Language Processing Systems

Evaluation

- Active sheets 10 %
- Exercise reports 30 %
- Midterm Exam 20 %
- Final Exam 40 %

Contact

- Send e-mail to

  hamada@u-aizu.ac.jp

- Course materials at

  www.u-aizu.ac.jp/~hamada/education.html

Check every week for update

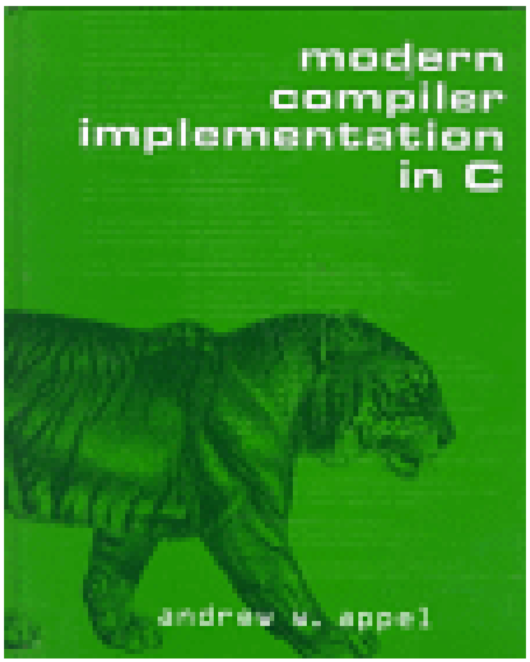
Books

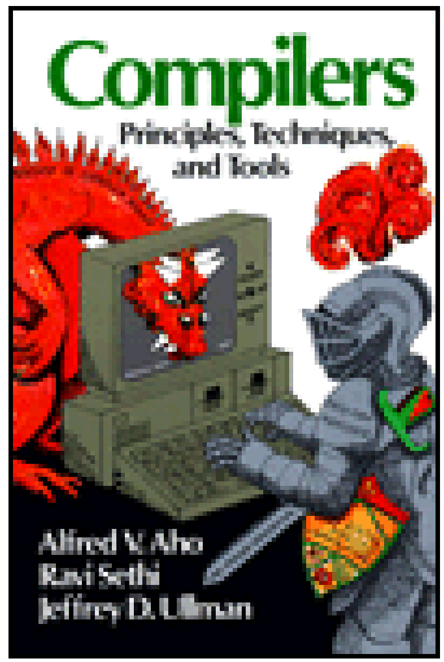
Andrew W. Appel : ***Modern Compiler Implementation in C***

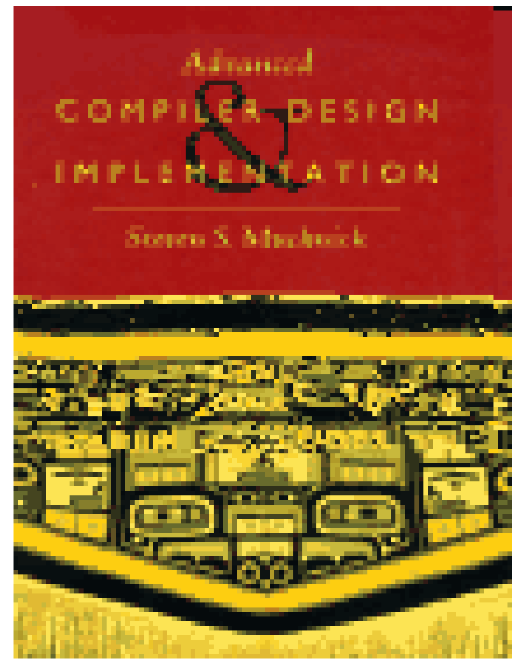
A. Aho, R. Sethi and J. Ullman, ***Compilers: Principles, Techniques and Tools*** *(*The Dragon Book*)*, Addison Wesley

S. Muchnick, ***Advanced Compiler Design and Implementation***, Morgan Kaufman, 1997

Books


modern
compiler
implementation
in C
andrew w. appel


Compilers
Principles, Techniques, and Tools
Alfred V. Aho
Ravi Sethi
Jeffrey D. Ullman


Advanced
COMPILER DESIGN
&
IMPLEMENTATION
Steven S. Muchnick

Goals

- understand the structure of a compiler
- understand how the components operate
- understand the tools involved
  - scanner generator, parser generator, etc.

- understanding means
  - [theory] be able to read source code
  - [practice] be able to adapt/write source code

The Course covers:

- Introduction
- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Intermediate Code Generation
- Code Generation
- Code Optimization (if there is time)

Related to Compilers

- Interpreters (direct execution)
- Assemblers
- Preprocessors
- Text formatters (non-WYSIWYG)
- Analysis tools

Today's Outline

- Introduction to Language Processing Systems
  - Why do we need a compiler?
  - What are compilers?
  - Anatomy of a compiler

Why study compilers?

- Better understanding of programming language concepts
- Wide applicability
  - Transforming “data” is very common
  - Many useful data structures and algorithms
- Bring together:
  - Data structures & Algorithms
  - Formal Languages
  - Computer Architecture
- Influence:
  - Language Design
  - Architecture (influence is bi-directional)

Issues Driving Compiler Design

- Correctness
- Speed (runtime and compile time)
  - Degrees of optimization
  - Multiple passes
- Space
- Feedback to user
- Debugging

Why Study Compilers?

- Compilers enable programming at a high level language instead of machine instructions.
  - Malleability, Portability, Modularity, Simplicity, Programmer Productivity
  - Also Efficiency and Performance

Compilers Construction touches many topics in Computer Science

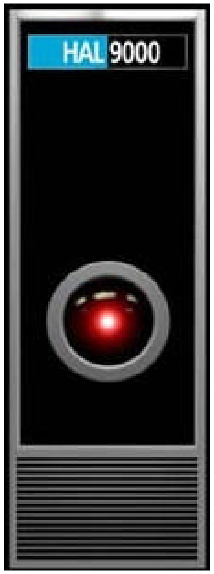
- Theory
  - Finite State Automata, Grammars and Parsing, data-flow
- Algorithms
  - Graph manipulation, dynamic programming
- Data structures
  - Symbol tables, abstract syntax trees
- Systems
  - Allocation and naming, multi-pass systems, compiler construction
- Computer Architecture
  - Memory hierarchy, instruction selection, interlocks and latencies
- Security
  - Detection of and Protection against vulnerabilities
- Software Engineering
  - Software development environments, debugging
- Artificial Intelligence
  - Heuristic based search

Power of a Language

- Can use to describe any action
  - Not tied to a “context”
- Many ways to describe the same action
  - Flexible

How to instruct a computer

- How about natural languages?
  - English??
  - “Open the pod bay doors, Hal.”
  - “I am sorry Dave, I am afraid I cannot do that”
  - We are not there yet!!

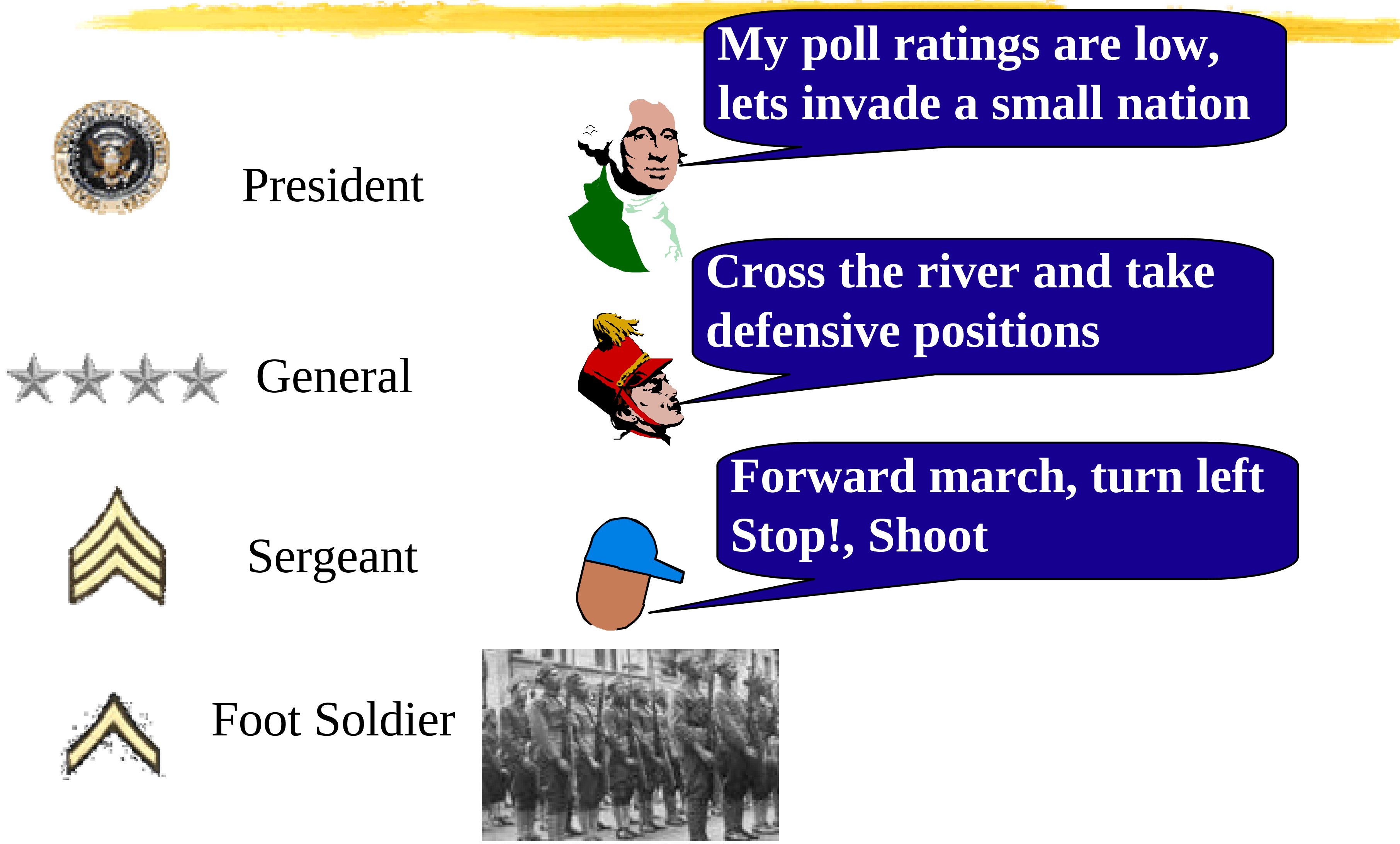


- Natural Languages:
  - Powerful, but…
  - Ambiguous
    - Same expression describes many possible actions

