Language Processing Systems

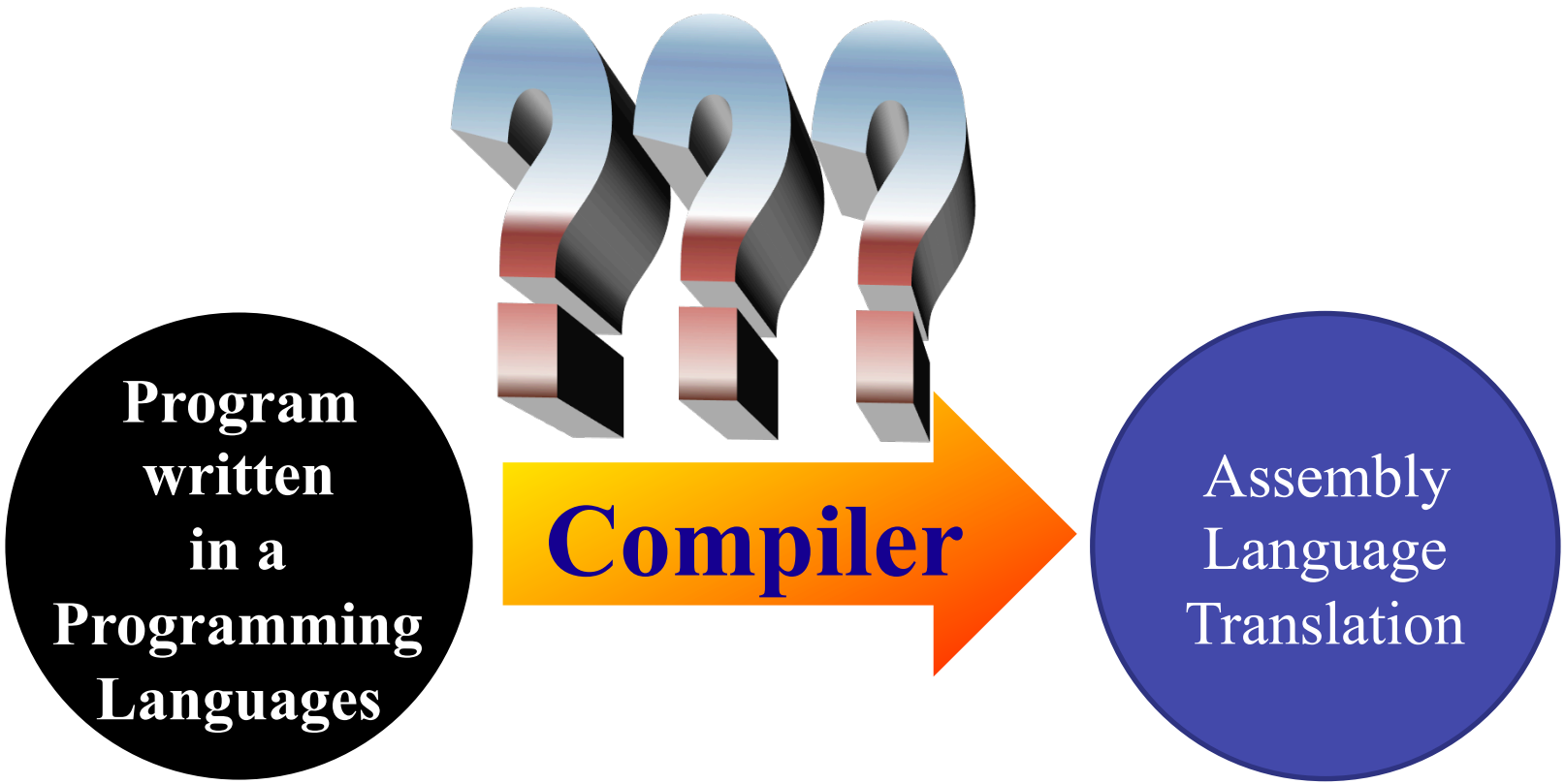
Prof. Mohamed Hamada

Software Engineering Lab.
The University of Aizu
Japan

Today's Outline

- Anatomy of a compiler
- Compiler front-end and back-end
- Regular expressions

Anatomy of a Compiler

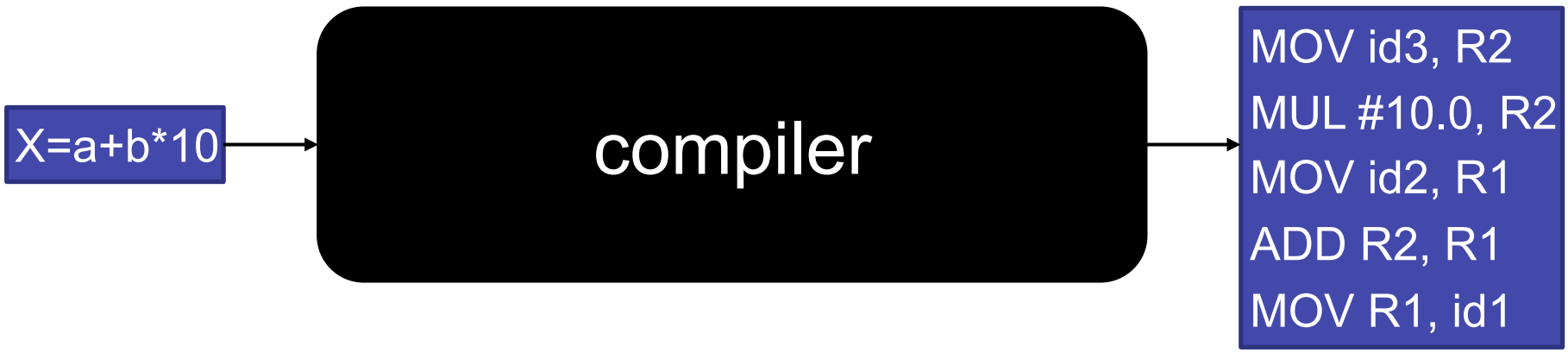


What is a compiler?

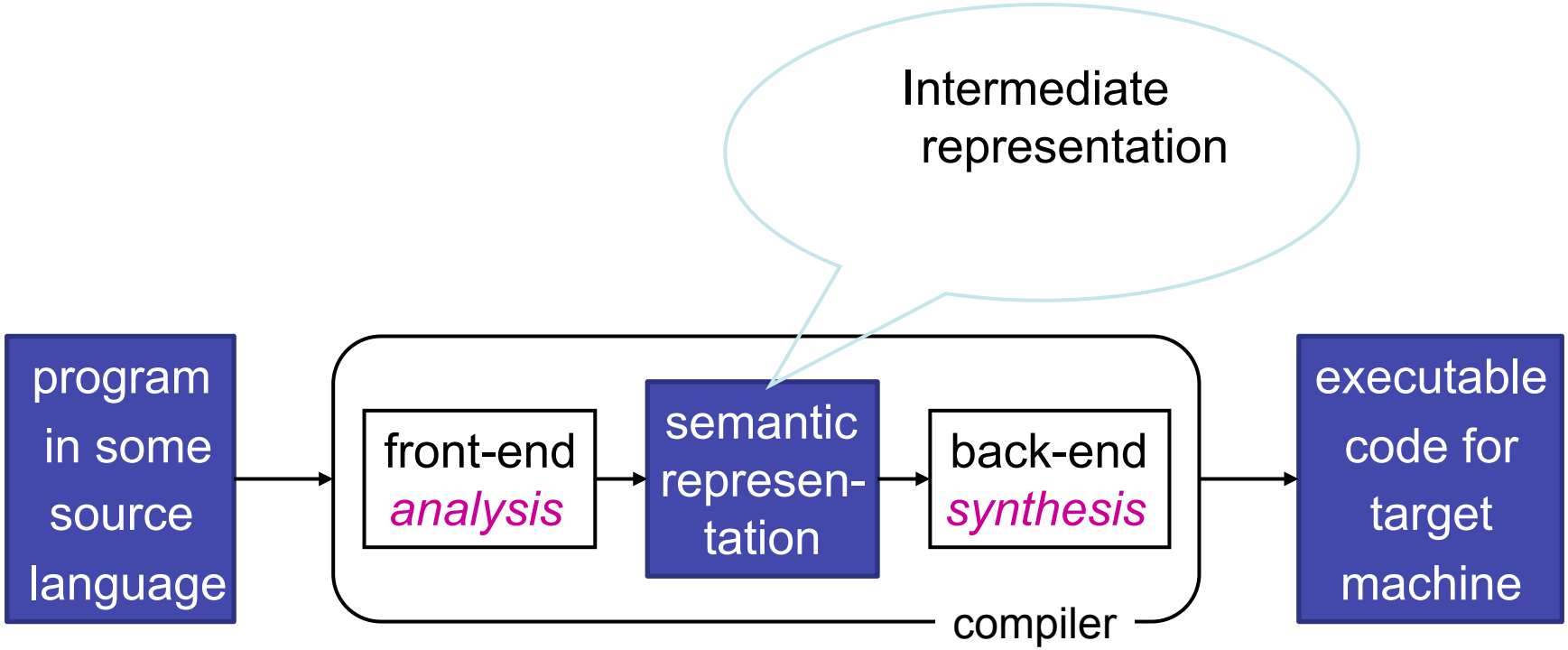
A compiler is a program that reads a program written in one language and translates it into another language.

