Introduction to Compilers

Greta Yorsh

Today

- What is this module about?
- What is a compiler?
- How does it work?
- Why should you care?

What is a compiler?

"A compiler is a **computer program** that **transforms** code written in one programming language (**source language**) into another language (**target language**).

The most common reason for wanting to transform code is to create an **executable program**."

--*Wikipedia, 2015*

What is a compiler?

C
C++
Pascal
Java
Postscript
TeX
Perl
JavaScript
Python
Ruby
Prolog
Lisp
Scheme
Scala
Haskel
OCaml ...

gcc
llvm
icc
javac
jikes
Visual C
CompCert
ghc
pypy
ocamlc
ocamlopt
scalac …

IA32
IA64
ARM
SPARC
MIPS

C
C++
Pascal
Java

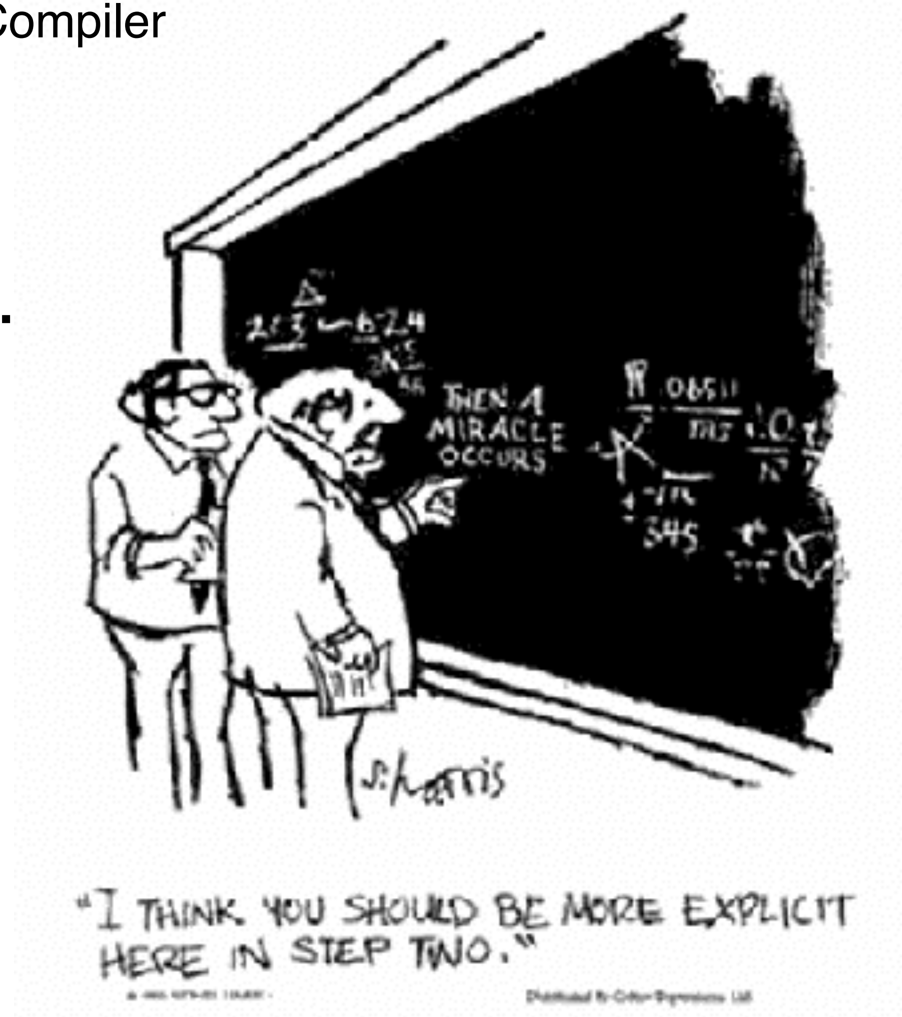
Java Bytecode
…

What is a compiler?

source code → Compiler → target code

source language
C

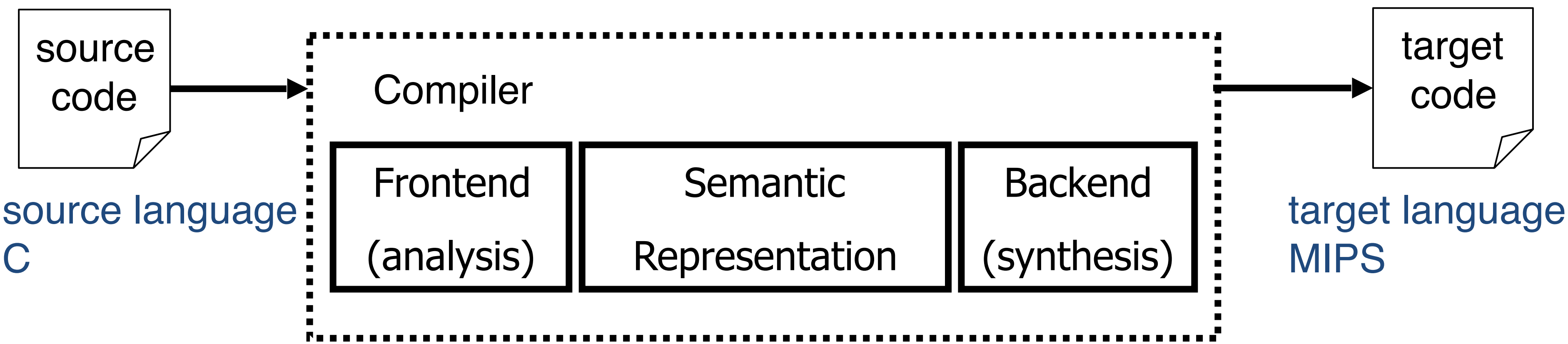
target language
MIPS




```
int a, b;
a = 2;
b = a*2 + 1;
```

```
li      $t0, 2
sll     $t1, $t0, 1
addiu $t1, $t1, 1
```

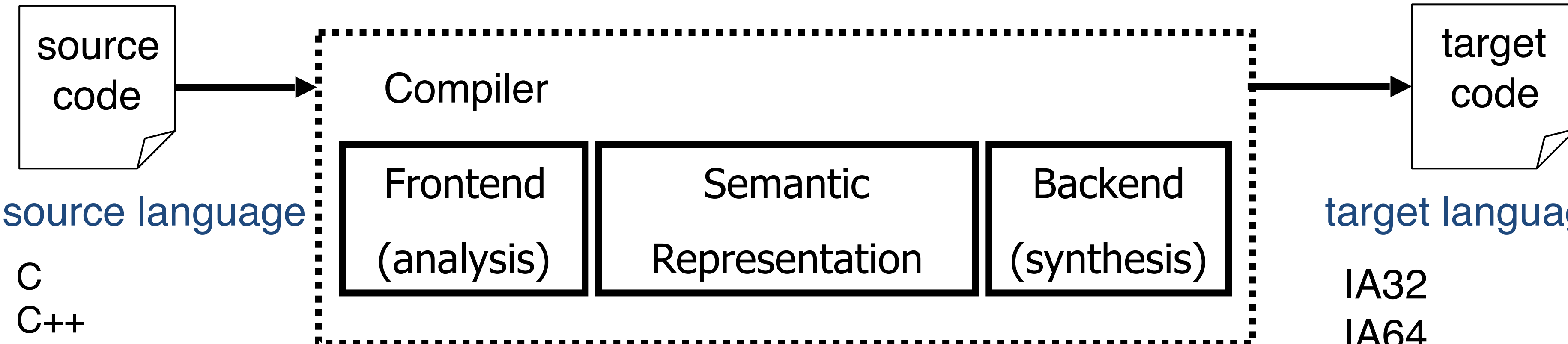
Anatomy of a compiler




```
int a, b;
a = 2;
b = a*2 + 1;
```

```
li      $t0, 2
sll     $t1, $t0, 1
addiu $t1, $t1, 1
```

Anatomy of a compiler



source language

C
C++
Pascal
Java
Postscript
TeX
Perl
JavaScript
Python
Ruby
Prolog
Lisp
Scheme
Scala
Haskel
OCaml ...

Why?