Programming Languages

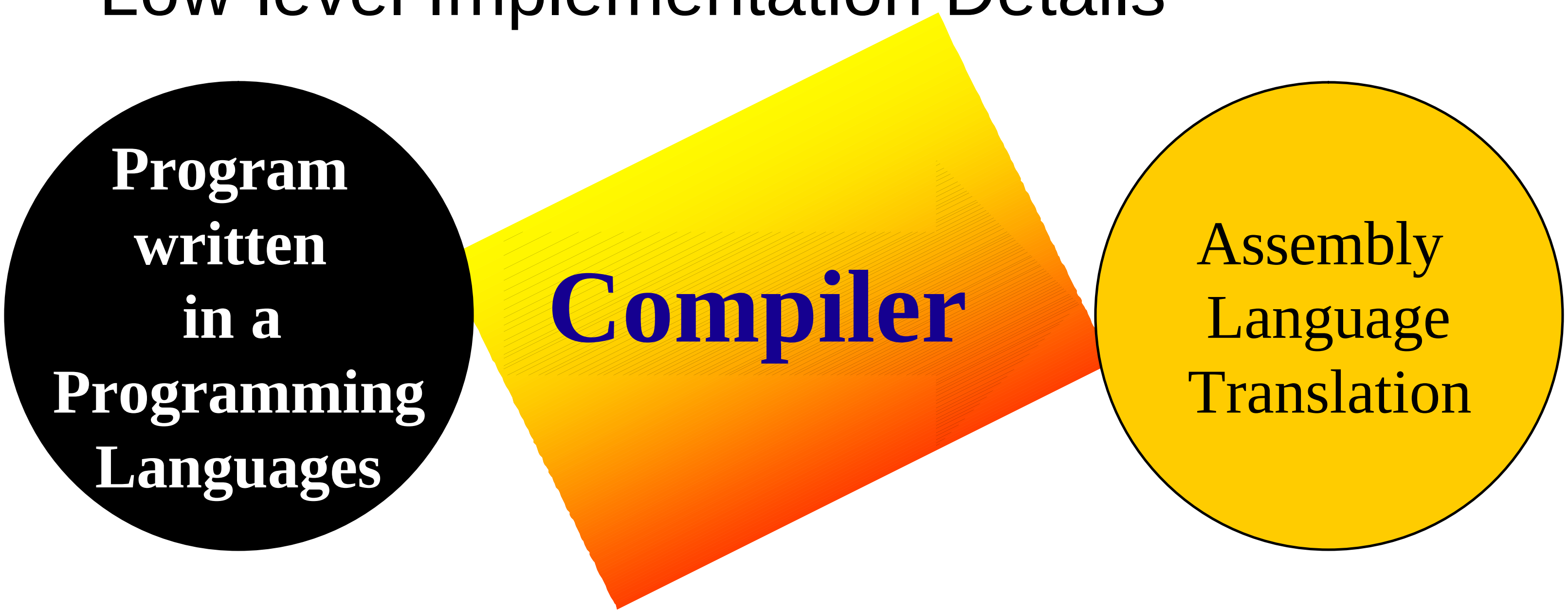
- Properties
  - need to be precise
  - need to be concise
  - need to be expressive
  - need to be at a high-level (lot of abstractions)

High-level Abstract Description to Low-level Implementation Details



1. How to instruct the computer

- Write a program using a programming language
  - – High-level Abstract Description
- Microprocessors talk in assembly language
  - Low-level Implementation Details



1. How to instruct the computer

- Input: High-level programming language
- Output: Low-level assembly instructions
- Compiler does the translation:
  - Read and understand the program
  - Precisely determine what actions it require
  - Figure-out how to faithfully carry-out those actions
  - Instruct the computer to carry out those actions

Input to the Compiler

- Standard imperative language (Java, C, C++)
  - State
    - Variables,
    - Structures,
    - Arrays
  - Computation
    - Expressions (arithmetic, logical, etc.)
    - Assignment statements
    - Control flow (conditionals, loops)
    - Procedures

Output of the Compiler

- State
  - Registers
  - Memory with Flat Address Space
- Machine code – load/store architecture
  - Load, store instructions
  - Arithmetic, logical operations on registers
  - Branch instructions

Example (input program)

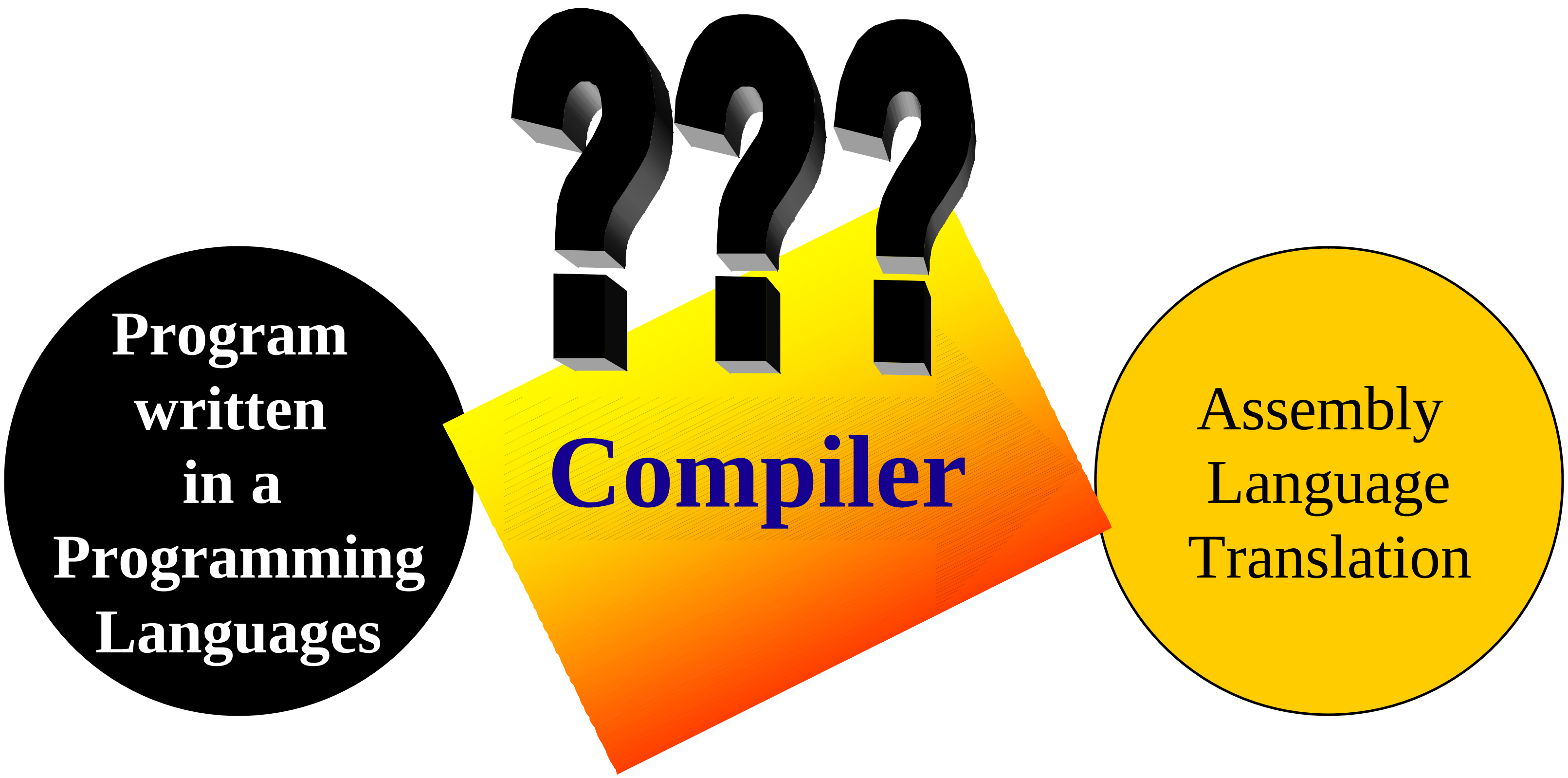
```
int sumcalc(int a, int b, int N)
{
    int i, x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
       x = x + (4*a/b)*i + (i+1)*(i+1);
      x = x + b*y;
    }
    return x;
}
```

Example (Output assembly code)

```
sumcalc:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -4(%rbp)
        movl    %esi, -8(%rbp)
        movl    %edx, -12(%rbp)
        movl    $0, -20(%rbp)
        movl    $0, -24(%rbp)
        movl    $0, -16(%rbp)
.L2:    movl    -16(%rbp), %eax
        cmpl    -12(%rbp), %eax
        jg      .L3
        movl    -4(%rbp), %eax
        leal    0(,%rax,4), %edx
        leaq    -8(%rbp), %rax
        movq    %rax, -40(%rbp)
        movl    %edx, %eax
        movq    -40(%rbp), %rcx
        cltd
        idivl   (%rcx)
        movl    %eax, -28(%rbp)
        movl    -28(%rbp), %edx
        imull   -16(%rbp), %edx
        movl    -16(%rbp), %eax
        incl    %eax
        imull   %eax, %eax
        addl    %eax, %edx
        leaq    -20(%rbp), %rax
        addl    %edx, (%rax)
        movl    -8(%rbp), %eax
        movl    %eax, %edx
        imull   -24(%rbp), %edx
        leaq    -20(%rbp), %rax
        addl    %edx, (%rax)
        leaq    -16(%rbp), %rax
        incl    (%rax)
        jmp     .L2
.L3:    movl    -20(%rbp), %eax
        leave
        ret
```

```
 .size   sumcalc, .-sumcalc
        .section
.Lframe1:
        .long   .LECIE1-.LSCIE1
.LSCIE1:.long   0x0
        .byte   0x1
        .string ""
        .uleb128 0x1
        .sleb128 -8
        .byte   0x10
        .byte   0xc
        .uleb128 0x7
        .uleb128 0x8
        .byte   0x90
        .uleb128 0x1
        .align 8
.LECIE1:.long   .LEFDE1-.LASFDE1
        .long   .LASFDE1-.Lframe1
        .quad   .LFB2
        .quad   .LFE2-.LFB2
        .byte   0x4
        .long   .LCFI0-.LFB2
        .byte   0xe
        .uleb128 0x10
        .byte   0x86
        .uleb128 0x2
        .byte   0x4
        .long   .LCFI1-.LCFI0
        .byte   0xd
        .uleb128 0x6
        .align 8
```