Traditionally, compilers go from high-level languages to low-level languages.

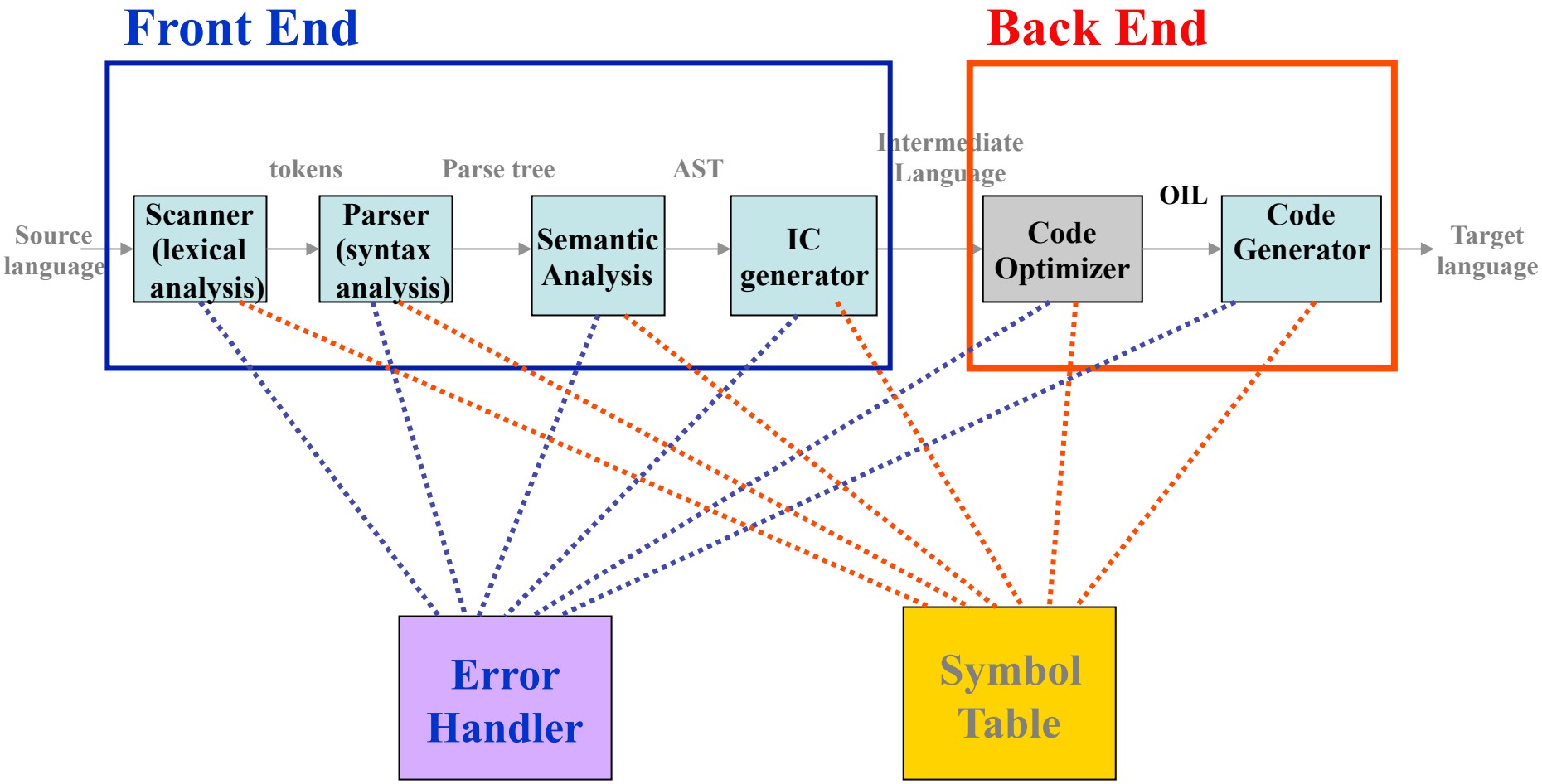
Example


X=a+b*10
compiler
MOV id3, R2
MUL #10.0, R2
MOV id2, R1
ADD R2, R1
MOV R1, id1

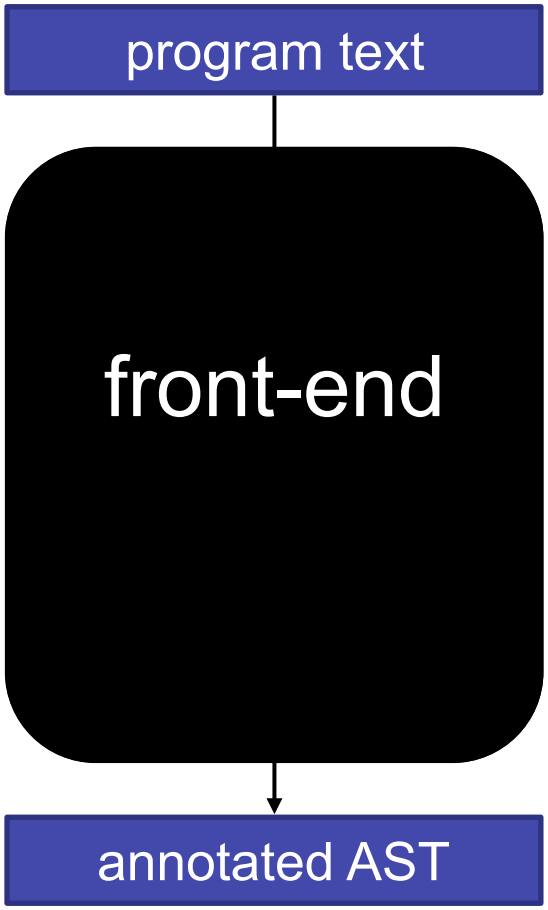
What is a compiler?



