Capítulo 5. Tradução Dirigida pela Sintaxe

5.1. - Formas de Representação Intermediária

Como observado no Capítulo 1, normalmente a tradução do programa fonte no programa objeto é feita em dois passos, e uma representação intermediária do programa fonte deve ser construída. Tipicamente, essa representação intermediária assume uma de duas formas: uma árvore (árvore sintática) ou uma seqüência de comandos em uma linguagem intermediária, cuja forma mais comum é a de quádruplas, ou seja, de comandos compostos de um operador e de três operandos, que quase sempre são endereços. Por exemplo, o comando de atribuição

```
x := (a+b)*c
```

pode ser representado por uma árvore

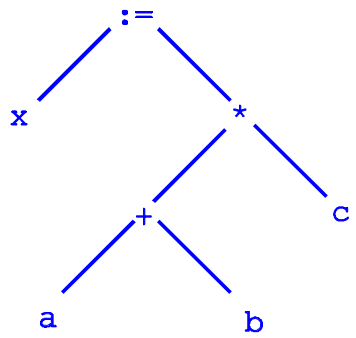


Fig. 1. - Representação de um comando em árvore

ou por uma seqüência de instruções

+	a	b	t1
*	t1	c	t2
:=	x	t1	-

Em ambos os casos, é simples a transformação para código de máquina convencional. Vamos discutir neste capítulo como podem ser construídas representações intermediárias dos dois tipos, para as construções mais comuns das linguagens de programação.

5.2. - Árvores Sintáticas

A primeira árvore que se poderia usar como base para uma representação intermediária é a árvore de derivação correspondente ao programa fonte, de acordo com a gramática da linguagem fonte usada na construção do analisador sintático. Entretanto, essa árvore de derivação em geral não é conveniente, uma vez que inclui muitos detalhes desnecessários, que foram incluídos na sintaxe da linguagem fonte para resolver problemas de ambigüidade, precedência, pontuação, clareza, legibilidade, etc. Por essa razão, em alguns casos é conveniente a definição de uma segunda sintaxe, chamada sintaxe abstrata, que ignora todos os aspectos não essenciais da sintaxe original, conhecida como sintaxe concreta.

Por exemplo, considere as duas gramáticas

1. L → L ; S	1. S → S ; S
2. L → S	
3. S → V := E	3. S → E := E
4. E → E + T	4. E → E + E
5. E → T	
6. T → T * F	6. E → E * E
7. T → F	
8. F → (E)	
9. F → V	
10. V → id	10. E → id
(sintaxe concreta)	(sintaxe abstrata)

A correspondência entre as regras das duas gramáticas é clara, e está indicada pela forma de numeração das regras da segunda gramática: atribuímos a cada regra da segunda gramática o mesmo número da regra correspondente na primeira. A primeira gramática poderia ser usada para a construção de um analisador sintático ascendente de um compilador; entretanto, a segunda gramática tem uma linguagem diferente, tem ambigüidade em várias construções, e não serviria como base para um analisador sintático. Entretanto, ela apresenta todas as construções essenciais da linguagem e pode servir de base a uma gramática de atributos. Na prática, podemos usar o analisador da sintaxe concreta para simular um analisador da sintaxe abstrata. Quando o analisador da sintaxe concreta sinaliza o uso de uma regra, a ação associada à regra correspondente da sintaxe abstrata (se esta ação existir) é executada.

Considere o trecho de programa

```
id:=id+id ; id:=id*(id+id)
```

A Fig 2 apresenta diversas árvores associadas a esta cadeia, construídas de acordo com as duas gramáticas.

- Em (a) temos a árvore de derivação segundo a sintaxe concreta, com nós rotulados como habitualmente por símbolos, terminais e nãoterminais.
- Em (b), a derivação está indicada através dos números das regras, ainda para a sintaxe concreta. Note que os terminais não estão indicados.
- Em (c) temos, em formato semelhante ao de (b), a forma de derivação correspondente à sintaxe abstrata. Esta árvore pode ser chamada de árvore de sintaxe abstrata, ou *abstract syntax tree*.
- Em (d), a mesma árvore (c) é repetida, substituindo os números de regras usados em (b) por símbolos terminais sugestivos das regras empregadas.

Podemos notar uma significativa diferença de tamanho entre a árvore (a) e as árvores (c) ou (d), pela supressão dos nós correspondentes aos símbolos terminais e dos nós das regras que deixaram de ser introduzidas na sintaxe abstrata, por dispensarem tratamento semântico.

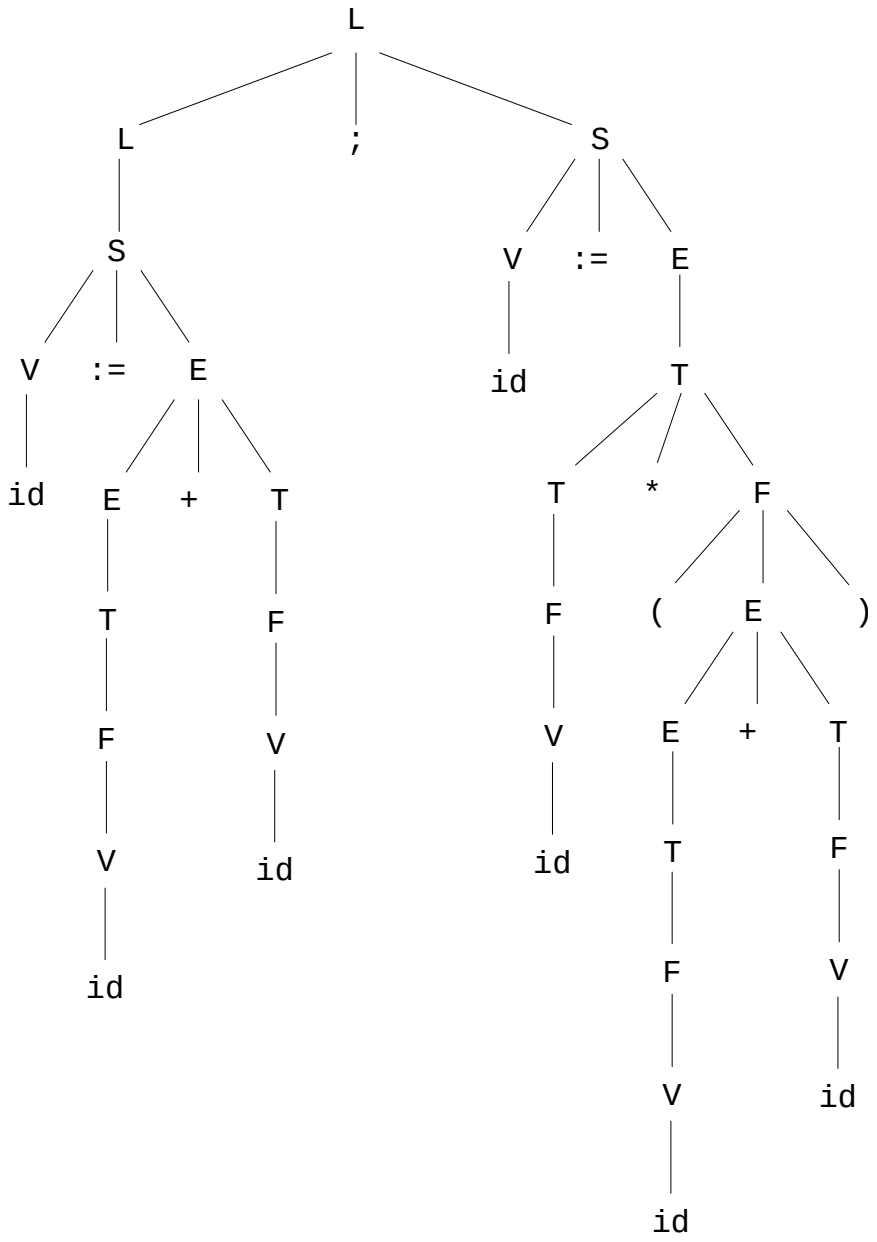


Fig 2(a) Árvore de derivação

Note que, quando se estabelece a correspondência entre as regras da sintaxe abstrata e da sintaxe concreta, só podem ficar sem correspondência regras da sintaxe concreta que tenham apenas um nãoterminal do lado direito, sob pena de destruir a estrutura da árvore de sintaxe abstrata. Para evitar problemas na montagem da árvore, as regras excluídas devem ser regras "simples", da forma A $\rightarrow$ B, em que um não terminal é substituído por outro, ou, num caso um pouco mais geral, A $\rightarrow \alpha$, caso em que α pode conter vários terminais, mas apenas um nãoterminal.

Exemplos de regras que podem ser excluídas são `L`$\rightarrow$`S` e `F`$\rightarrow$`(E)`. Uma regra como $L^0 \rightarrow L^1$;S, mesmo que não corresponda a nenhuma ação, precisa ser incluída, porque, retirado o nó L^0 , não há lugar para "pendurar" as sub-árvores correspondentes a L^1 e S.

