Contents

Introduce the phases of a compiler.

CS 536 / Spring 2020

Introduction to programming languages and compilers

Loris D’Antoni
loris-teach@cs.wisc.edu

About me

PhD at University of Pennsylvania

Joined University of Wisconsin in 2015

Research in

- Program verification
- Program synthesis

http://pages.cs.wisc.edu/~loris/

About the course

We will study compilers

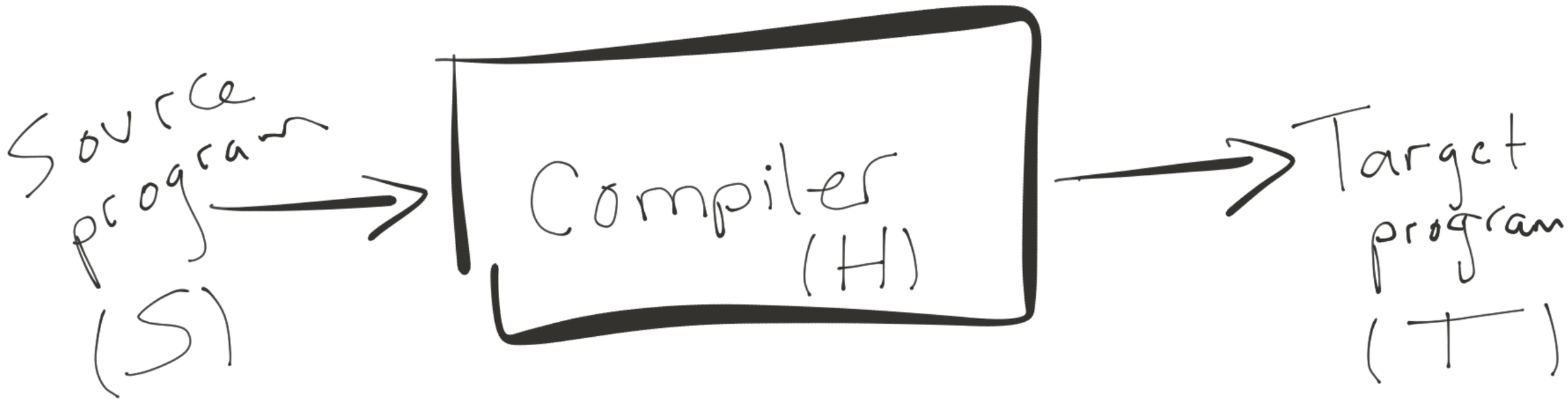
We will understand how they work

We will build a **full** compiler

We will have fun

Course Mechanics

- Home page: http://pages.cs.wisc.edu/~loris/cs536/
- Piazza: https://piazza.com/wisc/spring2020/cs536/
- Workload:
  - 6 Programs (40% = 5% + 7% + 7% + 7% + 7% + 7%)
  - 10 short homeworks (20%)
  - 2 exams (midterm: 20% + final: 20%)
- For information about late policy, collaboration, etc., see http://pages.cs.wisc.edu/~loris/cs536/info.html