Anatomy of a Computer

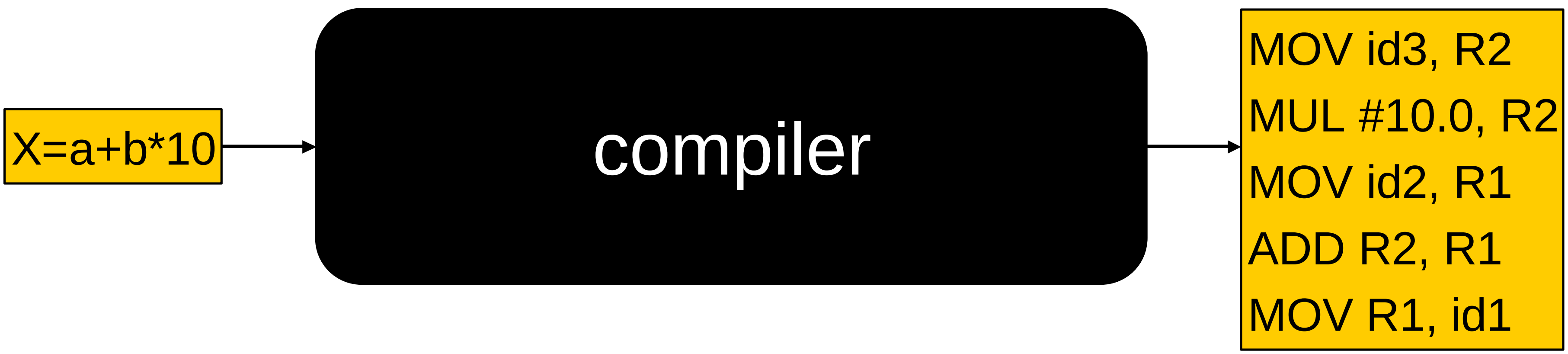


What is a compiler?

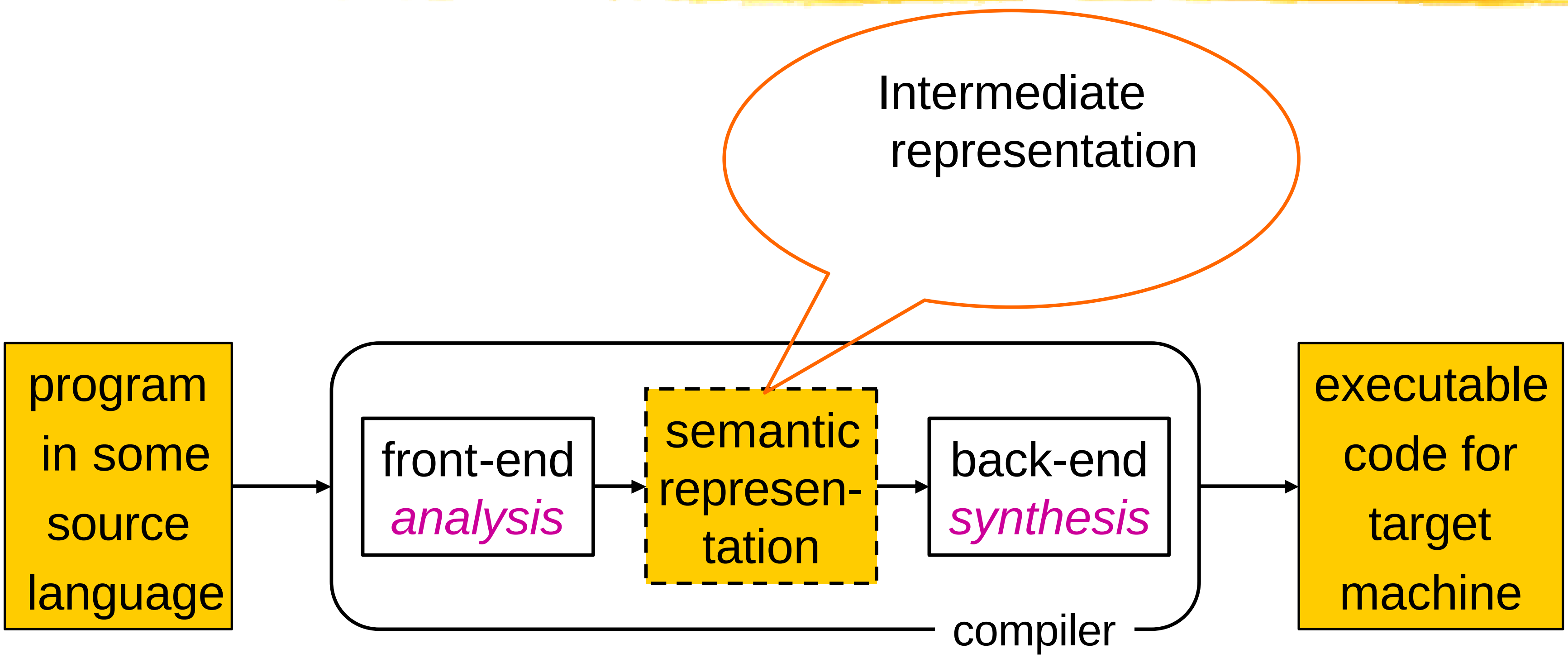
A compiler is a program that reads a program written in one language and translates it into another language.

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

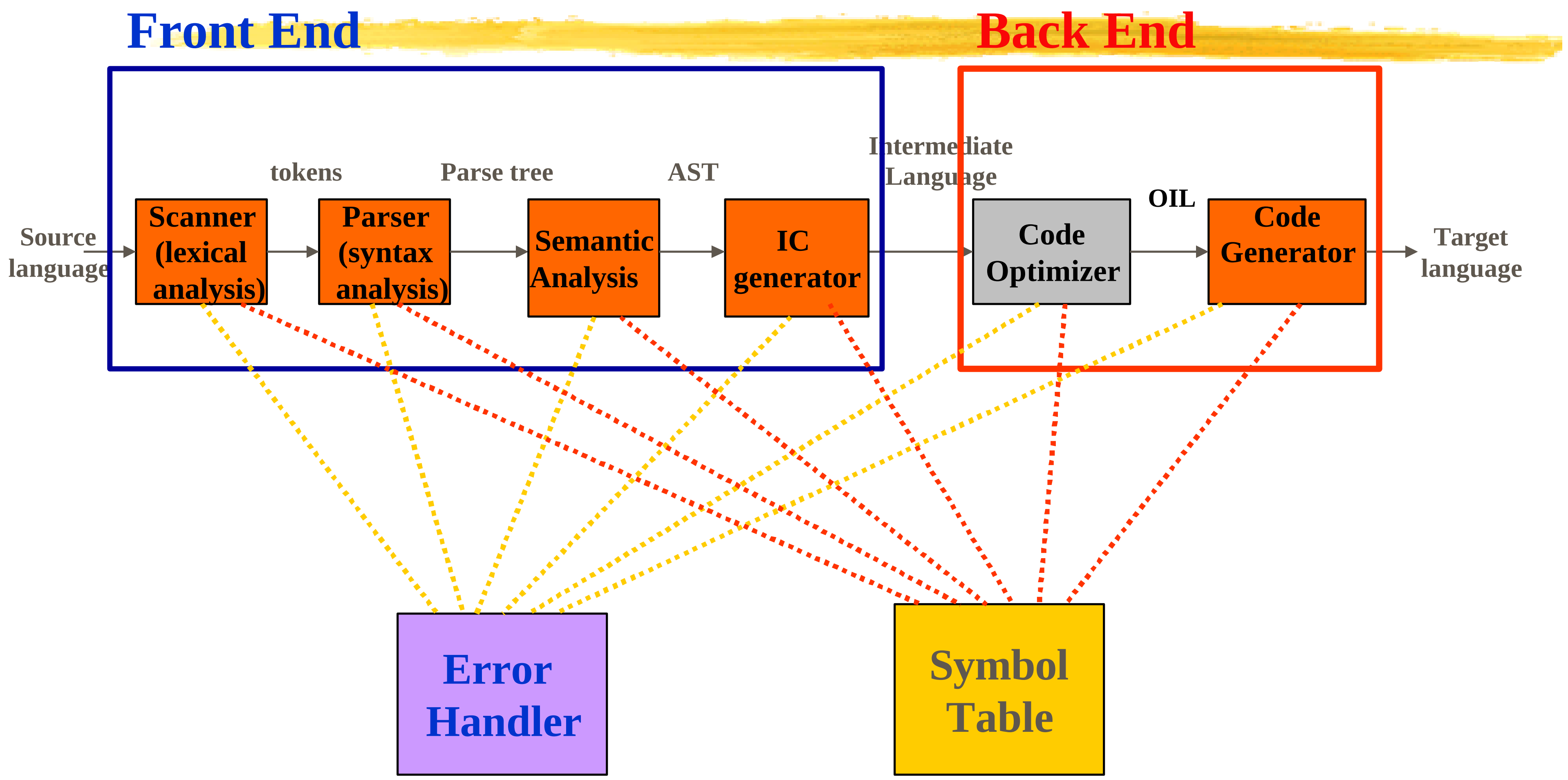
Example


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

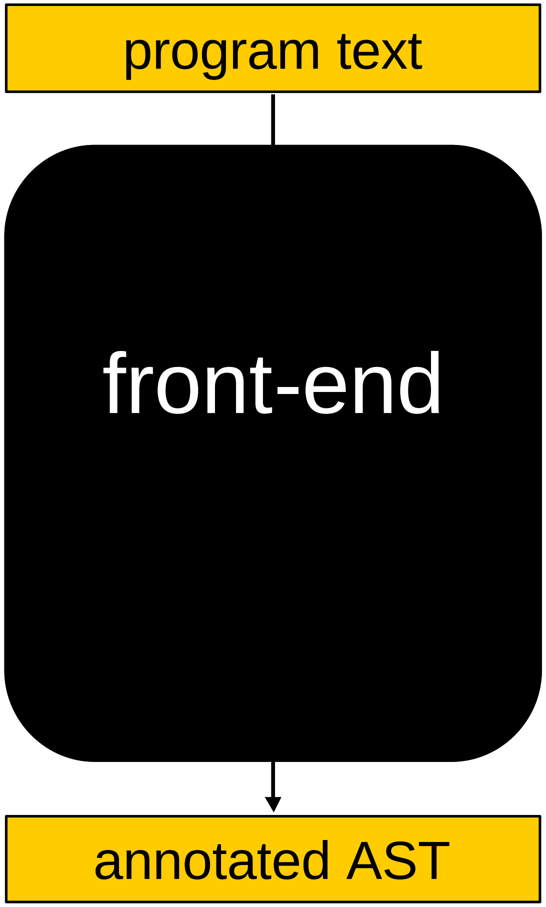
What is a compiler?



