

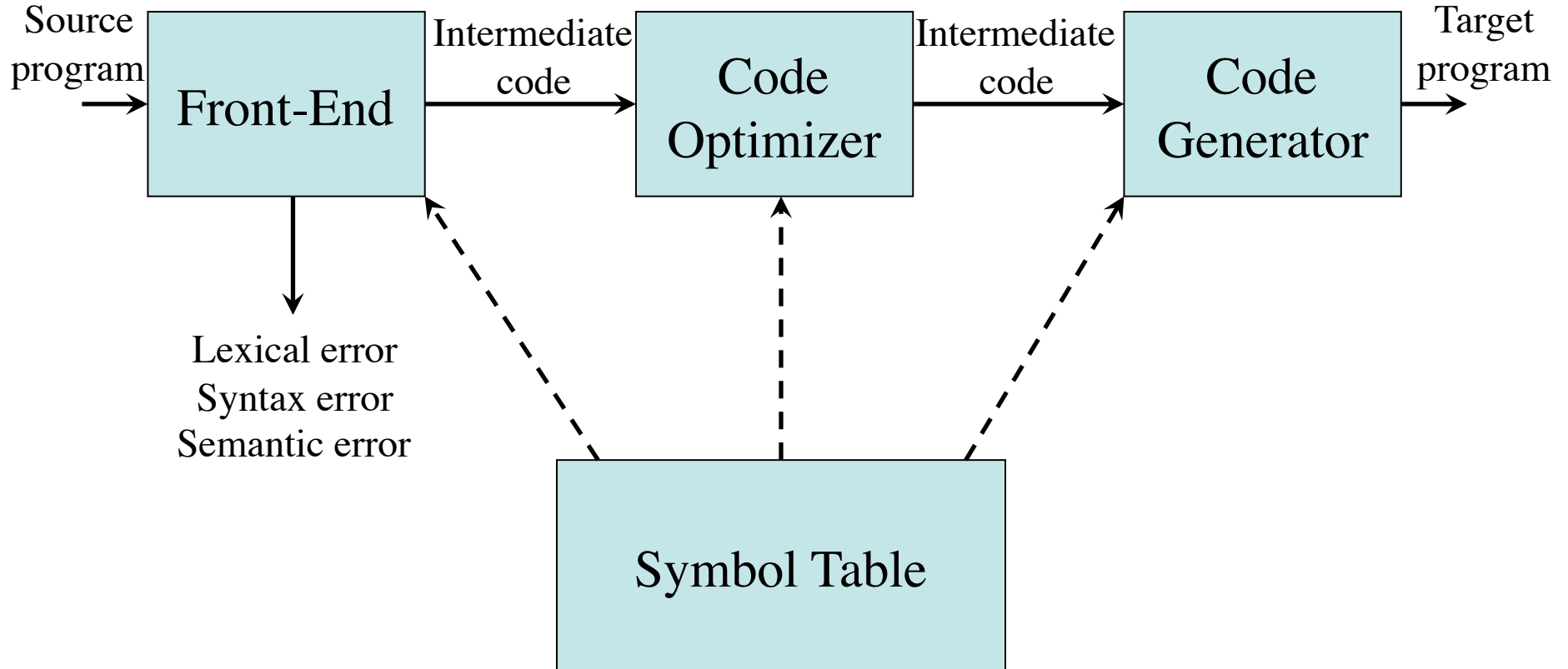
Code Generation

Part I

Chapter 8

(1st ed. Ch.9)

Position of a Code Generator in the Compiler Model



Code Generation

- Code produced by compiler must be correct
 - Source-to-target program transformation should be *semantics preserving*
- Code produced by compiler should be of high quality
 - Effective use of target machine resources
 - Heuristic techniques should be used to generate good but suboptimal code, because generating optimal code is undecidable

Target Program Code

- The back-end code generator of a compiler may generate different forms of code, depending on the requirements:
 - Absolute machine code (executable code)
 - Relocatable machine code (object files for linker)
 - Assembly language (facilitates debugging)
 - Byte code forms for interpreters (e.g. JVM)

The Target Machine

- Implementing code generation requires thorough understanding of the target machine architecture and its instruction set
- Our (hypothetical) machine:
 - Byte-addressable (word = 4 bytes)
 - Has n general purpose registers **R0**, **R1**, ..., **R n -1**
 - Two-address instructions of the form

op source, destination

The Target Machine: Op-codes and Address Modes

- Op-codes (*op*), for example
 - MOV** (move content of *source* to *destination*)
 - ADD** (add content of *source* to *destination*)
 - SUB** (subtract content of *source* from *dest.*)
- Address modes

