

– INF01147 –
Compiladores

Geração de Código Intermediário
Introdução, Taxonomia, Expressões

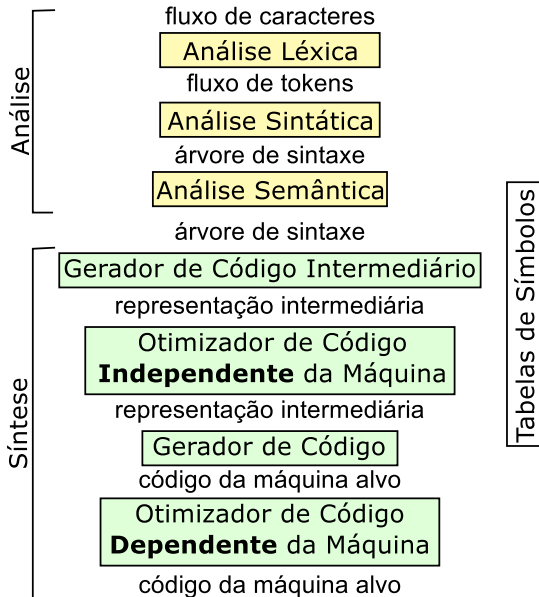
Prof. Lucas M. Schnorr
– Universidade Federal do Rio Grande do Sul –



Plano da Aula de Hoje

- ▶ Contextualização e Introdução
- ▶ Taxonomia
- ▶ Categorias
- ▶ Geração de IR

Estrutura de um **Compilador** em fases



Transformações em Passos – Vantagens

- ▶ Passos – construção de conhecimento gradativa
- ▶ Otimização
- ▶ Simplicidade
- ▶ Portabilidade

Representação Intermediária (IR)

- ▶ Derivar informações sobre o código a ser compilado
- ▶ Essas informações precisam ser mantidas na compilação
- ▶ **Representação Intermediária (IR)**
 - ▶ Necessária para se manter conhecimento – **expressividade**
 - ▶ Presente em praticamente todos os compiladores
- ▶ Como escolher uma IR?
 - ▶ Linguagem fonte (C *versus* Perl e ponteiros)
 - ▶ Máquina alvo (IR parecida com Assembly)
 - ▶ Aplicações (Objective-C *versus* C e hierarquia de classes)

Taxonomia

IR – Taxonomia

- ▶ Existem diversos tipos de IR
- ▶ Eixos conceituais principais
 - ▶ Organização estrutural
 - ▶ Nível de abstração
 - ▶ Espaço de nomes
- ▶ Três categorias de IR
 - ▶ Gráficas
 - ▶ Lineares
 - ▶ Híbridas

IR – Taxonomia – Organização Estrutural

- ▶ Influencia todos os processos
 - ▶ Análise
 - ▶ Otimização
 - ▶ Geração
- ▶ IR em formato de árvore
 - ▶ Passagens serão na forma de percurso na árvore
- ▶ IR linear, textual
 - ▶ Passagens seguirão a ordem linear

IR – Taxonomia – Nível de abstração

- ▶ **IR Alta** – High IR (HIR)
 - ▶ Usada nos primeiros estágios do compilador
 - ▶ Simplificação de construções gramaticais para guardar somente as informações necessárias para geração e otimização de código
- ▶ **IR Média** – Medium IR (MIR)
 - ▶ Boa base para a geração de código eficiente
 - ▶ Pode expressar todas as características de linguagens de programação de forma independente da linguagem
 - ▶ Representação de variáveis, temporários, registradores
- ▶ **IR Baixa** – Low IR (LIR)
 - ▶ Quase um para um com linguagem de máquina
 - ▶ Dependente da arquitetura

IR – Taxonomia – Nível de Abstração

- Supondo que temos uma construção **float a[20][10]**

HIR	MIR	LIR
$t1 \leftarrow a[i,j+2]$	$t1 \leftarrow j+2$ $t2 \leftarrow i*20$ $t3 \leftarrow t1 + t2$ $t4 \leftarrow 4 * t3$ $t5 \leftarrow \text{addr } a$ $t6 \leftarrow t5 + t4$ $t7 \leftarrow *t6$	$r1 \leftarrow [fp - 4] \quad i$ $r2 \leftarrow r1 + 2$ $r3 \leftarrow [fp - 8] \quad j$ $r4 \leftarrow r3 * 20$ $r5 \leftarrow r4 + r2 \quad \text{base}$ $r7 \leftarrow fp - 216 \quad \text{fp register}$ $f1 \leftarrow [r7 + r6]$

