CS 536 / Spring 2021

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/class/kkg8vpmtem865y
• Workload:
  • 6 Programs (50% = 5% + 9% + 9% + 9% + 9% + 9%)
  • 10 short homework problems (optional, not graded)
  • 2 online exams (midterm: 25% + final: 25%)

• 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

What will we name S? **WUMBO**
front end = understand source code $S$
IR = intermediate representation
back end = map IR to $T$
Phases of a compiler

- **P1**: Symbol table

- **P2**: Lexical analyzer (scanner)
  - Sequence of characters → Sequence of tokens

- **P3**: Syntax analyzer (parser)
  - Abstract-syntact tree (AST) → Augmented, annotated AST

- **P4, P5**: Intermediate code generator
  - Intermediate code → Optimized intermediate code

- **P6**: Optimizer
  - Optiominized intermediate code → Assembly or machine code

**Front end**

**Back end**
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 ^
- 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., \( x = y \ast= 5 \)
(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:

```java
... 
{
    int i = 4;
    i++; 
    i++; 
}
out of scope \rightarrow i = 5;
```
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

**Front end**

1. **P1** Symbol table

2. **P2**
   - Extracting sequence of characters
   - Lexical analyzer (scanner)

3. **P3**
   - Syntax analyzer (parser)
   - Abstract-syntactic tree (AST)
   - Semantic analyzer
   - Augmented, annotated AST

4. **P4, P5**
   - Intermediate code generator
   - Intermediate code
   - Optimizer
   - Optimized intermediate code

5. **P6**
   - Code generator
   - Assembly or machine code
   - Object program

**Back end**
Example

\[ a = 2 \times b + \text{abs}( -71 ) \]
Example (cont’d)

semantic analyzer

```
Assign
  id
    int a
  plus
    times
      intlit
        int 2
      id
        int b
    call
      id
        abs
          int
            int
          int
        neg
          intlit
            71
```

Symbol table:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Value</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>fun</td>
<td>int→int</td>
</tr>
</tbody>
</table>
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

```c
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