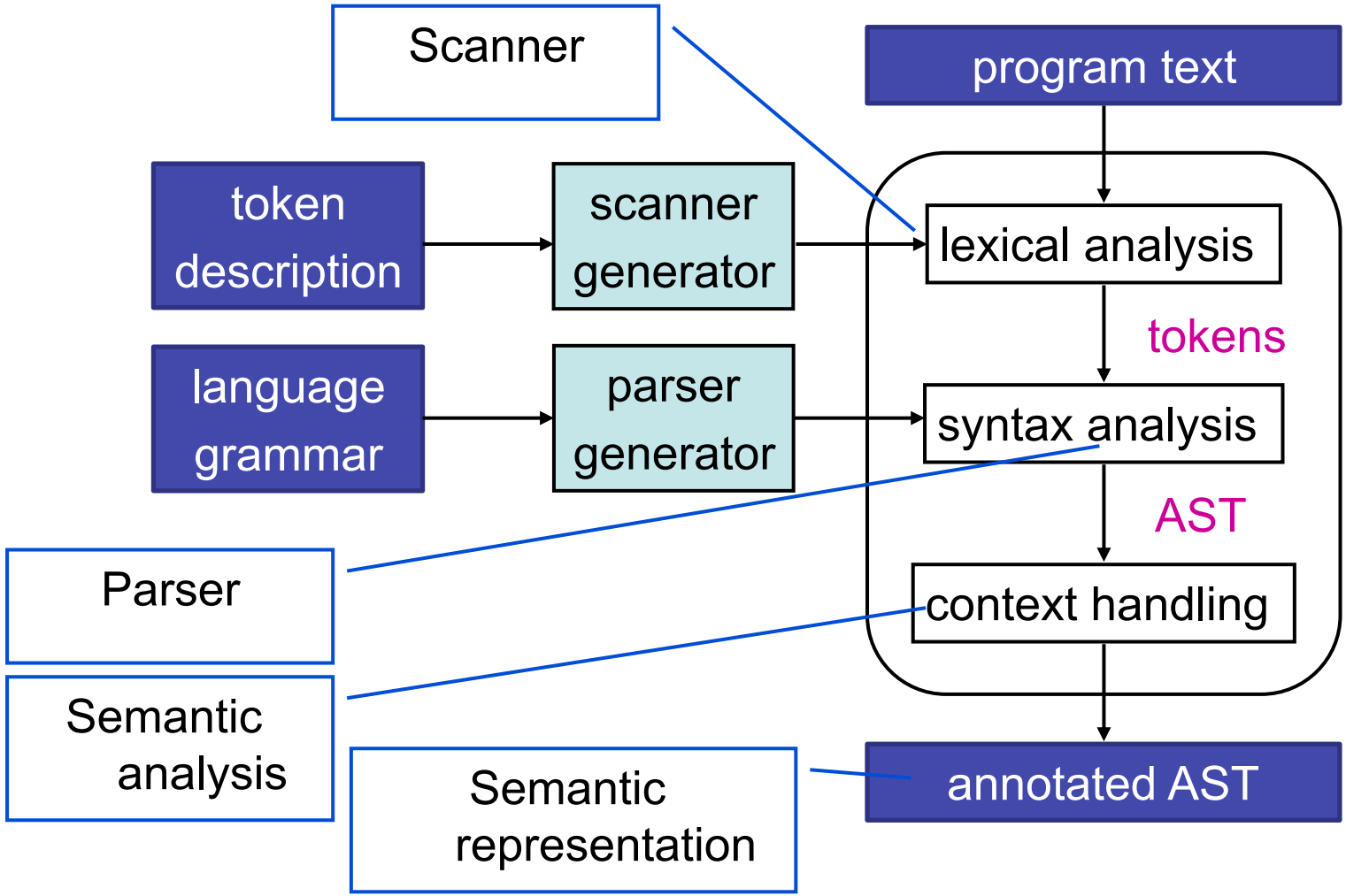
Compiler Architecture



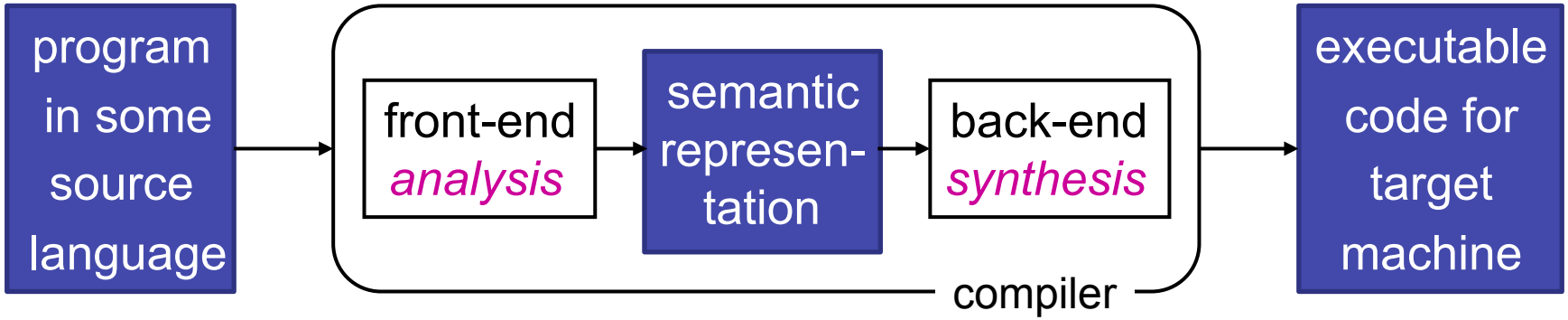
front-end:
from program text to AST



front-end: from program text to AST



Semantic representation



- heart of the compiler
- intermediate code
  - linked lists of pseudo instructions
  - abstract syntax tree (AST)

AST example

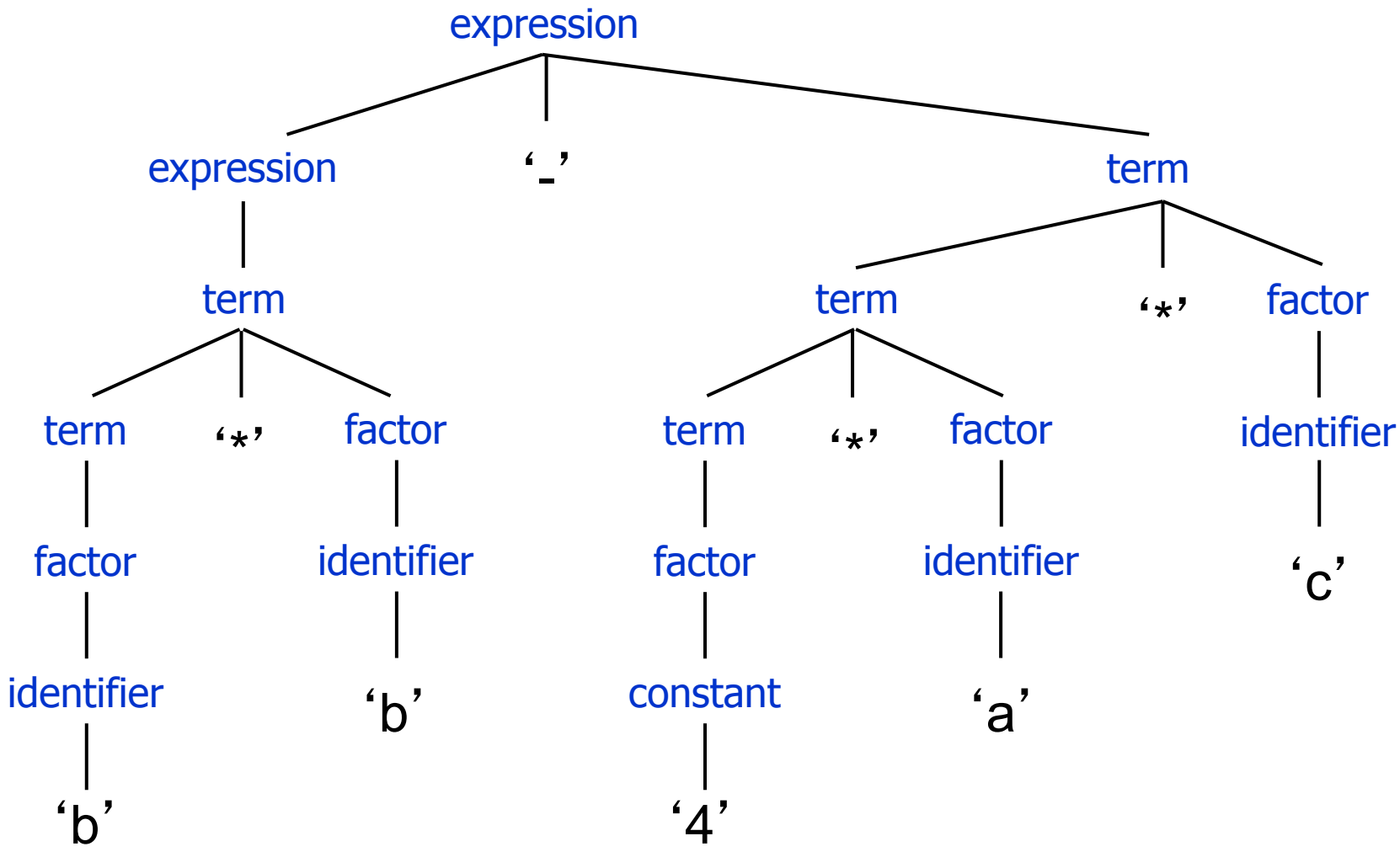
- expression grammar

  expression → expression ‘+’ term | expression ‘-’ term | term
  term → term ‘*’ factor | term ‘/’ factor | factor
  factor → identifier | constant | ‘(‘ expression ‘)’

