CS 536 / Fall 2021

Introduction to programming languages and compilers

Aws Albarghouthi
aws@cs.wisc.edu

About me

PhD at University of Toronto

Joined University of Wisconsin in 2015

Part of **madPL** group

- Program verification
- Program synthesis

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

About the course

We will study compilers

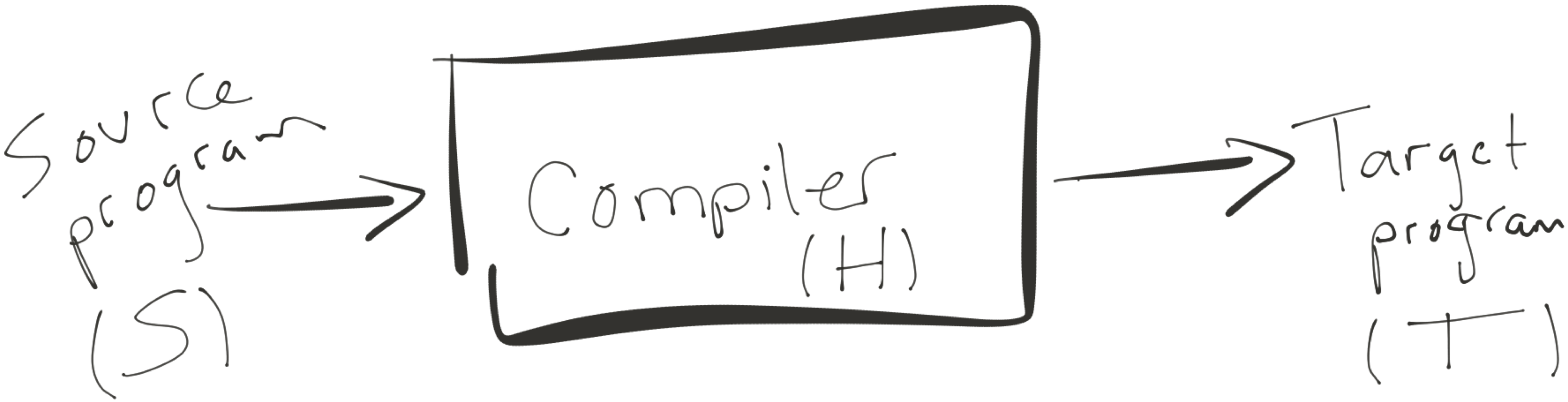
We will understand how they work

We will build a **full** compiler

We will have fun

Course Mechanics

- Home page: http://cs.wisc.edu/~aws/courses/cs536
- Workload:
    - 6 Programs (40% = 5% + 7% + 7% + 7% + 7% + 7%)
    - 2 exams (midterm: 30% + final: 30%)

