Geração de Código Intermediário

Um compilador pode ser considerado como um modelo de:

- **Análise**("front-end modules") - traduz um programa fonte em uma representação (linguagem) intermediária

e

- **Síntese**("back-end modules") - a partir do código intermediário gera o código objeto final.

Embora um programa fonte possa ser traduzido diretamente para a linguagem objeto, o uso de uma representação intermediária, independente de máquina, tem as seguintes vantagens:

1) Reaproveitamento de código, facilitando o transporte de um compilador para diversas plataformas de hardware; somente os módulos finais precisam ser refeitos a cada transporte.

2) Um otimizador de código independente de máquina pode ser usado no código intermediário.

3) Pode ser facilmente interpretado

Linguagens Intermediárias (formas internas)

- Notação Pós-Fixa ou Polonesa Reversa
- Árvores Sintáticas
- Código de Três Endereços

→Notação Pós-Fixa ou Polonesa Reversa

Os operandos vêm antes dos operadores. Adequada para representar expressões aritméticas e/ou lógicas.

As expressões pós-fixas podem ser obtidas caminhando-se, em pós-ordem, na árvore sintática correspondente.

Exemplo:

expressão infixa	**expressão pós-fixa**
a + b	ab+
a*(b+c)	abc+*
a/b + c*d	ab/cd*+
a∧b∨c	ab∧c∨

Os operandos unários podem ser manipulados de duas formas:

- Como operador binário, p/ ex: -b ⇒ 0-b ⇒ 0b-
- Introduzindo um novo símbolo para o operador unário
  Ex: -b ⇒ @b, assim, -b * a ⇒ b@a*

OBS: Na notação pós-fixa os operadores aparecem na ordem em que serão executados. Ótima notação para máquinas com estrutura de pilha (*stack machine*), como, calculadora de bolso.

As expressões pós-fixas podem ser avaliadas com uma simples varredura, da esquerda para a direita, utilizando-se uma pilha para os operandos.

Exemplo: Avaliar a expressão pós-fixa w = ab@cd*++

passo	**w_i**	**expressão posfixa**	**pilha**	**operações**
0		ab@cd*++		
1	a	b@cd*++	a	
2	b	@cd*++	ab	
3	@	cd*++	aT_1	$T_1 := -b$
4	c	d*++	aT_1c	
5	d	*++	aT_1cd	
6	*	++	aT_1T_2	$T_2 := c*d$
7	+	+	aT_3	$T_3 := T_1 + T_2$
8	+		R	$R := a + T_3$

$R := a+T_3 := a+T_1+T_2 := a+ T_1+ c*d := a+(-b)+c*d$

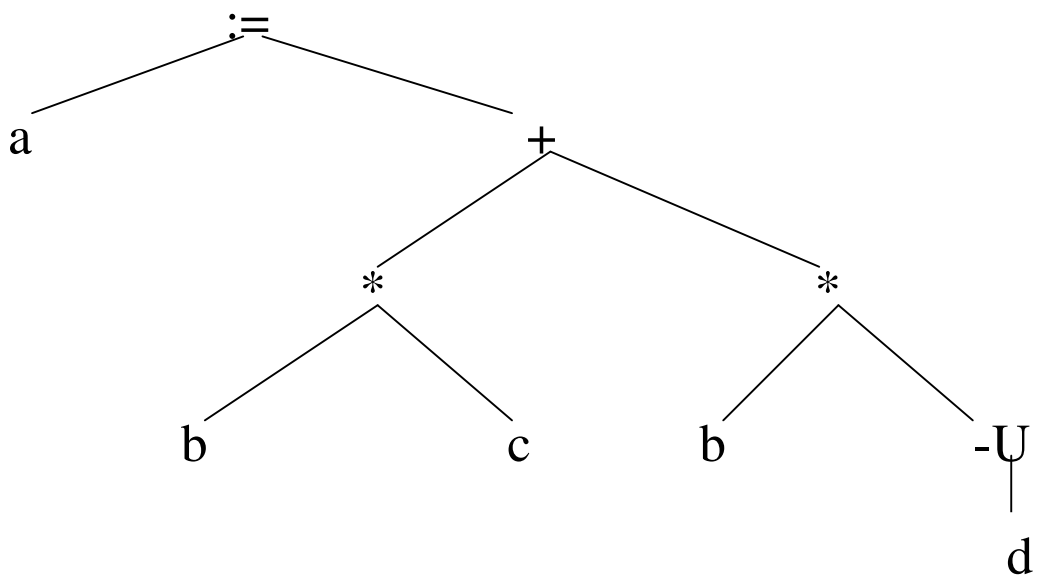
Nota: A notação pós-fixa é inadequada para o processo de otimização de código.