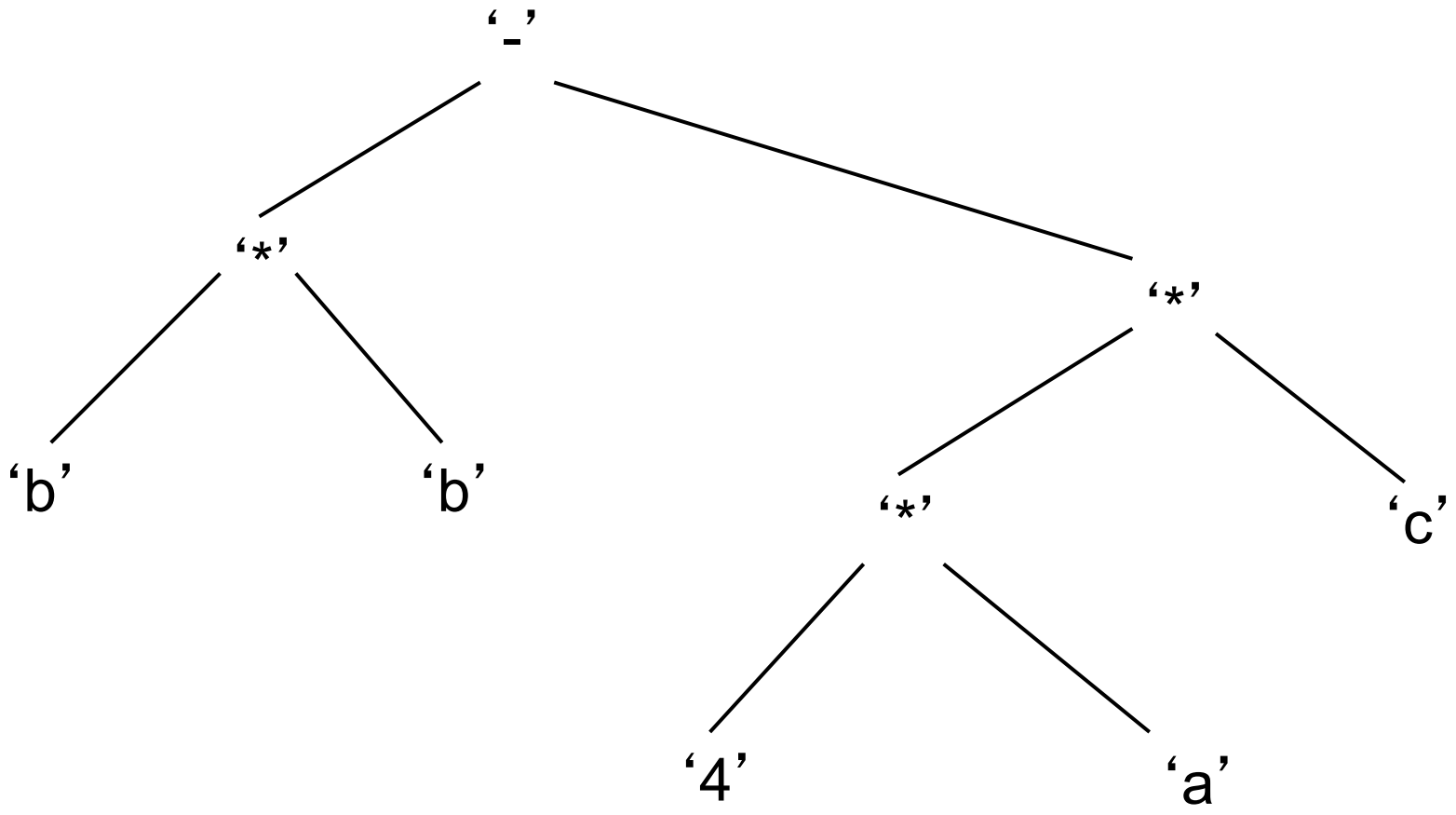
- example expression

  b*b – 4*a*c

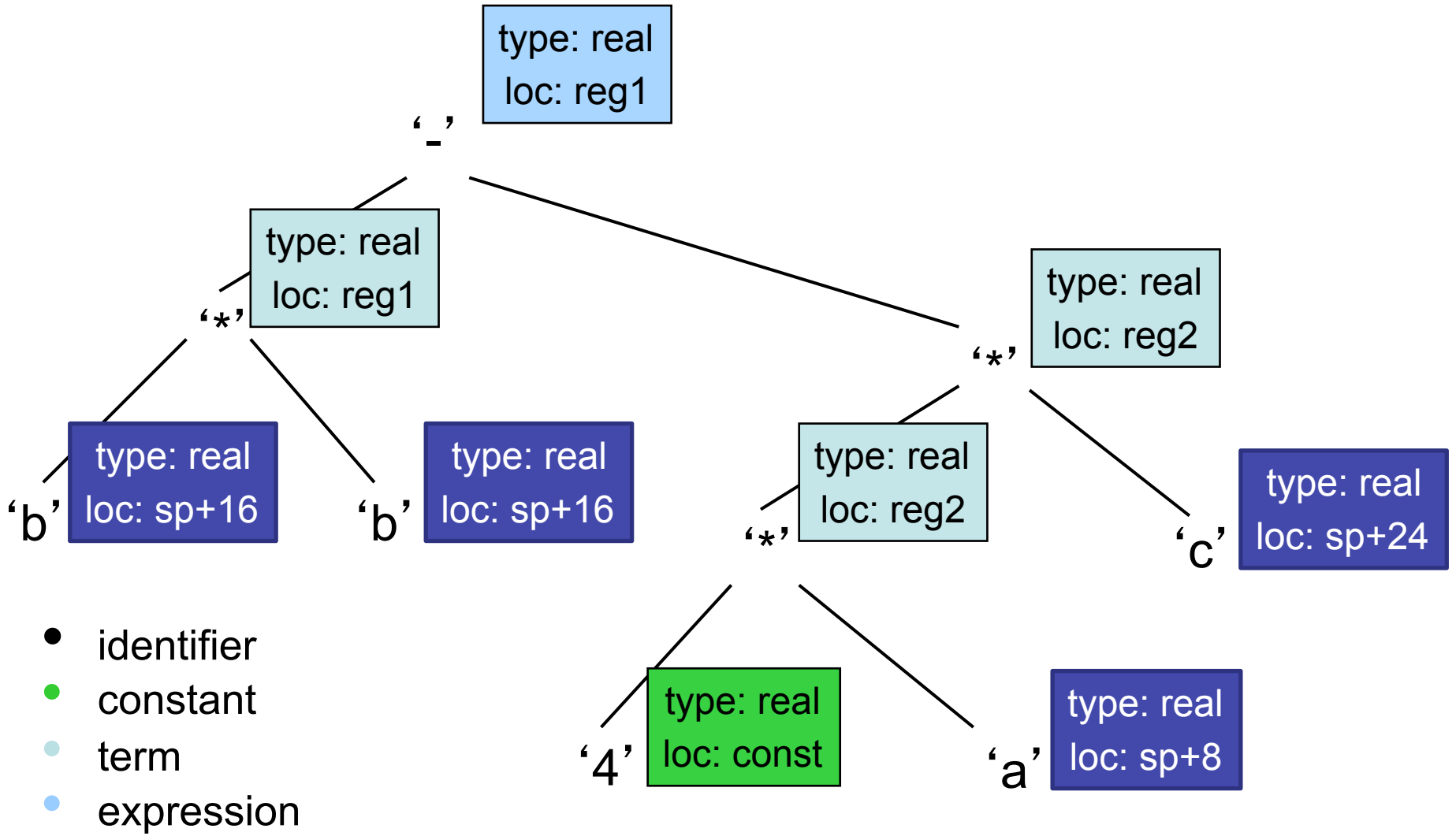
parse tree: b*b – 4*a*c



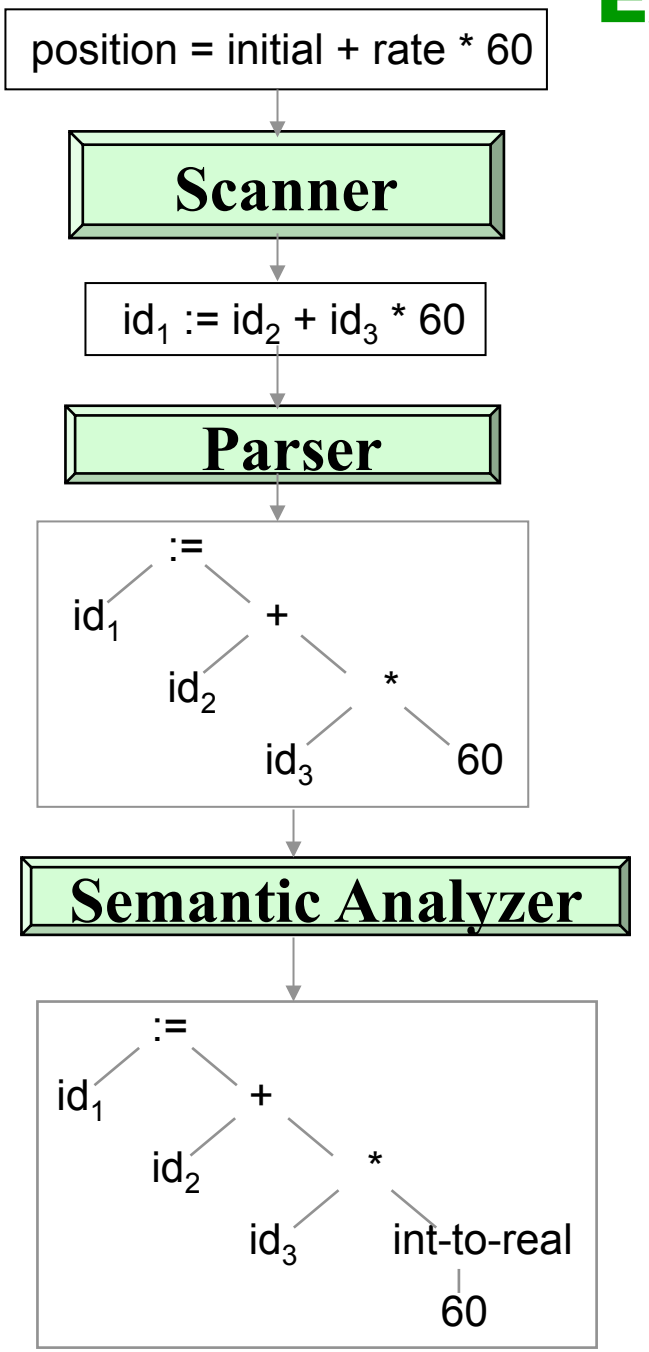
AST: b*b – 4*a*c



annotated AST: b*b – 4*a*c



Example


position = initial + rate * 60
Scanner
id1 := id2 + id3 * 60
Parser
:=
id1
+
id2
*
id3
60
Semantic Analyzer
:=
