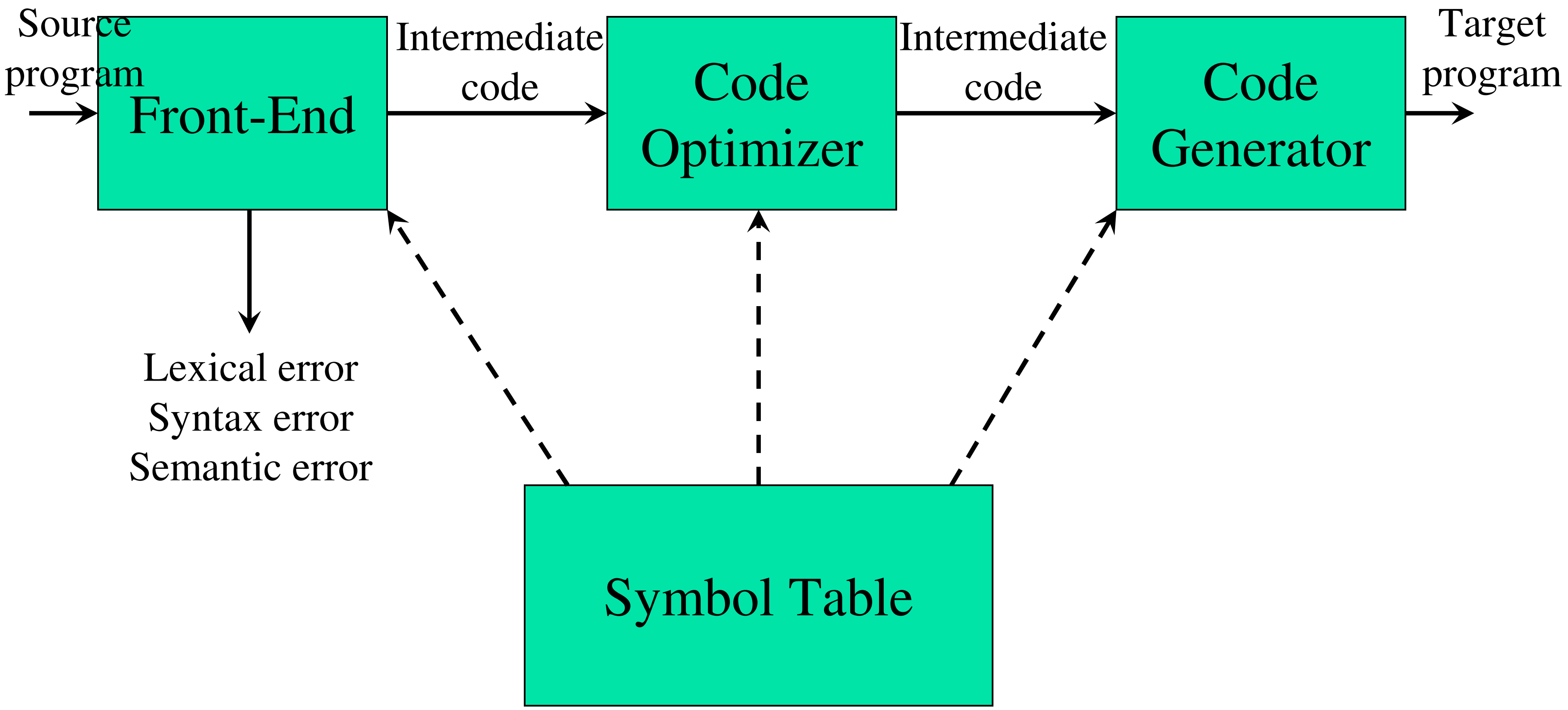
Code Generation

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
- Code generator must run efficiently

Issues in the Design of Code Generator

- Input to the code Generator
- Target Program
- Memory management
- Instruction Selection
- Register Allocation
- Choice of Evaluation Order.

Input to the code Generator

Input to the code generator in an intermediate may be of several forms:

- Linear representation-postfix notation
- three-address representation-quadruples
- Virtual machine representation- stack machine code
- Graphical representation – syntax tree, dag

Target Program

1. Absolute machine language
2. Relocatable machine language
3. Assembly language

Memory management

- Both front end and code generator performs the task of mapping the names in the source program to addresses to the data objects in run time memory.

