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

Instructor: Beck Hasti
- hasti@cs.wisc.edu
- Office hours to be determined

TAs
- Aaryan Patel
- Daniel Smedema
- Jack Stanek
- Nick Boddy

Course websites:

`canvas.wisc.edu`

`www.piazza.com/wisc/spring2025/compsci536`

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

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 27, 6:30 – 8 pm
- Midterm 2 (16%): Thursday, March 20, 6:30 – 8 pm
- Final (26%): Thursday, May 8, 6:30 – 8:30 pm

Programming Assignments (40%)
- 6 programs: 5% + 7% + 7% + 7% + 7%+ 7%

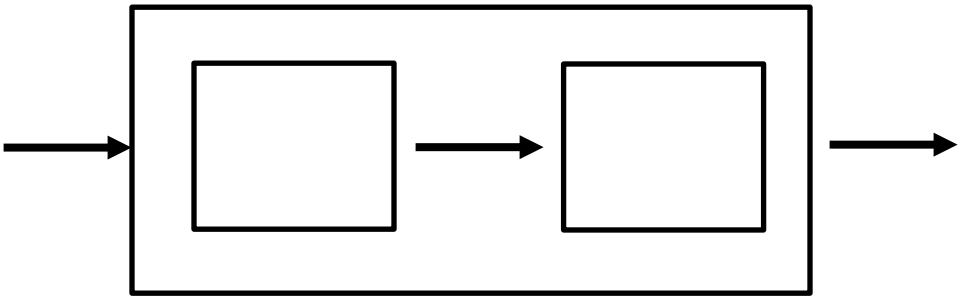
Homework Assignments
- 8 short homeworks (optional, not graded)

What is a compiler?

