GERAÇÃO DE CÓDIGO INTERMEDIÁRIO
CAPÍTULO 6 (6.3 A 6.9)

Mariza A S. Bigonha e Roberto S. Bigonha

UFMG

9 de AGOSTO de 2011

6.3 – Tipos e Declarações

As aplicações dos tipos podem ser agrupadas em verificação e tradução:

- Verificação de tipo usa regras lógicas para raciocinar sobre o comportamento de um programa em tempo de execução.

  Ela garante que os tipos dos operandos casam com o tipo esperado por um operador.

  Por exemplo, o operador && em Java espera que seus dois operandos sejam boolianos; o resultado também é do tipo booliano.

· · · Tipos e Declarações

- Aplicações de tradução. A partir do tipo de um nome, um compilador pode determinar qual armazenamento será necessário para esse nome durante a execução.

  A informação de tipo também é necessária para calcular:

  – o endereço denotado por uma referência a arranjo,

  – para inserir conversões de tipo explícitas e

  – para escolher a versão correta de um operador aritmético, entre outras aplicações.

Expressões de Tipos

Uma expressão de tipo é um tipo básico ou é formada pela aplicação de um operador chamado construtor de tipo a uma expressão de tipo.

Os conjuntos de tipos básicos e construtores depende da linguagem a ser verificada.

··· Expressões de Tipos

- **Tipos Básicos:**
  Um tipo básico é uma expressão de tipo. Exemplos: integer, real, char, etc.

  Tipos especiais:
  $type - error$: apontará para um erro durante a verificação de tipo.

  $void$: denota a ausência de valor, permite que comandos sejam verificados.

- **Nome:**
  Como expressões de tipos podem ser nomes, o tipo nome é uma expressão de tipo.

··· Expressões de Tipos

- **Construtor de Tipo:** Um tipo construtor aplicado a expressões de tipo é uma expressão de tipo.
  - **Arranjos:** Se T é uma expressão de tipo, então array(I,T) é uma expressão de tipo denotando o tipo de um arranjo com elementos do tipo T e um conjunto de índices I. I está freqüentemente no intervalo dos inteiros.
    Exemplo em Pascal: var A: array[1..10] of integer;
    associa a expressão de tipo array(1..10, integer) com A.
  - **Records:** O tipo de um registro é o produto dos tipos de seus campos.
    A verificação de tipos de um registro pode ser feita usando uma expressão de tipo formada pela aplicação do construtor record na tupla formada a partir dos nomes dos campos e seus tipos associados.

··· Expressão de Tipos

- **Produto cartesiano**
  Se T_1 e T_2 são expressões de tipo, então seu produto cartesiano T_1 X T_2 são expressões de tipo. Assume-se que X se associa à esquerda.
  A diferença entre record e produto cartesiano é que os campos de um record tem nomes.

- **Apontadores**
  Se T é uma expressão de tipo, então pointer(T) é uma expressão de tipo denotando o tipo apontador para um objeto do tipo T.

Nomes de Tipo e Tipos Recursivos

Quando uma classe é definida, seu nome pode ser usado como um **nome de tipo** em C++ ou Java; por exemplo, considere `Node` no fragmento de programa

```
public class Node { ··· }
···
public Node n;
```

Os nomes podem ser usados para definir os **tipos recursivos**, que são necessários para estruturas de dados como listas encadeadas. O pseudocódigo para um elemento de lista

```
class Cell { int info; Cell next; ··· }
```

define o tipo recursivo `Cell` como uma classe que contém um campo `info` e um campo `next` do tipo `Cell`.

· · · Nomes de Tipo e Tipos Recursivos

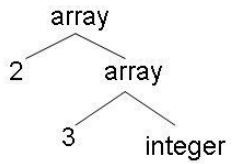
Tipos recursivos semelhantes podem ser definidos em C usando registros e apontadores.

Uma forma conveniente de representar uma expressão de tipo é usar um grafo.

· · · Nomes de Tipo e Tipos Recursivos

O método código numérico pode ser adaptado para construir um DAG para uma expressão de tipo da seguinte forma:

- nós interiores representam os construtores de tipo
- folhas representam os tipos básicos, por exemplo, nomes de tipo e variáveis de tipo.



Equivalência de Tipo

Quando duas expressões de tipo são equivalentes?

Muitas regras de verificação de tipo têm a forma se duas expressões de tipo são iguais, então retorne determinado tipo; senão, erro.

As ambigüidades possíveis surgem quando nomes são dados a expressões de tipo e estes nomes são então usados em expressões de tipo subseqüentes.

A questão-chave é se um nome em uma *expressão de tipo* tem significado próprio ou é uma abreviação para outra *expressão de tipo*.

· · · Equivalência de Tipo

Se *expressões de tipo* são representadas por grafos, 2 tipos são estruturalmente equivalentes se e somente se uma das condições a seguir for verdadeira:

1. Eles são do mesmo tipo básico.
2. Eles são formados pela aplicação do mesmo construtor para tipos estruturalmente equivalentes.
3. Um é um nome de tipo que denota o outro.

Se os *nomes de tipo* são tratados como tendo significado próprio, então as Condições (1) e (2) na definição anterior levam à equivalência de nome das expressões de tipo.

Às expressões com nomes equivalentes é atribuído o mesmo código numérico, se for usado o Algoritmo 6.3.

Equivalência estrutural pode ser testada usando-se o algoritmo de unificação (Seção 6.5.5).

RECORDANDO ··· Método Código Numérico para Construção de DAG

Algoritmo 6.3: Método código numérico para construir nós de um DAG.

ENTRADA: Rótulo op, nó l, e nó r.

SAÍDA: O código numérico de um nó no arranjo com assinatura $\langle op, l, r \rangle$.

MÉTODO:
Procure no arranjo um nó M com rótulo op, filho à esquerda l, e filho à direita r.

Se houver esse nó, retorne o código numérico de M.

Se não, crie no arranjo um novo nó N com rótulo op, filho à esquerda l e filho à direita r, e retorne seu código numérico.

Declarações

- Exemplo de uma linguagem simples

  P → D ";" E
  D → D ";" D
  | id ":" T
  T → char | integer | array[num] of T | ↑T
  E → literal | num | id | E mod E | E [E] | E↑

- Um programa gerado por esta gramática: key : integer
  kwy mod 1999

- Assuma que todo arranjo começa com 1: array[256] of char
  ⇓
  array (1..256, char)

Construtor array aplicado a 1..256 e do tipo char .

··· Exemplo de uma linguagem simples

- Como em Pascal, o operador prefixado ↑ em declarações constroi um tipo apontador, assim:

↑ integer
⇓
pointer (integer)

Construtor pointer aplicado a um tipo integer .

Leiaute de Armazenamento para Nomes Locais

O esquema de tradução (SDT) usa atributos sintetizados $type$ e $width$ para cada não-terminal e duas variáveis t e w para propagar as informações de tipo e largura a partir de um nó B na árvore de derivação para o nó da produção $C \to \epsilon$.

Em uma definição dirigida por sintaxe, t e w seriam atributos herdados por C.

T	→	B	$\{ t = B.type; w = B.width; \}$
		C	$\{ T.type = C.type; T.width = C.width; \}$
B	→	int	$\{ B.type = integer; B.width = 4; \}$
B	→	float	$\{ B.type = float; B.width = 8; \}$
C	→	ϵ	$\{ C.type = t; C.width = w; \}$
C	→	[num] C_1	$\{ C.type = array(\text{num}.value, C_1.type);$ $C.width = \text{num}.value \times C_1.width; \}$

··· Leiaute de Armazenamento para Nomes Locais

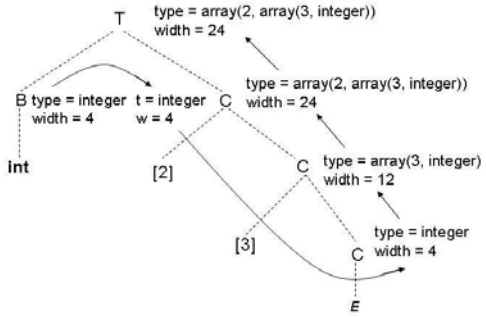
Atributos sintetizados: $type$ **e** $width$**.**

Atributos herdados por C**:** t **e** w**.**

T	$\rightarrow$	B C	$\{\ t = B.type;\ w = B.width;\ \}$
B	$\rightarrow$	int	$\{\ B.type = integer;\ B.width = 4;\ \}$
B	$\rightarrow$	float	$\{\ B.type = float;\ B.width = 8;\ \}$
C	$\rightarrow$	ϵ	$\{\ C.type = t;\ C.width = w;\ \}$
C	$\rightarrow$	[num] C_1	$\{\ array(\mathbf{num}.value,\ C_1.type);$ $C.width = \mathbf{num}.value \times C_1.width;\ \}$

Árvore de derivação para `int[2][3]`**: linhas pontilhadas.**

Linhas sólidas: propagação do tipo e tamanho.



Seqüência de Declarações

P $\rightarrow$ D : E		
D $\rightarrow$ D : D		
D $\rightarrow$ id : T		{ *addtype* (id.*entry*, T.*type*) }
T $\rightarrow$ char		{ T.type := *char* }
T $\rightarrow$ integer		{ T.type := *integer* }
T $\rightarrow$ $\uparrow T_1$		{ T.type := *pointer* (T_1.type) }
T $\rightarrow$ array [num] of T_1		{ T.type := *array* (1..num.val, T_1.type) }

• Neste esquema de tradução:

Ação associada à produção: D $\rightarrow$ id:T salva um tipo na entrada da tabela de símbolos para o um identificador.

A ação *addtype* (id.entry, T.type) é aplicada ao atributo sintetizado *entry* apontado pela entrada de id na tabela de símbolos e a expressão de tipo representada pelo atributo sintetizado *type* do não-terminal T.

Se T gera char ou integer, então *T.type* é definido por *char* ou *integer* .

O limite superior de um arranjo é obtido do atributo *val* do *token* num que fornece o inteiro representado por num.

Como D aparece antes de E do lado direito de P $\rightarrow$ D ; E é garantido que todos os tipos dos identificadores declarados serão salvos antes que a expressão gerada por E seja verificada.

··· Seqüência de Declarações

Linguagens como C e Java permitem que todas as declarações em um único procedimento sejam processadas como um grupo.

As declarações podem ser distribuídas dentro de um procedimento Java, mas ainda podem ser processadas quando o procedimento é analisado.
Portanto, podemos usar uma variável, por exemplo $offset$**, para obter o próximo endereço relativo disponível.**

O esquema de tradução trata uma seqüência de declarações da forma T id**, onde** T **gera um tipo.**

P	$\rightarrow$	D	$\{\ offset = \mathbf{0};\ \}$
D	$\rightarrow$	T id ; D_1	$\{\ top.put(\mathbf{id}.lexeme,\ T.type,\ offset);$ $offset = offset + T.width;\ \}$
D	$\rightarrow$	ϵ	

··· Seqüência de Declarações

P	$\rightarrow$	D	$\{\ offset = \mathbf{0};\ \}$
D	$\rightarrow$	T id ; D_1	$\{\ top.put(\mathbf{id}.lexeme,\ T.type,\ offset);$ $offset = offset + T.width;\ \}$
D	$\rightarrow$	ϵ	

A ação semântica da produção $D \rightarrow T$ id ; D_1 **cria uma entrada na tabela de símbolos executando** $top.put(\mathbf{id}.lexeme,\ T.type,\ offset)$**.**

top **representa a tabela de símbolos corrente.**

O método $top.put$ **cria uma entrada na tabela de símbolos para** $\mathbf{id}.lexeme$**, com tipo** $T.type$ **e endereço relativo** $offset$ **em sua área de dados.**

··· Seqüência de Declarações

Não-terminais gerando *offset*, chamados não-terminais marcadores, podem ser usados para reescrever produções de modo que todas as ações apareçam nos extremos dos lados direitos.

Usando um não-terminal marcador M,

$$P \rightarrow \{\ offset = 0;\ \}\ D \qquad (1)$$

pode ser reescrita como:

$$P \rightarrow M\ D$$
$$M \rightarrow \epsilon \qquad \{\ offset = 0;\ \}$$

··· Seqüências de Declarações

D → integer idlist
  | real idlist
idlist → idlist , id
  | id

- Dificuldade: **Ao formar idlist não se sabe o tipo dos** ids.
- Soluções: **(1) Fatore a gramática.**
  (2) Crie uma lista em separado.

- Rotinas Semânticas para Solução 1:

D→ integer **id** { **instala(**id.place,integer,−−**); D.atr :=** integer }

D → real **id** { **instala(**id.place, real, −−**); D.atr :=** real }

D→ D_1 **, id** {**instala(**id.place,D_1.atr,−−**); D.atr :=** D_1.atr }

Declarações de Procedimentos

D → prochead decls S

prochead → procedure id args

args → (parlist)
  | $\mathcal{E}$

parlist → idlist : type
  | parlist ; idlist : type

idlist → id
  | idlist , id

decls → D
  | decls ; D

··· Declarações de Procedimentos

Solução 1: Modificar a Gramática

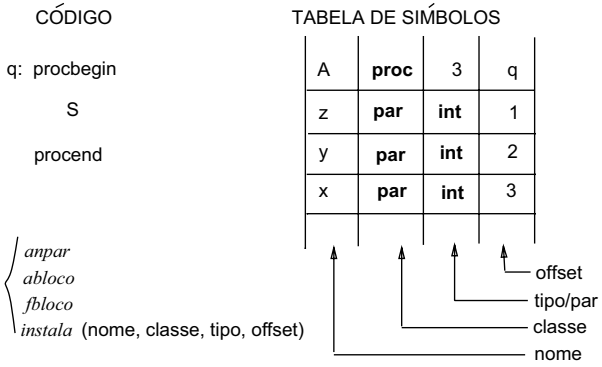
D → prochead decls S
prochead → proc args
proc → procedure id
args → (parlist)
  | $\mathcal{E}$
parlist → par
  | par ; parlist
par → id : type
  | id , par
decls → D
  | decls ; D

- Atributos: S.next, D.atr,proc.place, prochead.place, decls.size args.np, par.np, parlist.np = **número de parâmetros.** par.tipo = **tipo do parâmetro (**integer, real, **etc.).** type.tipo = **tipo.**

··· Declarações de Procedimentos

··· Solução 1:

procedure **A(x, y, z: integer) decls S**



··· Declarações de Procedimentos

··· Solução 1:

- Cálculo de offsets (estrutura auxiliar)

··· Declarações de Procedimentos

··· Solução 1:

- Rotinas Auxiliares
- *instala*(nome, classe, tipo, offset): **retorna endereço onde nome foi instalado na tabela de símbolos.**
- *abloco*
- *fbloco*
- *anpar*(pts, n): **atualiza campo número de parâmetros com valor n.**
- *asize*(pts, size): **– atualiza campo** size.

