for o nodo nulo ou um string. Quando tratarmos da informação contida em um nodo estaremos nos referindo a toda sub-árvore da qual o nodo é raiz, incluindo o nodo apontado pelo ponteiro *tipo*, que passaremos a chamar de nodo **tipo**. Consideramos também que o nodo **tipo** não faz parte da sub-árvore daquele nodo raiz. O nodo raiz apenas partilhará das informações inerentes ao nodo **tipo**.

As Classes

Os nodos de uma mesma classe possuem uma certa semelhança na formação de suas sub-árvores e desempenham funções correlatas na maioria dos casos. Estes nodos

também têm o mesmo tipo de atributo, sendo que cada classe tem seu atributo específico, diferente das demais classes.

As classes têm as seguintes caracteristicas.

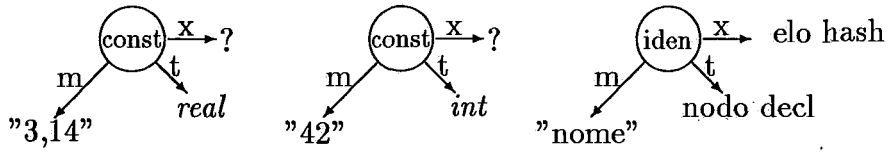


Figura A.2: Exemplo de sub-árvores da classe *iden*

iden Nodos desta classe se interligam a um string de caracteres ASCII, que pode representar o nome de um identificador, o valor de uma constante real ou inteira, um string vazio ou, finalmente um string de inteiros. Seu atributo define o escopo (se aplicável) do identificador. Seu **tipo** normalmente indica o nodo *def* a que pertence o identificador ou o tipo da constante. Seu ponteiro *prox* pode apontar para um nodo irmão, para o nodo nulo, ou para outro nodo *iden* com o mesmo código hash. A Figura 2 ilustra de algumas sub-árvores desta classe.

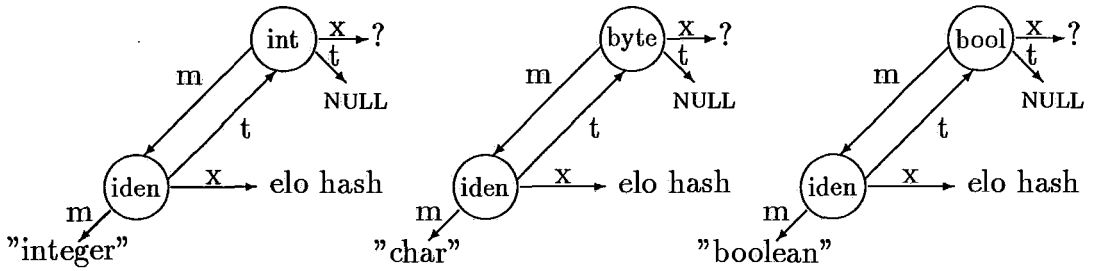


Figura A.3: Exemplos de sub-árvores da classe *predef*

predef Estes nodos caracterizam os tipos pré-definidos pelo compilador. Os demais tipos definidos pelo programa fonte são construídos a partir destes tipos. Não possuem atributos, têm estrutura semelhante às sub-árvores da classe *def*, mas *tipo* sempre contém NULL. A Figura 3 ilustra algumas sub-árvores desta classe.

def Interligam os identificadores declarados no programa fonte ao tipo correspondente, *prim* liga seu nodo a um nodo da classe *iden*, e *tipo* liga o nodo ao

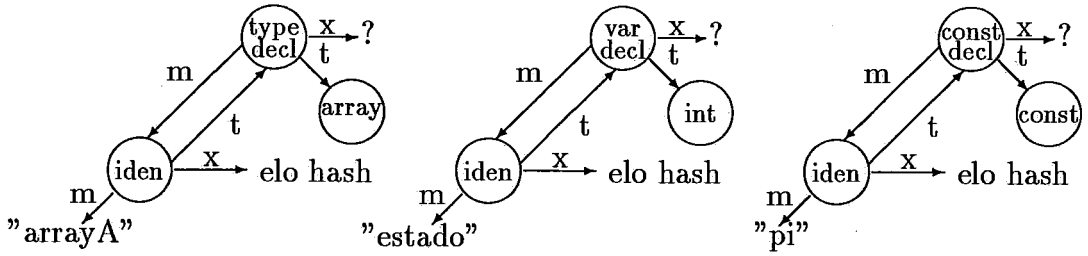


Figura A.4: Exemplos de sub-árvores da classe *predef*

nodo da classe *struct* ou da classe *predef* que caracteriza o tipo. Seu atributo determina a posição do identificador no espaço de nomes da LI. A Figura 4 ilustra algumas sub-árvores desta classe.

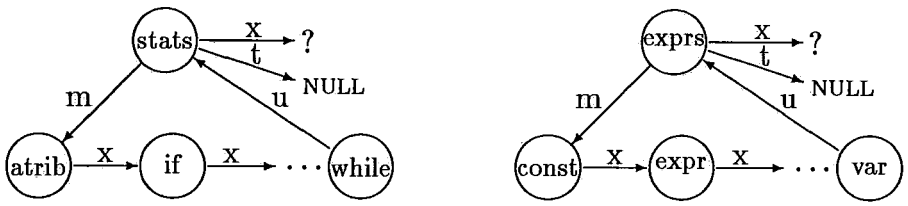


Figura A.5: Exemplo de sub-árvores da classe *grupo*

grupo A função básica dos nodos desta classe é a de agrupar várias outras sub-árvores correlatas como, por exemplo, os comandos de um bloco, os índices de um array, ou os parâmetros atuais de uma função. Os nodos desta classe têm por **tipo** o nodo nulo; *prim* e *prox* têm o significado usual. O atributo dos nodos desta classe contém o número de linhas necessárias à geração do código LI para todos os seus descendentes, não incluindo o próprio nodo. A Figura 5 ilustra algumas sub-árvores desta classe.