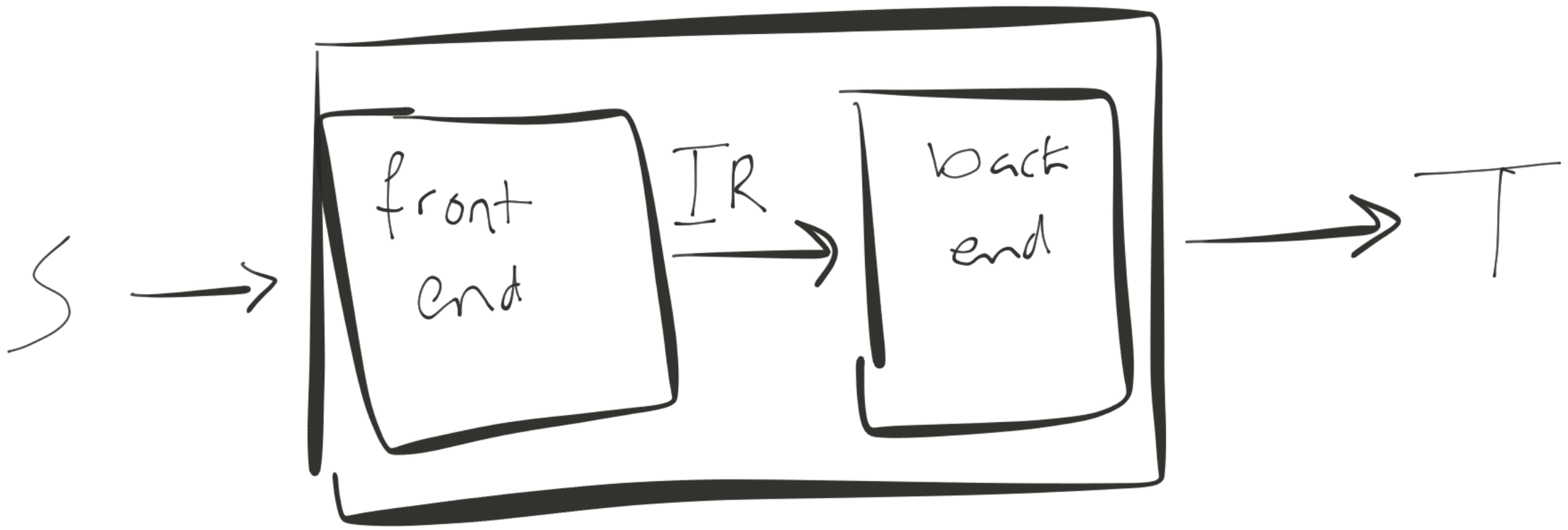


A compiler is a

- recognizer of language *S*
- a translator from *S* to *T*
- a program in language H

S: source program.
T: target program(machine code).

