Compiladores (CC3001)
Aula 10: Geração de código intermédio

Mário Florido

DCC/FCUP

2024

[dcc] FC FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Esta aula

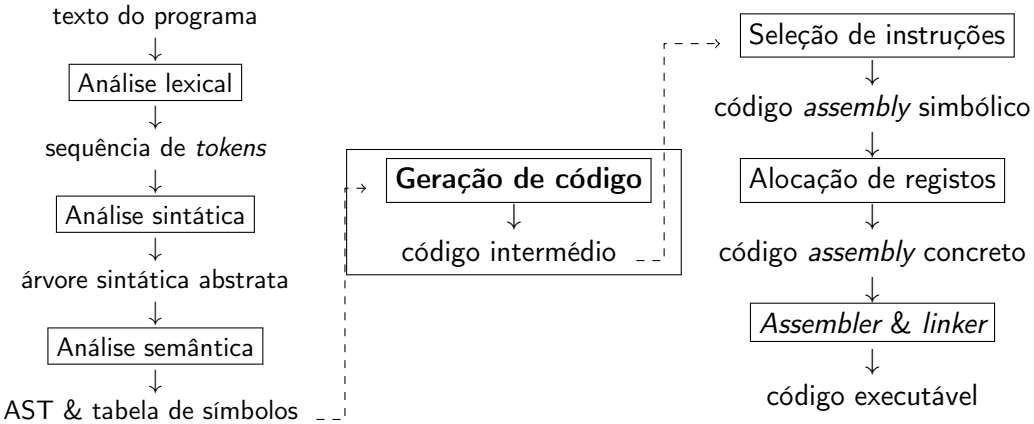
Códigos intermédios

Código de três endereços

Tradução para código intermédio

- Expressões
- Comandos
- Definição e chamada de funções
- Operadores lógicos
- *Arrays*

Re-lembrar: fases dum compilador



Códigos intermédios

- Linguagens usadas nas fases intermédias da compilação
- Compromisso entre a linguagem-fonte e o código-máquina real
- Objetivo: facilitar a compilação
- Diferentes opções:

  mais alto-nível facilita a tradução a partir da linguagem-fonte (mas dificulta a geração de código máquina)

  mais baixo-nível facilita a geração de código-máquina (mas dificulta tradução da linguagem-fonte)
- Também podem ser usados para implementar interpretadores

- As linguagens de alto-nível são mais distintas entre si do que as linguagens de máquina
- A linguagem fonte frequente determina o código intermédio, e.g.:
  - *Java Virtual Machine* (JVM) para compilar Java
  - *Low Level Virtual Machine* (LLVM) para compilar C/C++
- Mas também é possível re-utilizar o código intermédio:
  - Scala e Clojure compilam para a JVM
  - Rust, Swift e Julia compilam para LLVM
- Alguns compiladores usam mais do que um código intermédio (e.g. o GHC usa 3 linguagens intermédias na compilação de Haskell)

Vamos estudar um código intermédio de três endereços:

- Número arbitrário de registos temporários
- Operações com 2 ou 3 operandos
- Sem instruções de processadores específicos
- Adequado para implementar uma linguagem imperativa simples (e.g. C ou Pascal)

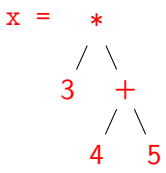
Exemplo de motivação

Queremos gerar codigo para a atribuição

```
x = 3*(4+5)
```

Podemos decompor em atribuições mais simples usando variáveis temporárias:

```
t1 = 3
t2 = 4
t3 = 5
t4 = t2+t3
t5 = t1*t4
x = t5
```



