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

• Home page: http://pages.cs.wisc.edu/~cs536-1/
• Piazza: 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
A compiler is a
recognizer of language S
a translator from S to T
a program in language H
**front end** = recognize source code S; map S to IR

**IR** = intermediate representation

**back end** = map IR to T

Executing the T program produces the same result as executing the S program?
Executing the T program produces the same result as executing the S program?

Hmm … how do we execute the S program?
  Use an **S interpreter**
How do we execute the T program?
  Use a **T interpreter** (which might be hardware)
Interpretation

• Evaluation method for programs in some PL L

• Given the representation $R_P$ for program $P \in L$ (e.g., as a tree or instruction sequence)

  \[
  \text{while} \ (\! \text{done}) \ \{
  \begin{align*}
  &\text{Examine some part of } R_P \\
  &\text{Perform computation step}
  \end{align*}
  \}
  \]

  \text{Eval}

  return answer
Each interpreter implements an abstract machine whose programs are sequences (or sometimes trees) of “instructions”.

http://flickr.com/photos/pelf81/324890844/
It’s Interpreters All the Way Down

“Interpretation approach”

\[ P, \text{ represented as } R_P \]
Interpreted by Eval, represented as \( x_{86}^{\text{Eval}} \)
Interpreted by \( x_{86} \) hardware

“Compilation approach”

\[ P, \text{ represented as } x_{86_P} \]
Interpreted by \( x_{86} \) hardware
It’s Interpreters All the Way Down

“Interpretation approach”

\[ P, \text{ represented as a tree} \]
Interpreted by EvalTree pseudocode

“Compilation approach”

\[ P, \text{ represented as } instSeq_P \]
Interpreted by EvalInstSeq pseudocode
Example

(5 * (((3+5) * 9) + (7 * 2)))
int EvalTree(Tree T) {
    switch(T.op) {
        case '*':
            l = EvalTree(T.leftChild);
            r = EvalTree(T.rightChild);
            return l * r;
        case '+':
            l = EvalTree(T.leftChild);
            r = EvalTree(T.rightChild);
            return l + r;
        default: // Leaf: T holds an int
            return T.value;
    }
}

EvalTree(5 * (((3+5) * 9) + (7 * 2)))
  | EvalTree(5)
  | EvalTree(((3+5) * 9) + (7 * 2)))
  |  | EvalTree((3+5) * 9)
  |  |  | EvalTree(3+5)
  |  |  |  | EvalTree(3)
  |  |  |  | EvalTree(5)
  |  |  |  | 3 + 5 = 8
  |  |  |  | EvalTree(9)
  |  |  |  | 8 * 9 = 72
  |  |  | EvalTree(7 * 2)
  |  |  |  | EvalTree(7)
  |  |  |  | EvalTree(2)
  |  |  |  | 7 * 2 = 14
  |  | 72 + 14 = 86
5 * 86 = 430
instSeq GenInstSeq(Tree T) {
    switch(T.op) {
    case '*':
        lSeq = GenInstSeq(T.leftChild);
        rSeq = GenInstSeq(T.rightChild);
        return lSeq || rSeq || Multiply;
    case '+':
        lSeq = GenInstSeq(T.leftChild);
        rSeq = GenInstSeq(T.rightChild);
        return lSeq || rSeq || Add;
    default:  // Leaf: T holds an int
        return InstSeq("Push", T.value);
    }
}

GenInstSeq(5 * (((3+5) * 9) + (7 * 2)))
  | GenInstSeq(5) → [Push(5)]
  | GenInstSeq(((3+5) * 9) + (7 * 2)))
  |     | GenInstSeq((3+5) * 9)
  |     |     | GenInstSeq(3+5)
  |     |     |     | GenInstSeq(3)
  |     |     |     | Push(3)
  |     |     | GenInstSeq(5)
  |     |     | Push(5)
  |     |     | Push(3); Push(5); Add
  |     | GenInstSeq(9) → [Push(9)]
  |     |     | Push(3); Push(5); Add; Push(9); Multiply
  |     | GenInstSeq(7 * 2)
  |     |     | GenInstSeq(7) → [Push(7)]
  |     |     | GenInstSeq(2) → [Push(2)]
  |     |     | [Push(7); Push(2); Multiply]
  |     |     | [Push(3); Push(5); Add; Push(9); Multiply; Push(7); Push(2); Multiply; Add]
  |     | [Push(5); Push(3); Push(5); Add; Push(9); Multiply; Push(7); Push(2); Multiply; Add; Multiply]