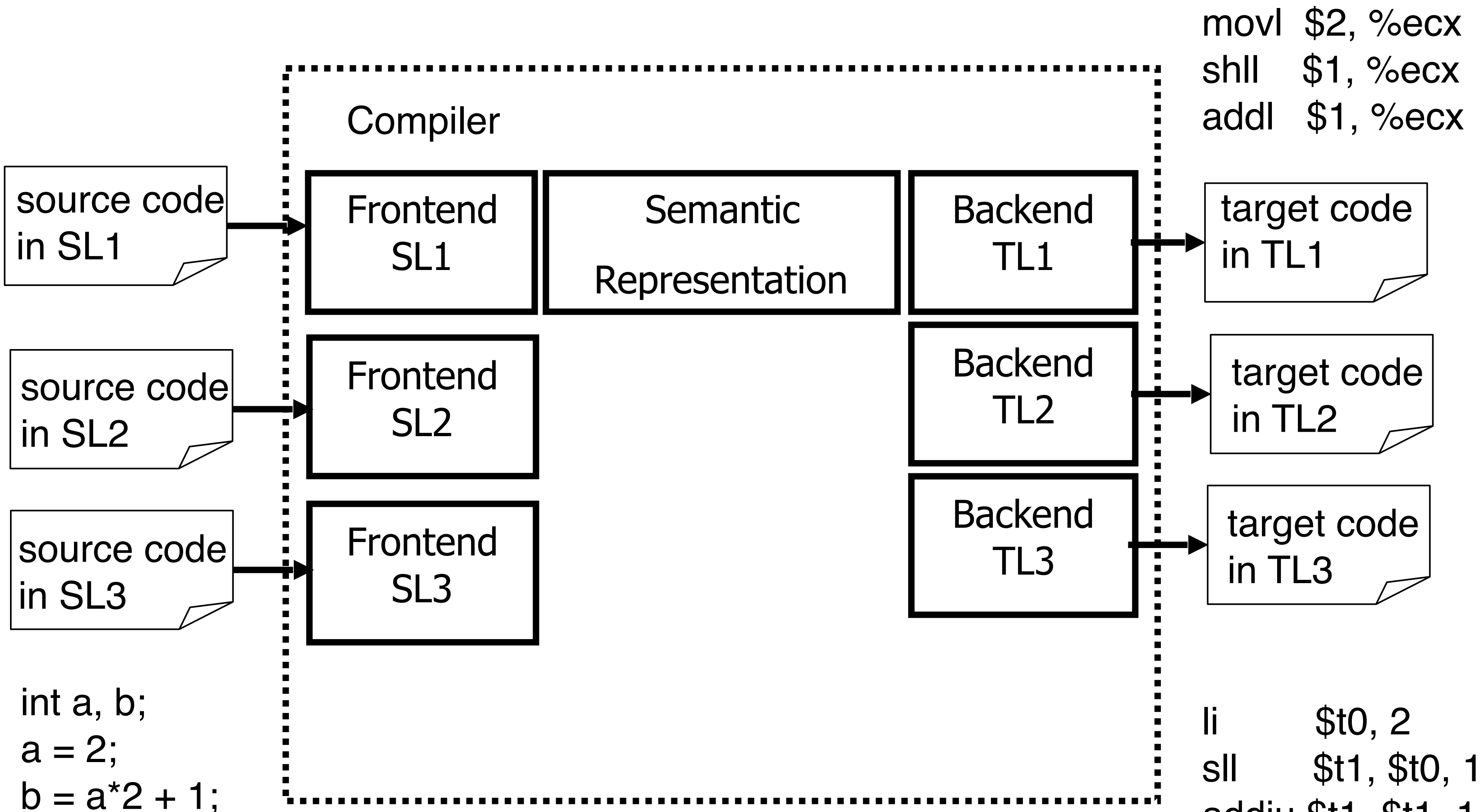
target language

IA32
IA64
ARM
SPARC
MIPS

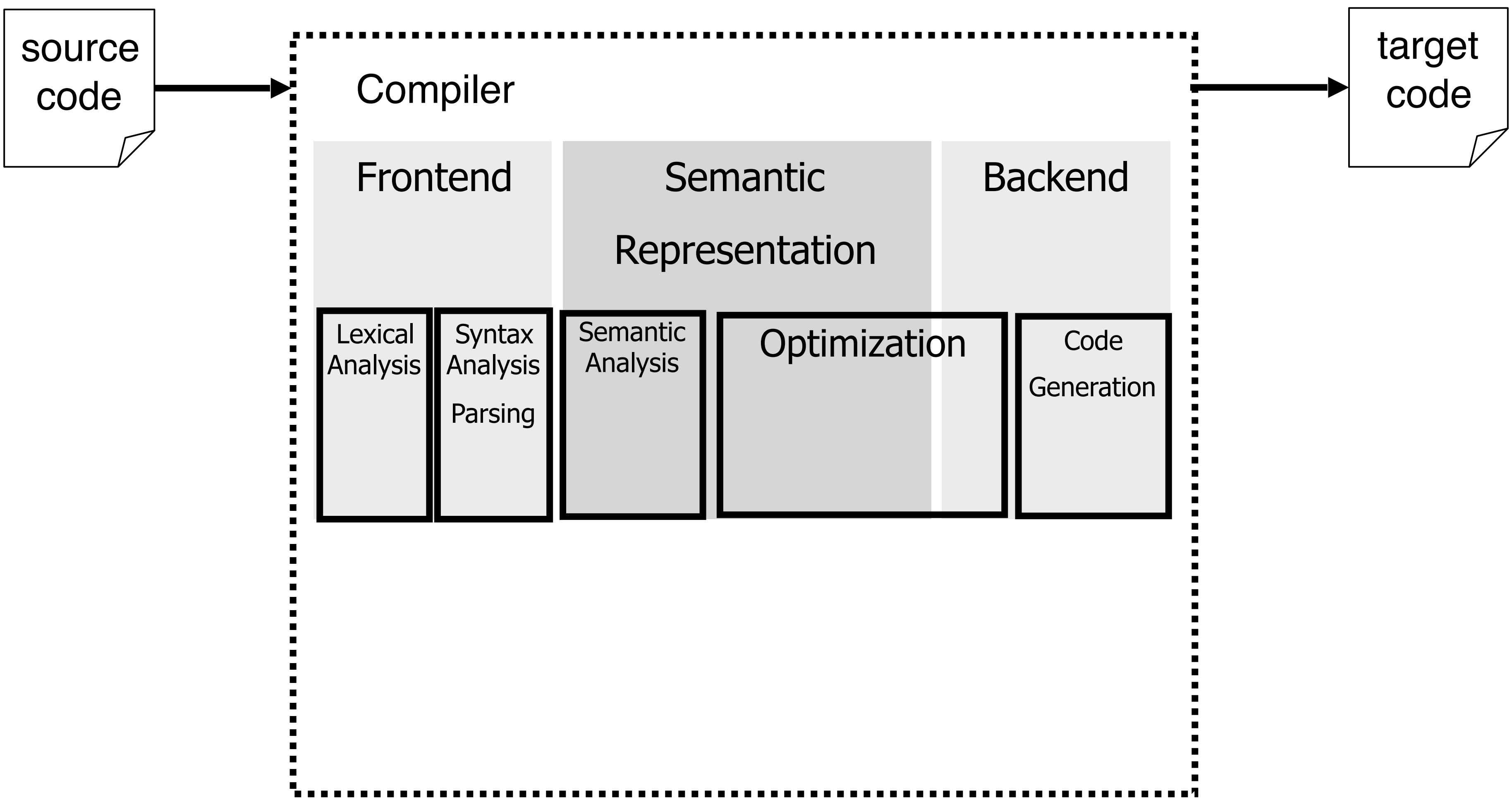
C
C++
Pascal
Java

Java Bytecode
…

Modularity


movl $2, %ecx
shll $1, %ecx
addl $1, %ecx
Compiler
source code in SL1
Frontend SL1
Semantic Representation
Backend TL1
target code in TL1
source code in SL2
Frontend SL2
Backend TL2
target code in TL2
source code in SL3
Frontend SL3
Backend TL3
target code in TL3
int a, b;
a = 2;
b = a*2 + 1;
li $t0, 2
sll $t1, $t0, 1
addiu $t1, $t1, 1

Anatomy of a modern compiler



JOURNEY INSIDE A COMPILER

Example

source
code

$$x = b*b - 4*a1*c2$$

Token Stream

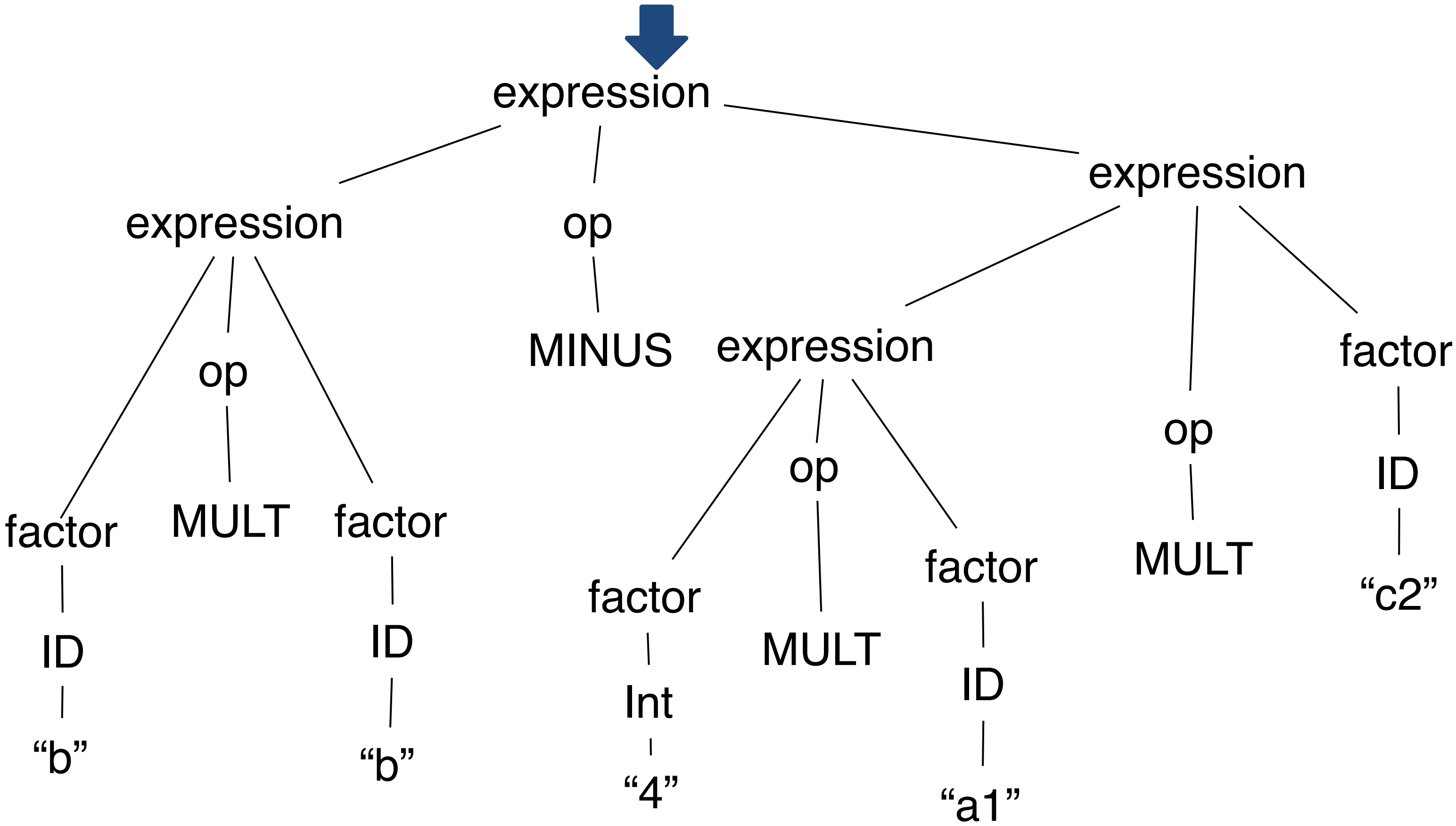
x = b*b – 4*a1*c2

ID,"x" EQ **ID,"b"** MULT **ID,"b"** MINUS INT,4 MULT **ID,"a1"** MULT **ID,"c2"**

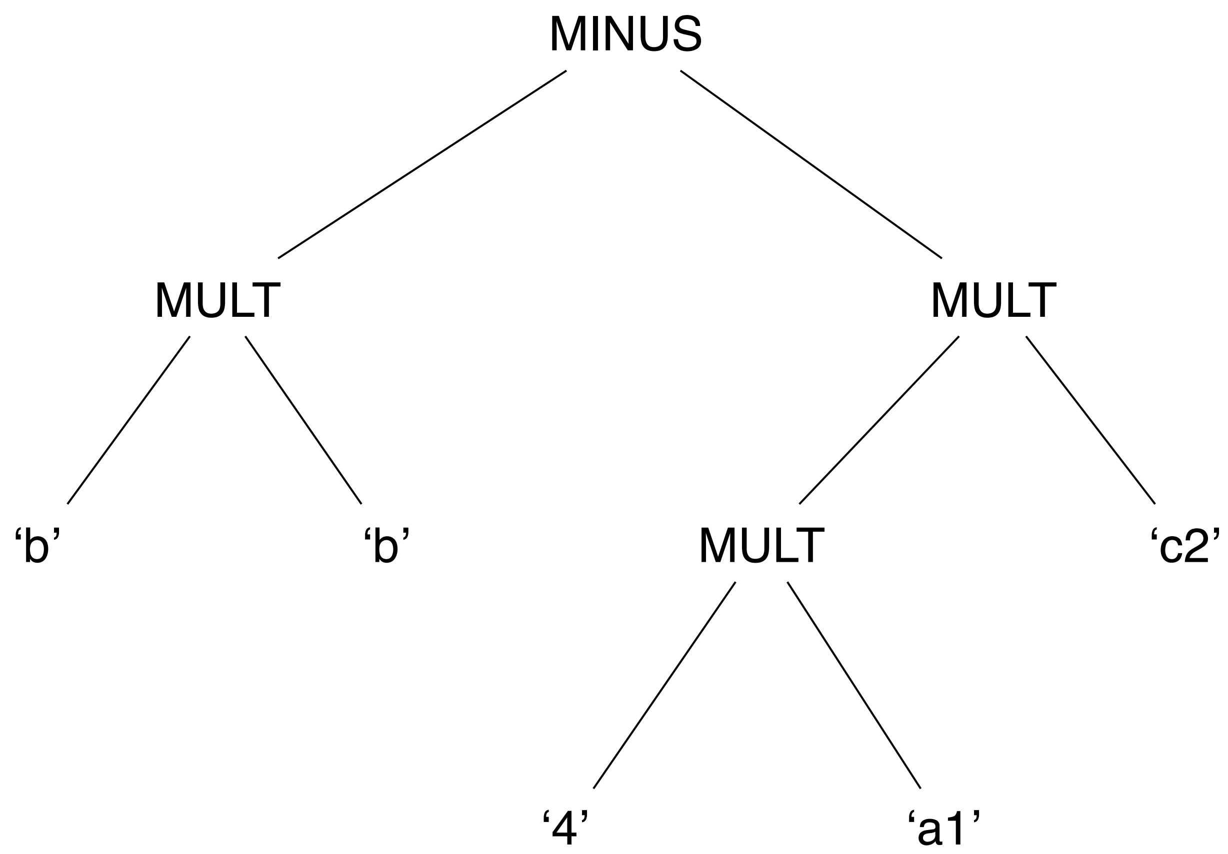
x = b*b – 4*a1*c2 → Lexical Analysis | Syntax Analysis | Semantic Analysis | Optimization | Code Generation

Syntax Tree (Parse Tree)

