Lecture 1

CMSC 350: COMPILER DESIGN
Why CMSC 350?

• You will learn:
  – Practical applications of theory
  – Lexing/Parsing/Interpreters
  – How high-level languages are implemented in machine language
  – Assembly language: HERA
  – A deeper understanding of code
  – A little about programming language semantics & types
  – Functional programming in Haskell
  – How to manipulate complex data structures
  – How to be a better programmer

• Expect this to be a very challenging, implementation-oriented course.
  – Programming projects can take 10+ hours per week…
Goal: build a complete compiler from a high-level, type-safe language to HERA.
Tour of the website at http://cs.brynmawr.edu/cs350
Resources

• Course textbook: (required)
  – *Modern Compiler Implementation in ML* (Appel)

• Additional compilers book:
    • a.k.a. “The Dragon Book”

• About Haskell:
  – *Real World Haskell* (O’Sullivan, Stewart, Goerzen)
    • book.realworldhaskell.org
Why Haskell?

• Haskell is inspired by ML, which originally stood for “Meta Language”
  – It was designed to enable easy manipulation of abstract syntax trees
  – Type-safe, pure, functional language with support for polymorphic algebraic datatypes
  – It is the right tool for this job
HW01: HelloHaskell

- Assignment 1 is available on the course web site.
  - Individual project – no groups
  - Due: Wednesday, Jan 30, 2019 at 11:59pm
  - Topic: Haskell programming, an introduction

- Haskell is installed on powerpuff:
  - Run ghci from the command line to invoke the top-level loop
  - Run ghc to run the compiler

- Use a proper text editor:
  - Emacs
  - Vim
  - Sublime
  - Atom
What is a compiler?
What is a Compiler?

- A compiler is a program that translates from one programming language to another.
- Typically: *high-level source code to low-level machine code*
  - Not always: Source-to-source translators, Java bytecode compiler, Google Web Toolkit goes from Java to JavaScript
Historical Aside

• This is an old problem!
• Until the 1950s: computers were programmed in assembly.
• 1951-1952: Grace Hopper developed the A-0 system for the UNIVAC I
  – She later contributed significantly to the design of COBOL
• 1957: the FORTRAN compiler was built at IBM
  – Team led by John Backus
• 1960’s: development of the first bootstrapping compiler for LISP
• 1970’s: language/compiler design blossomed
• Today: *thousands* of languages (most little used)
  – Some better designed than others...

1980s: ML / LCF
1984: Standard ML
1990: Haskell
1992: GHC
Source Code

• Optimized for human readability
  – Expressive: matches human ideas of grammar / syntax / meaning
  – Redundant: more information than needed to help catch errors
  – Abstract: exact computation possibly not fully determined by code

• Example C source:

```c
#include <stdio.h>

int factorial(int n) {
    int acc = 1;
    while (n > 0) {
        acc = acc * n;
        n = n - 1;
    }
    return acc;
}

int main(int argc, char *argv[]) {
    printf("factorial(6) = %d\n", factorial(6));
}
```
Low-level code

- Optimized for Hardware
  - Machine code hard for people to read
  - Redundancy, ambiguity reduced
  - Abstractions & information about intent is lost

- Assembly language
  - then machine language

- Figure at right shows (unoptimized) 32-bit code for the factorial function

```assembly
_factorial:
  ## BB#0:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl 8(%ebp), %eax
  movl %eax, -4(%ebp)
  movl $1, -8(%ebp)
  LBB0_1:
  cmpl $0, -4(%ebp)
  jle LBB0_3
  ## BB#2:
  movl -8(%ebp), %eax
  imull -4(%ebp), %eax
  movl %eax, -8(%ebp)
  movl -4(%ebp), %eax
  subl $1, %eax
  movl %eax, -4(%ebp)
  jmp LBB0_1
  LBB0_3:
  movl -8(%ebp), %eax
  addl $8, %esp
  popl %ebp
  retl
```
How to translate?

- Source code / Machine code mismatch
- Some languages are farther from machine code than others:
  - Consider: C, C++, Java, Lisp, ML, Haskell, Ruby, Python, JavaScript

Goals of translation:
- Source level expressiveness for the task
- Best performance for the concrete computation
- Reasonable translation efficiency ($< O(n^3)$)
- Maintainable code
- Correctness!
Correct Compilation

• Programming languages describe computation precisely…
  – therefore, *translation* can be precisely described
  – a compiler can be correct with respect to the source and target language semantics.

• Correctness is important!
  – Broken compilers generate broken code.
  – Hard to debug source programs if the compiler is incorrect.
  – Failure has dire consequences for development cost, security, etc.

• There are techniques for building correct compilers; see the following:
  – CompCert
  – Verified Software Toolchain
  – Vellvm
  – DeepSpec
Idea: Translate in Steps

• Compile via a series of program representations

• Intermediate representations are optimized for program manipulation of various kinds:
  – Semantic analysis: type checking, error checking, etc.
  – Optimization: dead-code elimination, common subexpression elimination, function inlining, register allocation, etc.
  – Code generation: instruction selection

• Representations are more machine specific, less language specific as translation proceeds
(Simplified) Compiler Structure

Source Code (Character stream)
if (b == 0) a = 0;

Lexical Analysis

Parsing

Abstract Syntax Tree

Intermediate Code Generation

Intermediate Code

Code Generation

Front End (machine independent)

Middle End (compiler dependent)

Back End (machine dependent)

Assembly Code
ADD(R0, R1, R0)
BNZR(skip)
ADD(R2, R0, R0)
LABEL(skip)
Typical Compiler Stages

- Lexing \rightarrow \text{token stream}
- Parsing \rightarrow \text{abstract syntax}
- Disambiguation \rightarrow \text{abstract syntax}
- Semantic analysis \rightarrow \text{annotated abstract syntax}
- Translation \rightarrow \text{intermediate code}
- Control-flow analysis \rightarrow \text{control-flow graph}
- Data-flow analysis \rightarrow \text{interference graph}
- Register allocation \rightarrow \text{assembly}
- Code emission

- Optimizations may be done at many of these stages
- Different source language features may require more/different stages
Compilation & Execution

Source code

Compiler

Assembly Code

Assembler

Object Code

Linker

Library code

Fully-resolved machine Code

Loader

Executable image

foo.c

gcc -S

foo.s

as

foo.o

ld

foo

(Usually: gcc -o foo foo.c)
INTERPRETERS

How to represent programs as data structures.
How to write programs that process programs.
Factorial: Everyone’s Favorite Function

Consider this implementation of factorial in a hypothetical programming language:

```plaintext
X = 6;
ANS = 1;
whileNZ (x) {
    ANS = ANS * X;
    X = X + -1;
}
```

We need to describe the constructs of this hypothetical language

- **Syntax**: which sequences of characters count as a legal “program”?
- **Semantics**: what is the meaning (behavior) of a legal “program”?
Grammar for a Simple Language

- Concrete syntax (grammar) for a simple imperative language
  - Written in “Backus-Naur form”
  - `<exp>` and `<cmd>` are nonterminals
  - ‘::=’, ‘|’, and `<...>` symbols are part of the meta language
  - keywords, like `skip` and `ifNZ` and symbols, like '{' and '+' are part of the object language

- Need to represent the abstract syntax (i.e. hide the irrelevant of the concrete syntax)
- Implement the operational semantics (i.e. define the behavior, or meaning, of the program)
Writing an interpreter

Simple.hs