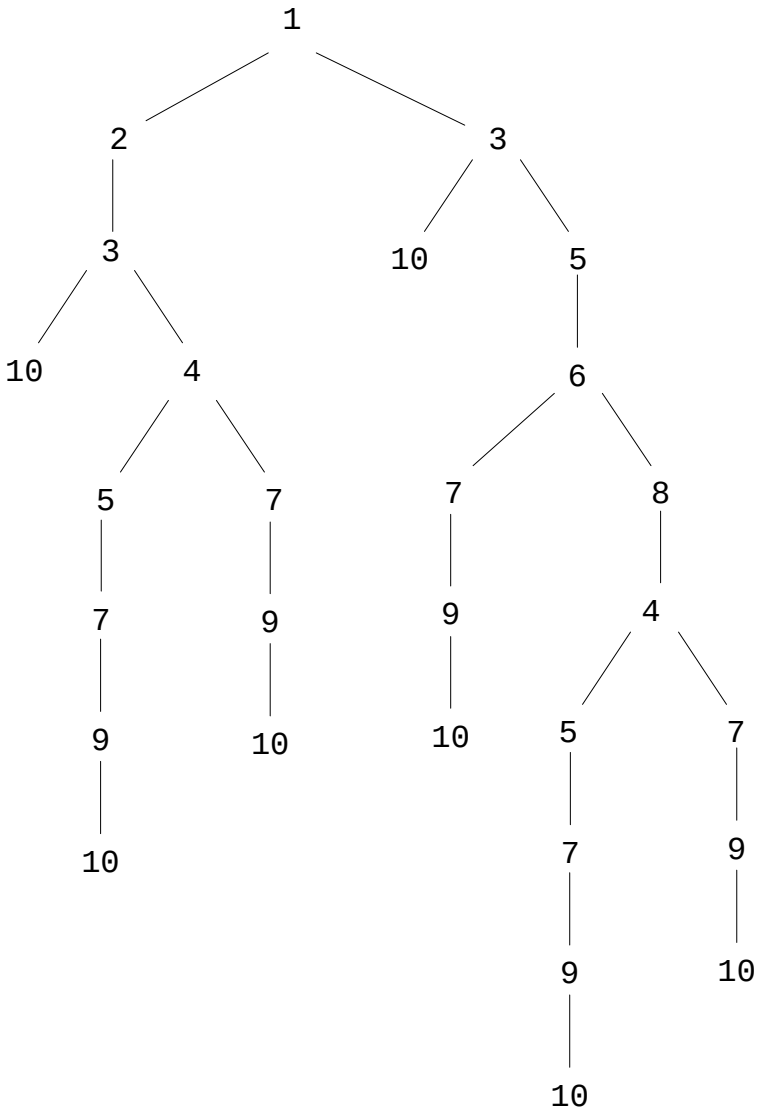


Fig 2(b) Árvore de derivação com os números das regras

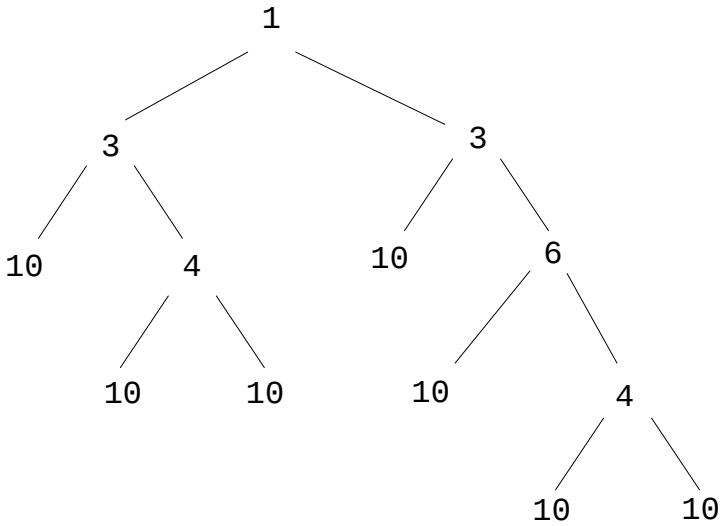


Fig 2(c) Árvore de sintaxe abstrata

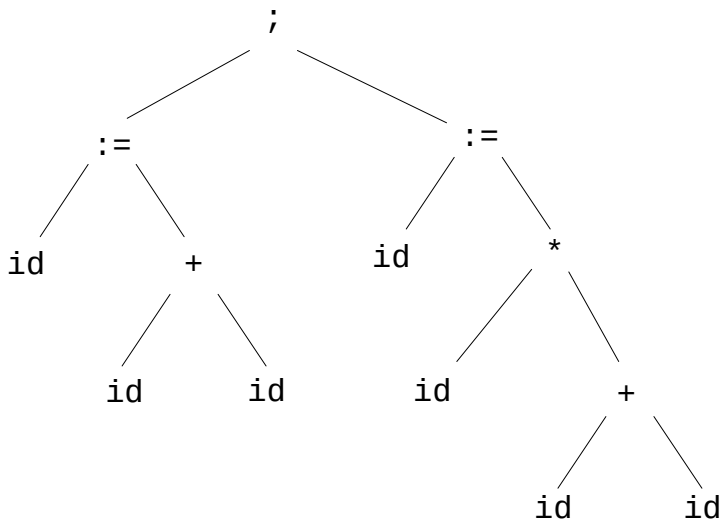


Fig 2(d) Árvore sintática

Não há inconveniente em fazer corresponder uma regra da sintaxe abstrata a duas ou mais regras da sintaxe concreta. Isso acontece quando há regras semanticamente equivalentes na sintaxe concreta, que oferecem sintaxes alternativas para uma construção.

A construção de árvores como as representadas na Fig. 2 pode ser feita usando as técnicas descritas na seção 4.1. Campos para os valores dos atributos a serem calculados devem ser incluídos nos nós, onde for apropriado.

Freqüentemente, a sintaxe abstrata não é especificada de forma explícita, de forma que as árvores construídas não podem são apresentadas como árvores de derivação de uma gramática conhecida. Para essas árvores, preferimos o nome *árvore sintática*. Uma árvore sintática, como por exemplo a árvore (d) da Fig. 2 pode ser entendida como um resumo, uma representação mais compacta da árvore de derivação correspondente à sintaxe concreta. Se forem acrescentados atributos (valores associados aos nós da árvore) podemos falar de uma árvore sintática "decorada", ou seja, "enfeitada" com os atributos.

Quando se deseja utilizar uma representação intermediária em forma de árvore sintática, ela pode ser construída durante o processo de análise sintática, usando-se um analisador sintático ascendente. A avaliação dos atributos associados aos nós da árvore sintática pode ser feita pela aplicação das técnicas vistas no Cap. 4.

5.3 - Quádruplas (Código de 3 endereços)

Antes de descrever a geração de código de 3 endereços (quádruplas), alguns pontos devem ser discutidos e algumas convenções estabelecidas.

Primeiro, devemos considerar a necessidade do uso de variáveis auxiliares, além daquelas declaradas pelo programador. Quando uma expressão (ou sub-expressão) é avaliada, um novo valor é criado, e, a não ser em casos particulares, esse valor não pode ser diretamente armazenado em nenhuma das variáveis declaradas pelo programador. Há necessidade, assim, de acrescentar variáveis temporárias, cuja finalidade específica é armazenar estes valores. Em princípio, uma nova variável temporária deve ser introduzida para cada ocorrência de um operador (+, *, ...). Isso não quer dizer, naturalmente, que, em tempo de execução, cada uma dessas variáveis

temporárias deverá corresponder, de forma permanente, a uma posição de memória, uma vez que, por sua natureza, os valores (temporários) armazenados nessas variáveis só tem interesse durante um intervalo de tempo durante a avaliação de uma expressão. Essa natureza temporária dos valores faz com que uma posição de memória possa conter valores de várias variáveis temporárias, em momentos distintos da execução.

Se possível, estes valores são mantidos em registradores da máquina ou no topo da pilha de hardware, onde o acesso pode ser feito de forma mais rápida. Registradores e posições no topo da pilha são rapidamente liberados quando a avaliação da expressão termina. Na fase de geração de código intermediário, portanto, não há inconveniente em criar uma temporária nova para cada ocorrência de um operador, uma vez que a criação de uma temporária nova não implica necessariamente em ocupação adicional de espaço de memória, em tempo de execução. Por essa razão, em nossos exemplos, vamos usar uma função `temp()`[1], que cria a cada chamada uma temporária nova. Dos atributos de uma variável declarada pelo programador (nome, endereço, tipo, forma de alocação, ...) uma variável temporária só não tem o nome.

Segundo, observamos que a geração de código em atributos seria extremamente custosa. Por exemplo, numa regra $L^0 \rightarrow L^1;S$ o código associado a L^0 é obtido pela concatenação (justaposição) dos trechos de código gerados para L^1 e para `S`. Em princípio, isso significa que todo o código gerado para L^1 e `S` deve ser copiado para obter o código de L^0. No caso de uma atribuição $S \rightarrow V:=E$ o código de `S` é composto pelo código que avalia a expressão `E`, mais uma instrução que armazena no endereço de `V` o valor de `E`. Assim, o código de `E` seria repetido dentro do código de `V:=E`. Naturalmente, em exemplos mais complicados, teríamos, além disso, que considerar a necessidade de gerar código associado a `V`, cuja finalidade seria calcular o endereço da variável.

Mais eficiente que o uso de atributos é o uso de uma tabela (ou um arquivo) para a geração do código: as ações executadas durante as reduções para L^1 e `S` geram o código correspondente na tabela em posições sucessivas. Durante a ação associada à redução para L^0, nenhuma ação adicional de geração de código precisa ser tomada. Da mesma forma, no caso do comando de atribuição, o código de `E` é gerado na tabela, e na geração de código de `C`, apenas a instrução adicional precisa ser gerada.

Vamos supor que uma tabela assim está sendo usada para a geração do código, e que uma instrução (uma quádrupla) `[op op1 op2 op3]` pode ser gerada através de uma chamada de função `gera(op, op1, op2, op3)`. Uma variável `prox` indica a primeira posição vazia na tabela; cada vez que uma instrução é gerada na posição `prox`, a função gera preenche os quatro campos e incrementa `prox`. Consideramos que todas as instruções tem tamanho `1`, e que cada instrução tem um número (ou endereço na área de código), pelo qual podemos fazer referência a ela, como será necessário quando considerarmos instruções de desvio.

Os operadores das quádruplas podem ser introduzidos, de acordo com as necessidades de cada caso. Para gerar código para as listas de comandos da gramática usada como exemplo na seção 4.2, são necessárias apenas instruções de soma, produto e cópia (atribuição). Podemos considerar que

[1] Os parênteses são usados apenas para lembrar que se trata de uma chamada de função, que devolve valores diferentes a cada vez.

+	a	b	c

tem como efeito a soma dos valores encontrados nos endereços `a` e `b` e a colocação do resultado no endereço `c`. A instrução de produto,

*	a	b	c

tem efeito semelhante. A instrução de cópia

:=	a	b	-

não tem propriamente um resultado. Seu efeito é copiar o conteúdo do endereço `a` para o endereço `b`. O terceiro operando não precisa ser especificado. Outras instruções podem ser usadas, quando necessário. Por exemplo,

JF	a	b	-

tem como efeito o desvio para a `b`-ésima instrução, se o conteúdo da posição `a` for interpretado como falso. Outro exemplo é o desvio incondicional (para a `b`-ésima instrução)

J	b	-	-

Durante o restante deste capítulo, vamos introduzir outras instruções, para acesso a arrays, para chamada e retorno de procedimentos, para passagem de parâmetros, etc.

Uma das tarefas a serem executadas na compilação é a atribuição de endereços efetivos às variáveis usadas no programa e às variáveis (temporárias) definidas durante o próprio processo de compilação. A não ser em casos especiais, não interessa ao usuário como essa alocação é feita, nem se os valores são guardados em posições fixas da memória. Por exemplo, a variável `i` usada no comando

```
for i:=1 to 10 do x
```

pode nunca ter seu valor escrito na memória: para executar `x` 11 vezes, basta escrever 11 em um registrador `r`, e cada vez que `x` for executada decrementar `r` de 1. O processo se encerra quando `r` contém 0. Note que nenhuma posição de memória contém o valor de `i`. Nem mesmo o registrador `r` contém esse valor.

Além disso, em uma linguagem com estrutura de blocos (Algol-like), as variáveis declaradas em um bloco só tem espaço alocado durante uma ativação de um bloco. (Este é o tipo de alocação *automático* de C.) Em caso de várias ativações de um bloco (chamadas recursivas de um procedimento, por exemplo), cada ativação do bloco contém uma ativação de cada variável declarada no bloco, de forma que o mais perto que podemos chegar do endereço de uma variável é a uma fórmula que nos permite calcular o endereço da variável no momento da execução. Voltaremos a esse assunto no Cap. 7, onde vamos discutir as estruturas de tempo de execução necessárias para a implementação de linguagens com estrutura de blocos.

A decisão sobre a alocação de endereços às variáveis é normalmente feita durante a geração de código, após a geração da representação intermediária. Portanto, os endereços usados nas quádruplas devem ser entendidos como uma identificação das variáveis envolvidas, a ser substituída pela forma de endereçamento efetivo das variáveis durante a geração de código. Normalmente, usamos como endereço de uma variável, nesta fase, o endereço do registro (a linha) na tabela de símbolos correspondente à variável.

Em particular, isso significa que as variáveis temporárias devem ser incluídas na tabela de símbolos, por ocasião de sua criação pela função `temp()`. Ao contrário das variáveis definidas pelo programador, estas variáveis não precisam de nomes, mas, por conveniência de notação, usaremos, nos exemplos, nomes como `t1`, `t2`, ... para fazer referências às diversas temporárias.

Nas próximas seções, vamos ver várias técnicas de geração de código de três endereços, para várias situações diferentes. Nesta fase inicial, suporemos que as declarações das variáveis usadas já foram examinadas, e que os dados relevantes se encontram em uma tabela de símbolos.

Alguma confusão pode ser proveniente da separação entre *tempo de compilação* e *tempo de execução*. Em tempo de compilação, geramos código, e manipulamos informações auxiliares (atributos, tabelas de símbolos). O código gerado não é executado nesta fase, e, portanto, as instruções individuais não precisam ser geradas na mesma ordem em que vão ser executadas[2]. Por exemplo, posições podem ser deixadas em branco durante a geração de uma seqüência de instruções, para serem preenchidas posteriormente. Por outro lado, atributos e tabelas de símbolos não podem ser consultados, durante a execução, uma vez que só existem em tempo de compilação.

5.4. - Técnicas Básicas de Geração de Quádruplas

Para apresentação das técnicas básicas de geração de código em quádruplas, vamos gerar código para as construções mais comuns, de uma forma relativamente simples. Considere a gramática

```
L → L ; S  | S
S → V := E | begin L end
E → E + T  | T
T → T * F  | F
F → ( E )  | V
V → id
```

Essa gramática gera comandos em uma linguagem de programação, e deve ser entendida como um fragmento de uma gramática maior, para a linguagem de programação completa, em que estão também incluídas regras para outros tipos de comandos, e para as declarações. Suporemos que o tratamento de declarações já foi feito, e que existe uma tabela de símbolos na qual podem ser encontrados os endereços das variáveis, em função dos nomes dos identificadores correspondentes (valores do atributo sintetizado `id.nome`). Como observado anteriormente, podemos considerar que esses endereços são os números das linhas ou os endereços dos registros correspondentes às variáveis na tabela de símbolos. Além disso, vamos supor que o código é gerado em uma tabela de quádruplas, com auxílio do procedimento `gera` e da variável `prox` discutidos na seção anterior. Os atributos (sintetizados) correspondentes a cada símbolo estão indicados abaixo.

