Compiladores e Computabilidade

Prof. Leandro C. Fernandes
UNIP – Universidade Paulista, 2018

GERAÇÃO DE CÓDIGO INTERMEDIÁRIO

Geração de Código Intermediário

- Corresponde a 1ª etapa do processo de Síntese (modelo de Análise e Síntese)
- Em geral, ocorre em duas fases:
  - Tradução da estrutura construída na análise sintática para um código em linguagem intermediária, usualmente independente do processador.
  - Tradução do código em linguagem intermediária para a linguagem simbólica do processador-alvo.
- A produção efetiva do código binário é realizada por outro programa (montador).

Por quê código intermediário?

- Vantagens:
  - Uma mesma entrada, porém com diferentes formas de saída;
  - Possibilita a otimização 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.
  - Possibilita a tradução de código intermediário para diversas máquinas

Tipos de Código Intermediário

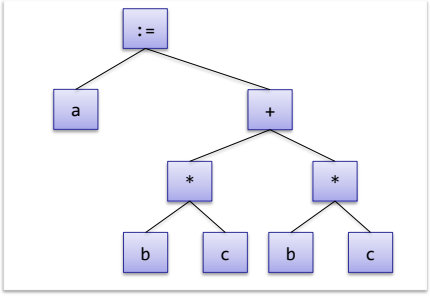
- HIR (*High Intermediate Representation*):
  - Usada nos primeiros estágios do compilador
  - Simplificação de construções gramaticais para somente o essencial para otimização/geração de código
- MIR (*Medium Intermediate Representation*):
  - Boa base para geração de código eficiente
  - Pode expressar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

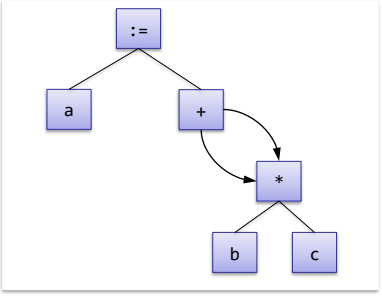
Tipos de Código Intermediário

- Há várias formas de representação de código intermediário, sendo as mais comuns:
  - **HIR:** Árvore e grafo de sintaxe.
    - Notações Pós-fixada e Pré-fixada.
    - Representações linearizadas.
  - **MIR:** Código de três endereços:
    - Quádruplas ou triplas
  - **LIR:** Instruções *assembler*

Árvore e grafo de sintaxe

- A árvore de sintaxe mostra a estrutura hierárquica de um programa fonte,
- Enquanto o grafo de sintaxe inclui simplificações da árvore de sintaxe.
- Exemplo: `a = b * c + b * c`





Representação Notação Pós-fixa

- C++

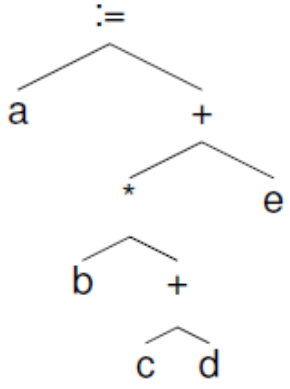
```
a = b*(c+d)+e;
```

- Tradução

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

(varredura em pós-ordem)

Árvore Sintática Abstrata



Código de Três Endereços

- Cada instrução referencia, no máximo, três elementos (dois operandos e um resultado).
  - Expressões complexas devem ser decompostas em várias expressões nesse formato
- Necessita variáveis temporárias!
- Formato independente de um processador específico
- Fácil de traduzir para linguagem simbólica de qualquer processador.

Tipos de Instruções

- Instruções de atribuição
  - cópia
  - resultado de operações binárias
  - resultado de operações unárias
- Instruções de desvio
  - incondicional
  - condicional
  - invocação de rotinas
- Operadores de endereçamento
  - indexado
  - indireto

Código de três endereços:

Instruções de atribuição

- Atribuição simples

```
le := ld
```

- Exemplo
  - Código C/C++:

```
x = a;
```

  - Tradução:

```
x := a
```

Código de três endereços:

Instruções de atribuição

- Operação binária com atribuição

```
le := ld1 op ld2
```

- Exemplo
  - Código C/C++:

```
x = a + b;
```

  - Tradução:

```
x := a + b
```

Código de três endereços:
Instruções de atribuição

- Operação unária com atribuição

```
le := op ld
```

- Exemplo
  - Código C/C++:

```
x = -a;
```

  - Tradução:

```
x := -a
```

Código de três endereços:
Instruções de atribuição

- Expressões mais complexas são traduzidas para uma série de expressões nesse formato
  - Variáveis temporárias são criadas para armazenar resultados intermediários, conforme necessário
- Exemplo
  - Expressão C/C++:

```
a = b + c * d;
```

  - Tradução:

```
_t1 := c * d
a := b + _t1
```

Código de três endereços:

Instruções de desvio

- Utilizados tanto para comandos de decisão ou de repetição, além de chamadas a sub-rotinas.

- Desvio incondicional

```
goto L
```

- Desvio condicional

```
if x opr y goto L
```

onde opr é um operador relacional

Código de três endereços:

Instruções de desvio

C++	**Tradução**

`while (i++ <= k)`
`  x[i] = 0;`
`x[0] = 0;`
	`_L1: if i > k goto _L2`
	`  i := i + 1`
	`  x[i] := 0`
	`  goto _L1`
	`_L2: x[0] := 0`

Código de três endereços:

Invocando sub-rotinas

- Padrão de invocação:
  - Parâmetros da função, se presentes, são registrados com instrução `param`
  - Sub-rotina é invocada com instrução `call`, com indicação do número de parâmetros
  - Retorno, se presente, pode ser obtido a partir de atribuição do resultado de `call`

Código de três endereços:

Invocando sub-rotinas

C++	**Tradução**

`x = f(a, g(b));`	`param a`
	`param b`
	`_t1 := call g, 1`
	`param _t1`
	`x := call f, 2`

Código de três endereços:

Modos de Endereçamento

- Modo direto

```
x := a
```

- Modo indexado

```
x := y[i]
x[i] := y
```

com `i` em bytes!