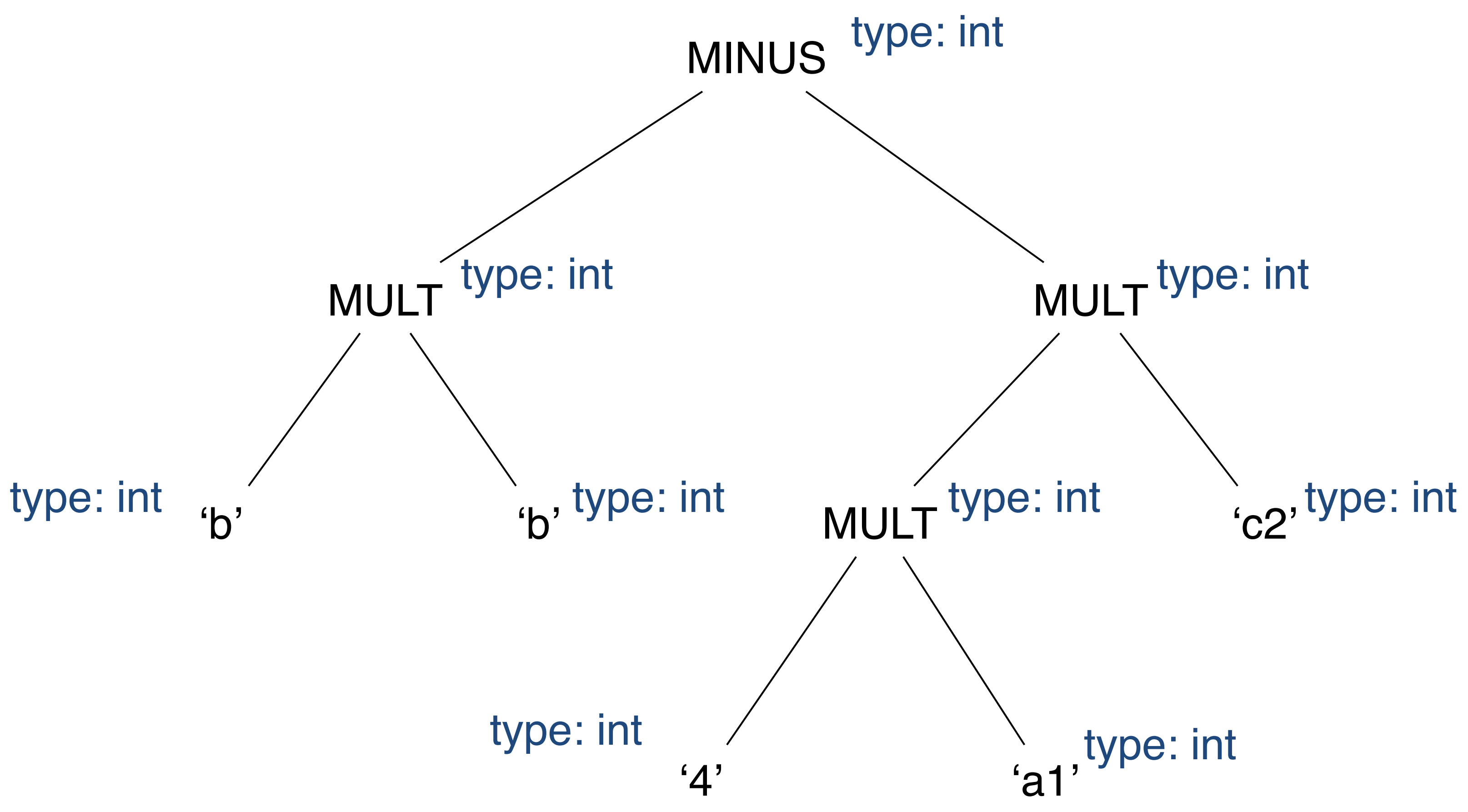
ID,”x” EQ **ID,”b”** MULT **ID,”b”** MINUS INT,4 MULT **ID,”a1”** MULT **ID,”c2”**

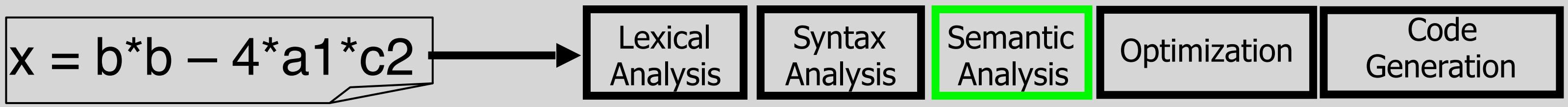


Abstract Syntax Tree (AST)

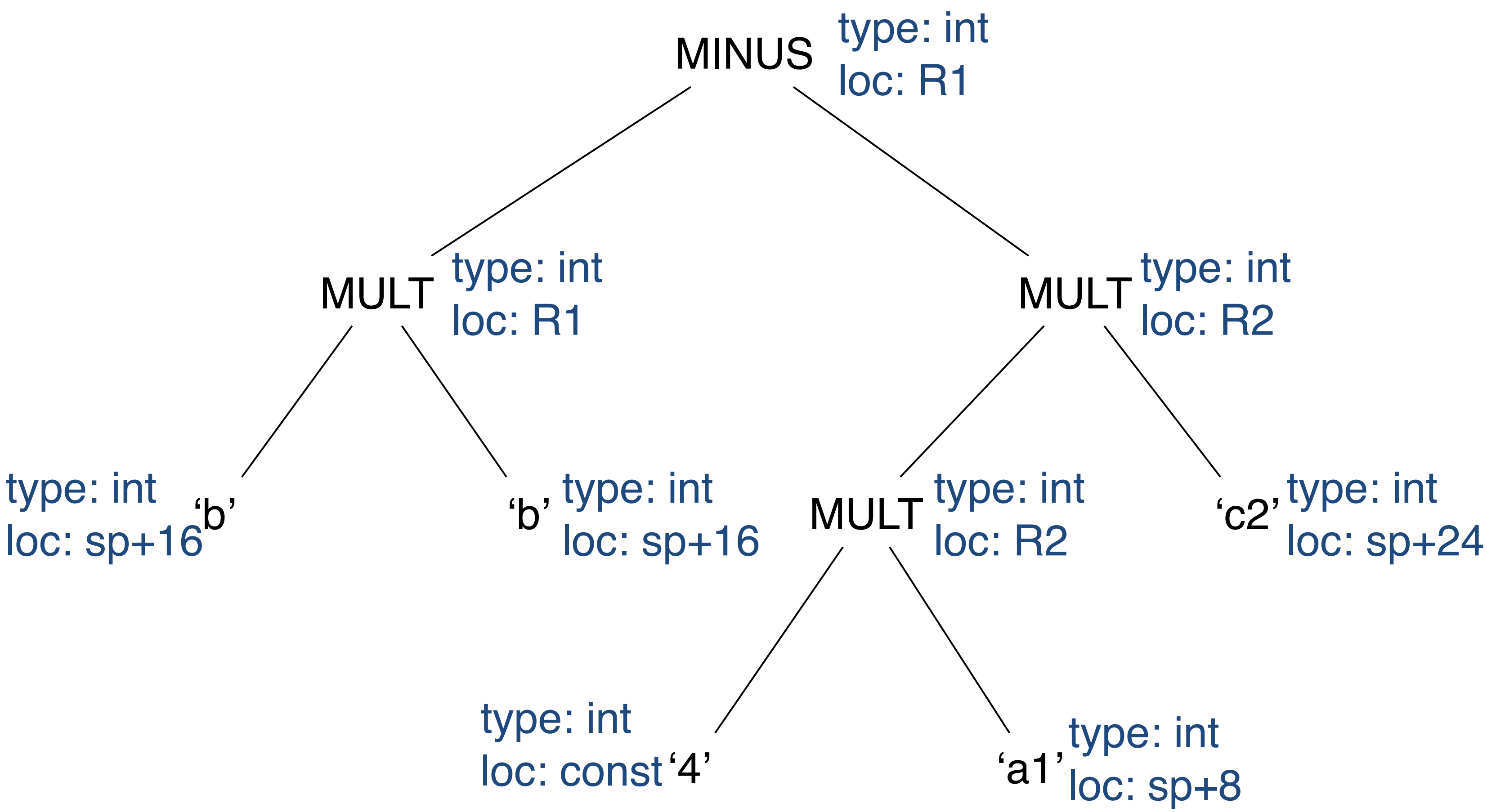


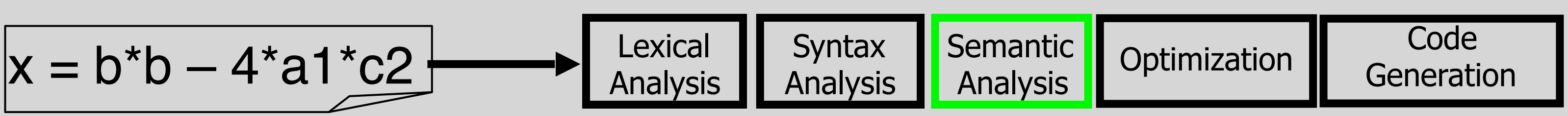
Annotated Abstract Syntax Tree



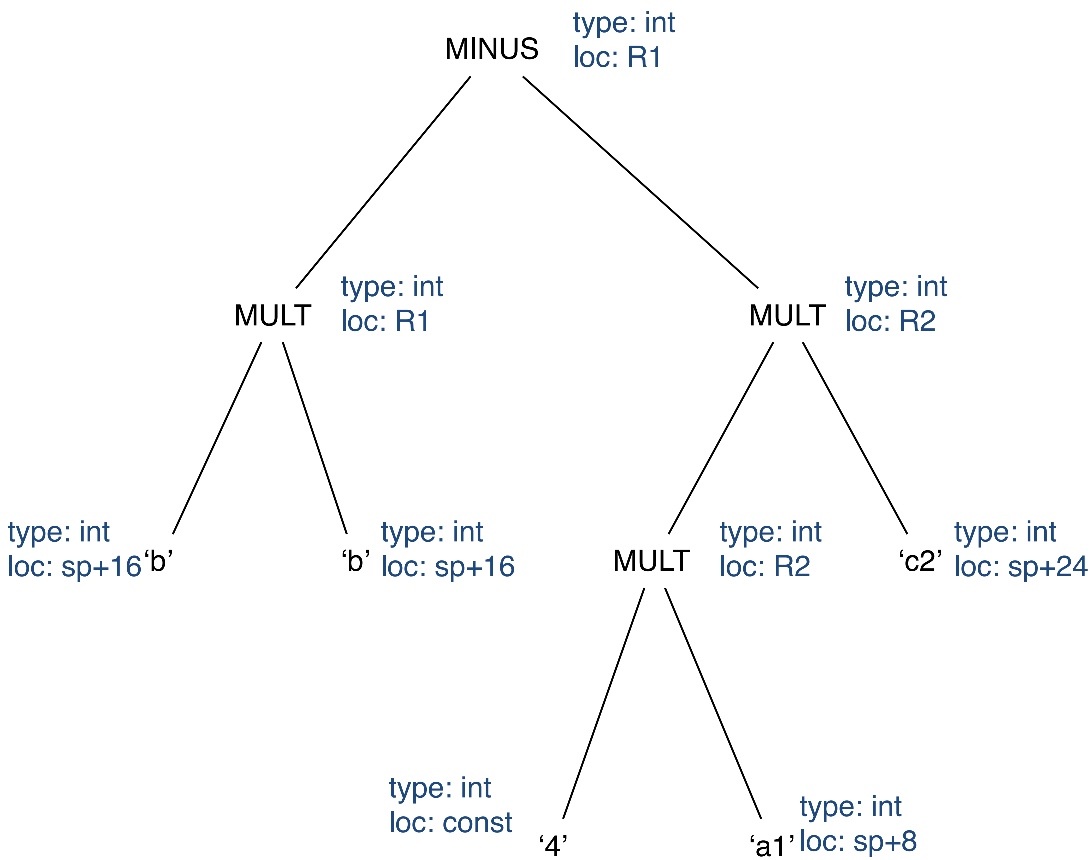


Annotated Abstract Syntax Tree



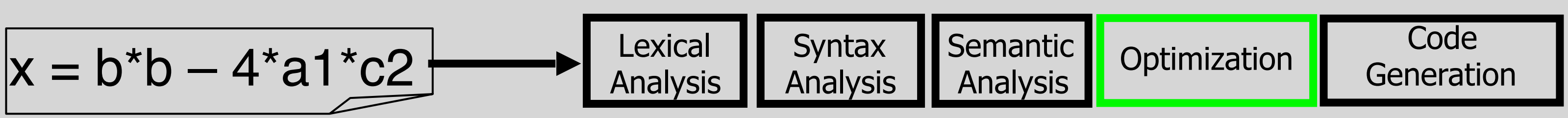


Intermediate Representation (IR)