[2] Em particular, o compilador não precisa "entrar em loop" para gerar as instruções de um comando de repetição. ☺

símbolos	atributos
L, S	-
E, T, F, V	end
id	nome

Vamos iniciar pela regra V→id. Para essa regra, devemos consultar a tabela de símbolos, através de uma rotina apropriada, e obter o endereço end de V, em função do nome nome de id. Ou seja,

```
V → id
      V.end:=consulta(id.nome);
```

Essa especificação deve se traduzir em código executável quando o analisador sintático (ascendente) sinalizar a redução pela regra V→id. Como indicado na seção final do Cap. 4, o atributo sintetizado id.nome pode ser encontrado na pilha de atributos, de onde deve ser retirado, (digamos) por uma rotina pop, e o atributo sintetizado V.end deve ser empilhado, após o seu cálculo, por uma rotina (digamos) push. Ou seja, o código verdadeiro a ser executado deve se assemelhar a

```
begin
  pop(idnome);
  vend:=consulta(idnome);
  push(vend);
end;
```

onde idnome e vend são variáveis locais dos tipos apropriados.

Considere agora a regra F→V. Sabemos que precisamos calcular F.end, em função de V.end. Mas F e V correspondem exatamente à mesma variável, e portanto os dois endereços são iguais. ou seja,

```
F → V
      F.end := E.end;
```

Mas isso corresponde a

```
begin
  pop(vend);
  fend:=vend;
  push(fend);
end;
```

cujo efeito é exatamente nenhum. (As variáveis fend e vend são variáveis locais de uso temporário, de forma que a alteração de seus valores é irrelevante.) Por essa razão, podemos indicar que nenhuma ação é necessária para essa regra por

```
F → V
      { F.end := E.end } ;
```

Neste caso, as chaves (notação de comentário em Pascal) indicam que a ação não precisa ser executada. Outra forma equivalente é

```
F → V
      ;
```

Para as regras E→T, T→F e F→(E), a situação é semelhante:

```
E → T
      ;
T → F
      ;
```

```
F → ( E )
      ;
```

A regra $E^0 \rightarrow E^1+T$, entretanto, deve gerar código para a soma. Para gerar a instrução correspondente, precisamos de um endereço para E^0. Esse endereço deve ser o de uma variável temporária, criada especialmente, por uma chamada da função `temp()`. A ação correspondente é

```
E⁰ → E¹ + T
      E⁰.end := temp();
      gera(+, E¹.end, T.end, E⁰.end);
```

correspondendo a

```
begin
  pop(tend);
  pop(e1end);
  e0end:=temp();
  gera(soma, e1end, tend, e0end);
  push(e0end);
end;
```

supondo que soma é a constante (normalmente um valor de um tipo de enumeração) que representa o operador +. Note que a ordem de execução é importante, uma vez que a chamada de gera não pode ser feita antes do cálculo de E^0`.end`. Para o produto, a situação é semelhante:

```
T⁰ → T¹ * F
      T⁰.end := temp();
      gera(*, T¹.end, F.end, T⁰.end);
```

A regra do comando de atribuição tem um tratamento simples:

```
S → V := E
      gera(:=, V.end, E.end, -);
```

onde o traço (-) indica a ausência de um elemento na instrução. No momento da geração da quádrupla, entretanto, um valor qualquer deve ser passado como quarto parâmetro de gera.

Para a regra $L^0 \rightarrow L^1;S$, nada há a fazer. O código correspondente aos símbolos do lado direito (L^1 e S) já se encontra gerado; terminada a execução do código de L^1, o código de S passa a ser executado automaticamente, *por gravidade*; L^0 não tem atributos a calcular. A situação das regras $L \rightarrow S$ e $S \rightarrow$`begin L end` é semelhante.

```
L⁰→L¹;S
      ;
L→S
      ;
S→begin L end
      ;
```

Exemplo 1: Considere a cadeia `c:=(b+c)*a;a:=b*c`. Vamos mostrar os passos da análise, as ações executadas, e as quádruplas correspondentes. Inicialmente, `prox` tem o valor `1`. As colunas da tabela mostram a pilha sintática, os primeiros símbolos do resto da entrada, a pilha de atributos e a ação correspondente. Para facilitar a notação, vamos usar `a`, `b` e `c` para representar os endereços das variáveis `a`, `b` e `c`, e `t1`, `t2`, ... para representar os endereços de temporárias fornecidos por chamadas sucessivas de `temp()`.

	pilha sintática	entrada	pilha de atributos	ação
		c:=(b+c)*a;a:=		empilha id
	id	:=(b+c)*a;a:=b	c	reduz V→id
	V	:=(b+c)*a;a:=b	c	empilha :=
	V:=	(b+c)*a;a:=b*c	c	empilha (
	V:=(	b+c)*a;a:=b*c	c	empilha id
	V:=(id	+c)*a;a:=b*c	c b	reduz V→id
	V:=(V	+c)*a;a:=b*c	c b	reduz F→V
	V:=(F	+c)*a;a:=b*c	c b	reduz T→F
	V:=(T	+c)*a;a:=b*c	c b	reduz E→T
	V:=(E	+c)*a;a:=b*c	c b	empilha +
	V:=(E+	c)*a;a:=b*c	c b	empilha id
	V:=(E+id	)*a;a:=b*c	c b c	reduz V→id
	V:=(E+V	)*a;a:=b*c	c b c	reduz F→V
	V:=(E+F	)*a;a:=b*c	c b c	reduz T→F
	V:=(E+T	)*a;a:=b*c	c b c	reduz E→E+T
		1:[+ b c t1]		prox:=2
	V:=(E	)*a;a:=b*c	c t1	empilha)
	V:=(E)	*a;a:=b*c	c t1	reduz F→(E)
	V:=F	*a;a:=b*c	c t1	reduz T→F
	V:=T	*a;a:=b*c	c t1	empilha *
	V:=T*	a;a:=b*c	c t1	empilha id
	V:=T*id	;a:=b*c	c t1 a	reduz V→id
	V:=T*V	;a:=b*c	c t1 a	reduz F→V
	V:=T*F	;a:=b*c	c t1 a	reduz T→T*F
		2:[* t1 a t2]		prox:=3
	V:=T	;a:=b*c	c t2	reduz E→T
	V:=E	;a:=b*c	c t2	reduz S→V:=E
		3:[:= c t2 -]		prox:=4
	S	;a:=b*c		reduz L→S
	L	;a:=b*c		empilha ;
	L;	a:=b*c		empilha id
	L;id	:=b*c	a	reduz V→id
	L;V	:=b*c	a	empilha :=
	L;V:=	b*c	a	empilha id
	L;V:=id	*c	a b	reduz V→id
	L;V:=V	*c	a b	reduz F→V
	L;V:=F	*c	a b	reduz T→F
	L;V:=T	*c	a b	empilha *
	L;V:=T*	c	a b	empilha id
	L;V:=T*id		a b c	reduz V→id
	L;V:=T*V		a b c	reduz F→V
	L;V:=T*F		a b c	reduz T→T*F
		4:[* b c t3]		prox:=5
	L;V:=T		a t3	reduz E→T
	L;V:=E		a t3	reduz S→V:=E
		5:[:= a t3 -]		prox:=6
	L;S			reduz L→L;S
	L			

Os comandos gerados neste exemplo são os seguintes:

```
1:[ +  b  c  t1 ]          { t1:=b+c;  }
2:[ *  t1 a  t2 ]          { t2:=a*t1; }
3:[ := c  t2 -  ]          { c:=t2;    }
4:[ *  b  c  t3 ]          { t3:=b*c;  }
5:[ := a  t3 -  ]          { a:=t3;    }
6:
```

5.5 - Geração de quádruplas para comandos condicionais e de repetição

Vamos agora acrescentar alguns comandos (comandos `if`, `while` e `repeat`) cujo tratamento é um tanto dificultado pela restrição feita: apenas atributos sintetizados, que são avaliados durante a análise sintática ascendente. As regras para esses comandos serão acrescentadas a uma gramática básica, que corresponde ao esquema de tradução abaixo:

```
1. S⁰ → S¹ ; S²
     ;
2. S → V := E
     gera(:=, V.end, E.end, -);
3. E⁰ → E¹ + E²
     E⁰.end := temp();
     gera(+, E¹.end, E².end, E⁰.end);
4. E⁰ → E¹ * E²
     E⁰.end := temp();
     gera(*, E¹.end, E².end, E⁰.end);
5. E → ( E )
     ;
6. E → V
     ;
7. V → id
     V.end:=consulta(id.nome);
```

Esta gramática é ambígua, (como outras que usaremos neste capítulo), mas seu tratamento semântico é mais simples do que o da gramática usada anteriormente. Neste caso, vamos supor que está disponível um analisador sintático da gramática, não nos interessando aqui a forma de sua construção. O analisador pode ter sido obtido por *"cirurgia"*, removendo seletivamente algumas entradas de tabelas para eliminação de conflitos, ou pode ser baseado em um analisador sintático construído para outra gramática equivalente não ambígua.

Comando `if`.

Vamos acrescentar duas regras para este comando:

```
8. S⁰ → if E then S¹
9. S⁰ → if E then S¹ else S²
```

Como observado acima, vamos ignorar aqui a ambigüidade implícita nessas regras, supondo que uma solução satisfatória para esse problema foi encontrada na construção do analisador sintático.

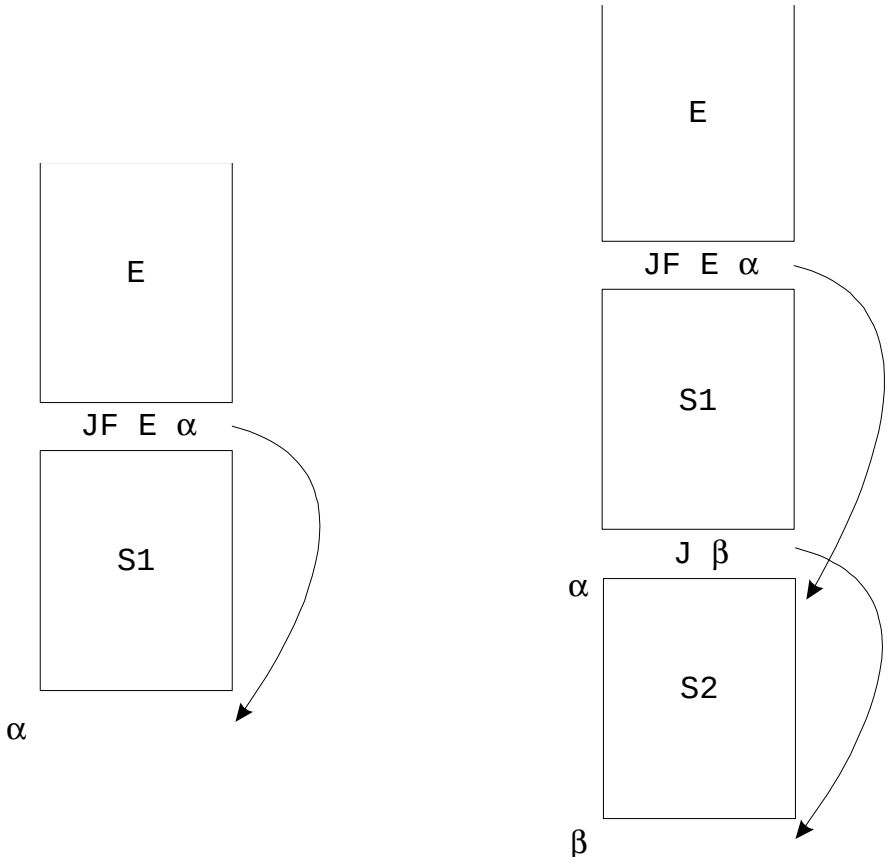


Fig. 3 - Código para o comando if

Considere a regra 8. Podemos observar que o código do comando `S`^{0} só pode incluir o código correspondente à expressão `E` e ao comando `S`^{1} na ordem em que eles ocorrem: primeiro o código de `E`, depois o código de `S`^{1}. Isso se deve à nossa decisão de só usar atributos sintetizados, e de calculá-los na ordem em que o analisador sintático trata as frases correspondentes na cadeia de entrada. Entretanto, não queremos que a execução se dê sempre nessa ordem, nem que o comando seja sempre executado. Para evitar a execução de `S`^{1} quando a expressão tem o valor falso, é necessário um desvio condicionado ao valor de `E` depois de sua avaliação e antes da execução de `S`^{1}. Usaremos para isso uma quádrupla `JF`, cujo primeiro operando (o endereço do valor a ser testado) é o endereço de `E`, e cujo segundo operando (o ponto para o qual o desvio é feito) é a instrução seguinte a `S`^{1}. (Ver Fig. 3(a).) Entretanto, nosso esquema seqüencial de geração de código não permite a inclusão de uma instrução exceto na primeira posição vazia, que é a posição indicada pela variável `prox`. Para que uma ação seja incluída entre a geração do código de `E` e a geração do código de `S`^{1}, é necessária uma redução nesse ponto, isto é, quando `prox` aponta para uma instrução (ainda) vazia imediatamente após a última instrução do código de `E`.

