Welcome to CS 536:
Introduction to Programming Languages and Compilers!

Instructor: Beck Hasti

- hasti@cs.wisc.edu
- Office hours to be determined

TAs

- Andrey Yao
- Robert Nagel
- Sadman Sakib
- Saikumar Yadugiri
- Ting Cai

Course websites:

`canvas.wisc.edu`

`www.piazza.com/wisc/spring2024/compsci536`

`pages.cs.wisc.edu/~hasti/cs536`

About the course

We will study compilers

We will understand how they work

We will build a full compiler

Course mechanics

Exams (60%)

- Midterm 1 (18%): Thursday, February 29, 7:30 – 9 pm
- Midterm 2 (16%): Thursday, March 21, 7:30 – 9 pm
- Final (26%): Sunday, May 5, 2:45 – 4:45 pm

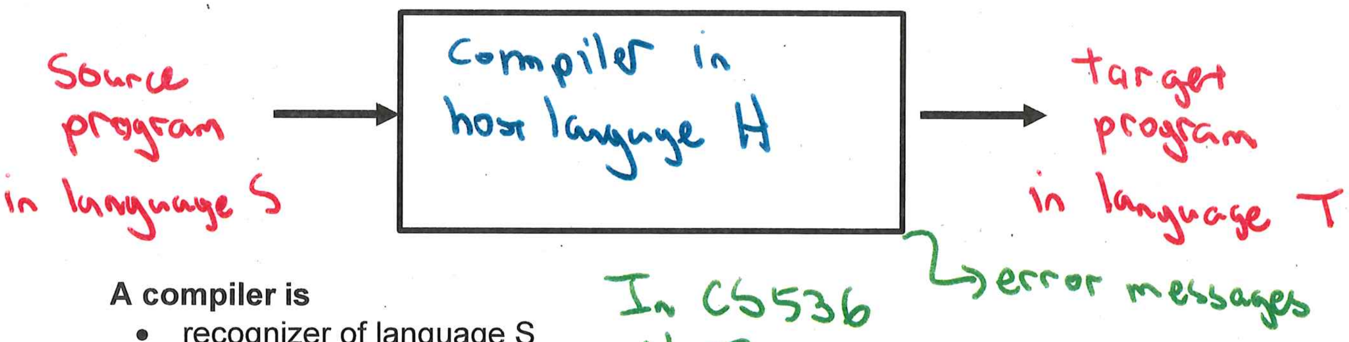
Programming Assignments (40%)

- 6 programs: 5% + 7% + 7% + 7% + 7%+ 7%

Homework Assignments

- 8 short homeworks (optional, not graded)

What is a compiler?

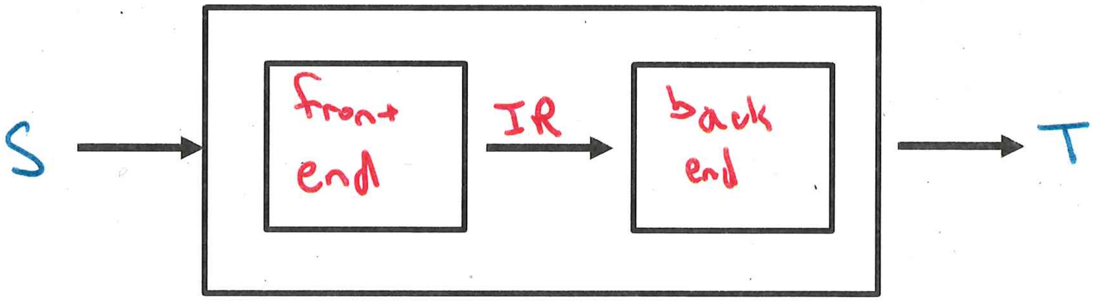

source program in language S
compiler in host language H
target program in language T
error messages