- Cada variável corresponde a um nó da expressão
- Cada operação tem (no máximo) três variáveis (“endereços”)
- Podemos gerar as atribuições com uma tradução dirigida pela sintaxe

$$
\begin{aligned}
\mathit{Instr} \rightarrow\ & \textbf{temp} := \mathit{Atom} \\
\mid\ & \textbf{temp} := \textbf{temp binop } \mathit{Atom} \\
\mid\ & \texttt{LABEL } \textbf{label} \\
\mid\ & \texttt{JUMP } \textbf{label} \\
\mid\ & \texttt{COND } \textbf{temp relop } \mathit{Atom}\ \ \textbf{label label}
\end{aligned}
$$

$$
\mathit{Atom} \rightarrow \textbf{temp} \mid \textbf{num}
$$

- identificadores (**temp**), atribuições e constantes (**num**)
- operações binárias **binop**: +, *, etc.
- comparações **relop**: <, >, ==, etc.
- etiquetas, saltos incondicionais e condicionais

Exemplo: algoritmo de Euclides

```
while (b != 0) {
  r = a%b;
  a = b;
  b = r;
}
```

```
LABEL loop
COND b != 0 next end
LABEL next
r := a % b
a := b
b := r
JUMP loop
LABEL end
```

Observações

- Sem declarações de variáveis ou funções (apenas o corpo)
- `LABEL` define uma etiqueta para um ponto no programa
- Apenas duas formas de controlo de fluxo:

  `JUMP` *label* salto incondicional para *label*

  `COND` *cond* $label_t$ $label_f$ testa uma condição e salta para $label_t$ ou $label_f$[1]
- As condições têm ser comparações simples

$$cond \rightarrow id \ \textbf{relop} \ id \ \mid \ id \ \textbf{relop} \ const$$

[1] No livro: `IF...THEN...ELSE`

Linguagem fonte

- Variáveis, expressões aritméticas e comparações
- Atribuições, *if/else*, ciclos *while*
- A tradução para código intermédio será dirigida pela sintaxe
  - uma função recursiva por cada categoria sintática (expressões, comandos, etc.)
  - passamos argumentos extra dependendo do contexto
  - o resultado é uma lista de instruções
  - na exposição usamos a sintaxe concreta; na implementação usamos a AST

$$Exp \rightarrow \textbf{num} \mid \textbf{id} \mid Exp\ \textbf{binop}\ Exp$$

- Uma tabela de símbolos para associar identificadores do programa com temporários do código intermédio
- Para gerar nomes temporários usamos *pseudo*-funções

$$newTemp : () \rightarrow Temp$$
$$newLabel : () \rightarrow Label$$

- Não são funções puras: devem retornar uma variáveis ou etiquetas distintas de cada vez que são chamadas
- A função de tradução

$$transExpr : (Exp, Table, Temp) \rightarrow [Instr]$$

recebe também o destino onde colocar o resultado (*atributo herdado*)

transExpr (*expr*, *table*, *dest*)	= case *expr* of
num	return [*dest* := **num**]
id	*temp* = lookup(**id**, *table*) return [*dest* := *temp*]
e_1 **binop** e_2	$t_1 = newTemp()$ $t_2 = newTemp()$ $code_1 = transExpr(e_1, table, t_1)$ $code_2 = transExpr(e_2, table, t_2)$ return $code_1 ++ code_2 ++ [dest := t_1$ **binop** $t_2]$

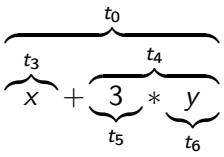
Exemplo

Assumindo a tabela $[x \mapsto t_1,\ y \mapsto t_2]$ traduzir a expressão

```
x + (3*y)
```

colocando o resultado em `t0`.

```
t3 := t1
t5 := 3
t6 := t2
t4 := t5 * t6
t0 := t3 + t4
```



(Usamos temporários $t_3, t_4, \ldots$.)

- Atribuições
- Condicionais (com e sem *else*)
- Ciclos *while*
- Comparações entre expressões
- Blocos

$$
\begin{aligned}
\mathit{Stm} \rightarrow\ & \textbf{id} = \mathit{Exp}; \\
\mid\ & \textbf{if}(\mathit{Cond})\ \mathit{Stm} \\
\mid\ & \textbf{if}(\mathit{Cond})\ \mathit{Stm}\ \textbf{else}\ \mathit{Stm} \\
\mid\ & \textbf{while}(\mathit{Cond})\ \mathit{Stm} \\
\mid\ & \{\ \mathit{StmList}\ \}
\end{aligned}
$$

$$\mathit{Cond} \rightarrow \mathit{Exp}\ \textbf{relop}\ \mathit{Exp}$$

$$\mathit{StmList} \rightarrow \mathit{Stm}\ \mathit{StmList} \mid \epsilon$$

Função de tradução:

$$\mathit{transStm} : (\mathit{Stm}, \mathit{Table}) \rightarrow [\mathit{Instr}]$$