Há duas maneiras simples de alterar a gramática para isso. Uma dessas alterações quebra o lado direito da regra em duas partes `I` e `T`:

`S`^{0} → `I T`
`I` → `if E`
`T` → `then S`^{1}

(Por comodidade, continuamos fazendo referência ao comando `if` como S^0, e ao comando da parte `then` como S^1.) A redução para `I` acontece então entre a redução para `E` e a redução para S^1, e pode gerar a instrução `JF`.

Outra alteração possível acrescenta um nãoterminal especial `M` à regra:

$S^0 \rightarrow$ `if E then M` S^1

`M` $\rightarrow \varepsilon$

Esse nãoterminal `M` não gera nenhum símbolo terminal, para não interferir com a sintaxe da linguagem, mas provê a redução adicional no ponto desejado.

Entretanto, uma complicação ainda persiste: as reduções assim criadas não permitem a construção completa da instrução `JF`. No caso da redução para `I`, `E.end` está disponível, mas como o código de S^1 ainda não foi gerado, não sabemos ainda para onde o desvio deve ser feito. No caso da redução para `M`, nem mesmo `E.end` está disponível para a construção da instrução `JF`, pela simples razão de que não há nenhuma ocorrência de `E` na regra `M`$\rightarrow\varepsilon$.

No primeiro caso, o tratamento pode ser

```
S⁰ → I T
     remenda(I.quad, prox);
I → if E
     I.quad:=prox;
     gera(JF, E.end, -, -);
T → then S¹
     ;
```

Note que (na redução para `I`), a instrução é gerada com o campo de desvio em branco, mas o endereço dessa quádrupla incompleta é anotado no atributo `quad` de `I`. Posteriormente (na redução para S^0) um remendo é feito: a informação que falta é acrescentada à instrução. Por ocasião desta última redução, o código de S^1 foi o último código gerado, e portanto `prox` aponta para a primeira instrução depois de S^1, que ainda se encontra vazia.

No outro caso, em que o nãoterminal `M` foi introduzido, preferimos não gerar a instrução incompleta `[JF - - -]`. Vamos apenas reservar uma posição para essa instrução, deixando para o remendo a construção da instrução completa, com todos os campos relevantes preenchidos. (Naturalmente, trata-se aqui de um procedimento `remenda` diferente.)

```
S⁰ → if E then M S¹
     remenda(M.quad, JF, E.end, prox, -);
M → ε
     M.quad:=prox;
     prox:=prox+1;
```

Note que `E.end` está disponível na redução para S^0, e que `prox` aponta nesse momento para a primeira instrução depois do código de S^1.

O tratamento de $S^0 \rightarrow$ `if E then` S^1 `else` S^2 é semelhante. Neste caso, devemos ter uma quádrupla `JF` entre `E` e S^1, que transfere o controle para S^2 quando a expressão tem o valor falso, e uma quádrupla de desvio incondicional `J` entre S^1 e S^2, que transfere o controle para a primeira instrução após o código de S^2, evitando sua

execução quando a expressão tem valor verdadeiro. Podemos ter, estendendo o tratamento anterior:

```
S⁰ → I T' E
     remenda(I.quad, T'.quad+1);
     remenda(T'.quad, prox);
I → if E                          { como anteriormente }
     I.quad:=prox;
     gera (JF, E.end, -, -);
T' → then S¹
     T'.quad:=prox;
     gera(J, - ,- ,-);
E → else S²
     ;
```

ou

```
S⁰ → if E then M¹ S¹ else M² S²
  remenda(M¹.quad, JF. E.end, M².quad+1, -);
  remenda(M².quad, J, prox, -, -);
M → ε                             { como anteriormente }
  M.quad:=prox;
  prox:=prox+1;
```

Usamos um novo nãoterminal `T'` na regra do if-then-else, porque queremos para `T'` uma semântica diferente da semântica de `T`.

A escolha entre gerar a instrução incompleta e depois remendar preenchendo apenas os campos relevantes vazios e a outra possibilidade de apenas reservar espaço para a instrução e depois remendar preenchendo todos os campos da instrução é independente da alteração sintática escolhida. Naturalmente, as funções `remenda` para essas duas possibilidades são diferentes.

Nossa preferência recai sobre a introdução dos nãoterminais `M` e da reserva de espaço para a instrução. A outra forma de alteração da sintaxe foi, durante muitos anos, a preferida pelos construtores de compiladores, simplesmente porque métodos populares de análise sintática, em particular os métodos de *precedência simples*, e de *precedência de operadores* não permitiam o tratamento de regras com lado direito vazio. Tais métodos são hoje considerados obsoletos.

Continuando nosso exemplo, temos então

```
8. S⁰ → if E then M S¹
     remenda(M.quad, JF, E.end, prox, -);
9. S⁰ → if E then M¹ S¹ else M² S²
     remenda(M¹.quad, JF. E.end, M².quad+1, -);
     remenda(M².quad, J, prox, -, -);
10. M → ε
     M.quad:=prox;
     prox:=prox+1;
```

Nenhuma das alterações sintáticas da forma aqui apresentada costuma introduzir conflitos novos na gramática, durante a alteração (reconstrução) do analisador sintático. Com o uso de um gerador de analisadores sintáticos essa reconstrução torna pouco importante o custo de ajustar a gramática à semântica, e a programação das ações semânticas pode ser feita da forma mais confortável possível, reduzindo a oportunidade para erros a um mínimo.

Observações.

1. Se um símbolo não terminal novo como `M` acima deve ser introduzido entre dois nãoterminais já existentes , por exemplo entre `E` e S^1 na regra

   $S^0 \rightarrow$ `if E then` S^1

a posição desse nãoterminal em relação aos terminais presentes na regra (`then`, no exemplo) é irrelevante. No exemplo, poderíamos fazer

$S^0 \rightarrow$ `if E M then` S^1

ou, equivalentemente,

$S^0 \rightarrow$ `if E then M` S^1.

2. Entretanto, devemos observar que um conflito pode ser introduzido se existir outra regra que tenha um trecho em comum com a regra considerada. Isso acontece, no exemplo, se decidirmos incluir as regras

   $S^0 \rightarrow$ `if E M then` S^1
   $S^0 \rightarrow$ `if E then M` S^1 `else` S^2.

O conflito ocorre no estado que contém os itens

`S` $\rightarrow$ `if E`•`M then S`
`S` $\rightarrow$ `if E`•`then M S else S`
`M` $\rightarrow$ •

uma vez que não é possível decidir entre um possível empilhamento de `then` e uma possível redução para `M`.

3. Devemos ainda observar que, mesmo que a introdução de nãoterminais como `M` não crie novos conflitos, pode, assim mesmo, alterar conflitos já existentes. Por exemplo, a presença das regras

   `S` $\rightarrow$ `if E then S`
   `S` $\rightarrow$ `if E then S else S`

leva a um conflito *empilha-reduz* (empilhar o `else` ou reduzir pela regra mais curta):

`S` $\rightarrow$ `if E then S`•
`S` $\rightarrow$ `if E then S`•`else S`

Com a introdução do `M`, teremos, em vez disso,

`S` $\rightarrow$ `if E M then S`•
`S` $\rightarrow$ `if E M then S`•`M else S`

e o conflito será da forma *reduz-reduz* (reduzir pela regra mais curta ou por M$\rightarrow\varepsilon$). O conflito ocorre, entretanto, no mesmo ponto da cadeia de entrada, imediatamente antes do `else`.

Exemplo 2: Considere o trecho de programa

```
if a+b then if b then a:=b else a:=c*a
```

Vamos gerar quádruplas para esse trecho de código usando as ações descritas acima. A coluna da entrada apresenta apenas os primeiros símbolos do resto da entrada; na coluna de ação, s indica empilhamento, e r indica redução.

pilha sintática	entrada	pilha de atributos	ação
	if a+b then if		s if
if	a+b then if b		s id
if id	+b then if b t	a	r7
if V	+b then if b t	a	r6
if E	+b then if b t	a	s +
if E +	b then if b th	a	s id
if E + id	then if b then	a b	r7
if E + V	then if b then	a b	r6
if E + E	then if b then	a b	r3
	1:[+ a b t1]		prox:2
if E	then if b then	t1	s then
if E then	if b then a:=b	t1	r10
	2:[- - - -]		prox:3
if E then M	if b then a:=b	t1 2	s if
if E then M if	b then a:=b el	t1 2	s id
if E then M if id	then a:=b else	t1 2 b	r7
if E then M if V	then a:=b else	t1 2 b	r6
if E then M if E	then a:=b else	t1 2 b	s then
if E then M if E then	a:=b else a:=c	t1 2 b	r10
	3:[- - - -]		prox:4
if E then M if E then M	a:=b else a:=c	t1 2 b 3	s id
if E then M if E then M id	:=b else a:=c*	t1 2 b 3 a	r7
if E then M if E then M V	:=b else a:=c*	t1 2 b 3 a	s :=
if E then M if E then M V :=	b else a:=c*a	t1 2 b 3 a	s id
if E then M if E then M V := id	else a:=c*a	t1 2 b 3 a b	r7
if E then M if E then M V := V	else a:=c*a	t1 2 b 3 a b	r6
if E then M if E then M V := E	else a:=c*a	t1 2 b 3 a b	r2
	4:[:= a b -]		prox:5
if E then M if E then M S	else a:=c*a	t1 2 b 3	s else
if E then M if E then M S else	a:=c*a	t1 2 b 3	r10
	5:[- - - -]		prox:6
if E then M if E then M S else M	a:=c*a	t1 2 b 3 5	s id
if E then M if E then M S else M id	:=c*a	t1 2 b 3 5 a	r7
if E then M if E then M S else M V	:=c*a	t1 2 b 3 5 a	s :=
if E then M if E then M S else M V :=	c*a	t1 2 b 3 5 a	s id
if E then M if E then M S else M V := id	*a	t1 2 b 3 5 a c	r7
if E then M if E then M S else M V := V	*a	t1 2 b 3 5 a c	r6

pilha sintática	entrada	pilha de atributos	ação
`if E then M if E then M S else M V := E`	`*a`	`t1 2 b 3 5 a c`	`s *`
`if E then M if E then M S else M V:=E *`	`a`	`t1 2 b 3 5 a c`	`s id`
`if E then M if E then M S else M V:=E * id`		`t1 2 b 3 5 a c a`	`r7`
`if E then M if E then M S else M V:=E * V`		`t1 2 b 3 5 a c a`	`r6`
`if E then M if E then M S else M V:=E * E`		`t1 2 b 3 5 a c a`	`r4`
	`6:[* c a t2]`		`prox:7`
`if E then M if E then M S else M V:=E`		`t1 2 b 3 5 a t2`	`r2`
	`7:[:= a t2 -]`		`prox:8`
`if E then M if E then M S else M S`		`t1 2 b 3 5`	`r9`
	`3:[JF b 6 -]` `5:[J 8 - -]`		
`if E then M S`		`t1 2`	`r8`
	`2:[JF t1 8 -]`		
`S`			

Os comandos gerados são

```
1:[ +  a  b  t1 ]        { t1:=a+b;          }
2:[ JF t1 8  -  ]        { if not t1 goto 8; }
3:[ JF b  6  -  ]        { if not b  goto 6; }
4:[ := a  b  -  ]        { a:=b;             }
5:[ J  8  -  -  ]        { goto 8;           }
6:[ *  c  a  t2 ]        { t2:=a*c;          }
7:[ := a  t2 -  ]        { a:=t2;            }
8:
```

Exercício 1: Substitua as regras dos comandos if acima pelas regras

$S \rightarrow$ `if E then S X`

$X \rightarrow$ `elsif E then S X | else S end | end`

que descrevem a sintaxe do comando `if` de Modula-2. Mostre como fazer o tratamento semântico correspondente.

Comandos de repetição: while, repeat.

Para os comandos `while` e `repeat`, a situação é semelhante. Partimos das regras $S^0 \rightarrow$`while E do` S^1 e $S^0 \rightarrow$`repeat` S^1 `until E`. Observando a Fig. 4, notamos que são necessárias alterações nessas regras, como no caso das regras do comando `if`.

No caso do comando `while`, um nãoterminal `M` (entre `E` e S^1) reserva uma quádrupla que é remendada posteriormente, de forma semelhante ao caso do comando if-then. Entretanto, um desvio adicional é necessário, para fechar o loop, e deve ser feito para a primeira instrução de `E`, cuja posição pode ser guardada num atributo

quad de um nãoterminal N, introduzido antes de E. Este nãoterminal deve ser diferente de M, porque não há, neste caso, necessidade de reservar uma instrução para preenchimento posterior.