··· Declarações de Procedimentos: Solução 1: Gramática Modificada

par → id : **type** { **instala(**id.**nome,** par**,type.tipo,offset)**
offset := offset + tam(type.tipo)
par.tipo := type.tipo
par.np := 1 }

par → id , par_1 { **instala(**id.**nome,**par**,**par_1**.tipo,offset)**
offset := offset + tam(par_1**.tipo)**
par.tipo := par_1**.tipo**
par.np := par_1**.np + 1** }

parlist → **par** { **parlist.np := par.np** }

parlist → **par ;** $parlist_1$ { **parlist.np := par.np +** $parlist_1$**.np** }

Declarações de Procedimentos: Solução 1: Gramática Modificada

args →	(parlist)	{ args.np := parlist.np }
args →	$\mathcal{E}$	{ args.np := 0 }
prochead →	proc args	{ prochead.place := proc.place anpar(proc.place, args.np) }
proc →	procedure id	{ proc.place := instala(id.nome,proc,0,nextc gen(procbegin proc.place) abloco push(offset) offset := 1 }

Declarações de Procedimentos: Solução 1: Modificar a Gramática

D →	prochead decls S	{ fbloco asize(prochead.place, decls.size) pop(offset) D.atr := proc backpatch(S.next, nextquad) gen(procend) }
decls →	D	{ decls.size := tam(D.atr) }
decls →	$decls_1$, D	{ decls.size:=$decls_1$.size+tam(D.atr)}

Declações de Procedimentos - Solução 2: Não modificar a gramática

procedure A(x, y, z: integer) decls S

A	proc	3	q	...
x	par	int	1	...
y	par	int	2	...
z	par	int	3	...

D	→	prochead decls S
prochead	→	proc args
proc	→	procedure id
args	→	(parlist) | $\mathcal{E}$
parlist	→	idlist : type
	|	parlist ; idlist : type
idlist	→	id | idlist , id
decls	→	D | decls ; D

Declações de Procedimentos - Solução 2: Não modificar a gramática

- Rotinas Auxiliares
- *instala*(nome, classe, tipo, offset): retorna endereço onde nome foi instalado na tabela de símbolos.
- *abloco*
- *fbloco*
- *anpar*(pts, n): atualiza campo número de parâmetros com "n".
- *asize*(pts, size) – atualiza campo size.

Declações de Procedimentos - Solução 2: Não modificar a gramática

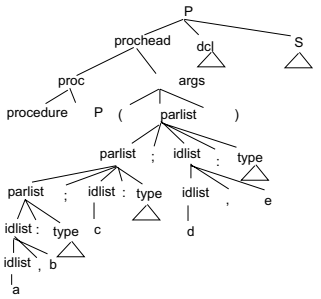
- $\cdots$ Rotinas Auxiliares

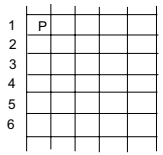
- *atipoeoffset*(k, n, tipo)

Para **cada um dos n parâmetros instalados na tabela de símbolos a partir do endereço k**
faça **tipo do parâmetro := tipo.**
offset do parâmetro := OFFSET.
OFFSET := OFFSET + tam(tipo).
fim

- *tam*(tipo): – **tamanho do tipo.**

Declações de Procedimentos - Solução 2: Não modificar a gramática





Declações de Procedimentos - Solução 2: Não modificar a gramática

proc $\rightarrow$ procedure id { **proc.place:=**
instala(id.nome,proc,**0,nextquad)**
gen(procbegin **proc.place)**
abloco
push(offset)
offset := 1 }

prochead $\rightarrow$ **proc args** { **anpar(proc.place, args.np)**
prochead.place := proc.place }

args $\rightarrow$ **(parlist)** { **args.np := parlist.np** }

args $\rightarrow$ $\mathcal{E}$ { **args.np := 0** }

Declações de Procedimentos - Solução 2: Não modificar a gramática

idlist $\rightarrow$ id
{ idlist.place := *instala*(id.nome,par,$--$,$--$)
idlist.np := 1 }

idlist $\rightarrow$ idlist$_1$, id
{ *instala*(id.nome,par,$-$,$-$)
idlist.place := $\boldsymbol{idlist_1}$.place
idlist.np := $\boldsymbol{idlist_1}$.np + 1 }

parlist $\rightarrow$ idlist : type
{ *atipoeoffset*(idlist.place, idlist.np, type.tipo)
parlist.np := idlist.np }

parlist $\rightarrow$ $\boldsymbol{parlist_1}$; idlist : type
{ *atipoeoffset*(idlist.place, idlist.np, type.tipo)
parlist.np := $\boldsymbol{parlist_1}$.np + idlist.np }

Declarações de Procedimentos - Solução 2: Não modificar a gramática

D → prochead decls S
{ asize(prochead.place, decls.size)
fbloco
pop(offset)
backpatch(S.next, nextquad)
gen(procend)
D.atr := proc }

decls → D { decls.size := tam(D.atr) }

decls → $decls_1$, D { decls.size := $decls_1$.size + tam(D.atr) }

Campos em Registros e Classes

A tradução das declarações mostradas pode tratar também os campos dos registros e das classes.

Os tipos registro podem ser adicionados à gramática

$$
\begin{array}{lcll}
T & \rightarrow & B & \{\ t = B.type;\ w = B.width;\ \} \\
 & & C & \\
B & \rightarrow & \text{int} & \{\ B.type = integer;\ B.width = 4;\ \} \\
B & \rightarrow & \text{float} & \{\ B.type = float;\ B.width = 8;\ \} \\
C & \rightarrow & \epsilon & \{\ C.type = t;\ C.width = w;\ \} \\
C & \rightarrow & \text{[num] } C_1 & \{\ array(\text{num}.value,\ C_1.type); \\
 & & & \quad C.width = \text{num}.value \times C_1.width;\ \}
\end{array}
$$

acrescentando-lhe a seguinte produção

$$T \rightarrow \text{record } '\{' \ D \ '\}'$$

··· Campos em Registros e Classes

$$T \rightarrow \text{record } '\{' \ D \ '\}'$$

Os campos nesse tipo registro são especificados pela seqüência de declarações geradas por D. A abordagem mostrada pode ser usada para determinar os tipos e endereços relativos dos campos, desde que tenhamos cuidado com os dois casos:

- Os nomes dos campos de um registro devem ser distintos; ou seja, um nome pode aparecer no máximo uma vez nas declarações geradas por D.
- O deslocamento ou endereço relativo para um nome de campo é relativo à área de dados para esse registro.

··· Campos em Registros e Classes

Exemplo: O uso de um nome x para um campo de um registro não entra em conflito com outros usos do nome fora do registro.

Portanto, os três usos de x nas declarações a seguir são distintos e não geram conflitos:

```
float x;
record { float x; float y; } p;
record { int tag; float x; float y; } q;
```

A atribuição subseqüente `x = p.x + q.x;` define a variável `x` como a soma dos campos chamados `x` nos registros `p` e `q`.

Observe que o endereço relativo de `x` em `p` difere do endereço relativo de `x` em `q`.

· · · Campos em Registros e Classes

Os tipos registro podem ser adicionados à gramática

$$
\begin{array}{lcll}
T & \rightarrow & B & \{\ t = B.type;\ w = B.width;\ \} \\
 & & C & \\
B & \rightarrow & \text{int} & \{\ B.type = integer;\ B.width = 4;\ \} \\
B & \rightarrow & \text{float} & \{\ B.type = float;\ B.width = 8;\ \} \\
C & \rightarrow & \epsilon & \{\ C.type = t;\ C.width = w;\ \} \\
C & \rightarrow & [\ \text{num}\ ]\ C_1 & \{\ array(\text{num}.value,\ C_1.type); \\
 & & & \quad C.width = \text{num}.value \times C_1.width;\ \}
\end{array}
$$

acrescentando-lhe a seguinte produção

$$
\begin{array}{lcll}
T & \rightarrow & \text{record}\ '\{'\ & \{\ Env.push(top);\ top = \text{new}\ Env(); \\
 & & & \quad Stack.push(offset);\ offset = 0;\ \} \\
 & & D\ '\}' & \{\ \boldsymbol{T.type} = record(\boldsymbol{top});\ \boldsymbol{T.width} = offset; \\
 & & & \quad top = Env.pop();\ offset = Stack.pop();\ \}
\end{array}
$$

· · · Campos em Registros e Classes

$$
\begin{array}{lcll}
T & \rightarrow & \text{record}\ '\{'\ & \{\ Env.push(top);\ top = \text{new}\ Env(); \\
 & & & \quad Stack.push(offset);\ offset = 0;\ \} \\
 & & D\ '\}' & \{\ \boldsymbol{T.type} = record(\boldsymbol{top});\ \boldsymbol{T.width} = offset; \\
 & & & \quad top = Env.pop();\ offset = Stack.pop();\ \}
\end{array}
$$

As ações semânticas mostram o pseudocódigo para uma implementação específica. A classe Env implementa a tabela de símbolos.

A chamada $Env.push(top)$ coloca a tabela de símbolos corrente, denotada por top, em uma pilha.

A variável top é então atribuída a uma nova tabela de símbolos.

De forma semelhante, $offset$ é colocado em uma pilha chamada $Stack$. Depois, a variável $offset$ é atribuída com 0.

Estruturas RECORDS na Linguagem C

$$
\begin{array}{lcl}
\textbf{type} & \rightarrow & \text{struct}\ \{\ \textbf{fieldlist}\ \} \\
 & | & \text{ptr} \\
 & | & \text{char} \\
 & | & \text{int} \\
 & & \\
\textbf{fieldlist} & \rightarrow & \textbf{fieldlist field} \\
 & | & \textbf{field} \\
 & & \\
\textbf{field} & \rightarrow & \textbf{type}\ \text{id} \\
 & | & \textbf{field [const]}
\end{array}
$$

· · · Estruturas RECORDS na Linguagem C

- Atributos e Subrotinas

field.width, fieldlist.width, type.width = tamanho do tipo

field.name = nome do campo, isto é, pts.

D-enter (nome, size): incrementa 1 no número de dimensões de nome e instala seu tamanho size.

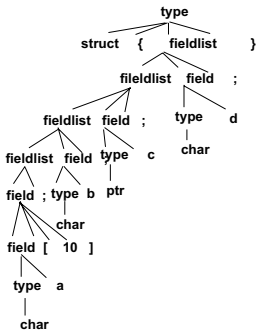
W-enter (nome, width): instala tamanho de nome.

O-enter (nome, offset): instala o offset de nome na tabela de símbolos.

Estruturas RECORDS na Linguagem C

field	→	type **id**	{ field.width := type.width field.name := **id**.name *W-enter* (**id**.name, type.width) }
field	→	$\boldsymbol{field}_1$ [const]	{ field.width := $\boldsymbol{field}_1$.width * const.val field.name := $\boldsymbol{field}_1$.name *D-enter* ($\boldsymbol{field}_1$.name, const.val) }
fieldlist	→	field;	{ *O-enter* (field.name, 0) fieldlist.width := field.width }
fieldlist	→	$\boldsymbol{fieldlist}_1$ field;	{ *O-enter* (field.name, $\boldsymbol{fieldlist}_1$.width) fieldlist.width := $\boldsymbol{fieldlist}_1$.width + field.width }
type	→	**struct** { fieldlist }	{ type.width := fieldlist.width }
	|	**char**	{ type.width := 1 }
	|	**ptr**	{ type.width := 4 }

Estruturas RECORDS na Linguagem C



	nome	dim	width	offset	...
1	a	10	1	0	
2	b		1	10	
3	c		4	11	
4					
5					
6					
7					
8					

Tradução de Expressões - Comandos de Atribuição com Inteiros

Produção	Regras Semânticas
$S \rightarrow$ id = E ;	$S.code = E.code \mid\mid gen(top.get(\text{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \text{new}\ Temp()$ $E.code = E_1.code \mid\mid E_2.code \mid\mid gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
| - E_1	$E.addr = \text{new}\ Temp()$ $E.code = E_1.code \mid\mid gen(E.addr\ '='\ 'minus'\ E_1.addr)$
| (E_1)	$E.addr = E_1.addr$ $E.code = E_1.code$
| id	$E.addr = top.get(\text{id}.lexeme)$ $E.code = ''$

top denota a tabela de símbolos corrente. A função $top.get$ recupera a entrada quando ela é aplicada à representação da cadeia $id.lexeme$ dessa instância de id.

Definições dirigidas por sintaxe: a função gen monta uma instrução e a retorna.

Esquemas de tradução: gen monta instrução e a emite de incrementalmente, colocando-a no fluxo de instruções geradas.

Tradução de Expressões - tradução incremental

Os atributos de código podem ser cadeias longas, de modo que usualmente elas são geradas de forma incremental.

gen não apenas constrói uma instrução de três endereços, mas a anexa à seqüência de instruções geradas até o momento.

