CS 536
Introduction to programming languages and compilers
Loris D’Antoni
About me

PhD at University of Pennsylvania
Joined University of Wisconsin in 2015
Research in
  Program verification
  Program synthesis
  Programming languages

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: Accessible from Canvas
- Workload:
  - 6 Programs (50% = 5% + 9% + 9% + 9% + 9% + 9%)
  - 10 short homework problems (optional, not graded)
  - 2 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

**Source Program**
- Sequence of characters
  - **P2**
    - lexical analyzer (scanner)
    - Sequence of tokens
    - **P3**
      - syntax analyzer (parser)
      - Abstract-syntax tree (AST)
    - **P4, P5**
      - semantic analyzer
    - Augmented, annotated AST
      - intermediate code generator
      - Intermediate code
      - **P6**
        - optimizer
        - Optimized intermediate code
        - code generator
        - Assembly or machine code
      - object program

**Symbol table**

**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., $x = y *= 5$
- (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++;
}

out of scope  
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

1. **P1** Symbol table
2. **P2**
   - Source Program
   - Sequence of characters
   - **lexical analyzer (scanner)**
   - Sequence of tokens
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**

**front end**

**back end**
Example

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

**scanner**

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

**parser**

```
assign
  |--- id
  |   |--- id
  |   |--- intlit
  |--- plus
      |--- times
          |--- intlit
              |--- id
                  |--- call
                      |--- neg
                          |--- intlit
```

14
Example (cont’d)

semantic analyzer

Symbol table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>var</td>
</tr>
<tr>
<td>b</td>
<td>var</td>
</tr>
<tr>
<td>abs</td>
<td>fun</td>
</tr>
<tr>
<td>int-(\rangle)</td>
<td>int-(\rangle)</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>id</th>
<th>var</th>
</tr>
</thead>
<tbody>
<tr>
<td>plus</td>
<td>int</td>
</tr>
<tr>
<td>assign</td>
<td>int</td>
</tr>
<tr>
<td>times</td>
<td>int</td>
</tr>
<tr>
<td>intlit</td>
<td>int</td>
</tr>
<tr>
<td>int2</td>
<td>int</td>
</tr>
<tr>
<td>id</td>
<td>int</td>
</tr>
<tr>
<td>abs</td>
<td>int</td>
</tr>
<tr>
<td>call</td>
<td>int</td>
</tr>
<tr>
<td>neg</td>
<td>int</td>
</tr>
<tr>
<td>intlit</td>
<td>int</td>
</tr>
<tr>
<td>71</td>
<td>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*