MIR: Árvore das regras de transição

1. Introdução

A arquitetura MIR é uma infraestrutura projetada para servir como base para compiladores concorrentes ASM. O projeto MIR, Machĭna Intermediate Representation, originou-se na necessidade de uma linguagem intermediária durante a compilação de código Machĭna para código C. O projeto foi ampliado e atualmente consiste em um projeto separado da compilação de Machĭna. Com o projeto **Arquitetura MIR** será possível realizar experiências com linguagens baseadas no modelo ASM.

2. Arquitetura MIR

Uma ASM seqüencial gera um conjunto de atualizações a partir da execução de uma única regra de transição. Uma ASM distribuída é formada por muitos agentes que podem estar executando diferentes programas. O conjunto de atualizações de uma ASM distribuída é a união dos conjuntos de atualizações gerados por um subconjunto dos agentes ativos, que são aqueles que estão executando uma regra de transição.

O desenvolvimento de uma infraestrutura geral para compiladores ASM tem por base a possibilidade de realizar experiências com linguagens orientadas ao modelo ASM. A arquitetura MIR foi projetada para suprir tal necessidade. Além disso, possui a capacidade de execução concorrente, útil na implementação de algoritmos concorrentes.

O objetivo principal da Infraestrutura MIR é produzir código ANSI C a partir de uma especificação MIR, permitindo, então, um desenvolvimento rápido para compiladores ASM. A Figura 1 mostra o contexto de MIR.

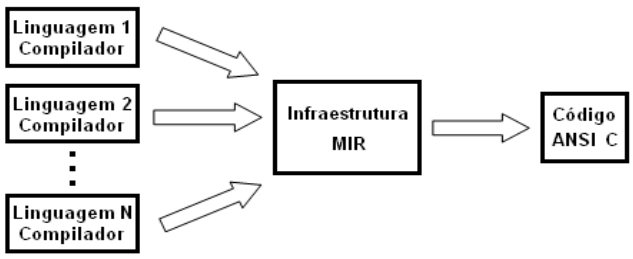


Figura 1: Contexto de MIR

Outro ponto importante sobre MIR é o fato de ser projetado para ser otimizado. As otimizações propostas no projeto MIR não sobrepõem as otimizações comumente realizadas por um compilador C, pois elas são otimizações que levam em consideração as características do modelo ASM.

A Infraestrutura MIR são todas as classes e componentes de software que compõe a infraestrutura implementada. A expressão Arquitetura MIR se refere à estrutura geral de uma especificação ASM usando a Infraestrutura MIR, podendo ser vista como a “linguagem” da infraestrutura. E, se a Arquitetura MIR é a linguagem, uma Especificação MIR é um “programa” escrito nessa linguagem.

2.1. Agentes, Modelos e outros elementos

Uma especificação MIR, como mostrada na Figura 2, é composta por agentes, cada um de um tipo, e por um Espaço de Nomes Globais comum a cada um dos agentes que pertencem à especificação MIR. O Espaço de Nomes Globais é uma tabela para as ações e funções estáticas, derivadas, dinâmicas e externas. Nesta tabela é possível identificar se a informação é uma ação ou função, assim como o tipo da função. Funções estáticas são aquelas cujos valores não mudam ao executar as regras de atualização. Tais funções são definidas por expressões parametrizadas e permanecem inalteradas durante toda a execução da arquitetura MIR. Não é permitido chamar funções dinâmicas de dentro das definições das funções estáticas.

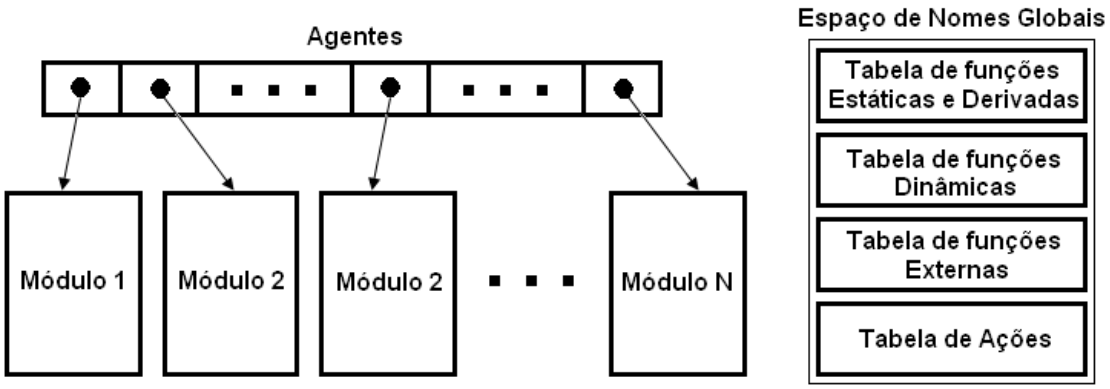


Figura 2: Arquitetura MIR

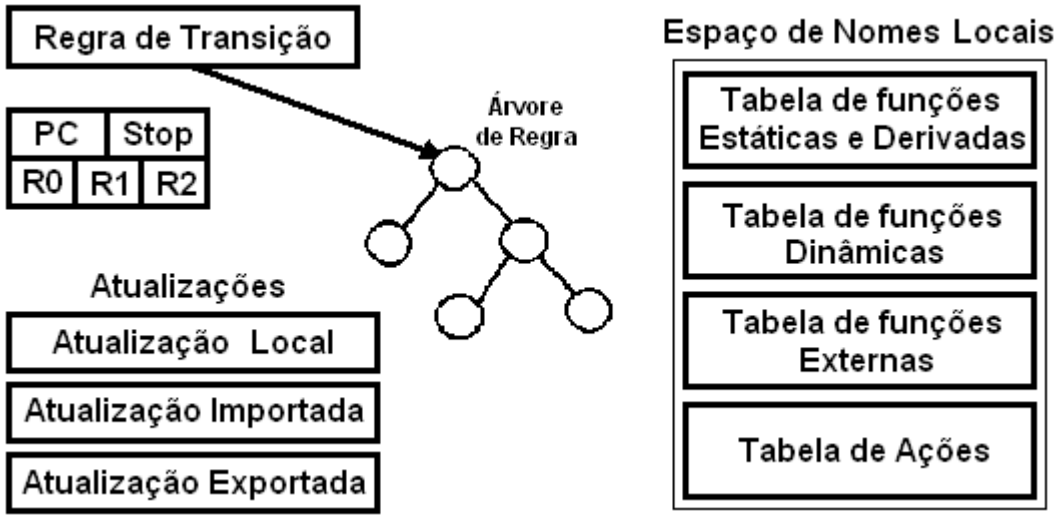


Figura 3: Um módulo de MIR

A Figura 1 mostra o processo de compilação de um programa em Machĭna, onde L_{MIR} representa a linguagem intermediária. L_{MIR} é uma linguagem simples, estruturada como uma árvore binária, cujas instruções possuem características próximas às instruções da linguagem fonte desta compilação, Machĭna. Ou seja, instruções MIR tem características de programação ASM.

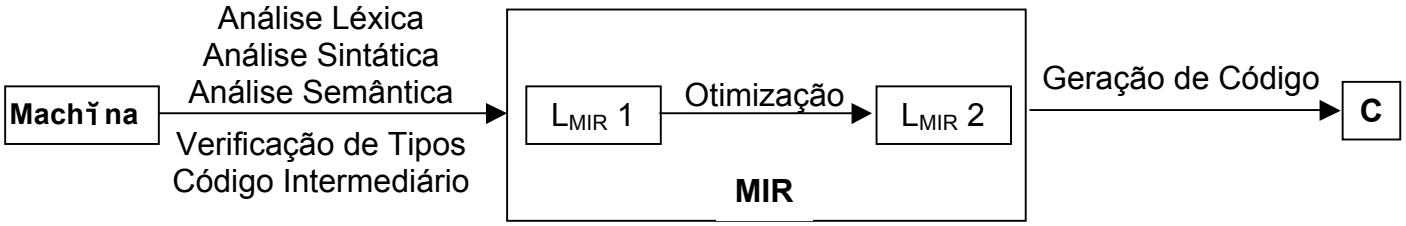


Figura 1: Compilador de Machĭna

Um programa em linguagem L_{MIR} denota uma regra de transição, que pode ser representada diretamente por estruturas de dados na memória. Existem duas estruturas de dados principais: uma árvore de nomes, com informações oriundas da tabela de símbolos gerada pelo compilador durante as fases de análise sintática e semântica, e uma árvore cujos nodos são as instruções MIR, que serão apresentadas nas Seções 3 e 4, respectivamente.

1.1 Componentes da arquitetura abstrata de MIR

Para ilustrar o uso dos componentes da arquitetura abstrata de L_{MIR} serão considerados os seguintes exemplos de código Machĭna: **v := 10 + z** e **if achou then v + 5 else 1**. Podemos ver os componentes da arquitetura abstrata de L_{MIR} para estes exemplos na Figura 2.

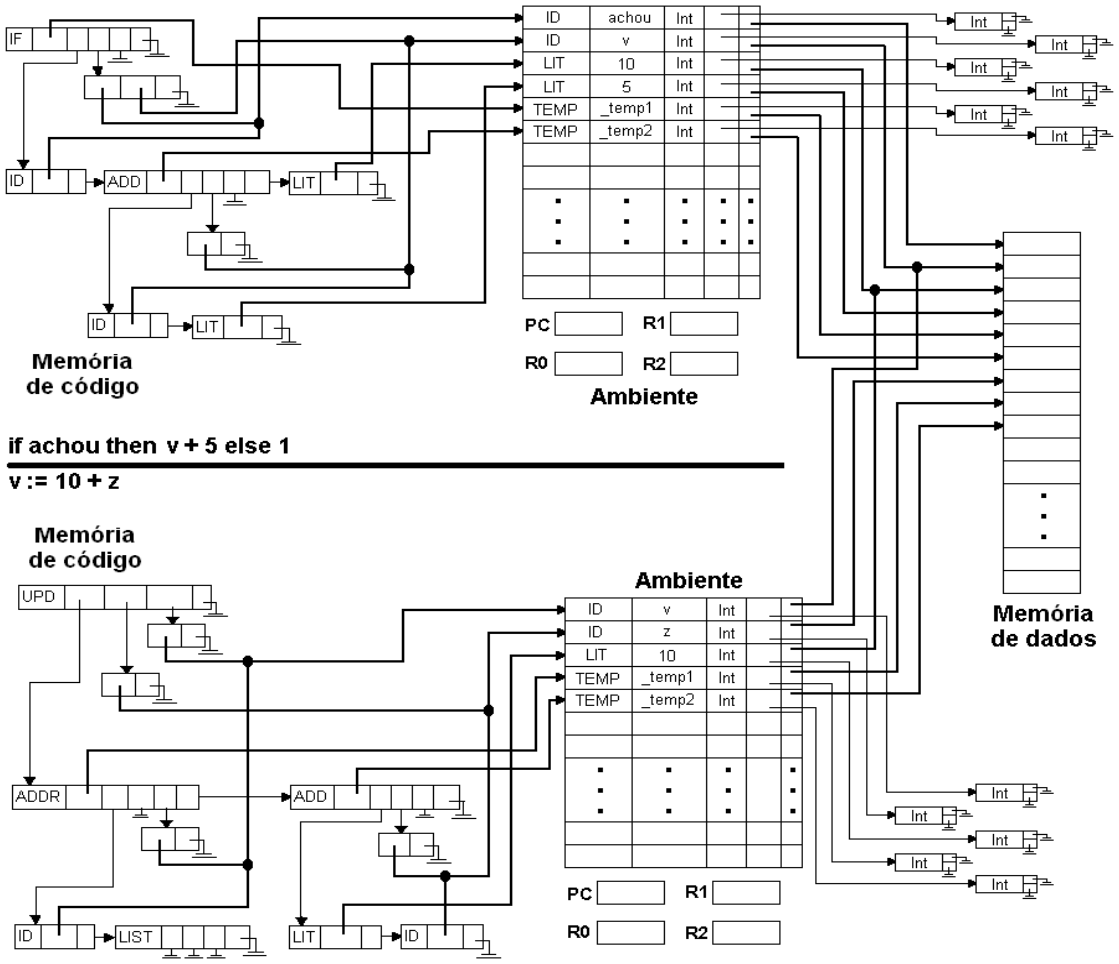


Figura 2: Componentes L_{MIR}

(1) Memória de dados (MD)
(2) Memória de código (MC)
(3) Vetor de Estado
(4) Registradores auxiliares PC, R0, R1, R2
(5) Pilha de contadores de programa, usado para caminhar na árvore na ordem de execução das instruções
(6) Um ambiente, que mapeia funções ASM intermediárias em seus valores

Memória de dados

A memória de dados, representada por uma pilha, armazena valores alocados estaticamente ou dinamicamente pelo gerador de código MIR. Esta região da memória contém os valores manipulados pelas instruções MIR, composta por palavras de quatro bytes. Cada tipo de dado tem uma representação nessa área de memória:

(1) **Int**: representada por uma palavra na pilha de dados.
(2) **Char**: representada por uma palavra na pilha de dados.
(3) **Bool**: representada por uma palavra na pilha de dados.
(4) **Real**: representada por quatro palavras na pilha de dados.
(5) **String**: representada por uma palavra na pilha de dados para indicar o tamanho **n** da cadeia de caracteres, mais **n** palavras para os **n** caracteres da cadeia de caracteres.
(6) **Enumeração**: representada por uma palavra na pilha de dados para indicar o tamanho **n** da enumeração, mais **n** palavras para os **n** itens da enumeração.
(7) **Intervalo**: representada por duas palavras na pilha de dados A primeira indica o limite inferior do intervalo e a segunda indica o limite superior do intervalo.
(8) **União Disjunta**: representada pelo tipo que ocupa a maior quantidade de palavras dos tipos que compõe a união.
(9) **Tupla**: representada por uma palavra na pilha de dados para indicar o tamanho **n** da tupla, mais **n** palavras indicando os **n** endereços dos tipos que compõe a tupla, mais os **n** endereços dos tipos que compõe a tupla.
(10) **Lista**: representada por uma palavra na pilha de dados para indicar o tamanho **n** da lista, mais **n** palavras indicando os **n** endereços dos tipos que compõe a lista.
(11) **Conjunto**: representada por uma palavra na pilha de dados para indicar o tamanho **n** do conjunto, mais **n** palavras indicando os **n** endereços dos tipos que compõe o conjunto.
(12) **Função**: representada por duas palavras na pilha de dados. A primeira indica o endereço do primeiro tipo que compõe a função e a segunda indica o endereço do segundo tipo que compõe a função, mais os dois endereços dos tipos que compõe a função.

Memória de código

A memória de código armazena uma árvore representando o código MIR. As instruções MIR estão definidas na Seção 4.

Vetor de Estado

PC	Err	Stop	E1	E2	E3	address

Um recurso importante da arquitetura abstrata de MIR é o vetor de estado. Ele é utilizado para controlar o processamento das instruções e prover informações sobre o estado da máquina. Especificamente ele é usado para redirecionar o fluxo de controle, sendo formado por vários componentes. O principal componente do vetor de estado é a situação corrente de operação da máquina abstrata. Neste sentido ele possui os seguintes valores: (1) o valor normal indicando o fluxo de controle seqüencial, (2) outros valores para indicar exceções e condições de parada. Seu segundo componente é responsável por fornecer mais informações sobre o fluxo de controle não-seqüencial. Ele pode ser um valor para o modo de retorno ou um nome de exceção mais valores possíveis para o modo exceção.

O vetor de estado deve incluir também o contador de programa que contém o endereço do nodo contendo a próxima instrução a executar. Inicialmente ele contém o endereço do nodo raiz da árvore de estrutura MIR, representando a regra de transição do programa.

Encontrando um erro durante a execução do programa MIR o campo correspondente no vetor de estado é marcado e a execução da regra de transição continua. Chegando no fim da execução da regra verifica-se o vetor de estado antes de recomeçar uma nova execução. Se o vetor de estado estiver no modo normal a execução continua. Caso contrário a execução da regra é interrompida, conforme a configuração do vetor de estado.

Registradores auxiliares

MIR contará com quatro registradores especiais que serão usados para descrever o efeito das instruções:

- PC: contador de programa, contendo o endereço da próxima instrução a ser executada, que será MC[PC].
- R0, R1, R2: registradores auxiliares para as instruções MIR.

Pilha de contadores de programa

Como cada instrução MIR tem uma forma específica de ser interpretada, pode ser necessário, em algum momento, guardar o valor atual do contador de programa para que ele passe a conter um novo valor, mas de forma que o valor que foi guardado possa ser recuperado.

Ambiente

Essa estrutura armazena um conjunto com funções da tabela de símbolos necessárias para as fases de otimização e de geração de código final. Um nodo dessa árvore consiste das seguintes informações: o nome o símbolo; classe do que pode ser uma função ou um literal; o tipo associado do símbolo, e o local na memória onde seu valor pode ser encontrado.

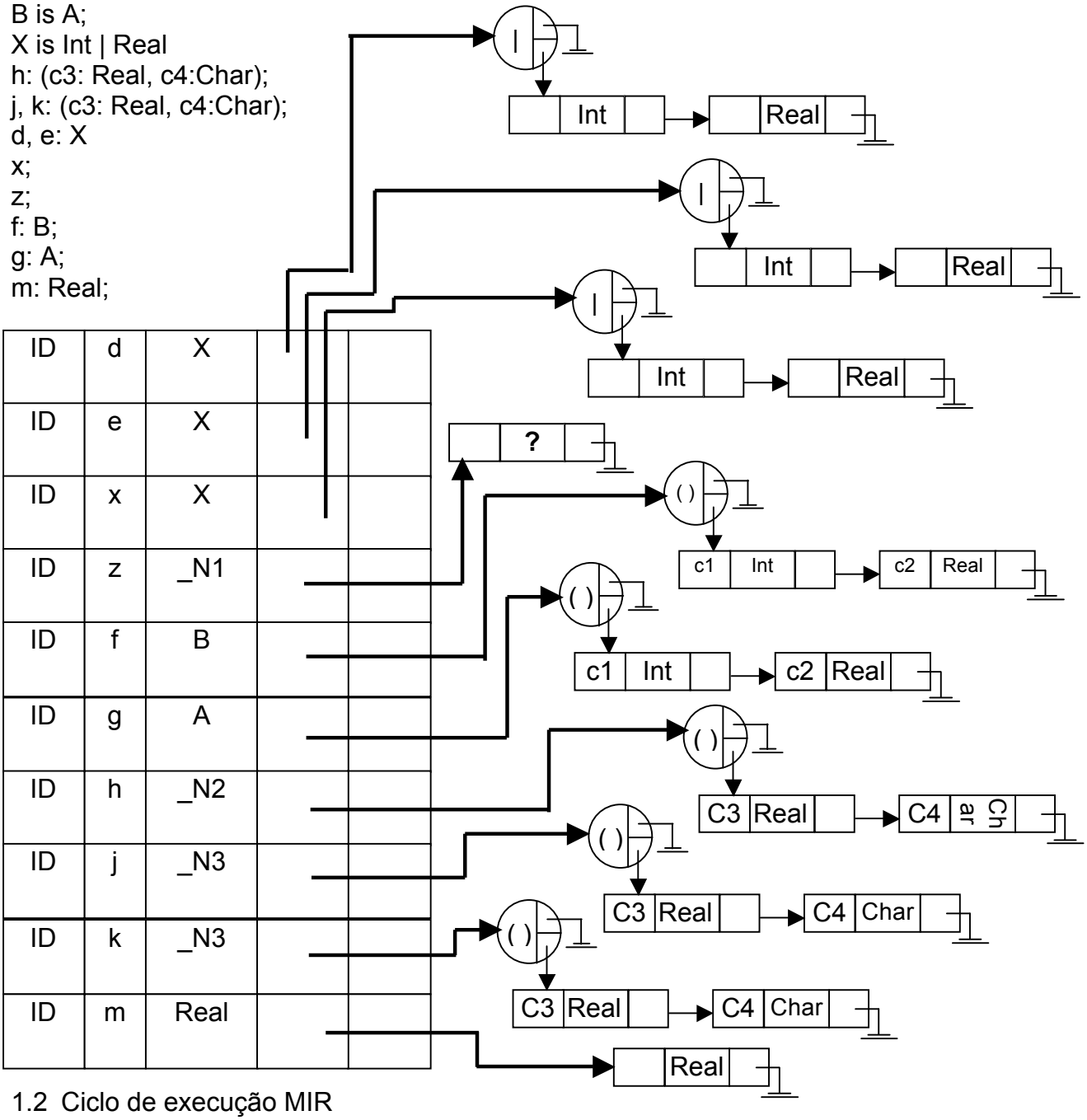
Info type	Name	Data type name	Data type definition	Definition

Figura 2: Dados de uma célula de Vocabulário

O campo **Info type** indica o tipo da informação guardada nesta célula: literal, função, temporário. O campo **Name** indica o nome da informação guardada. O campo **Data type name** indica o nome do tipo do dado armazenado, como por exemplo: (1) int, string, bool, para tipos básicos; (2) A, B, C, para tipos compostos; (3) _N1, _N2, _N3, para tipos compostos sem nome definido pelo usuário. O campo **Data type definition** indica a estrutura do tipo do dado armazenado. Esse campo indica uma árvore n-ária na qual os nodos folha representam os tipos básicos e os nodos internos são construtores de tipos tais como tuplas (produto cartesiano), listas, conjuntos, funcionalidade (mapeamento). Os nodos da árvore podem ter nomes rotulados. O campo **Definition** indica o endereço do símbolo armazenado na memória de dados.