S	→	id = E ;	{ $gen(top.get(\text{id}.lexeme)\ '='\ E.addr)$; }
E	→	E_1 + E_2	{ $E.addr = \text{new}\ Temp()$; $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$; }
	|	- E_1	{ $E.addr = \text{new}\ Temp()$; $gen(E.addr\ '='\ 'minus'\ E_1.addr)$; }
	|	(E_1)	{ $E.addr = E_1.addr$; }
	|	id	{ $E.addr = top.get(\text{id}.lexeme)$; }

· · · Comandos de Atribuição - Tradução de A:=−B*(C+D) para Quádruplas

Entrada	Pilha Sintática	Pilha Semântica	Código
A := −B * (C + D)			
:= −B * (C + D)	id	A	
−B * (C + D)	id:=	A●	
B * (C + D)	id:=−	A●●	
* (C + D)	id:=−id	A●●B	
* (C + D)	id:=−E	A●●B	T_1 := −B
* (C + D)	id:=E	A●T_1	
(C + D)	id:=E*	A●T_1●	
C + D)	id:=E*(	A ●T_1●●	
+ D)	id:=E*(id	A●T_1●●C	
+ D)	id:=E*(E	A●T_1●●C	
D)	id:=E*(E+	A●T_1●●C●	
)	id:=E*(E+id	A●T_1●●C●D	
)	id:=E*(E+E	A●T_1●●C●D	T_2 := C + D
)	id:=E*(E	A●T_1●●T_2	
	id:=E*(E)	A●T_1●●T_2●	
	id:=E* E	A●T_1●T_2	T_3 := T_1 * T_2
	id:=E	A●T_3	A := T_3
	A		

· · · Comandos de Atribuição

Tradução para Quádruplas para Inteiros e Reais

X, Y − reais
I, J − inteiros

- **Exemplo: X := Y + I * J**

T_1 := I int* J
T_2 := inttoreal $T1$
T_3 := Y real+ T_2
X := T_3

- Atributos: **mode: integer ou real**
  place: apontador para a tabela de símbolos.

· · · Comandos de Atribuição - Tradução para Quádruplas: E → E_1 op E_2

{ if E_1. mode = integer and E_2.mode = integer
then
  begin T := newtemp(integer)
    gen (T := E_1.place intop E_2.place)
    E.mode := integer
  end
elif E_1. mode = real and E_2.mode = real then
  begin T := newtemp(real)
    gen (T := E_1.place realop E_2.place)
    E.mode := real
  end
elif E_1. mode = integer % and E_2.mode = real %
then
  begin U := newtemp(real); T := newtemp(real);
    gen (U := inttoreal E_1.place)
    gen (T := U realop E_2.place)
    E.mode := real
  end
else % E_1. mode = real and E_2.mode = integer %
then
  begin U := newtemp(real); T := newtemp(real);
    gen (U := inttoreal E_2.place)
    gen (T := E_1.place realop U)
    E.mode := real
  end
E.place := T }

Endereçando Elementos de Arranjos

Elementos de um arranjo podem ser acessados rapidamente se os elementos são armazenados em blocos consecutivos de memória. Um arranjo de 2 dimensões é normalmente armazenado por linhas ou por colunas.

Fortran usa coluna.

Pascal usa linha.

··· Comandos de Atribuição - Arranjos

array **A**[L_1: U_1, L_2: U_2, ···, L_n: U_n]

A[L_1, L_2, ···, L_n] ··· **A**[L_1, U_2, U_3, ···, U_n]

A[L_{1+1}, L_2, ···, L_n] ··· **A**[L_{1+1}, U_2, U_3, ···, U_n]

⋮

A[i, L_2, ···, L_n] ··· **A**[i, U_2, U_3, ···, U_n]

⋮

A[U_1,L_2,···,L_n] ··· **A**[U_1, U_2, U_3, ···, U_n]

$$d_1 = U_1 - L_1 + 1$$
$$d_2 = U_2 - L_2 + 1$$
$$\vdots$$
$$d_n = U_n - L_n + 1$$ - Qual é o endereço de A[i,L_2,···,L_n] ?

··· Endereçando Elementos de Arranjos

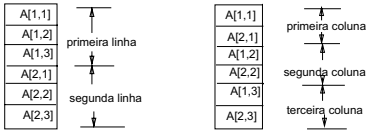
Se o tamanho de cada elemento do arranjo é "d" , então o i-ésimo elemento do arranjo A começa na localização: BASE + (i−L_1) * d_2*d_3*···*d_n

onde L é o limite inferior no subscrito e BASE é o endereço relativo de A[L_1].

E_i = Endereço de A[i,L_2,···,L_n] = BASE + (i−L_1) * d_2*d_3*···*d_n

E_{ij} = Endereço de A[i,j,L_3,···,L_n] = E_i + (j−L_2) * d_3 * d_4 * ··· * d_n

E_{ijk} = Endereço de A[i,j,k,L_4,···,L_n] = E_{ij}+(k−L_3) * d_4*d_5*···*d_n



··· Endereçando Elementos de Arranjos

• Endereçando elementos de arranjos: **A**[i,j,k, ···, l,m] =

BASE + (i - L_1) * d_2 * ··· * d_n +

(j - L_2) * d_3 * ··· * d_n +

(k - L_3) * d_4 * ··· * d_n +

⋮

(l - L_{n-1}) * d_n +

m - L_n = constpart + varpart

··· Endereçando Elementos de Arranjos

constpart = **BASE** − ((L_1*d_2+L_2)*d_3+L_3)*d_4+···L_{n-1})*d_n+L_n

varpart = (··· ((i*d_2+j) *d_3+···+l) * d_n + m)

• Cálculo de varpart: **varpart := i**
varpart := varpart*d_2 + **j**
varpart := varpart*d_3 + **k**
···
varpart := varpart*d_n+**m**

Constpart → **calculado só 1 vez na declaração do arranjo.**

··· Endereçando Elementos de Arranjos

(1) S → L := E
(2) E → E + E
(3) | (E)
(4) | L
⇒ (5) L → id [elist] - Que arranjo?
(6) | id
(7) elist → elist, E
(8) | E

A regra 5 gera um identificador e depois uma lista independente.

- Necessário "personalizar" elist

L → elist]
elist → id [E
| elist , E

··· Endereçando Elementos de Arranjos

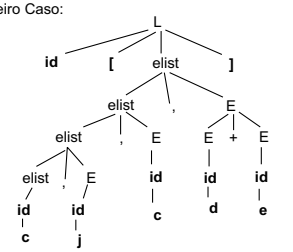
(1) S → L := E
(2) E → E + E
(3) | (E)
(4) | L
⇒ (5) L → elist]
(6) | id
(7) elist → id [E
(8) | elist, E

A regra 5 gera um identificador ou um `elist` seguido de].

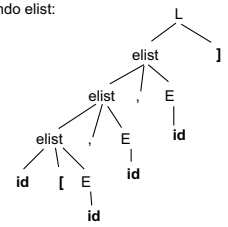
A regra 7 começa com o nome do arranjo seguido do primeiro índice ···

··· Endereçando Elementos de Arranjos - Árvores para L

Primeiro Caso:



Personalizando elist:



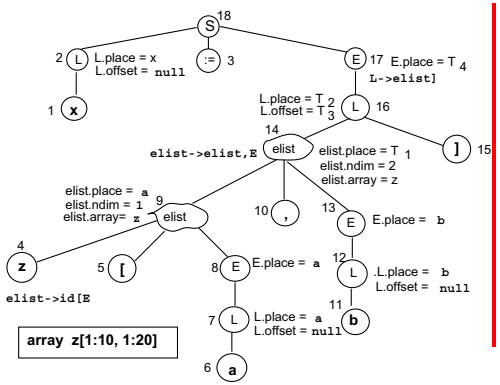
··· Endereçando Elementos de Arranjos

- Atributos

elist.array = "apontador" para id na T.S.,
elist.place = endereço de varpart.
elist.ndim = número de ordem da dimensão. Inicialmente vale 1.
Após redução: [elist → elist1, E] vale +1.

E.place = endereço do resultado de E.
L.place = se L não é arranjo
então endereço na tabela de símbolos do id,
senão endereço na tabela de símbolos do temporário que
contém a "constpart" .
L.offset = se L não é arranjo então null
senão é o offset do elemento do arranjo já linearizado,
(vai conter constpart+varpart).
bpw= "byte per word" , (porque cada elemento tem um tamanho).

... Endereçando Elementos de Arranjos



Exemplo: x := z[a, b]

14 T_1 := a * 20 varpart
14 T_1 := T_1 + b
16 T_2 := z_0 - 84 constpart
16 T_3 := 4 * T_1 varpart
17 T_4 := $T_2[T_3]$
18 x := T_4

T_1 = i * d_2 + j $\equiv$ T_1 = 1 * 20 + 1 = 21, 21 * 4 = 84
[elist.array − C] , elist.array $\equiv$ z = endereço do arranjo (BASE)
C = 84 $\equiv$ fórmula do constpart: T_2 = BASE − L_1 * d_2 + L_2.

... Endereçando Elementos de Arranjos

S → L := E { if L.offset = null then /* L é um id */
emit(L.place ":=" E.place)
else emit(L.place "[" L.offset "]" ":=" E.place) }

E → E_1 + E_2 { E.place := newtemp
emit(E.place ":=" E_1.place "+" E_2.place) }

E → (E_1) { E.place := E_1.place }

E → L { if L.offset = null then /* L é um id */
E.place := L.place
else begin
E.place := newtemp
emit(E.place ":=" L.place "[" L.offset "]") }

... Endereçando Elementos de Arranjos

L → elist] { T := newtemp
U := newtemp
emit(T ":=" $\underline{elist.array - C}$) (⋆)
emit(U ":=" bpw "*" elist.place)
L.place := T
L.offset := U }

L → id { L.place := id.place
L.offset := null }

(⋆) – parte sublinhada corresponde a constpart.

... Endereçando Elementos de Arranjos

elist → $elist_1$, E { T := newtemp
/* varpart := varpart * d_i + índice */
D := limit($elist_1$.array, $elist_1$.ndim + 1)
emit (T ":=" $elist_1$.place "*" D);
emit (T ":=" T "+" E.place);
elist.array := $elist_1$.array;
elist.place := T;
elist.ndim := $elist_1$.ndim + 1 }

elist → id [E { elist.place := E.place
elist.ndim := 1;
elist.array := id.place }

Tradução de Referências a Arranjo - OUTRO EXEMPLO

Considere que o não-terminal L gera um nome de arranjo seguido por uma seqüência de expressões denotando índices do arranjo:

$$L \rightarrow L\ [\ E\ ] \mid \mathrm{id}\ [\ E\ ]$$

Como em C e Java, suponha que o elemento de número mais baixo no arranjo seja 0.

Cálculo dos endereços com base nas larguras, usando a fórmula:

$$base + i_1 \times w_1 + i_2 \times w_2 + \cdots + i_k \times w_k \qquad (2)$$

em vez dos números de elementos.

··· Tradução de Referências a Arranjo - OUTRO EXEMPLO

O não-terminal L possui três atributos sintetizados:

1. $L.addr$ denota um temporário que é usado enquanto calcula o deslocamento da referência ao arranjo, somando os termos $i_j \times w_j$.

2. $L.array$ é um apontador para a entrada da T.S. para o nome do arranjo. O endereço de base do arranjo, $L.array.base$, é usado para determinar o valor-l corrente de uma referência ao arranjo depois que todas as expressões de índice forem analisadas.

3. $L.type$ é o tipo do subarranjo gerado por L. Para qualquer tipo t, assuma que sua largura é dada por $t.width$. Para qualquer tipo arranjo t, suponha que $t.elem$ dê o tipo do elemento.

Endereço base do arranjo: $L.array.base$.

$L.addr$: temporário que contém o deslocamento para a referência ao arranjo gerado por L.

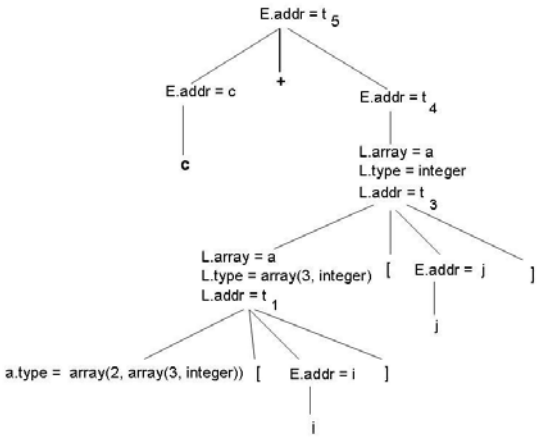
Localização para a referência ao arranjo: $L.array.base[L.addr]$.

··· Tradução de Referências a Arranjo - OUTRO EXEMPLO

S	$\rightarrow$	id = E ;	{ $gen($ $top.get($id.$lexeme$) '=' $E.addr$); }
	|	L = E ;	{ $gen(L.array.base$ '[' $L.addr$ ']' '=' $E.addr$); }
E	$\rightarrow$	E_1 + E_2	{ $E.addr =$ new $Temp$ (); $gen(E.addr$ '=' $E_1.addr$ '+' $E_2.addr$); }
	|	id	{ $E.addr = top.get($id.$lexeme$); }
	|	L	{ $E.addr =$ new $Temp$ (); $gen(E.addr$ '=' $L.array.base$ '[' $L.addr$ ']'); }
L	$\rightarrow$	id [E]	{ $L.array = top.get($id.$lexeme$)$; $L.type = L.array.type.elem$; $L.addr =$ new $Temp$ (); $gen(L.addr$ '=' $E.addr$ '*' $L.type.width$); }
	|	L_1 [E]	{ $L.array = L_1.array$; $L.type = L_1.type.elem$; $t =$ new $Temp$ (); $L.addr =$ new $Temp$ (); $gen(t$ '=' $E.addr$ '*' $L.type.width$); $gen(L.addr$ '=' $L_1.addr$ '+' t);}

··· Tradução de Referências a Arranjo - OUTRO EXEMPLO

Árvore de derivação anotada para: `c + a[i][j]`.



a: arranjo 2 x 3 de inteiros.
`c`, `i`, e `j`: inteiros.

Tipo de `a`:
$array(2, array(3, integer))$.
Largura de w: 24, supondo que inteiro tenha largura 4.

Tipo de `a[i]`: $array(3, integer)$.
Largura $w_1 = 12$.
Tipo de `a[i][j]`: $integer$.

```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a [ t3 ]
t5 = c + t4
```

6.5 VERIFICAÇÃO DE TIPOS

- Análise Semântica - **Verificação Estática**

- Exemplos de Verificação Estática:

Tipo: Um compilador pode reportar um erro se um *operador* é aplicado a um operando incompatível.

Exemplo: adição de uma variável do tipo *array* com uma variável do tipo *function*.

Fluxo de controle: comandos que fazem com que o fluxo de controle deixe uma construção deve ter algum lugar para onde transferir o fluxo de controle.

Exemplo: comando *break* de C faz com que o controle deixe os comandos *while* , *for* ou *switch* mais internos.