id1
+
id2
*
id3
int-to-real
60

AST exercise

- expression grammar

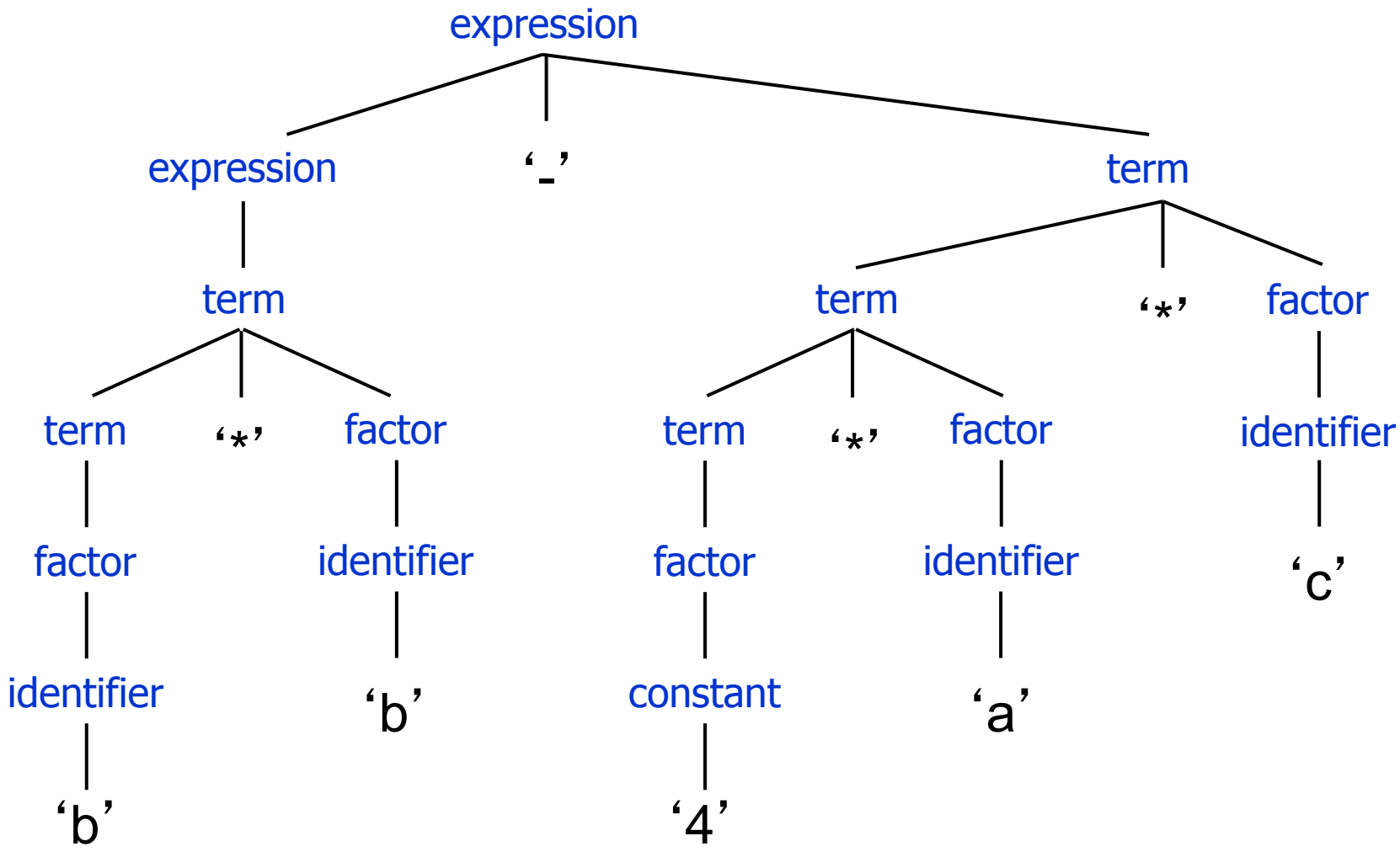
  expression → expression ‘+’ term | expression ‘-’ term | term
  term → term ‘*’ factor | term ‘/’ factor | factor
  factor → identifier | constant | ‘(‘ expression ‘)’

- example expression

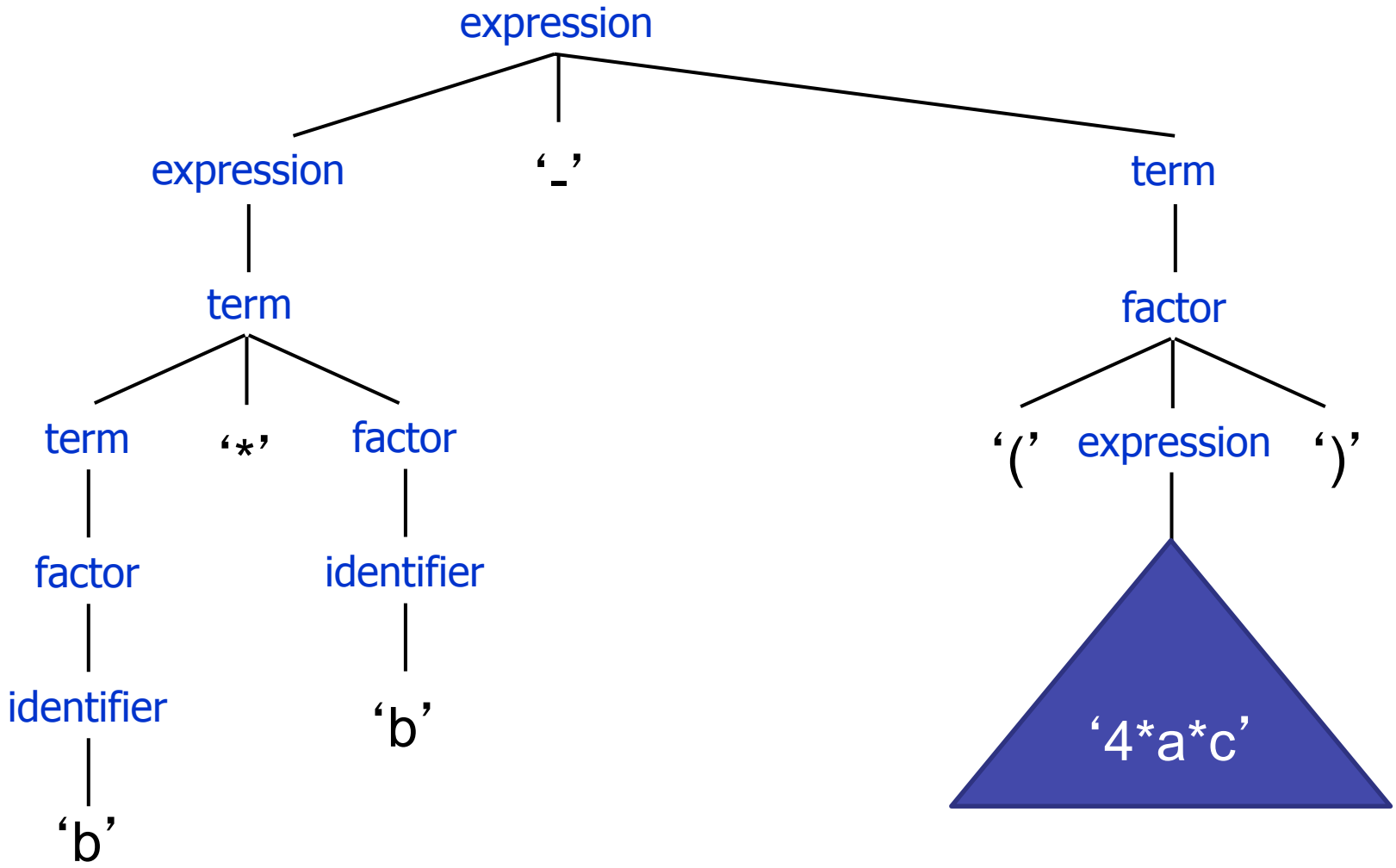
  b*b – (4*a*c)