A compiler is

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

Front end vs back end

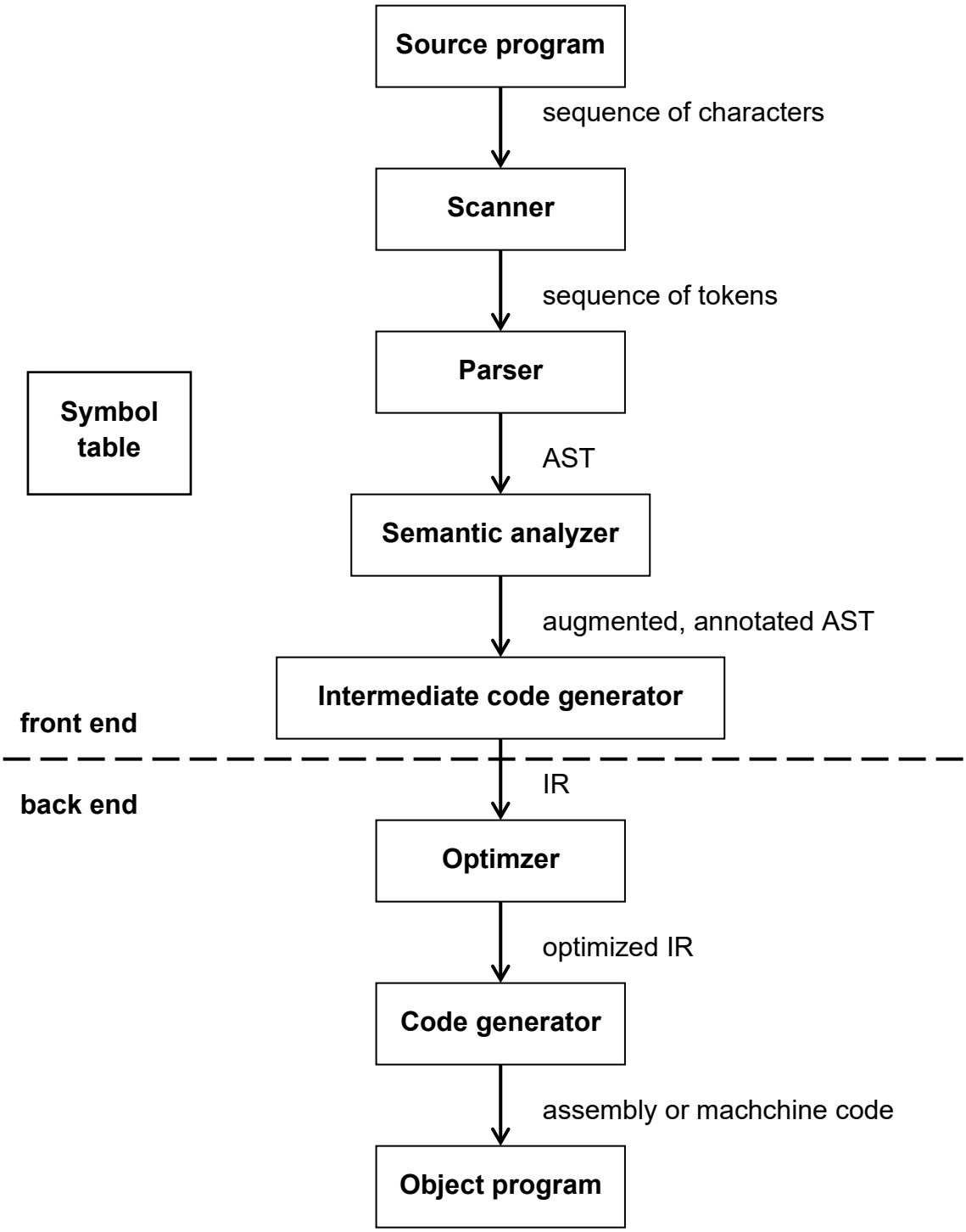


front end = understand source code S; map S to IR

IR = intermediate representation

back end = map IR to T

Overview of typical compiler



Scanner

Input: characters from source program

Output: sequence of tokens

Actions:

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

What errors can it catch?

- bad characters
- unterminated strings
- 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
- (possibly) *static semantic* errors

Semantic analyzer

Input: AST

Output: annotated AST

Actions: does more static semantic checks

- Name analysis
- Type checking

Intermediate code generator

Input: annotated AST

Output: intermediate representation (IR)

Example

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

Scanner produces tokens:

AST (from parser)

Symbol table

3-address code

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

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
- code generation
- optimizer

P1 : implement symbol table

Block-structured language

- nested visibility of names
- easy to tell which def of a name applies
- lifetime of data is bound to scope