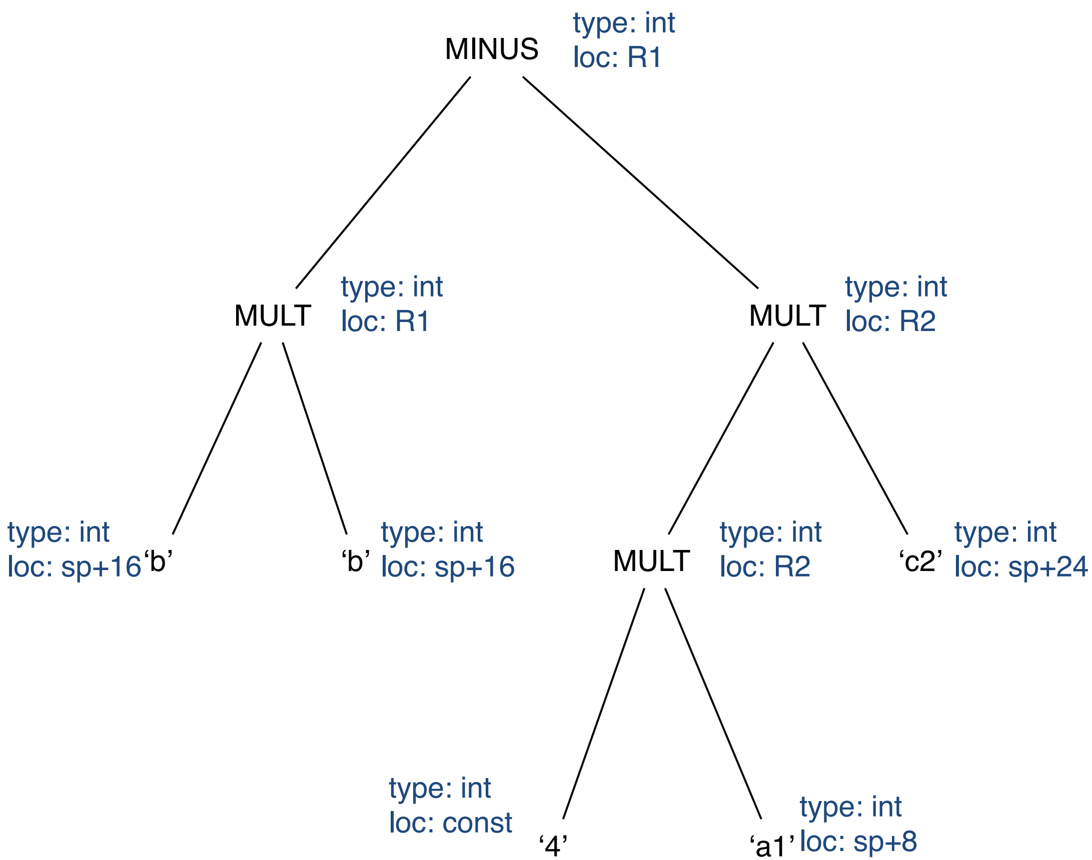


Intermediate Representation

R2 = 4*a1
R1=b*b
R2= R2*c2
R1=R1-R2



Target Code

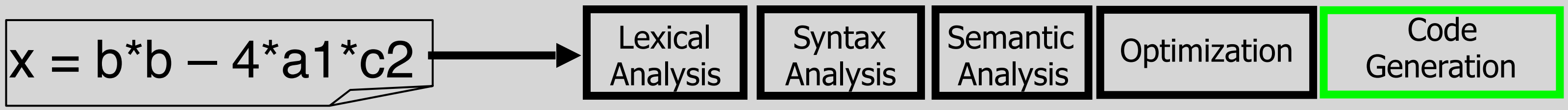


Intermediate Representation

R2 = 4*a1
R1=b*b
R2= R2*c2
R1=R1-R2

Assembly Code

```
lw    $t0,  8($sp)
sll   $t0, $t0, 2
lw    $t1, 16($sp)
mul  $t1, $t1, $t1
lw    $t2, 24($sp)
mul  $t1, $t1, $t2
subu $t0,$t1
```



Error Checking in Every Stage

- Lexical analysis: illegal tokens
- Syntax analysis: illegal syntax
- Semantic analysis: incompatible types, undefined variables, …

- Every phase tries to recover and proceed with compilation (why?)
- Divergence is a challenge

Errors in lexical analysis

pi = 3.141.562

Illegal token

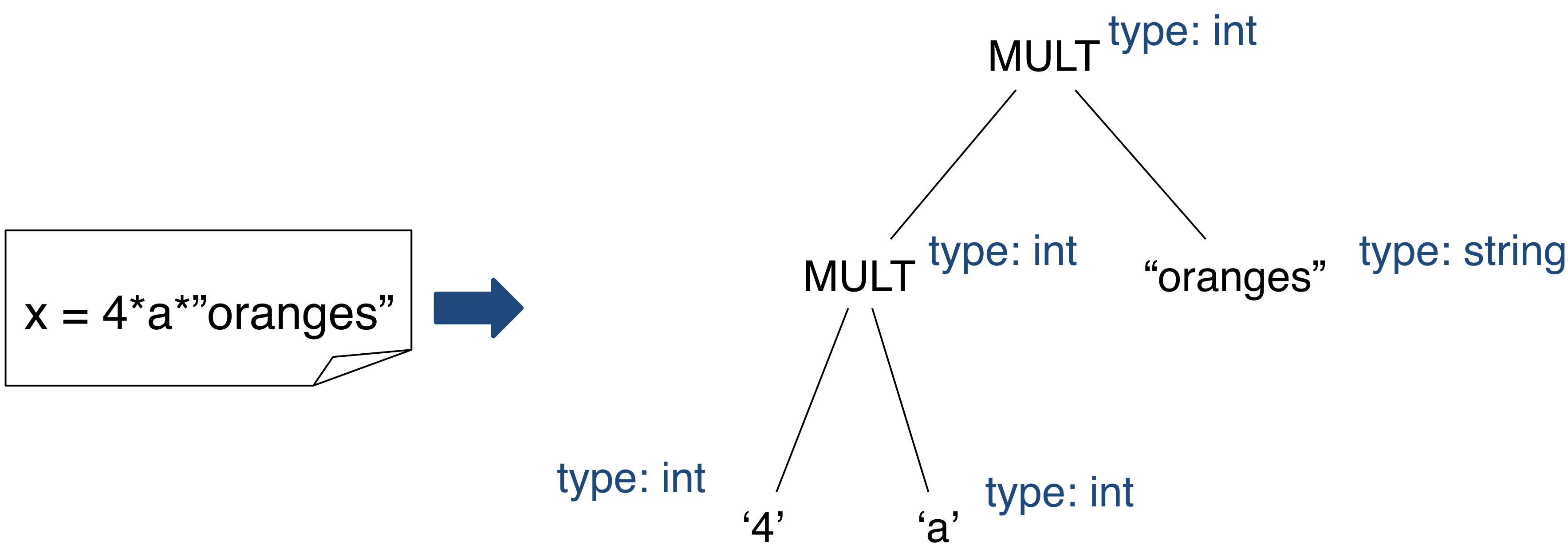
pi = 3oranges

Illegal token

pi = oranges3

<ID,”pi”>, <EQ>, <ID,”oranges3”>

Error detection: type checking

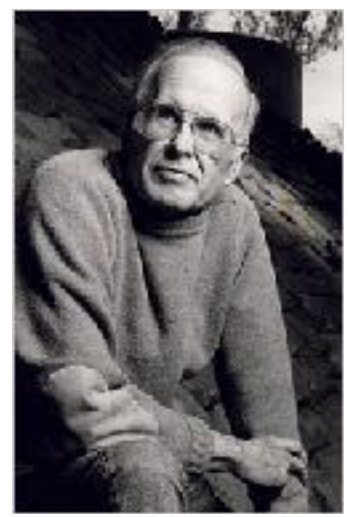


WHY SHOULD YOU CARE?

A very brief history of compilers

- First, there was nothing
- Then, there was machine code
- Then, there were assembly languages
- **Higher-level languages**

Grace Hopper, US Navy
inventor of COBOL
coined the term “compiler”



John Backus, IBM
team lead on FORTRAN

Central role of compilers in CS

source code

- Programming Languages
- Software Engineering
- Security

Compiler

- Useful formalisms
  - Regular expressions
  - Context-free grammars
- Data structures
- Algorithms

target code

- Runtime environment
- Virtual machines
- Garbage collection
- Operating system
- Computer Architecture

Turing Awards

1966	Alan Perlis	Compiler construction	1987	John Cocke	Compilers & RISC Architecture
1967	Maurice Wilkes	EDSAC Computer	1988	Ivan Sutherland	Computer Graphics
1968	Richard Hamming	Information Theory	1989	Velvel Kahan	Numerical Analysis, IEEE FP
1969	Marvin Minsky	Artificial Intelligence	1990	Fernando Corbato	Operating Systems, Timesharing (CTSS & Multics)
1970	James Wilkinson	Numerical Analysis	1991	Robin Milner	ML, type theory, CCS
1971	John McCarthy	Lisp	1992	Butler Lampson	Workstations
1972	Edsger Dijkstra	Algol, Science of programming	1993	Hartmanis & Stearns	Computational Complexity
1973	Charles Bachman	Database Technology	1994	Feigenbaum & Reddy	Large-scale AI
1974	Donald Knuth	The Art of Computer Programming	1995	Manuel Blum	Complexity & Cryptography
1975	Newell & Simon	AI & Cognition	1996	Amir Pnueli	Temporal Logic
1976	Rabin & Scott	Automata Theory	1997	Doug Engelbart	Interactive Computing
1977	John Backus	Fortran, Functional Programming	1998	Jim Gray	Transaction Processing
1978	Bob Floyd	Parsing, Semantics, Verification	1999	Fred Brooks	Software Engineering
1979	Ken Iverson	APL	2000	Andrew Yao	Complexity-based Theory
1980	Tony Hoare	Definition & design of languages	2001	Dahl & Nygaard	Object-oriented Programming
1981	Edgar Codd	Relational Databases	2002	R-S-A	Public-Key Cryptography
1982	Stephen Cook	Complexity of Computation	2003	Alan Kay	Smalltalk
1983	Thompson & Ritchie	Unix (also C)	2004	Cerf & Kahn	Internetworking
1984	Niklaus Wirth	Algol-W, Pascal, Modula	2005	Peter Naur	Languages, Compilers, ALGOL 60
1985	Dick Karp	Theory of NP-Completeness	2006	Fran Allen	Optimizing Compilers
1986	Hopcroft & Tarjan	Algorithms & Data Structures	2007	C E S	Model Checking

2008: Barbara Liskov
2013: Leslie Lamport

Why study compiler construction?

- Compiler construction is successful
  - clear problem
  - proper structure of the solution
  - judicious use of formalisms
  - … but some nitty-gritty programming
- Wider application
  - many conversions can be viewed as compilation
  - some techniques are reusable in other contexts

Become a better programmer

- Deepen your understating of programming languages, operating systems and computer architectures
- Increase your **ability to adapt quickly to new** programming languages, machines, compilation modes
- Collaborate on a software project
- Gain experience with standard software engineering practices
- Every person in this class will build a parser some day … or wish she knew how to build one…
- Become a compiler writer
- Become software verification engineer or security engineer

ADMIN

Who?

Lecturer

Greta Yorsh
g.yorsh@qmul.ac.uk

Demonstrators

Julian Nagele
j.nagele@qmul.ac.uk

Yu-Yang Lin Hou
yu-yang.lin@qmul.ac.uk

Students

Your Name

When and Where?