· · · Verificação de Tipos

- Análise Semântica

Unicidade: **Existem situações na qual um objeto deve ser definido exatamente uma vez.**

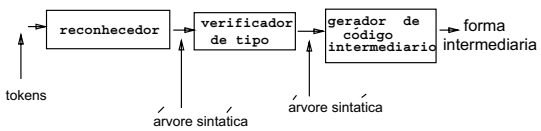
Exemplo: em Pascal, um identificador deve ser declarado somente uma vez, rótulos em um comando case devem ser distintos, etc.

Relacionados com nomes de blocos: As vezes um mesmo nome deve aparecer duas ou mais vezes. O compilador deve verificar que o mesmo nome é usado nos dois lugares.

Exemplo: em Ada um loop ou bloco pode ter um nome que aparece no início e no final destas construções.

· · · Verificador de Tipo

Um verificador de tipo analisa se o tipo de uma construção está de acordo com seu contexto.



Vários compiladores Pascal combinam a verificação estática com a geração de código intermediário e análise sintática.

Para construções mais complexas, como as existentes em ADA, é conveniente ter um passo separado de verificação de tipos entre a análise sintática e a geração de código intermediário.

· · · Verificador de Tipo - Alguns Exemplos

Um verificador de tipos verifica se o tipo de uma construção casa o esperado em seu contexto.

- **Um operador aritmético built-in mod do Pascal necessita de operandos inteiros. Então, o verificador de tipos deve verificar se os operandos do mod têm tipos inteiros.**

- **Indexação é feita somente com arranjos.**

- **Uma definição de função é aplicada para um número e tipo correto de parâmetros.**

As informações obtidas pelo verificador de tipos são usadas quando o código é gerado.

Sistemas de Tipo

Um verificador de tipos implementa um sistema de tipo.

Um Sistema de tipo é uma coleção de regras para atribuir expressões de tipo a várias partes do programa.

A verificação feita por um compilador é dita ser estática.

A verificação feita no momento em que o programa objeto é executado é dita ser dinâmica.

··· Sistemas de Tipos

O projeto de um sistema de tipos para uma linguagem é baseado em informações sobre a:

- Construções sintáticas na linguagem.
- Noção de tipos.
- Regras para atribuir tipos nas construções de uma linguagem.

Informações Relevantes para o Projetista do Compilador
(Pascal report e Manual de referência de C)

- "Se ambos os operandos de uma operação aritmética de adição, subtração e multiplicação são do tipo integer, então o resultado é do tipo integer".
- "O resultado de um operador unário & é um apontador para o objeto referenciado pelo operando. Se o tipo do operando é "··· " , o tipo do resultado é um " pointer to ··· " ."

Verificação de Tipos em Expressões

Nas regras a seguir o atributo sintetizado type para E estabele o tipo da expressão atribuida pelo sistema de tipo para a expressão gerada por E.

- As rotinas semânticas para as constantes representadas pelos tokens literal e num estabelecem que os mesmos têm tipos char e integer.

  E → literal { E.type := char }
  E → num { E.type := integer }

- Quando um identificador aparece em uma expressão, seu tipo declarado na tabela de símbolos é buscado pela função lookup (e), através da entrada apontada por e e é atribuido ao atributo type.

  E→id { E.type:=lookup (id.entry) }

··· Verificação de Tipos em Expressões

- A expressão formada pela aplicação do operador mod a duas sub-expressões do tipo integer tem tipo integer, senão caracteriza um erro de tipo.

  E → E_1 mod E_2 { E.type := if E_1.type = integer and
  E_2.type = integer then integer
  else type-error }

- Em uma referência a um arranjo E_1 [E_2], o índice da expressão E_2 deve ter tipo integer , neste caso o resultado é o elemento tipo t obtido do tipo array (s, t) de E_1 .

  E → E_1 [E_2] { E.type := if E_2.type = integer and
  E_1.type = array (s, t) then t
  else type-error }

··· Verificação de Tipos em Expressões

- Dentro de expressões, o operador na forma posfixada ↑ produz o objeto apontado por seu operando. O tipo de E↑ é o tipo t do objeto apontado pelo apontador de E.

$$\text{E} \rightarrow E_1 \uparrow \{ \text{ E.type := if } E_1\text{.type = pointer (t) then t else type-error } \}$$

- Observação: Para permitir que identificadores tenham tipo boolean basta introduzir a produção T → boolean na gramática dada.

A introdução de operadores de comparação como "<"e conectivos como "and" nas produções para E, permitem a construção de expressões do tipo boolean.

Verificação de Tipos em Comandos

- Construções de linguagens to tipo comando, tipicamente, não possuem valores, assim estas construções podem ter um tipo void .
- Comandos considerados: atribuição
  condicional
  while

- Seqüências de comando são separados por ";" .
- Obs.: As produções para comandos podem ser combinadas com as produções para tipos de um id definidas anteriormente substituindo a produção P → D ";" E por P → D ";" S.

O programa agora tem declarações seguida por comandos. Mas as regras anteriores para verificação de expressões continua sendo necessárias porque comandos podem ter expressões.

Esquema de Tradução para Verificar o Tipo dos Comandos

1. S → id := E { S.type := (if id.type = E.type then void else type-error }
2. S → if E then S_1 { S.type := (if E.type = boolean then S_1.type else type-error }
3. S → while E do S_1 { S.type := (if E.type = boolean then S_1.type else type-error }
4. S → S_1 ; S_2 { S.type := (if S_1.type = void and S_2.type = void then void else type-error }

- 1. Verifica se o lado esq. e o lado dir. de um comando de atribuição possuem o mesmo tipo.
- 2. e 3. Especificam que expressões em comandos condicionais e while devem ter tipo *boolean*.
- 4. Erros são propagados porque uma seqüência de comandos tem tipo *void* somente se cadas sub-expressão tem tipo *void*.

Verificação de Tipos em Funções

- A aplicação de uma função para seu argumento pode ser capturada pela seguinte produção:

$$\text{E} \rightarrow \text{E (E)}$$

As regras de associação de expressões de tipos com o não-terminal T podem ser acrescidas pela produção e ação abaixo para permitir tipos em funções na declaração.

$$\text{T} \rightarrow T_1 \text{ '}\rightarrow\text{' } T_2 \{ \text{ T.type := } T_1\text{.type} \rightarrow T_2\text{.type } \}$$

Aspas em '→' o diferencia de → usado como meta-símbolo na produção.

· · · Verificação de Tipos em Funções

- Regra para verificação de tipo na aplicação de uma função:

E $\rightarrow E_1\ (E_2)$ { E.type := (if E_2.type = s and
E_1.type = s $\rightarrow$ t then t
else type-error }

Esta regra diz que uma expressão formada pela aplicação de E_1 em E_2, o tipo de E_1 deve ser uma função s $\rightarrow$ t a partir do tipo s de E_2 para algum tipo t; o tipo de $E_1\ (E_2)$ é t.

· · · Verificação de Tipos em Funções

A generalização de função com mais de um argumento é feita construindo tipo produto consistindo dos argumentos.

Note que: n argumentos do tipo $T_1, \cdots, T_n$ pode ser visto como um único argumento do tipo T_1 X $\cdots$ XT_n.

Exemplo: root : (real $\rightarrow$ real) X real $\rightarrow$ real

Esta sintaxe separa a declaração do tipo de uma função dos nomes de seus parâmetros.
A função root que é a aplicação da função real em real e tem como argumento um real e retorna um real.

- Sintaxe em Pascal: function root(function f(real):real;x:real):real

Conversões de Tipo

Dada a expressão: x + i, onde x é real e i é inteiro.
Representação de inteiros e reais $\leftrightarrow$ diferente dentro do computador.

Diferentes instruções de máquina usadas para operações de inteiros e reais

Compilador pode ter que primeiro converter um dos operandos de "+" para assegurar que ambos os operandos sejam do mesmo tipo quando a adição tiver lugar.

· · · Conversões de Tipo

Conversões Implícitas ou Coerções

Uma conversão é dita ser implícita se for realizada automaticamente pelo compilador.

Linguagem C: converte implicitamente os caracteres ASCII para inteiros na faixa de 0 e 127.

··· Conversões de Tipo

Conversão Explícita

A conversão é explícita se o programador tem que escrever alguma coisa para provocar a conversão.

Todas as conversões em ADA são explícitas.

As conversões explícitas se parecem exatamente com aplicações de funções para um verificador de tipos.

Exemplos: ord - mapeia um caractere para um inteiro;

chr realiza o mapeamento inverso.

Regras para a Verificação de Tipos para a Coerção de Inteiro para Real

Produção	Regra	Semântica
E → num	E.tipo :=	inteiro
E → num.num	E.tipo :=	real
E → id	E.tipo :=	lookup(id.entrada)
E → E_1 op E_2	E.tipo :=	if E_1.tipo = integer
		and E_2.tipo = integer
		then integer
		else if E_1.tipo = integer
		and E_2.tipo = real
		then real
		else if E_1.tipo = real
		and E_2.tipo = integer
		then real
		else if E_1.tipo = real
		and E_2.tipo = real
		then real
		else type-error

Sobrecarga de Funções e Operadores

O operador "+" em Java denota concatenação de cadeia ou adição, dependendo dos tipos de seus operandos.

As funções definidas pelo usuário também podem ser sobrecarregadas, como em:

```
void err() { ··· }
void err(String s) { ··· }
```

Uma regra de síntese de tipo para funções sobrecarregadas:

if f pode ter tipo $s_i \to t_i$, para $1 \leq i \leq n$, onde $s_i \neq s_j$ para $i \neq j$
and x tem tipo s_k, para algum $1 \leq k \leq n$
then expressão $f(x)$ possui tipo t_k (3)

··· Sobrecarga de Funções e Operadores

O método código numérico(Seção 6.1.2) pode ser aplicado às expressões de tipo para resolver eficientemente a sobrecarga baseada nos tipos dos argumentos.

Dado um DAG representando uma expressão de tipo, a cada nó é atribuído um índice inteiro, chamado código numérico.

Usando o Algoritmo 6.3 constrói-se uma assinatura para um nó, consistindo em seu rótulo e nos códigos numéricos de seus filhos, ordenados da esquerda para a direita.

Assinatura para uma função: nome da função + tipos de seus argumentos.

Observação: nem sempre é possível resolver a sobrecarga examinando apenas os argumentos de uma função.

Em Ada, em vez de um tipo único, uma subexpressão isolada pode ter um conjunto de tipos possíveis para os quais o contexto precisa fornecer informações suficientes para reduzir as opções a um único tipo.

Inferência de Tipo e Funções Polimórficas

A inferência de tipo é útil para linguagens fortemente tipadas, como ML, e que não exige que os nomes sejam declarados antes de serem usados.

Inferência de tipo garante que nomes sejam usados de forma coerente.

O termo "polimórfico" refere-se a qualquer fragmento de código que pode ser executado com argumentos de diferentes tipos.

Nesta seção, consideramos o polimorfismo paramétrico.

Inferência de Tipo e Funções Polimórficas

Exemplo: definição de uma função $length$ em ML.

Tipo de $length$: "para qualquer tipo α, $length$ mapeia uma lista de elementos de tipo α para um inteiro".

$$\text{fun } length(x) \ = \text{if } null(x) \text{ then } 0 \text{ else } length(tl(x)) + 1;$$

Aplicação de $length$ a dois tipos diferentes de listas:

$$length([\texttt{"sun"}, \texttt{"mon"}, \texttt{"tue"}]) + length([10, 9, 8, 7]) \quad (4)$$

Usando o símbolo $\forall$ e o construtor de tipo $list$, o tipo de $length$ pode ser escrito como $\forall\alpha.\ list(\alpha) \rightarrow integer$

Inferência de Tipo e Funções Polimórficas

O símbolo $\forall$ é o quantificador universal, e a variável de tipo à qual ele é aplicado é considerada como estando ligada por ele.

Variáveis ligadas podem ser renomeadas, desde que todas as ocorrências da variável sejam renomeadas.

Portanto, a expressão de tipo: $\forall\beta.\ list(\beta) \rightarrow integer$ é equivalente a

$$\forall\alpha.\ list(\alpha) \rightarrow integer \quad (5)$$

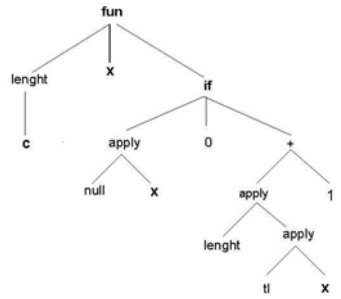
Uma expressão de tipo contendo um símbolo $\forall$ será referenciada informalmente como um "tipo polimórfico".

Inferência de Tipo e Funções Polimórficas

Usando implicitamente as regras de inferência de tipo a seguir, o exemplo infere informalmente um tipo para $length$

if $f(x)$ é uma expressão,
then para algum α e β, f possui tipo $\alpha \rightarrow \beta$ and x possui tipo α

Árvore de sintaxe abstrata

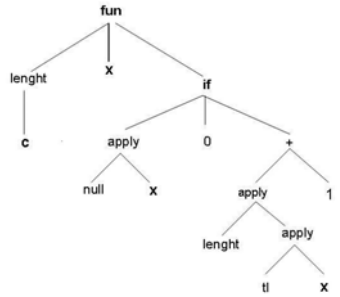


Definição de $length$

$$\text{fun } length(x) \ = \\ \text{if } null(x) \text{ then } 0 \text{ else } length(tl(x))$$

Inferência de Tipo e Funções Polimórficas

Árvore de sintaxe abstrata



A partir do corpo da função $length$, é possível inferir seu tipo.

Considere os filhos do nó rotulado com if, da esquerda para a direita.

Como $null$ espera ser aplicado a listas, x deve ser uma lista.

Usando a variável α como representante do tipo dos elementos da lista; ou seja, x tem o tipo "lista de α".

Se $null(x)$ for verdadeiro, então $length(x)$ é 0.

Então, o tipo de $length$ precisa ser "função da lista de a para inteiro".

Esse tipo inferido é consistente com o uso de $length$ na parte do else, $length(tl(x)) + 1$.

Inferência de Tipo e Funções Polimórficas

Como as variáveis podem aparecer nas expressões de tipo, temos de reexaminar a noção de equivalência de tipos.

Suponha que E_1 do tipo $s \rightarrow s'$ seja aplicado a E_2 do tipo t.