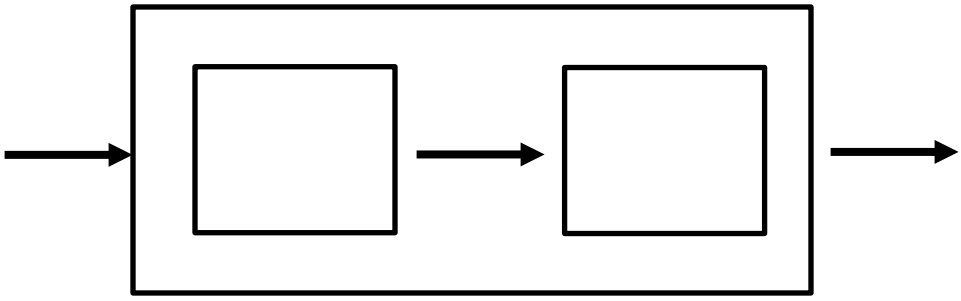
Example: (from C)

```
int x, y;

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

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

Recall

A compiler is

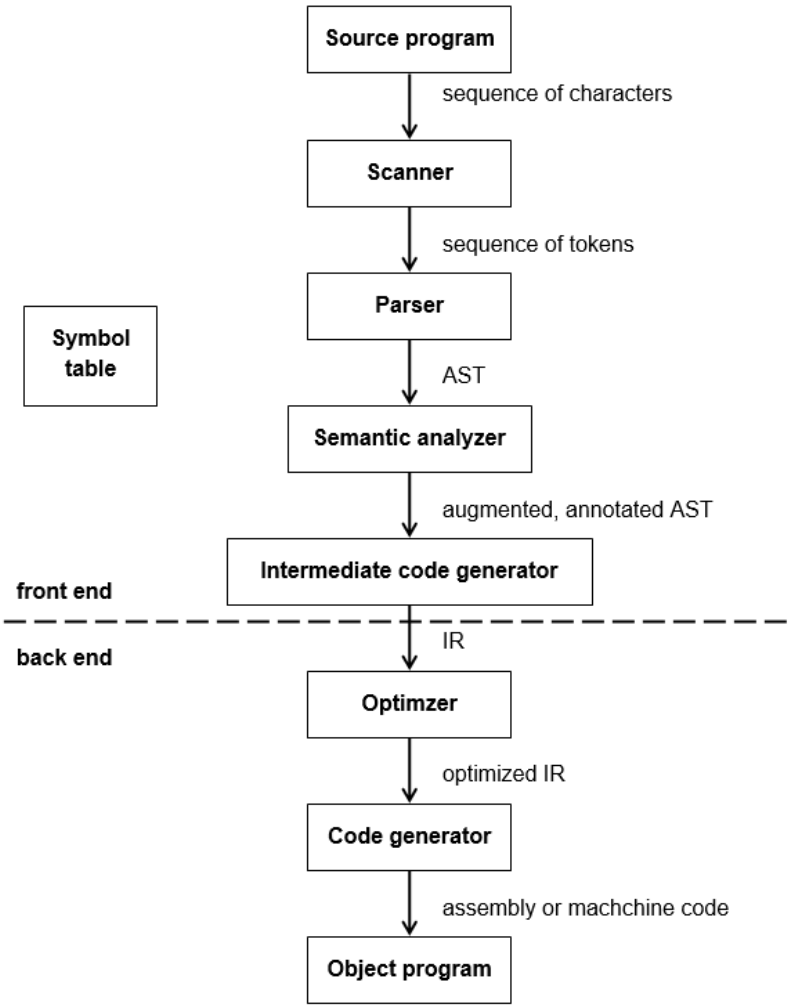
- recognizer of language S
- a translator from S to T
- a program in language H



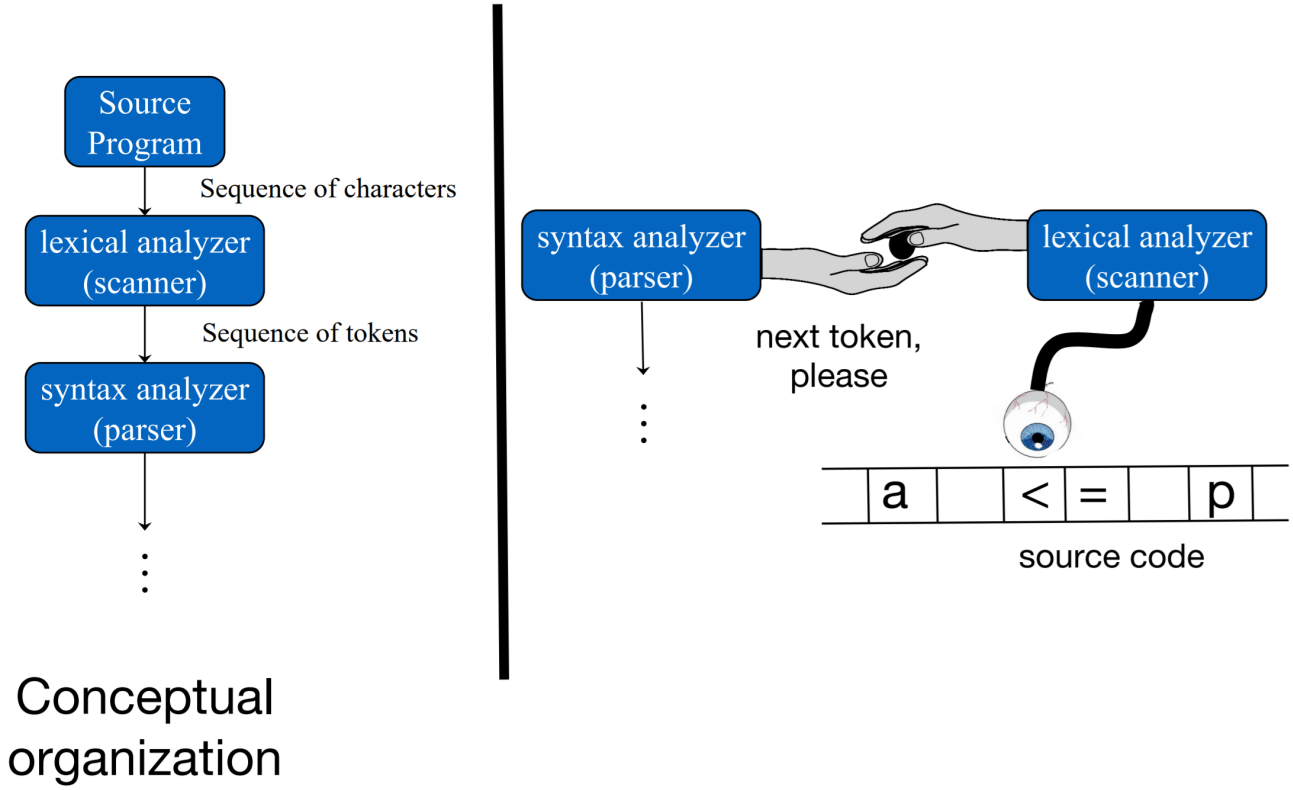
front end = understand source code S; map S to IR
IR = intermediate representation
back end = map IR to T

Why do we need a compiler?

- processors can execute only binaries (machine-code/assembly programs)
- writing assembly programs will make you lose your mind
- allows you to write programs in nice(ish) high-level languages like C; compile to binaries



Special linkage between scanner and parser (in most compilers)



Scanning

Scanner translates sequence of chars into sequence of tokens

Each time scanner is called it should:

- find longest sequence of chars corresponding to a token
- return that token

Scanner generator

- **Inputs**:
  - one regular expression for each token
  - one regular expression for each item to ignore (comments, whitespace, etc.)
- **Output**: scanner program

To understand how a scanner generator works, we need to understand FSMs

Finite-state machines
(aka finite automata, finite-state automata)

- **Inputs**: string (sequence of characters)
- **Output**: accept / reject

Language defined by an FSM = the set of strings accepted by the FSM

Example 1:

Language: single-line comments starting with // (in Java / C++)

Nodes are states

Edges are transitions

Start state has arrow point to it

Final states are double circles

How a finite state machine works

