

Geração de código intermediário:
Geração de código de fluxo de controle
Backpatching

Geração de código intermediário

- Em código que manipula o fluxo de controle de um programa (ifs, while, for, do..while, etc.), operadores booleanos &&, || e ! são traduzidos em comandos de desvios
- A idéia é permitir ações semânticas junto com a análise sintática
- As construções permitem gerar código de três endereços

Geração de código intermediário

- Exemplo:
 - if (x < 100 || x > 200 && x != y) x = 0;
 - É gerado como:
 - If x < 100 goto L2
 - goto L3
 - L3: if x > 200 goto L4
 - goto L1
 - If x != y goto L2
 - goto L1
 - x = 0;

Geração de código intermediário

- Nas regras a seguir, considere:
 - newlabel(): retorna um novo label
 - next, true e false: são atributos herdados dos não-terminais
 - code: é um atributo sintetizado
 - label(L): retorna o nome do label armazenado por L
 - gen('string'): retorna o valor string
 - Operador de concatenação ||: “une” expressões

Geração de código intermediário

Produção	Regra Semântica
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \parallel label(S.next)$
$S \rightarrow assign$	$S.code = assign.code$
$S \rightarrow if (B) S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \parallel label(B.true) \parallel S_1.code$
$S \rightarrow if (B) S_1 else S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code \parallel label(B.true) \parallel S_1.code \parallel$ $gen('goto' S.next) \parallel label(B.false) \parallel S2.code$

Geração de código intermediário

Produção	Regra Semântica
$S \rightarrow \text{while } (B) S_1$	<pre>begin=newlabel() B.true=newlabel() B.false=S.next S₁.next=begin S.code=label(begin) B.code label(B.true) S₁.code gen('goto' begin)</pre>
$S \rightarrow \text{do } S_1 \text{ while } (B)$	<pre>begin=newlabel() B.true=begin B.false=S.next S₁.next=newlabel() S.code=label(begin) S₁.code label(S₁.next) B.code</pre>

Geração de código intermediário

Produção

Regra Semântica

$B \rightarrow B_1 \parallel B_2$

$B_1.\text{true} = B.\text{true}$

$B_1.\text{false} = \text{newlabel}()$

$B_2.\text{true} = B.\text{true}$

$B_2.\text{false} = B.\text{false}$

$B.\text{code} = B_1.\text{code} \parallel \text{label}(B_1.\text{false}) \parallel B_2.\text{code}$

$B \rightarrow B_1 \ \&\& \ B_2$

$B_1.\text{true} = \text{newlabel}()$

$B_1.\text{false} = B.\text{false}$

$B_2.\text{true} = B.\text{true}$

$B_2.\text{false} = B.\text{false}$

$B.\text{code} = B_1.\text{code} \parallel \text{label}(B_1.\text{true}) \parallel B_2.\text{code}$

Geração de código intermediário

Produção	Regra Semântica
$B \rightarrow !B_1$	$B_1.\text{true} = B.\text{false}$ $B_1.\text{false} = B.\text{true}$ $B.\text{code} = B_1.\text{code}$
$B \rightarrow E_1 \text{ rel } E_2$	$B.\text{code} = E_1.\text{code} \parallel E_2.\text{code} \parallel$ $\text{gen}(\text{'if' } E_1.\text{addr rel.op } E_2.\text{addr 'goto' } B.\text{true})$ $\parallel \text{gen}(\text{'goto' } B.\text{false})$

Backpatching

- Na tradução de comandos de fluxo de controle, a abordagem que usa atributos herdados faz com que o processo de tradução seja, necessariamente, em duas etapas
- Em expressões do tipo if (B) S, as regras anteriores geram o código de B antes de gerar S, qual será, então o alvo do goto para o caso de B ser falso?
 - A solução anterior usa atributos herdados que capturam labels onde os gotos devem apontar

Backpatching

- A motivação para usar backpatching é possibilitar que a geração de código para comandos de fluxo de controle seja feita em um único passo
- Para tanto, faz-se uso das definições a seguir. Considere a expressão **if (B) S**:
 - Listas de gotos são passadas como atributos sintetizados
 - Quando um goto é gerado, o alvo do goto não é especificado imediatamente
 - Cada goto é colocado em uma lista de gotos cujos labels devem ser iguais, no momento em que for definido

Backpatching

- Existe agora dois atributos sintetizados: `truelist` e `falselist` de um símbolo não-terminal `B`, que são usados para manter os labels dos gotos
- Existe um atributo sintetizado `nextlist` denotando uma lista de gotos para a instrução após o código de `S`
- Marcadores `M` e `N` são usados para “guardar” uma posição dentro do código que está sendo gerado
- A variável `nextinstr` é um ponteiro para o endereço da instrução atual

Backpatching

- A função `makelist(i)` cria uma nova lista de gotos contendo apenas a instrução na linha `i`
- `merge(p1, p2)` concatena as listas apontadas por `p1` e `p2`
- `backpatch(p,i)` insere `i` como o label alvo das instruções contidas na lista `p`

Backpatching

Produção

Regra Semântica

$B \rightarrow B_1 \parallel M B_2$

`backpatch(B1.falselist, M.inst)`

`B.truelist=merge(B1.truelist, B2.truelist)`

`B.falselist=B2.falselist`

$B \rightarrow B_1 \ \&\& \ M \ B_2$

`backpatch(B1.truelist, M.inst)`

`B.truelist=B2.truelist`

`B.falselist=merge(B1.falselist, B2.falselist)`

$B \rightarrow E_1 \ \text{rel} \ E_2$

`B.truelist=makelist(nextinstr)`

`B.falselist=makelist(++nextinstr)`

`B.code= gen('if' E1.addr rel.op E2.addr 'goto _') || gen('goto _')`

Backpatching

Produção	Regra Semântica
$S \rightarrow \text{if (B) } M S_1$	$\text{backpatch}(B.\text{truelist}, M.\text{inst})$ $S.\text{nextlist} = \text{merge}(B.\text{falselist}, S_1.\text{nextlist})$
$S \rightarrow \text{if (B) } M_1 S_1 N_1$ $\text{else } M_2 S_2$	$\text{backpatch}(B.\text{truelist}, M_1.\text{inst})$ $\text{backpatch}(B.\text{falselist}, M_2.\text{inst})$ $\text{temp} = \text{merge}(S_1.\text{nextlist}, N_1.\text{nextlist})$ $S.\text{nextlist} = \text{merge}(\text{temp}, S_2.\text{nextlist})$
$M \rightarrow \epsilon$	$M.\text{inst} = ++\text{nextinstr}$
$N \rightarrow \epsilon$	$N.\text{nextlist} = \text{makelist}(++\text{nextinstr})$ $\text{write}(\text{'goto _'})$
$S \rightarrow \text{assign}$	$S.\text{nextlist} = \text{NULL}$ $S.\text{code} = \text{assign.code}$ $\text{gen}(S.\text{code})$

Backpatching

- Como seria a geração de código, baseado em backpatching, para a expressão a seguir?
- `If (x<100 || x> 200 && x!=y) x=0;`