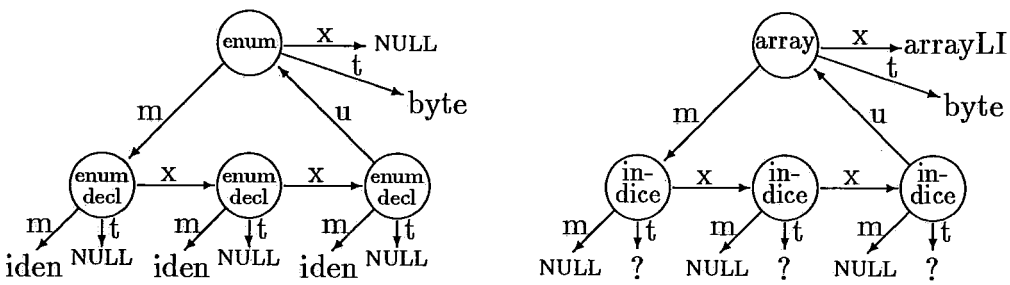


Figura A.6: Exemplo de sub-árvores da classe *struct*

struct Os nodos desta classe são utilizados para descrever a estrutura dos tipos definidos pelo programa fonte a partir de um dos tipos pré-definidos. Entre estes tipos estão os arrays, records, enumerados e subranges. O ponteiro *prox* aponta normalmente para o nodo nulo, *prim* tem o significado usual, e *tipo* aponta para um dos tipos pré-definidos, para o ponteiro nulo ou para outro nodo da classe *struct*. O atributo destes nodos é o número de arrays e de escalares definidos por sua estrutura. A Figura 6 ilustra algumas sub-árvores desta classe.

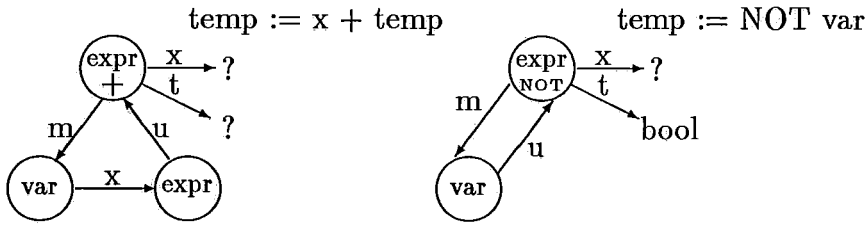


Figura A.7: Exemplos de sub-árvores da classe *predef*

expr Estes nodos representam operações unárias ou binárias, escalares e paralelas. Podem representar também uma variável indexada e chamada de funções e procedimentos. O atributo destes nodos especifica a operação a ser feita com os nodos filhos (soma, multiplicação, ...) e determina também o número da temporária que reterá o resultado parcial da expressão. O ponteiro *tipo* aponta para o tipo resultante da expressão, *prim* e *prox* têm o significado usual. A Figura 7 ilustra algumas sub-árvores desta classe.

Apêndice B

Sub-árvores

A árvore sintática é construída no primeiro passo da compilação, sendo utilizada para a geração de código no segundo passo. A árvore sintática é composta de varias sub-árvores, cada qual representando uma parte do programa fonte e contendo todas as informações de interêsse. A informação contida em uma sub-árvore é função do seu nodo raiz, do seu tipo e dos seus filhos. O nodo tipo pode ser o nodo raiz de outra sub-árvore ou não existir. O nodo filho pode ser um nodo raiz, um string ou não existir. Um nodo sem filhos é um nodo folha, raiz de uma sub-árvore composta de apenas um nodo.

A interligação entre os nodos é feita por ponteiros, cada nodo contém três ponteiros: *tipo*, *prim* e *prox*. O ponteiro *tipo* interliga o nodo a seu tipo, o ponteiro *prim* interliga o nodo a seu primeiro nodo filho, e o ponteiro *prox* interliga um nodo filho a seu irmão ou, se o nodo for o último filho, ao nodo raiz. Para identificar o significado de *prox* o nodo possui uma variável booleana chamada *último*, se *último* for FALSE o ponteiro estará apontando para o próximo nodo irmão, se TRUE o ponteiro estará apontando para o nodo raiz. A Figura 1 representa a iterligação de uma sub-árvore com três nodos filhos. Os ponteiros *tipo*, *prim* e *prox* são identificados pelas letras **t**, **m** e **x** ou **u** respectivamente.

Além dos três ponteiros usados na interligação da sub-árvore os nodos contêm um código e outros atributos. O código determina que atributos o nodo possui e define a regra de formação da sub-árvore da qual é raiz. Os codigos estão divididos em 6 classes: *cl_iden* (de identificadores), *cl_predef* (de tipos pré-definidos),

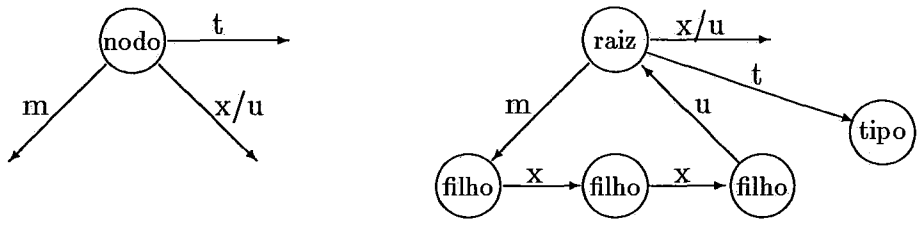


Figura B.1: Nodo genérico e Sub-árvore com três filhos

raiz(1)	t	⟶	tipo	m	⟶	filho	x	⟶	?
t.tipo	t	⟶	?	m	⟶	?	x	⟶	?
m.filho 1	t	⟶	?	m	⟶	?	x	⟶	filho
x.filho 2	t	⟶	?	m	⟶	?	x	⟶	filho
x.filho 3	t	⟶	?	m	⟶	?	u	⟶	(1)raiz

Figura B.2: Exemplo da descrição formal de uma sub-árvore

cl_def (de definições), *cl_grupo* (de grupamentos), *cl_struct* (de estruturas), e *cl_expr* (de expressões). Os nodos com códigos de uma mesma classe possuem os mesmos tipos de atributos e alguma semelhança na estrutura da sub-árvore.

A seguir apresentamos os nodos de cada classe com exemplos das sub-árvores. A Figura 2 descreve a sub-árvore do exemplo anterior, ilustrando o modo usado para descrever as sub-árvores.