→Árvores Sintáticas

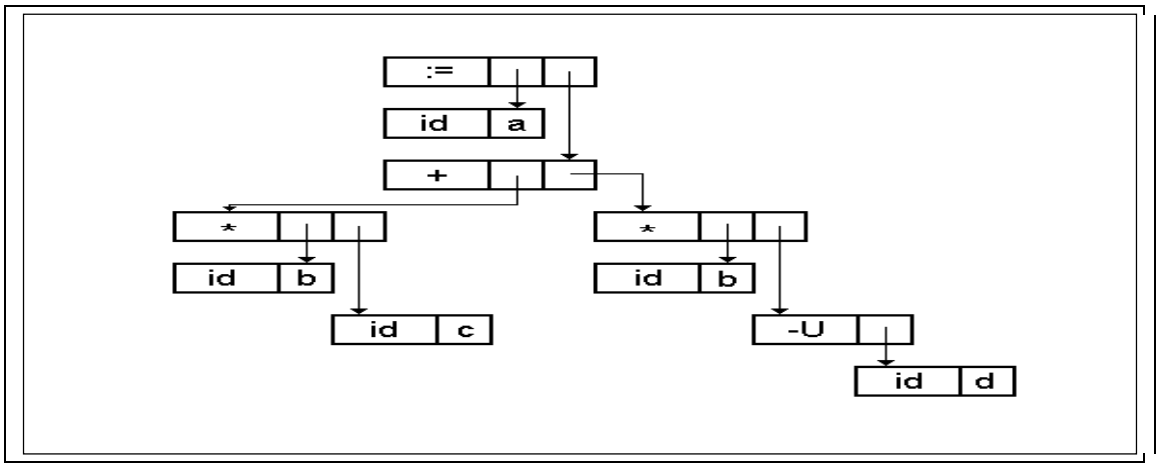
Uma árvore sintática (ou de sintaxe) mostra a estrutura hierárquica de um programa fonte. Por exemplo, a atribuição:

a := b * c + b * -d

seria representada graficamente por:



que poderia ter a seguinte implementação:



→Código de Três Endereços

O código de três endereços é uma seqüência de comandos da forma:

x := y op z

onde:

x, y e z - variáveis ou constantes definidas pelo programador ou variáveis temporárias geradas pelo compilador;

op - define um operador (aritmético, relacional ou lógico)

:= - define o operador de atribuição.

Para o comando de atribuição:

a := b * c + b * -d

poderíamos ter a seguinte seqüência de comandos:

comando	operando-1	operador	operando-2	operando-3
1	t1	*	b	c
2	t2	-U	d	
3	t3	*	b	t2
4	t4	+	t1	t3
5	a	:=	t4	

OBS: O código de três endereços é uma representação linearizada de uma árvore sintática

O código de três endereços é semelhante às instruções em linguagem de montagem. Nesse sentido, podemos ter comandos com rótulos simbólicos e comandos para controle de fluxo.

Comandos bastante comuns para a geração de código intermediário são os seguintes:

- **Atribuições:**

  x := y op z - op é uma operação binária (aritmética, lógica ou relacional)

  x := op y - op é uma operação unária (menos, negação lógica, deslocamento, conversão de tipos)

  x := y - atribui o valor de y a x

- **Desvio Incondicional:**

  goto L - desvia para o comando com rótulo L

- **Desvio Condicional:**

  if x op_rel y then goto L – aplica um operador relacional (<, =, ≠, etc.) a x e y. Se a relação for verdadeira executa o comando com rótulo L, caso contrário executa o comando após o bloco do **if**.

- **Ativação de Sub-Programa:**

  param x e call p, n – para ativação de procedimentos. Na ativação de sub(x_1, x_2, ..., x_n) é gerada a seguinte seqüência de três endereços:

  param x_1
  param x_2
  .
  .
  .
  param x_n
  call p, n

- **Atribuições Indexadas**

  x := y[i] – atribui a x o valor da localização i unidades de memória após a localização de y

  x[i] := y – atribui à localização i unidades de memória após a localização de x o valor y

- **Atribuição de Endereços e Apontadores:**

  x := &y – atribui a x o endereço (localização) de y

  x := *y – atribui a x o conteúdo de memória cuja localização é dada pelo valor de y

  ***x := y –** atribui o valor de y ao objeto apontado por x

A escolha dos operadores disponíveis é de fundamental importância no projeto de uma linguagem intermediária:

- O conjunto de operadores deve ser claro e rico o suficiente para implementar todas as operações da linguagem fonte.

