CS 536 / Fall 2015

Introduction to programming languages and compilers

Aws Albarghouthi
aws-teach@cs.wisc.edu
About the course

We will study compilers
We will understand how they work
We will build a full compiler
We will have fun
About me

Got PhD from UToronto—equally cold

Joined faculty here in Jan, 2015

Part of the **PL group**

- software verification/analysis
- software synthesis
- theorem proving
cs.wisc.edu/~aws/courses/cs536-f15/
piazza.com/wisc/fall2015/cs536/home
A compiler is a recognizer of language $S$.

a translator from $S$ to $T$

a program in language $H$

What will we name $S$? **YES**
**front end** = understand source code $S$

**IR** = intermediate representation

**back end** = map IR to $T$
Phases of a compiler

- **Front end**
  - Source Program
  - Lexical analyzer (scanner)
  - Syntax analyzer (parser)
  - Semantic analyzer
  - Intermediate code generator
  - Optimizer
  - Code generator

- **Back end**
  - Optimized intermediate code
  - Assembly or machine code

- **Symbol table**

**Phases**

1. **P1**
   - Symbol table

2. **P2**
   - Sequence of characters

3. **P3**
   - Sequence of tokens
   - Abstract-syntactic tree (AST)

4. **P4, P5**
   - Augmented, annotated AST
   - Intermediate code
   - Optimized intermediate code

5. **P6**
   - Assembly or machine code
   - Object program
Scanner

Input: characters from source program

Output: sequence of tokens

Actions:
  - group chars into lexemes (tokens)
  - Identify and ignore whitespace, comments, etc.

Error checking:
  - 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

**Error checking:**
- 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:

```

...  
{
    int i = 4;
    i++;  
    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

- **P1**: Symbol table

- **P2**: Lexical analyzer (scanner)
  - Input: Source program
  - Output: Sequence of characters

- **P3**: Syntax analyzer (parser)
  - Input: Sequence of characters
  - Output: Abstract-syntax tree (AST)

- **P4, P5**: Intermediate code generator
  - Input: Abstract-syntax tree (AST)
  - Output: Intermediate code

- **P6**: Code generator
  - Input: Intermediate code
  - Output: Assembly or machine code

The process is divided into two main parts:
- **Front end**: P1, P2, P3
- **Back end**: P4, P5, P6
Example

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

**Scanner**
- ident (a)
- asgn
- int lit (2)
- times
- ident (b)
- plus
- ident (abs)
- lparen
- minus
- int lit (71)
- rparen

**Parser**
Example (cont’d)

semantic analyzer

symbol table

a    var   int
b    var   int
abs  fun   int→int
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

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