A compiler is

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

In CS536
H: Java
S: base
T: mips

Front end vs back end

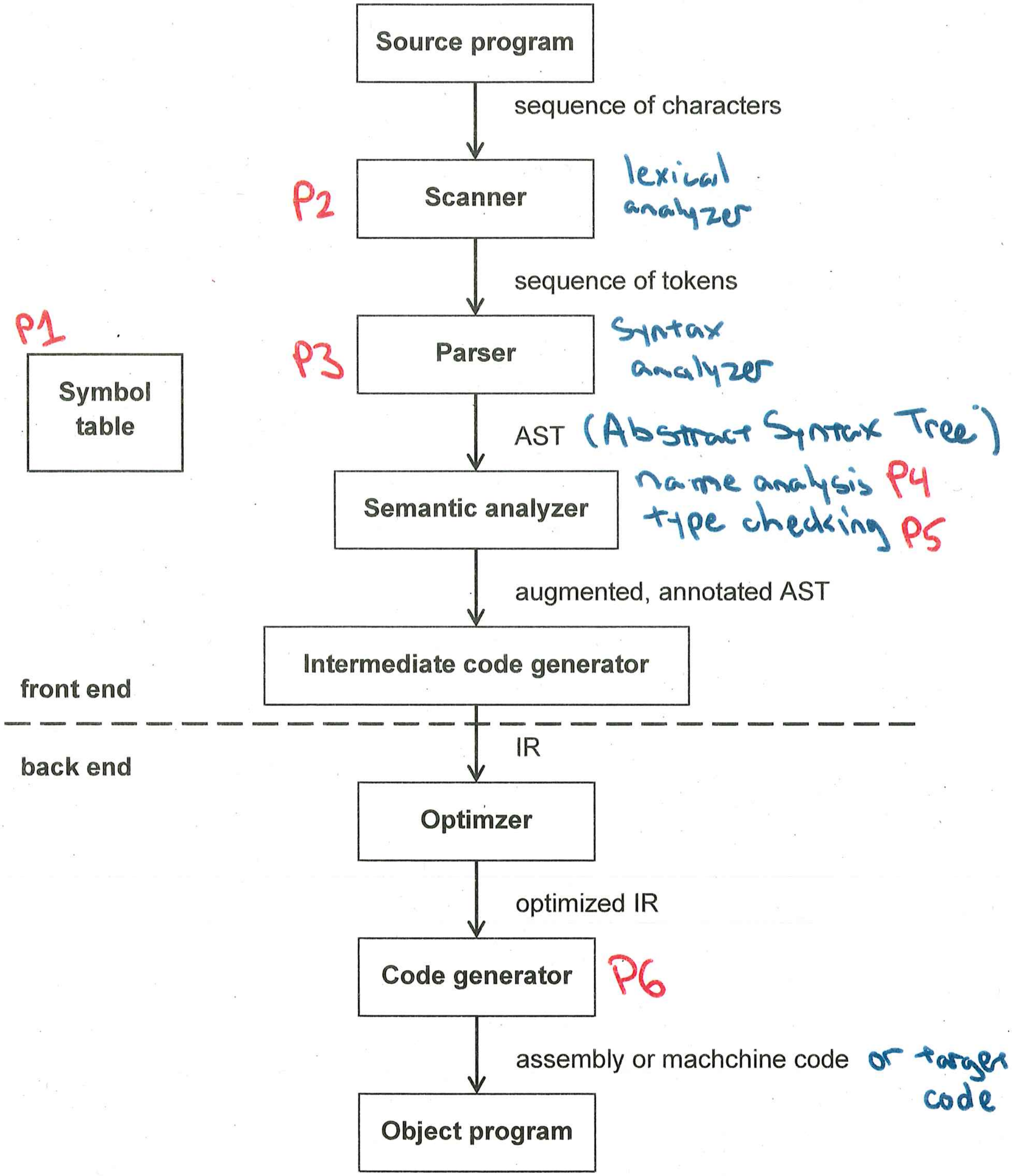


front end = understand source code S; map S to IR

IR = intermediate representation

back end = map IR to T

Overview of typical compiler



Scanner

Input: characters from source program

Output: sequence of tokens

└ possibly with associated info

Actions:

- group characters into lexemes (tokens)
- identify and ignore whitespace, comments, etc.

