Compilers History and Review

Friday Review 1

EECS 483

January 4, 2018
Course Information

- Office Hours: Thursday, 2-4pm in BBB 2717
- GSI: Ram Kannan
- IA: Lawrence Wu

- Please use Piazza as primary contact
Waitlisting

- Please be patient with the waitlist
- Protip! It is perfectly cromulent to audit the course. Anyone can use the autograder.
  - You will not receive course credit on your transcript if not enrolled
  - You will receive immense satisfaction in becoming an ultimate master of CS
Why Study History?

- Those who cannot remember George Santayana are condemned to misquote him.

*Supernatural, 1999*
Why Study History?

- Those who cannot remember the past are condemned to repeat it.

*George Santayana*
Why Study History?

▶ Those who cannot remember the past are condemned to repeat it.

George Santayana

▶ Through meticulous analysis of history, I will find a way to make the people worship me. By studying the conquerors of days gone by, I’ll discover the mistakes that made them go awry.

The Brain
PL and Compilers

- 1920: Computer = Human
PL and Compilers

- 1920: Computer = Human
- 1936: Church’s Lambda Calculus
PL and Compilers

- 1920: Computer = Human
- 1936: Church’s Lambda Calculus
- 1937: Digital circuits (thanks Shannon!)
PL and Compilers

- 1920: Computer = Human
- 1936: Church’s Lambda Calculus
- 1937: Digital circuits (thanks Shannon!)
- 1940’s: Digital computers
PL and Compilers

- 1920: Computer = Human
- 1936: Church’s Lambda Calculus
- 1937: Digital circuits (thanks Shannon!)
- 1940’s: Digital computers

- Everything programmed manually!
Surprise!

How many lines of code do you write per day?
Surprise!

How many lines of code do you write per day?

1. 10 LOC/day: Fred Brooks

Surprise!

How many lines of code do you write per day?

1. 10 LOC/day: Fred Brooks
2. 16–36 LOC/day: Caper Jones

2. https://dzone.com/articles/programmer-productivity
Surprise!

How many lines of code do you write per day?

1. 10 LOC/day: Fred Brooks
2. 16–36 LOC/day: Caper Jones
3. 1.5–25 LOC/day: McConnell

2. https://dzone.com/articles/programmer-productivity
Compilers to the rescue

- Allow software developers to specify program behavior with high level languages
Compilers to the rescue

- Allow software developers to specify program behavior with high level languages
- 10 LOC of C++ slightly better than 10 LOC of x86 assembly
1972: C Systems Programming

- 1972: C Systems Programming
- 1983: Ada DOD, type safety
- 1983: C++ Object-oriented!
- 1987: Perl Dynamic scripting!
- 1990: Python everyone’s favorite!
- 1991: Java Portable language for iTV
- 1994: PHP Perfect hypertext preprocessor
- 1996: OCaml Functional + imperative
- 2000: C# Microsoft Java Compiler Construction
1972: C
1983: Ada

Systems Programming
DOD, type safety
1972: C
1983: Ada
1983: C++

Systems Programming
DOD, type safety
Object-oriented!
1972: C
1983: Ada
1983: C++
1987: Perl

Systems Programming
DOD, type safety
Object-oriented!
Dynamic scripting!

I hired a creep to help determine our product's features.

You need more features. Good work.

When can you have that done?

GAAA!!
1972: C Systems Programming
1983: Ada DOD, type safety
1983: C++ Object-oriented!
1987: Perl Dynamic scripting!
1990: Python everyone’s favorite!
1991: Java Portable language for iTV
1994: PHP Perfect hypertext preprocessor
1996: OCaml Functional + imperative
2000: C# Microsoft Java
1972: C Systems Programming
1983: Ada DOD, type safety
1983: C++ Object-oriented!
1987: Perl Dynamic scripting!
1990: Python everyone’s favorite!
1991: Java Portable language for iTV
1972: C Systems Programming
1983: Ada DOD, type safety
1983: C++ Object-oriented!
1987: Perl Dynamic scripting!
1990: Python everyone’s favorite!
1991: Java Portable language for iTV
1994: PHP Perfect hypertext preprocessor
1972: C
1983: Ada
1983: C++
1987: Perl
1990: Python
1991: Java
1994: PHP
1996: OCaml

Systems Programming
DOD, type safety
Object-oriented!
Dynamic scripting!
everyone’s favorite!
Portable language for iTV

Perfect hypertext preprocessor
Functional + imperative
1972: C
1983: Ada
1983: C++
1987: Perl
1990: Python
1991: Java
1994: PHP
1996: OCaml
2000: C#

- Systems Programming
- DOD, type safety
- Object-oriented!
- Dynamic scripting!
- everyone’s favorite!
- Portable language for iTV
- *Perfect* hypertext preprocessor
- Functional + imperative
- Microsoft Java
What was the original name of Java before the creators discovered their intended name was already trademarked?
Compilers

- These languages need tools to automatically *lower* language to format that makes the CPU happy
Compilers

- These languages need tools to automatically lower language to format that makes the CPU happy
- Basically, convert program to assembly using principled, semantics-preserving transformations
Compilers

- These languages need tools to automatically lower language to format that makes the CPU happy
- Basically, convert program to assembly using principled, semantics-preserving transformations

- Use variable names?
  - map variable names to memory addresses later

- Use long math expressions? (x = 3 + 5 + 2)
- Convert math to equivalent sequence of assembly instructions
- Use functions?
- Convert each of them to code and link them up with jump instructions
- Use classes?
- Plan a known layout of each object's fields and methods
- Use shared libraries?
- Prepare to pull your hair out
Compilers