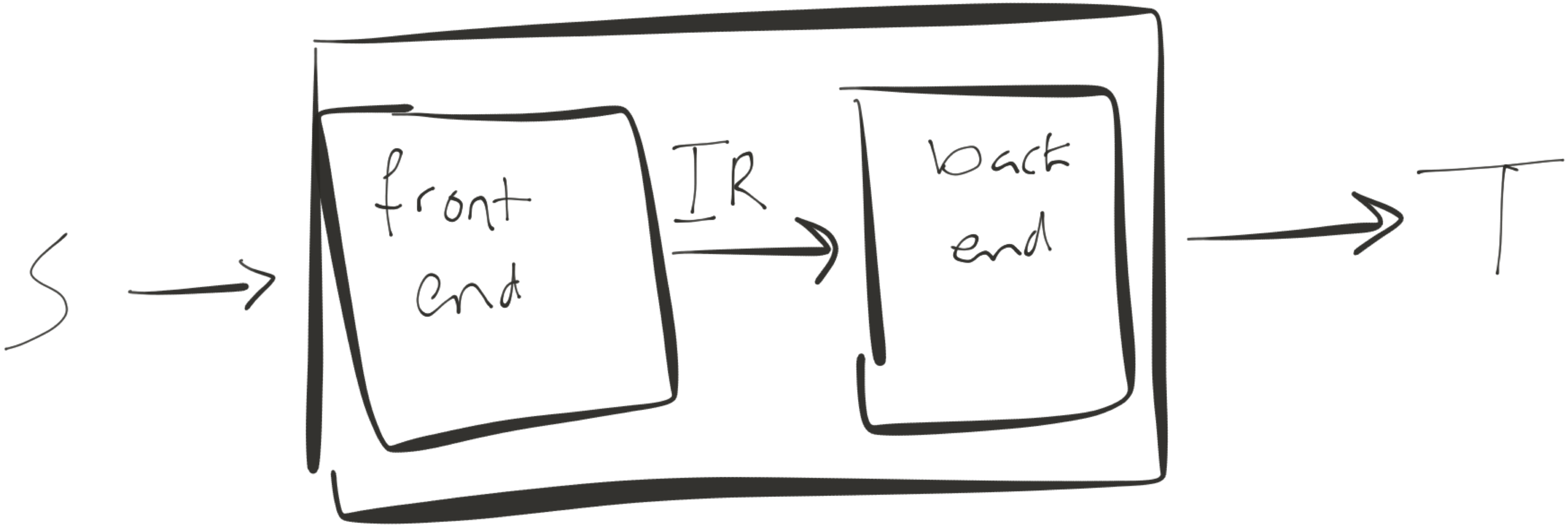


A compiler is a

recognizer of language *S*

a translator from *S* to *T*

a program in language H



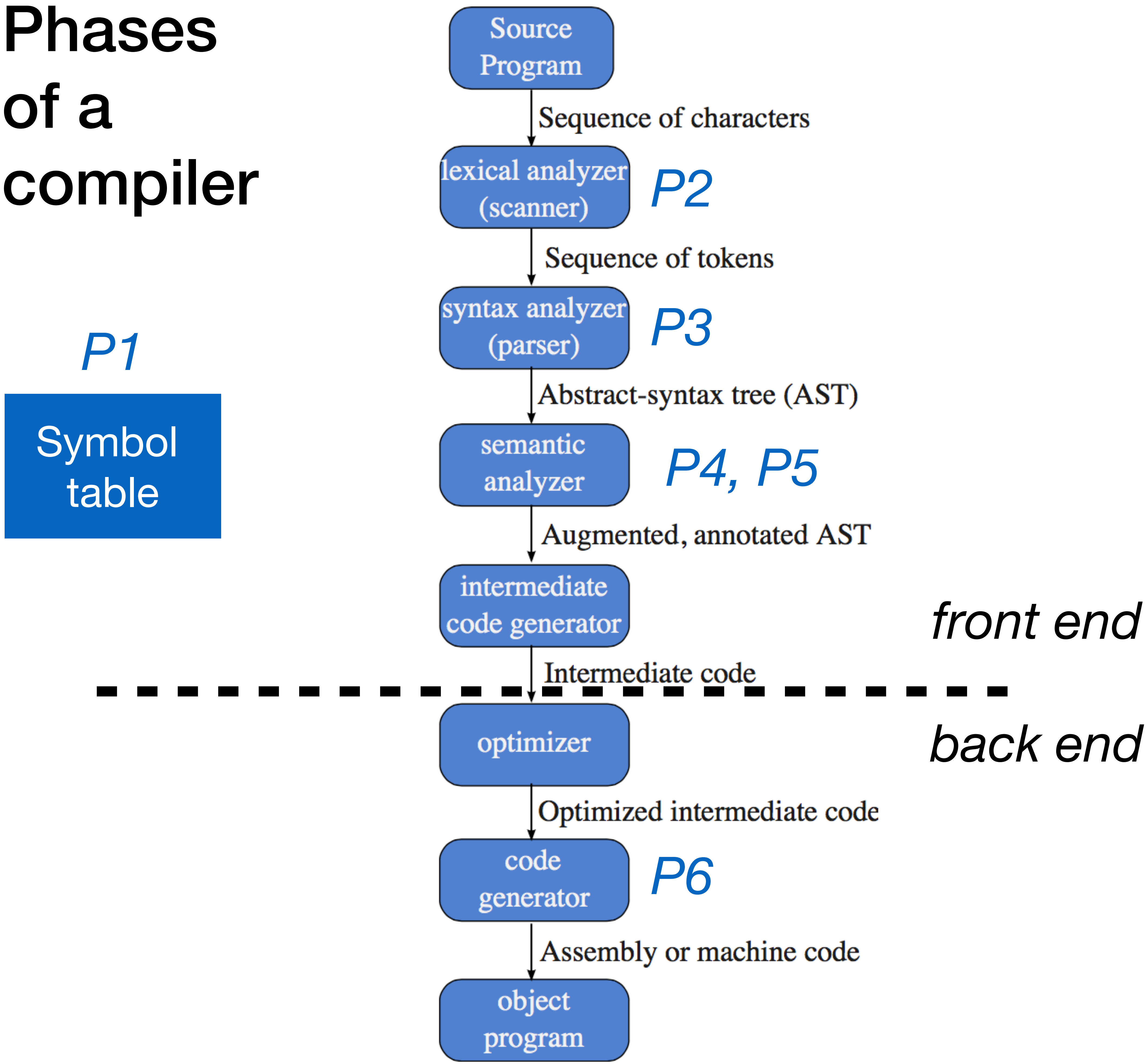
front end = recognize source code *S*;
map S to IR