- Um conjunto pequeno de operadores simplifica o transporte do compilador para novas plataformas. Entretanto, isso acarreta:

  - a geração de longas seqüências de comandos para algumas operações da linguagem fonte.

  - um trabalho maior do otimizador e do gerador de código intermediário para produzir um código final de qualidade.

→Implementações de Códigos de Três Endereços

- Código de três endereços é uma forma abstrata de código intermediário.

- Em um compilador real, esses comandos podem ser implementados como registros com campos para o operador e os operandos. Os dois tipos mais comuns desses registros são as *quádruplas* e as *triplas*.

Quádruplas

São estruturas de registros com quatro campos, contendo:

<operador, argumento_1, argumento_2, resultado>

Na forma de quádruplas, o código de três endereços
x := y + z é representado como: <+, y, z, x>

Exemplo:

Para o comando de atribuição:

a := b * -c + b * -c

temos a seguinte seqüência de quádruplas:

quádrupla	operador	argumento_1	argumento_2	resultado
(0)	-unário	c		t1
(1)	*	b	t1	t2
(2)	-unário	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

OBS:

- Os comandos com operadores unários não usam o *argumento_2*.

- Os operadores como *param* não usam nem *argum2* nem *resultado*

- Os desvios condicionais e incondicionais colocam o rótulo alvo em *resultado*.

- Os conteúdos dos campos *arg1*, *arg2* e *resultado* são normalmente apontadores para as entradas, na tabela de símbolos, dos nomes representados por esses campos.

Exemplo: Gerar quádruplas para o seguinte programa Pascal

```
program estat;
var
    soma, somaq, valor, media, variancia : integer;
begin
    soma := 0;    somaq := 0;
    for i := 1 to 100 do
    begin
        read(valor);
        soma := soma + valor;
        somaq := somaq + valor*valor;
    end;
    media := soma div 100;
    variancia := somaq div 100 – media*media;
    write(media, variancia);
end.
```

quádr.	op	arg1	arg2	result.	comentário
0	:=	0		soma	{soma := 0}
1	:=	0		somaq	{somaq := 0}
2	:=	1		i	{for i:= 1 to 100}
3	JGT	i	100	(14)	
4	param	valor			{read(valor)}
5	call	read			
6	+	soma	valor	t1	{soma := soma + valor}
7	:=	t1		soma	
8	*	valor	valor	t2	{somaq := somaq +
9	+	somaq	t2	t3	valor*valor}
10	:=	t3		somaq	

Adaptado de notas de aulas do prof. Giuseppe Mongiovi do DI/UFPB.

quádr.	op	arg1	arg2	result.	comentário
11	+	i	1	t4	{fim do loop for}
12	:=	t4		i	
13	JMP			(3)	
14	div	soma	100	t5	{media:=soma div 100}
15	:=	t5		media	
16	div	somaq	100	t6	{variancia :=
17	*	media	media	t7	somaq div 100
18	-	t6	t7	t8	- media*media}
19	:=	t8		variancia	
20	param	media			{write(media,
21	param	variancia			variancia)}
22	call	write			
23	halt				

Triplas

São estruturas de registros com três campos, contendo:

<*op*, *arg1*, *arg2*>

Os campos *arg1* e *arg2*, na qualidade de argumentos de *op*, ou são apontadores para a tabela de símbolos (para nomes definidos pelo programador ou constantes) ou apontadores para estruturas de triplas (para valores temporários).

Adaptado de notas de aulas do prof. Giuseppe Mongiovi do DI/UFPB.

Exemplo:

Para o comando de atribuição:

a := b * -c + b * -c

temos a seguinte seqüência de triplas:

tripla	operador	argumento_1	argumento_2
(0)	- unário	c	
(1)	*	b	(0)
(2)	- unário	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

Rotinas de Semântica (Geração das Formas Internas)

O analisador sintático informa apenas se uma sentença pertence ou não a uma linguagem. Entretanto, durante o processo de análise outras atividades terão que ser realizadas, por exemplo:

- verificar compatibilidade de tipos;
- gerar código intermediário (ou de máquina);
- recuperar erros sintáticos.

