

Compiladores

Geração de código intermediário (1)

Lembrando – análise semântica

- Parser Bottom-up:
 - Esquema S-atribuído sem problema
 - Apenas atributos sintetizados
 - Esquema L-atribuído: ok, mas deve-se usar variáveis globais para os atributos do pai.
 - Atributos sintetizados + herdados apenas do pai ou dos irmãos à esquerda.
- Parser Top-Down:
 - Esquema L-atribuído: sem problema
 - Avaliação de esquerda para direita.
 - Esquema S-atribuído: sem problema também...
 - É um sub-conjunto do caso L.

Implementação de esquemas de tradução L-atribuídos

```
Procedure dfvisit (n : node);  
Begin  
  Para cada filho  $m_i$  de n, da esquerda para direita, faça  
    Avalie os atributos herdados de  $m_i$   
    Dfvisit( $m_i$ )  
  Avalie os atributos sintetizados de n  
End
```

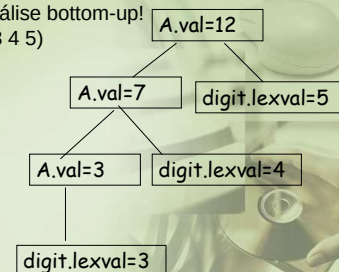
Exemplo de implementação top-down (1)

$A \rightarrow A_1 \text{ digit} \quad \{A.val := A_1.val + \text{digit.lexval}\}$

$A \rightarrow \text{digit} \quad \{A.val := \text{digit.lexval}\}$

(Obs: Possibilita uma análise bottom-up!)

Exemplo: derivação de 3 4 5)



Exemplo de implementação top-down (2)

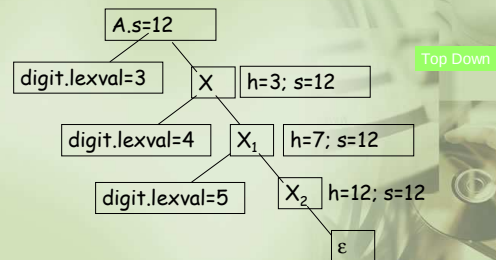
$A \rightarrow A_1 \text{ digit} \quad \{A.val := A_1.val + \text{digit.lexval}\}$
 $A \rightarrow \text{digit} \quad \{A.val := \text{digit.lexval}\}$

- Transformação em esquema L-atribuído
- ELIMINANDO A RECURSIVIDADE ...
 - Cria-se dois tipos de atributos (s e h)

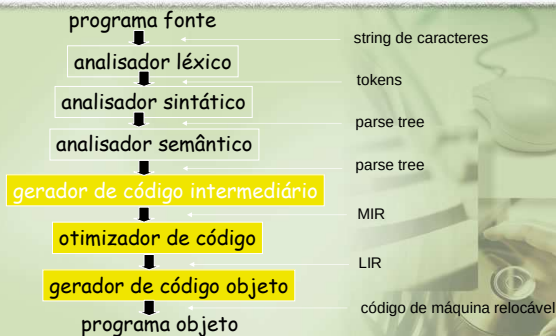
$A \rightarrow \text{digit} \quad \{X.h := \text{digit.lexval}\} \quad X \quad \{A.s := X.s\}$
 $X \rightarrow \text{digit} \quad \{X_1.h := X.h + \text{digit.lexval}\} \quad X_1 \quad \{X.s := X_1.s\}$
 $X_1 \quad \{X.s := X_1.s\}$
 $X \rightarrow \epsilon \quad \{X.s := X.h\}$

Árvore de derivação para 3 4 5

$A \rightarrow \text{digit} \quad \{X.h := \text{digit.lexval}\} \quad X \quad \{A.s := X.s\}$
 $X \rightarrow \text{digit} \quad \{X_1.h := X.h + \text{digit.lexval}\} \quad X_1 \quad \{X.s := X_1.s\}$
 $X \rightarrow \epsilon \quad \{X.s := X.h\}$



Modelo Clássico



Geração de código intermediário

- Vantagens em se fazer transformações em mais de um passo:

– Otimização:

- Possibilita a melhoria do código intermediário para prover um código otimizado mais eficiente

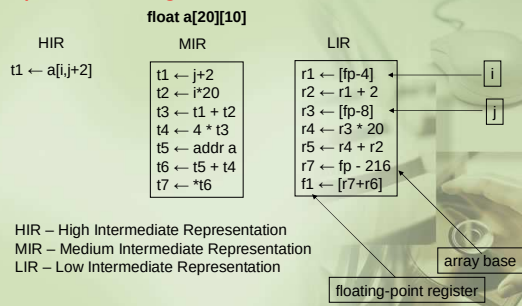
– Simplifica a implementação do compilador, resolvendo gradativamente as dificuldades de passagem de código fonte para código intermediário

– Portabilidade:

- possibilita a tradução de código intermediário para **diversas máquinas**

Linguagens Intermediárias

Tipos de código intermediário



Linguagens Intermediárias

- **HIR:**
 - Usada nos primeiros estágios do compilador
 - Simplificação de construções gramaticais para somente o essencial para otimização/geração de código
- **MIR:**
 - Boa base para geração de código eficiente
 - Pode expressar todas características de linguagens de programação de forma independente da linguagem
 - Representação de variáveis, temporários, registradores
- **LIR:**
 - Quase 1-1 para linguagem de máquina
 - Dependente da arquitetura

