# CS536 Lecture 2

Thursday 22 January 2015

Reminders:

- Register on Piazza.
- P1 assigned, Part 1 due Jan 28 (no late submissions), Part 2 due Feb 2.
- Email ansari@cs.wisc.edu about VISA, any conflicts, etc.

Last class:

- Course Intro and Overview
- Phases of a Compiler (analysis)

Today:

- Phases of a Compiler (synthesis)
- Finite State Machines

# Review: Phases of a Compiler

# Synthesis Phases (cont'd)

## Intermediate Code Generation

Input:
Output:

How?

Why?

Back to example:

```
temp1 = 0 - 3
move temp1 param1
call abs
move return1 temp2
temp3 = 2 * t
temp4 = temp2 + temp3
current_snow = temp4
```

## Optimization

Goal:

Input:
Output:

What kind of “optimization”?

Consequences:

## Code generation

Input: (optimized) intermediate representation

Output: target code

# More on the Symbol Table (P1)

The compiler needs to keep track of names in different phases:

Semantic Analyzer:

Code Generation:

Optimization:

Scope and Block Visibility:

Nested Blocks

Mapping variable use to declaration

Lifetime of variable

Example:

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

Symbol table needs mechanism to encode block structure for easy variable lookup. Implement it as a list of HashMaps (P1).

Operations on the symbol table:

Add scope: add a hashtable to the beginning of the list
Add declaration: add a symbol to the *first* hashtable in the list
Local/global lookup
Remove scope

# Scanning

The scanner translates a sequence of _____ into a sequence of _____.

Each time a scanner is called, it should:

- find the longest sequence of characters forming a token and return it.

We will use scanner *generators*:

- One “regular expression” for each token
- Regular expressions for things to ignore after recognizing them

We model the complexity of a scanner using abstractions from language theory: regular languages and finite state automata.

# Finite State Machines

**Function:** “Accept” (recognize) strings in a (specific type of) “language.”

FSMs are powerful enough for regular languages, which in turn are a good enough way of modeling tokens in a programming language.

Input: String (sequence of characters), finite.

Output: Accept/Reject

> For a given regular language, an FSM computes whether a string belongs to that language or not.

**Example:** An FSM that recognizes // single line comments

## Another Example

What language does this accept?

## How does an FSM work?

Concept:

Algorithm:

Set `current_state` = start state
Set `input_char` = first input character

do
    if there is an edge out of `current_state` labelled `input_char` going into `next_state`
        `current_state = next_state`
        `input_char = next input character`
    else
        **stuck**, throw exception
while
    not **stuck** and input string is not consumed yet

if input string is fully consumed and current_state is a final state: **accept**
else: **reject**

This procedure defines a language: the set of strings accepted by an FSM.

## Example: Hex literals

Rules:

- must start with 0x or 0X
- followed by at least one hex digit (0-9, a-f, A-F)
- can optionally have a long specifier at the end (L or l)

Example strings:

## Example: Hex Literals (cont'd)

Coding the FSM:

## Example: C/C++ identifiers

A C/C++ identifier is a sequence of one or more letters, digits, or underscores. It cannot start with a digit.

FSM:

What if you wanted to add the restriction that it can't end with an underscore?