Em vez de simplesmente determinar a igualdade de s e t, devemos "unificá-los".

Informalmente, verifica-se se s e t, podem tornar-se estruturalmente equivalentes, substituindo as variáveis de tipo em s e t por expressões de tipo.

Inferência de Tipo e Funções Polimórficas

Uma substituição é um mapeamento de variáveis de tipo para expressões de tipo.

Escrevemos $S(t)$ como o resultado de aplicar a substituição S às variáveis na expressão de tipo t.

Duas expressões de tipo t_1 e t_2 se unificam se houver alguma substituição S tal que $S(t_1) = S(t_2)$.

Na prática, estamos interessados em um unificador mais geral, o qual é uma substituição que impõe menos restrições sobre as variáveis nas expressões.

Inferência de Tipo e Funções Polimórficas

Algoritmo 6.16: Inferência de tipo para funções polimórficas.

ENTRADA: Um programa consistindo em uma seqüência de definições de função seguido por uma expressão a ser avaliada. Uma expressão é composta de aplicações de função e nomes, onde os nomes podem ter tipos polimórficos predefinidos.

SAÍDA: Tipos inferidos para os nomes no programa.

MÉTODO: Por simplicidade, vamos tratar apenas as funções unárias.

O tipo de uma função $f(x_1, x_2)$ com dois parâmetros pode ser representado por uma expressão de tipo $s_1 \times s_2 \rightarrow t$, onde s_1 e s_2 são os tipos de x_1 and x_2, respectivamente, e t é o tipo do resultado $f(x_1, x_2)$.

Inferência de Tipo e Funções Polimórficas

Algoritmo 6.16: Inferência de tipo para funções polimórficas.

- Para uma definição de função $\text{fun id}_1(\text{id}_2) = E$, crie novas variáveis de tipo α e β.

  Associe o tipo $\alpha \to \beta$ com a função id_1, e o tipo α a com o parâmetro id_2.

  Depois, infira um tipo para a expressão E. Suponha que a denote o tipo s e b denote o tipo t após a inferência de tipo para E.

  O tipo inferido da função id_1 é $s \to t$. Ligue quaisquer variáveis de tipo que permanecerem sem restrições em $s \to t$ por quantificadores $\forall$.

Inferência de Tipo e Funções Polimórficas

Algoritmo 6.16: Inferência de tipo para funções polimórficas.

- Para uma aplicação de função $E_1(E_2)$, infira os tipos para E_1 e E_2.

  Uma vez que E_1 é usado como uma função, seu tipo precisa ter a forma $s \to s'$.

  Considere que t seja o tipo inferido de E_1.

  Unifique s e t.

  Se a unificação falhar, a expressão tem um erro de tipo.

  Caso contrário, o tipo inferido de $E_1(E_2)$ is s'.

Inferência de Tipo e Funções Polimórficas

Algoritmo 6.16: Inferência de tipo para funções polimórficas.

- Para cada ocorrência de uma função polimórfica, substitua as variáveis ligadas em seu tipo por novas variáveis distintas e remova os quantificadores $\forall$.

  A expressão de tipo resultante é o tipo inferido dessa ocorrência.

- Para um nome que é encontrado pela primeira vez, introduza uma variável nova para seu tipo.

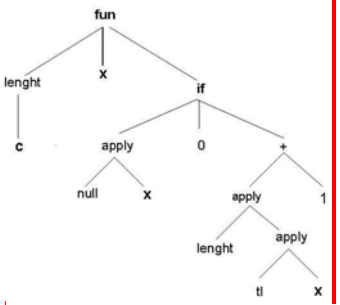
Inferência de Tipo e Funções Polimórficas

Inferindo um tipo para a função *length*

Linha	Expressão	:	Tipo	Unificação
1)	$length$	:	$\beta \to \gamma$	
2)	x	:	β	
3)	if	:	$boolean \times \alpha_i \times \alpha_i \to \alpha_i$	
4)	$null$	:	$list(\alpha_n) \to boolean$	
5)	$null(x)$	:	$boolean$	$list(\alpha_n) = \beta$
6)	0	:	$integer$	$\alpha_i = integer$
7)	$+$	:	$integer \times integer \to integer$	
8)	tl	:	$list(\alpha_t) \to list(\alpha_t)$	
9)	$tl(x)$	:	$list(\alpha_t)$	$list(\alpha_t) = list(\alpha_n)$
10)	$length(tl(x))$	:	γ	$\gamma = integer$
11)	1	:	$integer$	
12)	$length(tl(x)) + 1$	:	$integer$	
13)	if($\cdots$)	:	$integer$	

Inferência de Tipo e Funções Polimórficas

Árvore de sintaxe abstrata



A raiz da árvore de sintaxe é para uma definição de função, de modo que
- são introduzidas as variáveis β e γ,
- são associados o tipo $\beta \rightarrow \gamma$ com a função $length$,
- o tipo β com x.

Após a verificação da definição da função, α_n permanece no tipo de $length$.

Como nenhuma suposição foi feita sobre α_n, qualquer tipo pode ser substituído por ele quando a função for usada. Portanto, nós a tornamos uma variável ligada e escrevemos: $\forall\alpha_n.\ list(\alpha_n) \rightarrow integer$ para o tipo de length.

Um Algoritmo para Unificação

Implementação da unificação baseada na teoria dos grafos, onde

- Tipos são representados por grafos.
- Variáveis de tipo são representadas por folhas.
- Construtores de tipo são representados por nós interiores.

Os nós são agrupados em classes de equivalência; se dois nós estiverem na mesma classe de equivalência, então as expressões de tipo que eles representam precisam ser unificáveis. Assim, todos os nós interiores na mesma classe representam o mesmo construtor de tipo, e seus filhos correspondentes precisam ser equivalentes.

··· Um Algoritmo para Unificação

Exemplo: Considere as duas expressões de tipo

$$((\alpha_1 \rightarrow \alpha_2) \times list(\alpha_3)) \rightarrow list(\alpha_2)$$
$$((\alpha_3 \rightarrow \alpha_4) \times list(\alpha_3)) \rightarrow \alpha_5$$

A substituição S a seguir é o unificador mais geral para essas expressões

x	$S(x)$
α_1	α_1
α_2	α_2
α_3	α_1
α_4	α_2
α_5	$list(\alpha_2)$

Essa substituição mapeia as duas expressões de tipo para a seguinte expressão:

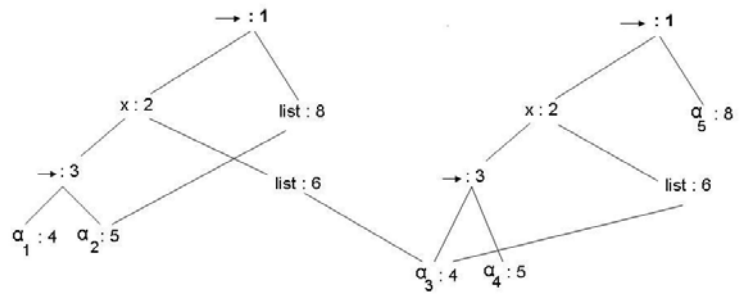
$$((\alpha_1 \rightarrow \alpha_2) \times list(\alpha_1)) \rightarrow list(\alpha_2)$$

··· Um Algoritmo para Unificação

$$((\alpha_1 \rightarrow \alpha_2) \times list(\alpha_3)) \rightarrow list(\alpha_2)$$
$$((\alpha_3 \rightarrow \alpha_4) \times list(\alpha_3)) \rightarrow \alpha_5$$

As duas expressões são representadas pelos dois nós rotulados com $\rightarrow$: 1.

Os valores inteiros nos nós indicam as classes de equivalência a que os nós pertencem depois que os nós numerados com 1 forem unificados.



··· Um Algoritmo para Unificação

Algoritmo 6.19: A unificação de um par de nós em um grafo de tipo.

ENTRADA: Um grafo representando um tipo e um par de nós m e n a serem unificados.

SAÍDA: Valor booleano true se as expressões representadas pelos nós m e n forem unificadas, caso contrário, false.

··· Um Algoritmo para Unificação

MÉTODO:

Um nó é implementado por um registro com campos para um operador binário e apontadores para os filhos à esquerda e à direita.

Os conjuntos de nós equivalentes são mantidos usando-se o campo set.

Um nó em cada classe de equivalência é escolhido para ser o único representante da classe de equivalência, fazendo seu campo set conter um apontador nulo.

Os campos set dos nós restantes na classe de equivalência apontarão, possivelmente indiretamente por meio de outros nós no conjunto, para o representante.

Inicialmente, cada nó n está em uma classe de equivalência por si só, com n como seu próprio nó representante.

··· Um Algoritmo para Unificação

Algoritmo 6.19: A unificação de um par de nós em um grafo de tipo.

```
boolean unify(Node m, Node n) {
    s = find(m); t = find(n);
    if ( s = t ) return true;
    else if ( nós s and t representam o mesmo tipo básico) return true;
    else if (s é um nó operador com filhos s1 e s2 and
             t é um nó operador com filhos t1 e t2) {
        union(s, t);
        return unify(s1, t1) and unify(s2, t2);
    }
    else if (s ou t) representa uma variável {
        union(s, t);
        return true;
    }
    else return false;
}
```


··· Um Algoritmo para Unificação

O algoritmo de unificação usa as operações a $find$ e $union$ nos nós:

- **find(n)** retorna o nó representante da classe de equivalência contendo correntemente o nó n.
- **union(m,n)** combina classes de equivalência contendo os nós m e n.

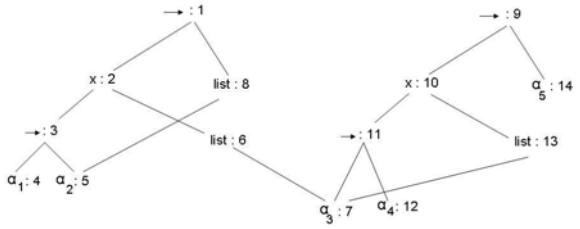
  Se um dos representantes das classes de equivalência de m e n for um nó que não representa variável, $union$ faz com que esse nó seja o representante para a classe de equivalência combinada; caso contrário, $union$ faz com que um dos representantes originais seja o novo representante.

  Essa assimetria na especificação de union é importante porque uma variável não pode ser usada como representante de uma classe de equivalência para uma expressão contendo um construtor de tipo ou um tipo básico.

  Caso contrário, duas expressões não equivalentes poderiam ser unificadas por meio dessa variável.

··· Um Algoritmo para Unificação

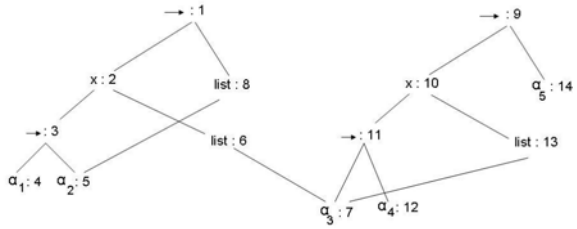
Suponha que as duas expressões $((\alpha_1 \to \alpha_2) \times list(\alpha_3)) \to list(\alpha_2)$
$((\alpha_3 \to \alpha_4) \times list(\alpha_3)) \to \alpha_5$
sejam representadas neste grafo, e cada nó esteja em sua própria classe de equivalência.



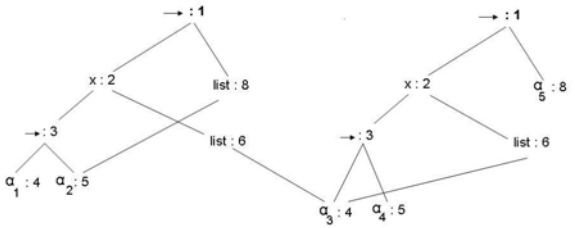
Quando o Algoritmo 6.19 é aplicado para computar $unify(1, 9)$, ele nota que os nós 1 e 9 representam o mesmo operador. Portanto, ele intercala 1 e 9 na mesma classe de equivalência e chama $unify(2, 10)$ e $unify(8, 14)$.

··· Um Algoritmo para Unificação

Grafo inicial com cada nó em sua própria classe de equivalência

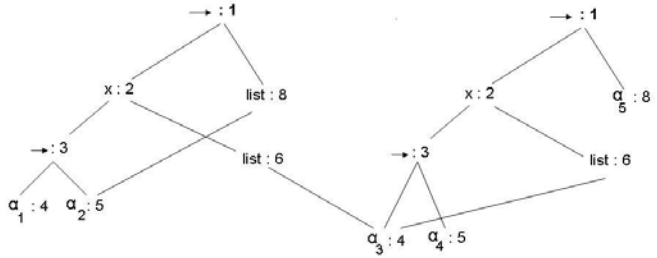


O resultado da computação de $unify(1, 9)$ é o grafo



··· Um Algoritmo para Unificação

Resultado da computação de $unify(1, 9)$ é o grafo

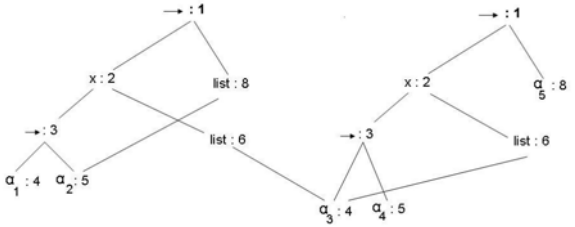


Se o Algoritmo 6.19 retornar $true$, podemos construir uma substituição S que funciona como o unificador, conforme mostrado a seguir.

Para cada variável α, $find(\alpha)$ fornece o nó n que é o representante da classe de equivalência de α. A expressão representada por n é $S(\alpha)$.

··· Um Algoritmo para Unificação

Por exemplo, vemos que o representante para α_3 é nó 4, que representa α_1.
O representante para α_5 é o nó 8, que representa $list(\alpha_2)$.



Substituição resultante S é como em

x	$S(x)$
α_1	α_1
α_2	α_2
α_3	α_1
α_4	α_2
α_5	$list(\alpha_2)$

Essa substituição mapeia as duas expressões de tipo para a seguinte expressão:

$$((\alpha_1 \to \alpha_2) \times list(\alpha_1)) \to list(\alpha_2)$$

6.6 FLUXO DE CONTROLE

A tradução de comandos como os comandos $if - else$ e $while$ está ligada à tradução das expressões boolianas.

