CS 536 / Spring 2017

Introduction to programming languages and compilers

Thomas Reps
reps@cs.wisc.edu
Guest Lectures (1,2,3)

Lecture 1: David Bingham Brown
Lectures 2 and 3: Venkatesh Srinivasan

Both are Ph.D. students working with Prof. Reps
About me

PhD at Cornell University

Joined University of Wisconsin in 1985

Research in

- Dataflow analysis (source code and machine code)
- Program verification
- Applications to computer security

http://pages.cs.wisc.edu/~reps/
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

- Piazza: [https://piazza.com/wisc/spring2017/compsci536_001_sp17/](https://piazza.com/wisc/spring2017/compsci536_001_sp17/)
- Workload:
  - 6 Programs (40% = 5% + 7% + 7% + 7% + 7% + 7%)
  - 10 short homeworks (20%)
  - 2 exams (midterm: 20% + final: 20%)
- For information about late policy, collaboration, etc., see [http://pages.cs.wisc.edu/~cs536-1/info.html](http://pages.cs.wisc.edu/~cs536-1/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 $S$? ...
**front end** = understand source code $S$

**IR** = intermediate representation

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

1. **Front end**
   - **P1**: Symbol table
   - **P2**: Source program → Sequence of characters
   - **P3**: Lexical analyzer (scanner) → Sequence of tokens
   - **P4, P5**: Syntax analyzer (parser) → Abstract-syntactic tree (AST)
   - **P6**: Intermediate code generator → Intermediate code
     - **P7**: Optimizer → Optimized intermediate code
     - **P8**: Code generator → Assembly or machine code
     - **P9**: Object program

2. **Back end**
Scanner (P2)

**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
Example

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

Whitespace (spaces, tabs, and newlines) filtered out

The scanner’s output is still the sequence
Parser (P3)

**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 (P4,P5)

**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 (P4,P5)

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. Front end
   - Lexical analyzer (scanner)
     - P2
     - Sequence of characters
   - Syntax analyzer (parser)
     - P3
     - Abstract-syntactic tree (AST)
   - Semantic analyzer
     - P4, P5
     - Augmented, annotated AST
   - Intermediate code generator
     - Intermediate code

2. Back end
   - Optimizer
     - Optimized intermediate code
   - Code generator
     - P6
     - Assembly or machine code
   - Object program
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
   a
   plus
   times
   intlit 2
   id b
call
   id
   abs
   neg
   intlit 71
```
Example (cont’d)

semantic analyzer

```
Semantic 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\times 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 (~P6)

**Input:** IR from optimizer

**Output:** target code
Symbol table (P1)

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

block structure: need symbol table with nesting

implement as list of hashtables

```
int x, y;

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

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