IR = intermediate representation

back end = map IR to T

Executing the T program produces the same result as executing the S program?

Phases of a compiler



Scanner (P2)

Input: characters from source program

Output: sequence of tokens

Actions:

group chars into lexemes (tokens)

Identify and ignore whitespace, comments, etc.

Error checking:

bad characters such as ^

unterminated strings, e.g., “Hello

int literals that are too large

Example

scanner

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

a	=	2	*	b	+	abs	(	-	71	)
ident (a)	asgn	int lit (2)	times	ident (b)	plus	ident (abs)	lparens	minus	int lit (71)	rparens

Whitespace (spaces, tabs, and newlines) filtered out

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

The scanner's output is still the sequence

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

Parser (P3)

Input: sequence of tokens from the scanner

Output: AST (abstract syntax tree)

Actions:

groups tokens into sentences

Error checking:

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

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

Semantic analyzer (P4,P5)

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 (P4,P5)

Scope example:

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

out of scope

Intermediate code generation

Input: annotated AST (assumes no errors)

Output: intermediate representation (IR)

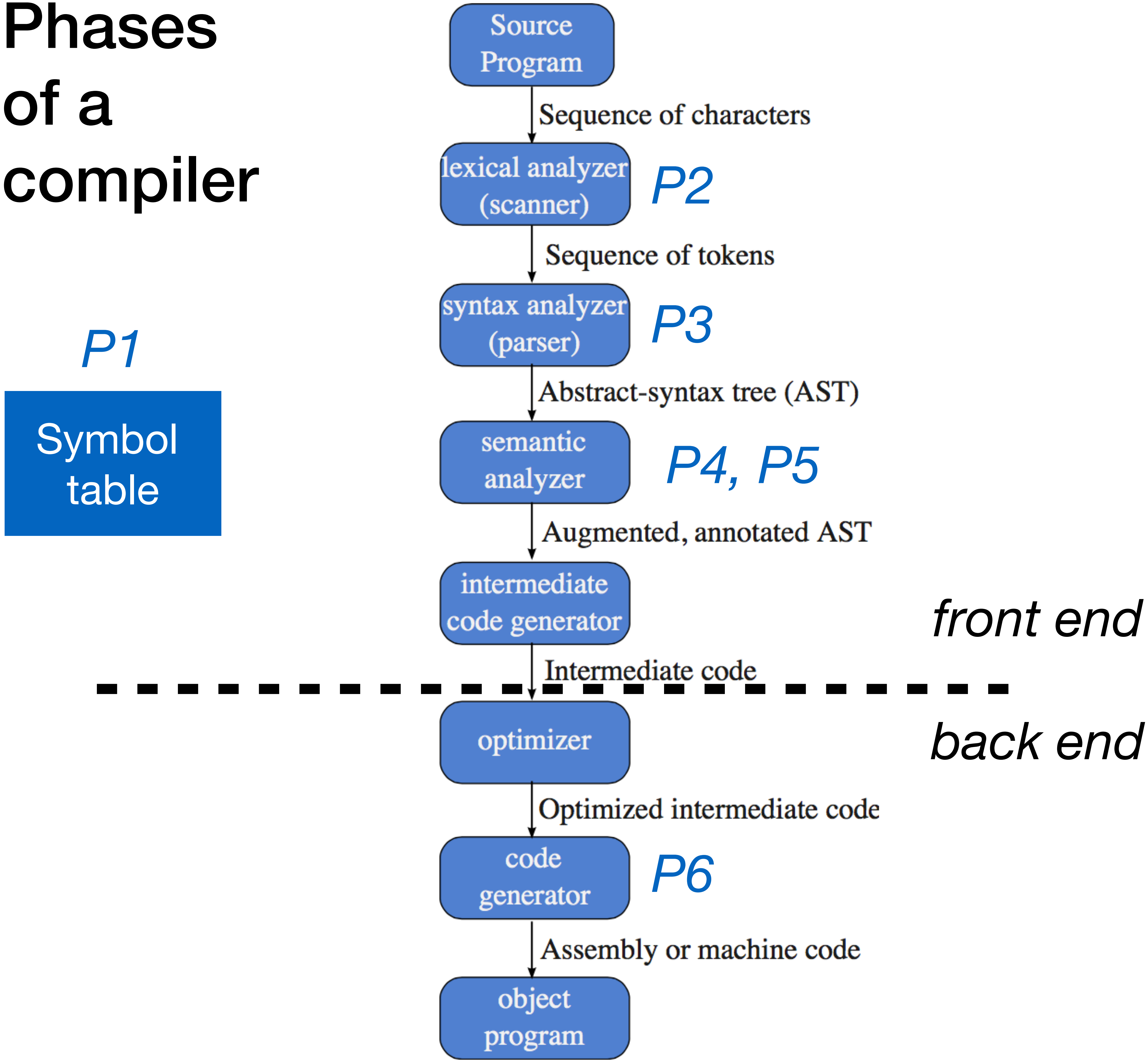
e.g., 3-address code

instructions have 3 operands at most

easy to generate from AST

1 instr per AST internal node

Phases of a compiler



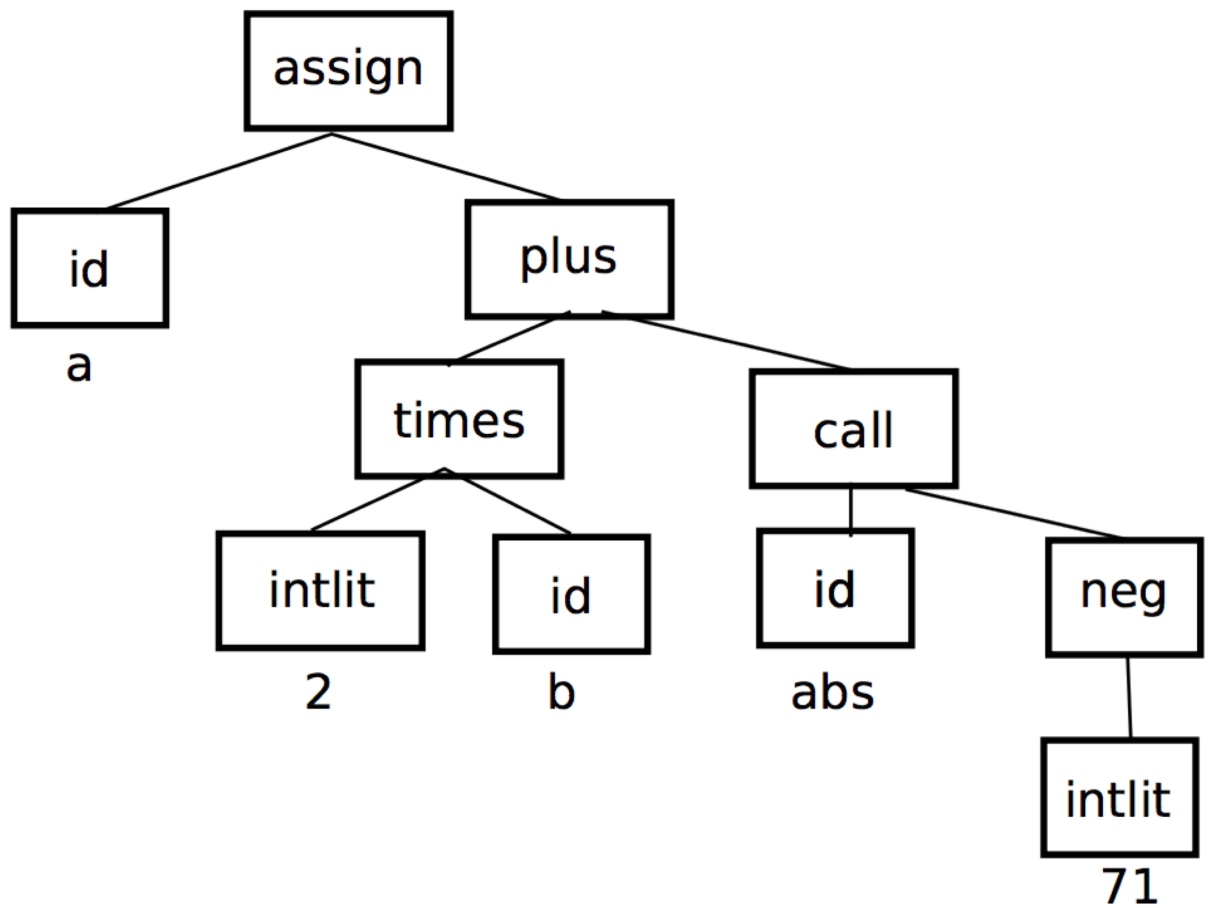
Example

scanner

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

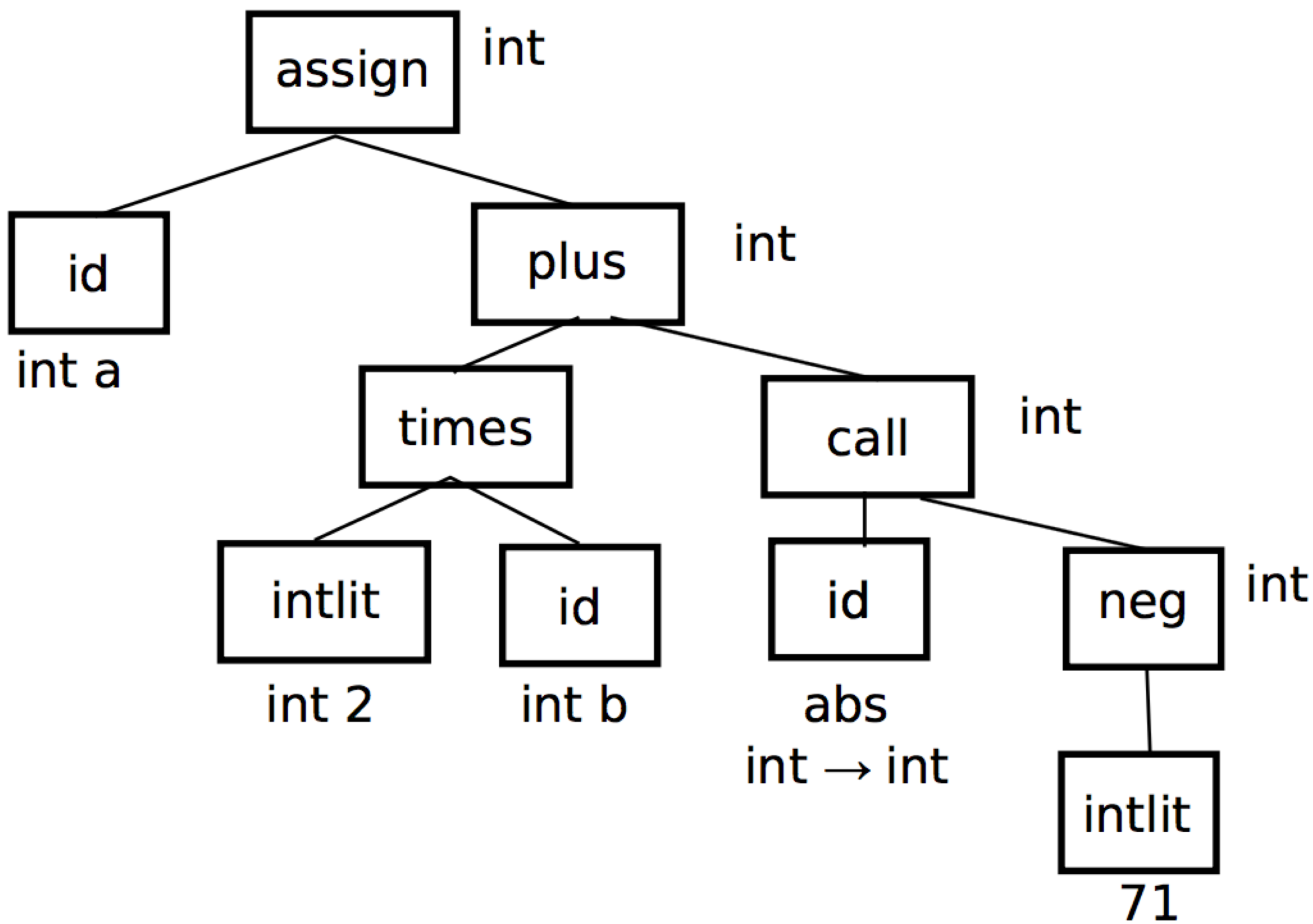
ident (a) | asgn | int lit (2) | times | ident (b) | plus | ident (abs) | lparens | minus | int lit (71) | rparens