IR – Taxonomia – Espaço de Nomes

- ▶ Nomes para representar valores no código

- ▶ Exemplo

- ▶ Para avaliar **a-2*b**

$$t_1 \rightarrow b$$

$$t_2 \rightarrow 2 * t_1$$

$$t_3 \rightarrow a$$

$$t_4 \rightarrow t_3 - t_2$$

- ▶ Poderíamos substituir t_2 e t_4 por t_1
- ▶ Escolha do **esquema de nomes** tem um efeito significativo
 - ▶ Otimização
 - ▶ Tempo de compilação

Categorías de IR

IR Gráficas

- ▶ Grafos – utilizada em muitos compiladores
- ▶ Fatores de diferenciação entre as IR gráficas
 - ▶ Organização estrutural
 - ▶ Nível de abstração
 - ▶ Correlação do grafo com o código
- ▶ São IR Gráficas
 - ▶ Baseadas na árvore sintática
 - ▶ Árvore de derivação
 - ▶ **Árvore Sintática Abstrata** (AST)
 - ▶ Grafos acíclicos direcionados
 - ▶ Baseadas em grafo
 - ▶ Grafo de fluxo de controle
 - ▶ Grafo de dependências

IR Gráficas (Baseadas na Árvore Sintática)

- ▶ Exemplo de base: considerando a gramática abaixo
→ Entradas $x = (b * c) + (b * c)$; e $x = a * 2 + a * 2 * b$;

$L \rightarrow x = E;$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid \text{num} \mid \text{name}$

- ▶ **Árvore de Derivação**
 - ▶ Representação gráfica da derivação
 - ▶ Grande demais
- ▶ **Árvore Sintática Abstrata (AST)**
 - ▶ Mantém somente o essencial de uma árvore de derivação
 - ▶ Sendo fiel a estrutura do código fonte
- ▶ **Grafos Acíclicos Direcionados (DAG)**
 - ▶ Contração da AST que evita duplicações
 - ▶ Mais compacta possível

IR Gráficas (Baseadas em Grafo)

- ▶ Modelar outros aspectos do comportamento do programa

- ▶ **Grafo de Fluxo de Controle (CFG)**

- ▶ Noção de **bloco básico**

- ▶ Exemplo

```
stmt0
```

```
while (i < 100) { stmt1 }
```

```
stmt2
```

```
if (x = y) { stmt3 } else { stmt4 }
```

```
stmt5
```

- ▶ Operações dentro do bloco?

- ▶ AST de expressões, DAG, ou uma IR linear

- ▶ **IR Híbrido**

- ▶ Conceito diferente de bloco básico → trade-off

- ▶ Muitas partes do compilador dependem do CFG

- ▶ Escalonamento de instruções
 - ▶ Alocação global de registradores

IR Gráficas (Baseadas em Grafo)

- ▶ **Grafo de Dependências de Dados**

- ▶ Codificam o fluxo de valores
- ▶ Do ponto da definição até sua utilização

- ▶ Nós representam operações

- ▶ Arestas representam relação entre definição e utilização

- ▶ Exemplo

```
1  x = 0
2  i = 1
3  while (i < 100)
4      if (a[i] > 0)
5          then x = x + a[i]
6          i = i + 1
7  print x
```

IR Lineares

- ▶ Consiste em uma sequência de operações
- ▶ Impõe ordem clara e útil
- ▶ Codifica fluxo de controle
 - ▶ Operações de salto e desvios
- ▶ Muitos tipos existem
 - ▶ Código de um endereço
 - ▶ Notação Pós-fixada e Pré-fixada
 - ▶ Código de dois endereços
 - ▶ Código de três endereços (TAC)

IR Lineares – Código de um endereço

- ▶ Modela máquinas de acumulação e de pilha
- ▶ Código é bem compacto
- ▶ **Código de Máquina-Pilha**
 - ▶ Assume a presença de uma pilha de operandos
 - ▶ Parâmetros dos operandos está no topo da pilha
 - ▶ Exemplo para a entrada $a - 2 * b$

```
push 2
push b
multiply
push a
subtract
```

- ▶ Todos os resultados e argumentos são transitórios
- ▶ Exemplo prático similar: Bytecodes do Java

LR Lineares – Notação Pós e Pré-fixada

► Conceito

Infixada	Pós-Fixada	Pré-fixada
$a + b * c$	$ab + c*$	$* + abc$
$a * b + c$	$abc + *$	$*a + bc$
$a + b * c$	$abc * +$	$+a * bc$

► Esquema de tradução para gerar uma LR pós-fixada

$E \rightarrow E_1 + T$	$\{ E.cod = E_1.cod \parallel T.cod \parallel "+" \}$
$E \rightarrow T$	$\{ E.cod = T.cod \}$
$T \rightarrow T_1 * F$	$\{ T.cod = T_1.cod \parallel F.cod \parallel "*" \}$
$T \rightarrow F$	$\{ T.cod = F.cod \}$
$F \rightarrow id$	$\{ F.cod = id.nome \}$