Exemplo:

```
A is (c1:Int, c2:Real);
B is A;
X is Int | Real
h: (c3: Real, c4:Char);
j, k: (c3: Real, c4:Char);
d, e: X
x;
z;
f: B;
g: A;
m: Real;
```



ID	d	X		
ID	e	X		
ID	x	X		
ID	z	_N1		
ID	f	B		
ID	g	A		
ID	h	_N2		
ID	j	_N3		
ID	k	_N3		
ID	m	Real		

1.2 Ciclo de execução MIR

A execução de um programa MIR inicia-se a partir da árvore de estrutura MIR, realizando os seguintes passos:

(a) executa o nodo raiz
(b) processa a lista de atualização
(c) verifica vetor de estado
(d) reinicia o processo

A execução de um programa MIR começa pelo nodo raiz da árvore MIR que representa a regra de transição. Neste momento o contador de programa aponta para este nodo. Durante essa execução são encontrados nodos de

comandos que devem ser avaliados de acordo com a descrição da seção 3. Quando o contador de programa indicar um nodo nulo, a execução atual chegou ao fim.

Uma particularidade importante da linguagem MIR é a existência de dois tipos distintos de comandos de atribuição. Um deles indica uma atualização no local em que o comando for encontrado (**local_UPD**). O outro indica uma inclusão em uma lista de comandos de atualização (**pos_UPD**), sendo que essa lista somente será executada no fim da regra de transição. Quando durante a execução da árvore forem encontrados comandos de atualização **pos_UPD**, essas atualizações serão atrasadas, para serem concluídas somente no fim da regra de transição.

Devido à existência de atualizações **pos_UPD**, quando a execução da árvore chega ao fim ainda existem atualizações pendentes, que se encontram em uma lista, a lista de atualização. Nesse momento deve-se executar estas atualizações, para que uma nova execução da regra de transição possa acontecer. Após processar a lista de atualização será necessário recomeçar a execução da regra de transição.

Como a execução que terminou pode gerar erros, ou mesmo pode ter executado o comando de parada da execução, antes de recomeçar é preciso verificar o vetor de estado desta execução, conforme descrito na seção 1.1.

3. Representações em MIR

3.1. Árvore de Estrutura

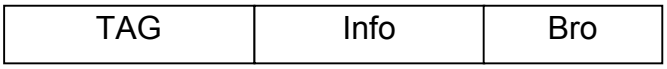


Figura 3: Dados de uma célula da Árvore de Estrutura

A **Árvore de Estrutura** armazena todos os dados do código intermediário. Cada instrução MIR constitui um nodo da **Árvore de Estrutura**. Um nodo dessa árvore consiste em três campos denominados **TAG**, **Info** e **Bro**, conforme mostra a Figura 3.
O primeiro campo, **TAG**, indica o tipo da regra que está sendo armazenada.
O segundo campo, **Info**, contém informações auxiliares dependentes do TAG. Ele é apresentado em mais detalhes na Seção 3.1.1. O terceiro campo, **Bro**, é um apontador para um nodo do tipo **Árvore de Estrutura**. Dependendo do contexto em que ocorre o nodo pode designar a próxima regra que deve ser executada, ou um componente da estrutura representada.

3.1.1. Campo Info

O segundo campo da **Árvore de Estrutura**, **Info**, é composto por uma nova estrutura, responsável por armazenar informações importantes para a regra e que são utilizadas nas fases de otimização e geração de código final. As informações contidas nesse campo variam de acordo com o tipo da regra a ser executada. Portanto a estrutura deste campo pode ser de três formas. A

Figura 4 ilustra estas estruturas: (a) Tipo Interno Expressão; (b) Tipo Interno Regra; (c) Tipo Folha.

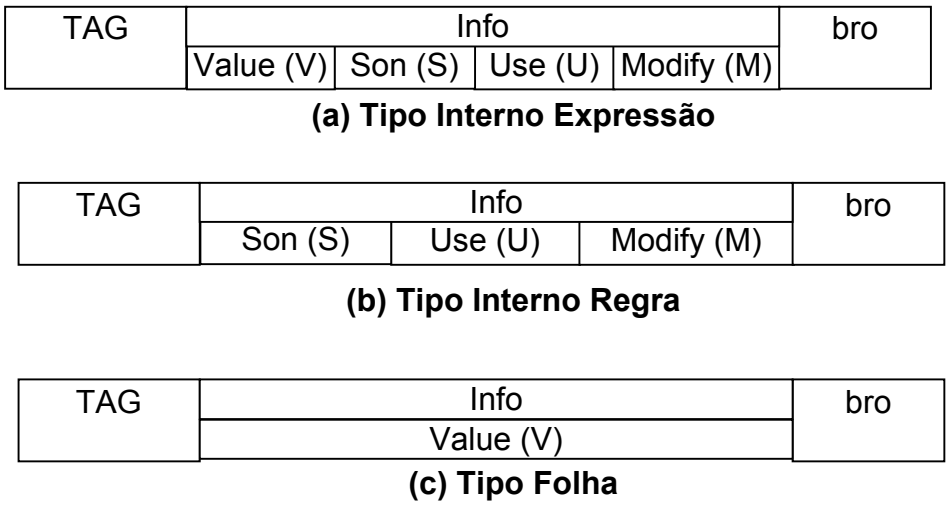


Figura 4: Possibilidades do campo Info da Árvore de Estrutura

(a) Tipo Interno Expressão

O primeiro campo, **Value**, aponta para a entrada do símbolo na estrutura do tipo **Vocabulário**, Figura 3, caso se refira a um símbolo. O segundo campo, **Son**, é um apontador para um nodo do tipo **Árvore de Estrutura**. Este campo aponta para o primeiro nodo que define a regra que está sendo representada. O terceiro campo, **Use**, é um apontador para uma lista de apontadores para estruturas do tipo **Vocabulário**. Esta lista contém os endereços dos símbolos utilizados nessa regra. O quarto campo, **Modify**, é um apontador para uma lista de apontadores para estruturas do tipo **Vocabulário**. Esta lista contém os endereços dos símbolos modificados nessa regra.

(b) Tipo Interno Regra

O primeiro campo, **Son**, é um apontador para um nodo do tipo **Árvore de Estrutura**. Este campo aponta para o primeiro nodo que define a regra que está sendo representada. O segundo campo, **Use**, é um apontador para uma lista de apontadores para estruturas do tipo **Vocabulário**. Esta lista contém os endereços dos símbolos utilizados nessa regra. O terceiro campo, **Modify**, é um apontador para uma lista de apontadores para estruturas do tipo **Vocabulário**. Esta lista contém os endereços dos símbolos modificados nessa regra.

(c) Tipo Folha

O campo **Value** aponta para a entrada do símbolo na estrutura do tipo **Vocabulário**, caso se refira a um símbolo.

4. Tipos de nodos em MIR

Nesta seção são apresentadas as **Árvores de Estrutura** para cada tipo de nodo em MIR, seguido de uma explicação sobre o que significa cada um dos campos na estrutura. A estrutura genérica de um nodo da **Árvore de Estrutura** foi apresentada na Figura X.

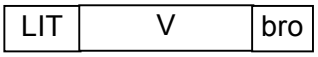
Esta seção também inclui um exemplo para cada construção introduzida. No exemplo, quando o campo de informação denotar um apontador para uma célula do vocabulário, será utilizado o símbolo "⇒" acrescido do nome do vocabulário para o qual tal campo estará apontando. Se for um apontador para um temporário será usado "⇒ temp".

4.1. Expressões

As expressões são classificadas em expressão literal, identificador, expressão unária, expressão binária.
As expressões usam para o campo de informações da Árvore de Estrutura Tipo Folha e Tipo Interno Expressão,

4.1.1. Expressão Literal

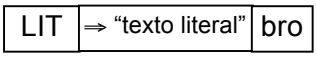
Árvore de Estrutura:



onde:
LIT: TAG do nodo
V: aponta para o literal instalado no vocabulário

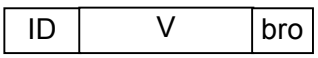
Interpretação semântica: O compilador Machĭna, ao encontrar um literal, busca seu valor na estrutura vocabulário e retorna o valor encontrado.

Exemplo: "texto literal"



4.1.2. Expressão com Identificador

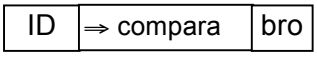
Árvore de Estrutura:



onde:
ID: TAG do nodo
V: aponta para o identificador instalado no vocabulário

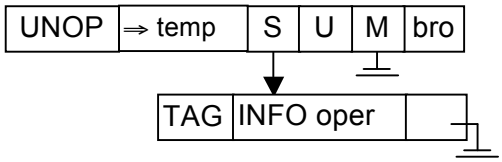
Interpretação semântica: O compilador Machĭna, ao encontrar um identificador, busca seu valor na estrutura vocabulário e retorna o valor encontrado.

Exemplo: compara



4.1.3. Expressão com operador unário {-,not,(type-name)}

Árvore de Estrutura:



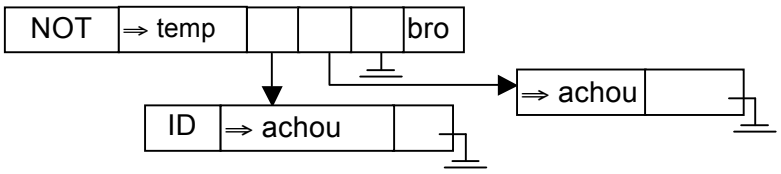
onde: UNOP: TAG do primeiro nodo
Cada UNOP terá um TAG: NEG, NOT, TYPENAME

- V: aponta para a área que contem o resultado da operação unária
- S: referencia o nodo MIR que denota o operando do UNOP
- U: lista das funções usadas pelo operador unário, que serão as mesmas funções usadas no operando.
- M: Nulo.

Interpretação semântica: O valor da expressão é o resultado da aplicação do operador designado pelo UNOP ao seu operando. O compilador Machĭna, ao encontrar um operador unário UNOP busca o valor do operando, aplica o operador e coloca o resultado na célula do vocabulário do nodo UNOP. O código a seguir ilustra seu funcionamento.

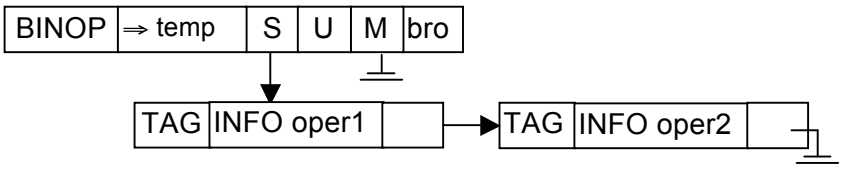
```
Empilha PC->Bro
Empilha PC
PC ← PC->Son
Avalia PC
Desempilha PC
Aplica UNOP a PC->Son->Value->Definition
Guarda resultado em PC->Value->Definition
Desempilha PC
```

Exemplo: not achou



4.1.4. Expressão com operador binário {*, /, %, +, -, =, !=, <, >, <=, >=, and, or, xor, ::, in, is}

Árvore de Estrutura:



onde: BINOP: TAG do primeiro nodo
Cada BINOP terá um TAG: EQ, NEQ, LS, GR, LEQ, GEQ, ADD, SUB, MULT, DIV, MOD, AND, OR, XOR, IN, IS

V: aponta para a área que contem o resultado da operação binária

S: Faz referencia a uma lista com dois nodos MIR, encadeados pelo campo **brother**, que denotam os operandos do BINOP.

U: Lista de funções usadas pelo operador binário, que serão as mesmas funções usadas nos dois operandos do BINOP.

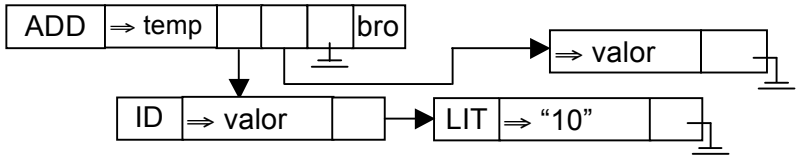
M: Nulo.

Lista de nodos em son: Essa lista é composta por dois nodos: **son** do nodo BINOP, primeiro operando da operação binária, e **brother** do **son** do nodo BINOP, segundo operando da operação binária.

Interpretação semântica: O valor da expressão é o resultado da aplicação do operador designado por BINOP aos seus dois operandos. O código a seguir mostra a seqüência de passos executados para os operadores binários.

```
Empilha PC->Bro;                  Empilha PC
Empilha PC->Son;                  PC ← PC->Son->Bro
Avalia PC;                        R2 ← PC->Value->Definition
Desempilha PC;                    Avalia PC
R1 ← PC->Value->Definition        Desempilha PC
Aplica BINOP a R1 e R2
Guarda resultado em PC->Value->Definition
Desempilha PC
```

Exemplo: valor + 10



4.1.5. Expressão com agregado

No agregado as expressões podem ser de dois tipos: agregados de conjunto e agregados de lista.

Árvore de Estrutura:

1) {expression1, expression2, ..., expressionN}

2) [expression1, expression2, ..., expressionN]

onde: AGR {} ou AGR []: TAG do primeiro nodo

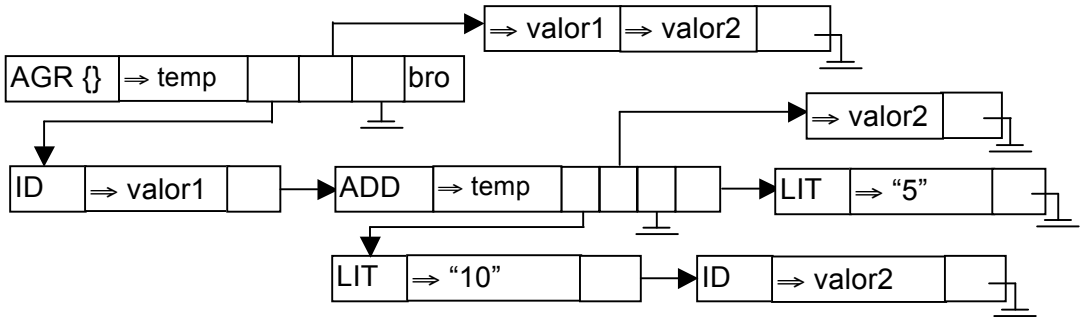
- S: Faz referencia a uma lista com vários nodos MIR, encadeados pelo campo **brother**, que denotam os componentes de AGR
- U: Lista de funções usadas pelo operador AGR, que serão as mesmas funções usadas nos componentes do agregado.
- M: Nulo.

Lista de nodos em son: O primeiro nodo que compõe essa lista está indicado pelo campo **son** do nodo AGR. A partir desse primeiro nodo, caminhando pelo campo **bro**, é possível encontrar os outros componentes.

Interpretação semântica: O valor da expressão é o resultado de agregar os componentes desse operador. O código a seguir mostra a seqüência de passos executados para o agregado.

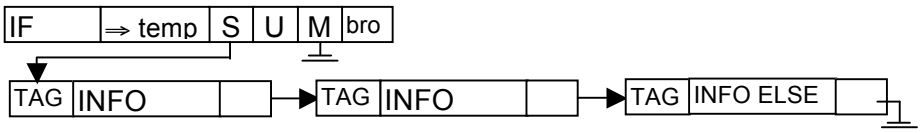
```
Empilha PC->Bro              R0 ← PC->Value->Definition
PC ← PC->Son                 i ← 1
enquanto PC não é nulo
    Avalia PC
    Armazena PC->Value->Definition
                em R0 na posição i
    PC ← PC->Bro
    i ← i + 1
fim enquanto
Desempilha PC
```

Exemplo: {valor1, 10 + valor2, 5}



4.1.6. Expressão condicional IF

Árvore de Estrutura:



onde:

IF: TAG do primeiro nodo

S: Faz referencia a uma lista com três nodos MIR, encadeados pelo campo **bro**, que denotam os três componente do IF: a condição e as clausulas THEN e ELSE.

U: Lista de funções usadas pelo condicional, que serão as mesmas funções usadas em cada um dos três componentes do condicional.

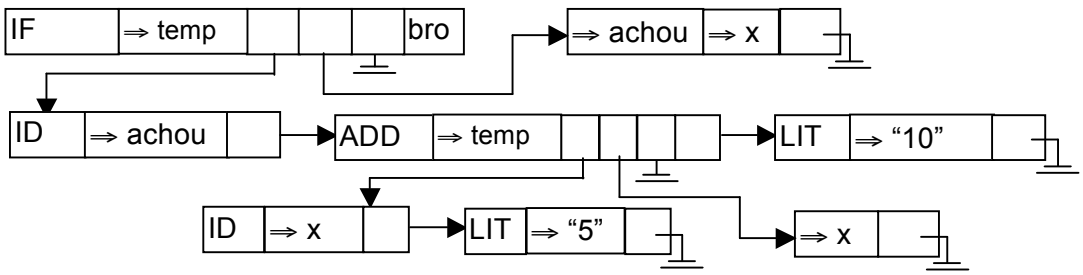
M: Nulo.