Solução:

Associar a cada símbolo da gramática um conjunto de atributos e, também, associar a cada produção um conjunto de *regras semânticas* para computar os valores dos atributos associados aos símbolos que figuram nessa produção (***definição dirigida pela sintaxe***).

OBS: Para uma mesma produção, o conteúdo das regras semânticas varia conforme a atividade que se deseja realizar. Por exemplo, gerar expressões pós-fixas, quádruplas, etc.

Exemplo: A seguinte ***definição dirigida pela sintaxe*** traduz expressões da notação infixa para a pós-fixa.

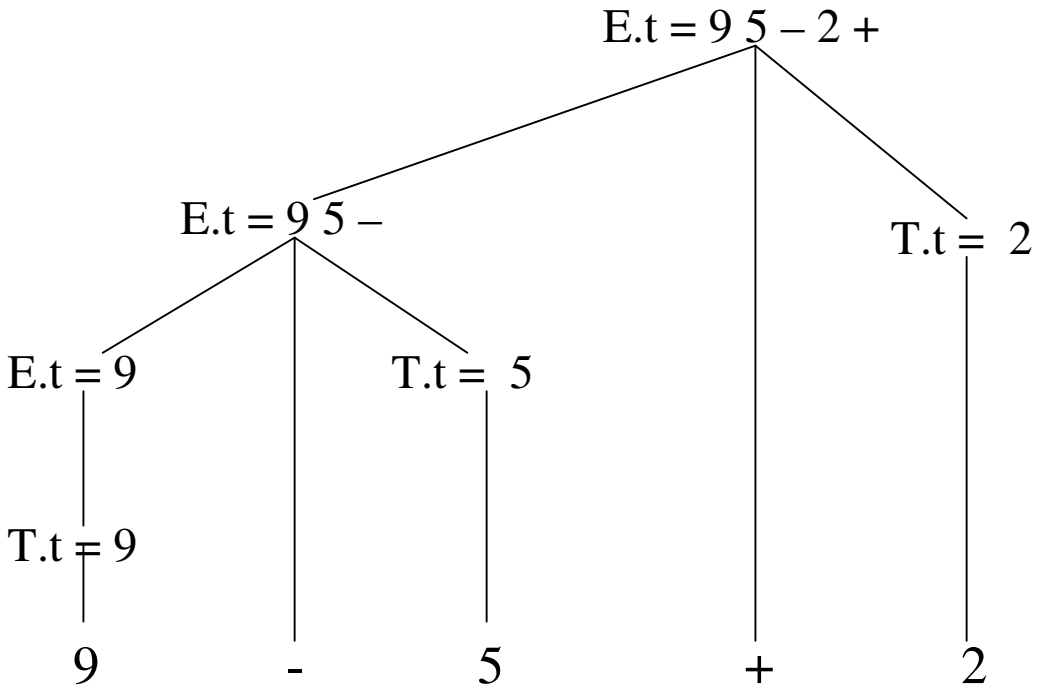
Produção	**Regra Semântica**
$E \rightarrow E + T$	E.t := E.t || T.t || '+'
$E \rightarrow E - T$	E.t := E.t || T.t || '-'
$E \rightarrow T$	E.t := T.t
$T \rightarrow 0$	T.t := '0'
$T \rightarrow 1$	T.t := '1'
. . .	. . .
$T \rightarrow 9$	T.t := '9'

X.t – significa associar o atributo t ao símbolo X

|| - símbolo de concatenação

Adaptado de notas de aulas do prof. Giuseppe Mongiovi do DI/UFPB.

A árvore gramatical com os valores dos atributos para a sentença 9 – 5 + 2 é a seguinte:



Adaptado de notas de aulas do prof. Giuseppe Mongiovi do DI/UFPB.