- **Lectures**: Wednesdays 9-11 (Laws:1.12)
- **Tutorials**: Mondays 13-14 (BR 3.02)
- **Labs:** Mondays 14-15 (ITL-1F LAB)

- **Office hours**: Wednesdays 11-13 (CS430) (**email me** to confirm)

Assessment: How?

- 50% final written exam
- 50% coursework: four programming assignments

  - **PA0**: individual, 5%
    - 100% pass rate on test suite
  - **PA1-PA3**: team, 15% each
    - 80% pass rate on test suite
    - 20% code review

Classroom **O**bject **O**riented **L**anguage (Cool)

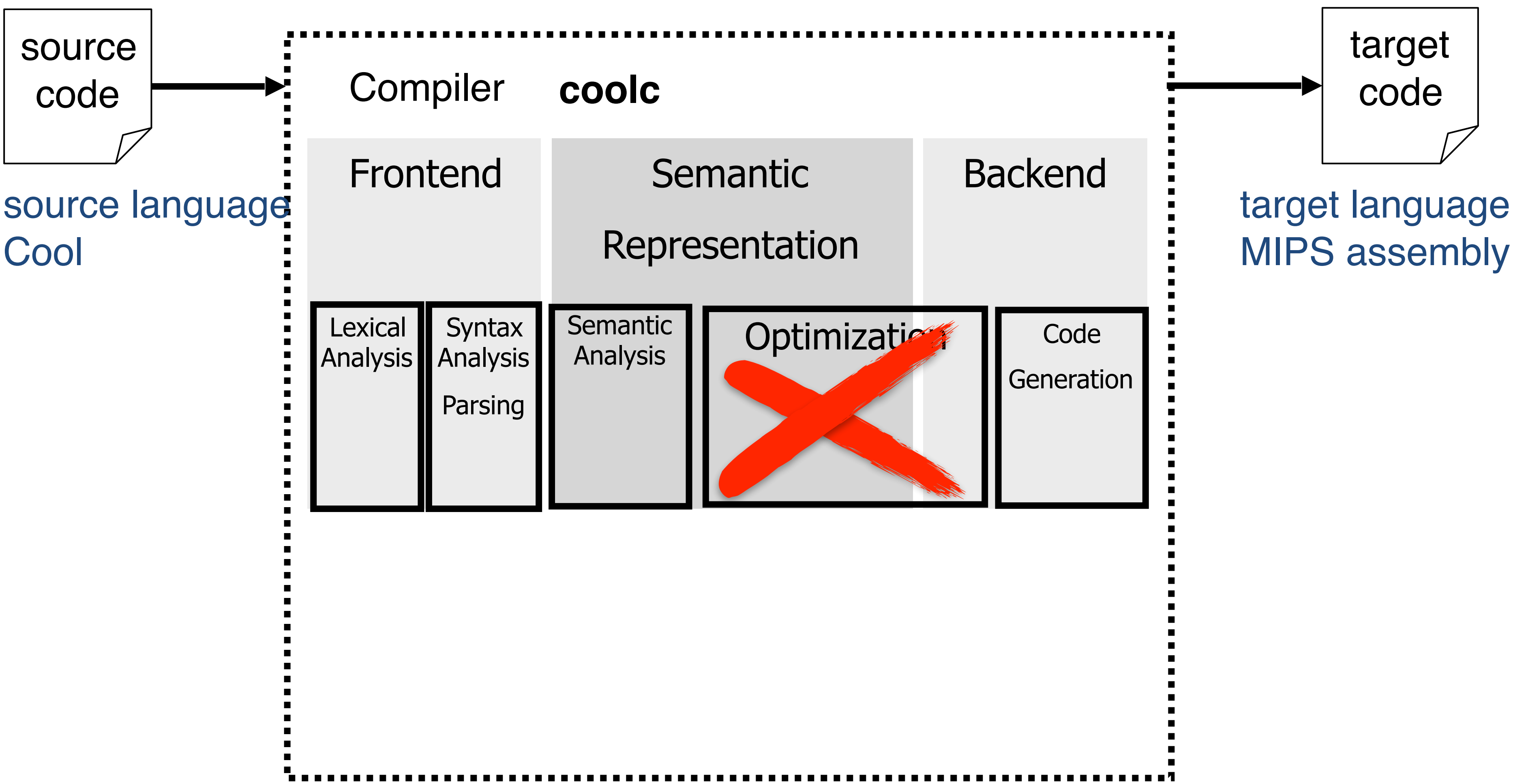
- Designed by Alex Aiken from Stanford for teaching compiler construction
- Supports modern language features
  - abstraction, reuse (inheritance), static typing, memory management
- Many features are left out
  - feasible to implement in a semester

Programming Assignments

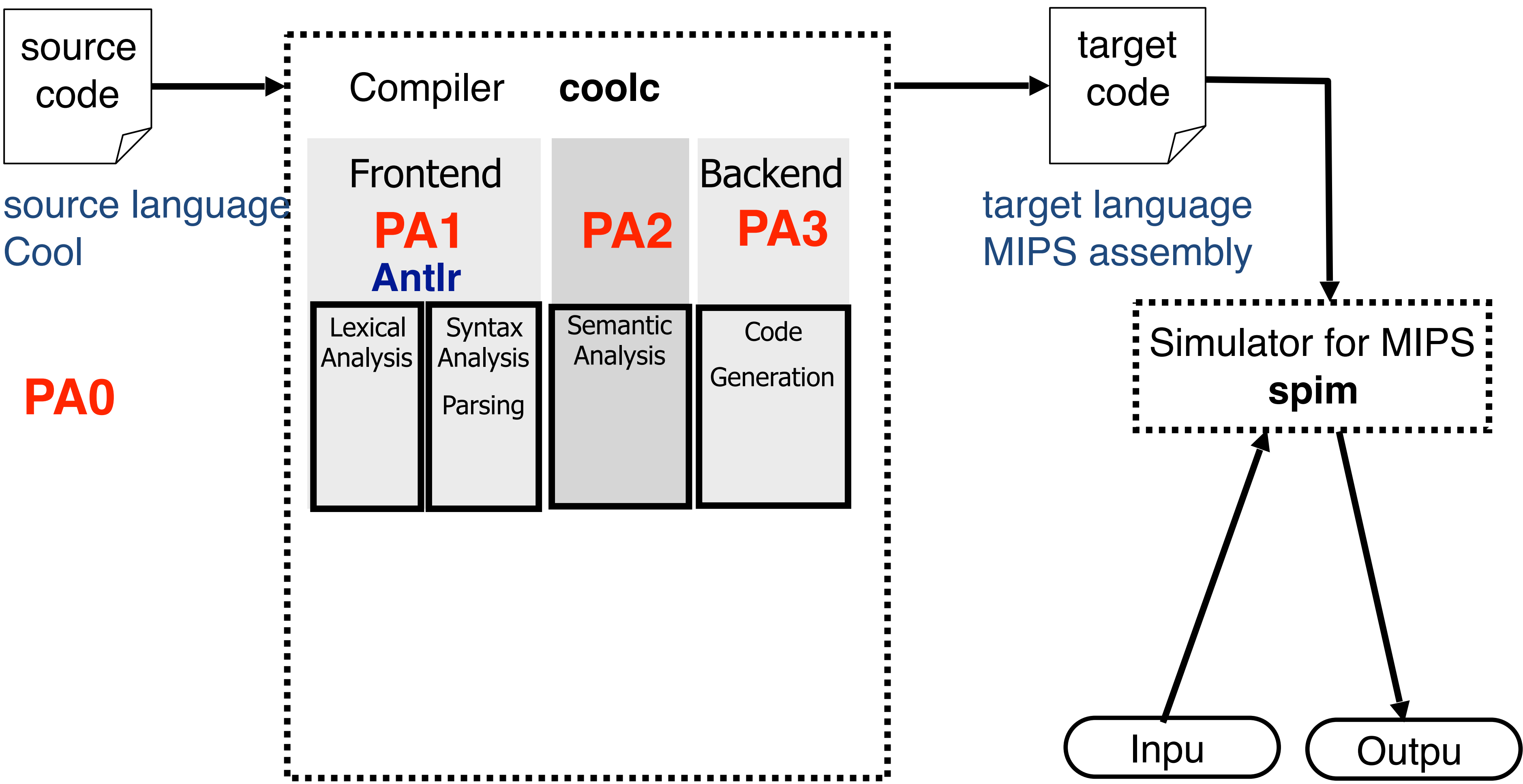
Goal: build a compiler from Cool to MIPS assembly

- implement in Java (not from scratch)
- split into **independent** assignments
- work in **teams** of 2-3 students
- all team members get the same mark
- use source control git
- policy regarding other collaborations

Cool compiler overview



Cool compiler overview



Exam

- Mock exam with model answers
  - will be available at the end of the semester
- Don’t worry too much
  - if you attend lectures and labs, and actively participate in programming assignments, you should do well in the exam
  - try to keep up with the material…

What will help us?

- **Focus on understanding and not on your grade**
- Come to lectures and labs
- Participate
- **Start programming assignments early: sooner than you think you need to**
- Actively participate in preparing all team programming assignments
- Meet deadlines
- Follow instructions (we have to be able to run your code)
- **Ask and answer questions**

Online resources

- QM+ forum: ask and answer questions
- QM+ resources
  - slides
  - module information: deadlines, guidelines, etc
  - programming assignment handouts
  - documentation
    - Cool reference manual, support code, runtime
    - Antlr4, SPIM
- QMUL GitHub https://github.research.its.qmul.ac.uk/ecs652
  - distro: reference compiler, source code templates, tests, …
  - your repository for development and submission
- External resources

Your slides don't have everything you say written on them

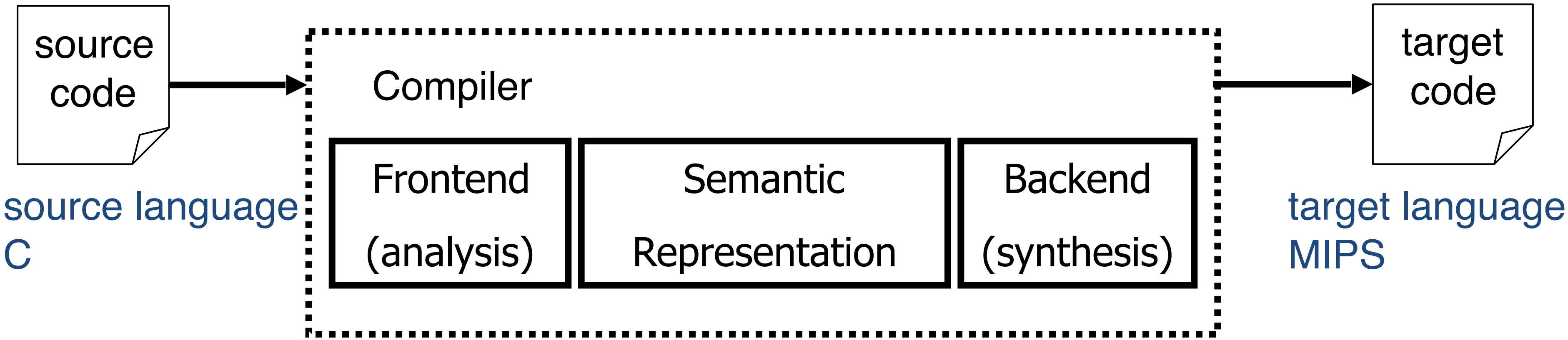
- Yes, I know, this is by design
- Slides are a teaching aid
- Not a replacement for coming to lectures
- If you don't attend lectures or attend and don't listen, you will miss some things
- Slides are enough to teach from, but not enough to learn from
- If you want slides that have all the material written on them nicely, that format is **available** and commonly known as a **textbook**
- See how horrible this slide is? You won't see many slides with so much text as this one in the rest of the course

Textbooks
(recommended, not required)