- A definição de *transStm* é dirigida pela sintaxe
- Casos mais simples: atribuições e blocos

transStm (*stm*, *table*) = case *stm* of	
id = *expr*;	*dest* = lookup(**id**, *table*) return *transExpr*(*expr*, *table*, *dest*)
{ $stm_1 \ldots stm_n$ }	$code_1$ = *transStm*(stm_1, *table*) ⋮ $code_n$ = *transStm*(stm_n, *table*) return $code_1 ++ \cdots ++ code_n$

Função auxiliar para condições

- Uma função auxiliar para compilar condições
- Argumentos extra: etiquetas $label_t$ e $label_f$ para onde saltar caso *true/false* (*atributos herdados*)
- Vai ser usada na tradução de *if/else* e *while*

$transCond : (Cond, Table, Label, Label) \rightarrow [Instr]$

$transCond\ (cond, tabl, label_t, label_f) =$ case $cond$ of

$expr_1$ **relop** $expr_2$	$t_1 = newTemp()$
	$t_2 = newTemp()$
	$code_1 = transExpr(expr_1, tabl, t_1)$
	$code_2 = transExpr(expr_2, tabl, t_2)$
	return $code_1 ++ code_2 ++ [\texttt{COND}\ t_1\ \textbf{relop}\ t_2\ label_t\ label_f]$

$transStm\ (stm,\ table) = \text{case}\ stm\ \text{of}$	
$\textbf{if}(cond)\ stm_1$	$label_1 = newLabel()$
	$label_2 = newLabel()$
	$code_1 = transCond(cond, label_1, label_2, table)$
	$code_2 = transStm(stm_1, table)$
	$\text{return}\ code_1 ++[\texttt{LABEL}\ label_1] ++code_2 ++[\texttt{LABEL}\ label_2]$

$transStm\ (stm,\ tabl) = \text{case } stm \text{ of}$

if$(Cond)\ stm_1$	$label_1 = newLabel()$
else stm_2	$label_2 = newLabel()$
	$label_3 = newLabel()$
	$code_1 = transCond(cond, label_1, label_2, table)$
	$code_2 = transStm(stm_1, table)$
	$code_3 = transStm(stm_2, table)$
	return $code_1 \mathbin{++} [\texttt{LABEL}\ label_1] \mathbin{++} code_2 \mathbin{++} [\texttt{JUMP}\ label_3]$
	$\mathbin{++}\ [\texttt{LABEL}\ label_2] \mathbin{++} code_3 \mathbin{++} [\texttt{LABEL}\ label_3]$

Assumindo a tabela $[x \mapsto t_0, y \mapsto t_1, z \mapsto t_2]$ vamos traduzir o comando seguinte.

```
if (x < y)
   z = y;
else
   z = x;
```

```
t3 := t0
t4 := t1
COND t3 < t4 label1 label2
LABEL label1
t2 := t1
JUMP label3
LABEL label2
t2 := t0
LABEL label3
```

transStm (*stm*, *table*) = case *stm* of	
while(*cond*) stm_1	$label_1 = newLabel()$
	$label_2 = newLabel()$
	$label_3 = newLabel()$
	$code_1 = transCond(cond, table, label_2, label_3)$
	$code_2 = transStm(stm_1, table)$
	return $[\texttt{LABEL}\ label_1] ++ code_1$
	$++ [\texttt{LABEL}\ label_2] ++ code_2$
	$++ [\texttt{JUMP}\ label_1, \texttt{LABEL}\ label_3]$