- These languages need tools to automatically lower language to format that makes the CPU happy
- Basically, convert program to assembly using principled, semantics-preserving transformations

- Use variable names?
  - map variable names to memory addresses later
- Use long math expressions? \((x = 3+5+2)\)
  - Convert math to equivalent sequence of assembly instructions
Compilers

- These languages need tools to automatically *lower* language to format that makes the CPU happy
- Basically, convert program to assembly using *principled, semantics-preserving transformations*

- Use variable names?
  - map variable names to memory addresses later
- Use long math expressions? \((x = 3 + 5 + 2)\)
  - Convert math to equivalent sequence of assembly instructions
- Use functions?
  - Convert each of them to code and link them up with jump instructions
Compilers

- These languages need tools to automatically lower language to format that makes the CPU happy
- Basically, convert program to assembly using principled, semantics-preserving transformations

- Use variable names?
  - map variable names to memory addresses later
- Use long math expressions? \((x = 3+5+2)\)
  - Convert math to equivalent sequence of assembly instructions
- Use functions?
  - Convert each of them to code and link them up with jump instructions
- Use classes?
  - Plan a known layout of each object’s fields and methods
Compilers

- These languages need tools to automatically lower language to format that makes the CPU happy
- Basically, convert program to assembly using principled, semantics-preserving transformations

- Use variable names?
  - map variable names to memory addresses later
- Use long math expressions? \((x = 3 + 5 + 2)\)
  - Convert math to equivalent sequence of assembly instructions
- Use functions?
  - Convert each of them to code and link them up with jump instructions
- Use classes?
  - Plan a known layout of each object’s fields and methods
- Use shared libraries?
  - Prepare to pull your hair out
Compilation stages

Front End: Reject all invalid programs!
1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis

Back End: Make code out of a valid program!
4. Code Generation
5. Optimization
Lexing

- Break input program into meaningful pieces:
  - Keywords: if, then, else, while, switch, class
  - Variable names: arbitrary strings of characters
  - Constant values: 0,1,2,..., "abc"
  - special characters: {, }, ;, #, //

- if (x < 5) { y = 2; }
  becomes
  if, LPAREN, x, LT, 5, LBRACE, y, EQ, 2, SEMI, RBRACE
Parsing

- Ensure program follows **grammar rules** as defined by the language

- if (x < 5) { y = 2; } else { y = 3; }
  becomes Abstract Syntax Tree (AST)

  ```plaintext
  ifelse
  ```
Parsing

- Ensure program follows **grammar rules** as defined by the language

- if \((x < 5)\) \{ \(y = 2;\) \} else \{ \(y = 3;\) \} becomes Abstract Syntax Tree (AST)

```
ifelse
| condition   | true_body | false_body |
```
Parsing

- Ensure program follows **grammar rules** as defined by the language

- if \((x < 5)\) \{ \(y = 2;\) \} else \{ \(y = 3;\) \}
becomes Abstract Syntax Tree (AST)

- `ifelse`
  - `condition`
  - `true_body`
  - `false_body`
  - `lt_compare`
  - `assignment`
  - `assignment`
Parsing

- Ensure program follows **grammar rules** as defined by the language

- `if (x < 5) { y = 2; } else { y = 3; }` becomes Abstract Syntax Tree (AST)

```
        elseif
         /
        /    \     \    
condition  true_body  false_body
            /
   lt_compare  assignment  assignment
   x    5    y    2    y    3
```
Parsing

- Ensure program follows **grammar rules** as defined by the language

- `if (x < 5) { y = 2; } else { y = 3; }` becomes Abstract Syntax Tree (AST)
Semantic Analysis

- Annotate AST with type information!

```
ifelse
  condition
    lt_compare (Boolean)
      x (Integer) 5 (Integer)
  true_body
    assignment (Integer)
      y (Integer) 2 (Integer)
  false_body
    assignment (Integer)
      y (Integer) 3 (Integer)
```

Compiler Construction
Code Generation

- Traverse AST and generate code for each node!

```assembly
load t0, x ; need to load x from memory
lt t1, 5 ; check if t1 < 5
bfalse L2 ; if not, skip to else body

; true body
li t1, 2
store t1, y
jmp L2 ; skip over else body

; else body L1:
li t1, 3
store t1, y

L2:
...```

Optimization

▶ Repeat after me: I will not prematurely optimize
▶ Your generated code will be insatiably dumb
▶ if (x < 5) y = 2; else y = 3;

```c
// return true if x is greater than or equal to y
bool value_to_return;
if(x > y) {
    value_to_return = true;
}
if(x < y) {
    value_to_return = false;
}
if(x == y) {
    value_to_return = true;
}
return value_to_return;
```
Optimization

▸ Repeat after me: I will not prematurely optimize
▸ Your generated code will be insatiably dumb

▸ if (x < 5) y = 2; else y = 3;

▸ What if you know x = 2?

```c
// return true if x is greater
// than or equal to y
bool value_to_return;
if(x > y) {
    value_to_return = true;
}
if(x < y) {
    value_to_return = false;
}
if(x == y) {
    value_to_return = true;
}
return value_to_return;
```
Optimization

- Repeat after me: I will not prematurely optimize
- Your generated code will be insatiably dumb

- if (x < 5) y = 2; else y = 3;

- What if you know x = 2?
- Skip to the end, just remove it all and keep y = 2;

```c
// return true if x is greater or equal to y
bool value_to_return;
if(x > y) {
    value_to_return = true;
}
if(x < y) {
    value_to_return = false;
}
if(x == y) {
    value_to_return = true;
}
return value_to_return;
```