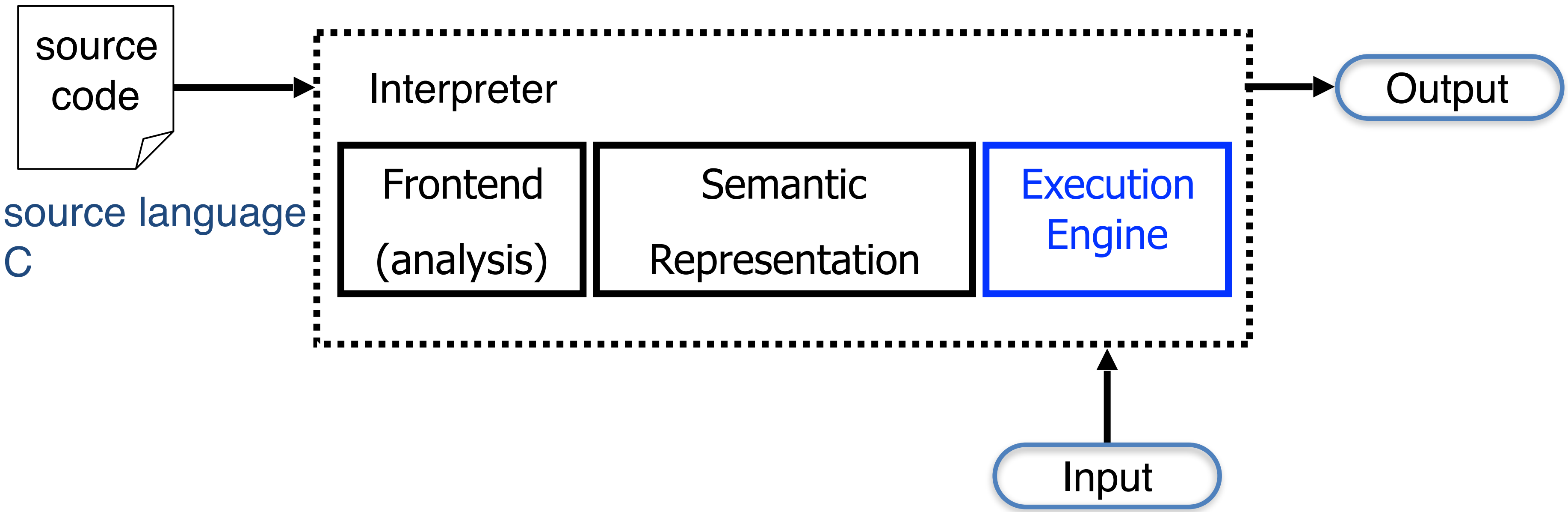
- **Dragon Book** “Compilers: principles, techniques and tools”
  by Aho, Lam, Sethi, and Ullman
- **Tiger book** “Modern compiler implementation in Java”
  by Appel
- **Whale book “**Advanced compiler design and implementation”
  by Muchnick
- **Ark book** “Engineering a Compiler”
  by Cooper and Torczon
- “Optimizing Compilers for Modern Architectures”
  by Allen and Kennedy