Lista de nodos em son: Essa lista é composta por três nodos: o primeiro componente, condição do IF, está indicado pelo campo **son** do nodo IF; o segundo componente, clausula THEN, está indicado pelo campo **bro** do campo **son** do nodo IF; o terceiro componente, clausula ELSE, está indicado pelo campo **bro** do campo **bro** do campo **son** do nodo IF.

Interpretação semântica: O valor de uma expressão IF é o resultado da avaliação do componente THEN ou do componente ELSE. Para encontrar o resultado deve-se executar o comando IF da seguinte forma: o primeiro componente da estrutura, condição do IF, deve ser avaliado. Se o resultado dessa avaliação for verdadeiro, o segundo componente, THEN, é avaliado, sendo este o resultado da expressão, e o terceiro componente, ELSE, é ignorado. Mas se o resultado da avaliação do primeiro componente for falso, o segundo componente, THEN, é ignorado e o terceiro componente, ELSE, é avaliado, sendo este o resultado da expressão. O código a seguir mostra a seqüência de passos executados para o IF.

```
Empilha PC->Bro;            Empilha PC
PC ← PC->Son;               Avalia PC
se PC->Value->Definition = verdadeiro então PC ← PC->Son
   senão    PC ← PC->Son->Bro
   fim se
Avalia PC;                  R0 ← PC->Value->Definition
Desempilha PC               Guarda R0 em PC->Value->Definition
Desempilha PC
```

Exemplo: if achou then x + 5 else 10

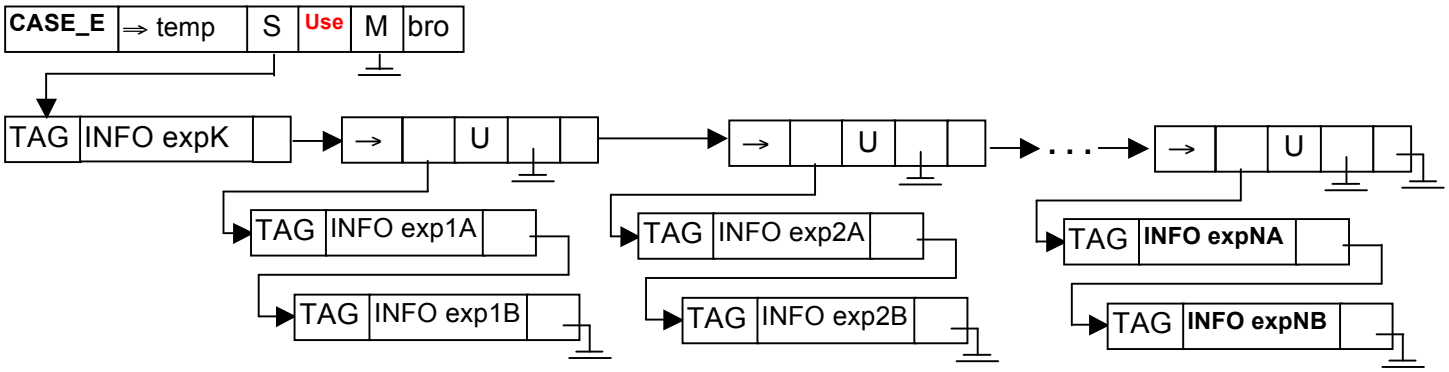


4.1.7. Expressão condicional CASE

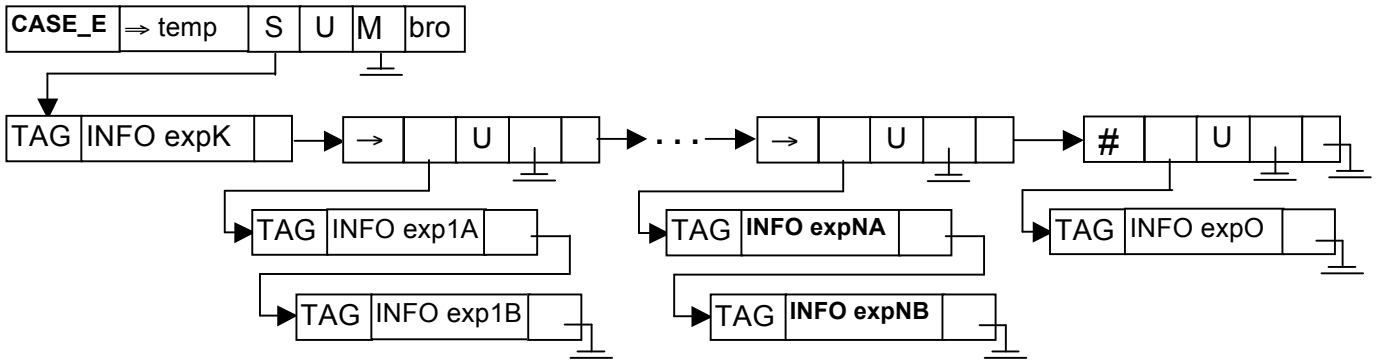
A expressão condicional **case** pode ser de dois tipos: sem cláusula otherwise ou com cláusula otherwise.

Árvore de Estrutura:

1) CASE sem cláusula otherwise



2) CASE com cláusula otherwise



onde:

CASE_E:	TAG do primeiro nodo
S:	Faz referencia a uma lista de nodos MIR, encadeados pelo campo **bro**. Esses nodos denotam os componentes de CASE_E.
U de →:	Endereço da lista de funções usadas nos 2 componentes de →.
U de #:	Endereço da lista de funções usadas no componente de #.
U de CASE_E:	Endereço da lista de funções usadas nos componentes de CASE_E.
M:	Nulo.

Lista de nodos em son de CASE_E: O primeiro nodo dessa lista denota a expressão principal para comparação e está indicado pelo campo **son** do

nodo CASE_E. A partir desse primeiro nodo, caminhando pelo campo **bro**, é possível encontrar os outros componentes do CASE, que denotam os componentes com TAG → ou #. Os nodos → referenciam, através do campo **son**, uma lista com dois nodos MIR, encadeados pelo campo **bro**. O nodo #, se existir, referencia, através do campo **son**, um nodo MIR.

Lista de nodos em son de →: Essa lista é composta por dois nodos: (1) o primeiro componente, que denota uma expressão para ser comparada com a expressão principal de comparação, está indicado pelo campo **son** do nodo → e (2) o segundo componente, que denota uma expressão a ser executada caso o primeiro nodo referente a esse segundo nodo seja compatível com a expressão principal de comparação, está indicado pelo campo **bro** do campo **son** do nodo →.

Nodo em son de #: Esse nodo representa a condição "otherwise" e é executado sempre que as comparações dos componente → anteriores não forem verdadeiras. Um nodo # só pode aparecer como o último nodo da lista de componentes de CASE.

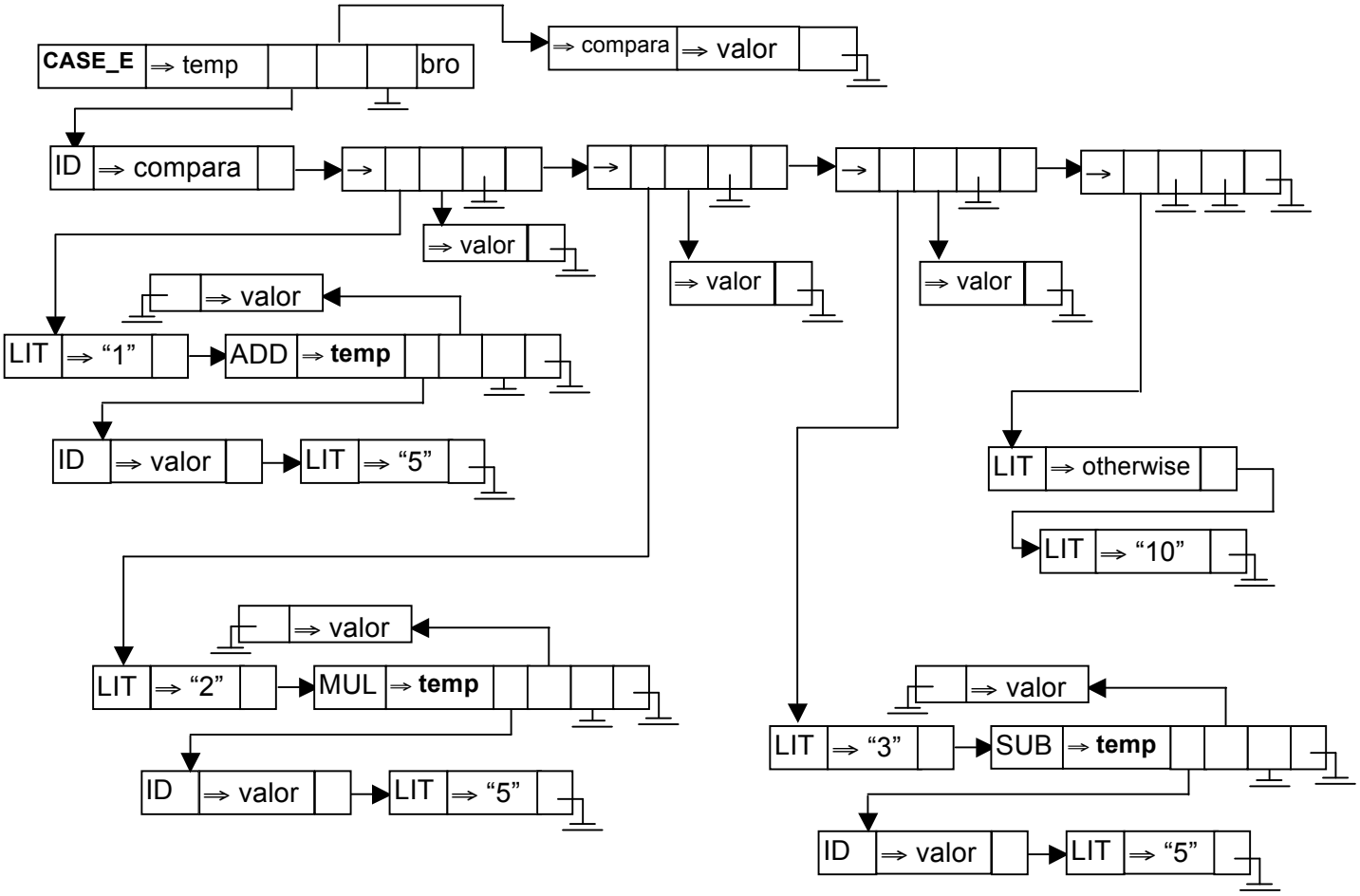
Interpretação Semântica: O valor de uma expressão CASE é o resultado da avaliação de um dos segundos componentes da lista de dois nodos que compõe um nodo →, ou do nodo componente do nodo de TAG #. Para encontrar o resultado deve-se executar o comando CASE_E da seguinte forma: o primeiro componente da estrutura, condição do CASE_E, deve ser avaliado e o resultado dessa avaliação é utilizado para comparar com os outros componentes do CASE_E. Deve-se, então, caminhar pelo campo **bro** das estruturas, a partir do segundo componente. No componente atual do caminhamento, nodo com TAG →, deve-se avaliar o primeiro de seus dois componentes. Se o resultado encontrado com esta avaliação for igual ao resultado da avaliação do primeiro componente de CASE_E, o segundo componente da estrutura de TAG → deve ser avaliado, sendo esse o resultado da operação, terminando a avaliação de CASE_E. Se o resultado encontrado com esta avaliação for diferente do resultado da avaliação do primeiro componente de CASE_E, o segundo componente do nodo → deve ser ignorado e o caminhamento pelo campo **bro** deve continuar. Se o componente atual do caminhamento for um nodo com TAG # é porque o caminhamento alcançou o último componente sem encontrar nenhuma comparação igual ao primeiro componente de CASE_E. Com isso o componente **son** desse nodo # deve ser avaliado, sendo esse o resultado da operação, terminando a avaliação de CASE_E. O código a seguir ilustra a atualização do campo PC do vetor de estados para a expressão case.

```
Empilha PC->Bro;                                    Empilha PC
PC ← PC->Son;                                       Avalia PC
R1 ← PC->Value->Definition;                         achou ← falso
enquanto (PC não é nulo) e (não achou)
    Empilha PC
    se TAG de PC é #
        PC ← PC->Son;                               Avalia PC
        R0 ← PC->Value->Definition
        achou ← verdadeiro;                         Desempilha PC
```

```
        senão o TAG é →
                PC ← PC->Son;                       Avalia PC
                R2 ← PC->Value->Definition
                se R1 = R2 então
                        PC ← PC->Son;               Avalia PC
                        R0 ← PC->Value->Definition
                         achou ← verdadeiro;        Desempilha PC
                senão  Desempilha PC;               PC ← PC->Bro
                fim se
        fim se
fim enquanto
Desempilha PC
se (achou)  então    Guarda R0 em PC->Value->Definition
fim se
Desempilha PC
```

Exemplo:

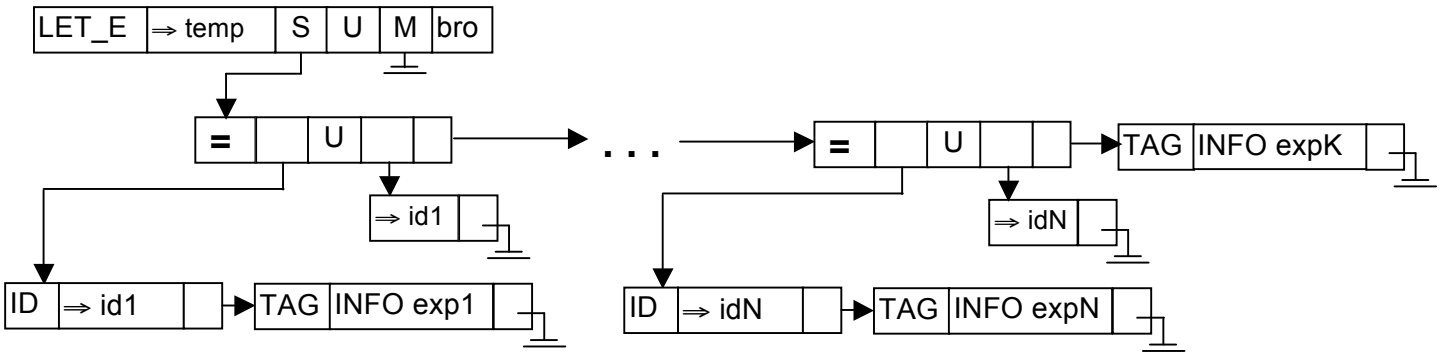
```
case compara
    1 → valor + 5
    2 → valor * 5
    3 → valor – 5
    otherwise 10
end
```



4.1.8. Expressão LET

```
let      id1 = expression1;
          id2 = expression2;
                ...
          idN = expressionN;
        IN expressionK
end
```

Árvore de Estrutura:



onde:

- LET_E: TAG do primeiro nodo
- S: Faz referencia a uma lista de nodos MIR, encadeados pelo campo **bro**. Esses nodos denotam os componentes de LET_E.
- U de **=**: Endereço da lista de funções usadas no segundo nodo da lista de dois nodos do componente **=**.
- M de **=**: Endereço da lista com a função definida no primeiro nodo da lista de dois nodos do componente **=**.
- U de LET_E: Endereço da lista de funções usadas nos segundos nodos da lista de dois nodos dos componentes **=** e no último nodo da lista de LET_E, subtraído do conjunto de identificadores definidos na cláusula LET_E, que são as funções definidas nos primeiros nodos da lista de dois nodos dos componentes **=**.
- M: Nulo.

Lista de nodos em son de LET_E: Essa lista contêm **n+1** nodos. Os **n** primeiros nodos dessa lista tem o TAG **=**, e tem como **son** uma lista com dois nodos MIR, encadeados pelo campo **bro**. O primeiro dos componentes de LET_E está indicado pelo campo **son** do nodo LET_E, e os próximos nodos são encontrados caminhando pelo campo **bro**. O último nodo da lista de

componentes de LET_E denota uma expressão a ser executada, onde serão utilizados os identificadores definidos nos **n** primeiros nodos.

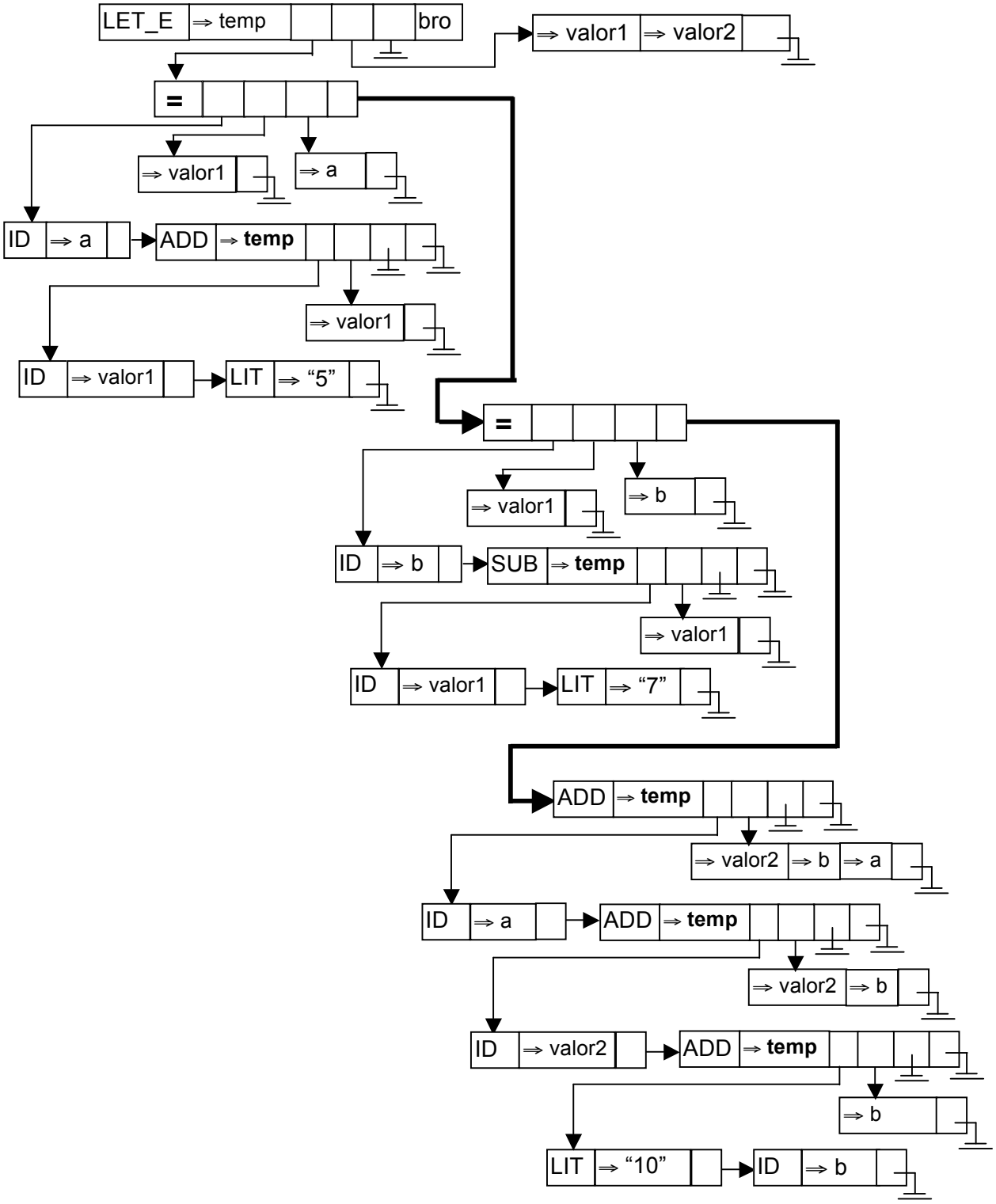
Lista de nodos em son de =: Essa lista é composta por dois nodos: o primeiro componente, que denota um identificador, está indicado pelo campo **son** do nodo **=**, e o segundo componente, que denota uma expressão associada ao identificador do primeiro nodo, está indicado pelo campo **bro** do campo **son** do nodo **=**.