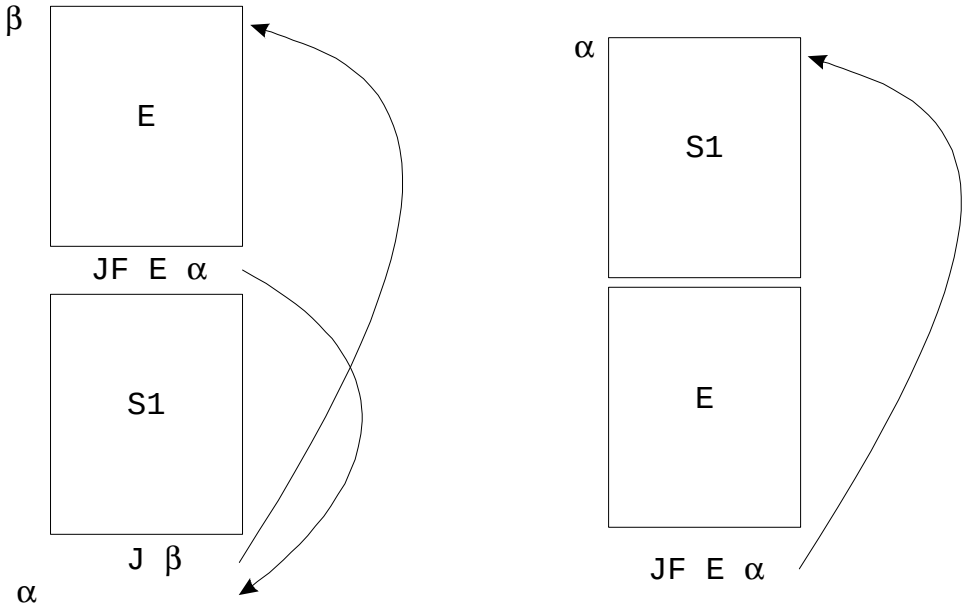


Fig 4. – Comandos while (a) e repeat (b)

O caso do repeat-until pode ser tratado de forma semelhante. Temos

```
10. M → ε                          { como anteriormente }
      M.quad:=prox;
      prox:=prox+1;
11. S⁰ → while N E do M S¹
      gera(J, N.quad, -, -);
      remenda(M.quad, JF, E.end, prox, -);
12. S⁰ → repeat N S¹ until E
      gera(JF, E.end, N.quad, -);
13. N → ε
      N.quad:=prox;
```

Exercício 2: Acrescente à gramática acima uma regra S→do S while E, com o tratamento semântico adequado.

5.6 - Um caso mais complicado: comandos loop e exit.

O tratamento da combinação loop-exit traz algumas complicações adicionais. Se tivermos, por exemplo

```
loop
  S1;
  if E1 then S2 else exit;
  S3;
  if E4 then exit;
  S5;
end;
```

teremos dois pontos de saída (exit) do loop. Cada um dos dois comandos exit corresponde a um desvio para a primeira instrução após o end do loop. Executado o comando S5, o controle deve ser transferido de volta para S1. Para a geração do

código, cada `exit` gera um desvio incondicional; a lista (ou conjunto) desses desvios (que devem ser remendados posteriormente) será guardada em um atributo `S.exits`, especialmente introduzido. Para os valores do atributo `exits`, que são listas de instruções, vamos usar a notação de conjuntos, deixando a efetiva implementação como exercício. Temos:

```
13. N → ε                          { como anteriormente }
      N.quad:=prox;
14. S⁰ → loop N S¹ end
      gera(J, N.quad, -, -);
      remenda(S¹.exits, J, prox, -, -);
      S⁰.exits:=∅;
15. S → exit
      S.exits:={prox};
      gera(J, -, -, -);
```

Note que a função remenda aqui usada é uma generalização da versão anterior: seu primeiro argumento é uma lista de instruções a remendar.

De uma forma geral, toda vez que uma regra é acrescentada ou alterada, o esquema de tradução precisa ser revisto para toda a gramática. No nosso caso presente, um atributo `S.exits` foi acrescentado, e, no mínimo, todas as regras de S devem ser revistas, para incluir o cálculo do novo atributo. Temos, neste caso:

```
1. S⁰ → S¹ ; S²
      S⁰.exits:=S¹.exits ∪ S².exits;
2. S → V := E
      gera(:=, V.end, E.end, -);
      S.exits:=∅;
8. S⁰ → if E then M S¹
      remenda(M.quad, JF, E.end, prox, -);
      S⁰.exits:=S¹.exits;
9. S⁰ → if E then M¹ S¹ else M² S²
      remenda(M¹.quad, JF. E.end, M².quad+1, -);
      remenda(M².quad, J, prox, -, -);
      S⁰.exits:=S¹.exits ∪ S².exits;
11. S⁰ → while N E do M S¹
      gera(J, N.quad, -, -);
      remenda(M.quad, JF, E.end, prox, -);
      S⁰.exits:=S¹.exits;
12. S⁰ → repeat N S¹ until E
      gera(JF, E.end, N.quad, -);
      S⁰.exits:=S¹.exits;
```

Exemplo 3: Considere a cadeia

```
loop
  a:=b+c;
  if a then exit;
  b:=a;
  if b then exit
  else a:=c
end
```

O código gerado pelo esquema de tradução apresentado para esta cadeia é

```
 1:[ +  b  c  t1 ]
 2:[ := a  t1 -  ]
 3:[ JF a  5  -  ]              { teste do primeiro if }
 4:[ J  11 -  -  ]              { primeiro exit }
 5:[ := a  b  -  ]
 6:[ JF b  9  -  ]              { teste do segundo if }
 7:[ J  11 -  -  ]              { segundo exit }
 8:[ J  10 -  -  ]              { pulo por cima do else }
 9:[ := a  c  -  ]
10:[ J  1  -  -  ]              { fecha o loop }
11:
```

A possibilidade de melhora (otimização) deste código é evidente. Veremos na última seção deste capítulo uma técnica de otimização (otimização de janela, ou *peephole optimization*) que, em nosso exemplo, é capaz de identificar e eliminar construções como desvios para desvios (8), e encontrar situações em que o código pode ser melhorado com a troca do teste a ser feito (por exemplo, trocar `JF` por `JT`). Usando essa técnica, podemos melhorar este código, obtendo como resultado:

```
1:[ +  b  c  t1 ]
2:[ := a  t1 -  ]
3:[ JT a  8  -  ]
4:[ := b  a  -  ]
5:[ JT b  8  -  ]
6:[ := a  c  -  ]
7:[ J  1  -  -  ]
8:
```

Exercício 3: Verifique, passo a passo, a construção do código correspondente ao trecho de programa do Exemplo 3. Verifique também a equivalência entre o código original e o código otimizado apresentado.

5.7 - Geração de código de fluxo de controle.

A maneira mais simples de gerar código para expressões booleanas é a de tratar os operadores booleanos `and`, `or`, `not`, ... em uma expressão booleana da mesma forma que os operadores aritméticos em uma expressão aritmética qualquer. Por exemplo, a expressão `a and b or not c and d` levaria à geração de

```
[ and a  b  t1 ]
[ not c  t2 -  ]
[ and t2 d  t3 ]
[ or  t1 t3 t4 ]
```

em que o endereço final da expressão é `t4`.

Entretanto, numa expressão booleana α `or` β, não há necessidade de avaliar β se o resultado da avaliação de α é verdadeiro; numa expressão booleana α `and` β, não há necessidade de avaliar β se o resultado da avaliação de α é falso. Para isto, supomos que a avaliação de β, em ambos os casos, não produz efeitos colaterais, isto é, nenhuma alteração de valores de variáveis globais ou ação de entrada/saída é provocada pela execução de β.

Este tipo de avaliação, chamada *avaliação de curto-circuito* leva a um código mais eficiente que aquele obtido pelo tratamento da expressão booleana como uma expressão aritmética comum, mas só pode ser aplicado se a inexistência de efeitos colaterais for garantida. A dificuldade de determinação, em tempo de compilação, da possibilidade de efeitos colaterais em tempo de execução faz com que o controle da forma de código gerado seja muitas vezes passado para o programador. Isto pode ser feito através de opções de compilação em que o usuário indica explicitamente a forma de geração de código desejada, ou através de duas sintaxes diferentes, uma para cada tipo de geração de código. Em Ada, a sintaxe permite a escolha entre `and` e `and then`; entre `or` e `or else`. As formas `and then` e `or else` indicam a geração de código de curto-circuito. Até certo ponto, a situação é semelhante em C, onde podem ser encontrados operadores `&` e `&&`, ou `|` e `||`.

Para a expressão `a and b or not c and d`, vamos gerar o código de curto-circuito

```
1: [ JT a - 3 ]
2: [ J  - - 5 ]
3: [ JT b - α ]
4: [ J  - - 5 ]
5: [ JT c - β ]
6: [ J  - - 7 ]
7: [ JT d - α ]
8: [ J  - - β ]
```

Para geração desse código, uma combinação `[ JT x - γ ] [ J - - δ ]` é gerada, para cada variável `x`. Os destinos γ e δ são determinados com base no contexto em que se encontra a variável, e de acordo com a lógica de curto-circuito. Naturalmente, a combinação equivalente `[ JF x - δ ] [J - - γ ]` poderia também ter sido usada.

Por exemplo, na instrução 1 o valor de `a` é testado. Se `a` é verdadeiro, a instrução seguinte é a 3, onde o valor de `b` é testado (se `a` é verdadeiro, `a and b = b`); se `a` é falso, `a and b` é falso, e `a and b or not c and d = not c and d`. Se `a` é falso, a instrução 1 não executa nenhum desvio, e o controle passa em seqüência para 2, e daí para 5, onde `c` vai ser testado, iniciando o cálculo de `not c and d`.

Para a variável `b`, a situação é semelhante. O valor de `b` só é testado (em 3) se `a` é verdadeiro. Logo, se `b` é verdadeiro, a expressão toda é verdadeira. O desvio é então feito para algum lugar, α, fora do código da expressão, que depende do contexto em que a expressão se encontra. Se esta expressão é usada em um comando if-then-else, α é a primeira instrução do comando após o then. Por outro lado, se `b` é falso, é necessário testar a segunda parte do or.

O restante é semelhante. β indica o destino quando a expressão toda é falsa. No exemplo do comando if-then-else, β seria a primeira instrução do comando após o else.

Outro nome para a geração de código de curto-circuito é "fluxo de controle" (*flow-of-control*). Este nome lembra que não há uma posição de memória que contenha o valor da expressão ou de uma das suas sub-expressões, mas que se pode deduzir o valor de cada (sub-)expressão pela maneira pela qual o controle flui pelo programa, isto é, em função das instruções do programa que são executadas.

Cada expressão pode conter vários pontos onde se verifica que a expressão é verdadeira, e esses pontos devem conter desvios para um ponto como α no exemplo acima; de forma dual, cada expressão pode conter vários pontos onde se verifica que a expressão é falsa, e esses pontos devem conter desvios para um ponto como β no exemplo. Os pontos α e β não podem ser determinados observando apenas a expressão: eles dependem essencialmente do contexto em que a expressão se encontra, que pode ser uma expressão maior ou um comando que contém a expressão.

Outra observação se refere a uma conveniência de notação. Por uniformidade, vamos supor que o campo destino das instruções de desvio é o último. Como os destinos das instruções `JT` e `J` geradas para cada variável são desconhecidos no momento da geração das instruções, o campo correspondente (sempre o quarto) deve ser deixado em branco até que seja preenchido por um remendo da instrução.

Vamos usar a seguinte sintaxe para as expressões booleanas:

`B` $\rightarrow$ `id | E = E | B or B | B and B | not B | ( B )`

onde `B` representa uma expressão booleana. O esquema para a geração de código vai usar atributos `B.true` e `B.false`, que (como `S.exits` na seção 4.6) representam listas (conjuntos) de instruções de desvio cujos destinos não podem ser especificados no momento da geração, e precisam ser remendados posteriormente. A função remenda é diferente das anteriores: uma chamada `remenda(L,`α`)` neste caso precisa apenas preencher o último campo das instruções de desvio pertencentes à lista `L` com o destino α. Para representar os valores das listas `true` e `false`, vamos usar a notação de conjuntos.