Nas linguagens de programação, as expressões boolianas normalmente são usadas para:

1. alterar o fluxo de controle
2. computar valores lógicos.

··· Fluxo de Controle

1. Computar valores lógicos. Uma expressão booliana pode representar true (verdadeiro) ou false (falso) como valores.

   Essas expressões boolianas podem ser avaliadas em analogia com as expressões aritméticas usando instruções de três endereços com operadores lógicos.
   Exemplo: uma expressão após a palavra-chave if é usada para alterar o fluxo de controle, enquanto uma expressão do lado direito de um comando de atribuição é usada para denotar um valor lógico.

2. Alterar o fluxo de controle. As expressões boolianas são usadas como expressões condicionais em comandos que alteram o fluxo de controle.

   O valor dessas expressões boolianas é dado implicitamente pela posição atingida em um programa.

   Exemplo: em `if (E) S`, a expressão E deve ser verdadeira se o comando S for alcançado.

Expressões Boolianas

As expressões boolianas são compostas dos operadores booleanos.

Conforme a conveniência, usaremos &&, ||, e !, usando a convenção da linguagem C para os operadores AND, OR e NOT, respectivamente aplicados a elementos que são variáveis boolianas ou expressões relacionais.

As expressões relacionais são da forma E_1 rel E_2,

onde E_1 e E_2 são expressões aritméticas.

··· Expressões Boolianas

Nesta seção, consideramos as expressões boolianas geradas pelas gramáticas:

$$
\begin{array}{lll}
\text{B} \rightarrow & B \;||\; B & \\
| & B \;\&\&\; B & \\
| & !\; B & \\
| & (\; B \;) & \\
| & E \text{ rel } E & \\
| & \text{true} & \\
| & \text{false} &
\end{array}
\qquad
\begin{array}{l}
\text{E} \rightarrow \text{id} := \text{E} \\
\text{E} \rightarrow \text{E and E} \\
\text{E} \rightarrow \text{E or E} \\
\text{E} \rightarrow \text{not E} \\
\text{E} \rightarrow \text{(E)} \\
\text{E} \rightarrow \text{id relop id} \\
\text{E} \rightarrow \text{id}
\end{array}
$$

Usamos o atributo $\text{rel}.op$ para indicar qual dos seis operadores de comparação $<$, $<=$, $=$, $!=$, $>$, ou $>=$ é representado por rel.

··· Expressões Booleanas

Assumimos que || e && sejam associativos à esquerda, e que || tenha a precedência mais baixa, então && e então !.

Dada a expressão B_1 || B_2, se determinarmos que B_1 é verdadeiro, podemos concluir que a expressão inteira é verdadeira sem ter de avaliar B_2 .

Dado $B_1 \&\& B_2$, se B_1 for falso, então a expressão inteira é falsa.

"A semântica das L.P. determinam quando todas as partes da expressão booleana devem ser avaliadas".

··· Expressões Booleanas - 1. Representação Numérica

true $\equiv$ 1 false $\equiv$ 0

A || B && C

T_1 := B && C
T_2 := A || T_1

A < B $\equiv$ if A < B then 1 else 0

(1) if A < B goto (4)
(2) T := 0
(3) goto (5)
(4) T := 1
(5)

A < B || C

(1) if A < B goto (4)
(2) T_1 := 0
(3) goto (5)
(4) T_1 := 1
(5) T_2 := T_1 or C

··· Expressões Booleanas - 1.Representação Numérica

- Rotinas Semânticas

emit: coloca o código de 3 endereços na saída; incrementa nextstat após a produção de cada comando de 3 endereços.

nextstat: fornece o índice do próximo comando de 3 endereços na próxima sequência de saída.

··· Expressões Booleanas - 1.Representação Numérica

E → id { E.place := id.place }

E → E_1 || E_2 { E.place := newtemp
emit (E.place ":=" E_1.place "||" E_2.place) }

E → E_1 && E_2 { E.place := newtemp
emit(E.place ":=" E_1.place "&&" E_2.place) }

E → ! E_1 { E.place := newtemp
emit(E.place ":=" "!" E_1.place) }

E → (E_1) { E.place := E_1.place }

··· Expressões Boolianas - 1.Representação Numérica

E → id_1 rel.op id_2 { E.place := newtemp;
emit("if" id_1.place rel.op id_2.place
"goto" nextstat + 3);
emit(E.place ":=" "0");
emit("goto" nextstat + 2);
emit(E.place ":=" "1");

E → true { E.place := newtemp;
emit(E.place ":=" "1") }

E → false { E.place := newtemp
emit(E.place ":=" "0"); }

··· Expressões Boolianas

2.Código de Curto-circuito

No código de curto-circuito (ou desvio), os operadores booleanos &&, ||, e ! (ou AND, OR e NOT) são traduzidos para desvios.

Os próprios operadores não aparecem no código; em vez disso, o valor de uma expressão booleana é representado por uma posição na seqüência de código.

··· Expressões Boolianas - 2.Código de Curto-circuito

Exemplo: O comando

```
if ( x < 100 || x > 200 && x != y ) x = 0;
```

poderia ser traduzido para o código

```
    if x < 100 goto L2
    ifFalse x > 200 goto L1
    ifFalse x != y goto L1
L2: x = 0
L1:
```

Nessa tradução, a expressão booliana é verdadeira se o controle alcançar o rótulo L_2.

Se a expressão for falsa, o controle vai imediatamente para L_1, saltando L_2 e a atribuição `x = 0`.

··· Expressões Boolianas - 2.Código de Curto-circuito

Tradução de uma expressão booleana em código de 3 endereços sem gerar código para qualquer um dos operadores booleanos.

Representa-se os valores das expressões pela posição na sequência de código.

Exemplo:
if A < B or C < D and E < F
(100) if A < B goto (103)
(101) T_1 := 0
(102) goto (104)
(103) T_1 := 1
(104) if C < D goto (107)
(105) T_2 := 0
(106) goto (108)
(107) T_2 := 1
(108) if E < F goto (111)
(109) T_3 := 0
(110) goto (112)
(111) T_3 := 1
(112) T_4 := T_2 and T_3
(113) T_5 := T_1 or T_4

Comandos de Fluxo de Controle

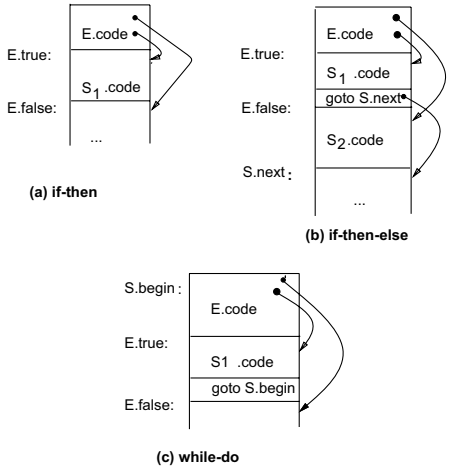
Agora, vamos considerar a tradução de expressões boolianas para um código de três endereços no contexto dos comandos como aqueles gerados pela gramática a seguir:

$$S \rightarrow \text{if } (E) S_1$$
$$S \rightarrow \text{if } (E) S_1 \text{ else } S_2$$
$$S \rightarrow \text{while } (E) S_1$$

Nessas produções, o não-terminal E representa uma expressão booleana e o não-terminal S representa um comando.

··· Comandos de Fluxo de Controle

$$S \rightarrow \text{if } (E) S_1$$
$$S \rightarrow \text{if } (E) S_1 \text{ else } S_2$$
$$S \rightarrow \text{while } (E) S_1$$



··· Comandos de Fluxo de Controle

Produções	Regras	semânticas
S → if (E) S_1	E.true:=	newlabel();
	E.false:=	S.next;
	S_1.next:=	S.next;
	S.code:=	E.code || `label`(E.true ":") || S_1.code
S→if (E) S_1 else S_2	E.true:=	newlabel();
	E.false:=	newlabel();
	S_1.next:=	S.next;
	S_2.next:=	S.next;
	S.code:=	E.code||`label`(E.true:":")||S_1.code || `gen`("goto" S.next) || `label`(E.false ":") || S_2.code

··· Comandos de Fluxo de Controle

Produções	Regras	semânticas
S → while (E) S_1	S.begin:=	newlabel();
	E.true:=	newlabel();
	E.false:=	S.next;
	S_1.next:=	S.begin;
	S.code:=	`label` (S.begin ":") || E.code || `label`(E.true ":") || S_1.code || `gen`("goto" S.begin)

E.true: rótulo para o qual o fluxo flui se E é verdadeiro.

E.false: rótulo para o qual o fluxo flui se E é falso.

Idéia: E ≡ a < b → if a < b goto E.true goto E.false
E ≡ E1 or E2 → if E1 = true then E igual a true.

··· Comandos de Fluxo de Controle

Produções	Regras semânticas
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow$ assign	$S.code =$ assign$.code$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$

Tradução de Fluxo de Controle de Expressões Boolianas

Produções	Regras semânticas
E $\rightarrow E_1$ or E_2	E_1.true := E.true; E_1.false := newlabel(); E_2.true := E.true; E_2.false := E.false; E.code:= E_1.code||label(E_1.false ":") || E_2.code
E $\rightarrow E_1$ and E_2	E_1.true := newlabel(); E_1.false := E.false; E_2.true := E.true; E_2.false := E.false; E.code:= E_1.code || label(E_1.true ":") || E_2.code

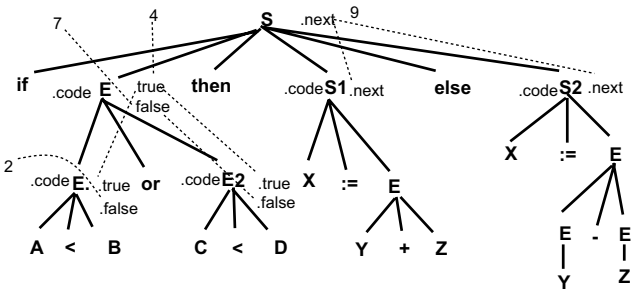
··· Tradução de Fluxo de Controle de Expressões Boolianas

Produções	Regras semânticas
E $\rightarrow$ (E_1)	E_1.true := E.true; E_1.false := E.false; E.code := E_1.code
E $\rightarrow id_1$ rel id_2	E.code := gen("if" id_1.place rel.op id_2.place "goto" E.true) | | gen ("goto" E.false)
E $\rightarrow$ true	E.code := gen ("goto" E.true)
E $\rightarrow$ false	E.code := gen ("goto" E.false)
E $\rightarrow$ not E_1	E_1.true := E.false; E_1.false := E.true; E.code := E_1.code

Exemplo 1

if A < B or C < D then X := Y + Z else X := Y − Z

(1) if A<B goto (4)
(2) if C<D goto (4)
(3) goto (7)
(4) T_1:= Y + Z then
(5) X:= T_1
(6) goto (9)
(7) T_2:= Y − Z else
(8) X:= T_2
(9)



Exemplo 2

Considere o comando a seguir

```
if( x < 100 || x > 200 && x != y ) x = 0;
```
(6)

Código gerado:

```
    if x < 100 goto L2
    goto L3
L3: if x > 200 goto L4
    goto L1
L4: if x != y goto L2
    goto L1
L2: x = 0
L1:
```


Evitando gotos Redundantes

Em `if( x < 100 || x > 200 && x != y ) x = 0;`
a comparação $x > 200$ **é traduzida para**

```
    if x > 200 goto L4
    goto L1
L4: ···
```

Em vez disso, considere a instrução:

```
    ifFalse x > 200 goto L1
L4: ···
```

Essa instrução `ifFalse` **tira proveito do fluxo natural de uma instrução para a próxima na seqüência, de modo que o controle simplesmente "segue" para o rótulo** L_4 **se** $x > 200$**, evitando assim um desvio.**

··· Evitando gotos Redundantes

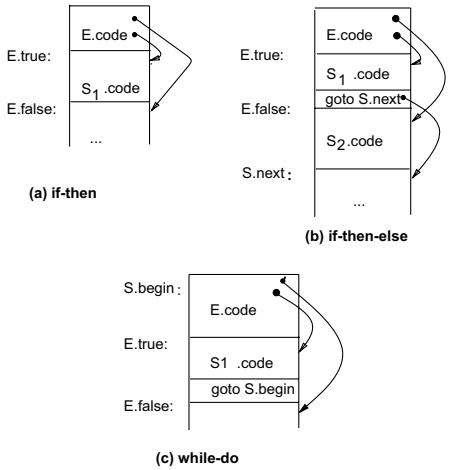
Nos leiautes de código para os comandos if **e** $while$ **o código para o comando** S_1 **segue imediatamente o código para a expressão booleana** E**.**

Usando um rótulo especial $fall$**, ou seja, "não gere nenhum desvio", é possível adaptar as regras semânticas dadas para permitir que o controle siga do código de** E **para o código de** S_1**.**

As novas regras semânticas para a produção $S \rightarrow \text{if } (E)\ S_1$ **definem** $E.true$ **como** $fall$**:**

$$E.true = fall$$
$$E.false = S_1.next = S.next$$
$$S.code = E.code \ || \ S_1.code$$

Da mesma forma, as regras para `if-else` **e** `while` **também definem** $E.true$ **como** $fall$**.**



··· Evitando gotos Redundantes

Agora, adaptamos as regras semânticas para as expressões boolianas, para permitir que o controle siga em frente sempre que for possível.

$$test = \boldsymbol{E}_1.addr \ \textbf{rel.}op \ \boldsymbol{E}_2.addr$$
$$\begin{aligned} s = \ & \textbf{if } \boldsymbol{E}.true \neq fall \textbf{ and } \boldsymbol{E}.false \neq fall \textbf{ then} \\ & \quad gen(\text{'if'}\ test\ \text{'goto'}\ \boldsymbol{E}.true) \ || \ gen(\text{'goto'}\ \boldsymbol{E}.false) \\ & \textbf{else if } \boldsymbol{E}.true \neq fall \textbf{ then } gen(\text{'if'}\ test\ \text{'goto'}\ \boldsymbol{E}.true) \\ & \textbf{else if } \boldsymbol{E}.false \neq fall \textbf{ then } gen(\text{'ifFalse'}\ test\ \text{'goto'}\ \boldsymbol{E}.false) \\ & \textbf{else } '' \end{aligned}$$
$$\boldsymbol{E}.code = \boldsymbol{E}_1.code \ || \ \boldsymbol{E}_2.code \ || \ \boldsymbol{s}$$

Observação: Em C e Java, as expressões podem conter atribuições dentro delas, então precisa ser gerado código para as subexpressões E_1 **e** E_2**, mesmo que** `E.true` **e** `E.false` **sejam** $fall$**.**

Se desejar, o código morto pode ser eliminado durante a fase de otimização.