Interpretação semântica: O valor de uma expressão LET é o resultado da avaliação do último componente da lista de LET_E. Para encontrar o resultado deve-se executar o comando LET_E da seguinte forma: para os **n** primeiros nodos componentes de LET_E, nodos com TAG **=**, deve-se associar o identificador, primeiro componente da lista de dois nodos do nodo com TAG **=**, ao valor da expressão, segundo componente da lista de dois nodos do nodo com TAG **=**. Então, o valor da expressão é o resultado da avaliação do último componente da lista de LET_E, levando-se em consideração a associação de identificadores feita. O código a seguir mostra a seqüência de passos executados para o LET.

```
Empilha PC->Bro;        Empilha PC;         PC ← PC->Son
enquanto (PC->Bro não é nulo)
    Empilha PC;             PC ← PC->Son                Empilha PC
    Avalia PC;              R1 ← PC->Value->Definition
    Desempilha PC;          PC ← PC->Bro;               Avalia PC
    Atribui PC->Value->Definition a R1
    Desempilha PC;          PC ← PC->Bro
fim enquanto
Avalia PC, considerando os valores associados
R0 ← PC->Value->Definition;             Desempilha PC
Guarda R0 em PC->Value->Definition
Desempilha PC
```

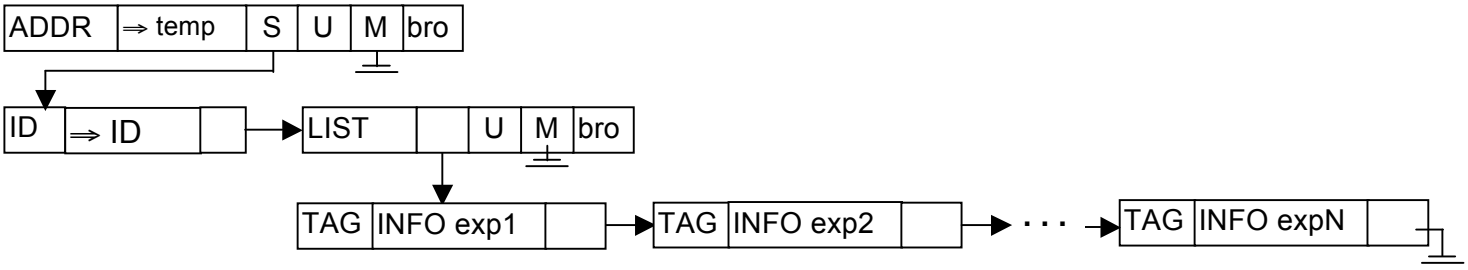
Exemplo:

```
let   a = valor1 + 5
      b = valor1 – 7
      in a + valor2 + 10 + b
end
```


LET_E
⇒ temp
bro
⇒ valor1
⇒ valor2
=
⇒ valor1
⇒ a
ID
⇒ a
ADD
⇒ temp
⇒ valor1
ID
⇒ valor1
LIT
⇒ “5”
=
⇒ valor1
⇒ b
ID
⇒ b
SUB
⇒ temp
⇒ valor1
ID
⇒ valor1
LIT
⇒ “7”
ADD
⇒ temp
⇒ valor2
⇒ b
⇒ a
ID
⇒ a
ADD
⇒ temp
⇒ valor2
⇒ b
ID
⇒ valor2
ADD
⇒ temp
⇒ b
LIT
⇒ “10”
ID
⇒ b

4.1.9. Expressão ID[(expr1, expr2, ..., exprN)]

Árvore de Estrutura:



onde:

ADDR: TAG do primeiro nodo

S: Faz referencia a uma lista de dois nodos MIR, encadeados pelo campo **bro**, que denotam os componentes de ADDR. O segundo desses dois nodos, LIST, é composta por uma lista de nodos MIR, encadeados pelo campo **bro**.

U de LIST: Endereço da lista de funções usadas nos nodos da lista do componente LIST.

U de ADDR: Endereço da lista de funções usadas no nodo LIST mais a função ID.

M: Nulo.

Lista de nodos em son de ADDR: Essa lista contêm dois nodos: o primeiro tem o TAG ID e está indicado pelo campo **son** do nodo ADDR, e o segundo nodo tem o TAG LIST e está indicado pelo campo **bro** do campo **son** do nodo ADDR.

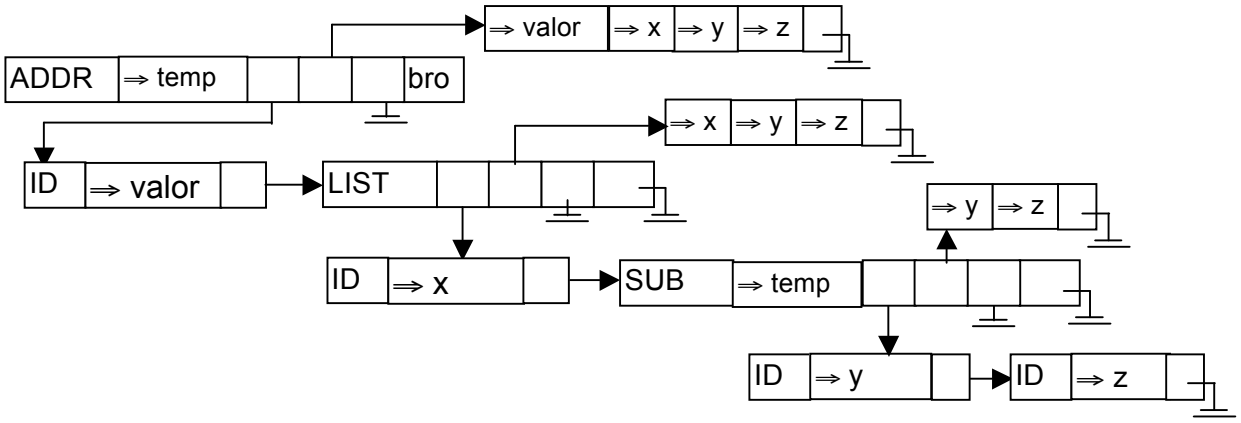
Lista de nodos em son de LIST: Essa lista é composta por nodos em que o primeiro componente está indicado pelo campo **son** do nodo LIST e os próximos componentes estão indicados caminhando pelo campo **bro**.

Interpretação semântica: O valor de uma expressão ADDR é o resultado da avaliação do componente ID nos pontos indicados pela lista de expressões LIST. Para encontrar o resultado deve-se executar o comando ADDR da seguinte forma: avaliar as expressões dos nodos componentes de LIST, e, então, o valor da expressão é o resultado da avaliação do primeiro componente de ADDR nos pontos encontrados pelas expressões de LIST. O código a seguir mostra a seqüência de passos executados para essa expressão.

```
Empilha PC->Bro;                Empilha PC;                PC ← PC->Son;     Avalia PC
R0 ← PC->Value->Definition      PC ← PC->Bro->Son ;        i ← 1
enquanto PC não é nulo
    Avalia PC                   Armazena PC->Value->Definition em R0 na posição i
    PC ← PC->Bro;               i ← i + 1
fim enquanto
```

Desempilha PC	PC->Value->Definition ← R0	Desempilha PC

Exemplo: valor(x , y - z)

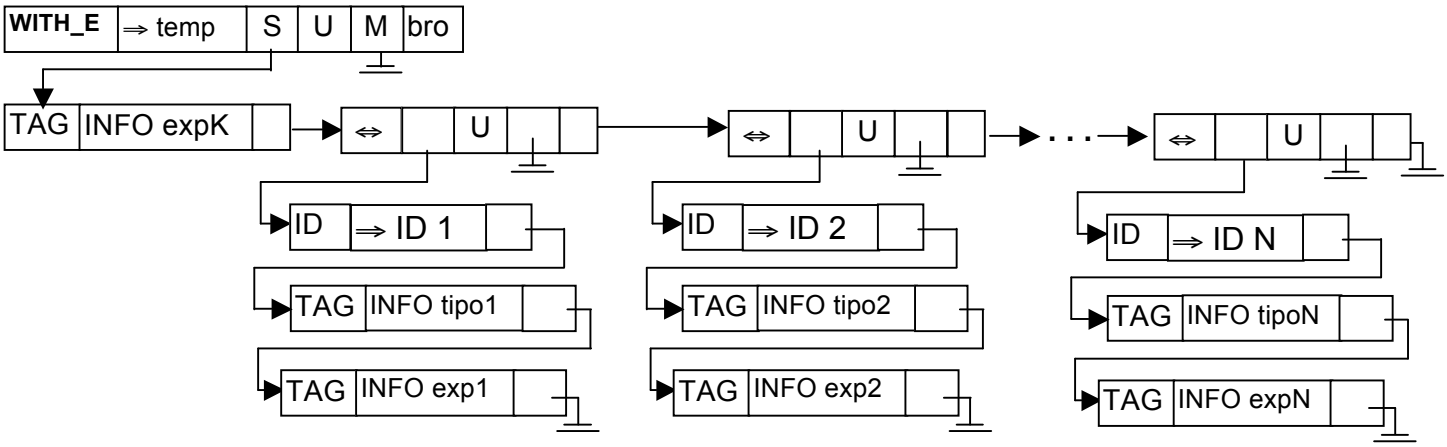


4.1.10. Expressão WITH

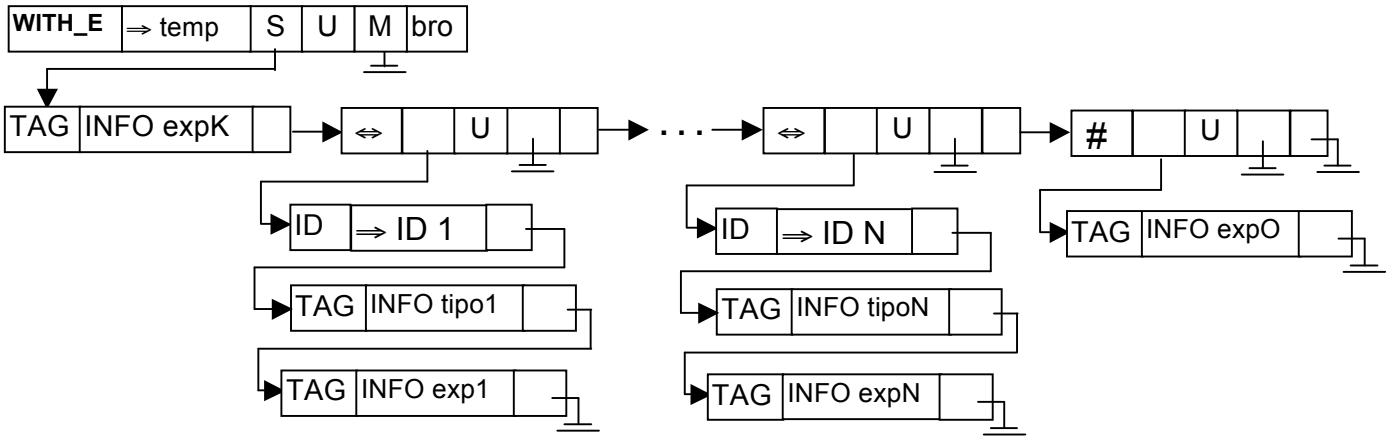
WITH expressionK AS
    id1 : Tn1 → expression1
    id2 : Tn2 → expression2
    ...
    idN : TnN → expressionN
    otherWise → expressionOtherwise
end

A expressão with pode ser de dois tipos: sem cláusula otherwise ou com cláusula otherwise.

1) WITH sem cláusula otherwise



2) WITH com cláusula otherwise



Campo	Significado
TAG do 1° nodo	WITH_E
son	Referencia uma lista de nodos MIR, encadeados pelo campo **brother**. Esses nodos denotam os componentes de WITH.
U de ⇔	Endereço da lista de funções usadas nos componentes de ⇔.
U de #	Endereço da lista de funções usada no componente de #.
U de WITH_E	Endereço da lista de funções usadas nos componentes de WITH_E.
M	Nulo

Lista de nodos em son de WITH_E: O primeiro nodo dessa lista denota a expressão principal para comparação e está indicado por **son** do nodo WITH_E. A partir desse primeiro nodo, caminhando pelo campo **brother**, é possível encontrar os outros componentes do WITH, que denotam os componentes com TAG ⇔ ou #. Os nodos ⇔ referenciam, através do campo **son**, uma lista com três nodos MIR, encadeados pelo campo **brother**. O último nodo pode ter o TAG ⇔ descrito anteriormente ou o TAG #, que contém um nodo MIR.

Lista de nodos em son de ⇔: Essa lista é composta por três nodos: (1) o primeiro componente denota um identificador e está indicado por **son** do nodo ⇔; (2) o segundo componente denota um tipo para ser comparado com o tipo da expressão principal de comparação e está indicado por **brother** do **son** do nodo ⇔; e (3) o terceiro componente denota uma expressão a ser executada, com o identificador do primeiro nodo associado à expressão principal de comparação, no caso do tipo no segundo nodo ser compatível com o tipo da expressão principal de comparação. O terceiro componente está indicado por **brother** do **brother** do **son** do nodo ⇔.

Nodo em son de #: Esse nodo representa a condição “otherwise” e será executado sempre que as comparações de tipo dos componente ⇔ anteriores não forem verdadeiras. Um nodo # só pode aparecer como o último nodo da lista de componentes de WITH.

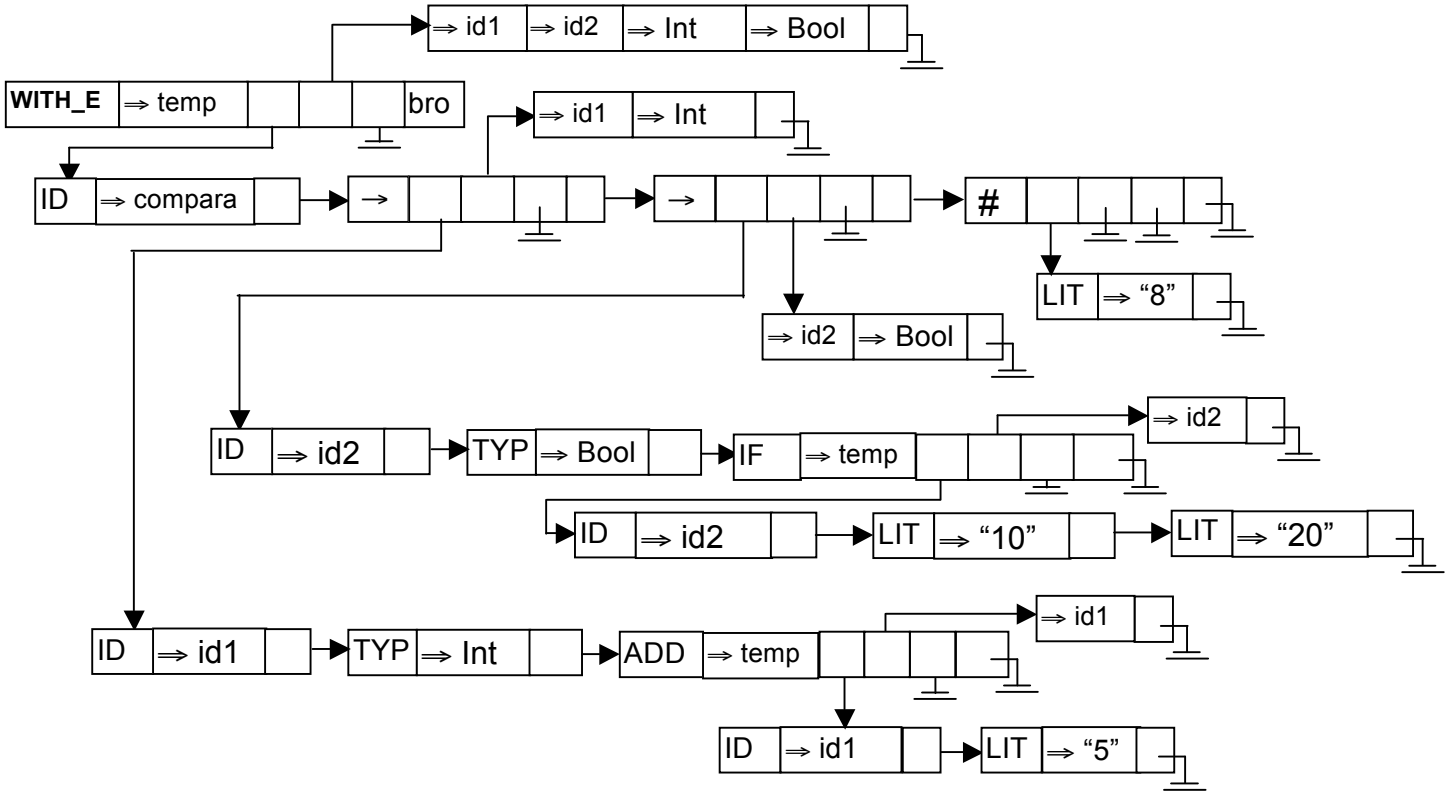
Interpretação semântica: O valor de uma expressão WITH é o resultado da avaliação de um dos terceiros componentes da lista de três nodos que compõe um nodo ⇔, ou do componente que compõe um nodo #. Para encontrar o resultado deve-se executar o comando WITH_E da seguinte forma: O primeiro componente da estrutura, condição do WITH_E, deve ser avaliado. Tanto o tipo do resultado dessa avaliação quanto o próprio resultado serão utilizados. O tipo do resultado dessa avaliação será comparado com os tipos dos outros componentes do WITH_E. Deve-se, então, caminhar pelo campo **brother** das estruturas, a partir do segundo componente. No componente atual do caminhamento, nodo com TAG ⇔, deve-se avaliar o segundo de seus componentes. Se o tipo encontrado neste nodo for igual ao tipo do resultado da avaliação do primeiro componente de WITH_E, o terceiro componente da estrutura de TAG ⇔ deverá ser avaliado, sendo esse o resultado da operação, terminando a avaliação de WITH_E. Na avaliação desse terceiro componente o identificador do primeiro nodo deve estar associado à expressão principal de comparação. Se o tipo encontrado com a avaliação do segundo componente for diferente do tipo do resultado da avaliação do primeiro componente de WITH_E, o terceiro componente do nodo ⇔ deverá ser ignorado e o caminhamento pelo campo **brother** deve continuar. Se o componente atual do caminhamento for um nodo com TAG # é porque o caminhamento alcançou o último componente sem encontrar nenhuma comparação de tipo igual ao tipo do primeiro componente de WITH_E. Com isso o componente referente a esse nodo deverá ser avaliado, sendo esse o resultado da operação, terminando a avaliação de WITH_E.

```
Empilha PC->Bro;                    Empilha PC
PC ← PC->Son;                       Avalia PC
R1 ← PC->Value->Definition;         achou ← falso
enquanto (PC não é nulo) e (não achou)
  Empilha PC
  se TAG de PC é #      PC ← PC->Son
                        Avalia PC
                        R0 ← PC->Value->Definition
                        achou ← verdadeiro
                        Desempilha PC
  senão o TAG é ⇔       PC ← PC->Son
                        Empilha PC
                        PC ← PC->Bro
                        Avalia PC
                        R2 ← PC->Value->Definition
                        se (R2 é tipo) e (Tipo(R1) = R2) então
                              Desempilha PC
                              Avalia PC
                              R2 ← PC->Value->Definition
```

