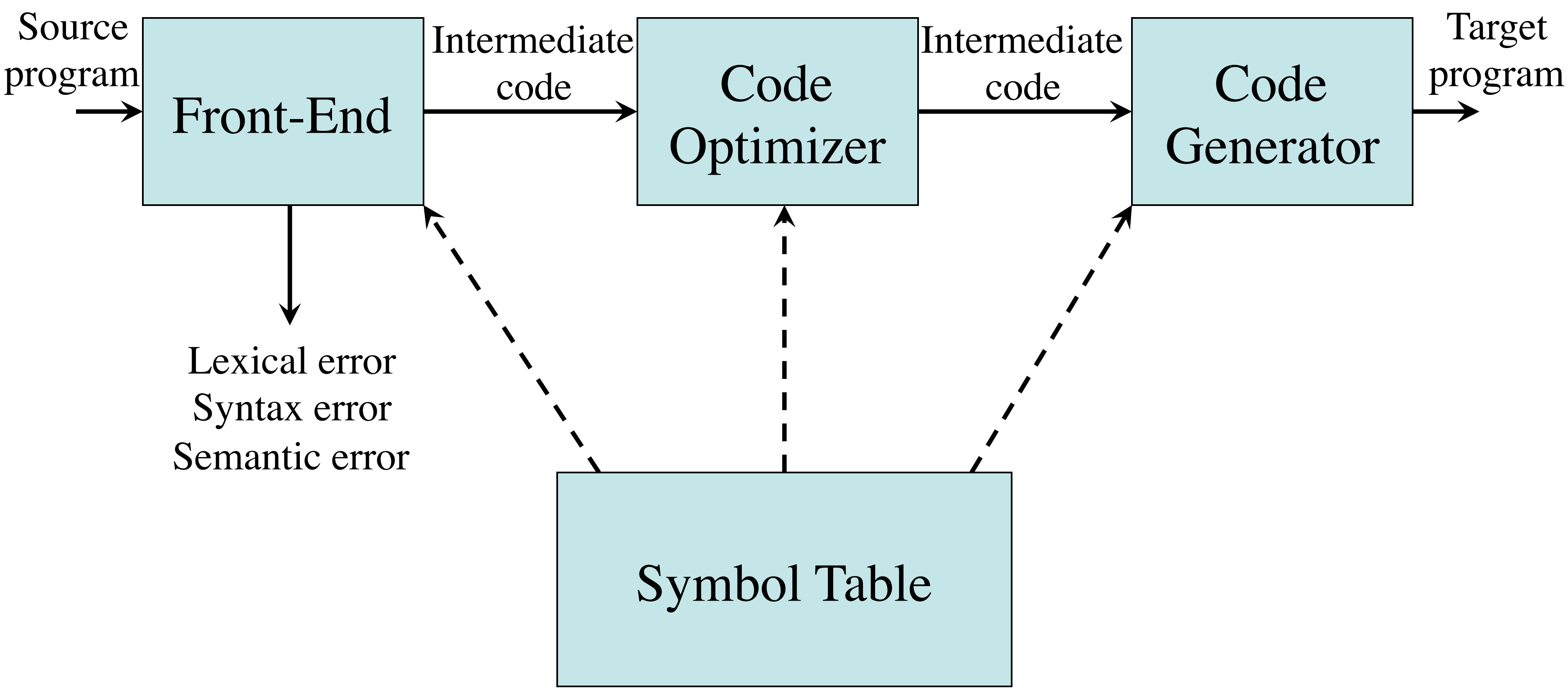
Position of a Code Generator in the Compiler Model



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*(**`R`**)	*c*+*contents*(**`R`**)	1
Indirect register	**`*R`**	*contents*(**`R`**)	0
Indirect indexed	**`*`***c*(**`R`**)	*contents*(*c*+*contents*(**`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 + cost(*source*-mode) + cost(*destination*-mode)

Examples

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

{ R1=t }

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

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

{ R0=a, R1=t }

```
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

```
a+b-(c+d)*e
```

```
t1:=a+b
t2:=c+d
t3:=e*t2
t4:=t1-t3
```

```
MOV a,R0
ADD b,R0
MOV R0,t1
MOV c,R1
ADD d,R1
MOV e,R0
MUL R1,R0
MOV t1,R1
SUB R0,R1
MOV R1,t4
```

reorder

```
t2:=c+d
t3:=e*t2
t1:=a+b
t4:=t1-t3
```

```
MOV c,R0
ADD d,R0
MOV e,R1
MUL R0,R1
MOV a,R0
ADD b,R0
SUB R1,R0
MOV R0,t4
```

Generating Code for Stack Allocation of Activation Records

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

                       500: …
                       564: MOV R0,12(SP)      Store return value
                       572: GOTO *SP           Return to caller
```

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