What will we name S? **...**



front end = understand source code *S*

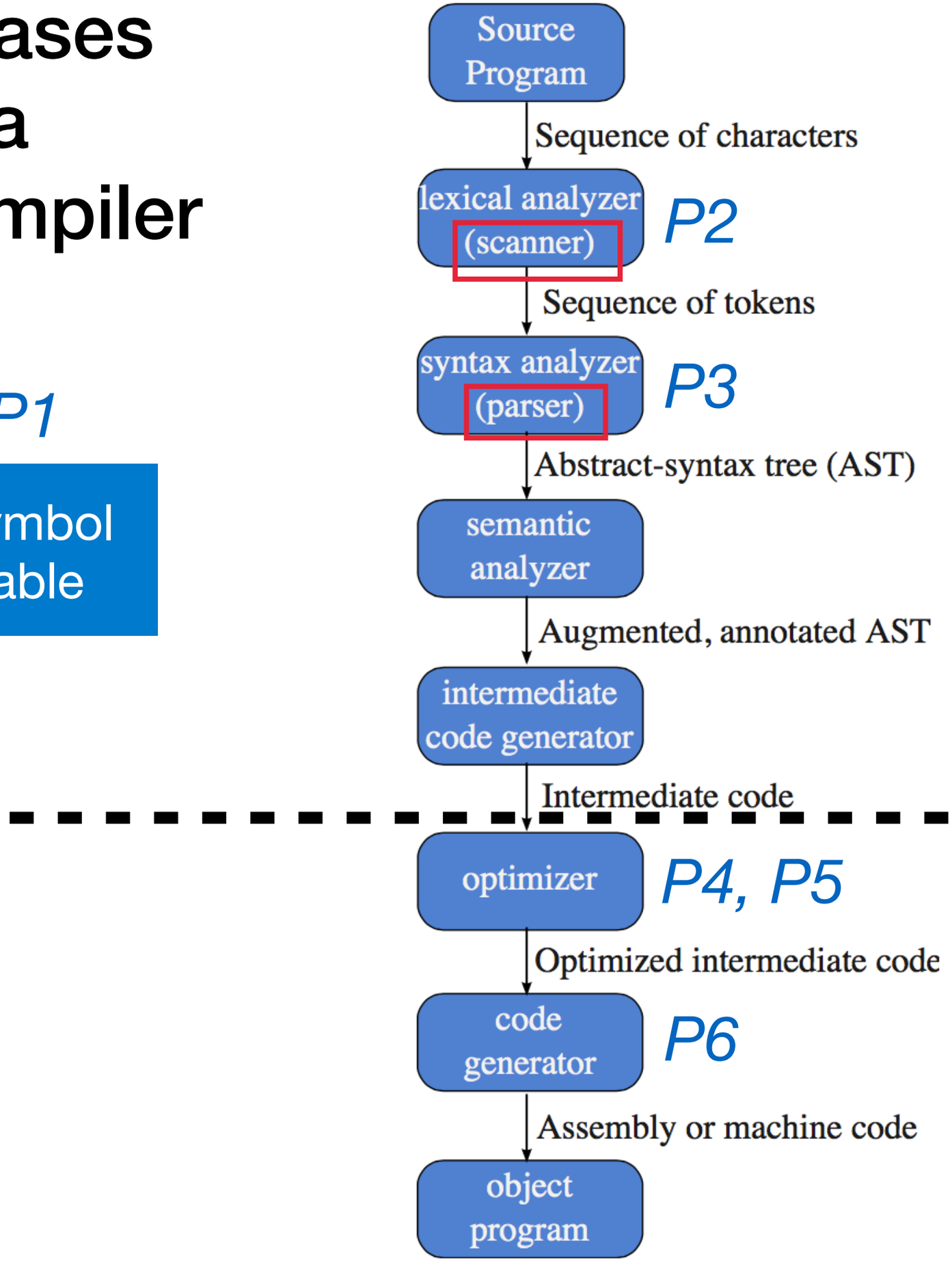
IR = intermediate representation

back end = map IR to T

Phases of a compiler

P1

Symbol table



Scanner

Input: characters from source program

Output: sequence of tokens

Actions:

group chars into lexemes (tokens)

Identify and ignore whitespace, comments, etc.

What errors can it catch?

bad characters such as ^ Some of unicode characters are not allowed.

unterminated strings, e.g., “Hello

int literals that are too large

Parser

Input: sequence of tokens from the scanner

Output: AST (abstract syntax tree)

Actions:

groups tokens into sentences

What errors can it catch?

syntax errors, e.g., x = y *= 5

(possibly) *static semantic* errors, e.g., use of undeclared variables

Semantic analyzer

Input: AST

Output: annotated AST

Actions: does more static semantic checks

Name analysis

process declarations and uses of variables

enforces scope

Type checking

checks types

augments AST w/ types

Semantic analyzer

Scope example:

```
...
{
    int i = 4;
    i++;
}
i = 5;
```

out of scope ⟶

Intermediate code generation

Input: annotated AST (assumes no errors)

Output: intermediate representation (IR)