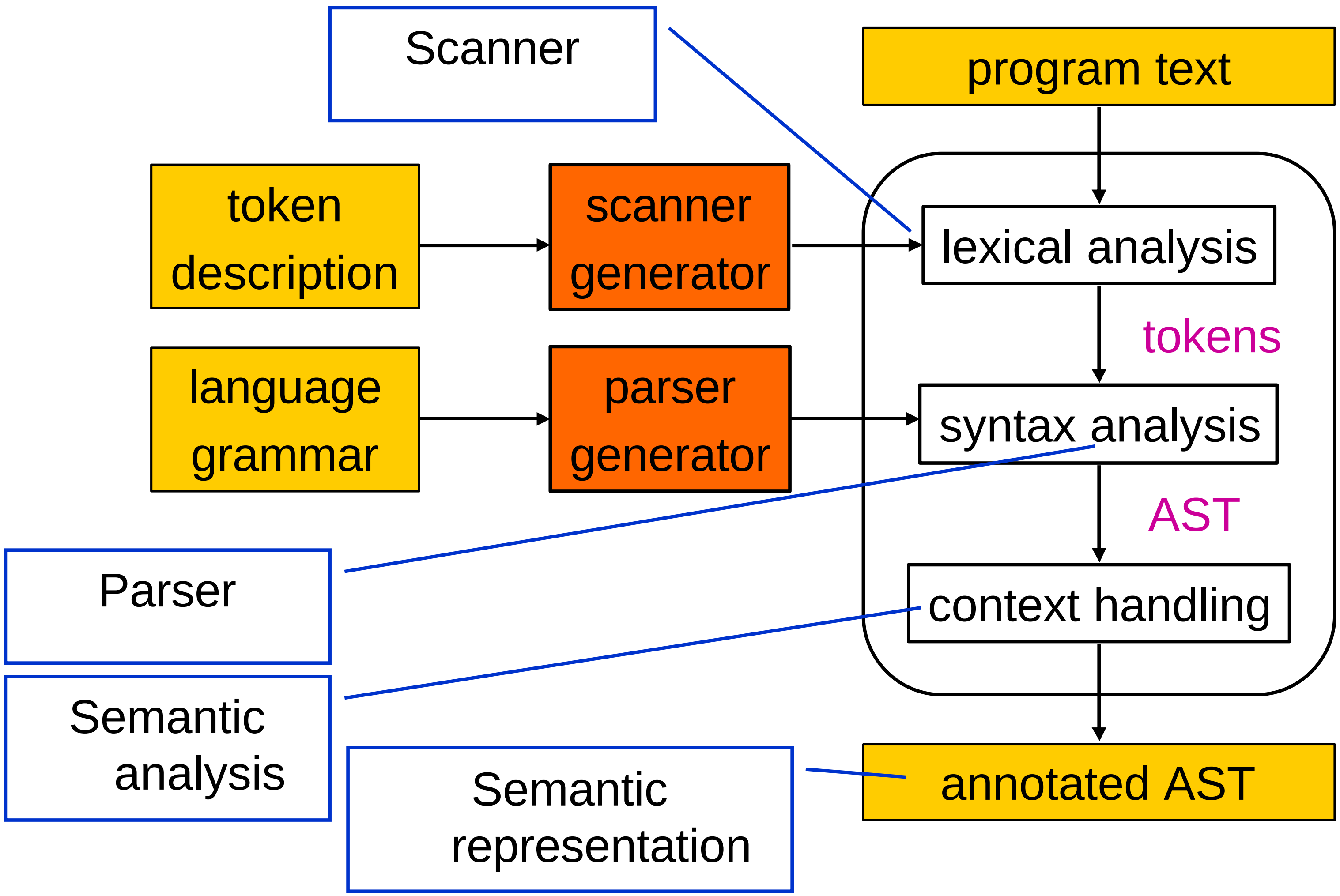
Compiler Architecture



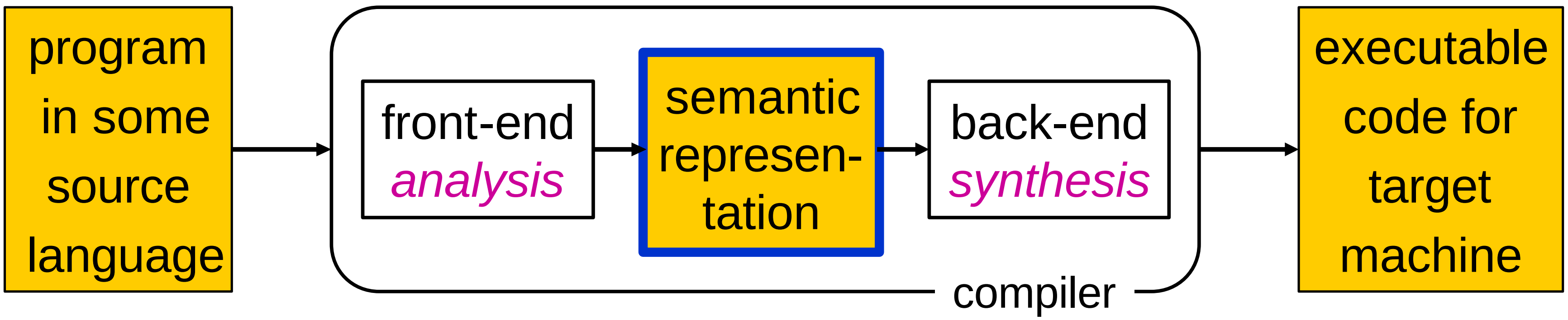
front-end:
from program text to AST



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

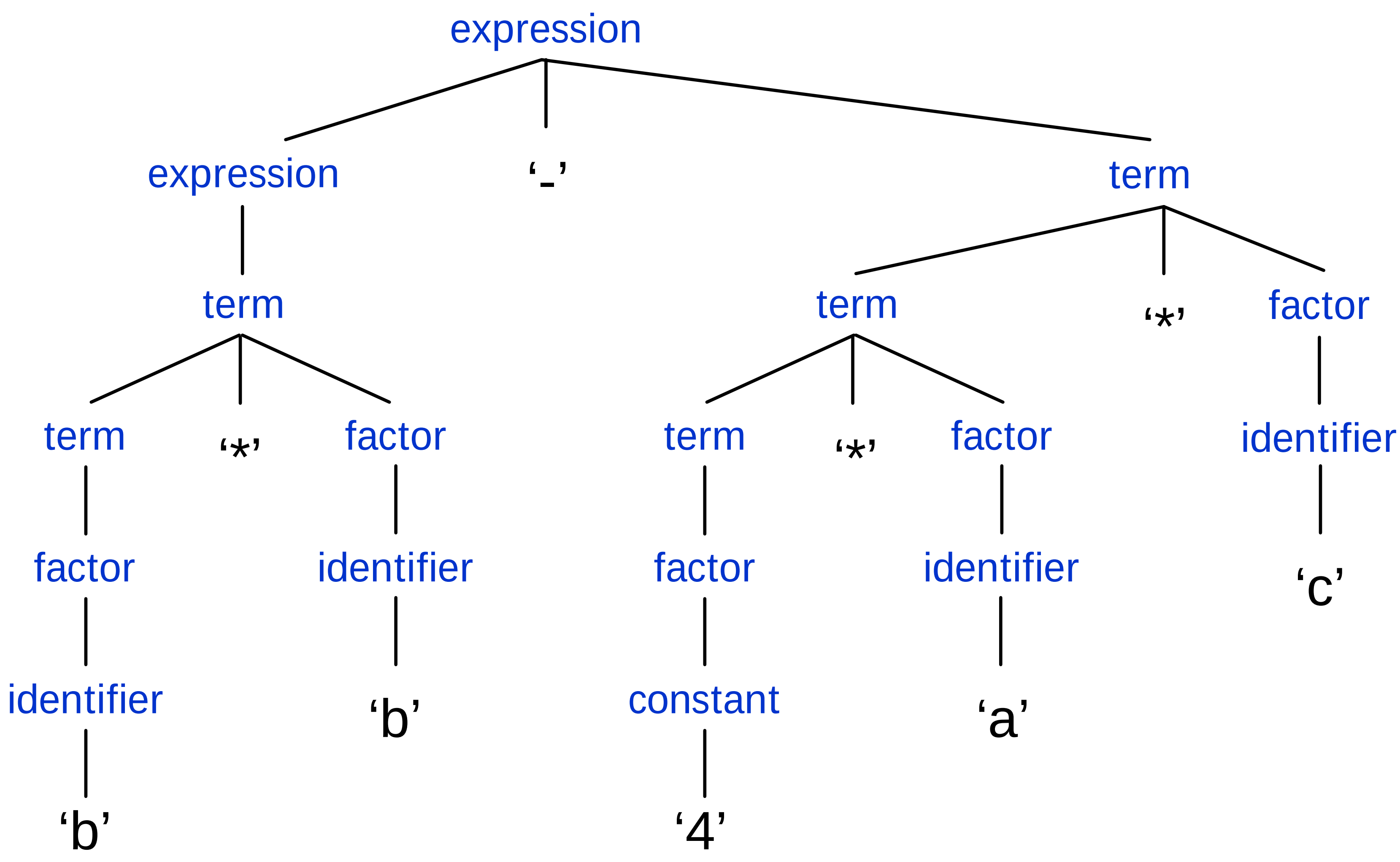
- expression grammar

  expression $\rightarrow$ expression '+' term | expression '-' term | term
  term $\rightarrow$ term '*' factor | term '/' factor | factor
  factor $\rightarrow$ identifier | constant | '(' expression ')'