Mode	Form	Address	Added Cost
Absolute	M	M	1
Register	R	R	0
Indexed	$c(\mathbf{R})$	$c + \text{contents}(\mathbf{R})$	1
Indirect register	*R	$\text{contents}(\mathbf{R})$	0
Indirect indexed	*c(R)	$\text{contents}(c + \text{contents}(\mathbf{R}))$	1
Literal	#c	N/A	1

Instruction Costs

- Machine is a simple, non-super-scalar processor with fixed instruction costs
- Realistic machines have deep pipelines, I-cache, D-cache, etc.
- Define the cost of instruction
= $1 + \text{cost}(\textit{source-mode}) + \text{cost}(\textit{destination-mode})$

Examples

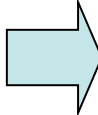
Instruction	Operation	Cost
MOV R0 , R1	Store <i>content</i> (R0) into register R1	1
MOV R0 , M	Store <i>content</i> (R0) into memory location M	2
MOV M , R0	Store <i>content</i> (M) into register R0	2
MOV 4 (R0) , M	Store <i>contents</i> (4+ <i>contents</i> (R0)) into M	3
MOV *4 (R0) , M	Store <i>contents</i> (<i>contents</i> (4+ <i>contents</i> (R0))) into M	3
MOV #1 , R0	Store 1 into R0	2
ADD 4 (R0) , *12 (R1)	Add <i>contents</i> (4+ <i>contents</i> (R0)) to value at location <i>contents</i> (12+ <i>contents</i> (R1))	3

Instruction Selection

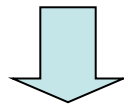
- Instruction selection is important to obtain efficient code
- Suppose we translate three-address code

$x := y + z$

to: **MOV** $y, R0$
ADD $z, R0$
MOV $R0, x$

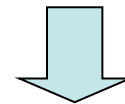
$a := a + 1$  **MOV** $a, R0$
ADD $\#1, R0$
MOV $R0, a$
 Cost = 6

Better



ADD $\#1, a$
 Cost = 3

Best



INC a
 Cost = 2

Instruction Selection: Utilizing Addressing Modes

- Suppose we translate $a := b + c$ into

```
MOV b, R0
ADD c, R0
MOV R0, a
```
- Assuming addresses of **a**, **b**, and **c** are stored in **R0**, **R1**, and **R2**

```
MOV *R1, *R0
ADD *R2, *R0
```
- Assuming **R1** and **R2** contain values of **b** and **c**

```
ADD R2, R1
MOV R1, a
```

Need for Global Machine-Specific Code Optimizations

- Suppose we translate three-address code

$x := y + z$

to: **MOV** $y, R0$
ADD $z, R0$
MOV $R0, x$

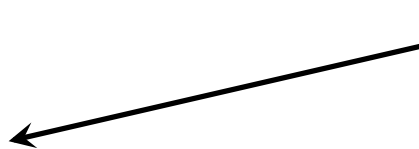
- Then, we translate

$a := b + c$

$d := a + e$

to: **MOV** $a, R0$
ADD $b, R0$
MOV $R0, a$
MOV $a, R0$
ADD $e, R0$
MOV $R0, d$

Redundant



Register Allocation and Assignment

- Efficient utilization of the limited set of registers is important to generate good code
- Registers are assigned by
 - *Register allocation* to select the set of variables that will reside in registers at a point in the code
 - *Register assignment* to pick the specific register that a variable will reside in
- Finding an optimal register assignment in general is NP-complete

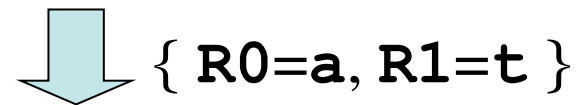
Example

```
t:=a*b  
t:=t+a  
t:=t/d
```



```
MOV a,R1  
MUL b,R1  
ADD a,R1  
DIV d,R1  
MOV R1,t
```