e.g., 3-address code

instructions have 3 operands at most

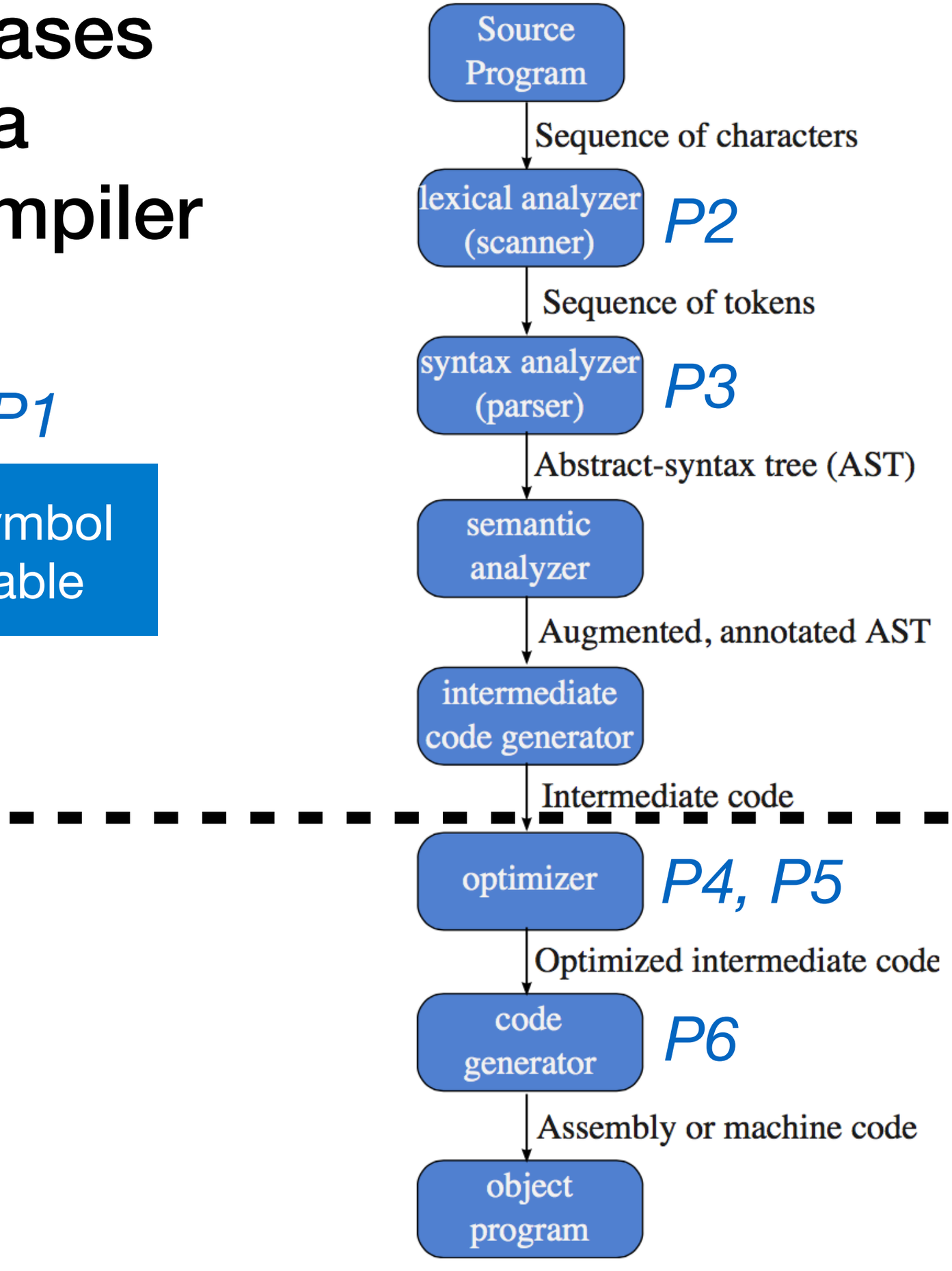
easy to generate from AST

1 instr per AST internal node