Para representar as sub-árvores adotamos a seguinte metodologia:

- Cada nodo é descrito em uma linha, contendo seu código e os nodos aos quais está ligado. Uma sub-árvore com n nodos, quando toda descrita, ocupará n linhas. Os demais atributos dos nodos não são descritos.

- Os ponteiros são representados por letras: *tipo* por **t**, *prim* por **m**, e *prox* por **x** se estiver apontando para o próximo nodo irmão, ou por **u** se estiver apontando para o nodo pai.

- O nodo raiz é descrito antes dos nodos aos quais se interliga. Estes nodos, se descritos, são identados e aparecem na seguinte ordem: o ligado por **t**, o ligado por **m** e finalmente o ligado por **x** ou **u**.

- A referência cruzada entre nodos (Ex. pai e filho) é feita com o número entre parentêses à direita do nodo, quando este aparece pela primeira vez. O número é colocado a esquerda quando o nodo já foi referenciado anteriormente.
- NULL indica que o ponteiro não é usado, ? indica que o nodo apontado não é relevante para a representação em questão.

B.1 Classe dos identificadores

Os nodos desta classe têm códigos: *iden*, *field*, *const*, *strg* e *vetores*.

Estes nodos têm como atributo o nível hierárquico no qual o identificador foi declarado, se esta informação não for de interêsse, seu conteúdo será zero. Seu filho é normalmente um string de caracteres ASCII. O atributo é colocado nos campos nomeados *nivelh* e *nivelv*.

O nodo *iden* interliga o nome da variável à sub-árvore que define seu tipo. Se inserido na tabela hash passa a fazer parte da tabela de símbolos. Quando o escopo do identificador termina, o nodo é retirado da tabela hash. Seu atributo é o número do bloco em que foi declarado, sendo utilizado para definir o escopo da variável.

iden	t	⟶	cl_def	m	⟶	"nome"	x	⟶ elo hash
field	t	⟶	cl_def	m	⟶	"nome"	x	⟶ field
const	t	⟶	cl_predef	m	⟶	"valor"	x	⟶ ?
strg	t	⟶	NULL	m	⟶	"string"	x	⟶ NULL
vetores	t	⟶	edpLI	m	⟶	"lista"	x	⟶ int

O nodo *field* serve para identificar o campo de um record. Não é colocado na tabela de símbolos. O tipo deste nodo pode ser um tipo pré-definido ou um tipo definido pelo programa fonte.

O nodo *const* é usado sempre que for necessário representar uma

constante real ou inteira. O valor da constante está em um string ASCII, apontado por *prim*.

O nodo *strg* é semelhante ao nodo *const*, representando um string alfanumérico.

O nodo *vetores* é utilizado para representar uma lista de inteiros. Seu tipo pode ser *int* ou *enum*.

B.2 Classe dos tipos pré-definidos

Os nodos desta classe têm códigos: *int*, *int32*, *int64*, *real*, *real64*, *byte* e *boolean*.

Estes nodos representam os tipos pré-definidos pelo compilador e permanecem na tabela de símbolos até o fim da compilação. O nodo *byte* corresponde ao tipo CHAR de Actus.

int(1)	t	⟶	NULL	m	⟶	iden	x	⟶ NULL
m.iden	t	⟶	(1)int	m	⟶	"integer"	x	⟶ elo hash
int32(2)	t	⟶	NULL	m	⟶	iden	x	⟶ NULL
m.iden	t	⟶	(2)int32	m	⟶	"int32"	x	⟶ elo hash
int64(3)	t	⟶	NULL	m	⟶	iden	x	⟶ NULL
m.iden	t	⟶	(3)int64	m	⟶	"int64"	x	⟶ elo hash
real(4)	t	⟶	NULL	m	⟶	iden	x	⟶ NULL
m.iden	t	⟶	(4)real	m	⟶	"real"	x	⟶ elo hash
real64(5)	t	⟶	NULL	m	⟶	iden	x	⟶ NULL
m.iden	t	⟶	(5)real64	m	⟶	"real64"	x	⟶ elo hash
byte(6)	t	⟶	NULL	m	⟶	iden	x	⟶ NULL
m.iden	t	⟶	(6)byte	m	⟶	"byte"	x	⟶ elo hash
boolean(7)	t	⟶	NULL	m	⟶	iden	x	⟶ NULL
m.iden	t	⟶	(7)boolean	m	⟶	"boolean"	x	⟶ elo hash

B.3 Classe das definições

Os nodos desta classe têm códigos: *vardecl*, *typedecl*, *constdecl*, *progdecl*, *procdecl*, *fundecl*, *tempdecl*, *enumdecl*, *labeldecl*, *indexdecl*, *parcdecl*, *paramval*, *paramref*, *edpindex*, *step*, *edp*, *arrayLI*, *edpLI*, *parindex*, *rotate*, *shift*, *file*, *assign* e *var*.

Estes nodos são raízes de sub-árvores que representam uma estrutura de dados definida no programa fonte. Podem ser encontrados na tabela de símbolos através de seus identificadores. Têm como atributo a posição que ocupam no espaço de nomes utilizados na linguagem intermediária (LI). O atributo é colocado no campo nomeado *offset*. Os nodos que não ocuparem um nome na LI, terão o *offset* zero.

vardecl(1)	t	⟶	c_struct	m	⟶	iden	x	⟶ ?
m.iden	t	⟶	(1)vardecl	m	⟶	"nome"	x	⟶ elo hash
typedecl(2)	t	⟶	c_struct	m	⟶	iden	x	⟶ ?
m.iden	t	⟶	(2)typedecl	m	⟶	"nome"	x	⟶ elo hash
constdecl(3)	t	⟶	const	m	⟶	iden	x	⟶ ?
t.const	t	⟶	cl_predef	m	⟶	"valor"	x	⟶ NULL
m.iden	t	⟶	(1)vardecl	m	⟶	"nome"	x	⟶ elo hash

O tipo dos nodos *vardecl*, *typedecl* e *fundecl* pode ser pré-definido ou definido no programa fonte, neste caso *tipo* aponta para um nodo *struct*.