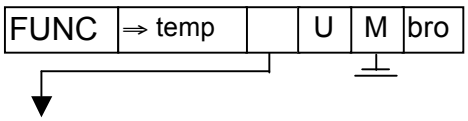
```
                                    Atribui R1 a R2
                                    PC ← PC->Bro->Bro
                                    Avalia PC
                                    R0 ← PC->Value->Definition
                                    achou ← verdadeiro
                                    Desempilha PC
                               senão
                                    Desempilha PC
                                    PC ← PC->Bro
                               fim se
  fim se
fim enquanto
Desempilha PC
se (achou) então      Guarda R0 em PC->Value->Definition
fim se
Desempilha PC
```

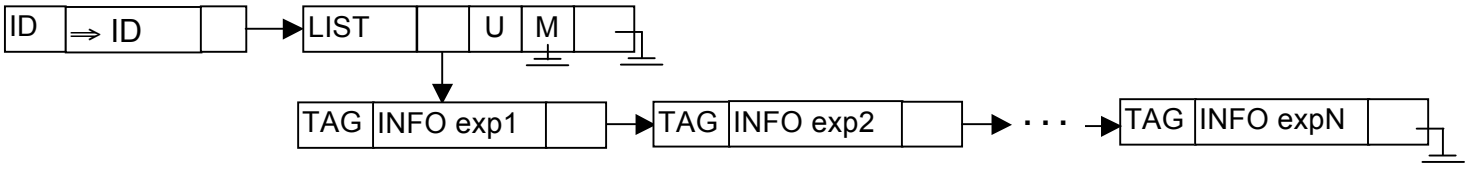
Exemplo:

```
with compara as
    id1 : Int → id1 + 5
    id2 : Bool → if id2 then 10 else 20 end
    otherwise 8
end
```



(K) Chamada de função ID[(expression1, expression2, ..., expressionN)]





Campo	Significado
TAG do 1° nodo	FUNC
son	Referencia uma lista de dois nodos MIR, encadeados pelo campo **brother**, que denotam os componentes de FUNC. O segundo desses dois nodos, LIST, é composta por uma lista de nodos MIR, encadeados pelo campo **brother**,
U de LIST	Endereço da lista de funções usadas nos nodos da lista do componente LIST.
U de FUNC	Endereço da lista de funções usadas em LIST mais a função ID.
M	Nulo

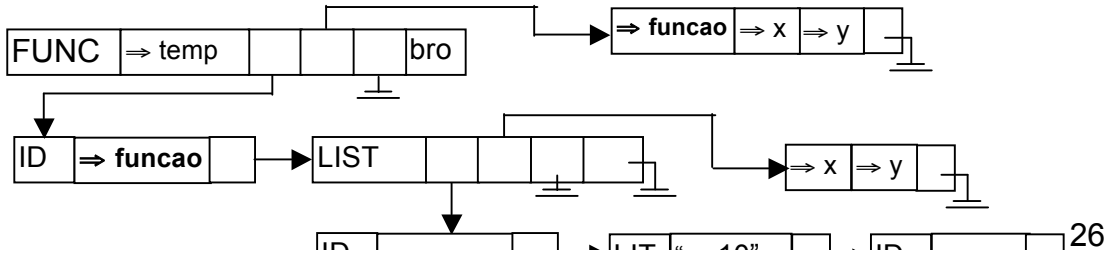
Lista de nodos em son de FUNC: Essa lista contêm dois nodos: o primeiro tem o TAG ID e está indicado por **son** do nodo ADDR, e o segundo nodo tem o TAG LIST e está indicado por **brother** do **son** do nodo FUNC.

Lista de nodos em son de LIST: Essa lista é composta por nodos em que o primeiro componente está indicado por **son** do nodo LIST e os próximos componentes estão indicados caminhando pelo campo **brother**.

Interpretação semântica: O valor de uma expressão FUNC é o resultado da avaliação do componente ID nos pontos indicados pela lista de expressões LIST. Para encontrar o resultado deve-se executar o comando FUNC da seguinte forma: Avaliar as expressões dos nodos componentes de LIST, Então, o valor da expressão é o resultado da avaliação do primeiro componente de FUNC nos pontos encontrados pelas expressões de LIST.

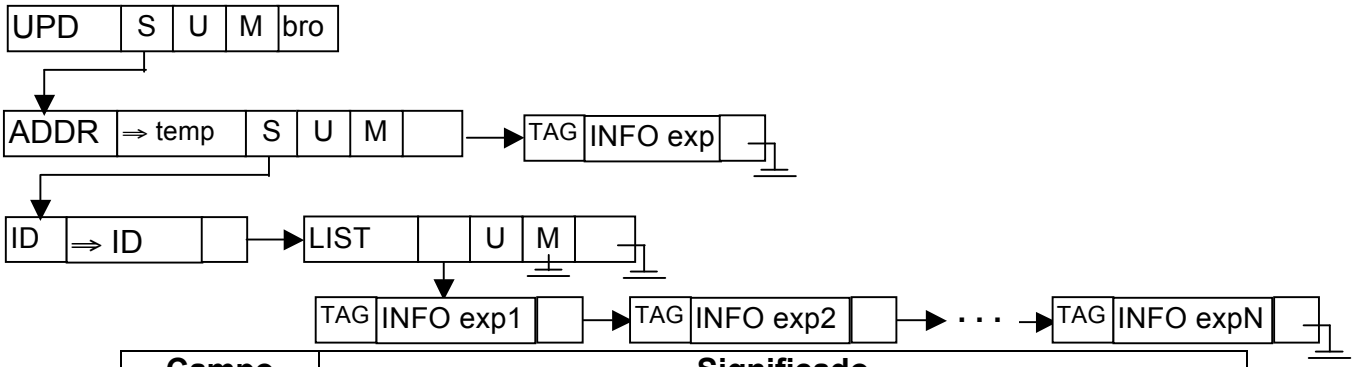
```
Empilha PC->Bro;                    PC ← PC->Son
R0 ← PC->Value->Definition
PC ← PC->Bro->Son;                  i ← 1
enquanto PC não é nulo
    Avalia PC
    Armazena PC->Value->Definition
                em R0 na posição i
    PC ← PC->Bro;                   i ← i + 1
fim enquanto
Desempilha PC
```

Exemplo: funcao(x , 10, y)



4.2. Regras

(A) id [(expression1, expression2, ..., expressionN)] := expression



Campo	Significado
TAG do 1° nodo	UPD
son	Referencia uma lista de dois nodos MIR, encadeados pelo campo **brother**, que denotam os componentes de UPD. O primeiro desses dois nodos, ADDR, é composta por uma lista de dois nodos MIR, encadeados pelo campo **brother**,
U de LIST	Endereço da lista de funções usadas nos nodos da lista do componente LIST.
U de ADDR	Endereço da lista de funções usadas no nodo LIST.
M de ADDR	Endereço da lista com a função ID.
U de UPD	Endereço da lista de funções usadas no nodo LIST mais as funções usadas no nodo de expression.
M de UPD	Endereço da lista de funções modificadas no nodo ADDR.

Lista de nodos em son de UPD: Essa lista contêm dois nodos: o primeiro tem o TAG ADDR, endereço a ser atualizado, e está indicado por **son** do nodo UPD, e o segundo nodo é a expressão para atualizar o endereço informado no nodo anterior e está indicado por **brother** do **son** do nodo UPD.

Lista de nodos em son de ADDR: Essa lista contêm dois nodos: o primeiro tem o TAG ID e está indicado por **son** do nodo ADDR, e o segundo nodo tem o TAG LIST e está indicado por **brother** do **son** do nodo ADDR.

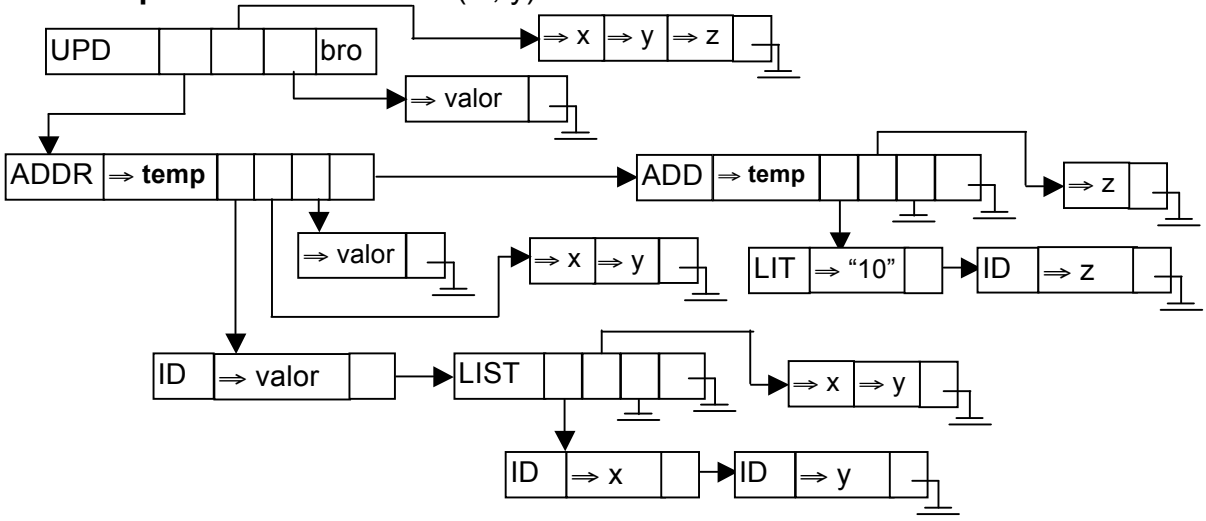
Lista de nodos em son de LIST: Essa lista é composta por nodos em que o primeiro componente está indicado por **son** do nodo LIST e os próximos componentes estão indicados caminhando pelo campo **brother**.

Interpretação semântica: Esse comando significa uma atualização de endereço. Para que a atualização ocorra é preciso avaliar o endereço para atualização assim como o valor a ser colocado no endereço encontrado. O endereço para atualização é indicado pela expressão ADDR, através do resultado da avaliação do componente ID nos pontos indicados pela lista de

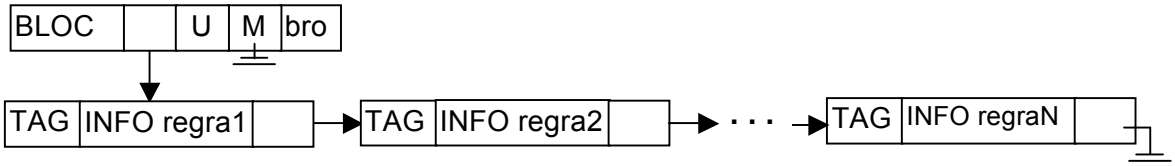
expressões LIST. Para encontrar o resultado deve-se executar o comando ADDR da seguinte forma: Avaliar as expressões dos nodos componentes de LIST, Então, o endereço para atualização é o resultado da avaliação do primeiro componente de ADDR nos pontos encontrados pelas expressões de LIST. Para calcular o valor a ser colocado no endereço encontrado é preciso avaliar a expressão que está no segundo nodo da lista de nodos de **son** do nodo UPD.

```
Empilha PC->Bro;                Empilha PC->Son
PC ← PC->Son->Bro
Avalia PC
Desempilha PC
Avalia PC
MD[PC->Value->Definition] ← PC->Bro->Value->Definition
Desempilha PC
```

Exemplo: valor(x , y) := 10 + z



(B) regra1, regra2, ... , regraN



Campo	Significado
TAG do 1° nodo	BLOC
son	Referencia uma lista com vários nodos MIR, encadeados pelo campo **brother**, que denotam as regras que compõem esse bloco de regras.
U	Endereço da lista de funções usadas nas regras que compõem esse bloco de regras.
M	Endereço da lista de funções modificadas nas regras que compõem esse bloco de regras.

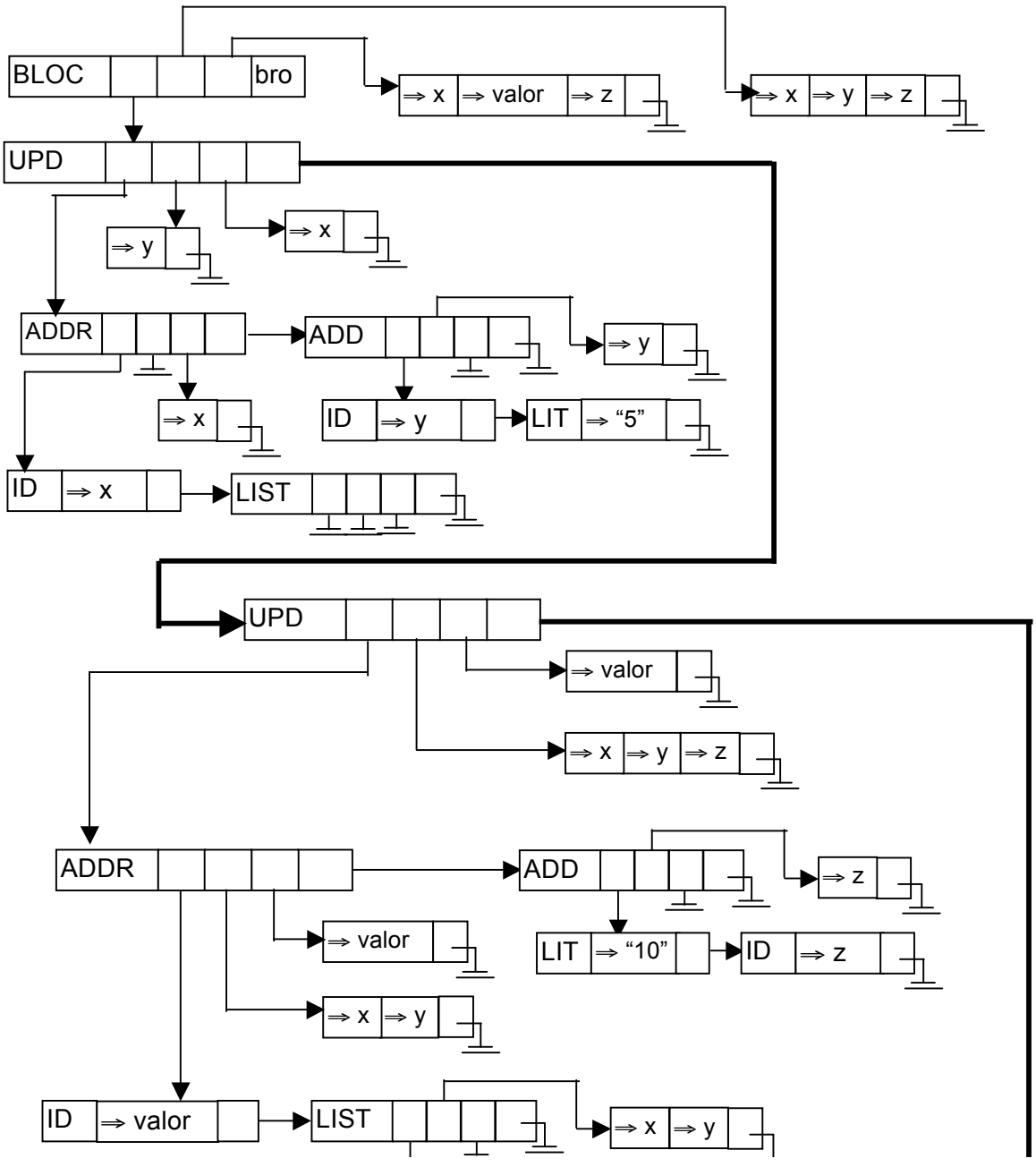
Lista de nodos em son de BLOC: Essa lista é composta por nodos em que o primeiro componente está indicado por **son** do nodo BLOC e os próximos componentes estão indicados caminhando pelo campo **brother**.

Interpretação semântica: Esse comando significa um conjunto de regras ASM. Essas regras são paralelas, ou seja, poderão ser executadas em qualquer ordem.

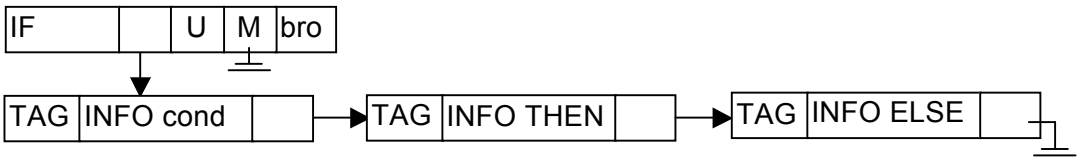
Empilha PC->Bro;	PC ← PC->Son
enquanto PC não é nulo	
Avalia PC;	PC ← PC->Bro
fim enquanto	
Desempilha PC	

Exemplo:

x := y + 5
valor(x , y) := 10 + z
z := z – 7



(C) condicional IF



Campo	Significado
TAG do 1° nodo	IF
son	Referencia uma lista com três nodos MIR, encadeados pelo campo **brother**, que denotam os três componente do IF: a condição e as clausulas THEN e ELSE.
U	Lista de funções usadas pela condição do IF e pelas clausulas THEN e ELSE.
M	Lista de funções modificadas pelas clausulas THEN e ELSE.