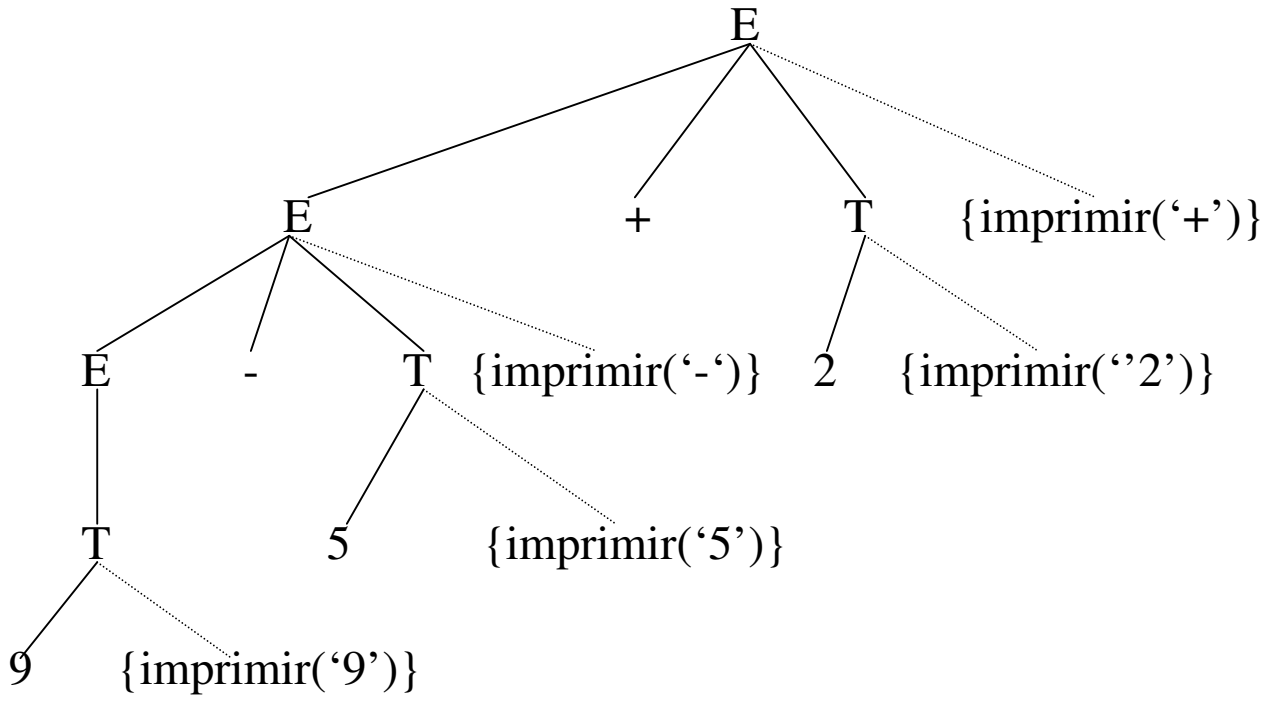
→Esquema de tradução

Um esquema de tradução é como uma *definição dirigida pela sintaxe*, exceto que a ordem de avaliação das regras semânticas é explicitamente mostrada. Por exemplo, um esquema de tradução para o exemplo anterior seria:

Produção	**Regra Semântica**
E → E + T	{imprimir '+'}
E → E - T	{imprimir '-'}
E → T	
T → 0	{imprimir '0'}
T → 1	{imprimir '1'}
. . .	. . .
T → 9	{imprimir '9'}

As regras semânticas, no caso {imprimir '*'}, são vistas como símbolos acrescentados às produções da gramática. São acionadas sempre que as produções forem usadas.

A árvore gramatical com as regras semânticas para traduzir 9 – 5 + 2 em 9 5 – 2 + é a seguinte:



Adaptado de notas de aulas do prof. Giuseppe Mongiovi do DI/UFPB.

OBS: Na transformação de uma gramática com regras semânticas, por exemplo para eliminar a recursividade à esquerda, essas regras são tratadas como símbolos quaisquer.