··· Evitando gotos Redundantes

Nas novas regras para $E \rightarrow E_1 \;||\; E_2$ em

$$
\begin{aligned}
E_1.true &= \text{if } E.true \neq fall \text{ then } E.true \text{ else } newlabel() \\
E_1.false &= fall \\
E_2.true &= E.true \\
E_2.false &= E.false \\
E.code &= \text{if } E.true \neq fall \text{ then } E_1.code \;||\; E_2.code \\
&\quad\;\; \text{else } E_1.code \;||\; E_2.code \;||\; label(E_1.true)
\end{aligned}
$$

Observe que o significado do rótulo $fall$ para E é diferente de seu significado para E_1.

Suponha que `E.true` seja $fall$. Embora E seja verdadeiro se E_1 também o for, $E_1.true$ precisa garantir que o controle desvia sobre o código de E_2 para chegar à próxima instrução após E.

Por outro lado, se E_1 for avaliado como falso, o valor verdade de E é determinado pelo valor de E_2, então as regras dadas garantem que $E_1.false$ corresponde ao controle seguindo de E_1 para o código de E_2.

··· Evitando gotos Redundantes

Exemplo 3: Com as novas regras usando o rótulo especial $fall$

```
if( x < 100 || x > 200 && x != y ) x = 0;
```

é traduzido no código

```
     if x < 100 goto L2
     ifFalse x > 200 goto L1
     ifFalse x != y goto L1
L2:  x = 0
L1:
```

Observação: a diferença do Exemplo 2 para Exemplo 3 é que o atributo herdado $E.true$ é $fall$ quando as regras semânticas para $E \rightarrow E_1 \;||\; E_2$ são aplicadas ($E.false$ is L_1).

Valores Boolianos e Código de Desvio

Uma expressão booliana é usada para alterar o fluxo de controle dos comandos, e também pode ser avaliada pelo seu valor, como nos comandos de atribuição: `x = true;` ou `x = a<b;`.

Uma forma de tratar estes dois papéis é primeiro construir uma árvore de sintaxe para as expressões, usando uma das seguintes abordagens:

1. Usar dois passos. Construa uma árvore de sintaxe completa para a entrada e então percorra a árvore em profundidade, computando as traduções especificadas pelas regras semânticas.

2. Usar um passo para comandos, mas dois passos para expressões. Traduz-se E de $while(E)\ S_1$ antes de S_1 ser examinado. A tradução de E, contudo, seria feita construindo-se sua árvore de sintaxe e então percorrendo-a.

··· Valores Boolianos e Código de Desvio

Dada a gramática para expressões:

$$
\begin{aligned}
S &\rightarrow \text{id} = E \;;\; | \;\text{if (}\; E \;\text{)}\; S \;|\; \text{while (}\; E \;\text{)}\; S \;|\; S\; S \\
E &\rightarrow E \;||\; E \;|\; E \,\&\&\, E \;|\; E \;\text{rel}\; E \;|\; E + E \;|\; (E) \;|\; \text{id} \;|\; \text{true} \;|\; \text{false}
\end{aligned}
$$

A atribuição `x = a < b && c < d` pode ser implementada pelo código

```
     ifFalse a < b goto L1
     ifFalse c < d goto L1
     t = true
     goto L2
L1:  t = false
L2:  x = t
```

6.7 REMENDOS (*BACKPATCHING*)

Um problema importante quando se gera código para expressões booleanas e comandos de fluxo de controle é o de casar uma instrução de desvio com o destino do desvio.

Exemplo: a tradução da expressão booleana B em `if ( B ) S` contém um desvio, no caso de B ser falso, para a instrução seguinte ao código de S.

Em uma tradução de um passo, B precisa ser traduzido antes que S seja examinado.

Qual, então, é o destino do `goto` que desvia para o código após S?

··· 6.7 REMENDOS (*BACKPATCHING*)

Qual, então, é o destino do `goto` que desvia para o código após S?

Soluções:

1. Passar **rótulos como atributos herdados** para onde as instruções de desvio relevantes são geradas. MAS um passo separado é então necessário para vincular os rótulos aos endereços.

2. Abordagem complementar, usar **remendo**, na qual são passadas **listas de desvios como atributos sintetizados**.
   Quando um desvio é gerado, o objeto do desvio fica temporariamente sem especificação.

   Cada desvio desse tipo é colocado em uma lista de desvios cujos rótulos serão preenchidos quando o rótulo apropriado puder ser determinado. Todos os desvios em uma lista possuem o mesmo rótulo de destino.

Geração de Código em UM PASSO Usando Remendo

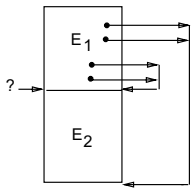
- E.trueList: Apontador para lista de "quadruplas incompletas" e que são desvios para o caso de E = true.
- E.falseList: Idem, porém para E = false.
- *Subrotinas Auxiliares :*

makelist(q): cria a lista q | →, onde q = endereço de uma quádrupla. Retorna endereço da lista.

merge(p_1, p_2): une duas listas cujos endereços são p_1 e p_2. Retorna o endereço da lista resultante.

backpatch(p, i): coloca o valor "i" em todas quádruplas da lista p.

··· Geração de Código em UM PASSO Usando Remendo - PROBLEMAS



- Soluções:

1. **Fatoração:** E → E' E_2
   E' → E_1 and { backpatch(E_1.trueList, NEXTQUAD) }

2. **Marcação:**
   E → E_1 and M E_2 {backpatch(E_1.trueList, M.QUAD)}
   M → $\mathcal{E}$ { M.QUAD := NEXTQUAD }

Remendos para Expressões Boolianas

1) E $\rightarrow$ id

2) E $\rightarrow$ id_1 rel id_2

3) E $\rightarrow$ not E_1

4) E $\rightarrow$ E_1 and M E_2

5) E $\rightarrow$ E_1 or M E_2

6) M $\rightarrow$ $\mathcal{E}$

7) E $\rightarrow$ (E_1)
8) E $\rightarrow$ true
9) E $\rightarrow$ false

... Remendos para Expressões Boolianas

1) E $\rightarrow$ id { E.trueList := makelist(NEXTQUAD)
E.falseList := makelist(NEXTQUAD + 1)
emit("if" id.place "goto _")
emit("goto _") }

2) E $\rightarrow$ id_1 rel id_2 { E.trueList := makelist(NEXTQUAD)
{ E.falseList := makelist(NEXTQUAD+1)
emit("if" id_1.place rel.op id_2.place "goto_")
emit("goto _") }

3) E $\rightarrow$ not E_1 { E.trueList := E_1.falseList }
E.falseList := E_1.trueList }

... Remendos para Expressões Boolianas

4) E $\rightarrow$ E_1 and M E_2 { backpatch(E_1.trueList, M.QUAD)
E.trueList := E_2.trueList
E.falseList := merge(E_1.falseList, E_2.falseList)

5) E $\rightarrow$ E_1 or M E_2 { backpatch(E_1.falseList, M.QUAD)
E.trueList := merge(E_1.trueList, E_2.trueList)
E.falseList := E_2.falseList }

6) M $\rightarrow$ $\mathcal{E}$ { M.QUAD := NEXTQUAD }

... Remendos para Expressões Boolianas

7) E $\rightarrow$ (E_1) { E.trueList := E_1.trueList
E.falseList := E_1.falseList }

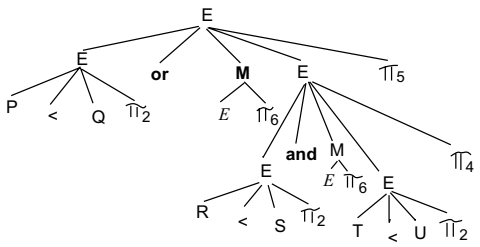
8) E $\rightarrow$ true { E.trueList := makelist(NEXTQUAD)
E.falseList := makelist()
emit("goto _" }

9) E $\rightarrow$ false { E.falseList := makelist(NEXTQUAD)
E.trueList := makelist()
emit("goto _" }

... Remendos para Expressões Booleanas

- **Exemplo:** P < Q or R < S and T < U

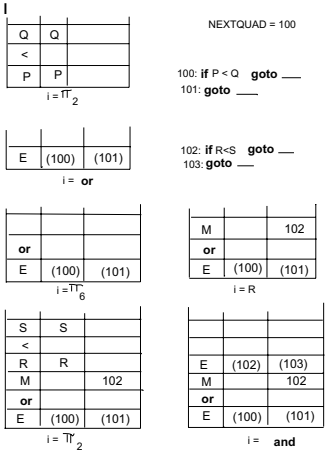
1 E → id π_1
2 E → id rel id π_2
3 E → not id π_3
4 E → E and M E π_4
5 E → E or M E π_5
6 M → $\mathcal{E}\pi_6$



P < Q π_2 or π_6 R < S π_2 and π_6 T < U π_2 Π_4 π_5
π_2 π_6 π_2 π_6 π_2 π_4 π_5 = inverso da derivação mais à direita.

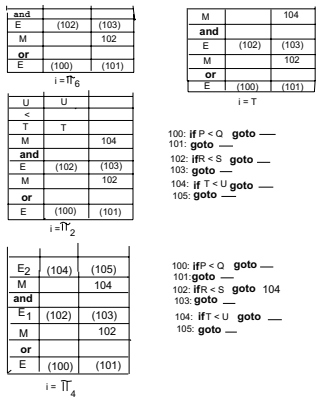
... Remendos para Expressões Booleanas - Exemplo:

P < Q π_2 or π_6 R < S π_2 and π_6 T < U π_2 π_4 π_5
↑
i



... Remendos para Expressões Booleanas - Exemplo:

P < Q π_2 or π_6 R<S $\pi_2$and π_6 T<U π_2 π_4 π_5
↑ i



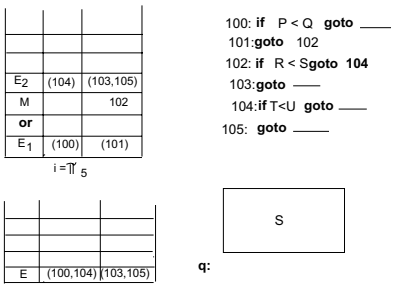
backpatch ((102), 104)

(102): como argumento ≡ apontador para lista contendo somente 102.

backpatch preenche goto incompleto do cmd 102 com 104.

... Remendos para Expressões Booleanas - Exemplo:

P<Q π_2 or π_6 R<S π_2 and π_6 T<U π_2 π_4 π_5
↑
i



100: if P < Q goto 106
101: goto 102
102: if R < S goto 104
103: goto q
104: if T < U goto 106
105: goto q
106:

backpatch ((101), 102)

S → while E do M S { backpatch(E.trueList,M.QUAD) ≡ 106:
backpatch(E.falseList, NEXTQUAD) ≡ q: ···

· · · Remendos para Expressões Booleanas - Exemplo:

Considerações:

A expressão inteira será verdadeira se e somente se **os gotos dos comandos 100 e 104 forem atingidos.**

A expressão inteira será falsa se e somente se **os gotos dos comandos 103 e 105 forem atingidos.**

Estas instruções terão os gotos preenchidos mais à frente durante a compilação, no momento em que se souber o que deve ser feito (dependo do resultado falso ou verdadeiro da expressão).

· · · Remendos para Expressões Booleanas - Outro Exemplo:

Dada a expressão: $x < 100 \;||\; x > 200 \;\&\&\; x \;!=\; y$

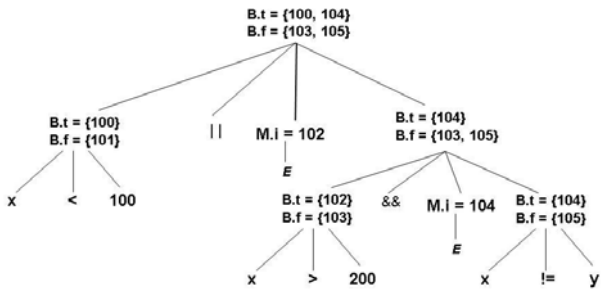
(a) Após o remendo 104 na instrução 102.

```
100: if x < 100 goto _
101: goto _
102: if x > 200 goto 104
103: goto _
104: if x != y goto _
105: goto _
```

(b) Após o remendo 102 na instrução 101.

```
100: if x < 100 goto _
101: goto 102
102: if x > 200 goto 104
103: goto _
104: if x != y goto _
105: goto _
```

Árvore de derivação anotada



Comandos para Controle de Fluxo

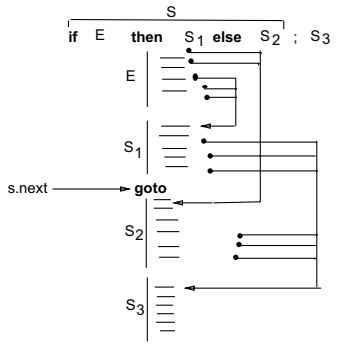
(1) S → if E then S
(2) | if E then S else S
(3) | while E do S
(4) | begin L end
(5) | A
(6) L → L ";" S
(7) | S

- **Atributos:** E.trueList, E.falseList
  L.next, S.next : **Apontadores para uma lista de** gotos **que devem transferir o controle para a quádrupla que será executada após L ou S.**

S denota um comando. L denota uma lista de comandos.
A denota um comando de atribuição.
E denota uma expressão booliana.

· · · Comandos para Controle de Fluxo

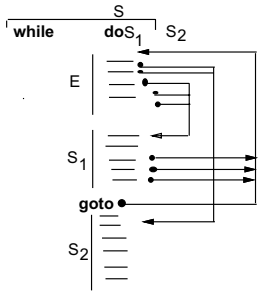
- **IF-THEN-ELSE:** if E then M_1 S_1 N else M_2 S_2; S_3



- N.next **será uma lista consistindo dos números das quádruplas dos comandos** goto **que são pelas rotinas semânticas para** N.