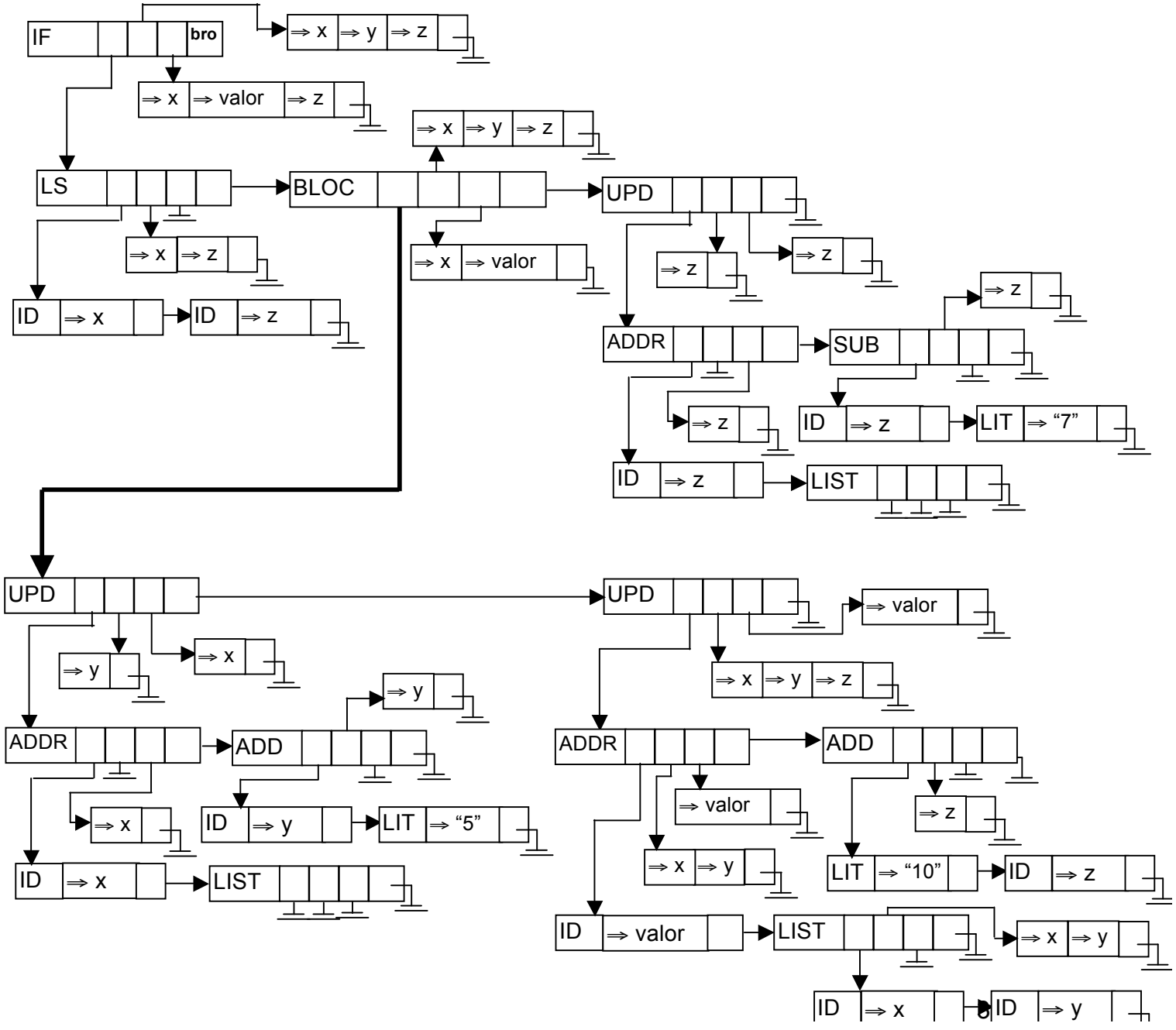
Lista de nodos em son: Essa lista é composta por três nodos: o primeiro componente, condição do IF, está indicado por **son** do nodo IF; o segundo componente, clausula THEN, está indicado por **brother** do **son** do nodo IF; o terceiro componente, clausula ELSE, está indicado por **brother** do **brother** do **son** do nodo IF.

Interpretação semântica: Executar a regra IF significa executar as regras da clausula THEN ou as regras da clausula ELSE. Para encontrar qual grupo de regras executar, deve-se executar o comando IF da seguinte forma: O primeiro componente da estrutura, condição do IF, deve ser avaliado. Se o resultado dessa avaliação for verdadeiro, as regras do segundo componente, THEN, serão executadas, e as regras do terceiro componente, ELSE, serão ignoradas. Mas se o resultado da avaliação do primeiro componente for falso, as regras do segundo componente, THEN, serão ignoradas e as regras do terceiro componente, ELSE, serão executadas.

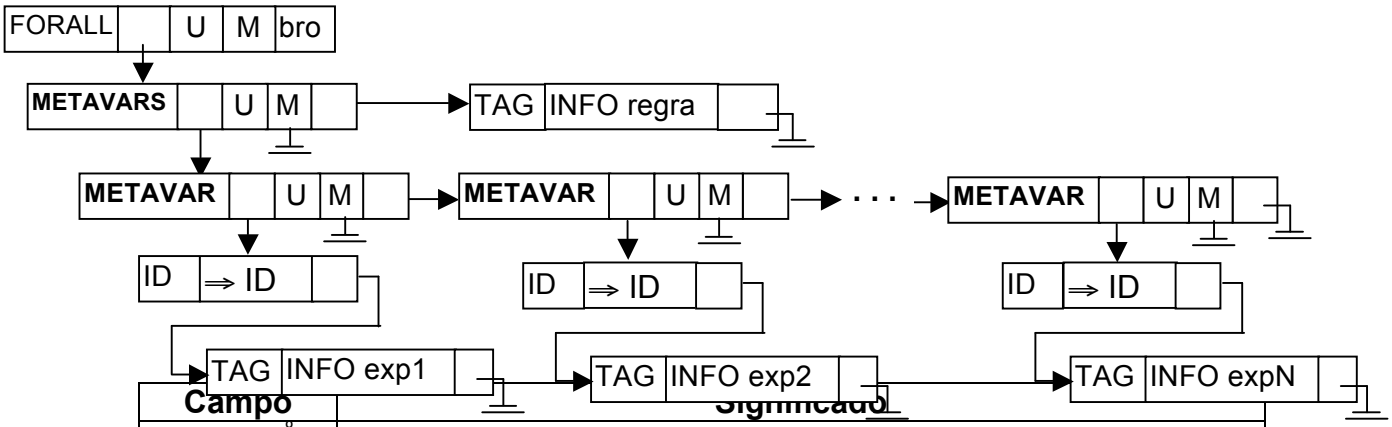
```
Empilha PC->Bro
PC ← PC->Son
Avalia PC
se PC->Value->Definition = verdadeiro então
   PC ← PC->Son
senão
   PC ← PC->Son->Bro
fim se
Avalia PC
Desempilha PC
```

Exemplo:

```
if x < z then
   x := y + 5
   valor(x , y) := 10 + z
else
   z := z – 7
end
```



(D) FORALL exp1, exp2, ..., expK do regra end



Campo	Significado
TAG do 1° nodo	FORALL
son	Referencia uma lista de dois nodos MIR, encadeados pelo campo **brother**, que denotam os componentes de FORALL. O primeiro, METAVARS, é composto por uma lista de nodos MIR do tipo METAVAR, encadeados pelo campo **brother**,
U de METAVAR	Endereço da lista de funções usadas no nodo filho de METAVAR.
U de METAVARS	Endereço da lista de funções usadas nos nodos da lista do componente METAVARS.
U de FORALL	Endereço da lista de funções usadas no nodo METAVARS mais as funções usadas na regra, segundo nodo da lista de dois nodos de **son** do nodo FORALL.
M de METAVAR	Nulo
M de METAVARS	Nulo
M de FORALL	Endereço da lista de funções modificadas na regra, segundo nodo da lista de dois nodos de **son** do nodo FORALL.

Lista de nodos em son de FORALL: Essa lista contém dois nodos: o primeiro nodo tem o TAG METAVARS e está indicado por **son** do nodo FORALL, e o segundo, que é a regra a ser executada, está indicado por **brother** do **son** do nodo FORALL

Lista de nodos em son de METAVARS: Essa lista é composta por nodos em que o primeiro componente está indicado por **son** do nodo METAVARS e os próximos componentes estão indicados caminhando pelo campo **brother**. Todos os nodos dessa lista têm TAG METAVAR.

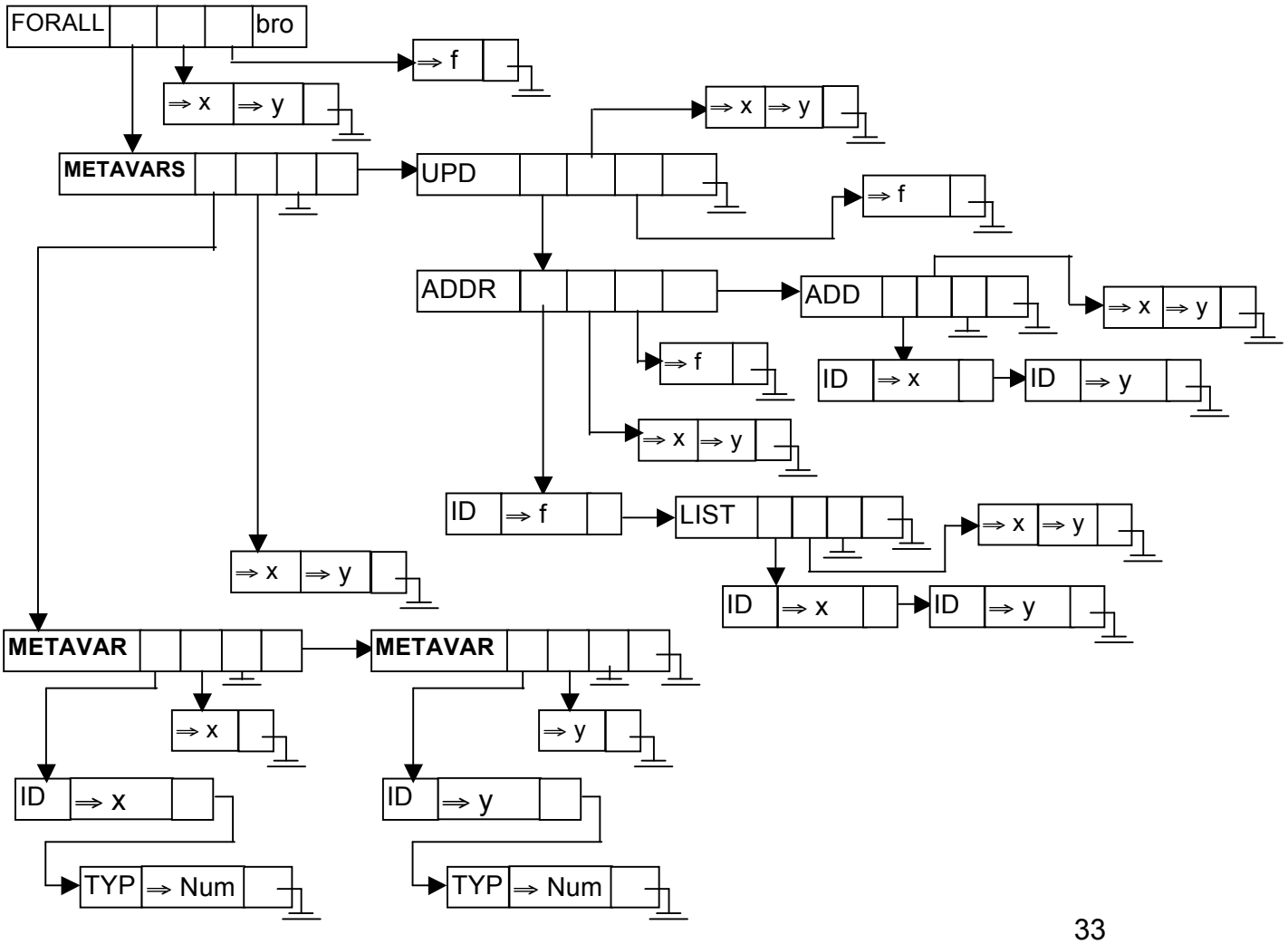
Nodo em son de METAVAR: É composto pelo nodo que representa um componente de METAVARS.

Interpretação semântica: Executar a regra FORALL significa executar as regras associadas à regra FORALL. Deve-se executar o comando FORALL da seguinte forma: Para cada grupo único de valores associado às expressões do FORALL, deve-se executar as regras associadas à regra FORALL, até que todas as possibilidades de grupo de valores associado às expressões do FORALL sejam avaliados.

```
Empilha PC->Bro;      Empilha PC->Son;      PC ← PC->Son->Son
Enquanto PC não é nulo
  Empilha PC;  PC ← PC->Son->Bro;  Avalia PC;  Desempilha PC
  Guarda PC->Son->Bro->Value->Definition em
              PC->Son->Value->Definition
  PC ← PC->Bro
fim enquanto
Desempilha PC;            PC ← PC->Bro
Avalia PC;                Desempilha PC
```

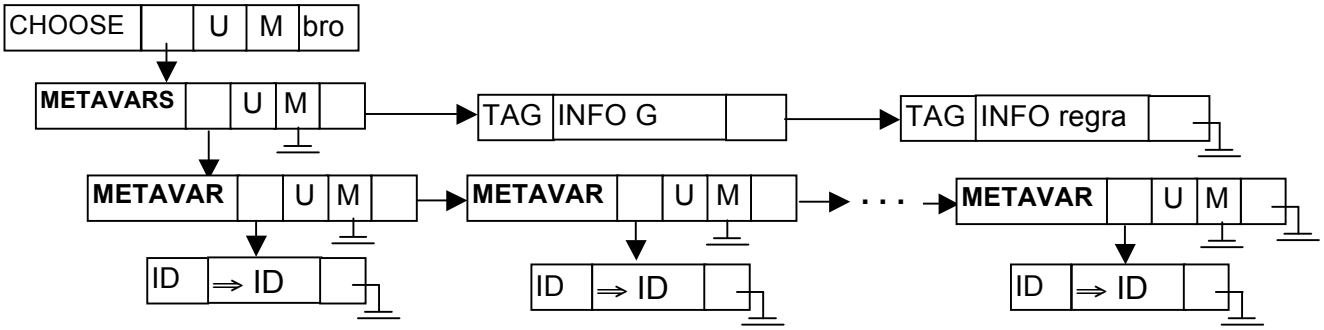
Exemplo:

```
type Num = 1 .. 3;
dynamic f(Int, Int):Int;
transition
  forall x:Num, y:Num do
    f(x, y) := x + y;
  end
end
```

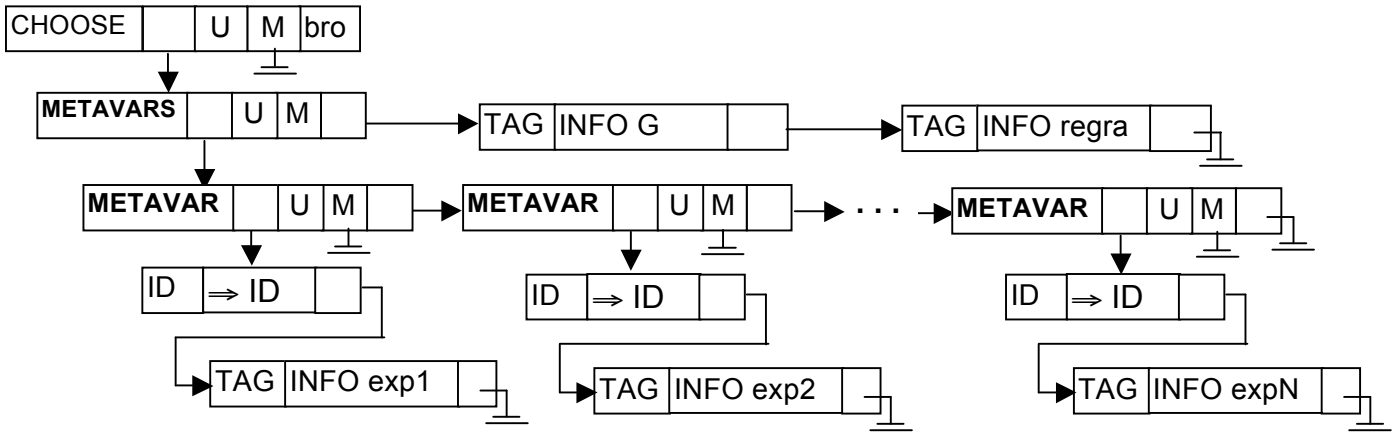


(F) CHOOSE e1, e2, ..., ek SATISFYING G do regra end

1) CHOOSE com METAVAR ID



2) CHOOSE com METAVAR ID associado a TYPE ou SET



Campo	Significado
TAG do 1° nodo	CHOOSE
son	Referencia uma lista de três nodos MIR, encadeados pelo campo **brother**, que denotam os componentes de CHOOSE. O primeiro desses três nodos, METAVARS, é composto por uma lista de nodos MIR do tipo METAVAR, encadeados pelo campo **brother**,
U de METAVAR	Endereço da lista de funções usadas no nodo filho de METAVAR.
U de METAVARS	Endereço da lista de funções usadas nos nodos da lista do componente METAVARS.
U de CHOOSE	Endereço da lista de funções usadas no nodo METAVARS mais as funções usadas na expressão G a ser satisfeita, segundo nodo da lista de três nodos de **son** do nodo CHOOSE, e as funções usadas na regra, terceiro nodo da lista de três nodos de **son** do nodo CHOOSE.

M de METAVAR	Nulo
M de METAVARS	Nulo
M de CHOOSE	Endereço da lista de funções modificadas na regra, terceiro nodo da lista de três nodos de **son** do nodo CHOOSE.

Lista de nodos em son de FORALL: Essa lista contêm três nodos: o primeiro nodo tem o TAG METAVARS e está indicado por **son** do nodo FORALL, o segundo, que é a expressão G a ser satisfeita, está indicado por **brother** do **son** do nodo CHOOSE, e o terceiro que é a regra a ser executada, está indicado por **brother** do **brother** do **son** do nodo CHOOSE.

Lista de nodos em son de METAVARS: Essa lista é composta por nodos em que o primeiro componente está indicado por **son** do nodo METAVARS e os próximos componentes estão indicados caminhando pelo campo **brother**. Todos os nodos dessa lista tem TAG METAVAR.

Nodo em son de METAVAR: É composto pelo nodo que representa um dos componente de METAVARS.