parser



Example (cont'd)

semantic analyzer

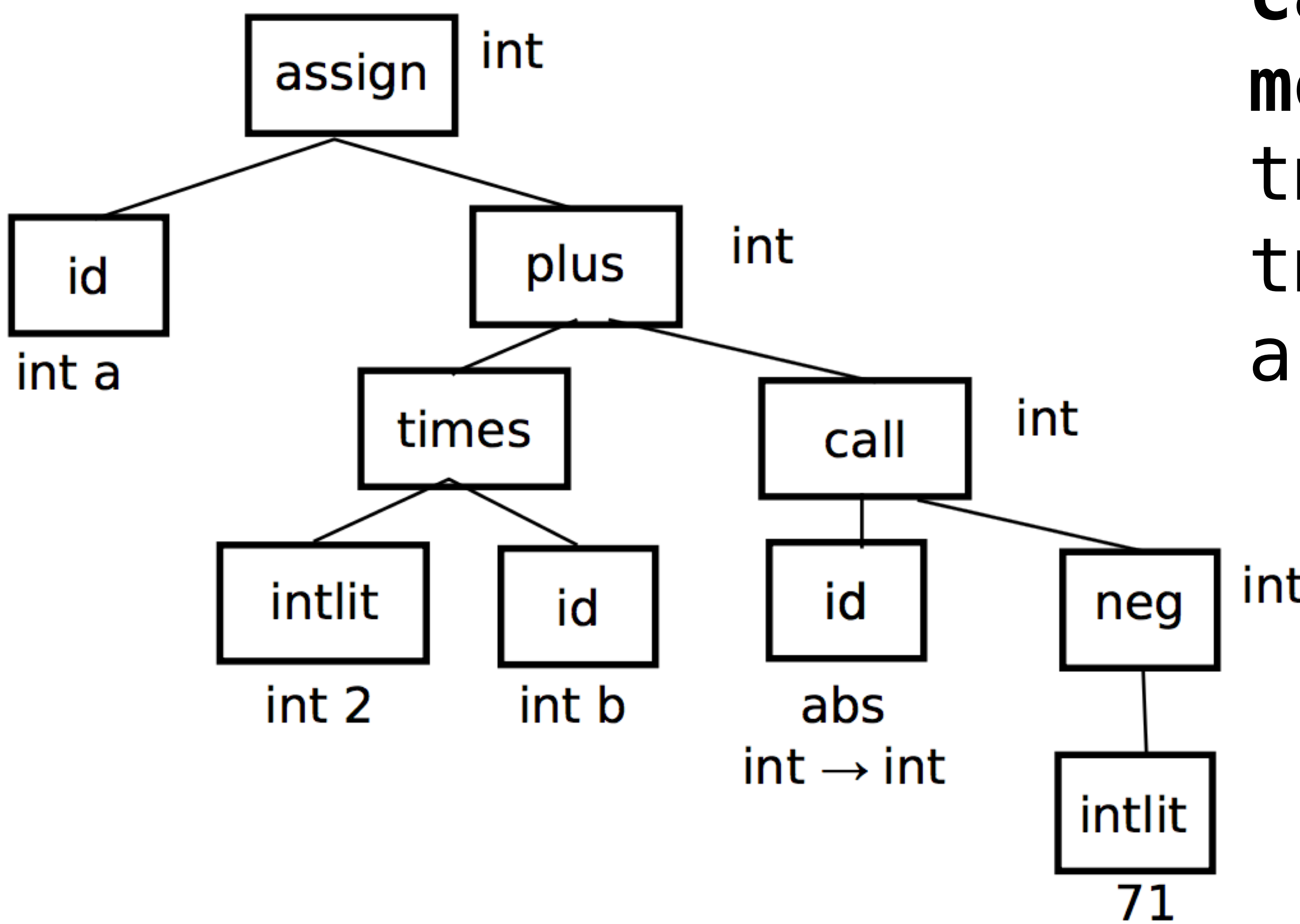


Symbol table

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 (~P6)

Input: IR from optimizer

Output: target code

Symbol table (P1)

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