··· Comandos para Controle de Fluxo

- WHILE-DO: S → while M_1 E do M_2 S_1 ; S_2



O "next" de S_1 é a primeira instrução de E.
O "next" de S é a primeira instrução de S_2.
A lista S.next inclui todos os jumps que saem de S_1 e S_2 bem como os jumps gerados por M.

··· Comandos de Controle de Fluxo: Rotinas Semânticas

1) S → if E then M_1 S_1 N else M_2 S_2
{ backpatch(E.trueList, M_1.quad);
backpatch(E.falseList, M_2.quad);
S.next := merge(S_1.next,N.next,S_2.next)}

2) N → $\mathcal{E}$
{ N.next := makelist(nextquad)
emit("goto" _}

3) M → $\mathcal{E}$
{ M.quad := nextquad }

4) S → if E then M S_1
{ backpatch(E.trueList, M.quad);
S.next := merge(E.falseList,S_1.next)}

··· Comandos de Controle de Fluxo: Rotinas Semânticas

5) S → while M_1 E do M_2 S_1 { backpatch(S_1.next, M_1.quad);
backpatch(E.trueList, M_2.quad);
S.next := E.falseList
emit("goto" M_1.quad) }

6) S → begin L end { S.next := L.next }

7) S → A { S.next := makelist( ) }

8) L → L_1 ; M S { backpatch(L_1.next, M.quad);
L.next := S.next }

9) L → S { L.next := S.next }

··· Comandos de Controle de Fluxo

- Considerações sobre as Rotinas Semânticas
  - O comando seguindo L_1 na ordem de execução é o início de S. Portanto a lista L_1.next é backpatch ao início do código de S o qual é dado por M.quad.
  - Note que nenhuma quádrupla nova é gerada nestas rotinas, exceto pelas regras (2) e (5).

    Todos os outros códigos são gerados pelas rotinas semânticas associadas com os comandos de atribuição e expressões.
  - O que o fluxo de controle faz é permitir um apropriado backpatching, de tal forma que as atribuições e expressões boolianas sejam conectadas corretamente.

RÓTULOS e GOTOS

- **GOTOS:** construção mais simples para mudar o fluxo de controle em um programa.

```
S → goto id { k := get-entry(id.nome)
              if k ≠ 0 and TS.definido[k]="D"
              then gen(goto TS.valor[k])
              else begin
                        if k = 0 then instala(id.nome, "N" ,0,k)
                        x := makelist(nextquad)
                        gen(goto −)
                        y := merge(x, TS.head[k])
                        TS.head[k] := y
              end
              S.next := makelist( ) }
```


··· RÓTULOS e GOTOS

- **RÓTULOS:** construção mais simples para mudar o fluxo de controle em um programa.

```
label → id
      { k := get-entry(id.nome)
        if k = 0 then instala(id.nome,"D" ,nextquad,k)
        else if TS.definido[k] = "D"  then erro
             else begin
                  TS.definido[k] := "D"
                  TS.valor[k] := nextquad
                  backpatch(TS.head[k], nextquad)
             end }

S → label : S1
      { S.next := S1.next }
```


Comandos BREAK e CONTINUE

Java não possui *goto*, MAS permite desvios disciplinados, chamados comandos comandos break, que desviam o controle para fora de uma construção envolvente, e comandos continue, que disparam a próxima iteração de um laço envolvente.

O trecho a seguir ilustra os comandos *break* e *continue* simples:

```
1) for ( ; ; readch() ) {
2)     if( peek == ' ' || peek == '\t' ) continue;
3)     else if( peek == '\n' ) line = line + 1;
4)     else break;
5) }
```

O controle desvia de *break* na linha 4 para o próximo comando após o laço *for*.

O controle desvia de *continue* em (2) para o código que avalia *readch*() e então para o *if* em (2).
Se S é a construção loop envolvente, então um *break* é um desvio para a primeira instrução após o código de S.

··· Comandos BREAK e CONTINUE

Gera-se código para o *break*:

1. controlando o contexto do comando envolvente S,
2. gerando um desvio não preenchido para a instrução *break*, e
3. colocando esse desvio não preenchido em $S.next$, onde *next* é conforme discutimos.

Em um front-end de dois passos que constrói árvores de sintaxe, S.next pode ser implementada como um campo no nó para S.

É possível controlar o contexto de S usando a T.S. para mapear um identificador especial *break* ao nó que corresponde o comando envolvente S.

Essa abordagem também trata os comandos *break* rotulados de Java, porque a T.S pode ser usada para mapear o rótulo para o nó da árvore de sintaxe da construção rotulada.

··· Comandos BREAK e CONTINUE

··· Gera-se código para o $break$:

Outra alternativa: em vez de usar a T.S. para acessar o nó para S, coloque um apontador para $S.next$ na T.S.

Agora, quando um comando $break$ é alcançado, gera-se um desvio não preenchido, pesquisa por $next$ na T.S., e incluí o desvio para a lista, onde ele será remendado conforme discutido anteriormente.

O comando $continue$ pode ser tratado de maneira semelhante ao comando $break$.

Diferença principal entre os dois: o destino do desvio gerado é diferente.

6.8 – Comando SWITCH ou CASE

```
switch  expression
        begin
                case value: statement
                case value: statement ···
                case value: statement
                default:
        end
```

```
switch  E
        begin
                case V1: S1
                case V2: S2 ···
                case Vn-1: Sn-1
                default: Sn
        end
```


Tradução Dirigida por Sintaxe para o CASE

```
switch  E
        begin
                case V1: S1
                case V2: S2
                ···
                case Vn-1: Sn-1
                default: Sn
        end
```

Nota: • Usar uma seqüência de gotos condicionais, máximo 10. Valores maiores que 10 ⇒ tabela hash. (com os rótulos dos vários enunciados como entradas).

```
E       código para avaliar E em t
        ( t ≡ temporário)
        goto test
L1 :    código para S1
        goto next ···
Ln-1 :  código para Sn-1
        goto next
Ln :    código para Sn
        goto next
test:   if t = V1 goto L1 ···
        if t = Vn-1 goto Ln-1
        goto Ln
next:
```


Código Gerado

```
case   V1 L1
case   V2 L2
···    ···
case   Vn-1 Ln-1
case   t Ln
label  next
```

Forma mais compacta para implementar seqüência de gotos condicionais: criar tabela de pares, cada par: valor, rótulo

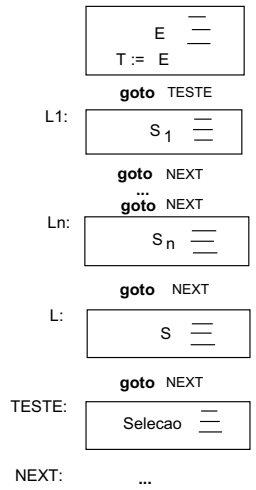
O código é gerado de forma a colocar ao final da tabela o valor da própria expressão associada ao rótulo para o enunciado default.
Um loop é gerado pelo compilador para comparar valor expressão com cada valor na tabela, garantindo que se nenhum casar, o último o fará.

Comando CASE

case E of
of $V_{11}, V_{12}, \cdots : S_1$
of $V_{21}, V_{22}, \cdots : S_2$
$\cdots$ $\cdots$
of $V_{n1}, V_{n2}, \cdots : S_n$
else

CASE – Seleção

TESTE: if T = V_{11} goto L_1
if T = V_{12} goto L_1 $\cdots$
if T = V_{21} goto L_2
if T = V_{21} goto L_2 $\cdots$
if T = V_{nm} goto L_n
goto L



... Comando Case - Rotinas Semânticas

S → case E clauses else S
clauses → clause
| clauses clause
clause → of clist ":" S
clist → const
| clist "," const

- NOTA: A gramática usada a seguir já foi fatorada convenientemente.

... Comando CASE - Rotinas Semânticas

C → of const { c.list := makelist((const.place, NEXTQUAD)) }

caseE → case E { caseE.place := E.place
caseE.test := makelist(NEXTQUAD)
gen(goto −) }

C → C_1 , const { T := makelist((const.place, NEXTQUAD))
c.list := merge(C_1.list, T) }

clause → C : S { clause.list := c.list
T := makelist(NEXTQUAD)
gen(goto −)
clause.next := merge(s.next, T) }
clauses → clause { clauses.list := clause.list
clauses.next := clause.next }

... Comando CASE - Rotinas Semânticas

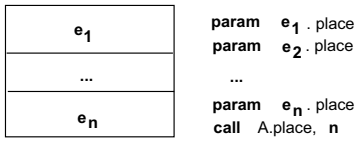
clauses → $clauses_1$ clause
{ clauses.list := merge($clauses_1$.list, clause.list)
clauses.next := merge($clauses_1$.next, clause.next)

M → $\mathcal{E}$ { M.quad := NEXTQUAD }

S → caseE clauses else M S_1
{ U := makelist(NEXTQUAD)
gen(goto −)
S.next := merge(clauses.next, S_1.next, U)
backpatch(caseE.test, NEXTQUAD)
for cada par(V,L) em clauses.list
do gen(if caseE.place = V goto L);
gen(goto M.quad) }

6.9 Código Intermediário para Procedimentos

A(e_1, e_2, $\cdots$, e_n)



S → procid (elist)
elist → elist , E
elist → E

- Atributos: procid.place
  elist.queue: apontador para fila de parâmetros (E.place:) (guarda os valores de E.place para cada expressão E em id(E,E,E,$\cdots$,E))
- Subrotinas Auxiliares:
  makequeue(pts): inicializa elist.queue para conter somente E.place
  insertqueue(queue, pts): adiciona E.place ao final de elist.queue.

$\cdots$ Chamada de Procedimentos - Rotinas Semânticas

S → procid (elist) { n := 0
for cada item p em elist.queue
do n := n + 1;
emit(" param" p)
emit(" call" procid.place, n) }

elist → elist , E { insertqueue($elist_1$.queue, E.place)
elist.queue := $elist_1$.queue }

elist → E {elist.queue := makequeue(E.place) }

- Note bem: Para verificar compatibilidade dos tipos dos parâmetros e argumentos, os parâmetros devem estar na tabela de símbolos no momento da chamada.

Comando FOR - Código Gerado

S → for L := E_1 by E_2 to E_3 do S

	index	:= addr (L)
	*index	:= E_1
	incr	:= E_2
	limit	:= E_3
q_1	:	if *index > limit goto q_2
$\cdots$	S	$\cdots$
	index	:= *index + incr
	goto	q_1
q_2	:	$\cdots$

$\cdots$ Comando FOR - Rotinas Semânticas

F → for L { F.index := L.index ($\star$ $\star$) }

- Gramática Fatorada: F → for L
  T → F := E_1 by E_2 to E_3 do S
  S → T S_1

T → F := E_1 by E_2 to E_3 do { gen(*F.index := E_1.place)
incr := newtemp
limit := newtemp
gen(incr := E_2.place)
gen(limit := E_3.place)
T.quad := nextquad
T.next := makelist(nextquad)
gen(if * F.index > limit goto $-$)
T.index := F.index
T.incr := incr }

... Comando FOR - Rotinas Semânticas

F → for L { F.index := L.index (⋆ ⋆) }

S → T S_1 { backpatch(S_1.next, nextquad)
gen(*T.index := *T.index + T.incr)
gen(goto T.quad)
S.next := T.next }

(⋆ ⋆) L.index ≡ l-value

Usando Fatoração – WHILE-DO

S → while M_1 E do M_2 S

S → C S
C → W E do ⇐
W → while

W → while { W.quad := nextquad }

C → W E do { C.quad := W.quad
backpatch(E.trueList, nextquad);
C.falseList := E.falseList }

S → C S_1 { backpatch(S_1.next, C.quad);
S.next := C.falseList
gen(goto C.quad) }

Usando Fatoração – IF-THEN-ELSE

S → if E then M_1 S_1 N else M_2 S_2

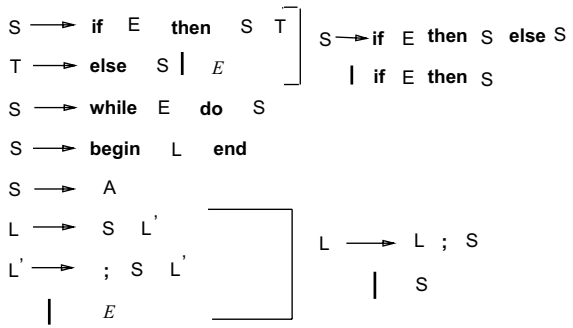
C → if E then
B → S_1 else
S → C B S_2

C → if E then { backpatch(E.true, nextquad)
C.false := E.false }

B → S_1 else { B.next := merge(S_1.next, makelist(nextquad)
gen(goto −);
B.quad := nextquad }

S → C B S_2 { S.next := merge(B.next, S_2.next)
backpatch(C.falseList, B.quad) }

Tradução com Reconhecedor Descendente



S.next, L.next ≡ valores retornados pelos procedimentos S, T, L e L'.

Tradução com Reconhecedor Descendente

```
procedure S returns apontador para lista "next"
case primeiro token de entrada of
     "if": advance
          (E.trueList, E.falseList) := E
          backpatch(E.trueList, nextquad)
          if próximo token de entrada é "then"
          then advance; S1.next := S
          else ERROR
          if próximo token de entrada é "else"
          then advance; gen(goto ??)
               backpatch(E.falseList, nextquad)
               S1.next := merge(S1.next,{nextquad − 1})
               S2.next := S
               return merge(S1.next, S2.next)
          else return merge(S1.next, E.falseList)
```


Tradução com Reconhecedor Descendente

```
procedure S returns apontador para lista "next"
     of "while":
          advance
          quad := nextquad
          (E.trueList, E.falseList) := E
          backpatch(E.trueList, nextquad)
          if próximo token de entrada é "do"
          then advance;
               S.next := S
               backpatch(S.next, quad)
               gen(goto quad)
               return E.falseList
          else ERROR
```


Tradução com Reconhecedor Descendente

```
procedure S returns apontador para lista "next"
     of "begin":
          advance
          L.next := L
          if próximo token de entrada é "end"
          then
               advance;
               return L.next
          else ERROR

     of "id": A
          return makelist( )

     default: ERROR
end S
```


FIM