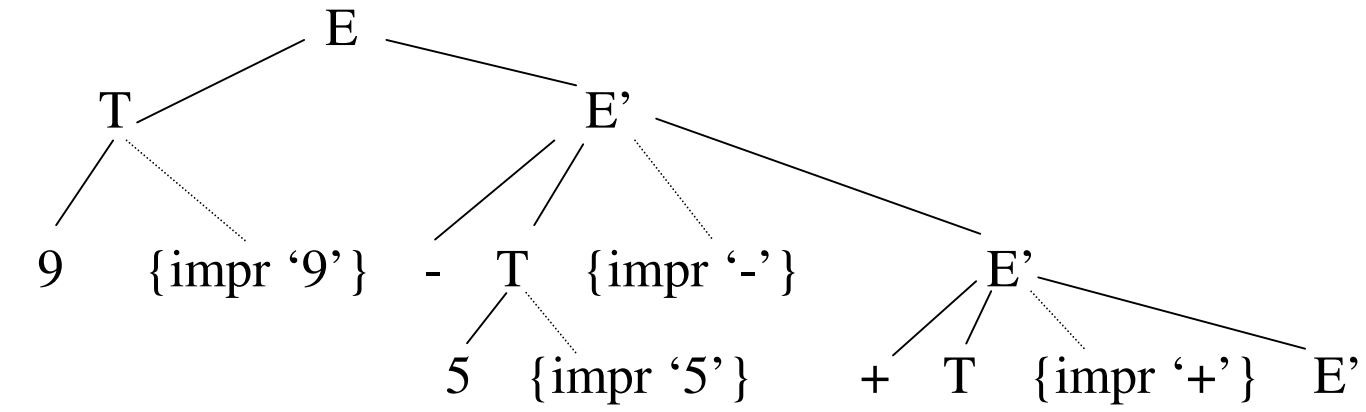
Exemplo: A gramática do esquema de tradução do exemplo anterior é recursiva à esquerda, portanto, não pode ser usada diretamente por um *Parsing* Preditivo ou Analisador Sintático Descendente Preditor. Nesse caso, elimina-se a recursividade à esquerda, lembrando que as regras semânticas são "símbolos" da gramática.

Eliminando a recursividade à esquerda da gramática com semântica do exemplo anterior temos:

$E \rightarrow TE'$

$E' \rightarrow +T\{\text{imprimir '+'}\}E' \mid -T\{\text{imprimir '-'}\}E' \mid \varepsilon$

$T \rightarrow 0\{\text{imprimir '0'}\} \mid 1\{\text{imprimir '1'}\} \mid \ldots \mid 9\{\text{imprimir '9'}\}$



Tradução de 9-5+2
em
95-2+

Adaptado de notas de aulas do prof. Giuseppe Mongiovi do DI/UFPB.

Geração de Código Intermediário para Expressões Aritméticas

Vamos começar com expressão aritmética e comando de atribuição, considerando a gramática abaixo:

$$
\begin{aligned}
A &\rightarrow id := E \\
E &\rightarrow T\ E' \\
E' &\rightarrow + T\ E' \mid \varepsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow * F\ T' \mid \varepsilon \\
F &\rightarrow id \mid nro\_int \mid nro\_real \mid (\ E\ )
\end{aligned}
$$

Uma forma simples de geração de código intermediário consiste no uso de uma Pilha de controle de Operandos (PcO), que vai conter os operandos usados nas operações realizadas em uma expressão aritmética.

Quando o analisador sintático aplicar as regras

$F \rightarrow id$ ou $F \rightarrow nro\_int$ ou $F \rightarrow nro\_real$

vamos simplesmente empilhar na PcO o identificador, número inteiro ou número real correspondente (ou indicador de onde se encontra esse operando).

Quando o analisador sintático aplicar as regras

$E' \rightarrow + T\ E'$ ou $T' \rightarrow * F\ T'$

sabemos que ele reconheceu uma operação de adição ou multiplicação e que, não havendo problemas de tipos, pode gerar código intermediário para a realização de tais operações. Para tal, basta utilizar os operandos que estiverem no topo e subtopo da PcO, deixando o resultado em uma variável temporária e empilhando essa variável na PcO.

Vamos então modificar as regras para introduzir ações de geração de código necessárias (semelhantes às ações semânticas).

$A \rightarrow id := E$	$A \rightarrow id\ \langle G1 \rangle := E\ \langle G2 \rangle$
$E' \rightarrow + T\ E'$	$E' \rightarrow + T\ E'\ \langle G3 \rangle$
$T' \rightarrow * F\ T'$	$T' \rightarrow * F\ T'\ \langle G4 \rangle$
$F \rightarrow id$	$F \rightarrow id\ \langle G5 \rangle$
$F \rightarrow nro\_int$	$F \rightarrow nro\_int\ \langle G6 \rangle$
$F \rightarrow nro\_real$	$F \rightarrow nro\_real\ \langle G7 \rangle$

As ações de geração de código são as seguintes:

<G1> /* lembrar do lado esquerdo de uma atribuição e
iniciar o contador de temporárias em 1 */

```
lado_esquerdo <- símbolo;
reset_temp();
```

<G2> /* gerar código para atribuição */

```
oper_1 <- pop( PcO );
emitir(':=', oper_1, “” ,lado_esquerdo);
```

<G3> /* aplicou regra + TE', gerar código para adição */

```
oper_2 = pop( PcO );
oper_1 = pop( PcO );
result = nova_temp();
emitir('+', oper_1, oper_2, result);
```