- draw parse tree and AST

answer
parse tree: b*b – 4*a*c



answer
parse tree: b*b – (4*a*c)

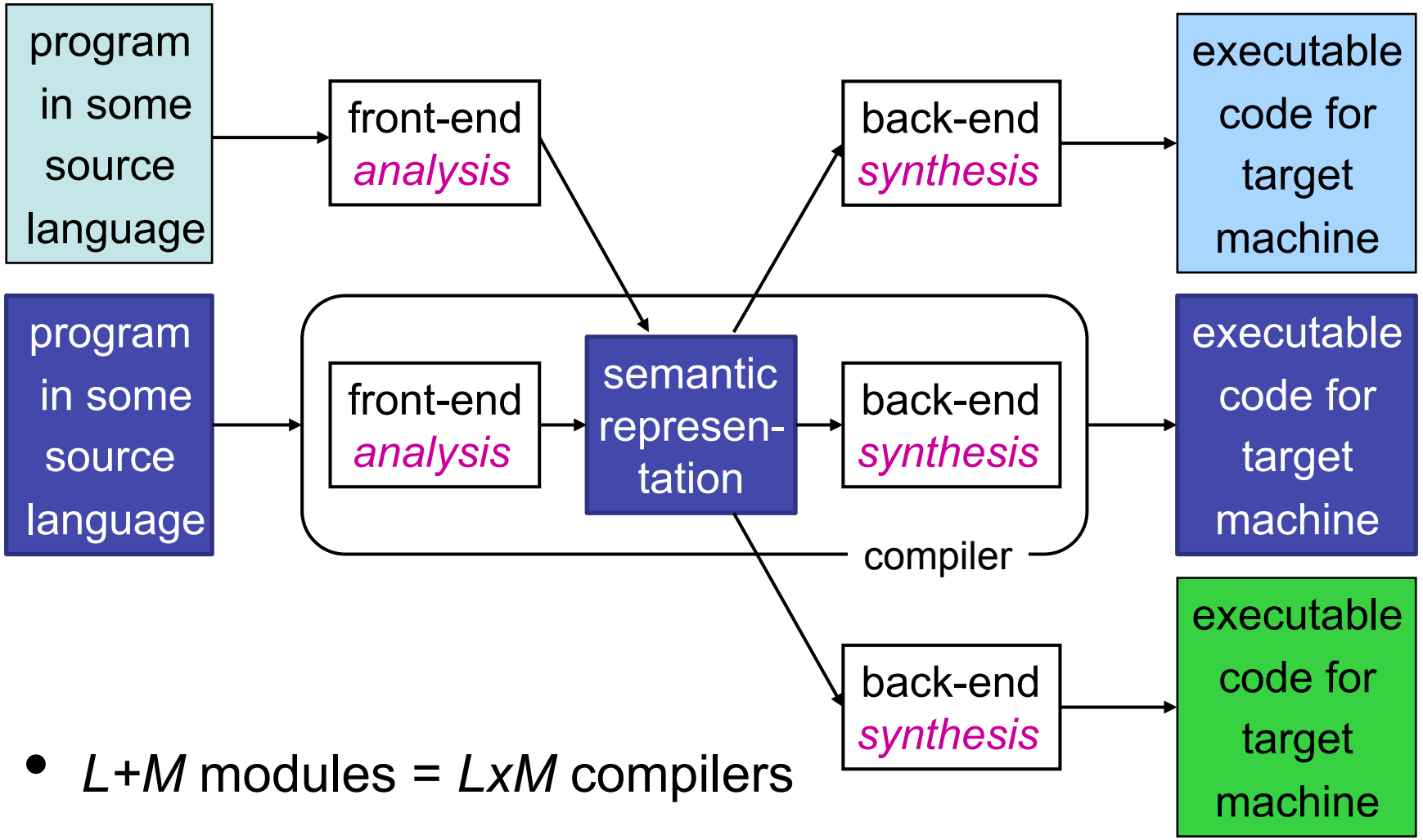


Advantages of Using Front-end and Back-end

1. **Retargeting** - Build a compiler for a new machine by attaching a new code generator to an existing front-end.
2. **Optimization** - reuse intermediate code optimizers in compilers for different languages and different machines.

Note: the terms "intermediate code", "intermediate language", and "intermediate representation" are all used interchangeably.