Os nodos *typedecl* e *const* têm *offset* zero, uma vez que não são traduzidos para a LI.

progdecl(4)	t	⟶	NULL	m	⟶	iden	x	⟶ bloco
m.iden	t	⟶	(4)progdecl	m	⟶	"nome"	x	⟶ elo hash
procdecl(5)	t	⟶	NULL	m	⟶	iden	x	⟶ params
m.iden	t	⟶	(5)proxdecl	m	⟶	"nome"	x	⟶ elo hash
fundecl(6)	t	⟶	c_struct	m	⟶	iden	x	⟶ params

m.iden	t	$\longrightarrow$	(6)fundecl	m	$\longrightarrow$	"nome"	x	$\longrightarrow$	elo hash
tempdecl	t	$\longrightarrow$	c_predef	m	$\longrightarrow$	NULL	x	$\longrightarrow$	?

O nodo *progdecl* está ligado através de *prox* à sub-árvore com o corpo do programa, representado pelo nodo *bloco.*

Os nodos *procdecl* e *fundecl* estão ligados através de *prox* à sub-árvore que contém seus parâmetros que, por sua vez está interligada à sub-árvore que contém o corpo do procedimento ou função.

O nodo *tempdecl* representa uma quantidade variável (determinada pelo atributo *offset*) de temporárias de um mesmo tipo. Se as temporárias forem paralelas, o ponteiro *tipo* apontará para um nodo *array.*

enumdecl(1)	t	$\longrightarrow$	c_struct	m	$\longrightarrow$	iden	x	$\longrightarrow$	enumdecl
m.iden	t	$\longrightarrow$	(1)enumdecl	m	$\longrightarrow$	"nome"	x	$\longrightarrow$	elo hash
labeldecl(2)	t	$\longrightarrow$	NULL/lbstat	m	$\longrightarrow$	iden	x	$\longrightarrow$	?
m.iden	t	$\longrightarrow$	(2)labeldecl	m	$\longrightarrow$	"nome"	x	$\longrightarrow$	elo hash
indexdecl(3)	t	$\longrightarrow$	step/vetores	m	$\longrightarrow$	iden	x	$\longrightarrow$	?
m.iden	t	$\longrightarrow$	(3)indexdecl	m	$\longrightarrow$	"nome"	x	$\longrightarrow$	elo hash
parcdecl(4)	t	$\longrightarrow$	step/vetores	m	$\longrightarrow$	iden	x	$\longrightarrow$	?
m.iden	t	$\longrightarrow$	(4)parcdecl	m	$\longrightarrow$	"nome"	x	$\longrightarrow$	elo hash

O nodo *enumdecl* está ligado por *prox* aos demais elementos do mesmo tipo enumerado. Seu *offset* indica a sua posição relativa dentro do tipo enumerado.

O nodo *labeldecl* se não for utilizado para rotular um comando terá tipo NULL, caso contrário, seu tipo será o nodo *lbstat* que antecede o comando rotulado.

Os nodos *indexdecl* e *parcdecl* definem um conjunto de valores que, se tiverem formação regular, serão representados por *step* e, se irregular, por *vetores.*

paramval(1)	t	$\longrightarrow$	c_struct	m	$\longrightarrow$	iden	x	$\longrightarrow$	?
m.iden	t	$\longrightarrow$	(1)paramval	m	$\longrightarrow$	"nome"	x	$\longrightarrow$	elo hash
paramref(2)	t	$\longrightarrow$	c_struct	m	$\longrightarrow$	iden	x	$\longrightarrow$	?
m.iden	t	$\longrightarrow$	(2)paramref	m	$\longrightarrow$	"nome"	x	$\longrightarrow$	elo hash
edpindex(3)	t	$\longrightarrow$	c_predef	m	$\longrightarrow$	iden	u	$\longrightarrow$	index
m.iden	t	$\longrightarrow$	(3)edpindex	m	$\longrightarrow$	"nome"	x	$\longrightarrow$	elo hash
step(4)	t	$\longrightarrow$	edpLI	m	$\longrightarrow$	var/const	x	$\longrightarrow$	NULL/expr
m.var/const	t	$\longrightarrow$	?	m	$\longrightarrow$	?	x	$\longrightarrow$	var/const
x.var/const	t	$\longrightarrow$	?	m	$\longrightarrow$	?	u	$\longrightarrow$	(4)step
edp	t	$\longrightarrow$	edpLI	m	$\longrightarrow$	indexdecl	x	$\longrightarrow$	edp

Se o nodo *step* tiver seu tamanho conhecido em tempo de compilação (ambos os filhos são *const*) o ponteiro *prox* conterá NULL. Caso contrário *prox* apontará para um nodo *expr* que calculará o tamanho em tempo de execução. O *offset* do nodo *step* contém o valor do incremento.

O *offset* de *edp* será zero, se for regular e 1, se irregular.

arrayLI	t	$\longrightarrow$	array	m	$\longrightarrow$	NULL	x	$\longrightarrow$	arrayLI
edpLI	t	$\longrightarrow$	edp/sh/rot	m	$\longrightarrow$	NULL/...	x	$\longrightarrow$	edpLI
parindex	t	$\longrightarrow$	edpLI	m	$\longrightarrow$	NULL/...	x	$\longrightarrow$	?
rotate	t	$\longrightarrow$	edpLI	m	$\longrightarrow$	edp	x	$\longrightarrow$	NULL
shift	t	$\longrightarrow$	edpLI	m	$\longrightarrow$	edp	x	$\longrightarrow$	NULL

Para os nodos *arrayLI* e *edpLI* o atributo *offset* indica a entrada correspondente na tabela LI de arrays ou de edps.

Se o nodo *edpLI* representa uma edp irregular, *prim* aponta para um nodo *var* ou *pexpr* que substituirá a edp nas indexacões.

O nodo *parindex* define um índice paralelo. A entrada na tabela de edps LI é encontrada somando-se o valor do *offset* à entrada definida pelo nodo

edpLI. Se *prim* for NULL a indexação é direta, caso contrário, é indireta, utilizando *pexpr* ou *var*.