<G4> /* aplicou regra *FT', gerar código para multiplicação */

```
oper_2 = pop( PcO );
oper_1 = pop( PcO );
result = nova_temp();
emitir('*', oper_1, oper_2, result);
```

<G5> /* viu identificador como operando; se for global, colocar nome dele na PcO; se for local, colocar referência relativa à pilha do programa (Exemplo: [base_da_pilha+endereço_relativo]) */

pesquisa_tab_símbolo(símbolo);
se símbolo.escopo = global
    push(PcO, símbolo);
senão
    push(PcO, [base_da_pilha+endereço_relativo]);

<G6>

<G7> /* viu número inteiro ou real como operando; colocar símbolo na PcO */

push(PcO, símbolo);

As rotinas auxiliares usadas são as seguintes:

```
nova_temp() /* gera nome de nova variável
               temporária */
{
   var = concat( "t", prox_temp );
   prox_temp = prox_temp + 1;
   retorna( var );
}
```

```
reset_temp()
{
   prox_temp = 1;
}
```

Vejamos um exemplo com o comando x := a + b * c
Observando a pilha de análise.

PILHA DE ANÁLISE	ENTRADA	PcO	OBS.
$ A	**x := a+b*c$**		
$ <G2>E := <G1>id			**x OK**
$ <G2>E := <G1>	**:= a+b*c$**		**lado_esquer = x; prox_temp = 1;**
$ <G2>E :=			**:= OK**
$ <G2>E	**a + b * c $**		
$ <G2>E' T			
$ <G2>E' T' F			
$ <G2>E' T' <G5>id			**a OK**
$ <G2>E' T' <G5>	**+ b * c $**	**a**	**push(PcO, a)**
$ <G2>E' T'			
$ <G2>E'			
$ <G2><G3>E' T +			**+ OK**
$ <G2><G3>E' T	**b * c $**		
$ <G2><G3>E' T' F			
$ <G2><G3>E' T' <G5>id			**b OK**
$ <G2><G3>E' T' <G5>	*** c $**	**a b**	**push(PcO, b)**
$ <G2><G3> E' T'			
$ <G2><G3>E' <G4>T' F *			*** OK**
$ <G2> <G3> E' <G4> T' F	**c $**		
$ <G2><G3>E' <G4>T' <G5>id			**c OK**
$ <G2><G3>E' <G4>T' <G5>	**$**	**a b c**	**push(PcO, c)**
$ <G2><G3>E' <G4>T'			
$ <G2><G3>E' <G4>		**a t1**	**t1 := b * c**
$ <G2><G3>E'			
$ <G2><G3>		**t2**	**t2 := a + t1**
$ <G2>			**x := t2**
$	**$**		**$ OK**

Adaptado de notas de aulas do prof. Giuseppe Mongiovi do DI/UFPB.

Geração de Código Intermediário para Comandos de Controle de Fluxo

Comandos de controle de fluxo podem ser basicamente resumidos aos seguintes:

<cmdo> → if <cond> then <cmdo> <pelse>
| while <cond> do <cmdo>

<pelse> → else <cmdo>
| ε

Exemplo_1: esqueleto para o comando: **if** E **then** S1 **else** S2

Código para avaliar E
JFALSE Início_else {*desvia se E for falso*}
Código para S1
JMP Fim_if {*desvia para o fim do* IF}
Início_else:
Código para S2
Fim_if: . . .

Exemplo_2: esqueleto para o comando: **while** E **do** S

Início_while:

Código para avaliar E

JFALSE Fim_while

Código para S

JMP Início_while {*desvio incondicional*}

Fim_while: . .

Para tratá-los, vamos usar uma pilha de controle de rótulos (*labels*) e uma tabela de rótulos, necessárias para resolver o endereço de um rótulo referenciado antes de ser definido.

Considera-se que ao final da geração do código intermediário, a tabela de rótulos contenha somente entradas resolvidas (com rótulo e endereço) para que se faça a respectiva substituição no código intermediário.