IR Lineares – Código de Três Endereços (TAC)

- ▶ Forma da maioria das operações possíveis

$$i \leftarrow j \text{ op } k$$

- ▶ Algumas operações não precisam de todos os argumentos
- ▶ Exemplo para a expressão $a - 2 * b$

$$t_1 \leftarrow 2$$

$$t_2 \leftarrow b$$

$$t_3 \leftarrow t_1 * t_2$$

$$t_4 \leftarrow a$$

$$t_5 \leftarrow t_4 + t_3$$

- ▶ Vantagens

- ▶ Razoavelmente compacto, **sem operações destrutivas**
- ▶ Importância da escolha do espaço de nomes

- ▶ Nível de abstração é controlável

- ▶ TAC de baixo nível (LIR) \rightarrow Assembly
- ▶ TAC de médio nível (MIR)

\Rightarrow `mvcl source dest count # move characters long`

- ▶ Permite abstrair a complexidade da operação

IR Lineares – Implementando TACs

- ▶ Como manter o código TAC em memória?
- ▶ Considerando
 - ▶ Renomear variáveis temporárias
 - ▶ Trocar a ordem de operações
 - ▶ Otimizações
- ▶ Três abordagens
 - ▶ Tabela (ou matriz)
 - ▶ Vetor de ponteiros
 - ▶ Lista encadeada

IR – Projeto de Compiladores

- ▶ Etapa 3: Gerar uma IR Gráfica – AST
- ▶ Etapa 4...: Gerar uma IR Linear – TAC

Geração de IR

Código de Três Endereços (TAC)

Geração de IR – TAC

- ▶ Expressões
- ▶ Declarações (escopo simples)
- ▶ Declarações (escopo aninhados)
- ▶ Comandos de atribuição
- ▶ Vetores e Registros
- ▶ Expressões booleanas
- ▶ Comandos de decisão
- ▶ Comandos de iteração

Geração – Expressões

- ▶ Considerando a gramática aritmética simplificada

$$S \rightarrow \text{nome} = E;$$
$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow (E) \mid \text{id}$$

- ▶ Supondo que temos a entrada $\text{var} = x + y * z$;
 - ▶ O código TAC correspondente terá três operações

$$t_1 = y * z$$
$$t_2 = x + t_1$$
$$\text{var} = t_2$$

Geração – Expressões Implementação

- ▶ Dois mecanismos
- ▶ Atributos **sintetizados**
 - ▶ Atributo *local* armazena o nome da variável – t_1 ou t_2
 - ▶ Atributo *codigo* armazena o código TAC sintetizado
- ▶ Funções auxiliares
 - ▶ Função *gera()* escreve o código
 - ▶ Símbolo $||$ significa concatenação
 - ▶ A função *temp()* retorna um nome de temporário – t_x

Geração – Expressões Esquema de Tradução

$S \rightarrow$	$\text{nome} = E;$	$\{ S.\text{codigo} = E.\text{codigo}; \parallel$ $\text{gera}(\text{nome.local} = E.\text{local}); \}$
$E \rightarrow$	$E_1 + E_2$	$\{ E.\text{local} = \text{temp}();$ $E.\text{codigo} = E_1.\text{codigo} \parallel E_2.\text{codigo} \parallel$ $\text{gera}(E.\text{local} = E_1.\text{local} + E_2.\text{local}); \}$
$E \rightarrow$	$E_1 * E_2$	$\{ E.\text{local} = \text{temp}();$ $E.\text{codigo} = E_1.\text{codigo} \parallel E_2.\text{codigo} \parallel$ $\text{gera}(E.\text{local} = E_1.\text{local} * E_2.\text{local}); \}$
$E \rightarrow$	(E_1)	$\{ E.\text{local} = E_1.\text{local};$ $E.\text{codigo} = E_1.\text{codigo} \}$
$E \rightarrow$	id	$\{ E.\text{local} = \text{id.lexval};$ $E.\text{codigo} = ""; \}$

- ▶ Testar com o exemplo $\text{var} = x + y * z;$
 - ▶ Gerar árvore de derivação ou **AST**
 - ▶ Executar as ações semânticas de tradução

Conclusão

- ▶ Leituras Recomendadas
 - ▶ Livro do Keith
 - ▶ Capítulo 5
 - ▶ Livro do Dragão
 - ▶ Seções 6.1, 6.2 e 6.3
 - ▶ Série Didática
 - ▶ Seção 5.1
- ▶ Próxima Aula

Geração de Código Intermediário