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

Instructor: Beck Hasti
- hasti@cs.wisc.edu
- Office hours (in 5375 CS):
  - Tuesday 2:30 – 3:30 pm
  - Wednesday 1:00 – 2:00 pm
  - Friday 9:30 – 10:30 am
  - and by appointment

TAs
- Aoran Wu
- Nicholas Sorenson
- Evan Wireman

Course websites:
- pages.cs.wisc.edu/~hasti/cs536/
- www.piazza.com/wisc/spring2022/compsci536
- https://canvas.wisc.edu

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 (17%): Wednesday, March 2, 7:30 – 9 pm
- Midterm 2 (17%): Wednesday, March 30, 7:30 – 9 pm
- Final (26%): Sunday, May 8, 12:25 – 2:25 pm

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

Homework Assignments
- ~10 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

Source program
  sequence of characters

Scanner
  lexical analyzer
  sequence of tokens (one at a time)

Symbol table

Parser
  syntax analyzer
  AST (abstract syntax tree)
  name analysis P4
  type checking P5

Semantic analyzer
  augmented, annotated AST

Intermediate code generator
  IR

Optimzer
  optimized IR

Code generator
  assembly or machine code or target code

Object program
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
  process decls & uses of variable
  match uses vs/decls
  enforces scoping rules
  can find multiply-declared vars, undeclared use
- 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

\[ a = 2 \times b + \text{abs}(-71); \]

Scanner produces tokens:

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

AST (from parser)

Name analysis gives us symbol table:

<table>
<thead>
<tr>
<th>ID</th>
<th>Kind</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>var</td>
<td>int</td>
</tr>
<tr>
<td>b</td>
<td>var</td>
<td>int</td>
</tr>
<tr>
<td>abs</td>
<td>func</td>
<td>int - int</td>
</tr>
</tbody>
</table>

Type checker passed \( \checkmark \)

3-address code

\[
\begin{align*}
\text{temp1} &= 2 \times b \\
\text{temp2} &= 0 - 71 \\
\text{move} &\quad \text{temp2} \quad \text{param1} \\
\text{call} &\quad \text{abs} \\
\text{move} &\quad \text{return}1 \quad \text{temp3} \\
\text{temp4} &= \text{temp1} \quad \text{temp3} \\
a &= \text{temp1} + \text{temp3} \\
\end{align*}
\]

Optimize to

\[ a = \text{temp4} \]
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

For 53e or 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

e.g., Java, C, C++

Ideas:

- 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

Example: (from C)

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

block structures =>

- need symbol table(s) with
- nesting

- implement as a list of
  - hashtables