Os nodos *rotate* e *shift* guardam no atributo *offset* o valor do deslocamento requerido para a edp.

file(1)	t	⟶	c_predef	m	⟶	vardecl	x	⟶ ?
m.vardecl(2)	t	⟶	(1)file	m	⟶	iden	x	⟶ ?
m.iden	t	⟶	(2)vardecl	m	⟶	"nome"	x	⟶ elo hash
assign	t	⟶	file	m	⟶	"nome"	x	⟶ ?
var	t	⟶	c_struct	m	⟶	iden	x	⟶ ?
m.iden	t	⟶	vardecl	m	⟶	"nome"	x	⟶ elo hash

O nodo *file* tem *offset* zero se o arquivo não estiver aberto, 1 se aberto para leitura e 2 se aberto para escrita. Associa o nome lógico do arquivo ao seu tipo.

O nodo *assign* associa o arquivo ao seu nome físico.

O nodo *var* representa a váriavel em expressões, seu ponteiro *prim* aponta para o mesmo nodo *iden* filho da sub-árvore *vardecl* que define a variável. O *offset* de *var* é o mesmo de *vardecl*.

B.4 Classe dos grupos

Os nodos desta classe têm códigos: *bloco*, *decls*, *stats*, *exprs*, *parms*, *rparams*, *pfbody*, *atrib*, *if*, *ifelse*, *while*, *repeat*, *for*, *case*, *option*, *using*, *config*, *lbstat*, *gostat*, *gotos*, *cpvec*, *open*, *create*, *close*, *finish*, *newline*, *read*, *write*, *printvector* e *inicvector*.

Estes nodos são raízes de sub-árvores que têm a função básica de agrupar outras sub-árvores. Via de regra, estes nodos não possuem tipo e informam no atributo *linhas* o número de linhas que serão usadas na geração de código LI por todos os seus descendentes. Este atributo permite que na geração de código o número de linhas LI, ocupadas por um comando estruturado possa ser definido

antes da geração dessas linhas.

bloco(1)	t	⟶	NULL	m	⟶	decls	u	⟶	pfbody
m.decls	t	⟶	?	m	⟶	?	x	⟶	stats
x.stats	t	⟶	?	m	⟶	?	u	⟶	(1)bloco
decls	t	⟶	NULL	m	⟶	cl_def	x	⟶	stats
stats	t	⟶	NULL	m	⟶	cl_grupo	u	⟶	bloco
exprs	t	⟶	NULL	m	⟶	expr	u	⟶	?
params	t	⟶	NULL	m	⟶	paramref	x	⟶	bloco
rparams	t	⟶	NULL	m	⟶	paramref	x	⟶	bloco
pfbody	t	⟶	NULL	m	⟶	params	x	⟶	?

Os nodos *decls*, *stats*, *exprs*, *params*, *rparams* e *pfbody* têm um número variável de filhos, que formam uma lista de um ou mais nodos.

O nodo *decls* agrupa todas as declarações do bloco, seus filhos são todos nodos da classe *c_def*.

O nodo *stats* agrupa uma sequência de comandos como: *atrib*, *if*, *repeat*, *while*, ...

O nodo *exprs* agrupa os n nodos que representam expressões, váriaveis, temporárias utilizadas, por exemplo, como índices de um array de dimensão n.

Os nodos *params* e *rparams* agrupa os nodos *paramref* e *paramval* que definem os parâmetros de uma função ou procedimento. O nodo *params* faz parte da sub-árvore de uma função ou procedimento não recursiva. O nodo *rparams* indica recursividade, esta informação é necessária a LI.

O nodo *pfbody* agrupa todos os nodos referentes à declaração de uma procedure ou função. Seus filhos são os nodos *procdecl* ou *fundecl*, os nodos *params* ou *rparams*, e o nodo *bloco*.

atrib(1)	t	⟶	NULL	m	⟶	var/...	x	⟶	?

m.var	t	$\longrightarrow$	?	m	$\longrightarrow$	?	x	$\longrightarrow$ expr/...
x.expr	t	$\longrightarrow$	?	m	$\longrightarrow$	?	x	$\longrightarrow$ (1)atrib
if(2)	t	$\longrightarrow$	NULL	m	$\longrightarrow$	expr	x	$\longrightarrow$?
m.expr	t	$\longrightarrow$	?	m	$\longrightarrow$	?	x	$\longrightarrow$ stats
x.stats	t	$\longrightarrow$	?	m	$\longrightarrow$	?	u	$\longrightarrow$ (2)if
ifelse(3)	t	$\longrightarrow$	NULL	m	$\longrightarrow$	expr	x	$\longrightarrow$?
m.expr	t	$\longrightarrow$	?	m	$\longrightarrow$	?	x	$\longrightarrow$ stats
x.stats	t	$\longrightarrow$	?	m	$\longrightarrow$	?	u	$\longrightarrow$ stats
x.stats	t	$\longrightarrow$	?	m	$\longrightarrow$	?	x	$\longrightarrow$ (3)ifelse
while(4)	t	$\longrightarrow$	NULL	m	$\longrightarrow$	expr	x	$\longrightarrow$?
m.expr	t	$\longrightarrow$	?	m	$\longrightarrow$	?	x	$\longrightarrow$ stats
x.stats	t	$\longrightarrow$	?	m	$\longrightarrow$	?	u	$\longrightarrow$ (4)while
repeat(5)	t	$\longrightarrow$	NULL	m	$\longrightarrow$	stats	x	$\longrightarrow$?
m.stats	t	$\longrightarrow$	?	m	$\longrightarrow$	?	x	$\longrightarrow$ exprs
x.exprs	t	$\longrightarrow$	?	m	$\longrightarrow$	?	u	$\longrightarrow$ (5)repeat

O nodo *atrib* pode ser escalar ou paralelo, definido pelo nodo à esquerda (i.e o primeiro filho), se este nodo for paralelo, o comando de atribuição será paralelo, se for escalar o comando também o será. O nodo à esquerda pode ser *var*, *expr* ou *pexpr* assim como o nodo a direita.

Os comandos representados pelos nodos *if*, *ifelse*, *while* e *repeat* também podem ser paralelos ou escalares. Isto é definido pela variável de contrôle do comando, o nodo *expr*. Se paralelo este nodo será um nodo *pexpr*, e seu valor será usado na geração de código como máscara para determinar os índices que devem ser avaliados.