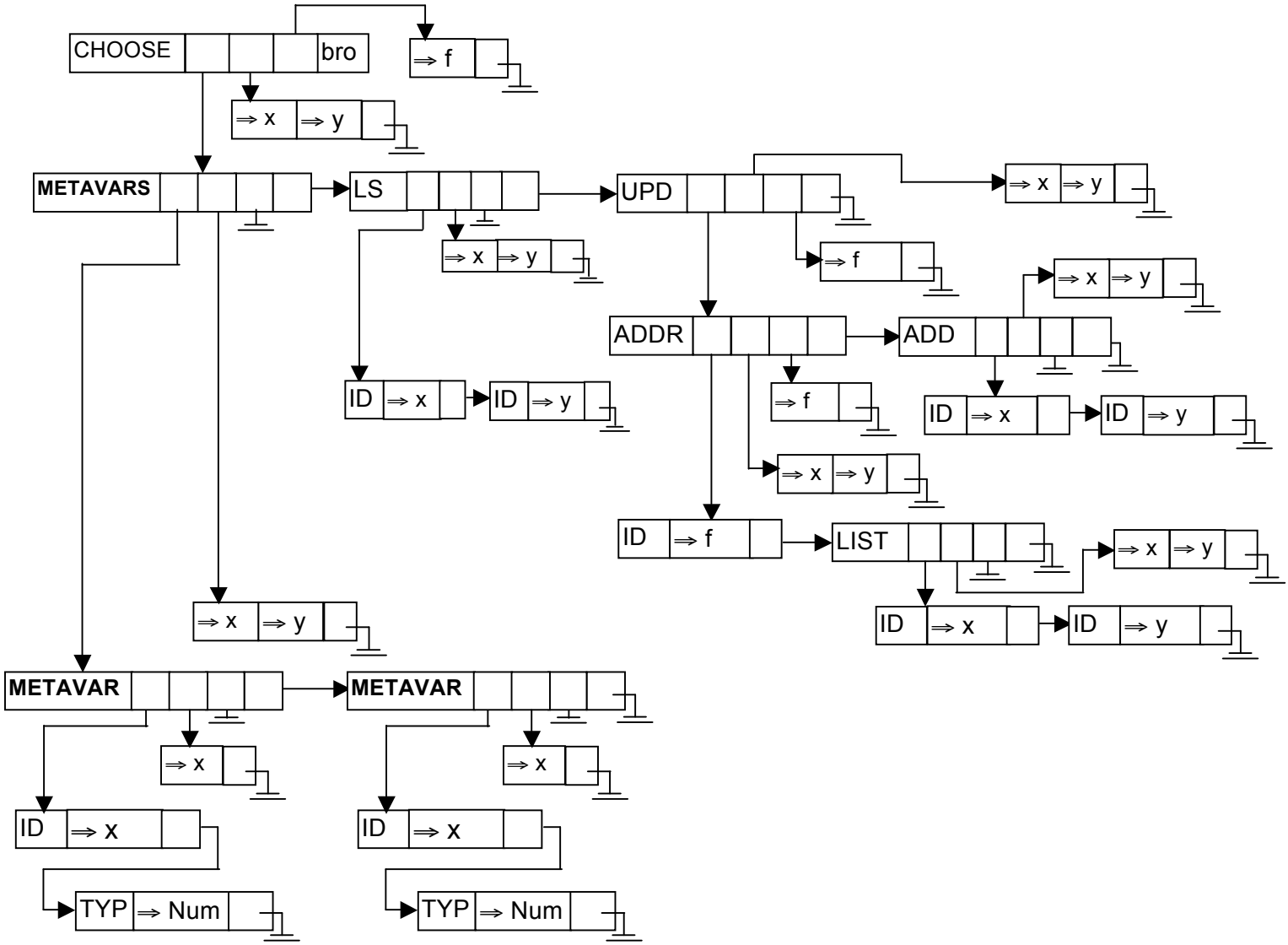
Interpretação semântica: Executar a regra CHOOSE significa executar as regras associadas à regra CHOOSE. Deve-se executar o comando CHOOSE da seguinte forma: Para cada grupo único de valores associado às expressões METAVARS do CHOOSE, deve-se avaliar se a expressão G é satisfeita. Ao encontrar um grupo único de valores que satisfaz à expressão G outros grupos não serão mais procurados. Com o grupo de valores encontrado deve-se, então, executar as regras associadas à regra CHOOSE.

```
Empilha PC->Bro;                PC ← PC->Son
R0 ← PC->Bro;                   Empilha PC;          achou ← falso
enquanto (não achou) e (não testou todas as combinações)
    PC ← PC->Son
    enquanto PC não é nulo
        Empilha PC;   PC ← PC->Son
        se (PC->Bro é nulo) então
                Guarda um valor do tipo do identificador de PC em
                                PC->Value->Definition
         senão   Avalia PC->Bro;
                Guarda um valor de PC->Bro->Value->Definition em
                                PC->Value->Definition
         fim se
         Desempilha PC;        PC ← PC->Bro
     fim enquanto
     Avalia R0 considerando os valores escolhidos
     se R0->Value->Definition é verdadeiro então achou ← verdadeiro
     senão Desempilha PC
     fim se
fim enquanto
if achou então
```

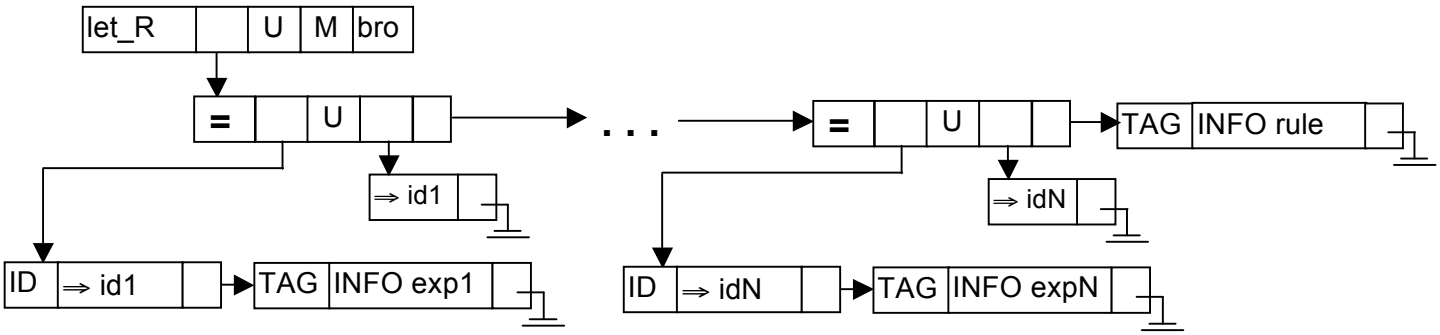
```
        Avalia R0->Bro considerando os valores escolhidos
        Desempilha PC
        Guarda PC->Son->Bro->Bro->Value->Definition em
                                    PC->Value->Definition
senão
        Desempilha PC
        Guarda nulo em PC->Value->Definition
fim se
Desempilha PC
```

Exemplo:

```
type Num = 1 .. 3;
dynamic f(Int, Int):Int;
transition
  choose x:Num, y:Num satisying x < y do
    f(x, y) := x + y;
  end
end
```



```
(G)   let     id1 = expression1;
              id2 = expression2;
                  ...
              idN = expressionN;
            IN rule
          end
```



Campo	Significado
TAG do 1° nodo	LET_R
son	Referencia uma lista de nodos MIR, encadeados pelo campo **brother**. Esses nodos denotam os componentes de LET.
U de **=**	Endereço da lista de funções usadas no segundo nodo da lista de dois nodos do componente **=**.
M de **=**	Endereço da lista com a função definida no primeiro nodo da lista de dois nodos do componente **=**.
U de let_R	Endereço da lista de funções usadas nos segundos nodos da lista de dois nodos dos componentes **=** e no último nodo da lista de LET_R, subtraído do conjunto de identificadores definidos na cláusula LET_R, que são as funções definidas nos primeiros nodos da lista de dois nodos dos componentes **=**.
M	Endereço da lista de funções modificadas na regra, último nodo da lista de nodos de **son** do nodo let_R.

Lista de nodos em son de LET_R: Essa lista contêm nodos, sendo que os **n** primeiros nodos dessa lista tem o TAG **=**, e tem como **son** uma lista com dois nodos MIR, encadeados pelo campo **brother**. O primeiro nodo dos componentes de LET_R está indicado por **son** do nodo LET_R, e os próximos nodos são encontrados caminhando pelo campo **brother**. O último nodo da lista de componentes de LET_R denota uma regra a ser executada, onde serão utilizados os identificadores definidos nos **n** primeiros nodos.

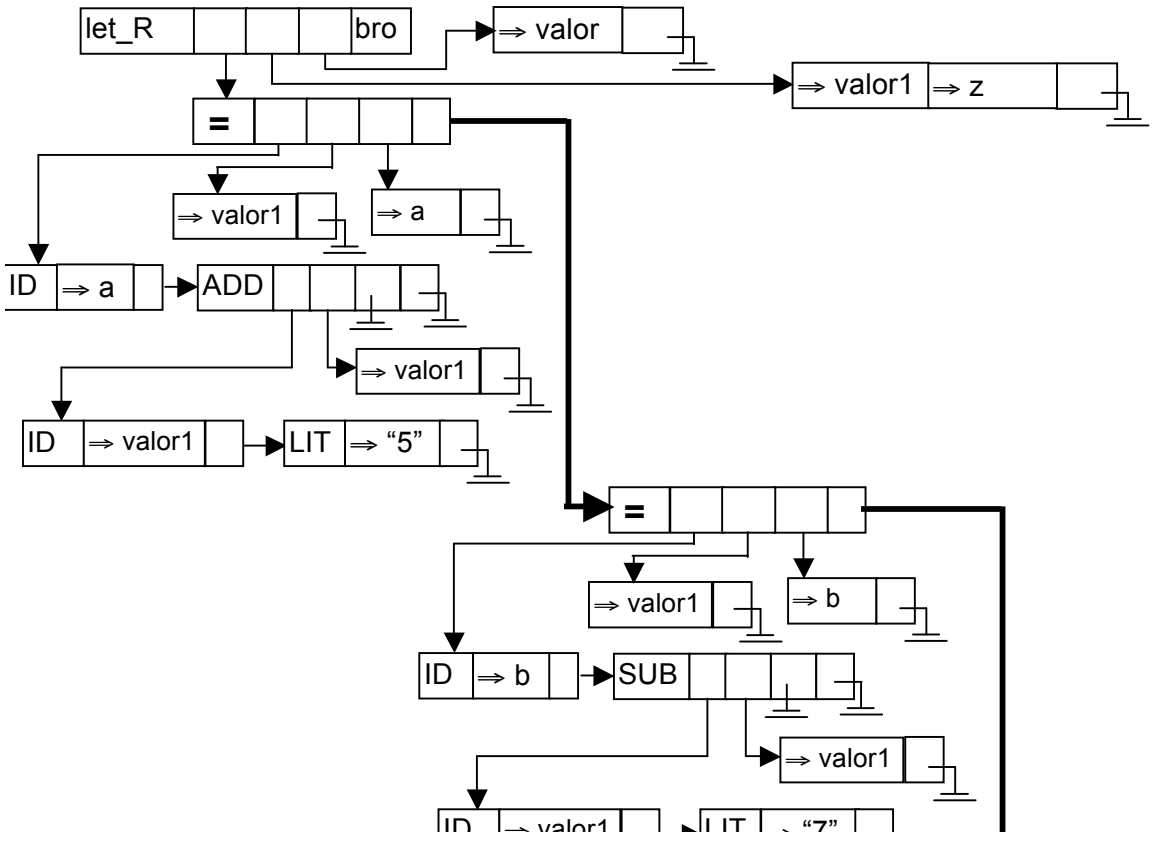
Lista de nodos em son de =: Essa lista é composta por dois nodos: o primeiro componente, que denota um identificador, está indicado por **son** do nodo **=**, e o segundo componente, que denota uma expressão associada ao identificador do primeiro nodo, está indicado por **brother** do **son** do nodo **=**.

Interpretação semântica: Executar a regra LET_R significa executar a regra associada à regra LET_R, que se encontra no último nodo da lista de nodos de **son** do nodo LET_R. Deve-se executar o comando LET_R da seguinte forma: Para os **n** primeiros nodos componentes de LET_E, nodos com TAG **=**, deve-se associar o identificador, primeiro componente da lista de dois nodos do nodo com TAG **=**, ao valor da expressão, segundo componente da lista de dois nodos do nodo com TAG **=**. Então, a regra, último nodo da lista de nodos de **son** do nodo LET_R, deve ser executada.

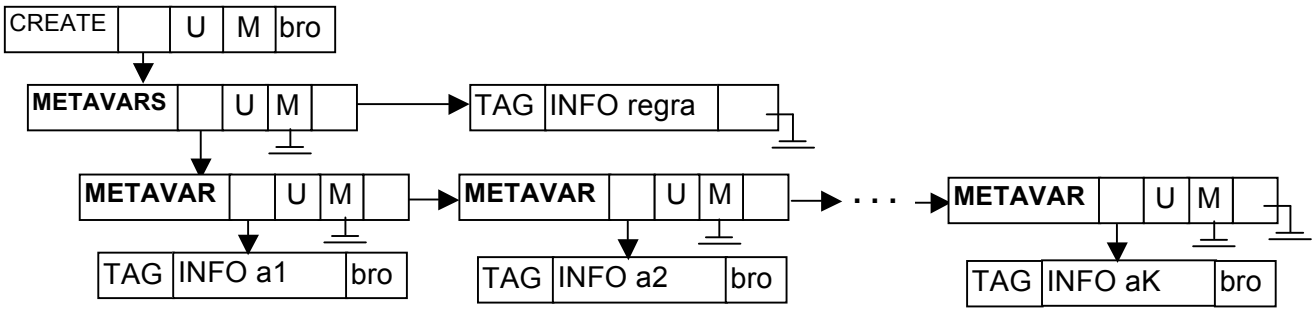
```
Empilha PC->Bro;        Empilha PC;          PC ← PC->Son
enquanto (PC->Bro não é nulo)
    Empilha PC;                              PC ← PC->Son
    Empilha PC;                              Avalia PC
    R1 ← PC->Value->Definition
    Desempilha PC;                           PC ← PC->Bro
    Avalia PC
    Atribui PC->Value->Definition a R1
    Desempilha PC;                           PC ← PC->Bro
fim enquanto
Avalia PC, considerando os valores associados
R0 ← PC->Value->Definition
Desempilha PC
Guarda R0 em PC->Value->Definition
Desempilha PC
```

Exemplo:

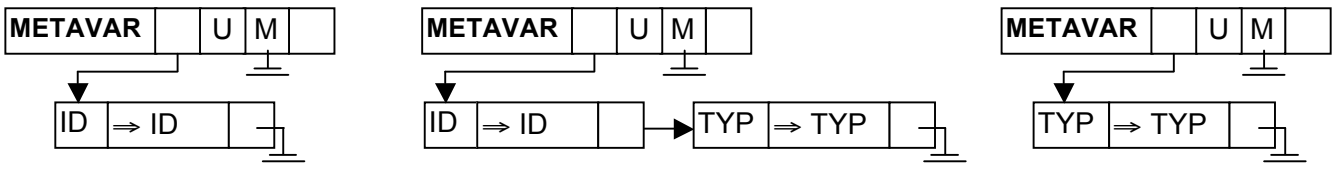
```
let    a = valor1 + 5
       b = valor1 – 7
       in valor(a , b) := 10 + z
end
```



(H) CREATE a1, a2, ..., ak do regra end



Possibilidades para um nodo METAVAR:



Campo	Significado
TAG do 1° nodo	CREATE
son	Referencia uma lista de dois nodos MIR, encadeados pelo campo **brother**, que denotam os componentes de CREATE. O primeiro desses dois nodos, METAVARS, é composto por uma lista de nodos MIR do tipo METAVAR, encadeados pelo campo **brother**,
U de METAVAR	Endereço da lista de funções usadas no nodo filho de METAVAR.

U de METAVARS	Endereço da lista de funções usadas nos nodos da lista do componente METAVARS.
U de CREATE	Endereço da lista de funções usadas no nodo METAVARS mais as funções usadas na regra, segundo nodo da lista de dois nodos de **son** do nodo CREATE.
M de METAVAR	Nulo
M de METAVARS	Nulo
M de CREATE	Endereço da lista de funções modificadas na regra, segundo nodo da lista de dois nodos de **son** do nodo CREATE.

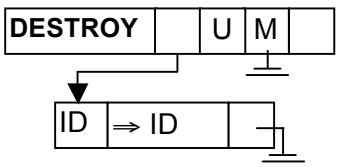
Lista de nodos em son de CREATE: Essa lista contêm dois nodos: o primeiro nodo tem o TAG METAVARS e está indicado por **son** do nodo CREATE, e o segundo, que é a regra a ser executada, está indicado por **brother** do **son** do nodo CREATE

Lista de nodos em son de METAVARS: Essa lista é composta por nodos em que o primeiro componente está indicado por **son** do nodo METAVARS e os próximos componentes estão indicados caminhando pelo campo **brother**. Todos os nodos dessa lista tem TAG METAVAR.

Nodo em son de METAVAR: É composto pelo nodo que representa um componente de METAVARS.

Interpretação semântica: Executar a regra CREATE significa criar um agente que executará a regra do módulo M.

(I) DESTROY id

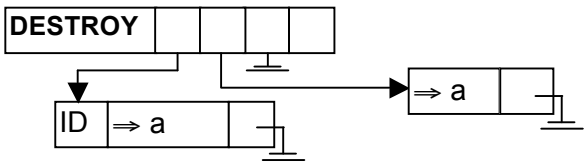


Campo	Significado
TAG do 1° nodo	DESTROY
son	Referencia uma lista de um nodo MIR que denota o identificador de agente a ser destroído.
U de DESTROY	Endereço da lista com a função ID.
M de DESTROY	Nulo

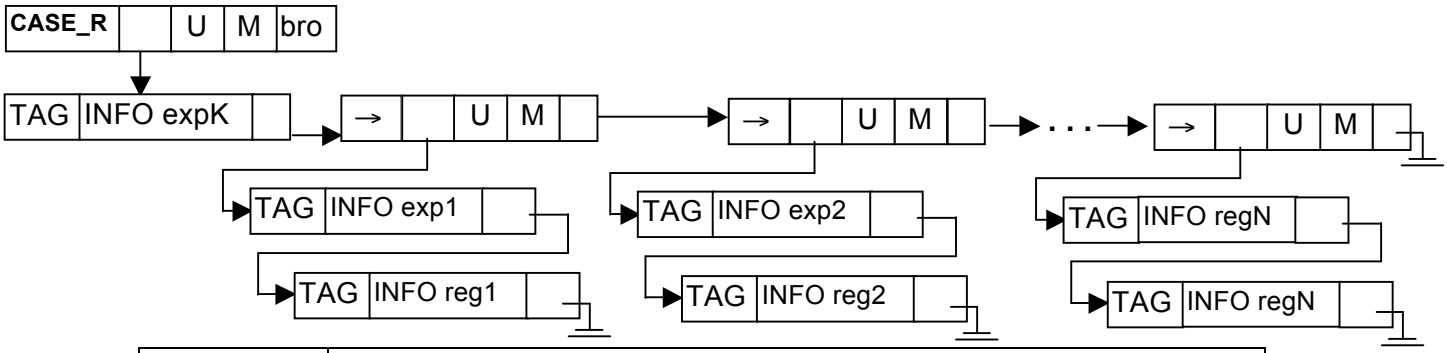
Lista de nodos em son de DESTROY: Essa lista contêm um nodo com TAG ID e está indicado por **son** do nodo DESTROY.

Empilha PC->Bro; Interrompe execução de Desempilha PC	PC ← PC->Son PC

Exemplo: destroy a



(J) condicional CASE



Campo	Significado
TAG do 1° nodo	CASE_R
son	Referencia uma lista de nodos MIR, encadeados pelo campo **brother**. Esses nodos denotam os componentes de CASE.
U de →	Endereço da lista de funções usadas nos 2 componentes de →.
U de CASE_R	Endereço da lista de funções usadas nas expressões e regras dos componentes de CASE_R, primeiro nodo de CASE_R e nodos das listas de dois nodos dos componentes →.
M	Endereço da lista de funções modificadas nas regras, segundo nodo da lista de dois nodos dos componentes →.

Lista de nodos em son de CASE_R: O primeiro nodo dessa lista denota a expressão principal para comparação e está indicado por **son** do nodo CASE_R. A partir desse primeiro nodo, caminhando pelo campo **brother**, é possível encontrar os outros componentes do CASE, que denotam os componentes com TAG →. Esses nodos → referenciam, através do campo **son**, uma lista com dois nodos MIR, encadeados pelo campo **brother**.

Lista de nodos em son de →: Essa lista é composta por dois nodos: (1) o primeiro componente, que denota uma expressão para ser comparada com a expressão principal de comparação, está indicado por **son** do nodo → e (2) o segundo componente, que denota uma regra a ser executada caso o primeiro nodo referente a esse segundo nodo seja compatível com a expressão principal de comparação, está indicado por **brother** do **son** do nodo →. Se o primeiro componente de um nodo → representar o literal "otherwise" o segundo componente, referente ao nodo "otherwise", será executado sempre que as comparações dos componente → anteriores não forem verdadeiras. Um nodo → com o literal "otherwise" como seu primeiro componente só pode aparecer como o último nodo da lista de componentes de CASE.