Para o comando if, as ações de geração de código introduzidas seriam as seguintes:

<cmdo> → if <cond> then <cmdo> <AG-5> <pelse>
<AG-6>
<cond> → <AG-1><expr-arit><AG-2><op-rel><AG-3>
<expr-arit> <AG-4>
<op-rel>→ > | >= | < | <= | == | !=
<pelse> → else <cmdo>

<AG-1>[Re]inicializa o contador de temporárias
reset_temp();

<AG-2> Salva valor do lado esquerdo da expressão relacional
lado_esquerdo = pop(PcO);

<AG-3> Salva valor do operador relacional
oper_relacional = código; // do símbolo lido

<AG-4> Gera código para teste da condição com respectivo desvio

```
result = nova_temp();
emitir(oper_relacional, lado_esquerdo, pop( PcO ), result );
label = novo_label();
push( PcL, label );
emitir( "JFALSE", result, "" , label );
```

<AG-5>Resolve endereço do *label* anterior e gera desvio da parte else

```
label-aux = pop( PcL );
label = novo_label();
push (PcL, label );
emitir( "JMP", "", "", label );
insere( Tab_label, label-aux, quádrupla_atual );
```

<AG-6>Resolve endereço do *label* de fim de if

```
label = pop( PcL );
insere( Tab_label, label, quádrupla_atual );
```

A rotina novo_label() retorna sempre um rótulo único a cada chamada no formato L.xxx, onde xxx é um número de 3 dígitos, iniciando em 001.

A rotina insere(Tab_label, label, quádrupla_atual) insere, com o respectivo endereço, um rótulo na Tabela de rótulos. Ao final da geração do código intermediário, o mesmo deve ser varrido em busca de instruções de desvio para que os rótulos simbólicos (L.xxx) sejam substituídos pelos respectivos endereços de quádruplas.

Como um exemplo, mostramos o código intermediário gerado para o comando:

```
If ( a + 2 > k/5-3 ) then
        . . .  // comando associado ao then
else
        . . .  // comando associado ao else
```

Qdr.	OP	Oper_1	Oper_2	Result	Observação
					AG-1: [re]inicializa contador de temporárias
10	+	a	2	T.001	Avalia lado esquerdo da expressão relacional
					AG-2: salva valor do lado esquerdo da expressão relacional
					AG-3: salva valor do operador relacional
11 12	/ -	K T.002	5 3	T.002 T.003	Avalia lado direito da expressão relacional
13 14	> JFALSE	T.001 T.004	T.003	T.004 L.001	AG-4: gera código para teste da condição com respectivo desvio
15 .					Código para comando then
30	JMP			L.002	AG-5: resolve endereço de label anterior e gera desvio do comando else
31 .					Código para o comando else
40					AG-6: resolve endereço do label do fim do if

Tabela de Labels

Label	**Endereço**
L.001	31
L.002	40

Adaptado de notas de aulas do prof. Giuseppe Mongiovi do DI/UFPB.

Para o comando while, as ações de geração introduzidas seriam as mesmas usadas em <cond> do comando if, acrescidas da seguintes:

<cmdo> ::= while < AG-7> <cond> do <cmdo> <AG-8>
endwhile

<AG-7> Gera *label* de início do ciclo

```
label = novo_label();
insere( Tab_label, label, quádrupla_atual );
push( PcL, label );
```

<AG-8> Gera desvio para início do ciclo e resolve endereço para *label* de fim de while

```
label-fim = pop( PcL );
label-início = pop( PcL );
emitir("JMP", "", "",label-início);
insere( Tab_label, label-fim, quádrupla_atual );
```

Como um exemplo, mostramos o código intermediário gerado para o seguinte comando:

```
while (a + 2 > k/5-3) do
      . . .  // comando associado ao ciclo
endwhile
```

Qdr.	OP	Oper_1	Oper_2	Result	Observação
					AG-7: gera label de início do ciclo
					AG-1: [re]inicializa contador de temporárias
100	+	a	2	T.001	Avalia lado esquerdo da expressão relacional
					AG-2: salva valor do lado esquerdo da expressão relacional
					AG-3: salva valor do operador relacional
101 102	/ -	K T.002	5 3	T.002 T.003	Avalia lado direito da expressão relacional
103 104	> JFALSE	T.001 T.004	T.003	T.004 L.001	AG-4: gera código para teste da condição com respectivo desvio
105 .					Código para comando while
300	JMP			L.002	AG-8: gera código de desvio para o início do ciclo e resolve endereço para label do fim do ciclo
301					

Tabela de Labels

Label	**Endereço**
L.002	100
L.001	301

- Referências

Notas de aulas do prof. Giuseppe Mongiovi do DI/UFPB, 2002.

Appel, A. W. **Modern Compiler Implementation in C.** Cambridge University Press, 1998. (Capítulo 5, seções 5.1, 5.3 e 5.4; e Capítulo 7).