What errors can it catch?

- bad characters eg # in Java
- unterminated strings "Hello
- integer literals that are too large

Parser

Input: sequence of tokens from the scanner

Output: AST (abstract syntax tree)

Actions:

- group tokens into sentences

What errors can it catch?

- syntax errors X = Y =* 5;
- (possibly) *static semantic* errors use of undeclared variable

Semantic analyzer

Input: AST

Output: annotated AST

Actions: does more static semantic checks

- Name analysis

  process decls & use of variables
  match uses w/decls
  enforces scoping rules
  errors - multiply-declared vars, use of undeclared variables

- Type checking

  check types & augment AST

Intermediate code generator

Input: annotated AST - assumes no syntax / static-semantic errors

Output: intermediate representation (IR)

eg 3-address code

- instructions have at most 3 operands
- easy to generate from AST

  ↳ 1 instr per AST internal node

Example

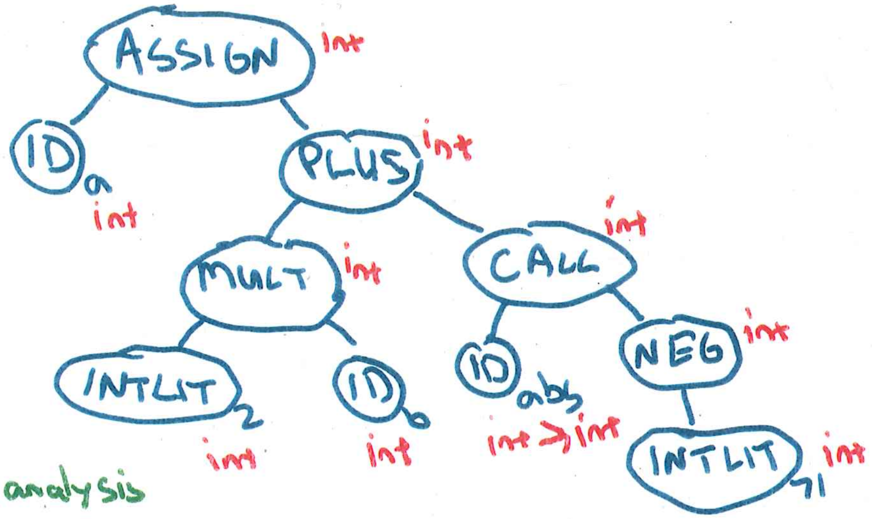
```
a = 2 * b + abs(-71);
```

Scanner produces tokens:

ID(a) ASSIGN INTLIT(2) TIMES ID(b) PLUS ID(abs) LPAREN MINUS INTLIT(71) RPAREN SEMICOLON

Scanner doesn't know if unary or binary

AST (from parser)



Symbol table Name analysis gives us symbol table!

ID	Kind	type
a	var	int
b	var	int
abs	fctn	int → int

3-address code

```
temp1 = 2 * b
temp2 = 0 - 71
move temp2 param1
call abs
move return1 temp3
temp4 = temp1 + temp3
a = temp4
```

a = temp1 + temp3

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

local = look at a few instr at a time.
global = look at entire function or whole prog

Code generator

Input: IR from optimizer

Output: target code

For 536 our IR is an AST

Symbol Table

Compiler keeps track of names in
- semantic analyzer – both name analysis & type checking
- code generation – offsets into stack
- optimizer – could use to keep track of def-use info

P1 : implement symbol table

Block-structured language eg Java, C, C++, base
- nested visibility of names – no access outside of scope of name
- easy to tell which def of a name applies (usually nearest enclosing)
- lifetime of data is bound to scope of identifier that denotes it

Example: (from C)

```
int x, y;

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

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

block structure ⇒
- need nesting of Sym Tabs
⇒ list of hashtables