Assumindo a tabela $[n \mapsto t_0, r \mapsto t_1]$ vamos traduzir os comandos seguintes.

```
{
  n = 5;
  r = 1;
  while (n>0) {
     r = r*n;
     n = n-1;
  }
}
```

```
t2 := 5
t0 := t2
t3 := 1
t1 := t3
LABEL label1
t4 := t0
t5 := 0
COND t4 > t5 label2 label3
LABEL label2
t6 := t1
t7 := t0
t1 := t6 * t7
t8 := t0
t9 := 1
t0 := t8 - t9
JUMP label1
LABEL label3
```

- Vamos acrescentar a definição e chamada de funções como ao código intermédio
- Restrição: no código intermédio os argumentos de funções devem ser temporários
  - expressões complexas têm de ser calculadas e colocadas em temporários antes da chamada da função
- A implementação de passagem de parâmetros usando registos e/ou pilha fica para a tradução para código *assembly*
- Simplificação: um único tipo **int**

Linguagem fonte

$$
\begin{aligned}
\mathit{Function} &\rightarrow \textbf{int}\ \textbf{id}(\mathit{Decls})\ \{\mathit{Decls}; \mathit{Stms}\} \\
\mathit{Decls} &\rightarrow \textbf{int}\ \textbf{id}, \ldots, \textbf{id} \\
\mathit{Stm} &\rightarrow \cdots \\
&\quad \mid \textbf{return}\ \mathit{Exp}; \\
\mathit{Exp} &\rightarrow \cdots \\
&\quad \mid \textbf{id}(\mathit{Exp}, \ldots, \mathit{Exp})
\end{aligned}
$$

Código intermédio

$$
\begin{aligned}
\mathit{Function} &\rightarrow \textbf{id}(\mathit{TempList})\ [\ \mathit{InstrList}\ ] \\
\mathit{TempList} &\rightarrow \textbf{temp}, \ldots, \textbf{temp} \\
\mathit{InstrList} &\rightarrow \mathit{Instr}; \ldots; \mathit{Instr} \\
\mathit{Instr} &\rightarrow \cdots \\
&\quad \mid \textbf{temp} := \texttt{CALL}\ \textbf{id}(\mathit{TempList}) \\
&\quad \mid \texttt{RETURN}\ \textbf{temp}
\end{aligned}
$$

```
int max(int x, int y)
{
  if (x<y) return y;
  else return x;
}
```

```
max(t0,t1) [
COND t0 < t1 ltrue lfalse
LABEL ltrue
RETURN t1
LABEL lfalse
RETURN t0
]
```

Exemplos (cont.)


```
int max3(int x, int y, int z)
{
  int m;
  m = x;
  if (m < y) m = y;
  if (m < z) m = z;
  return m;
}
```

```
max(t0,t1,t2) [
t3 := t0
COND t3 < t1 L1 L2
LABEL L1
t3 := t1
LABEL L2
COND t3 < t2 L3 L4
LABEL L3
t3 := t2
LABEL L4
RETURN t3
]
```

```
int max3(int x, int y, int z)
{
  return max(x,max(y,z));
}
```

```
max3(t0, t1, t2) [
t3 := CALL max(t1, t2)
t4 := CALL max(t0, t3)
RETURN t4
]
```

- Na tradução da chamada $f(e_1, \ldots, e_n)$ necessitamos de traduzir cada um dos argumentos
- Usamos uma função auxiliar *transArgs* que retorna também a lista de temporários
- Na tradução de **return** *e* usamos *transExpr* para gerar código para a expressão

transExpr (*exp*, *table*, *dest*)	= case *exp* of
id(*exps*)	(*code*, *temps*) = *transArgs*(*exps*, *table*) return *code* ++[*dest* := `CALL` **id**(*temps*)]
transArgs (*exps*, *table*)	= case *exps* of
$e_1, \ldots, e_n$	$t_1 = newTemp()$ $\vdots$ $t_n = newTemp()$ $code_1 = transExpr(e_1, table, t_1)$ $\vdots$ $code_n = transExpr(e_n, table, t_n)$ return $(code_1 ++ \cdots ++ code_n, [t_1, \ldots, t_n])$