Para a regra `B`$\rightarrow$`id`, temos as ações que geram as duas instruções `JT` e `J` incompletas:

```
1. B → id
      B.true:= { prox };
      gera(JT, id.end, -, ? );
      B.false:= { prox };
      gera(J, -, -, ? );
```

supondo que `id.end` é o endereço da variável. Para a regra `B`$\rightarrow$`E=E`, usamos uma instrução `J=` (desvia se igual) em vez de `JT`.

```
2. B → E¹ = E²
      B.true:= { prox };
      gera(J=, E¹.end, E².end, ? );
      B.false:= { prox };
      gera(J, -, -, ? );
```

Outros operadores de comparação podem ser tratados de forma semelhante. Para a regra $B^0 \rightarrow B^1$ `or` B^2, basta observar que se B^1 tem o valor verdadeiro, B^0 tem o mesmo valor; se B^1 tem o valor falso, B^2 deve ser avaliada, e B^0 tem o mesmo valor de B^2. O atributo `N.quad` anota a primeira instrução de B^2. Consequentemente, temos

```
3. B⁰ → B¹ or N B²
      remenda(B¹.false, N.quad);
      B⁰.true:=B¹.true ∪ B².true;
      B⁰.false:=B².false;
```

O tratamento do and é dual:

```
4. B⁰ → B¹ and N B²
       remenda(B¹.true, N.quad);
       B⁰.false:=B¹.false ? B².false;
       B⁰.true:=B².true;
```

ou seja, B^2 só é avaliada se B^1 tiver o valor verdadeiro. Para a regra do not basta apenas trocar as listas true e false:

```
5. B⁰ → not B¹
       B⁰.true:=B¹.false;
       B⁰.false:=B¹.true;
```

Note que estas três últimas regras não geram código: apenas remendam código gerado anteriormente e manipulam atributos. Naturalmente, à regra $B^0 \rightarrow (B^1)$ não corresponde nenhuma ação:

```
6. B⁰ → ( B¹ )
       ;
```

A regra de N é semelhante à usada anteriormente.

```
7. N→ε
       N.quad := prox;
```

Exemplo 4: Considere a expressão booleana acima.

```
a and b or not c
```

A simulação das ações está feita a seguir, com as mesmas convenções anteriores. Para representar as listas true e false, convencionamos que a lista true sempre será empilhada antes da lista false.

Pilha sintática	entrada	Pilha de atributos	ação
	a and		s id
id	and b	a	r1
	1:[JT a - ?]		
	2:[J - - ?]		prox:3
B	and b	{1} {2}	s and
B and	b or n	{1} {2}	r7
B and N	b or n	{1} {2} 3	s id
B and N id	or not	{1} {2} 3 b	r1
	3:[JT b - ?]		
	4:[J - - ?]		prox:5
B and N B	or not	{1} {2} 3 {3} {4}	r4
	1:[JT a - 3]		
B	or not	{3} {2,4}	s or
B or	not c	{3} {2,4}	r7
B or N	not c	{3} {2,4} 5	s not
B or N not	c and	{3} {2,4} 5	s id
B or N not id	and d	{3} {2,4} 5 c	r1
	5:[JT c - ?]		
	6:[J - - ?]		prox:7
B or N not B	and d	{3} {2,4} 5 {5} {6}	r5
B or N B	and d	{3} {2,4} 5 {6} {5}	s and
B or N B and	d	{3} {2,4} 5 {6} {5}	r7
B or N B and N	d	{3} {2,4} 5 {6} {5} 7	s id
B or N B and N id		{3} {2,4} 5 {6} {5} 7 d	r1
	7:[JT d - ?]		
	8:[J - - ?]		prox:9

Pilha sintática	entrada	Pilha de atributos	ação
`B or N B and N B`		`{3} {2,4} 5 {6} {5} 7 {7} {8}`	`r4`
	`6:[ J - - 7 ]`		
`B or N B`		`{3} {2,4} 5 {7} {5,8}`	`r4`
	`2:[ J - - 5 ]` `4:[ J - - 5 ]`		
`B`		`{3,7} {5,8}`	

As instruções 3 e 7 devem ser remendadas para o ponto α correspondente ao valor verdadeiro da expressão; idem 5 e 8, para o valor falso β. O código gerado para a expressão tem posições a serem completadas nas posições 3, 7, 5 e 8.

```
1: [ JT a - 3 ]
2: [ J  - - 5 ]
3: [ JT b - ? ]          { remendar para α }
4: [ J  - - 5 ]
5: [ JT c - ? ]          { remendar para β }
6: [ J  - - 7 ]
7: [ JT d - ? ]          { remendar para α }
8: [ J  - - ? ]          { remendar para β }
9:
```

A técnica de otimização de janela (*peephole*) também pode ser aplicada com vantagens a código como este.

5.8 - Geração de código de curto-circuito para outros comandos.

Vamos agora apresentar os esquemas de tradução correspondentes à geração de código de curto-circuito para os comandos `if`, `while` e `repeat`. Temos aqui duas possibilidades a considerar: a primeira consiste em fazer apenas as modificações necessárias para compatibilizar o código de curto-circuito das expressões booleanas com o código dos comandos, que continuaria a ser gerado da maneira vista anteriormente. Uma outra possibilidade mais interessante, que nos força a modificações mais extensas é a de usar listas de pontos de saída também para os comandos. Neste caso, vamos acrescentar um atributo `S.next`, que contém a lista de pontos (instruções de desvio para serem remendadas) em que a execução do comando se encerra.

Comando `if`. Temos, usando as mesmas convenções já vistas:

```
8. S⁰ → if B then N S¹
     remenda(B.true, N.quad);
     S⁰.next:=B.false ∪ S¹.next;
9. S⁰ → if B then N¹ S¹ else N² S²
     remenda(B.true,  N¹.quad);
     remenda(B.false, N².quad);
     S⁰.next:=S¹.next ∪ S².next;
10. S⁰ → while N¹ B do N² S¹
     remenda(S¹.next, N¹.quad);
     remenda(B.true, N².quad);
     S⁰.next:=B.false;
```

```
11. S⁰ → repeat N S¹ until N² B
      remenda(S¹.next, N².quad);
      remenda(B.false, N¹.quad);
      S⁰.next:=B.true;
```

Por exemplo, no caso do `if-then` temos :

- se a expressão `B` é verdadeira, o comando S^1 deve ser executado;
- a execução de S^0 se encerra quando a expressão `B` é falsa, ou quando se encerra a execução de S^1.

Como outro exemplo, no caso do `while` temos:

- se a expressão é verdadeira, o comando S^1 deve ser executado;
- se a execução do comando S^1 se encerra, a expressão `B` deve ser reavaliada;
- a execução de S^0 se encerra quando a expressão `B` é falsa.

Além desses comandos, devemos alterar as regras para os comandos de atribuição, e para a concatenação de comandos.

```
12. S⁰ → S¹ ; N S²
      remenda(S¹.next, N.quad);
      S⁰.next:=S².next;
13. S → V := E
      gera(:=, V.end, E.end, -);
      S.next:= { prox };
      gera(J, -, -, ?)
```

Note que dependemos de alguma otimizaçào futura para melhorar o código de uma série de atribuições. Por exemplo, se tivermos

```
v1:=w1;
v2:=w2;
v3:=w3
```

o código gerado será

```
1: [ := w1 v1 - ]
2: [ J  -  -  3 ]
3: [ := w2 v2 - ]
4: [ J  -  -  5 ]
5: [ := w3 v3 - ]
6: [ J  -  -  ? ]
```

com uma série de desvios inúteis: desvios para o comando seguinte.

Para o exemplo seguinte, o tratamento dos nãoterminais `V` e `E` é o mesmo das seções anteriores, com as seguintes regras (renumeradas)

```
14. E⁰ → E¹ + E²
      E⁰.end := temp();
      gera(+, E¹.end, E².end, E⁰.end);
15. E⁰ → E¹ * E²
      E⁰.end := temp();
      gera(*, E¹.end, E².end, E⁰.end);
16. E → ( E )
      ;
```

```
17. E → V
      ;
18. V → id
      V.end:=consulta(id.nome);
```

Exemplo 5. Considere a cadeia

```
repeat
  a:=c;
  if a or b then
    c:=a
  else
    c:=b
until a = b
```

Vamos apresentar as ações executadas durante a geração de código para o trecho de programa acima. Note que alguns símbolos estão abreviados.

pilha sintática	entrada	pilha de atributos	ação
	rp a:=c; if		s rp
rp	a:=c; if a		r7
rp N	a:=c; if a	1	s id
rp N id	:=c; if a o	1 a	r18
rp N V	:=c; if a o	1 a	s :=
rp N V :=	c; if a or	1 a	s id
rp N V := id	; if a or b	1 a c	r18
rp N V := V	; if a or b	1 a c	r17
rp N V := E	; if a or b	1 a c	r13
	1:[:= a c -] 2:[J - - ?]		prox:3
rp N S	; if a or b	1 {2}	s ;
rp N S ;	if a or b t	1 {2}	r7
rp N S ; N	if a or b t	1 {2} 3	s if
rp N S ; N if	a or b th c	1 {2} 3	s id
rp N S ; N if id	or b th c:=	1 {2} 3 a	r1
	3:[JT a - ?] 4:[J - - ?]		prox:5
rp N S ; N if B	or b th c:=	1 {2} 3 {3} {4}	s or
rp N S ; N if B or	b th c:= a	1 {2} 3 {3} {4}	r7
rp N S ; N if B or N	b th c:= a	1 {2} 3 {3} {4} 5	s id
rp N S ; N if B or N id	th c:=a el	1 {2} 3 {3} {4} 5 b	r1
	5:[JT b - ?] 6:[J - - ?]		prox:7
rp N S ; N if B or N B	th c:=a el	1 {2} 3 {3} {4} 5 {5} {6}	r3
	4:[J - - 5]		
rp N S ; N if B	th c:=a el	1 {2} 3 {3,5} {6}	s th
rp N S ; N if B	th c:=a el	1 {2} 3 {3,5} {6}	r7
rp N S ; N if B th N	c:=a el c:=	1 {2} 3 {3,5} {6} 7	s id
rp N S ; N if B th N id	:=a el c:=b	1 {2} 3 {3,5} {6} 7 c	r18

pilha sintática	entrada	pilha de atributos	ação
rp N S ; N if B th N V	:=a el c:=b	1 {2} 3 {3,5} {6} 7 c	s :=
rp N S ; N if B th N V :=	a el c:=b u	1 {2} 3 {3,5} {6} 7 c	s id
rp N S ; N if B th N V := id	el c:=b un	1 {2} 3 {3,5} {6} 7 c a	r18
rp N S ; N if B th N V := V	el c:=b un	1 {2} 3 {3,5} {6} 7 c a	r17
rp N S ; N if B th N V := E	el c:=b un	1 {2} 3 {3,5} {6} 7 c a	r13
	7:[:= c a -] 8:[J - - ?]		prox:9
rp N S ; N if B th N S	el c:=b un	1 {2} 3 {3,5} {6} 7 {8}	s el
rp N S ; N if B th N S el	c:=b un a=b	1 {2} 3 {3,5} {6} 7 {8}	r7
rp N S ; N if B th N S el N	c:=b un a=b	1 {2} 3 {3,5} {6} 7 {8} 9	s id
rp N S ; N if B th N S el N id	:=b un a=b	1 {2} 3 {3,5} {6} 7 {8} 9 c	r18
rp N S ; N if B th N S el N V	:=b un a=b	1 {2} 3 {3,5} {6} 7 {8} 9 c	s :=
rp N S ; N if B th N S el N V :=	b un a=b	1 {2} 3 {3,5} {6} 7 {8} 9 c	s id
rp N S ; N if B th N S el N V := id	un a=b	1 {2} 3 {3,5} {6} 7 {8} 9 c b	r18
rp N S ; N if B th N S el N V := V	un a=b	1 {2} 3 {3,5} {6} 7 {8} 9 c b	r17
rp N S ; N if B th N S el N V := E	un a=b	1 {2} 3 {3,5} {6} 7 {8} 9 c b	r13
	9:[:= c b -] 10:[J - - ?]		prox:11
rp N S ; N if B th N S el N S	un a=b	1 {2} 3 {3,5} {6} 7 {8} 9 {10}	r9
	3:[JT a - 7] 5:[JT b - 7] 6:[J - - 9]		
rp N S ; N S	un a=b	1 {2} 3 {8,10}	r12
	2:[J - - 3]		
rp N S	un a=b	1 {8,10}	s un
rp N S un	a=b	1 {8,10}	r7
rp N S un N	a=b	1 {8,10} 11	s id
rp N S un N id	=b	1 {8,10} 11 a	r18
rp N S un N V	=b	1 {8,10} 11 a	r17
rp N S un N E	=b	1 {8,10} 11 a	s =
rp N S un N E =	b	1 {8,10} 11 a	s id
rp N S un N E = id		1 {8,10} 11 a b	r18
rp N S un N E = V		1 {8,10} 11 a b	r17
rp N S un N E = E		1 {8,10} 11 a b	r2
	11:[J= a b ?] 12:[J - - ?]		prox:13

pilha sintática	entrada	pilha de atributos	ação
`rp N S un N B`		`1 {8,10} 11 {11} {12}`	`r2`

	`8:[ J - - 11 ]`
	`10:[ J - - 11 ]`
	`12:[ J - - 1 ]`		

`S`		`{11}`	`r2`

O código gerado é, portanto,

```
 1:[ := a c  - ]
 2:[ J  - -  3 ]
 3:[ JT a -  7 ]
 4:[ J  - -  5 ]
 5:[ JT b -  7 ]
 6:[ J  - -  9 ]
 7:[ := c a  - ]
 8:[ J  - - 11 ]
 9:[ := c b  - ]
10:[ J  - - 11 ]
11:[ J= a b  ? ]          { ← saída do repeat }
12:[ J  - -  1 ]
```

Como já observamos anteriormente, o código acima pode ser melhorado, através da otimização de janela (*peephole optimization*), que será o objeto da seção 5.11. No caso, o código obtido poderia ser

```
1:[ := a c - ]
2:[ JT a - 4 ]
3:[ JF b - 6 ]
4:[ := c a - ]
5:[ J  - - 7 ]
6:[ := c b - ]
7:[ J≠ a b 1 ]
8:[ J  - - ? ]          { ← saída do repeat }
```

A transformação das duas últimas instruções só faz sentido se, como seria provável, a instrução a executar depois do `repeat` fosse exatamente a instrução seguinte ao comando. Nesse caso, a última instrução seria dispensável.