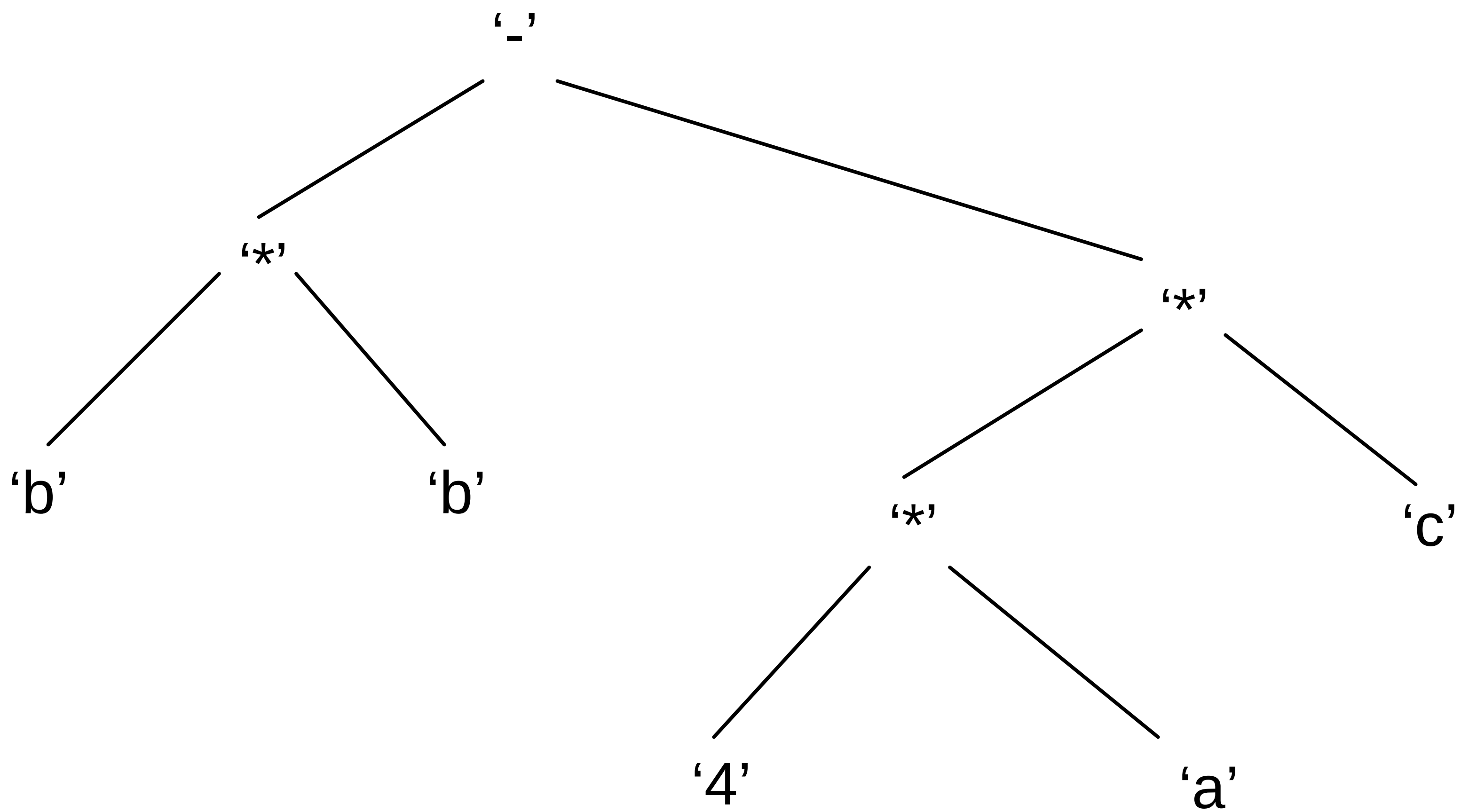
- example expression

  b*b – 4*a*c

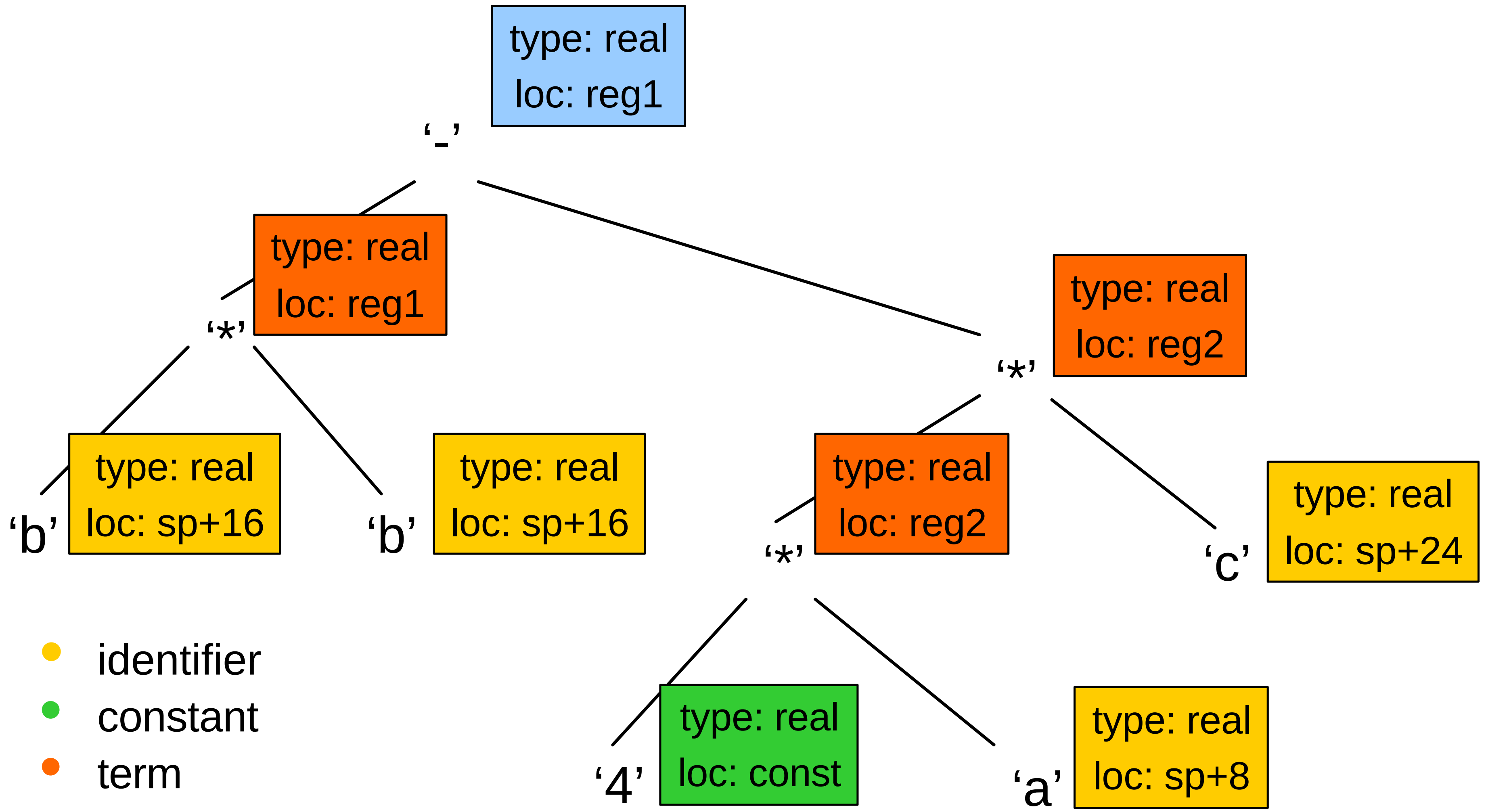
parse tree: b*b – 4*a*c



AST: b*b – 4*a*c

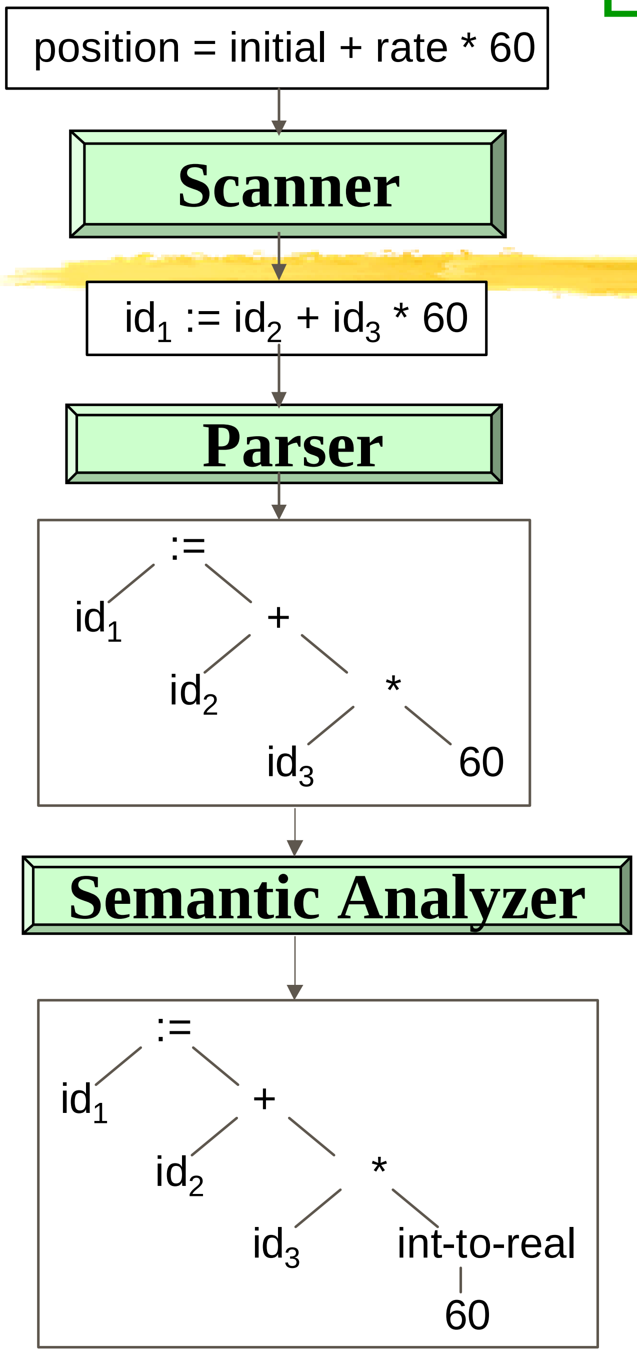


annotated AST: b*b – 4*a*c



- identifier
- constant
- term
- expression

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 (5 min.)