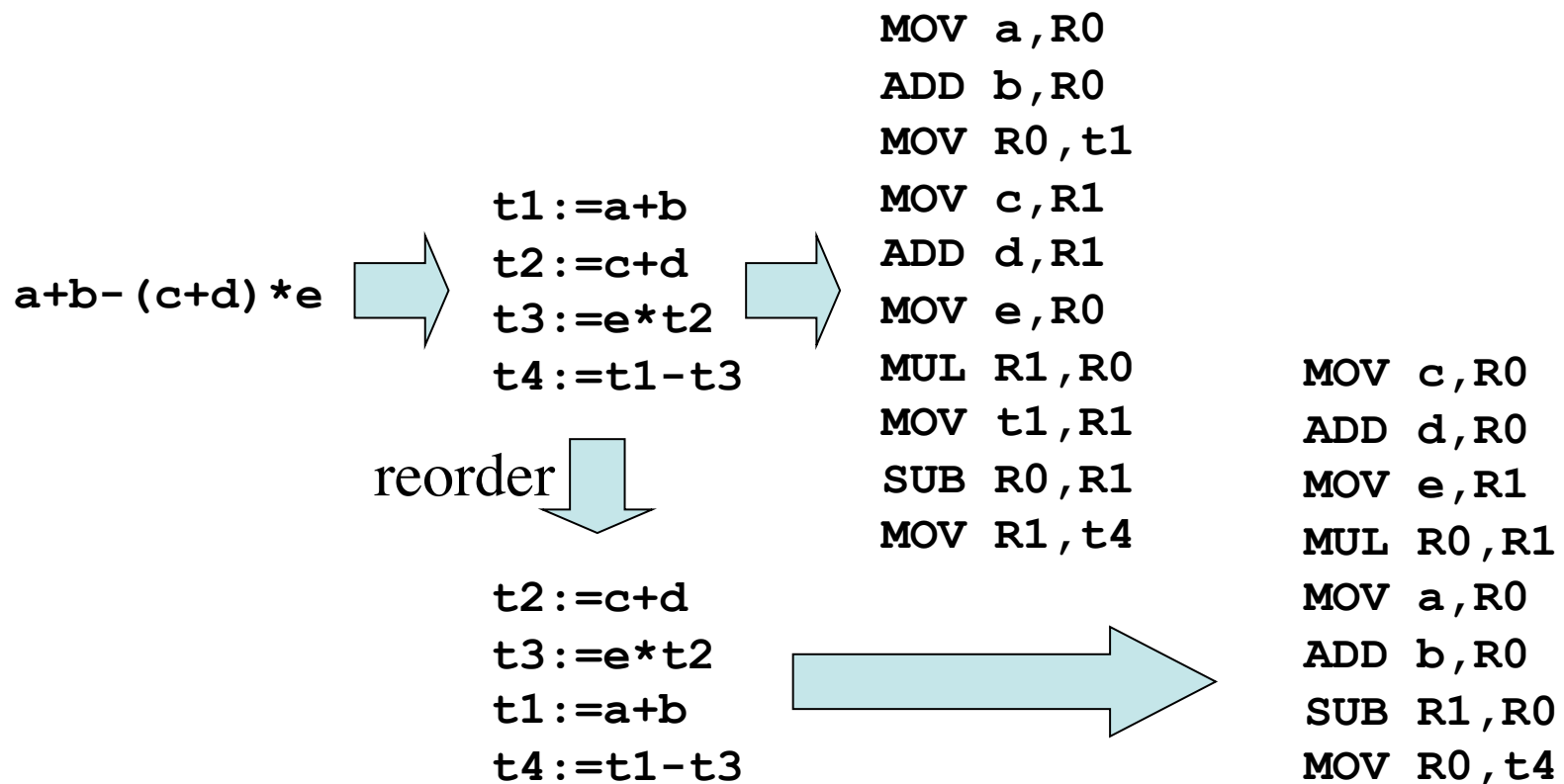
```
t:=a*b  
t:=t+a  
t:=t/d
```



```
MOV a,R0  
MOV R0,R1  
MUL b,R1  
ADD R0,R1  
DIV d,R1  
MOV R1,t
```

Choice of Evaluation Order

- When instructions are independent, their evaluation order can be changed



Generating Code for Stack Allocation of Activation Records

<code>t1 := a + b</code>	<code>100: ADD #16, SP</code>	Push frame
<code>param t1</code>	<code>108: MOV a, R0</code>	
<code>param c</code>	<code>116: ADD b, R0</code>	
<code>t2 := call foo, 2</code>	<code>124: MOV R0, 4 (SP)</code>	Store a+b
<code>...</code>	<code>132: MOV c, 8 (SP)</code>	Store c
	<code>140: MOV #156, *SP</code>	Store return address
	<code>148: GOTO 500</code>	Jump to foo
<code>func foo</code>	<code>156: MOV 12 (SP), R0</code>	Get return value
<code>...</code>	<code>164: SUB #16, SP</code>	Remove frame
<code>return t1</code>	<code>172: ...</code>	
	<code>500: ...</code>	
	<code>564: MOV R0, 12 (SP)</code>	Store return value
	<code>572: GOTO *SP</code>	Return to caller

Note: Language and machine dependent
Here we assume C-like implementation with SP and no FP