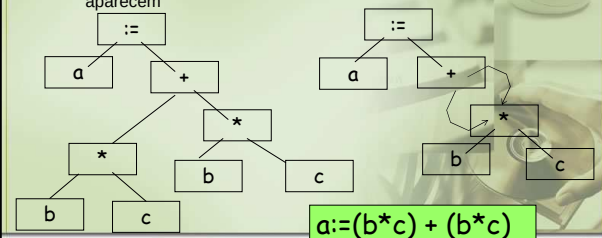
Linguagens Intermediárias

Tipos de código intermediário

- HIR e MIR:
 - Árvore e grafo de sintaxe
 - Notações Pós-fixada e Pré-fixada:
 - Representações linearizadas
- MIR:
 - Código de três endereços:
 - quádruplas
 - triplas
- LIR:
 - Instruções assembly

HIR: Árvore e grafo de sintaxe

- **Árvore de sintaxe**
 - Forma condensada da árvore de derivação
 - As cadeia de produções simples ($A \rightarrow B$, $B \rightarrow C$) não aparecem
- **Grafo de sintaxe**
 - Inclui simplificações da árvore de sintaxe
 - Faz a fatoração de sub-expressões comuns



HIR: Notações pós e pré-fixadas

Infixada	Pós-fixada	Pré-fixada
$(a+b)*c$	$ab+c^*$	$*+abc$
$a*(b+c)$	$abc+^*$	$*a+bc$
$a+b*c$	abc^*+	$+a*bc$

Geração de código fácil com uma pilha!

MIR: Código de três endereços (TAC)

- Cada instrução referencia, no máximo, três variáveis (endereços de memória)
 - Necessita variáveis temporárias!
 - As instruções possíveis são as seguintes
 - $A := B \text{ op } C$
 - $A := \text{op } B$
 - $A := B$
 - goto L
 - If A op_rel B goto L
- A, B, C: nomes, constantes ou tmp
 op: qualquer operador (só um !):
- operação binária
 - $:= \text{op}$ (onde op é unário, ou vazio)
- Pode-se usar também indireção: $x[i]$
 Pode-se usar também ponteiro (*) ou endereço (&)

Implementação de enunciados TAC

- O compilador deverá gerar uma seqüência de enunciados TAC para traduzir cada conjunto de instruções analisadas.
- Um enunciado ' $x := y \text{ op } z$ ' será implementado como:
 - Uma quádrupla:
 - Struct quad { int cod_op; void* arg1; void* arg2; void* res; }
 - arg1 == y, arg2 == z, res == x.
 - Questão: para que vão apontar os ponteiros arg1, arg2 e res? E qual é o problema?
 - Uma tripla:
 - Um temporário criado pelo n-ésimo enunciado será substituído por n.
 - Struct tripla { int cod_op; void* arg1; void* arg2; }

Exemplo: seqüências de TAC para $A := B*(-C+D)$

Quádruplas

	oper	arg1	arg2	result
(0)	-u	C		T1
(1)	+	T1	D	T2
(2)	*	B	T2	T3
(3)	:=	T3		A

Triplas

	oper	arg1	arg2
(0)	-u	C	
(1)	+	(0)	D
(2)	*	B	(1)
(3)	:=	A	(2)

Geração de TAC para as principais construções

- Geração de código de três endereços:
 - Expressões
 - Declarações (escopo simples)
 - Declarações (escopos aninhados)
 - Comandos de atribuição
 - Arrays e Registros
 - Expressões booleanas
 - Comandos de decisão
 - Comandos de iteração

Geração de TAC para as principais construções

- Geração de código de três endereços:
 - Expressões
 - Declarações (escopo simples)
 - Declarações (escopos aninhados)
 - Comandos de atribuição
 - Arrays e Registros
 - Expressões booleanas
 - Comandos de decisão
 - Comandos de iteração

tradução em código TAC de expressões

$S \rightarrow id := E$
 $E \rightarrow E_1 + E_2$
 $E \rightarrow E_1 * E_2$
 $E \rightarrow (E_1)$
 $E \rightarrow id$

Exemplo:
 $A := X + Y * Z$
 $T1 := Y * Z$
 $T2 := X + T1$
 $A := T2$

Usa-se dois mecanismos:

1. Atributos sintetizados:

- O atributo 'local' armazena o nome da variável
- O atributo 'codigo' armazena o código TAC sintetizado

2. Funções auxiliares:

- A função "geracod" escreve (na tela) código
- A função "novo_tmp" retorna um nome de temporário tmp1, tmp2...

Expressões (cont.) (" || " significa concatenação)

$S \rightarrow id := E$ $\{S.codigo = E.codigo || \text{geracod}(id.local := E.local)\}$
 $E \rightarrow E_1 + E_2$ $\{E.local = \text{novo_tmp}(); E.codigo = E_1.codigo || E_2.codigo || \text{geracod}(E.local := E_1.local "+" E_2.local); \}$
 $E \rightarrow E_1 * E_2$ $\{E.local = \text{novo_tmp}(); E.codigo = E_1.codigo || E_2.codigo || \text{geracod}(E.local := E_1.local "*" E_2.local); \}$
 $E \rightarrow (E_1)$ $\{E.local = E_1.local; E.codigo = E_1.codigo \}$
 $E \rightarrow id$ $\{E.local = id.local; E.codigo = " " \}$

Árvore de derivação anotada

$A := X + Y * Z$