Tradução de definições de funções

- Traduzimos cada definição de função separadamente
- Os temporários são locais a cada função
- Os primeiros temporários t_0, t_1, etc. são usados para os argumentos da funções
- Os restantes são usados para as variáveis locais e valores intermédias
- Construimos a tabela de símbolos e aplicamos a tradução de comandos

```
int foo(int x, int y, int z)  // x->t0, y->t1, z->t2
{
  int a, b;                   // a->t3, b->t4

  ...                         // t5, t6, etc. como temporários
}
```

Operadores lógicos

- Vamos extender condições com operadores lógicos: conjunção (`&&`), disjunção (`||`) e negação (`!`)
- A maioria das linguagens de programação implementa conjunção e disjunção usando *short-circuit evaluation*:
  - não avaliar o segundo argumento se o primeiro determinar o resultado
- Permite simplificar o controlo de fluxo, e.g.

  ```
  if (k!=0 && n%k==0) { ... }
  ```

  é equivalente a

  ```
  if (k!=0) { if (n%k==0) { ... } }
  ```

- Vamos também permitir usar de condições como expressões e vice-versa, e.g.

  ```
  c = a==0 || b==0;
  if(c) { ... }
  ```

Operadores lógicos (cont.)

$$
\begin{aligned}
\mathit{Exp} \rightarrow\ & \textbf{num} \\
\mid\ & \textbf{id} \\
\mid\ & \mathit{Exp}\ \textbf{binop}\ \mathit{Exp} \\
\mid\ & \textbf{id}(\mathit{Exps}) \\
\mid\ & \texttt{true} \\
\mid\ & \texttt{false} \\
\mid\ & \mathit{Cond}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Cond} \rightarrow\ & \mathit{Exp}\ \textbf{relop}\ \mathit{Exp} \\
\mid\ & \texttt{!}\,\mathit{Cond} \\
\mid\ & \mathit{Cond}\ \texttt{\&\&}\ \mathit{Cond} \\
\mid\ & \mathit{Cond}\ \texttt{||}\ \mathit{Cond} \\
\mid\ & \texttt{true} \\
\mid\ & \texttt{false} \\
\mid\ & \mathit{Exp}
\end{aligned}
$$

- Representamos valores lógicos pelos inteiros 0 e 1
- Traduzimos `true` e `false` por uma atribuição direta
- Outros casos: usamos a tradução de condições para atribuir o resultado correto

Condições como expressões (cont.)

transExpr (*exp*, *table*, *dest*)	= case *expr* of
⋮	
`true`	return [*dest* := 1]
`false`	return [*dest* := 0]
cond	$label_1 = newLabel()$ $label_2 = newLabel()$ $label_3 = newLabel()$ $code_1 = transCond(cond, table, label_1, label_2)$ return $code_1$ ++[`LABEL` $label_1$, *dest* := 1, `JUMP` $label_3$] ++ [`LABEL` $label_2$, *dest* := 0, `LABEL` $label_3$]

Recordar: a função de tradução

$$transCond\ (cond,\ table,\ label_t,\ label_f)$$

gera código que salta para $label_t$ e $label_f$ conforme a condição é verdadeira ou falsa.

Vamos ver como extender para os operadores lógicos.

Tradução de condições (cont.)

transCond (*cond*, *table*, $label_t$, $label_f$) =	case *cond* of
e_1 **relop** e_2	⋮ *(como anteriormente)*
`true`	return [`JUMP` $label_t$]
`false`	return [`JUMP` $label_f$]
! $cond_1$	*transCond*($cond_1$, *table*, $label_f$, $label_t$)

- Constantes `true` e `false` traduzem-se em saltos incondicionais
- A tradução da negação de uma condição apenas troca as etiquetas

$transCond\ (cond, table, label_t, label_f) = \text{case } cond \text{ of}$	
$cond_1$ && $cond_2$	$label_2 = newLabel()$
	$code_1 = transCond(cond_1, table, label_2, label_f)$
	$code_2 = transCond(cond_2, table, label_t, label_f)$
	return $code_1 ++ [\texttt{LABEL}\ label_2] ++ code_2$
$cond_1$ || $cond_2$	$label_2 = newLabel()$
	$code_1 = transCond(cond_1, table, label_t, label_2)$
	$code_2 = transCond(cond_2, table, label_t, label_f)$
	return $code_1 ++ [\texttt{LABEL}\ label_2] ++ code_2$