Exercício 4: Construa esquemas de tradução capazes de gerar código de curto-circuito (ou fluxo de controle) para os comandos `if-elsif-end`, `case`, `loop-end` e `exit`.

5.9 - Geração de código para chamadas de procedimentos

Vamos mostrar aqui como gerar código de três endereços para chamadas de procedimentos em linguagens (tipo) Algol[3]. Para fixar as idéias, suponha que um procedimento `p` contém o seguinte comando de atribuição:

```
x:=f(a, g(b, h(c), d), e);
```

Neste comando, `a`, `b`, `c`, `d`, e `e` representam expressões. Se se tratar de uma linguagem em que as chamadas de função não podem ter efeitos colaterais, a ordem de execução das diversas chamadas é irrelevante, já que o único produto da avaliação é o resultado

[3] Embora C use um esquema diferente deste para chamadas de funções, em muitos compiladores este protocolo de passagem de parâmetros pode ser especificado pelo uso da palavra reservada `pascal`.

da função. Entretanto, a maioria das linguagens admite efeitos colaterais em procedimentos e funções, de forma que a ordem de avaliação é importante. A linguagem Fortran, por exemplo, especifica que o efeito do comando acima deve ser equivalente a

```
t1:=h(c);
t2:=g(b, t1, d);
t3:=f(a, t2, e);
x:=t3;
```

ou seja, que as chamadas de função devem ser avaliadas antes das demais expressões usadas como argumentos. Isso significa, por exemplo, que, se a função `h` altera o valor de `a` e de `b`, que esses valores alterados serão usados na avaliação das chamadas de `g` e de `f`. A ordem de avaliação, em Algol, é definida de forma recursiva: para avaliar uma chamada $f(x_1, x_2, \ldots x_n)$, avaliamos x_1, x_2,... x_n, nessa ordem, e depois a função `f` é chamada. Para o exemplo, isso é equivalente a:

```
1. v1:=a;
2. v2:=b;
3. v3:=c;
4. v4:=h(v3);
5. v5:=d;
6. v6:=g(v2, v4, v5);
7. v7:=e;
8. v8:=f(v1, v6, v7);
9. x:=v8;
```

Temos:

```
1-8: avaliação de f(a,g(b,h(c),d),e)
     1: avaliação de a
     2-6: avaliação de g(b,h(c),d)
          2: avaliação de b
          3-4: avaliação de h(c)
               3: avaliação de c
               4: chamada de h
          5: avaliação de d
          6: chamada de g
     7: avaliação de e
     8: chamada de f (8).
```

O modelo de implementação característico de Algol prevê, adicionalmente, que cada argumento, depois de avaliado, deve ser empilhado, numa posição que fará posteriormente parte do registro de ativação do procedimento chamado. Isso significa, por exemplo, que no passo `1` acima, os valores de `a` e `b` são empilhados, em posições que farão parte, posteriormente, dos registros de ativação de `f` e de `g`. Coerentemente, o resultado de uma função é deixado na pilha, após a execução da função. No nosso exemplo, teríamos configurações como as da Fig. 5(a, b, c). Após a chamada de `f` o valor final `v8` deve ser copiado em `x`, liberando sua posição no topo da pilha. Note que, neste esquema, só é necessário gerar código para os argumentos que não são chamadas de função.

Dependendo da forma de passagem de parâmetros, o tratamento de um parâmetro pode variar: no caso de *passagem por valor*, o argumento é avaliado e seu valor é empilhado; no caso de *passagem por referência*, o argumento deve ser uma variável, e

o endereço da variável é empilhado. No caso de *passagem por nome*, cada argumento é associado a um trecho de código (*thunk*), que corresponde a uma chamada de função sem parâmetros, que devolve, a cada chamada, o endereço corrente do argumento; nesse caso o endereço do *thunk* deve ser empilhado.

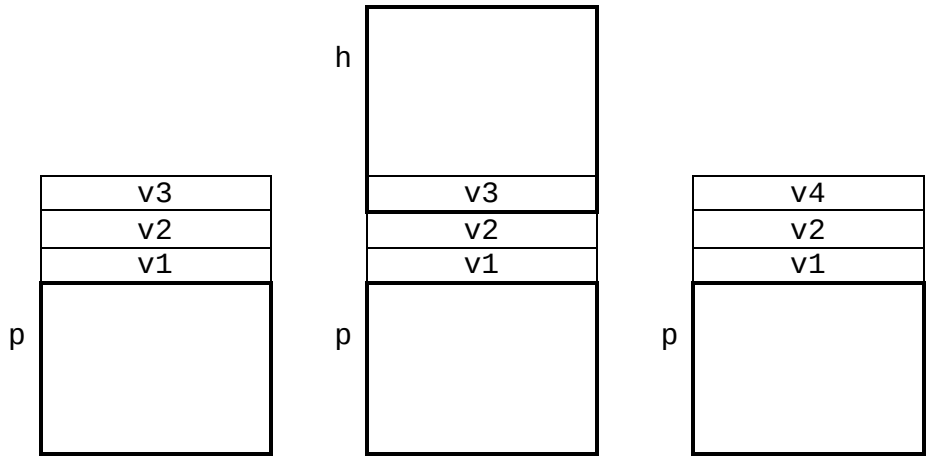


Fig 5(a) Chamada de h

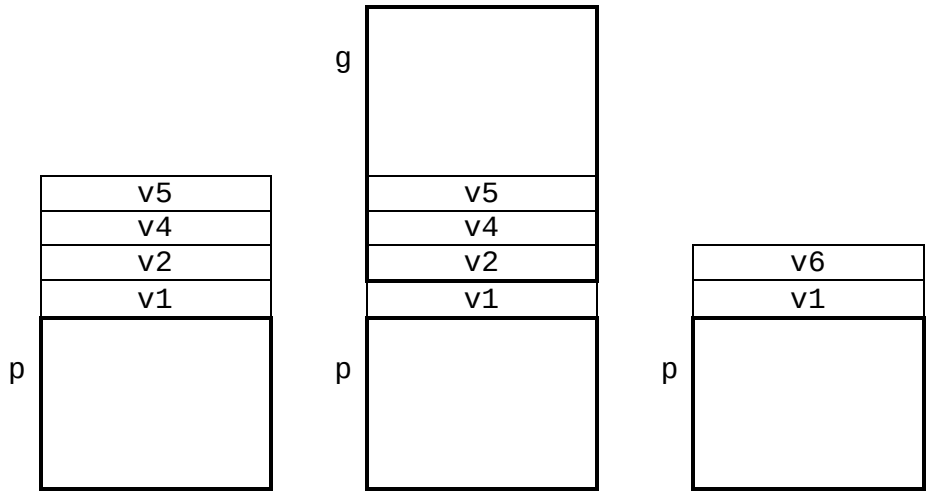


Fig 5(b) Chamada de g

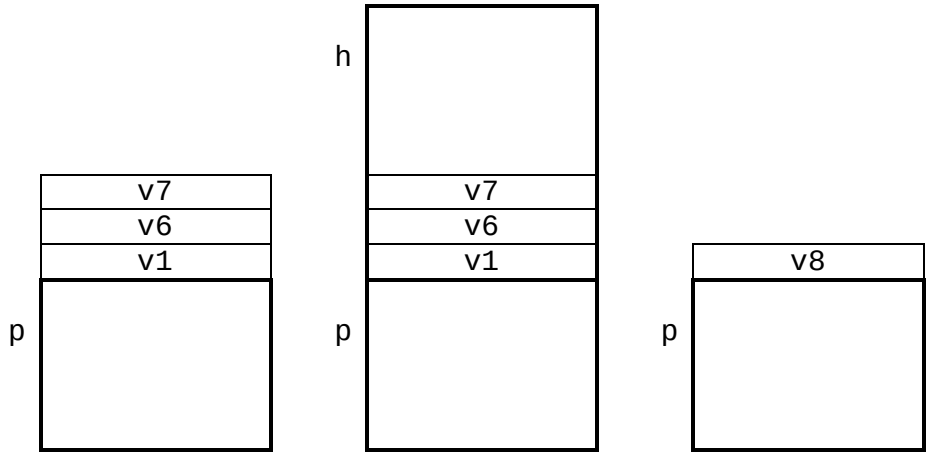


Fig 5(c) Chamada de f

A passagem de *parâmetros por resultado* e *por valor resultado* pode ser implementada de forma semelhante, exceto que os resultados são deixados na pilha pelo procedimento chamado, para tratamento pelo chamador. Entretanto, é comum a implementação de passagem por valor-resultado de forma idêntica à de passagem por referência.

A linguagem Ada prevê três formas de passagem de parâmetros: `in`, que corresponde a valores passados para o procedimento, que não podem ser alterados; `out`, que correspondem a valores gerados pelo procedimento, e fornecidos como resultados; e `in out`, que corresponde a valores recebidos e alterados pelo procedimento. Entretanto, Ada não prevê uma forma de implementação fixa para essas formas de passagem de parâmetros, o que significa que parâmetros `in out` podem ser passados por valor-resultado ou por referência. Outro ponto sobre o qual nada é dito é sobre a ordem de avaliação dos parâmetros, que pode ser qualquer.

Programas em Ada cujo funcionamento depende da ordem de avaliação dos parâmetros ou da forma de implementação da passagem de parâmetros são considerados *errôneos*. Isto significa que são programas com erro, mas que um compilador de Ada não precisa indicar este erro. O melhor que se pode dizer a respeito é que programas cujo bom funcionamento depende destes aspectos de implementação são programas inseguros, cuja construção deve ser evitada.