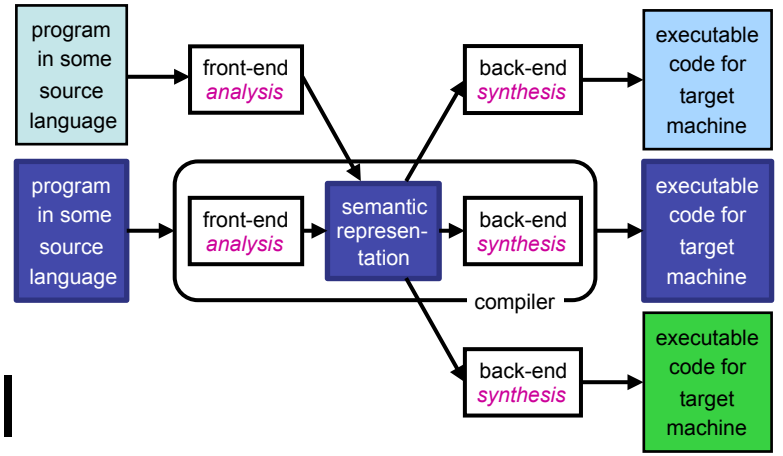
Compiler structure



- *L*+*M* modules = *LxM* compilers

Limitations of modular approach

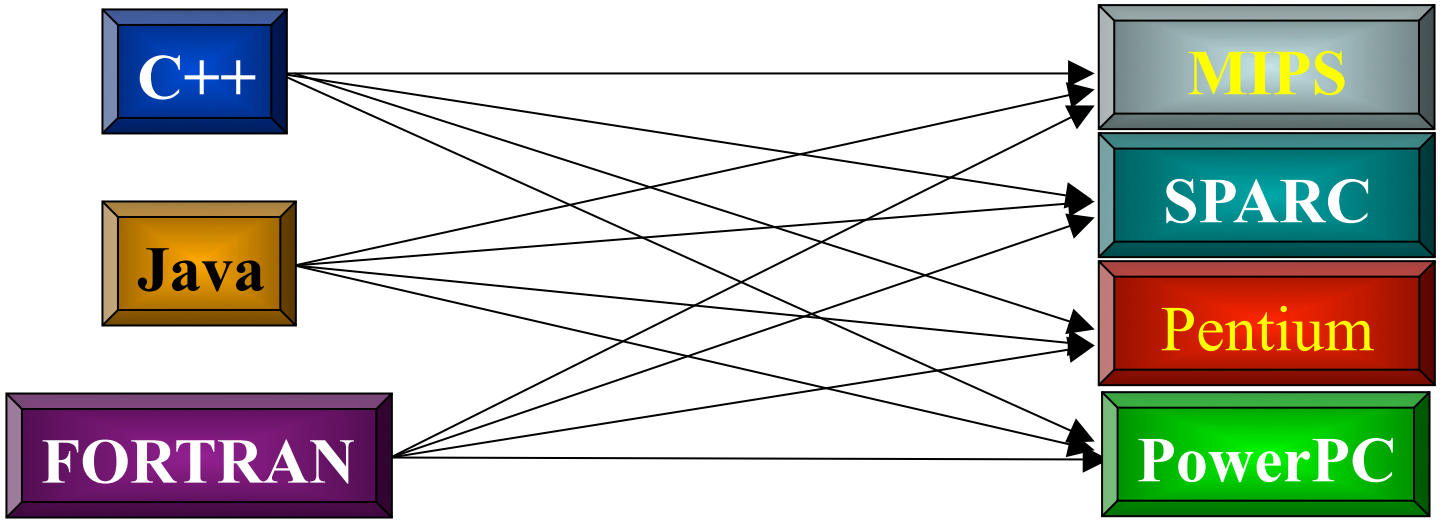
- performance
  - generic vs specific
  - loss of information



- variations must be small
  - same programming paradigm
  - similar processor architecture

Front-end and Back-end

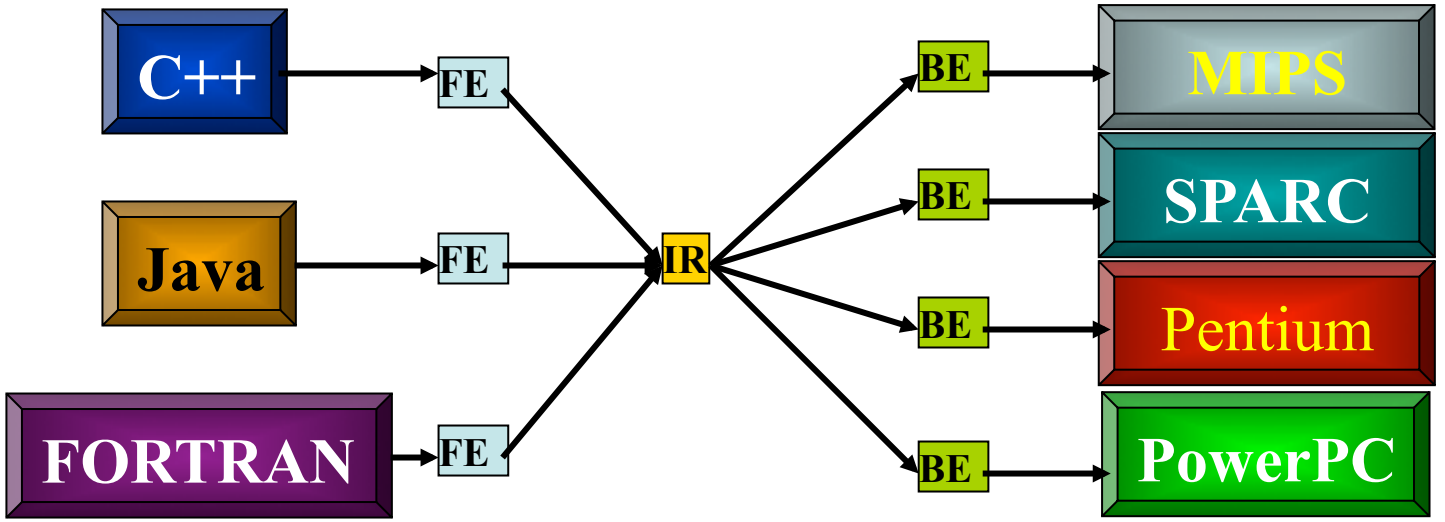
- Suppose you want to write **3** compilers to **4** computer platforms:



We need to write 12 programs

Front-end and Back-end

- But we can do it better

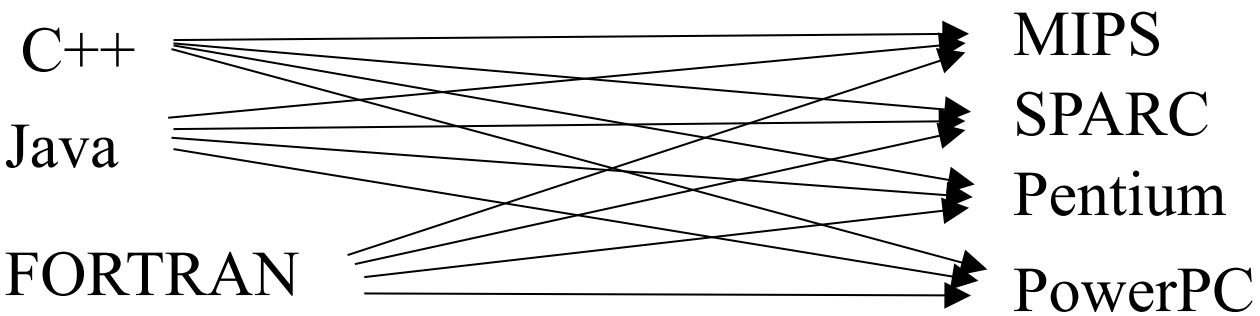


We need to write 7 programs only

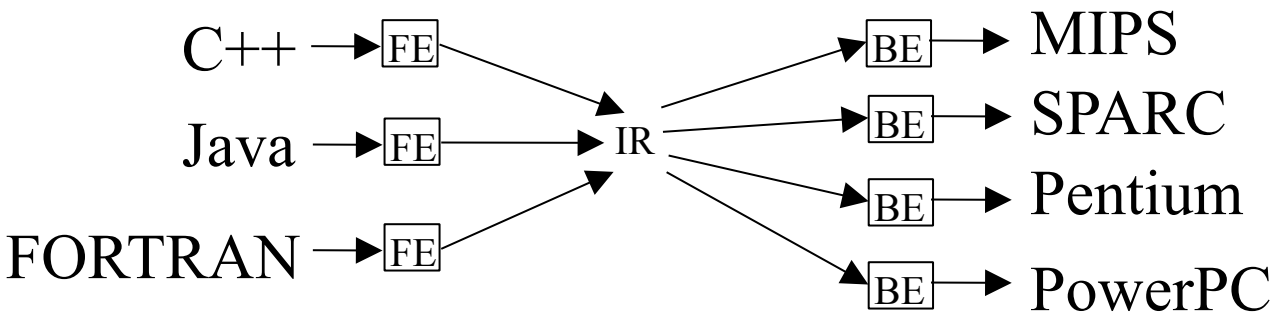
- **IR: Intermediate Representation**
- **FE: Front-End**
- **BE: Back-End**

Front-end and Back-end

- Suppose you want to write compilers from `m` source languages to `n` computer platforms. A naïve solution requires `n*m` programs:



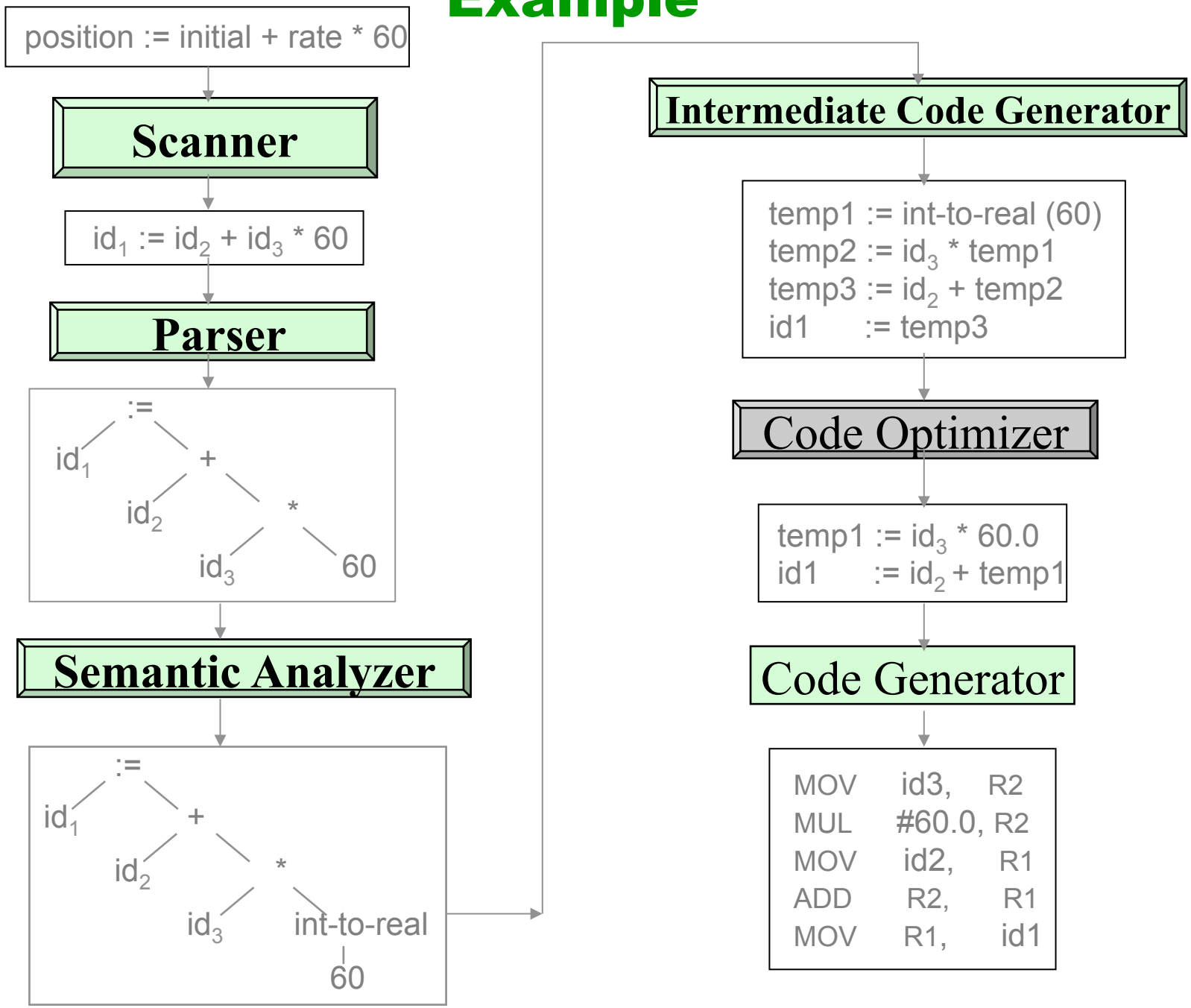
- but we can do it with `n+m` programs:



  - IR: Intermediate Representation
  - FE: Front-End
  - BE: Back-End

Compiler Example

Example


position := initial + rate * 60
Scanner
id1 := id2 + id3 * 60
Parser
:=
id1
+
id2
*
id3
60
Semantic Analyzer
:=
id1
+
id2
*
id3
int-to-real
60
Intermediate Code Generator
temp1 := int-to-real (60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
Code Optimizer
temp1 := id3 * 60.0
id1 := id2 + temp1
Code Generator
MOV id3, R2
MUL #60.0, R2
MOV id2, R1
ADD R2, R1
MOV R1, id1

Regular Expressions

A regular expression is built up out of simpler regular expressions using a set of defining rules.

Empty Set:

Φ A regular expression formed by **Empty set**.

Lambda:

λ A regular expression formed by Empty string.

Symbol:

a A regular expression formed by **a**.

Alternation:

M | N A regular expression formed by **M** or **N**.

Concatenation:

(M • N) A regular expression formed by **M** followed by **N**.

Repetition:

(M*) A regular expression formed by zero or more repetitions of **M**.

Regular Expressions

Operators Precedence:

() > * > • > |

This can simplify regular expressions.

Example:

(a) | ((b) * (c)) can be written as: a | b * c.

Language:

The language denoted by a regular expression r will be expressed as L(r)

Regular expressions allows us to define tokens of programming Languages such as identifiers and numbers.

Regular Expressions

Examples:

1. **a* is a regular expression denotes the set {λ,a,aa,…}**

2. **a|b is a regular expression denotes the set {a}U{b}**

3. **a*|b is a regular expression denotes the set {λ,a,aa,…}U{b}**

4. **a*b is a regular expression denotes the set {b,ab,aab,…}**

Match and Create the Regular Expressions

1. 0(0|1)*0
2. ((λ|0)1*)*
3. ((0|1)0(0|1))*

a. 000000
b. 01010
c. 010101
d. 101010
e. 001100

- All strings of 0's and 1's that does not contain the substring 011

Match and Create the Regular Expressions

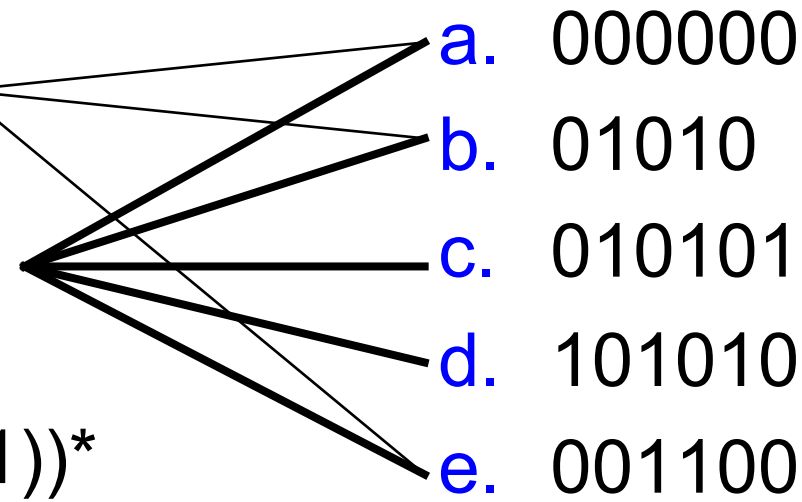
1. 0(0|1)*0
2. ((λ|0)1*)*
3. ((0|1)0(0|1))*

a. 000000
b. 01010
c. 010101
d. 101010
e. 001100

- All strings of 0's and 1's that does not contain the substring 011

Match and Create the Regular Expressions

1. 0(0|1)*0
2. ((λ|0)1*)*
3. ((0|1)0(0|1))*



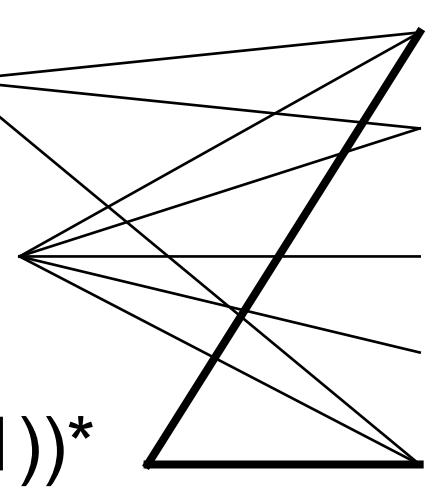
a. 000000
b. 01010
c. 010101
d. 101010
e. 001100

- All strings of 0's and 1's that does not contain the substring 011

Match and Create the Regular Expressions

1. 0(0|1)*0
2. ((λ|0)1*)*
3. ((0|1)0(0|1))*

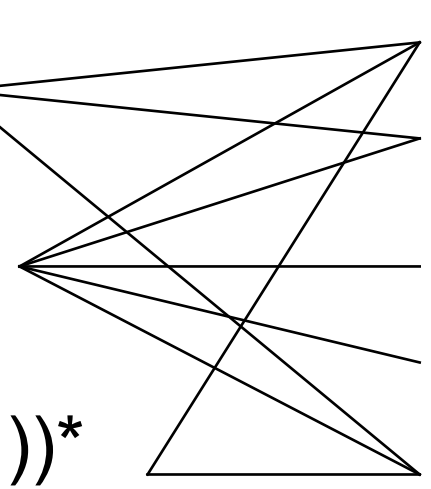
a. 000000
b. 01010
c. 010101
d. 101010
e. 001100



- All strings of 0's and 1's that does not contain the substring 011

Match and Create the Regular Expressions

1. 0(0|1)*0
2. ((λ|0)1*)*
3. ((0|1)0(0|1))*



a. 000000
b. 01010
c. 010101
d. 101010
e. 001100

- All strings of 0's and 1's that does not contain the substring 011
  - 1*((010)*0*)*(λ|1)

END