- expression grammar

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

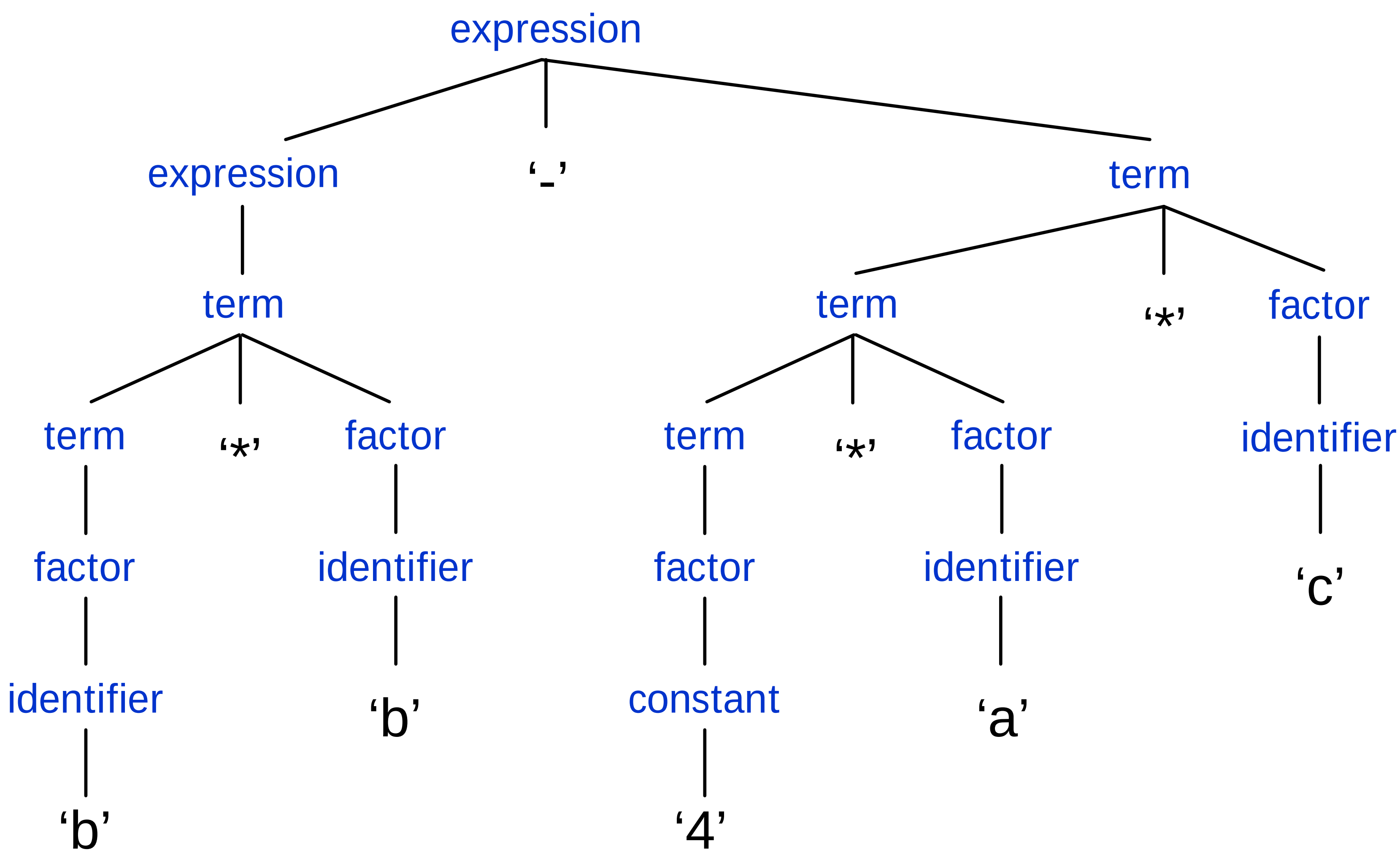
- example expression

  b*b – (4*a*c)

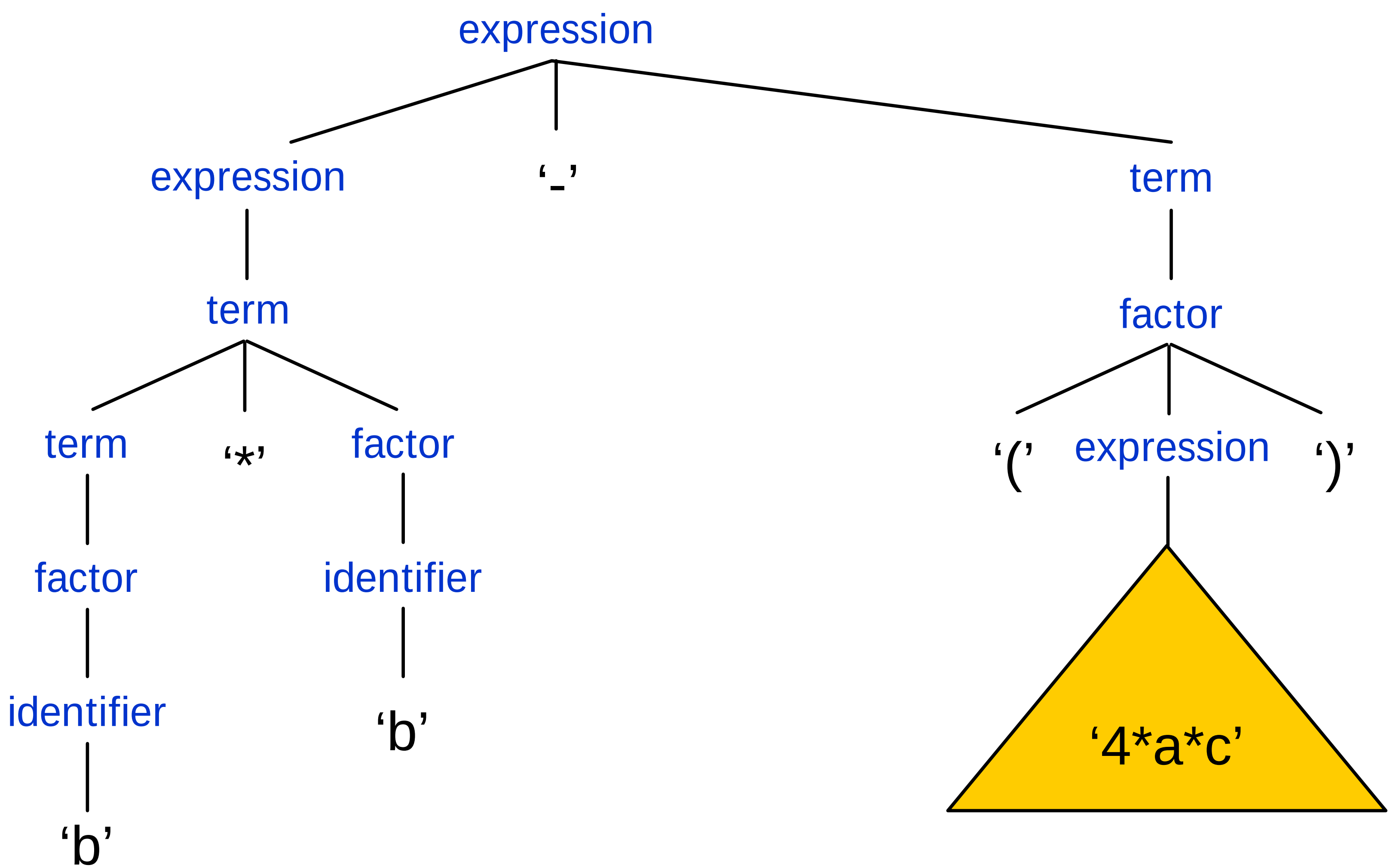
- draw parse tree and AST

Answers

answer
parse tree: b*b – 4*a*c



answer
parse tree: b*b – (4*a*c)



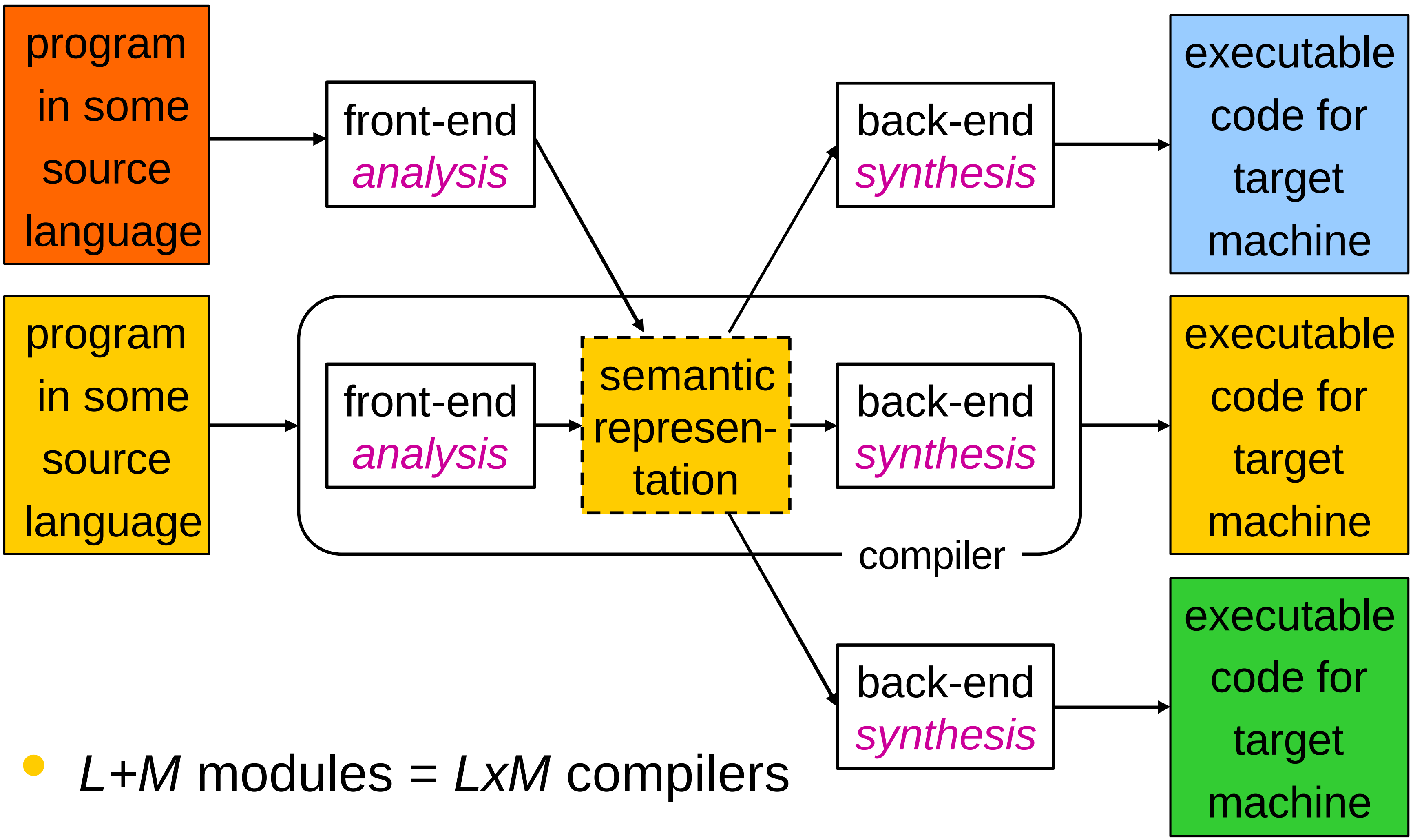
Break

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.