Para a geração de código de 3 endereços, podemos usar uma instrução `PARAM`, que corresponde ao empilhamento do valor ou do endereço do parâmetro, como apropriado. Se uma linguagem permite mais de uma forma de passagem de parâmetros, podemos ter várias instruções diferentes, para os diversos casos. No caso do exemplo visto acima,

```
x:=f(a, g(b, h(c), d), e);
```

as instruções geradas seriam

```
[ PARAM a - - ]
[ PARAM b - - ]
[ PARAM c - - ]
[ CALL  h - - ]
[ PARAM d - - ]
[ CALL  g - - ]
[ PARAM e - - ]
[ CALL  f - - ]
[ POP   x - - ]
```

Uma possibilidade alternativa é a de colocar as variáveis temporárias no topo da pilha, em todos os casos, e não apenas no caso de resultados de funções. Neste caso, poderíamos tratar `a+b` como uma chamada da função `+`, que espera encontrar seus argumentos no topo da pilha, e que devolve seu resultado no mesmo lugar, como uma função qualquer. Neste caso, para o comando

```
x:= a*b+c*d
```

teríamos o seguinte código gerado:

```
[ PARAM a - - ]
[ PARAM b - - ]
[ CALL  * - - ]
[ PARAM c - - ]
[ PARAM d - - ]
[ CALL  * - - ]
```

```
[ CALL  + - - ]
[ POP   x - - ]
```

em vez do mais familiar

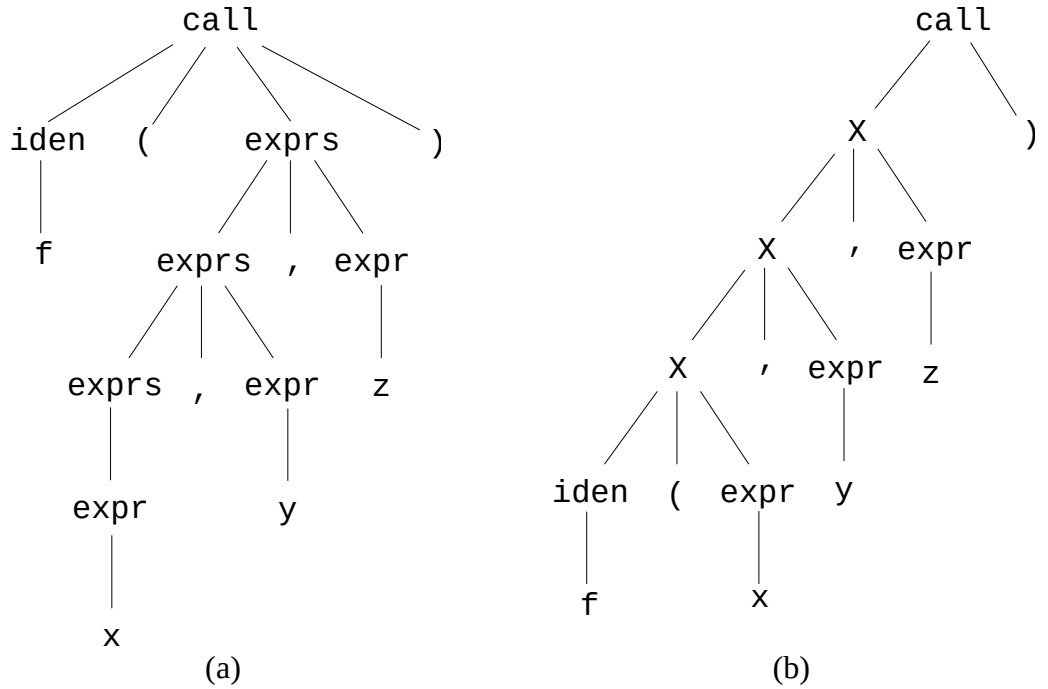
```
[ *  a  b  t1 ]
[ *  c  d  t2 ]
[ +  t1 t2 t3 ]
[ := x  t3 -  ]
```

Claramente, a primeira possibilidade se aplica melhor à geração de código para máquinas de pilha, em que as instruções de hardware correspondentes a `+` e `*` procuram na pilha de hardware seus argumentos, e deixam na pilha os seus resultados.

Um problema adicional a ser tratado é puramente sintático. Espera-se para chamada de função ou de procedimento algumas regras como

```
call → iden ( exprs )
exprs → exprs , expr | expr
```

Entretanto, de acordo com essas regras, uma chamada `f(x,y,z)` corresponde à árvore da Fig. 6(a). Nessa sintaxe, a menos que atributos herdados sejam usados, não existe possibilidade de tratamento das expressões `x`, `y`, `z` como argumentos de uma chamada de `f` até que a regra `call→iden(exprs)` seja atingida. Ou seja, toda a informação sobre os argumentos deve ser reunida em um atributo de `exprs` que só será tratado na redução mencionada, que deverá se encarregar da verificação de que os parâmetros `x`, `y`, `z` são adequados para `f`, e da geração das instruções `PARAM` correspondentes a `x`, `y`, `z`.



(a) (b)

Fig 6 – duas sintaxes para chamada de procedimentos

Outra sintaxe possível, se bem que menos intuitiva, resolve esse problema. Temos

```
call → X )
X → X , expr
  | iden ( expr
```

Neste caso, a árvore correspondente a `f(x,y,z)` é a da Fig. 6(b). Atributos de `X` devem conter a informação sobre a função correspondente a `iden` (`f`, no exemplo), e o número de argumentos que já foram encontrados. Assim, a redução por `X→iden(expr` identifica a função `f`, verifica que `x` é um primeiro parâmetro aceitável, e gera a instrução `PARAM x`. As duas reduções por `X→X,expr` verificam que `y` e `z` são parâmetros aceitáveis (segundo e terceiro) para `f`, e geram as instruções `PARAM y` e `PARAM z`. A redução pela regra `call→X)` verifica que a chamada não tem parâmetros além do número necessário, e gera a instrução `CALL f`.

5.10 - Geração de código para referências a arrays.

Em uma linguagem como Pascal, a definição de tipos array é recursiva. Por exemplo, em vez de definir explicitamente uma matriz, Pascal define recursivamente um tipo "*array of array*". Assim, em Pascal, as declarações

```
type T = array[T1] of array[T2] of array[T3] of T4;
type T = array[T1,T2,T3] of T4;
```

são equivalentes. De forma semelhante, se `a` é uma variável do tipo `T`, as referências `a[i][j][k]` e `a[i,j,k]` são equivalentes. No caso, as informações relevantes a serem guardadas na tabela de símbolos (assunto que deixamos para um capítulo posterior) são:

- `a` é uma variável do tipo `T`;
- `T` é um tipo array com índices do tipo `T1` e componentes do tipo `X1`;
- `X1` é um tipo array com índices do tipo `T2` e componentes do tipo `X2`;
- `X2` é um tipo array com índices do tipo `T3` e componentes do tipo `T4`;

Note que os pontos de entrada da tabela de símbolos que nos interessam são `a` e `T`, além, naturalmente, de `T1`, `T2`, `T3` e `T4`, que devem ter sido declarados anteriormente. `X1` e `X2` são tipos sem nome, cujas entradas na tabela de símbolos só podem ser alcançadas a partir de `T`. A verificação da correção da referência de `a[i][j][k]` deve ser feita também de forma recursiva:

- o tipo de `a` é identificado: `T`;
- a partir da definição de `T`, verifica-se que o tipo de `i` é `T1`, e conclui-se que `a[i]` é do tipo `X1`;
- a partir da definição de `X1`, verifica-se que o tipo de `j` é `T2`, e conclui-se que `a[i][j]` é do tipo `X2`;
- a partir da definição de `X2`, verifica-se que o tipo de `k` é `T3`, e conclui-se que `a[i][j][k]` é do tipo `T4`;

A sintaxe mais natural para referências a arrays, é

```
var → var [ exprs ] | iden
exprs → exprs , expr | expr
```

que cria um problema na análise de referências como `a[i,j,k]`, semelhante ao problema visto no tratamento de chamadas de procedimentos. Podemos utilizar a sintaxe abaixo, que, entretanto, é menos natural que a original.

```
var → Y ] | iden
Y → var [ expr | Y , expr
```

A geração de código para uma referência a array consiste em geral no cálculo do endereço de uma componente como `a[i,j,k]`. Temos

```
end( a[i,j,k] ) = a0 + k1*(i-i0) + k2*(j-j0) + k3*(k-k0)
```

onde

as constantes `k1`, `k2`, e `k3` são, respectivamente, os tamanhos (número de posições de memória ocupadas) dos tipos `X1`, `X2` e `T4`;

a constante `a0` é o endereço inicial da área alocada para a variável `a`;

`i0`, `j0` e `k0` são, respectivamente, os valores iniciais dos tipos `T1`, `T2` e `T3`.

Portanto, o código para avaliação do endereço pode ser gerado de forma recursiva, já que sua definição também pode ser feita de forma recursiva:

```
end( a ) = a0
end( a[i] ) = end( a ) + k1*(i-i0)
end( a[i,j] ) = end( a[i] ) + k2*(j-j0)
end( a[i,j,k] ) = end( a[i,j] ) + k3*(k-k0)
```

Os passos indicados correspondem às reduções pelas regras `Y→var[expr`, que trata a variável a e o primeiro índice `i`, e pelas duas reduções pela regra `Y→Y,expr`, que tratam o segundo e o terceiro índices (`j` e `k`). Naturalmente, a discussão se aplica ao caso geral, e não apenas ao exemplo apresentado.

5.11 - Otimização de janela.

Vamos ver nesta seção os princípios da otimização de janela (peephole optimization). Este tipo de otimização pode ser aplicado a código intermediário, como foi feito em alguns exemplos apresentados anteriormente, mas pode também ser aplicado a código executável de máquinas reais. A idéia por trás desse tipo de otimização é extremamente simples: em vez de examinar uma fração importante do código para procurar oportunidades de otimização, apenas algumas poucas instruções contíguas (as instruções visíveis através de uma "janela") são examinadas, em busca de padrões que caracterizem uma possibilidade de melhora do código. Exemplos desses padrões são:

- instruções de desvio para a instrução seguinte;
- instruções de desvio para uma instrução de desvio incondicional;
- (quando existem instruções de desvio de vários tamanhos) instruções de desvio longo para uma posição alcançável por uma instrução de desvio curto;
- combinações de desvio condicional e desvio incondicional que podem ser substituídas por uma instrução única de desvio condicional para a condição oposta (por exemplo, `JT/J` pode ser substituída por em alguns casos por `JF`);
- alguns casos de código morto (código não alcançável), por exemplo causados por um teste impossível de ser satisfeito;
- expressões constantes que podem ser avaliadas em tempo de compilação (por exemplo, `2*PI` pode ser substituída por `6.28`);

- substituição de instruções por equivalentes mais simples, em casos particulares - por exemplo, instruções de soma e subtração de 1 podem ser substituídas por instruções de incremento e decremento; produto/divisão por potências de 2 podem ser substituídos por instruções de deslocamento (shift);
- expressões com simplificações triviais: multiplicação por 0 ou 1, divisão por 1, soma ou subtração com 0, conjunção (and) com false, disjunção (or) com true, dupla negação, dupla troca de sinal, etc.

Em princípio, pelo menos, podemos evitar todas estas situações, escrevendo o gerador de código com mais cuidado; entretanto, pode ser mais barato deixar para uma fase de otimização a "limpeza" do código.

A implementação da otimização requer, em geral, a renumeração das instruções, devido à possibilidade de eliminação de algumas instruções, ou, no caso de código de máquina real, da substituição de instruções por outras mais curtas. Em alguns casos, mais de uma passada pelo código pode ser conveniente, porque uma passada pode criar novas oportunidades de otimização para a seguinte. Por exemplo, a substituição de uma instrução por outra mais curta, ou a eliminação de uma instrução inútil pode diminuir a distância entre uma instrução de desvio e a instrução alvo, permitindo que mais um desvio longo seja substituído por um desvio curto.

Exemplo 6. Considere a seqüência de instruções gerada no Exemplo 5. Para ter uma seqüência bem definida de instruções, vamos completar a instrução `11` colocando como alvo do desvio o comando `13` (o comando que segue o comando `repeat` original).

```
 1:[ := a c  - ]
 2:[ J  - -  3 ]
 3:[ JT a -  7 ]
 4:[ J  - -  5 ]
 5:[ JT b -  7 ]
 6:[ J  - -  9 ]
 7:[ := c a  - ]
 8:[ J  - - 11 ]
 9:[ := c b  - ]
10:[ J  - - 11 ]
11:[ J= a b 13 ]
12:[ J  - -  1 ]
```

Podemos reconhecer os seguintes padrões:

- as instruções `2`, `4`, `10` são dispensáveis (desvios para a instrução seguinte), e podem ser removidas;
- invertendo-se o teste `JT` da instrução `5` temos, em lugar de `5` e `6`,

```
5′:[ JF b -  9 ]
6′:[ J  - -  7 ]
```

- portanto, podemos eliminar a instrução `6′`.
- o mesmo acontece com o teste `J=` da instrução `11`. Em lugar de `11` e `12`, podemos ter

```
11′:[ J≠ a b  1 ]
12′:[ J  - - 13 ]
```

- portanto, podemos eliminar a instrução `12′`.

O resultado dessas transformações é

```
  1:[ := a c  - ]
  3:[ JT a -  7 ]
 5′:[ JF b -  9 ]
  7:[ := c a  - ]
  8:[ J  - - 11 ]
  9:[ := c b  - ]
11′:[ J≠ a b  1 ]
```

Renumerando as instruções, temos como resultado

```
1:[ := a c  - ]
2:[ JT a -  4 ]
3:[ JF b -  6 ]
4:[ := c a  - ]
5:[ J  - -  7 ]
6:[ := c b  - ]
7:[ J≠ a b  1 ]
```

com uma redução de 5 instruções em um total de 12, ou seja, de mais de 40%. Mesmo que esse número não seja atingido, os resultados da otimização de janela costumam ser significativos.

(maio, 1999)