Para os comandos *if* e *ifelse* a avaliação da expressão é feita por nodos colocados fora de suas sub-árvores, *expr* retém apenas o resultado da avaliação. O nodo *expr* dos comandos *while* e *repeat* incorporam a avaliação completa da expressão, que deve ser reavaliada a cada loop.

Se o comando for paralelo o resultado da expressão é utilizado para

definir um vetor máscara que determina quais os elementos de cada array envolvido no comando devem ser avaliados.

for(1)	t	⟶	NULL	m	⟶	expr	x	⟶ ?
m.expr	t	⟶	?	m	⟶	?	x	⟶ stats
x.stats	t	⟶	?	m	⟶	?	u	⟶ (1)for
case(2)	t	⟶	NULL	m	⟶	expr	x	⟶ ?
m.expr	t	⟶	?	m	⟶	?	x	⟶ exprs
x.exprs	t	⟶	?	m	⟶	option	u	⟶ (2)case
option(3)	t	⟶	NULL	m	⟶	step/...	x	⟶ ?
m.step	t	⟶	?	m	⟶	?	x	⟶ stat
x.stat	t	⟶	?	m	⟶	?	u	⟶ (3)option

A sub-árvore do comando *for* inclui a atualização da variável de contrôle que pode ser lida, mas não alterada, pelos comandos pertencentes ao *for*. A avaliação dos valores inicial e final da variável de contrôle, bem como o cálculo do número de loops do comando é feita por expressões definidas pelas sub-árvores que antecedem o *for*.

A sub-árvore *case* só é construída para comandos escalares. Os comandos *case* paralelos são transformados em uma sequência de *if*s paralelos, uma vez que a LI não reconhece *case* paralelo. O nodo *expr* contém a variável de contrôle. O nodo *exprs* contém a lista de opções do comando, representada pelos nodos *option*.

A sub-árvore *option* define a opção de cada um dos comandos de *case*. O ponteiro *prim* aponta para o nodo com a sequência de escalares a ser utilizada. A sequência regular é definida pelo nodo *step*, a sequência irregular é definida pelo nodo *vetores*.

using(2)	t	⟶	NULL	m	⟶	config	x	⟶ ?
m.config	t	⟶	?	m	⟶	?	x	⟶ stats
x.stats	t	⟶	?	m	⟶	?	u	⟶ (2)using

config(1)	t	$\longrightarrow$	tempdecl	m	$\longrightarrow$	edp	x	$\longrightarrow$?
m.edp	t	$\longrightarrow$	?	m	$\longrightarrow$	indexdecl	u	$\longrightarrow$ (1)config

Os nodos *using* contêm os comados paralelos, não é permitida a existência de um comando paralelo fora da sub-árvore *using*. A extensão de paralelismo é definida pelo nodo *config*.

O nodo *config* tem um ou dois filhos *edp*, cada um podendo ser regular ou irregular, de acordo com o nodo *indexdecl*.

lbstat	t	$\longrightarrow$	NULL	m	$\longrightarrow$	labeldecl	x	$\longrightarrow$ stat
gostat	t	$\longrightarrow$	labeldecl	m	$\longrightarrow$	labeldecl	x	$\longrightarrow$?
gotos	t	$\longrightarrow$	labeldecl	m	$\longrightarrow$	lbstats	x	$\longrightarrow$ gotos
cpvec(3)	t	$\longrightarrow$	vetores	m	$\longrightarrow$	var/pexpr	x	$\longrightarrow$?
m.var	t	$\longrightarrow$	?	m	$\longrightarrow$	"nome"	u	$\longrightarrow$ (3)cpvec

O nodo *lbstat* é utilizado para anteceder o nodo *stat* que se deseja rotular. O nome do rótulo é dado pelo nodo *labeldecl*.

O nodo *gostat* representa um comando de desvio para o comando que receber o rótulo cujo nome está definido por *labeldecl*.

O nodo *gotos* faz parte de uma lista que mantem uma referência cruzada entre os nodos *lbstat* e *gostat*, com a função de supervisionar o uso correto dos rótulos. Os nodos *gotos* só existem durante o primeiro passo da compilação.

O nodo *cpvec* transforma os índices irregulares de *vetores* em índices regulares indiretos, atribuindo os valores de *vetores* a *pexpr* ou *var* que passam a ser usadas como índice.

open	t	$\longrightarrow$	file	m	$\longrightarrow$	NULL	x	$\longrightarrow$?
create	t	$\longrightarrow$	file	m	$\longrightarrow$	NULL	x	$\longrightarrow$?

close	t	⟶	file	m	⟶	NULL	x	⟶ ?
finish	t	⟶	file	m	⟶	NULL	x	⟶ ?
newline	t	⟶	file	m	⟶	NULL	x	⟶ ?
read(1)	t	⟶	cl_predef	m	⟶	var	x	⟶ ?
m.var	t	⟶	file	m	⟶	iden	x	⟶ var
x.var	t	⟶	cl_predef	m	⟶	iden	u	⟶ (1)read
write(2)	t	⟶	cl_predef	m	⟶	var	x	⟶ ?
m.var	t	⟶	file	m	⟶	iden	x	⟶ var
x.var	t	⟶	cl_predef	m	⟶	iden	u	⟶ (2)write
printvector(3)	t	⟶	NULL	m	⟶	var	x	⟶ ?
m.var	t	⟶	?	m	⟶	"nome"	x	⟶ var
x.var	t	⟶	?	m	⟶	iden	u	⟶ printvector(3)
inicvector(4)	t	⟶	NULL	m	⟶	var	x	⟶ ?
m.var	t	⟶	?	m	⟶	"nome"	x	⟶ var
x.var	t	⟶	?	m	⟶	iden	u	⟶ inicvector(4)

Os nodos supracitados são utilizados para a operação com arquivos. Os nodos *open*, *create*, *close*, *finish* e *newline* não possuem parâmetros e apenas indicam o arquivo no qual se deve realizar a operação.

O primeiro filho dos nodos *read* e *write* define o nome lógico do arquivo, o segundo filho define a váriavel que deve receber o dado lido ou fornecer o dado a ser escrito.

Os nodos *printvector* e *inicvector* representam procedimentos pré-definidos existentes na LI.

B.5 Classe das estruturas

Os nodos desta classe têm códigos: *subr*, *psubr*, *enum*, *array*, *record* e *índice*.