Phases of a compiler

P1

Symbol table



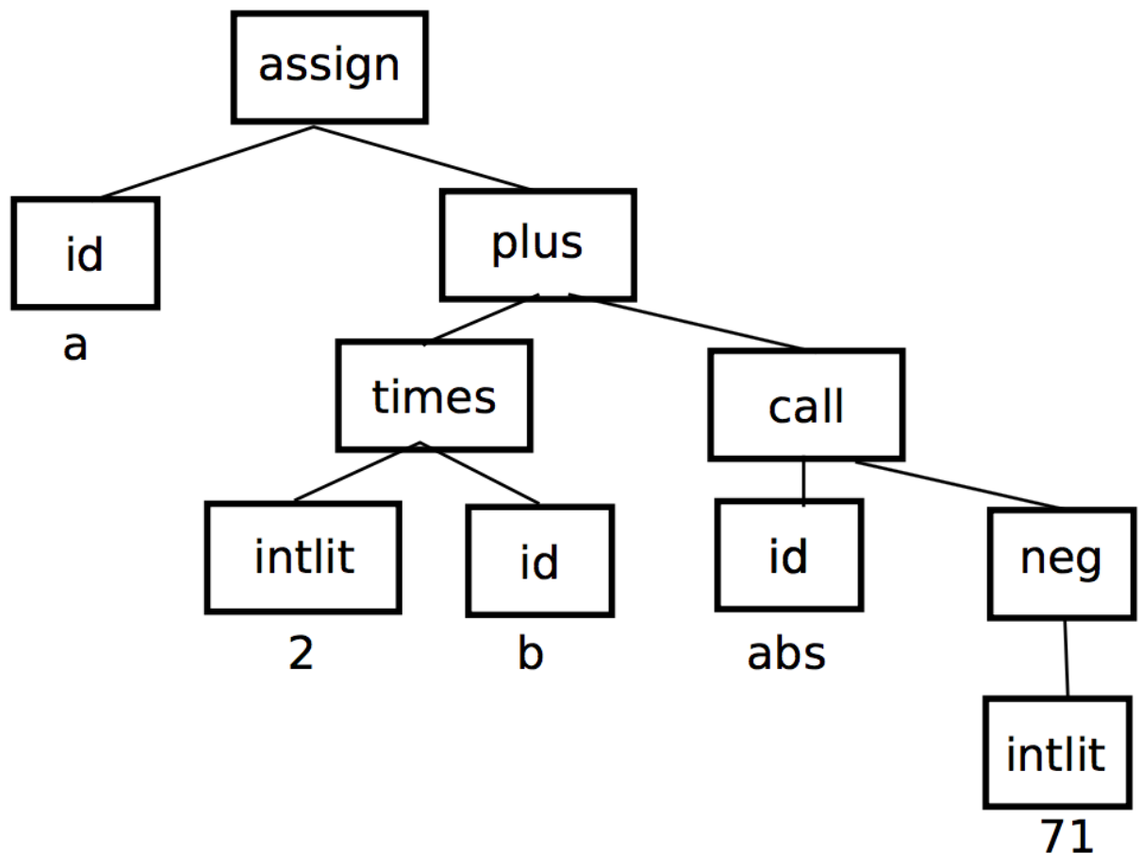
Example

scanner

```
a = 2 * b + abs(-71)
```

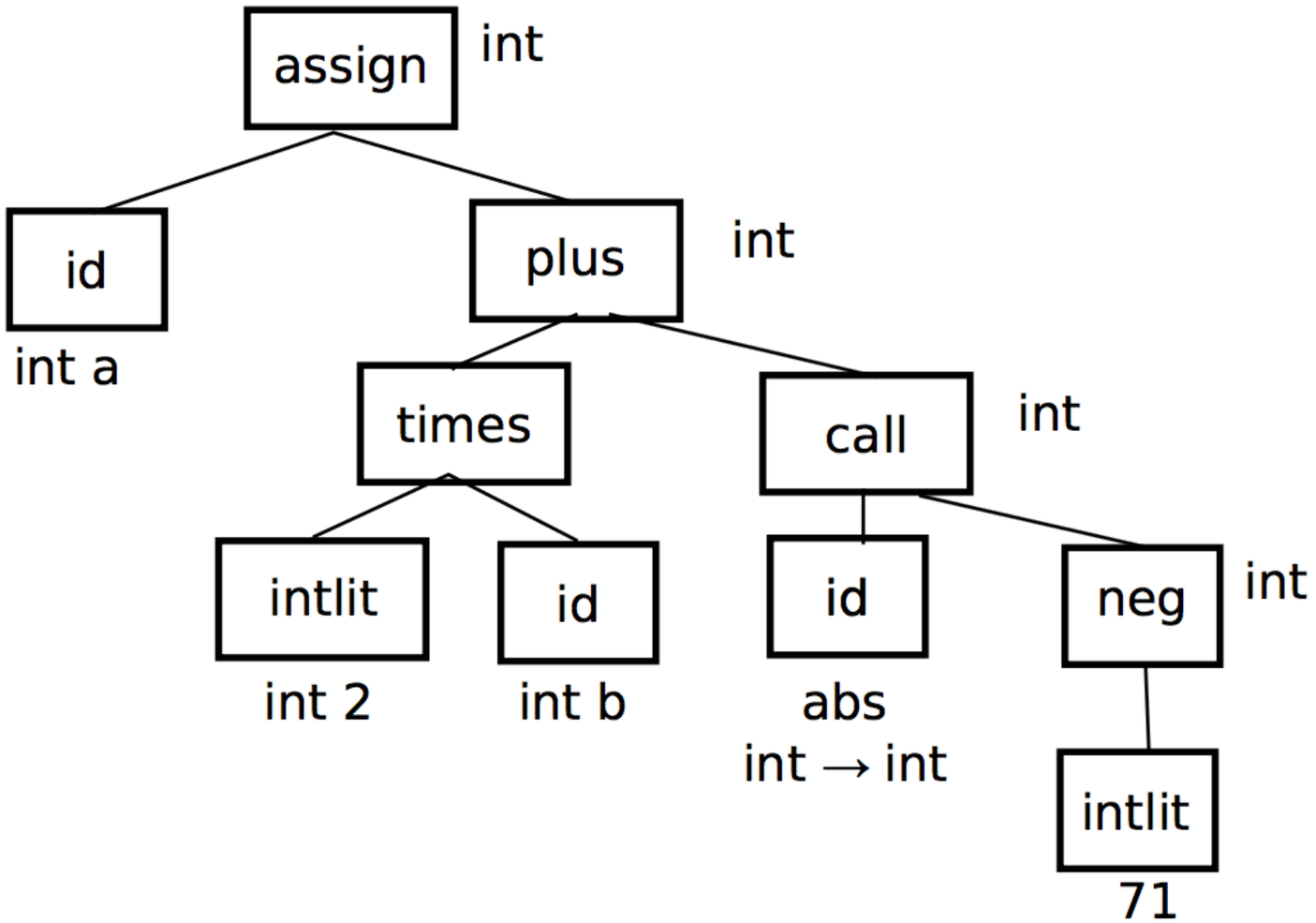
ident (a) | asgn | int lit (2) | times | ident (b) | plus | ident (abs) | lparens | minus | int lit (71) | rparens

parser



Example (cont'd)