```
curr_state = start_state
let in_ch= current input character

repeat

    if there is edge out of curr_state with
            label in_ch into next_state
        curr_state = next_state
        in_ch = next char of input

    otherwise

        stuck // error condition

until stuck or input string is consumed

if entire string is consumed and
     curr_state is a final state

    accept string

otherwise

    reject string
```

Formalizing finite-state machines

alphabet (Σ) = finite, non-empty set of elements called **symbols**

string over Σ = finite sequence of symbols from Σ

language over Σ = set of strings over Σ

finite state machine *M* = (Q, Σ, δ, q, F) where

- Q = set of states
- Σ = alphabet
- δ = state transition function $Q \times \Sigma \rightarrow Q$
- q = start state
- F = set of accepting (or final) states

L(*M*) = the language of FSM *M* = set of all strings *M* accepts

finite automata *M* **accepts** $x = x_1 x_2 x_3 \ldots x_n$ iff

$$\delta(\delta(\delta(\ldots\ \delta(\delta(\delta(s_0, x_1), x_2), x_3), \ldots\ x_{n-2}), x_{n-1}), x_n)$$

Example 2: hexadecimal integer literals in Java

Hexadecimal integer literals in Java:

- must start 0x or 0X
- followed by at least one hexadecimal digit (hexdigit)
  - hexdigit = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, A, B, C, D, E, F
- optionally can add long specifier (`l` or `L`) at end

Q =

Σ =

δ =

q =

F =

State transition table

	0	1 – 9	a – f	A – F	x	X	l	L
s_0								
s_1								
s_2								
s_3								
s_4								
s_e								

Coding a state transition table

```
curr_state = start_state
done = false
while (!done)
    ch = nextChar()
    next = transition[curr_state][ch]
    if (next == error || ch == EOF)
        done = true
    else
        curr_state = next

return final_states.contains(curr_state) && next != error
```

Example 3: identifiers in C/C++

A C/C++ identifier

- is a sequence of one or more letters, digits, underscores
- cannot start with a digit

Deterministic vs non-deterministic FSMs

deterministic

- no state has >1 outgoing edge with same label
- edges can only be labelled with elements of Σ

non-deterministic

- states may have multiple outgoing edges with same label
- edges may be labelled with special symbol ε (empty string)

ε -transitions can happen without reading input

Example 2 (revisited): hexadecimal integer literals in Java

Example 4: FSM to recognize keywords `for`, `if`, `int`

Recap

- The scanner reads a stream of characters and tokenizes it (i.e., finds tokens)
- Tokens are defined using regular expressions
- Scanners are implemented using (deterministic) FSMs
- FSMs can be non-deterministic

Next time

- regular expressions
- understand the connections between
  - DFAs and NFAs
  - NFAs and regular expressions
- language recognition → tokenizers
- scanner generators
- JLex

Programming Assignment 1

- released tomorrow (Friday, Jan. 24)
- test code (part 1) due Sunday, Feb. 2 by 11:59 pm
- other files (part 2) due Thursday, Feb. 6 by 11:59 pm