Estes nodos são raízes de sub-árvores que caracterizam um tipo definido pelo programa fonte, normalmente a partir dos tipos pré-definidos.

O atributo é colocado no campo nomeado *esc* e *mtrz*, contendo o número de escalares e arrays do nodo. Esta informação é para a maioria dos nodos $esc = 1$, $mtrz = 0$. Para o nodo *array* temos: $esc = 0$ e $mtrz > 0$. Para o nodo *record* temos: $esc >= 0$ e $mtrz >= 0$.

subr(1)	t	⟶	cl_predef	m	⟶	const	x	⟶ NULL
m.const	t	⟶	?	m	⟶	"valor"	x	⟶ const
x.const	t	⟶	?	m	⟶	"valor"	u	⟶ (1)subr
psubr(2)	t	⟶	cl_predef	m	⟶	const	x	⟶ NULL
m.const	t	⟶	?	m	⟶	"valor"	x	⟶ const
x.const	t	⟶	?	m	⟶	"valor"	u	⟶ (2)psubr
enum(3)	t	⟶	byte	m	⟶	enumdecl	x	⟶ NULL
m.enumdecl	t	⟶	?	m	⟶	iden	x	⟶ enumdecl
m.enumdecl	t	⟶	?	m	⟶	iden	x	⟶ enumdecl
enumdecl	t	⟶	?	m	⟶	iden	u	⟶ (3)enum
array(4)	t	⟶	...	m	⟶	indice	x	⟶ arrayLI
m.indice	t	⟶	?	m	⟶	NULL	x	⟶ indice
x.indice	t	⟶	?	m	⟶	NULL	u	⟶ (4)array
record(5)	t	⟶	NULL	m	⟶	field	x	⟶ NULL
m.field	t	⟶	?	m	⟶	"nome"	x	⟶ field
x.field	t	⟶	?	m	⟶	"nome"	u	⟶ (5)record
indice	t	⟶	...	m	⟶	NULL	x	⟶ indice

O nodo *subr* define os limites de um tipo subrange escalar e, o nodo *psubr* o de um subrange paralelo. O tipo desses nodos pode ser também *enum* se for subrange de um tipo enumerado.

O nodo *enum* tem tantos filhos quantos forem seus elementos.

O nodo *array* tem como tipo um nodo da classe *cl_predef* ou *cl_struct* com exceção do nodo *índice*.

Assim como em Pascal, é possível definir arrays de records, arrays de

arrays, records de arrays e records de records recursivamente.

O nodo *índice* pode ter como tipo: *enum*, *subr*, *psubr* ou *edpindex*.

B.6 Classe das expressões

Os nodos desta classe têm códigos: *expr* e *pexpr*.

Estes nodos são raízes de sub-árvores que determinam uma operação binária ou unária. Podem também representar uma variável indexada ou a chamada de uma função ou procedimento.

Estes nodos possuem dois atributos: *temp* que contém o número da temporária que retém o resultado parcial da expressão e *op* que indica a operação a ser feita.

O atributo *op* pode representar os operadores unários e binários (relacionais, atitméticos e lógicos), e casos especiais como:

callfun, callproc Chamada de função ou procedimento e seus parâmetros.

atrib Variável indexada e seus índices.

result Temporária que retorna com o resultado de uma função.

conv Operação de conversão de tipos.

expr(1)	t	⟶	cl_predef	m	⟶	var/...	x	⟶ ?
m.var	t	⟶	?	m	⟶	?	x	⟶ var/...
x.var	t	⟶	?	m	⟶	?	u	⟶ (1)expr
expr(2)	t	⟶	cl_predef	m	⟶	var/...	x	⟶ ?
x.var	t	⟶	?	m	⟶	?	u	⟶ (2)expr
pexpr(3)	t	⟶	cl_predef	m	⟶	var/...	x	⟶ ?
m.var	t	⟶	?	m	⟶	?	x	⟶ var/...
x.var	t	⟶	?	m	⟶	?	u	⟶ (3)pexpr
pexpr(4)	t	⟶	cl_predef	m	⟶	var/...	x	⟶ ?

x.var	t	⟶	?	m	⟶	?	u	⟶	(4)pexpr

Acima estão representadas sub-árvores binárias e unárias, escalares e paralelas. Os nodos filhos podem ser *var*, *const*, *expr* ou *pexpr*.

Estes nodos correspondem, na LI, a operação de avaliação da expressão e atribuição do resultado, a uma temporária. A temporária é definida pelo tipo da expressão e pelo seu atributo *temp*.

As temporárias paralelas definidas nos nodos *pexpr* são indexadas implicitamente pelos índices (edp) do comando *using* a que pertencem.

expr(1)	t	⟶	cl_predef	m	⟶	var	x	⟶	?
m.var	t	⟶	array	m	⟶	iden	x	⟶	exprs
x.exprs(2)	t	⟶	cl_predef	m	⟶	expr/...	u	⟶	(1)expr
m.expr	t	⟶	?	m	⟶	?	x	⟶	expr/...
x.expr	t	⟶	?	m	⟶	?	u	⟶	(2)exprs

A sub-árvore acima representa uma variável indexada de duas dimensões. Os índices são os filhos da sub-árvore *exprs* e podem ser também *var* e *const*. O atributo *op* do nodo raiz é *atrib*.

A sub-árvore que representa a chamada de função e procedimento tem a mesma construção acima, onde os nodos filhos de *exprs* passam a ser os parâmetros. O atributo *op* neste caso será *callfun* ou *callproc*. Se a sub-árvore corresponder a chamada de função ,*temp* indica a temporária que receberá o resultado da função. Se corresponder a chamada de procedimento, *temp* será zero.

Apêndice C

Gramática Actus

Apresentamos, a seguir, a BNF da gramática utilizada na implementação deste compilador.

Os não terminais são descritos por strings contendo somente minúsculas. Os terminais são descritos por strings entre aspas ou por strings contento somente maiúsculas. Os meta-símbolos utilizados na descrição da gramática são:

$\longrightarrow$	Liga um não terminal à sua regra de formação.
|	Separa as várias regras de formação de um mesmo não terminal.
ε	O string vazio.