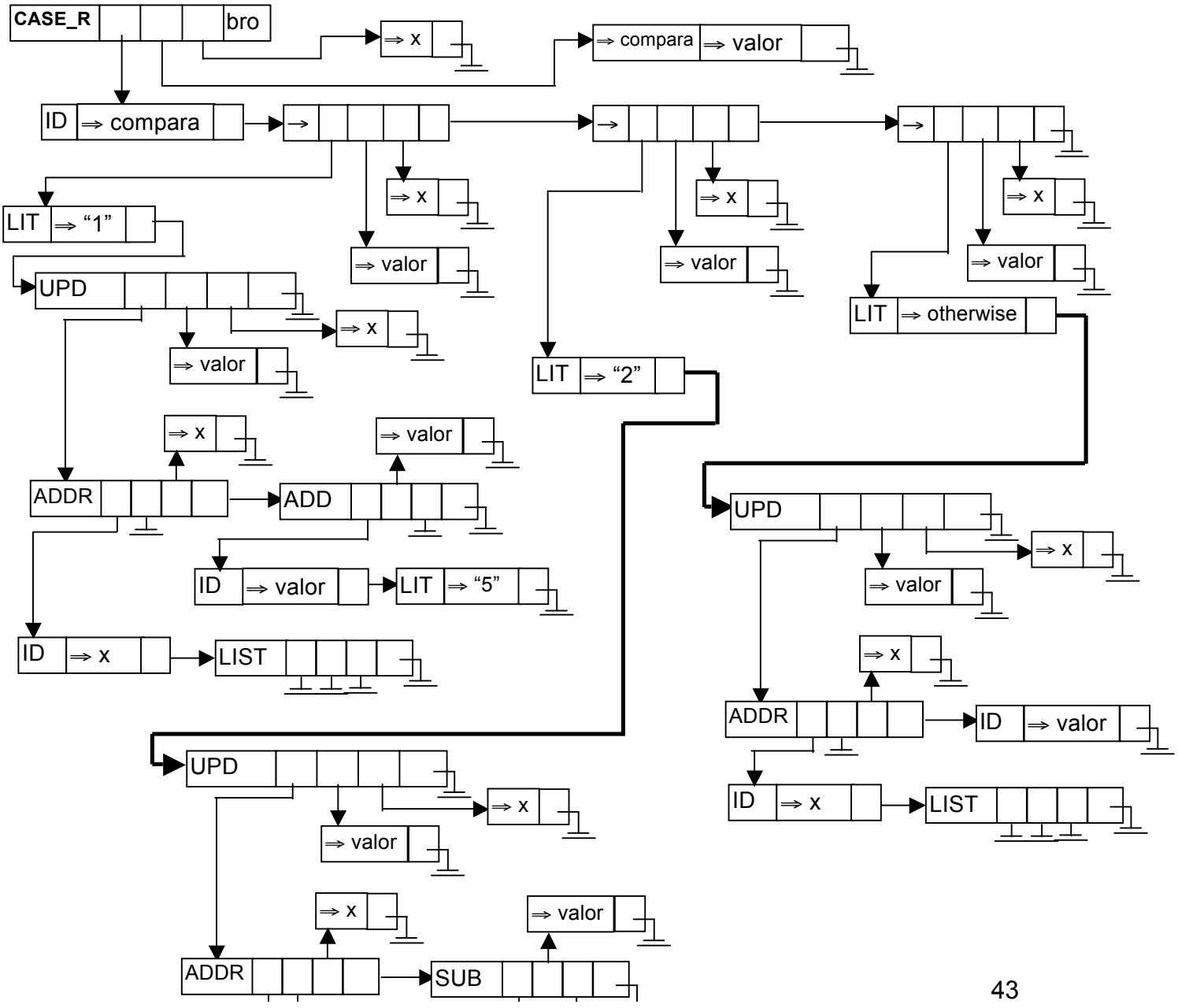
Interpretação semântica: Executar a regra CASE significa executar um dos segundos componentes da lista de dois nodos que compõe um nodo →. Deve-se executar o comando CASE_R da seguinte forma: O primeiro componente da estrutura, condição do CASE_R, deve ser avaliado e o resultado dessa avaliação será utilizado para comparar com os outros componentes do CASE_R. Deve-se, então, caminhar pelo campo **brother** das estruturas, a partir do segundo componente. No componente atual do caminhamento (nodo →) deve-se avaliar o primeiro de seus dois componentes. Se o resultado encontrado com esta avaliação for igual ao resultado da avaliação do primeiro componente de CASE_R, a regra no segundo componente da estrutura de TAG → deverá ser executada. Se o resultado encontrado com esta avaliação for diferente do resultado da avaliação do primeiro componente de CASE_R, o segundo componente do nodo → deverá ser ignorado e o caminhamento pelo campo **brother** deve continuar. Se o componente atual do caminhamento (nodo →) tiver como primeiro componente um nodo com o literal "otherwise" é porque o caminhamento alcançou o último componente sem encontrar nenhuma comparação igual ao primeiro componente de CASE_R. Com isso a regra no segundo componente, referente ao nodo "otherwise", deverá ser executada, terminando a avaliação de CASE_R.

```
Empilha PC->Bro;                          Empilha PC
PC ← PC->Son;                             Avalia PC
R1 ← PC->Value->Definition;               achou ← falso
enquanto (PC não é nulo) e (não achou)
  Empilha PC
  se TAG de PC é #     PC ← PC->Son;                Avalia PC
                       R0 ← PC->Value->Definition
                       achou ← verdadeiro;          Desempilha PC
  senão o TAG é →      PC ← PC->Son;                Avalia PC
                       R2 ← PC->Value->Definition
                       se R1 = R2 então
                             PC ← PC->Son;          Avalia PC
                             R0 ← PC->Value->Definition
                             achou ← verdadeiro;    Desempilha PC
                       senão
                             Desempilha PC;         PC ← PC->Bro
                       fim se
  fim se
fim enquanto
Desempilha PC
se (achou)  então
```

```
    Guarda R0 em PC->Value->Definition
fim se
Desempilha PC
```

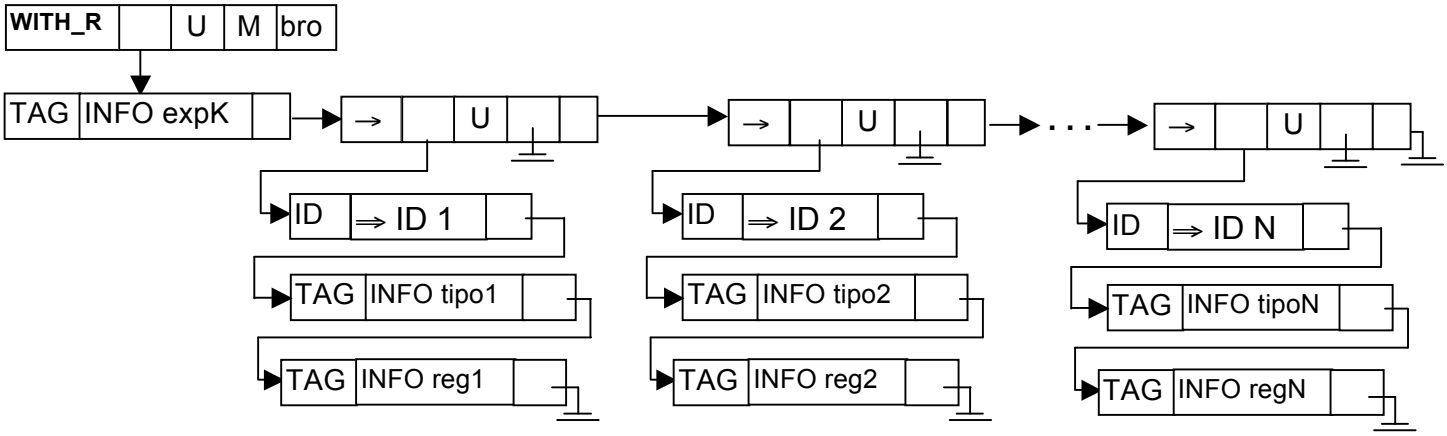
Exemplo:

```
case compara
    1 → x := valor + 5
    2 → x := valor – 5
    otherwise x := valor
end
```

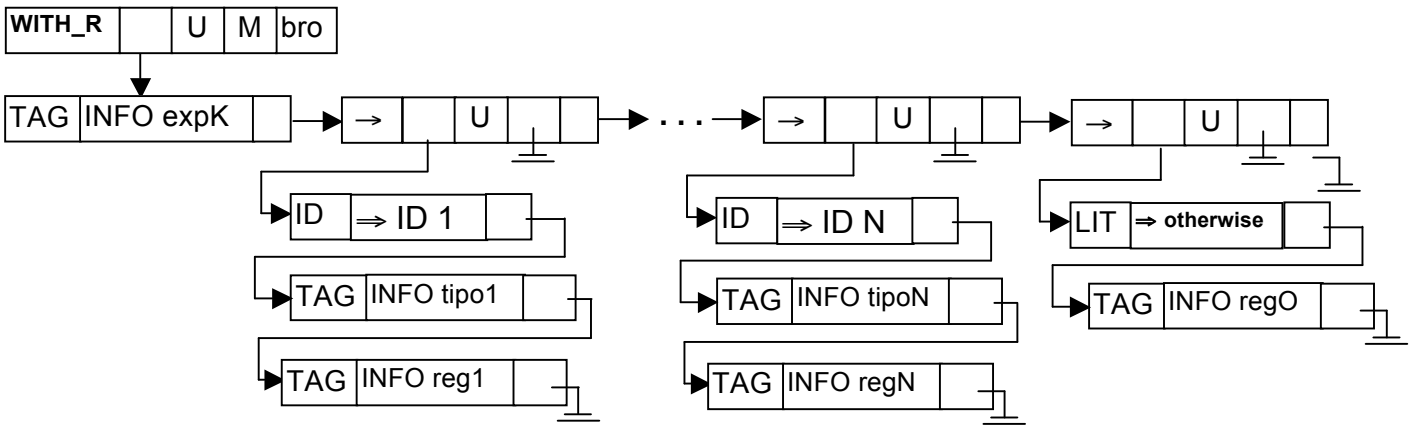


(K) WITH expressionK AS
id1 : Tn1 → regra1
id2 : Tn2 → regra2
...
idN : TnN → regraN
otherWise → regraOtherwise
end

1) WITH sem cláusula otherwise



2) WITH com cláusula otherwise



Campo	Significado
TAG do 1° nodo	WITH_R
son	Referencia uma lista de nodos MIR, encadeados pelo campo **brother**. Esses nodos denotam os componentes de WITH.
U de →	Endereço da lista de funções usadas nos componentes de →.
U de WITH_R	Endereço da lista de funções usadas nos componentes de WITH_R.
M de →	Endereço da lista de funções modificadas nos componentes de →.
M	Endereço da lista de funções modificadas nos componentes de WITH_R.

Lista de nodos em son de WITH_R: O primeiro nodo dessa lista denota a expressão principal para comparação e está indicado por **son** do nodo WITH_R. A partir desse primeiro nodo, caminhando pelo campo **brother**, é possível encontrar os outros componentes do WITH, que denotam os componentes com TAG →. Esses nodos → referenciam, através do campo **son**, uma lista com três nodos MIR, encadeados pelo campo **brother**. O último nodo com TAG → pode conter uma lista com apenas dois nodos MIR, também encadeados pelo campo **brother**.

Lista de nodos em son de →: Essa lista é composta por três nodos: (1) o primeiro componente denota um identificador e está indicado por **son** do nodo →; (2) o segundo componente denota um tipo para ser comparado com o tipo da expressão principal de comparação e está indicado por **brother** do **son** do nodo →; e (3) o terceiro componente denota uma regra a ser executada, com o identificador do primeiro nodo associado à expressão principal de comparação, no caso do tipo no segundo nodo ser compatível com o tipo da expressão principal de comparação. O terceiro componente está indicado por **brother** do **brother** do **son** do nodo →. Se o primeiro componente de um nodo → representar o literal “otherwise” esse nodo → terá apenas dois componentes, sendo que o segundo componente, referente ao nodo “otherwise”, será uma regra que será executada sempre que as comparações de tipo dos componente → anteriores não forem verdadeiras. Um nodo → com o literal “otherwise” como seu primeiro componente só pode aparecer como o último nodo da lista de componentes de WITH.

Interpretação semântica: Executar a regra WITH significa executar um dos terceiros componentes da lista de três nodos que compõe um nodo →, ou executar o segundo componente quando o primeiro nodo é o literal “otherwise”. Deve-se executar o comando WITH_R da seguinte forma: O primeiro componente da estrutura, condição do WITH_R, deve ser avaliado. Tanto o tipo do resultado dessa avaliação quanto o próprio resultado serão utilizados. O tipo do resultado dessa avaliação será comparado com os tipos dos outros componentes do WITH_R. Deve-se, então, caminhar pelo campo **brother** das estruturas, a partir do segundo componente. No componente atual do caminhamento (nodo →) deve-se avaliar o segundo de seus

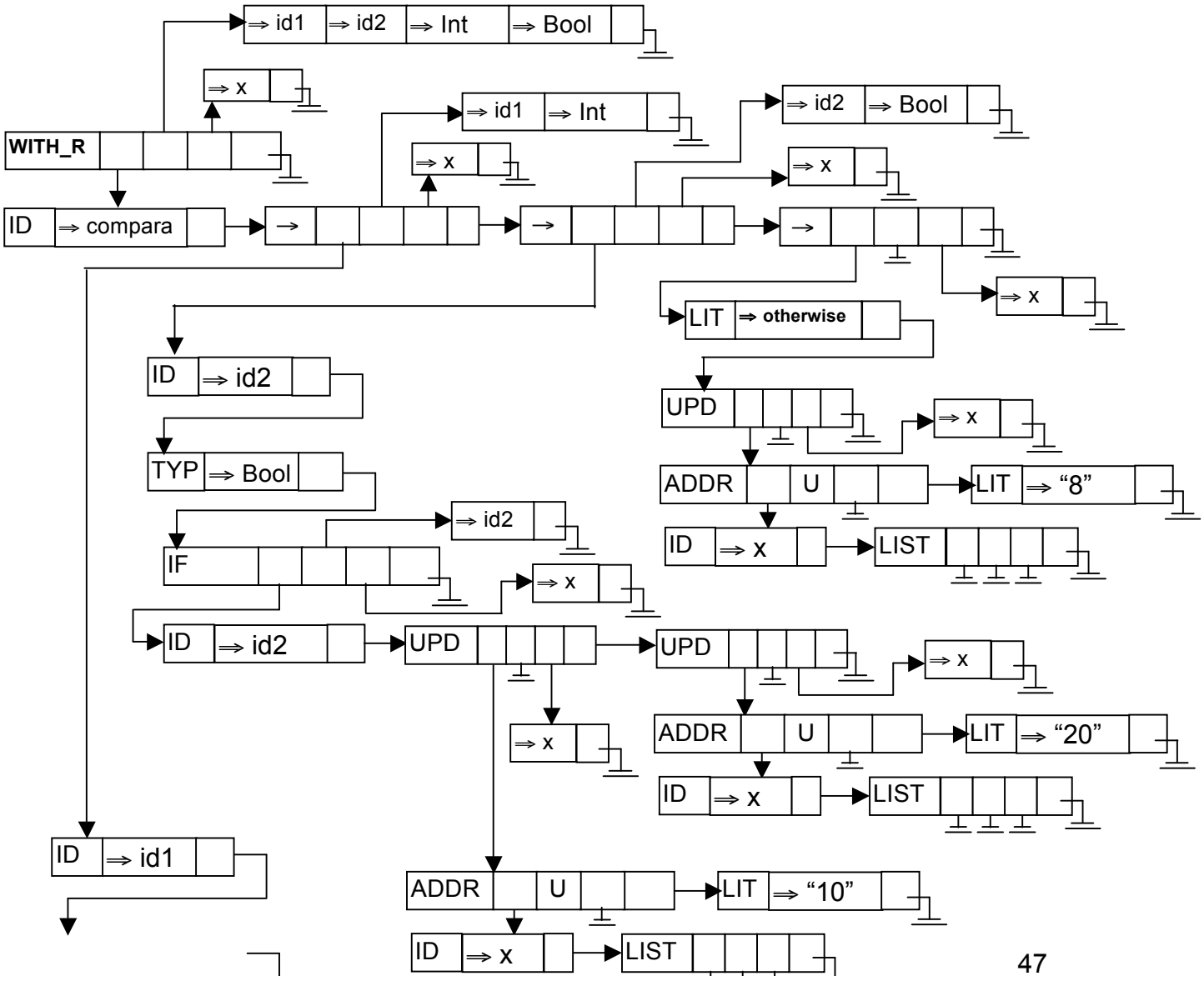
componentes. Se o tipo encontrado neste nodo for igual ao tipo do resultado da avaliação do primeiro componente de WITH_R, A regra do terceiro componente da estrutura de TAG → será executada, terminando a execução de WITH_R. Na avaliação desse terceiro componente o identificador do primeiro nodo deve estar associado à expressão principal de comparação. Se o tipo encontrado com a avaliação do segundo componente for diferente do tipo do resultado da avaliação do primeiro componente de WITH_R, o terceiro componente do nodo → deverá ser ignorado e o caminhamento pelo campo **brother** deve continuar. Se o componente atual do caminhamento (nodo →) tiver como primeiro componente um nodo com o literal "otherwise" é porque o caminhamento alcançou o último componente sem encontrar nenhuma comparação de tipo igual ao tipo do primeiro componente de WITH_R. Com isso o segundo componente, referente ao nodo "otherwise", contém uma regra que será executada, terminando a execução de WITH_R.

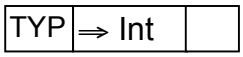
```
Empilha PC->Bro;                    Empilha PC
PC ← PC->Son;                       Avalia PC
R1 ← PC->Value->Definition;         achou ← falso
enquanto (PC não é nulo) e (não achou)
  Empilha PC
  se TAG de PC é #      PC ← PC->Son
                        Avalia PC
                        R0 ← PC->Value->Definition
                        achou ← verdadeiro
                        Desempilha PC
  senão o TAG é ⇔       PC ← PC->Son
                        Empilha PC
                        PC ← PC->Bro
                        Avalia PC
                        R2 ← PC->Value->Definition
                        se (R2 é tipo) e (Tipo(R1) = R2) então
                              Desempilha PC
                              Avalia PC
                              R2 ← PC->Value->Definition
                              Atribui R1 a R2
                              PC ← PC->Bro->Bro
                              Avalia PC
                              R0 ← PC->Value->Definition
                              achou ← verdadeiro
                              Desempilha PC
                        senão
                              Desempilha PC
                              PC ← PC->Bro
                        fim se
  fim se
fim enquanto
```

```
Desempilha PC
se (achou) então
    Guarda R0 em PC->Value->Definition
fim se
Desempilha PC
```

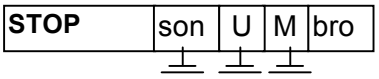
Exemplo:

```
with compara as
    id1 : Int → x := id1 + 5
    id2 : Bool → if id2 then x := 10 else x := 20 end
    otherwise x := 8
end
```





(L) STOP



Campo	Significado
TAG do 1° nodo	STOP
son	Nulo
U de STOP	Nulo
M de STOP	Nulo

Interpretação semântica: Executar a regra STOP significa marcar no vetor de estados que o programa deve terminar sua execução.

```
Vetor de estados->Stop ← verdadeiro
PC ← PC->Bro
```