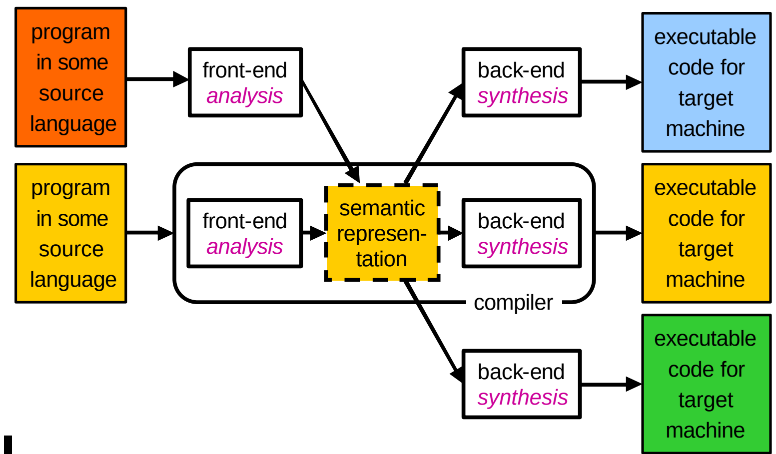
Compiler structure



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

Limitations of modular approach

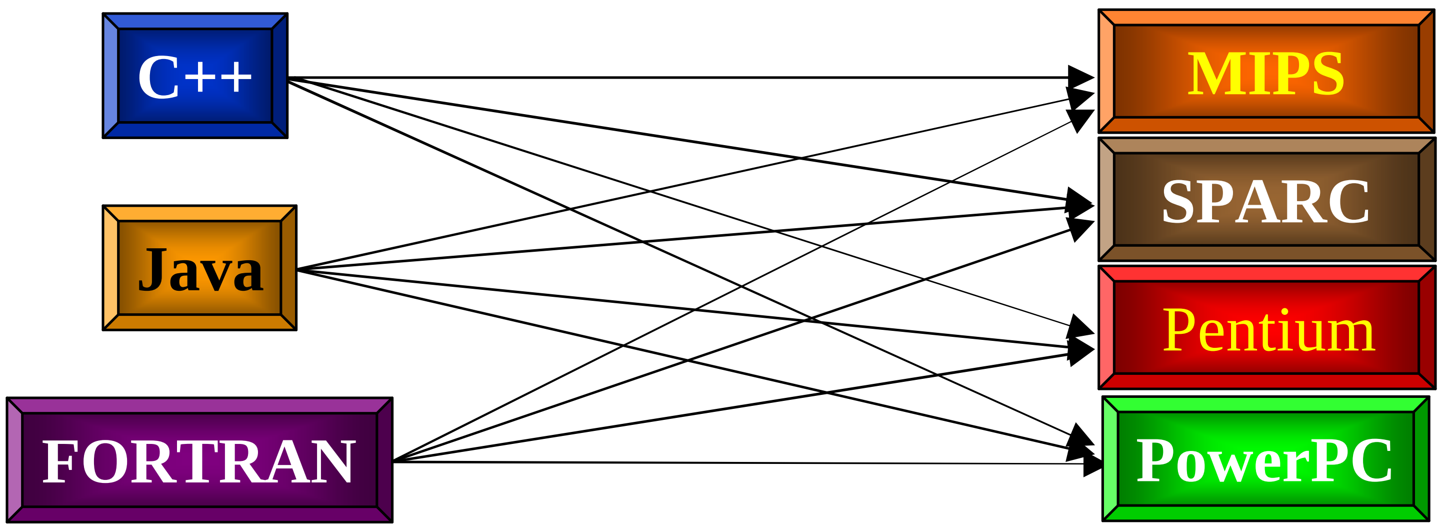
- performance
  - generic vs specific
  - loss of information



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

Front-end and Back-end

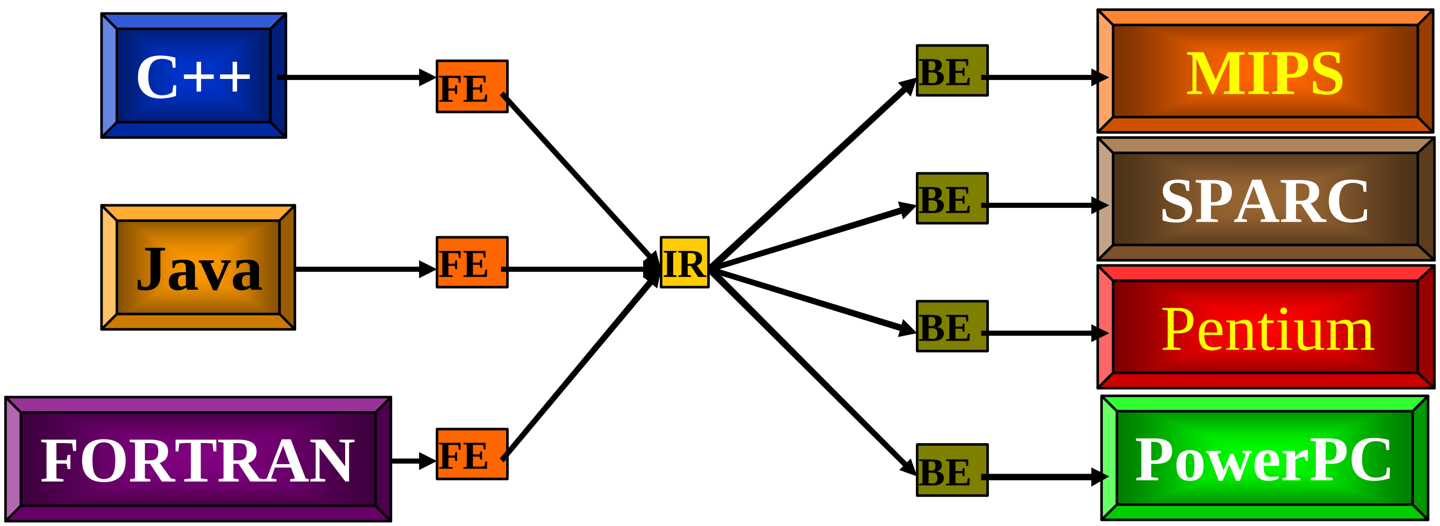
- Suppose you want to write **3** compilers to **4** computer platforms:



We need to write 12 programs

Front-end and Back-end

- But we can do it better

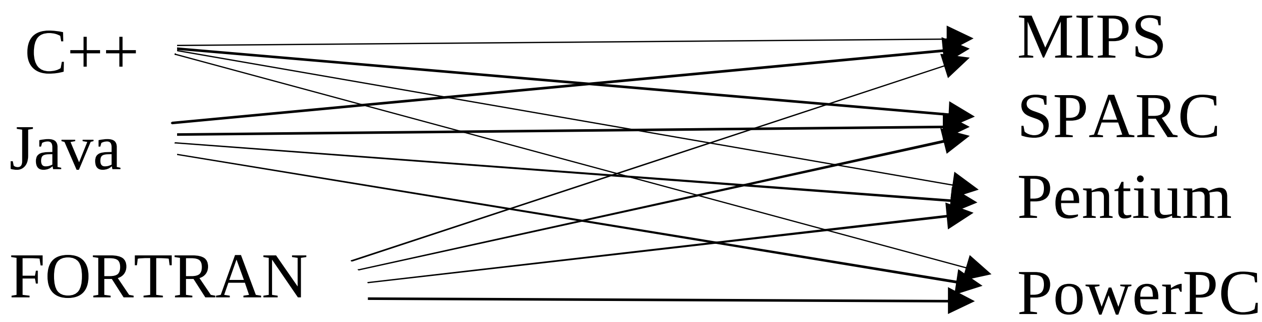


We need to write 7 programs only

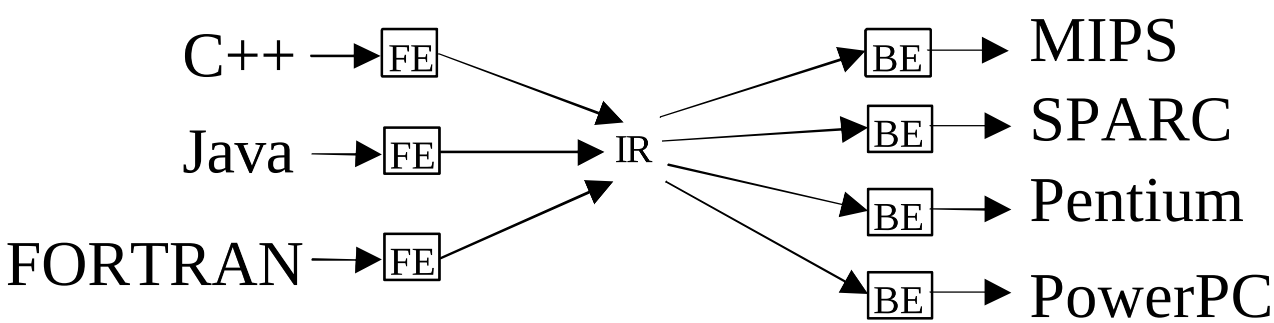
- IR: Intermediate Representation
- FE: Front-End
- BE: Back-End

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:



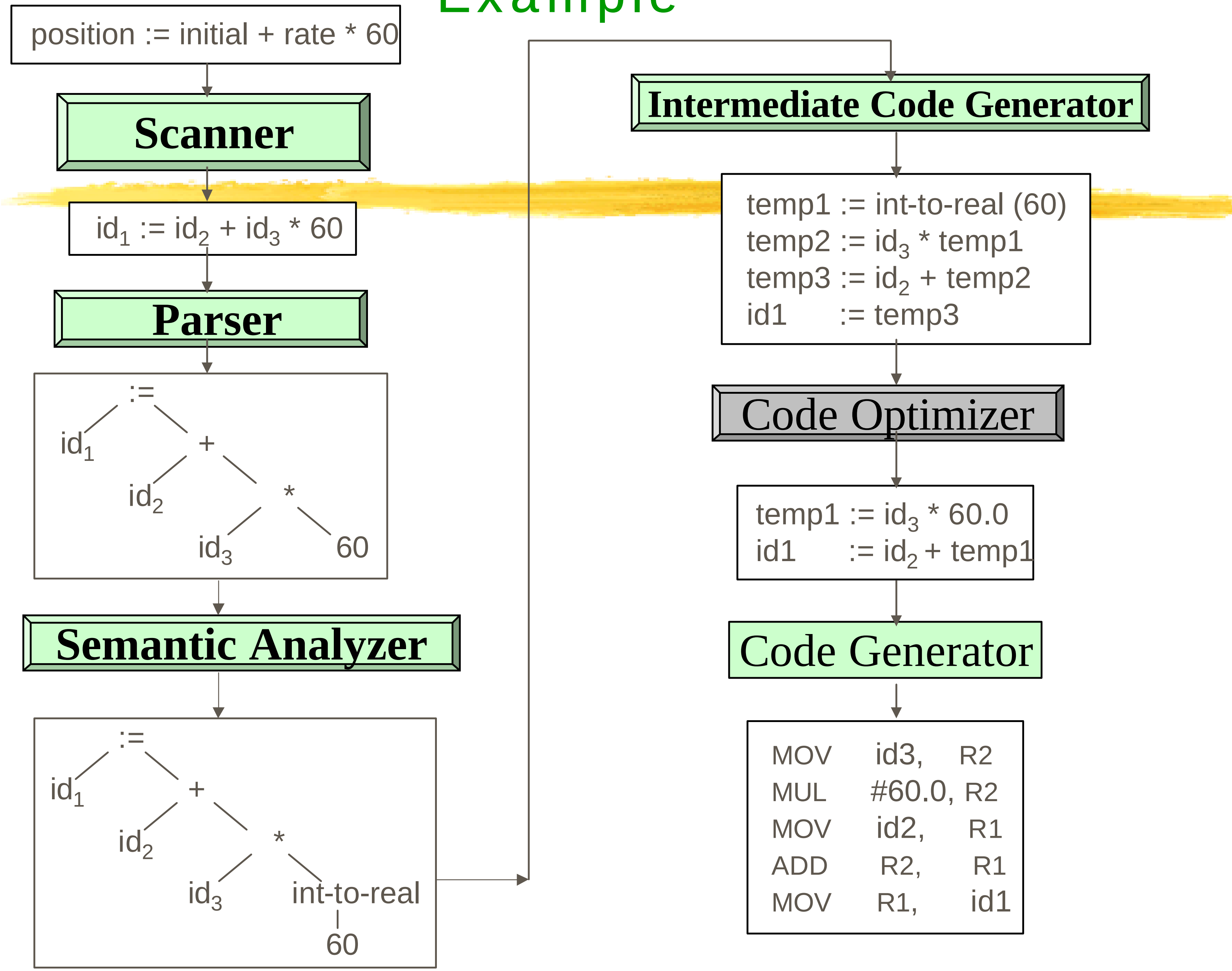
- but we can do it with `n+m` programs:



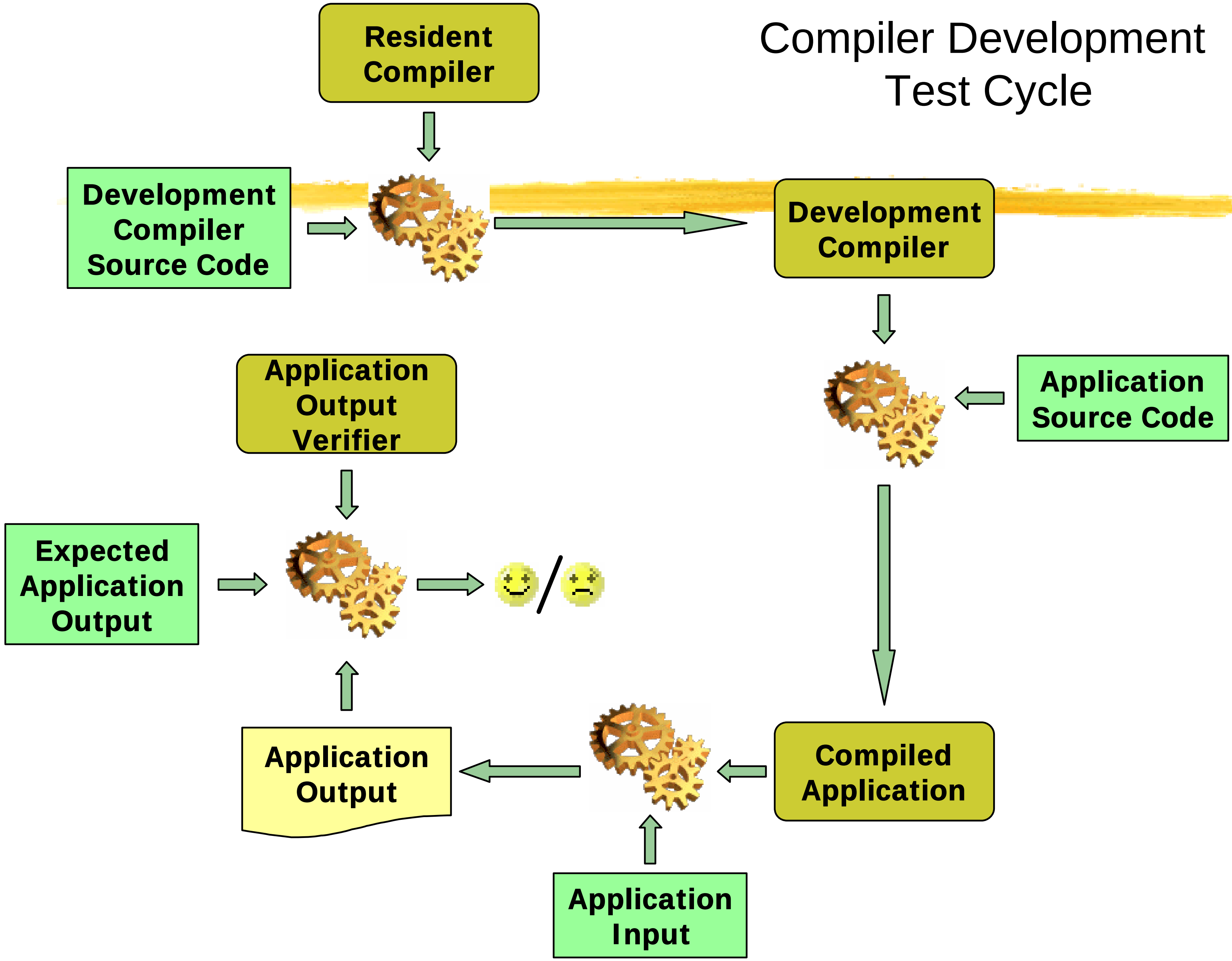
  – IR: Intermediate Representation
  – FE: Front-End
  – BE: Back-End

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

Compiler Development Test Cycle



A Simple Compiler Example

Our goal is to build a very simple compiler its source program are expressions formed from digits separated by plus (+) and minus (-) signs in **infix** form. The target program is the same expression but in a **postfix** form.

Infix expression: Refer to expressions in which the operations are put between its operands.

Example: a+b*10

Postfix expression: Refer to expressions in which the operations come after its operands.

Example: ab10*+

Infix to Postfix translation

1. If E is a digit then its postfix form is E

2. If $E=E_1+E_2$ then its postfix form is $E_1`E_2`+$

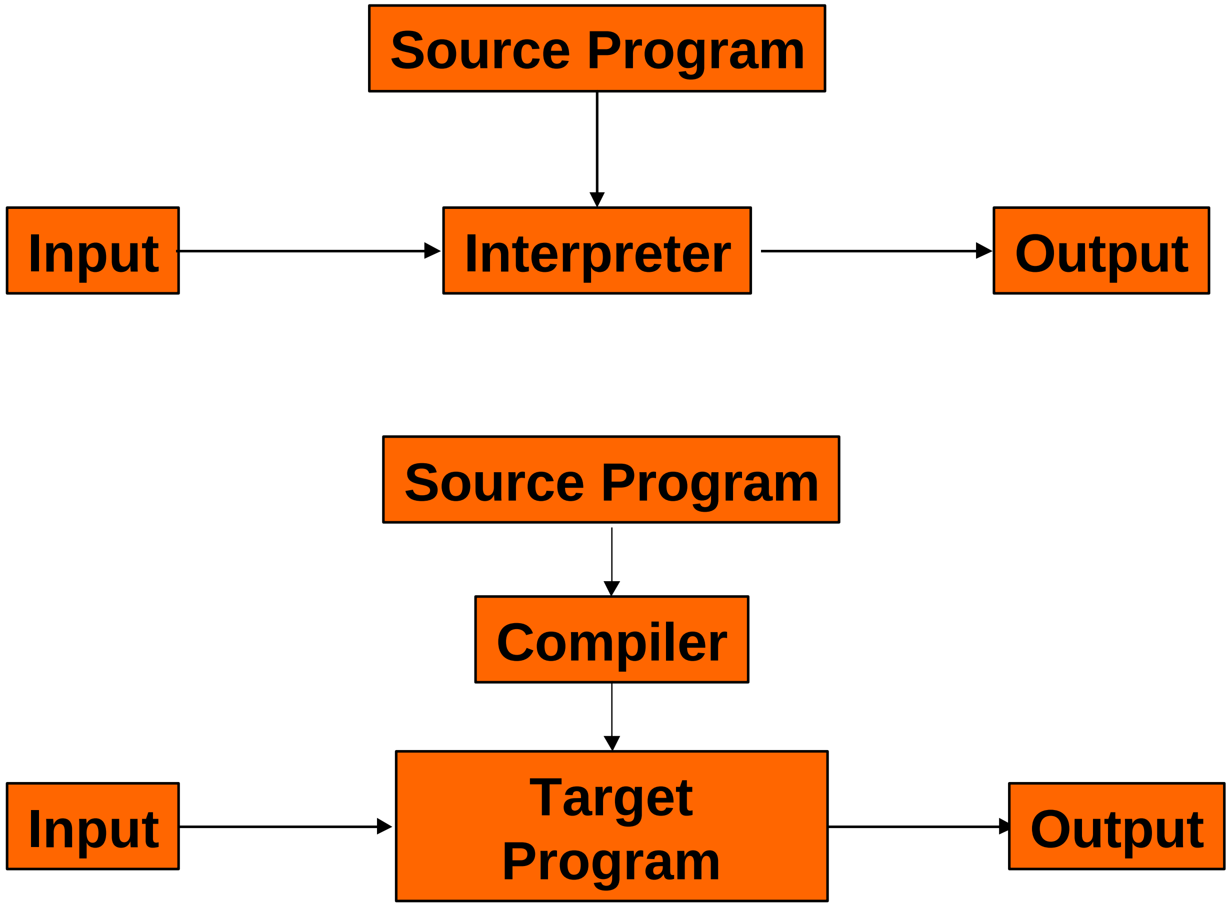
3. If $E=E_1-E_2$ then its postfix form is $E_1`E_2`-$

4. If $E=(E_1)$ then E and E_1 have the same postfix form

Where in 2 and 3 $E_1`$ and $E_2`$ represent the postfix forms of E_1 and E_2 respectively.

END

Interpreter vs Compiler



Typical Compiler