C.1 BNF da Gramática Actus

prog	$\longrightarrow$	*prog_head bloc "."*
bloc	$\longrightarrow$	*decls comp_stat*
prog_head	$\longrightarrow$	*PROGRAM iden ";"*
decls	$\longrightarrow$	*decls decl*
	|	ε
decl	$\longrightarrow$	*label_part*
	|	*const_part*
	|	*parconst_part*

	|	*type_part*
	|	*var_part*
	|	*index_part*
	|	*proc_part*
label_part	⟶	*LABEL idens ";"*
const_part	⟶	*CONST const_decls*
parconst_part	⟶	*PARCONST parconst_decls*
type_part	⟶	*TYPE type_decls*
var_part	⟶	*VAR var_decls*
index_part	⟶	*INDEX index_decls*
proc_part	⟶	*proc_head bloc ";"*
	|	*proc_head FORWARD ";"*
const_decls	⟶	*const_decls const_decl*
	|	*const_decl*
parconst_decls	⟶	*parconst_decls parconst_decl*
	|	*parconst_decl*
type_decls	⟶	*type_decls type_decl*
	|	*type_decl*
var_decls	⟶	*var_decls var_decl*
	|	*var_decl*
index_decls	⟶	*index_decls index_decl*
	|	*index_decl*
const_decl	⟶	*iden "=" expr ";"*
parconst_decl	⟶	*iden "=" exprs ";"*
type_decl	⟶	*iden "=" type ";"*
var_decl	⟶	*idens ":" type ";"*
index_decl	⟶	*idens ":" expr ";"*
proc_head	⟶	*PROCEDURE iden formal_part ";"*
	|	*FUNCTION iden formal_part ":" iden ";"*
formal_part	⟶	*"(" param_groups ")"*
	|	ε
param_groups	⟶	*param_groups ";" param_group*

	|	*param_group*
param_group	⟶	*VAR idens ":" p_type*
	|	*idens ":" iden*
p_type	⟶	*iden*
	|	*ARRAY "[" index_specs "]" OF p_type*
type	⟶	*simple_type*
	|	*ARRAY "[" index_types "]" OF type*
	|	*RECORD fields END*
	|	*FILE OF iden*
simple_type	⟶	*iden*
	|	*"(" idens ")"*
	|	*const ".." const*
	|	*const ":" const*
idens	⟶	*idens "," iden*
	|	*iden*
index_types	⟶	*index_types "," simple_type*
	|	*simple_type*
fields	⟶	*fields ";" field*
	|	*field*
field	⟶	*idens ":" type*
index_specs	⟶	*index_specs ";" index_spec*
	|	*index_spec*
index_spec	⟶	*iden ".." iden ":" iden*
	|	*INDEX iden ":" iden*
comp_stat	⟶	*BEGIN stats END*
stats	⟶	*stats ";" stat*
	|	*stat*
stat	⟶	*atrib_stat*
	|	*if_stat*
	|	*ifelse_stat*
	|	*while_stat*
	|	*repeat_stat*

	|	*for_stat*
	|	*case_stat*
	|	*using_stat*
	|	*callproc_stat*
	|	*label_stat*
	|	*goto_stat*
	|	*comp_stat*
	|	*empty_stat*
atrib_stat	$\longrightarrow$	*var ":=" expr*
if_stat	$\longrightarrow$	*IF expr THEN stat*
ifelse_stat	$\longrightarrow$	*IF expr THEN stat ELSE stat*
while_stat	$\longrightarrow$	*WHILE expr DO stat*
repeat_stat	$\longrightarrow$	*REPEAT stats UNTIL expr*
for_stat	$\longrightarrow$	*FOR iden ":=" expr sentido expr DO stat*
case_stat	$\longrightarrow$	*CASE expr OF cases END*
using_stat	$\longrightarrow$	*USING edps_do stat*
callproc_stat	$\longrightarrow$	*iden "(" exprs ")"*
label_stat	$\longrightarrow$	*iden ":" stat*
goto_stat	$\longrightarrow$	*GOTO iden*
empty_stat	$\longrightarrow$	ε
sentido	$\longrightarrow$	*TO*
	|	*DOWNTO*
cases	$\longrightarrow$	*cases ";" case*
	|	*case*
case	$\longrightarrow$	*case_const ":" stat*
case_const	$\longrightarrow$	*const ".." const*
	|	*const_seq*
const_seq	$\longrightarrow$	*const_seq "," const*
	|	*const*
edps_do	$\longrightarrow$	*edps DO*
edps	$\longrightarrow$	*edps "," edp*
	|	*edp*

edp	⟶	*iden* “:=” *expr*
	|	*iden*
expr	⟶	*simple_expr*
	|	*simple_expr rel_op simple_expr*
	|	*iden desloc const*
simple_expr	⟶	*term*
	|	*simple_expr add_op term*
term	⟶	*factor*
	|	*term mul_op factor*
factor	⟶	*number*
	|	*string*
	|	*var*
	|	*call_func*
	|	*un_op factor*
	|	“(” *expr* “)”
call_func	⟶	*iden* “(” *exprs* “)”
exprs	⟶	*exprs* “,” *expr*
	|	*expr*
var	⟶	*iden*
	|	*var* “[” *exprs* “]”
	|	*var* “.” *iden*
add_op	⟶	“+”
	|	“-”
	|	*OR*
mul_op	⟶	“*”
	|	“/”
	|	*DIV*
	|	*MOD*
	|	*AND*
	|	“:” *step*
rel_op	⟶	“=”
	|	“<>”

	|	"<"
	|	"<="
	|	">"
	|	">="
sign	⟶	"+"
	|	"-"
un_op	⟶	*sign*
	|	*ANY*
	|	*ALL*
	|	*NOT*
step	⟶	"[" *const* "]"
	|	ε
desloc	⟶	*SHIFT*
	|	*ROTATE*
const	⟶	*number*
	|	*iden*
	|	*string*
	|	*sign number*
	|	*sign iden*
number	⟶	*int*
	|	*boolean*
	|	*real*
boolean	⟶	*FALSE*
	|	*TRUE*
iden	⟶	*ID*
int	⟶	*INT*
real	⟶	*REAL*
string	⟶	*STRING*