COMPILER VS INTERPRETER

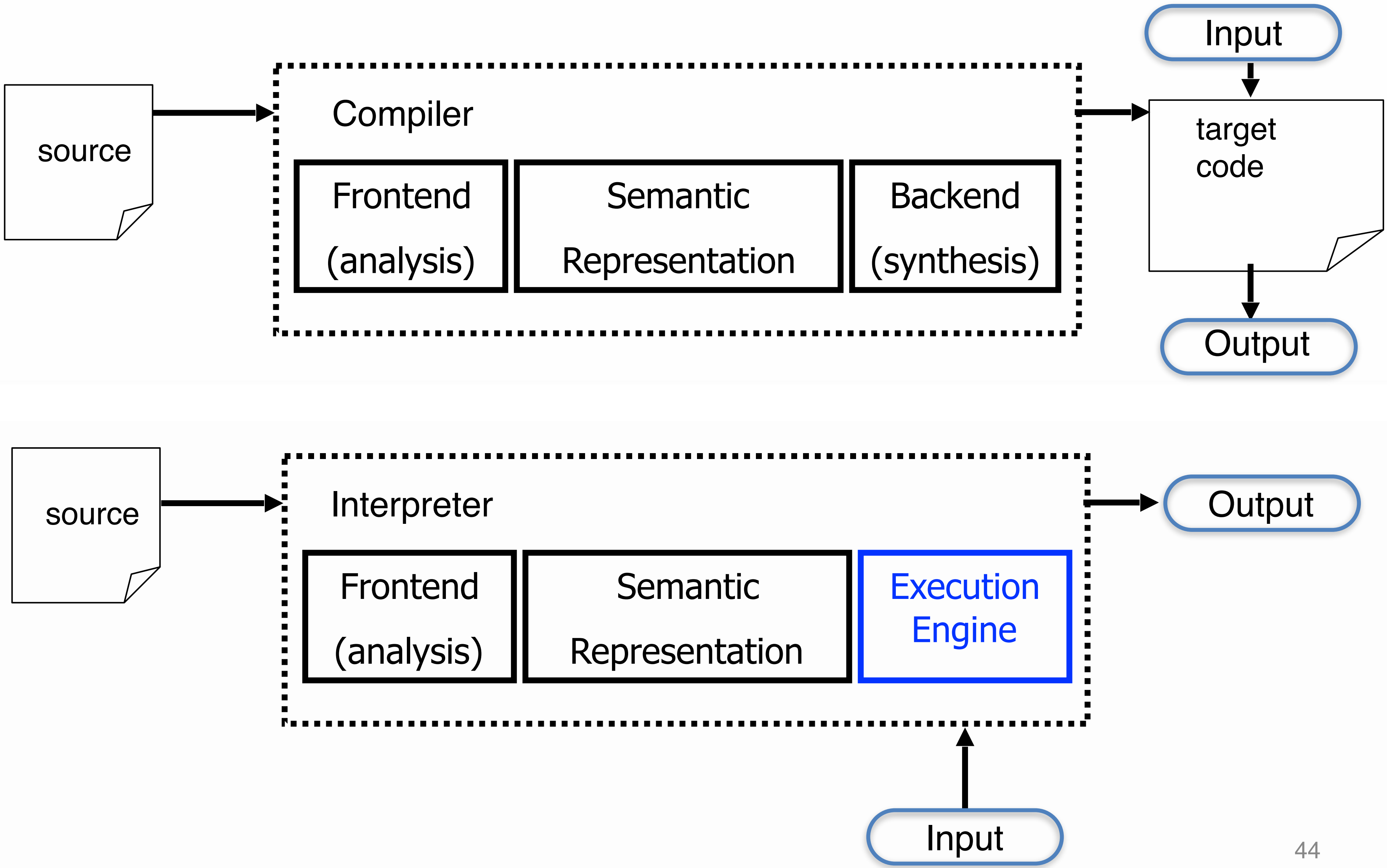
Anatomy of a compiler



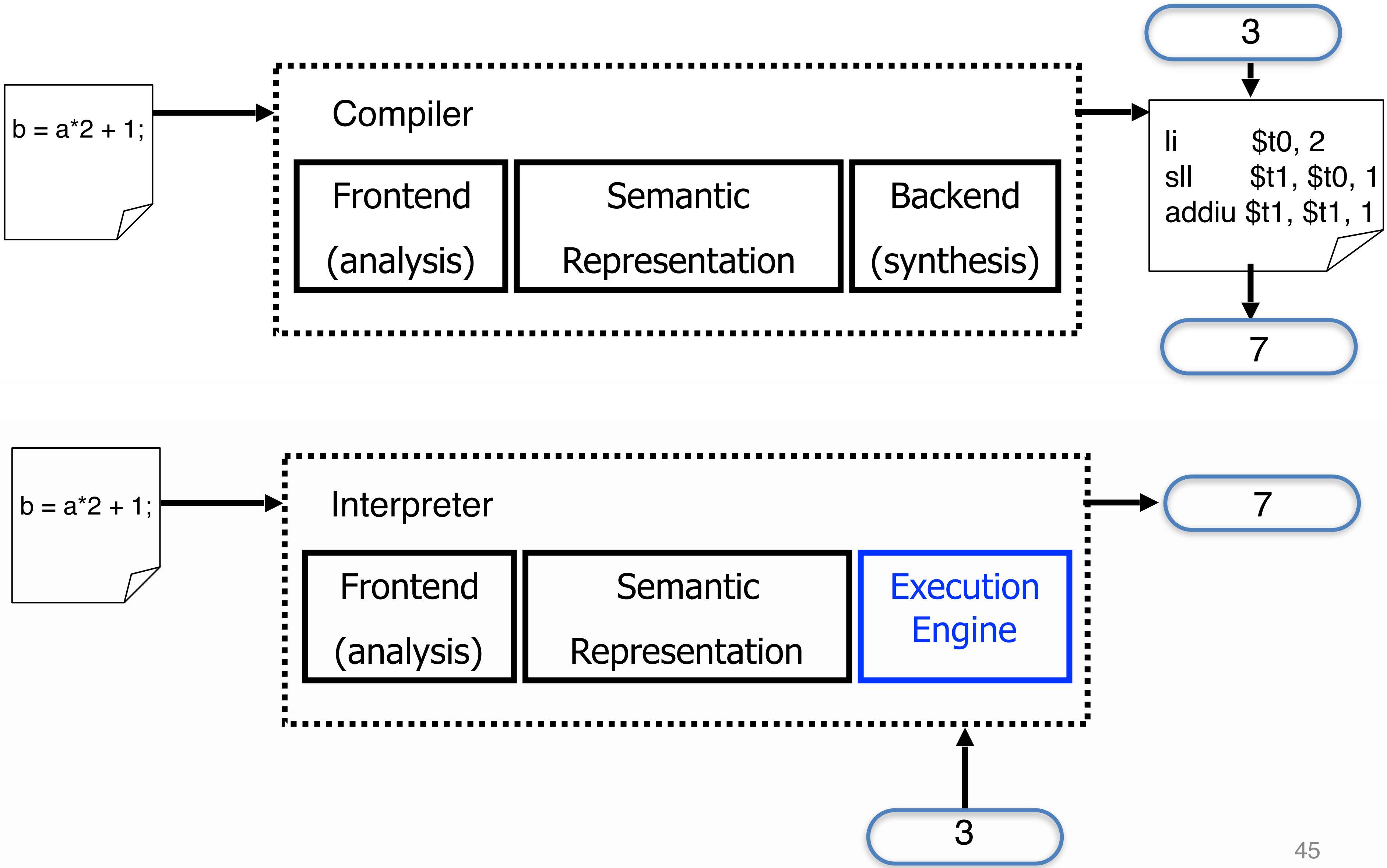
Interpreter



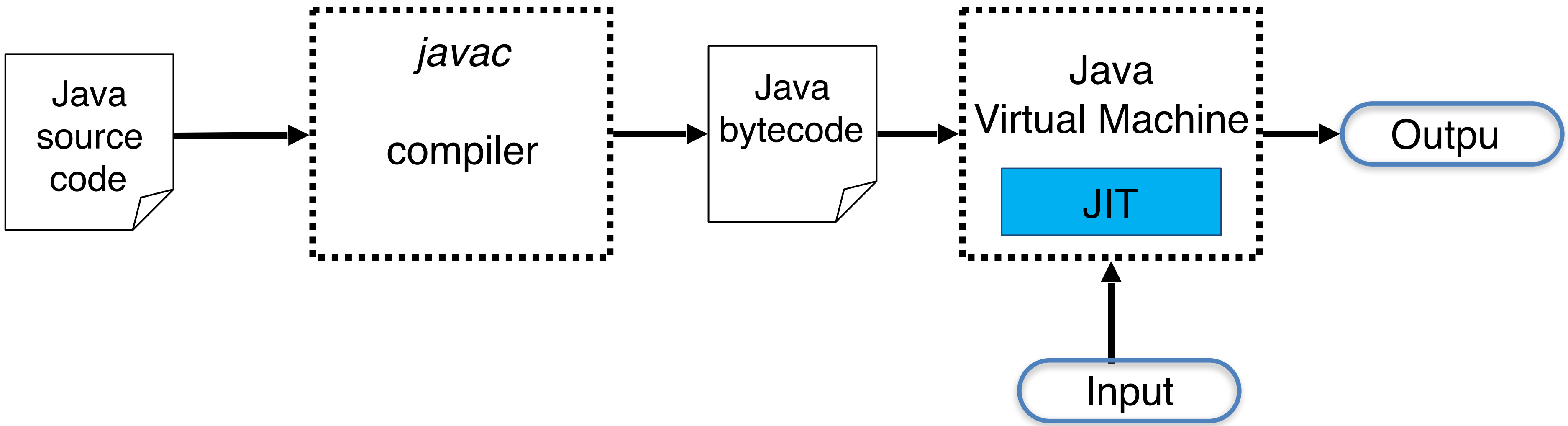
Compiler vs. Interpreter



Compiler vs. Interpreter



Just-in-time compiler for Java

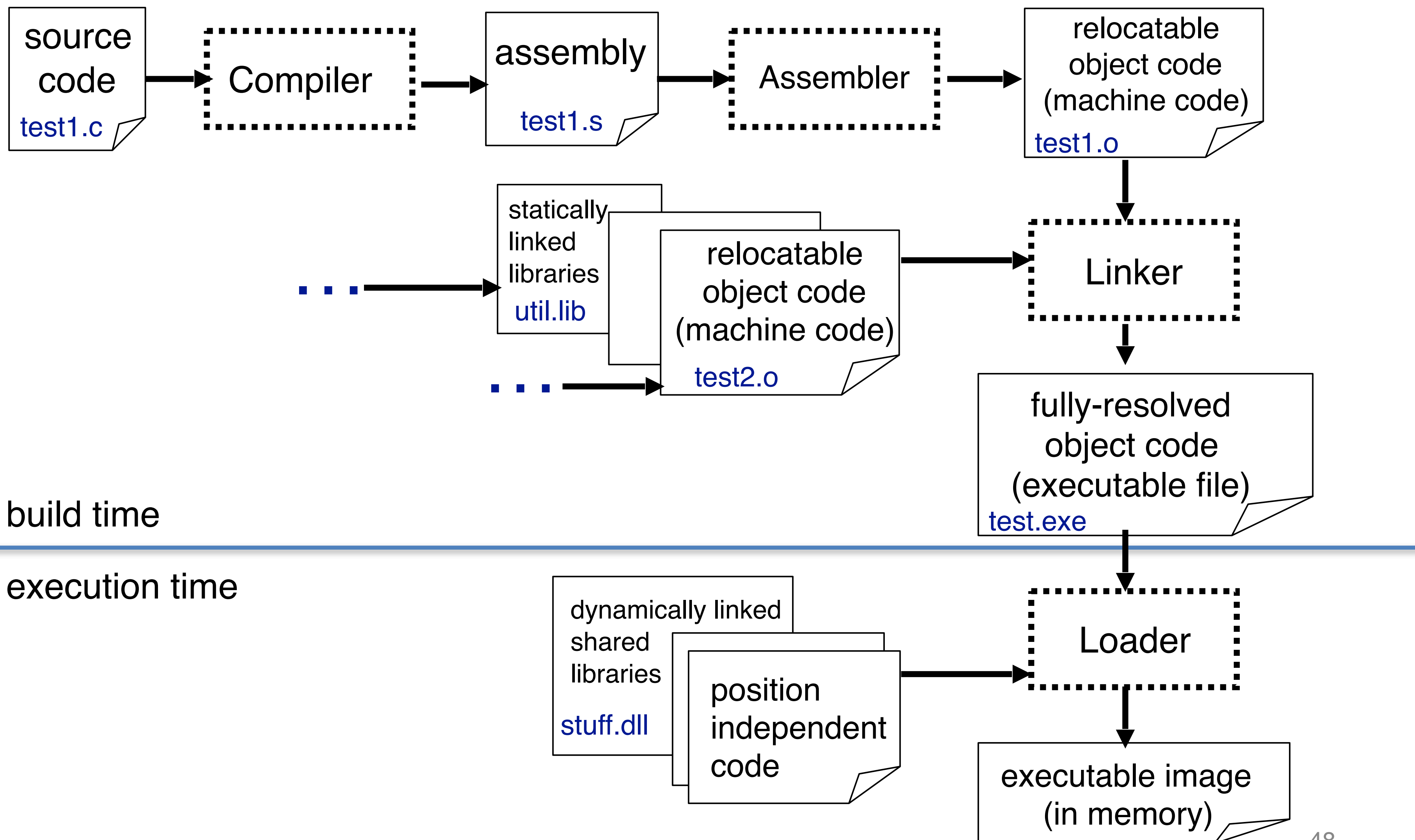


Just-in-time (JIT) compilation: bytecode interpreter in the VM identifies "hot" program fragments and compiles them to avoid expensive re-interpretation.

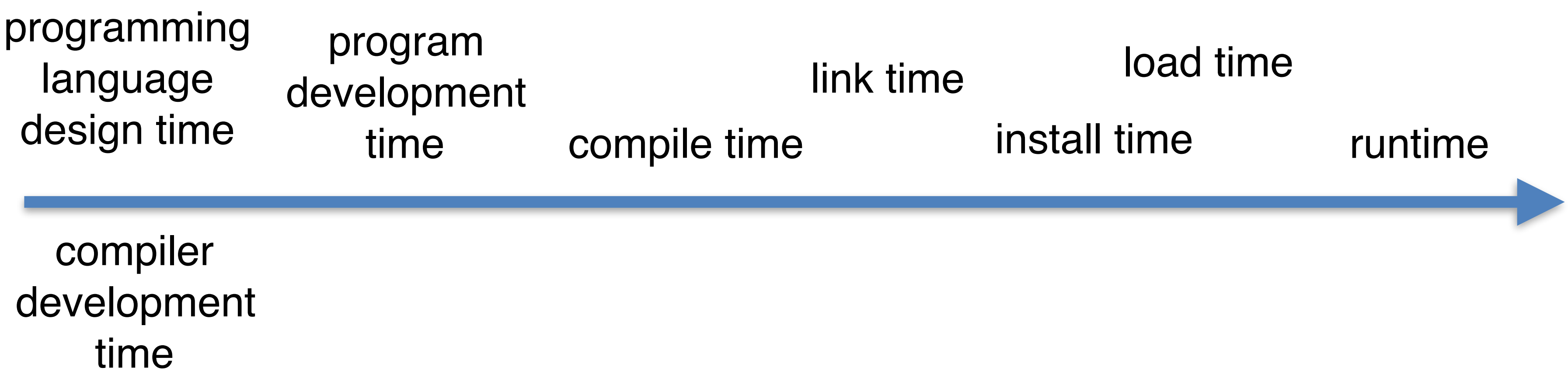
Just-in-time compiler for Javascript

- The V8 execution engine **compiles JavaScript to native machine code** before executing it
- Instead of interpreting the JS code
- The compiled code is optimized **dynamically at runtime**, based on runtime behavior