A tradução de conjunções e disjunções introduz uma etiqueta temporária para avaliar as condições em sequência

transCond (*cond*, *table*, $label_t$, $label_f$) = case *cond* of	
exp	$t = newTemp()$ $code_1 = transExpr(exp, table, t)$ return $code_1$ ++[`COND` t`!=0` $label_t$ $label_f$]

Outros casos: traduzimos uma expressão e comparamos o resultado com zero

Exemplo

Supondo $[a \mapsto t_0, b \mapsto t_1]$.

```
if (a != 0 && b > a) {
   b = b - a;
}
```

```
t2 := t0
t3 := 0
COND t2 != t3 label2 label4
LABEL label2
t4 := t1
t5 := t0
COND t4 > t5 label3 label4
LABEL label3
t6 := t1
t7 := t0
t1 := t6 - t7
LABEL label4
```

Observações

- Não existem operadores lógicos na linguagem intermédia (são implementados como saltos condicionais)
- Alternativa: tratar operadores lógico tal como os aritméticos (mas isso obrigaria a avaliar ambos os argumentos)
- *Cond* e *Exp* têm muita sobreposição (gramática âmbigua)
- Alternativa: uma só categoria gramatical *Exp* e duas funções de tradução mutuamente recursivas

$$transExpr : (Exp, Table, Temp) \rightarrow [Instr]$$
$$transCond : (Exp, Table, Label, Label) \rightarrow [Instr]$$

Usamos *transExpr* nas expressões e *transCond* nas condições de *if* e *while*

Extensões para *arrays*

- Indexar *arrays*

$$Exp \to \cdots \mid \mathbf{id}[Exp]$$

- Atribuir um índice de um *array*

$$Stm \to \cdots \mid \mathbf{id}[Exp] := Exp$$

- Novas instruções de código intermédio para ler e escrever na memória

$$\begin{aligned} Instr \to \cdots \\ \mid\ & \mathbf{temp} := \mathtt{M}[Atom] \\ \mid\ & \mathtt{M}[Atom] := \mathbf{temp} \end{aligned}$$

(*Atom* é um temporário ou uma constante.)

Extensões para *arrays* (cont.)

- Tabela de símbolos associa o nome do *array* ao endereço base
- Função auxiliar para calcular o endereço de um índice (exemplo: elementos de 4 *bytes*)
- Resultado: o código e o temporário com o endereço

$transIndex\ (exp,\ table) =$	case exp of
$\mathbf{id}[exp_1]$	$base = \text{lookup}(\mathbf{id}, table)$
	$addr = newTemp()$
	$code_1 = transExpr(exp_1, table, addr)$
	return $(code_1 +\!\!+ [addr := addr * 4, addr := addr + base], addr)$

$transExpr\ (exp,\ table,\ dest)$	$=$ case exp of
$\mathbf{id}[e_1]$	$(code_1, addr) = transIndex(\mathbf{id}[e_1], table)$ return $code_1 ++ [dest := \mathtt{M}[addr]]$

$transStm\ (stm,\ table)$	$=$ case stm of
$\mathbf{id}[e_1] := e_2$	$(code_1, addr) = transIndex(\mathbf{id}[e_1], table)$ $t = newTemp()$ $code_2 = transExpr(e_2, table, t)$ return $code_1 ++ code_2 ++ [\mathtt{M}[addr] := t]$

- Necessitamos ainda de tratar declarações de *arrays* e alocar memória para os elementos
- A alocação pode ser estática (global) ou dinâmica (na *pilha* ou na *heap*)
- Alocação na pilha: próxima aula
- Alocação estática: declaramos espaço no segmento de *data*

```
// linguagem C
int my_array[10];

# Assembly MIPS
.data
my_array:     .space 40 # 4 bytes * 10 elementos
```