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
```

Instruction Selection Cont…

- 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

a=a + 1

```
MOV a,R0
 Add #1,R0                  INC A
 Mov R0, a
```

Register Allocation

- 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

Register Allocation Cont…

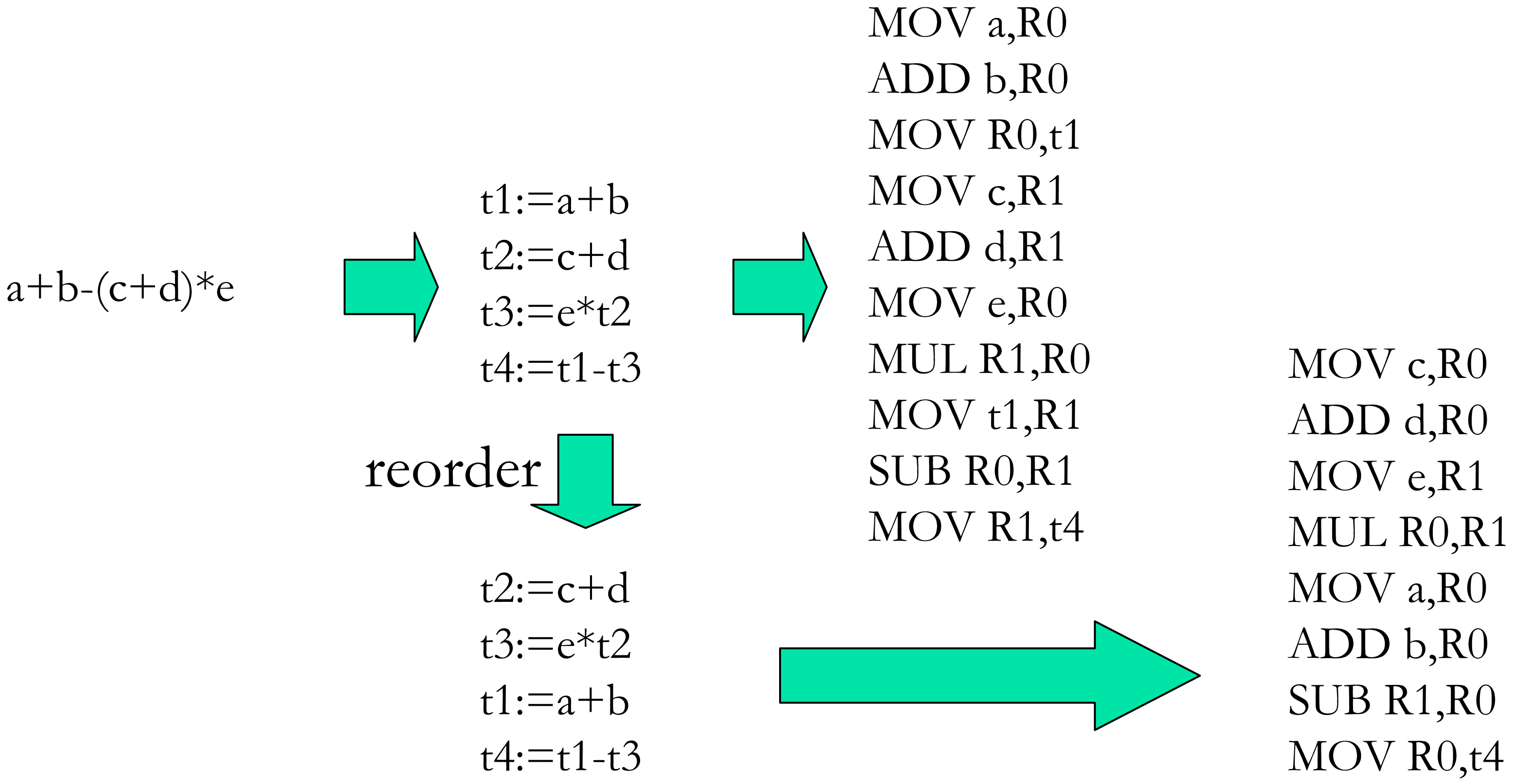
```
t:=a*b
t:=t+a
t:=t/d
```

⇓ { R1=t }

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

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
```

Target Machine

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

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

- Define the cost of instruction
  = 1 + cost(*source*-mode) + cost(*destination*-mode)

Source and destination having registers : cost =0

Source and destination having mem.loc. : cost =1

Source and destination having literals :cost =1

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