semantic analyzer

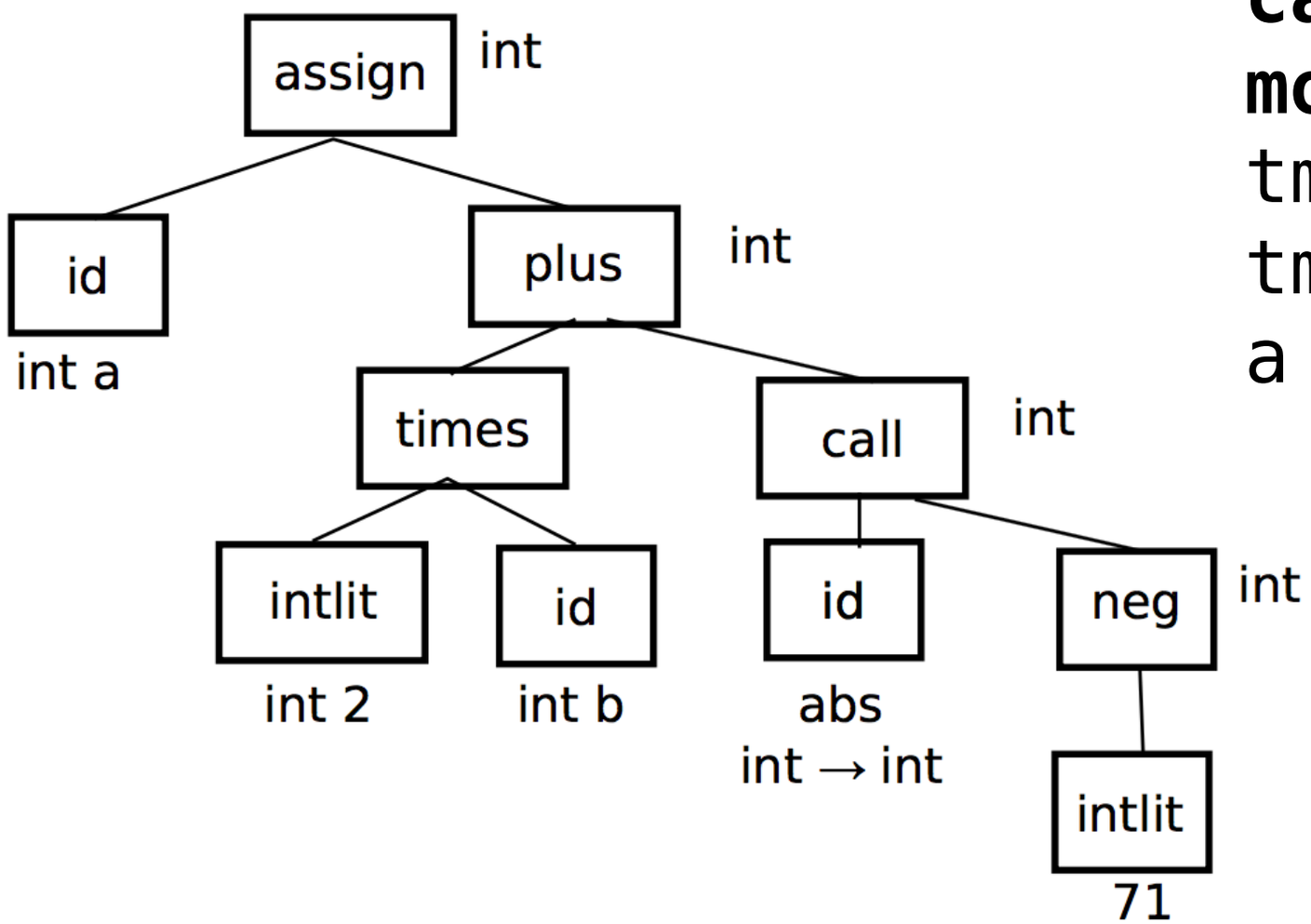


Symbol table

```
a     var    int
b     var    int
abs   fun    int->int
```

Example (cont'd)

code generation




```
tmp1 = 0 - 71
move tmp1 param1
call abs
move ret1 tmp2
tmp3 = 2*b
tmp4 = tmp3 + tmp2
a = tmp4
```

Optimizer

Input: IR

Output: optimized IR

Actions: *Improve code*

make it run faster; make it smaller

several passes: local and global optimization

more time spent in compilation; less time in execution

Code generator

Input: IR from optimizer

Output: target code

Symbol table

Compiler keeps track of names in

- semantic analyzer — both name analysis and type checking
- code generation — offsets into stack
- optimizer — def-use info

P1: implement symbol table

Symbol table

Block-structured language

java, c, c++

Ideas:

nested visibility of names (no access to a variable out of scope)

easy to tell which def of a name applies (nearest definition)

lifetime of data is bound to scope

Symbol table

```
int x, y;
void A() {
  double x, z;
  C(x, y, z)
}
void B() {
  C(x, y, z);
}
```

block structure: *need symbol table with nesting*

implement as list of hashtables

They are multiple symbol tables: one for each scope.