From source to running program



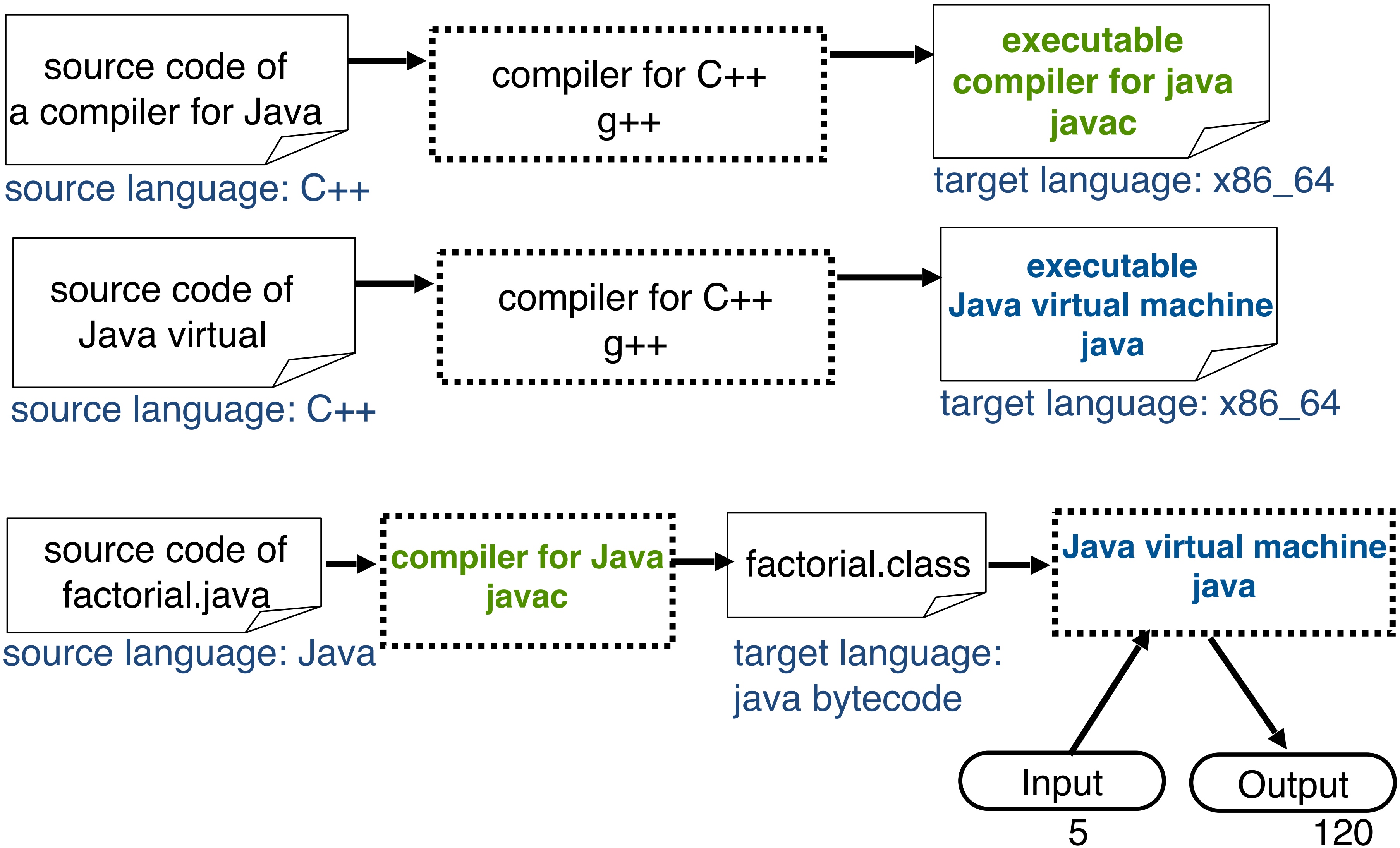
Time of events: trade offs



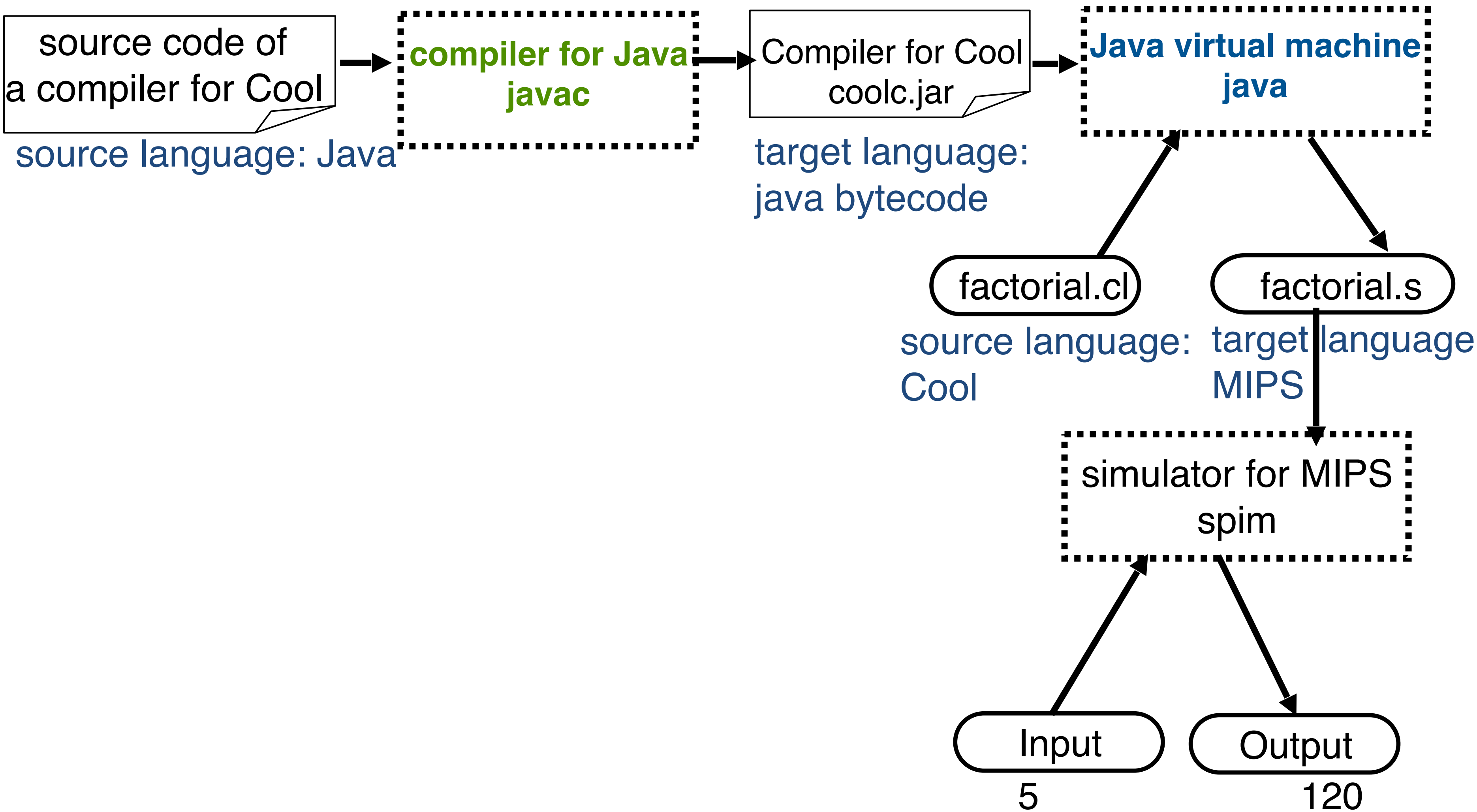
Bootstrapping

- Both compilers and interpreters are programs written in high-level languages
- How to compile the compiler/interpreter?

Bootstrapping compiler/interpreter

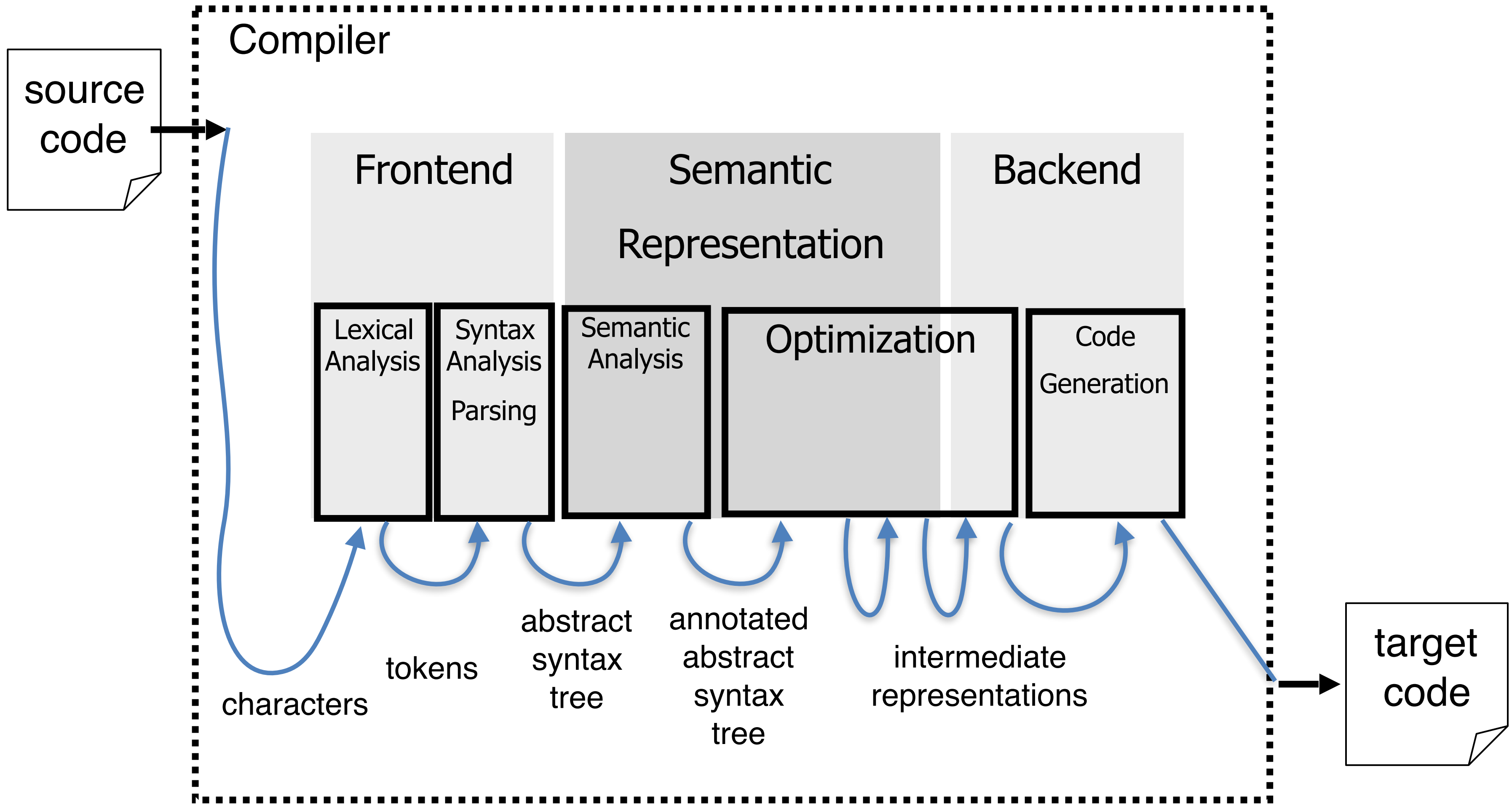


Bootstrapping Cool compiler

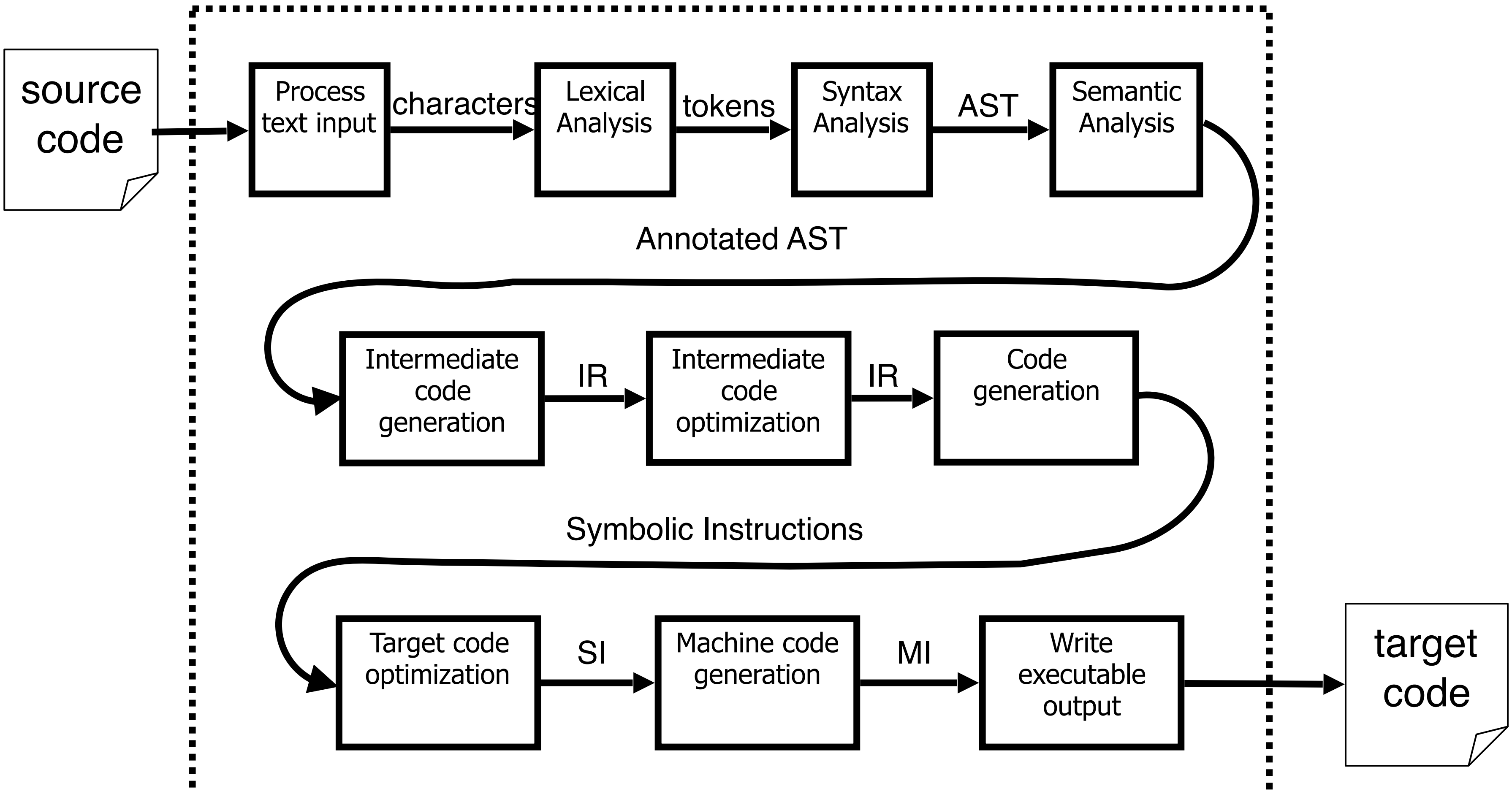


WHAT IS A GOOD COMPILER?

Anatomy of a modern compiler



The real anatomy of a modern compiler



Compiler correctness

- Generated code correctly implements the source code

- Concerns with correctness of translation
- Different from code correctness
- Compilers do not guarantee to generate “correct code”
- For example, consider a program that throws a NullPointerException at runtime

Compiler design goals

- Correctness: generated code correctly implements the source code
- Metrics for generated code
  - performance/speed
  - size
  - power consumption/energy efficiency
  - security/reliability
  - easy to debugging
  - portable
- Metrics for compilers
  - fast/efficient compilation
  - good error reporting

Optimizations

- "Optimal code" is out of reach
  - many problems are undecidable or too expensive
  - use approximation and/or heuristics
  - optimizations must guarantee compiler correctness
  - should (mostly) improve code
- Majority of compilation time is spent in optimizations
- Leverage compile-time information to save work at runtime (precompute)

Example optimizations

- Loop optimizations: hoisting, unrolling
- Peephole optimizations
- Constant propagation
- Dead code elimination
- Instruction selection: convert IR to machine instructions
- Instruction scheduling: reorder instructions
- Register allocation: assign variables to memory locations
  - optimal register assignment is NP-Complete
  - in practice, known heuristics perform well
- Modern architectures include challenging features
  - multicore
  - memory hierarchies

Compiler construction tools

- Parts of the compiler are automatically generated from specification
  - simplify compiler construction
  - less error prone
  - more flexible
  - use of pre-canned tailored code
  - use of dirty programming tricks
  - reuse of specification

Compiler construction tools

- Lexical analysis generators
  - lex, flex, jflex, antlr
- Parser generators
  - yacc, bison, java_cup, antlr
- Syntax-directed translators
- Dataflow analysis engines

Summary

- Compiler is a **program** that translates code from **source** language to **target** language
- Compilers play a central role
  - bridge from high-level programming languages to machines
  - many useful techniques
  - many useful tools (e.g., lexer/parser generators)
- Compiler vs Interpreter
- Just-In-Time compilation
- Time of events: compiler, linker, loader, runtime
- Bootstrapping a compiler
- Compiler constructed from modular phases