- Modo indireto

```
x := &y
w := *x
*x := z
```

Código de três endereços:

Estruturas de representação

- Quádruplas
  - Tabela com quatro colunas:
  - Operador, primeiro argumento, segundo argumento e resultado
- Triplas
  - Coluna com resultado é omitida
  - Referência à linha da tabela é utilizada para indicar resultados intermediários
  - Operador de atribuição simples deve ser utilizado para explicitar armazenamento de resultados não temporários

Representação por Quádruplas

- C++

```
a = b + c * d;
```

- Quádrupla:

	Operador	Arg_1	Arg_2	Resultado
1	*	c	d	_t1
2	+	b	_t1	a

Representação por Triplas

- C++

```
a = b + c * d;
```

- Tripla:

	Operador	Arg_1	Arg_2
1	*	c	d
2	+	b	(1)
3	:=	a	(2)

OTIMIZAÇÃO

Otimização de código

- Código gerado automaticamente pode ser ineficiente
  - Redundâncias
  - Instruções desnecessárias
- Ineficiência pode vir também da codificação original.
- Heurísticas de otimização podem ser aplicadas no código em formato intermediário antes da produção do código em linguagem simbólica.

Eliminação de subexpressões comuns

- Operações que se repetem sem que seus argumentos sejam alterados podem ser realizadas uma única vez
- Exemplo:
  - Válido se não houver atribuição as variáveis a ou b entre as duas instruções:

```
x = a + b + c;
...
y = a + b + d;
```

Eliminação de subexpressões comuns

Sem otimização:

```
_t1 := a + b
x := _t1 + c
...
_t2 := a + b
y := _t2 + d
```

Com otimização:

```
_t1 := a + b
x := _t1 + c
...
y := _t1 + d
```

Eliminação de código redundante

- Instruções sem efeito podem ser eliminadas
- Exemplo
  - Sem nenhuma atribuição a x ou a y entre as duas instruções,

```
x := y          x := y
...             ...
x := y          z := k
z := k
```

a segunda instrução pode ser seguramente eliminada

Propagação de cópias

- Variáveis que só mantêm cópia de um valor, sem outros usos, podem ser eliminadas
- Exemplo:
  - Sem outras atribuições a y e sem outros usos de x

```
x := y
...
z := x
```

pode ser reduzido a:

```
...
z := y
```

Eliminação de desvios desnecessários

- Desvio incondicional para a próxima instrução pode ser eliminado
- Exemplo:

```
        a := _t2
        goto _L6
  _L6:  c := a + b
```

equivale a:

```
        a := _t2
        c := a + b
```

Eliminação de código não-alcançável

- Instruções que nunca serão executadas podem ser eliminadas.
- Exemplo:

```
      goto _L3
      _t1 := x
_L4:  _t2 := b + c
_L3:  d := a + _t2
```

equivale a:

```
      goto _L3
_L4:  _t2 := b + c
_L3:  d := a + _t2
```

Movimentação de código

- Retirar do corpo de comandos iterativos (laços) cálculo de expressões invariáveis.

- Exemplo:

```
while ( i < 2*max )
   a[i] = i + max/4;
```

  - Como valor de `max` não varia, equivale a:

```
_t1 = 2*max;
_t2 = max/4;
while (i < _t1)
   a[i] = i + _t2;
```

Uso de propriedades algébricas

- Relaciona-se a substituição de certas expressões aritméticas por formas equivalentes e de melhor desempenho.

Original	Equivalente
X + Y	Y + X
X + 0	X
X – 0	X
X * Y	Y * X
X * 1	X
2 * X	X + X
X^2	X * X

GERAÇÃO DE CÓDIGO (LINGUAGEM SIMBÓLICA)

Geração de código final

Objetivo:

- Obter, a partir das instruções elementares usadas no código em formato intermediário, código equivalente na linguagem simbólica do processador-alvo.
  - Diferentes processadores podem ter distintos formatos de instruções.

Geração de código final

- Classificação pelo número de endereços na instrução:
  - 3 dois operandos e o resultado
  - 2 dois operandos (resultado sobrescreve primeiro operando)
  - 1 só segundo operando, primeiro operando implícito (registrador acumulador), resultado sobrescreve primeiro operando
  - 0 operandos e resultado numa pilha, sem endereço explícitos

Tradução para linguagem simbólica

- Tradução ocorre segundo gabaritos definidos de acordo com o tipo de máquina.

Exemplo: `x := y + z`

3-end:

```
ADD y, z, x
```

2-end:

```
MOVE Ri, y
ADD Ri, z
MOVE x, Ri
```

1-end:

```
LOAD y
ADD z
STORE x
```

0-end:

```
PUSH y
PUSH z
ADD
POP x
```

Otimização em linguagem simbólica

- Ao combinar sequências de instruções traduzidas pelos gabaritos, as estratégias de otimização podem ser também aplicadas ao código em linguagem simbólica
  - Eliminação de código redundante
  - Eliminação de instruções desnecessárias
- Além disso, há também a oportunidade de realizar otimizações que são específicas para um determinado processador
  - Otimizações dependentes de máquina
  - Permitem o aproveitamento de instruções pouco usuais, de uso restrito
  - Muitas vezes, é difícil para o compilador reconhecer a possibilidade de uso dessas instruções