In essence, GenInstSeq prints the tree in postfix
Stack S = EmptyStack;

int EvalInstSeq(InstSeq IS) {
    while(!null(IS)) {
        inst = IS.head;
        switch(inst.opCode) {
        case Multiply:
            r = S.pop();
            l = S.pop();
            S.push(l * r);
            break;
        case '+':
            r = S.pop();
            l = S.pop();
            S.push(l + r);
            break;
        default: // Push(…)
            S.push(inst.argument);
            break;
        }
        IS = IS.tail;
    }
    return S.pop();
}

EvalInstSeq([P(5); P(3); P(5); A; P(9); M; P(7); P(2); M; A; M])
S = [ ]
IS = [P(5); P(3); P(5); A; P(9); M; P(7); P(2); M; A; M]
S = [5]
IS = [P(3); P(5); A; P(9); M; P(7); P(2); M; A; M]
S = [5, 3]
IS = [P(5); A; P(9); M; P(7); P(2); M; A; M]
S = [5, 3, 5]
IS = [A; P(9); M; P(7); P(2); M; A; M]
S = [5, 8]
IS = [P(9); M; P(7); P(2); M; A; M]
S = [5, 8, 9]
IS = [M; P(7); P(2); M; A; M]
S = [5, 72]
IS = [P(7); P(2); M; A; M]
S = [5, 72, 7]
IS = [P(2); M; A; M]
S = [5, 72, 7, 2]
IS = [M; A; M]
S = [5, 72, 14]
IS = [A; M]
S = [5, 86]
IS = [M]
S = [430]
IS = [ ]
A compiler is a recognizer of language $S$
a translator from $S$ to $T$
a program in language $H$
Phases of a compiler

1. **Front end**
   - **Symbol table**
   - **Lexical analyzer (scanner)**
     - Sequence of characters
     - Sequence of tokens
   - **Syntax analyzer (parser)**
     - Abstract-syntact tree (AST)
   - **Semantic analyzer**
     - Augmented, annotated AST
   - **Intermediate code generator**
     - Intermediate code

2. **Back end**
   - **Optimizer**
     - Optimized intermediate code
   - **Code generator**
     - Assembly or machine code
   - **Object program**
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

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

The scanner’s output is still the sequence

\[ \text{ident} (a) \text{ asgn} \text{ int lit} (2) \text{ times} \text{ ident} (b) \text{ plus} \text{ ident} \text{ (abs) lparens} \text{ int lit} (71) \text{ rparens} \text{ minus} \text{ int lit} (71) \text{ rparens} \]
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

- **Front end**
  - Source Program → Sequence of characters
  - Lexical analyzer (scanner) → Sequence of tokens
  - Syntax analyzer (parser) → Abstract-syntax tree (AST)
  - Semantic analyzer → Augmented, annotated AST
  - Intermediate code generator → Intermediate code
  - Optimizer → Optimized intermediate code
  - Code generator → Assembly or machine code
  - Object program

- **Back end**

**Symbol table**
Example

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

**scanner**

\[
\begin{align*}
\text{ident} &\quad \text{assign} &\quad \text{int lit} &\quad \text{times} &\quad \text{ident} &\quad \text{plus} &\quad \text{ident} &\quad \text{lparens} &\quad \text{int lit} &\quad \text{rparens} \\
(a) &\quad (2) &\quad (b) & & & & &\quad (-71) \\
\end{align*}
\]

**parser**

```
assign
  id
    a
  plus
    times
      intlit
        2
      id
        b
    call
      id
        id
    neg
      intlit
        71
```
Example (cont’d)

semantic analyzer

```
semantic analyzer

Symbol table

a var int
b var int
abs fun int->int
```
Example (cont’d)

**code generation**

```plaintext
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 (~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++
Idea:

* 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