15-411: Compiler Design — Fall 2017

Recitation 2: Lexing — 15 September

Before you can do any sort of analysis or transformations in your compiler, you have to perform **lexical analysis**. Lexical analysis transforms your program from a string of characters to a list of tokens, and assigns each token a grammatical meaning. From here, we parse those tokens to generate the abstract syntax tree.

Checkpoint 0

Lex the following segment of a C0 program.

```
if (score < 100) {
   return 1;
}
```

Regular Expressions

First we need to define our alphabet. For L1, (and all of the other labs), we will be working with the ASCII character set.

Once we have an alphabet, we define our tokens. Our definitions will be **regular expressions**. From the lecture notes:

Regular expressions r, s are expressions that are recursively built of the following form:

regex	matches
a	matches the specific character a from the input alphabet
$[a-z]$	matches a character in the specified range of letters a to z
ϵ	matches the empty string
$r\|s$	matches a string that matches r or one that matches s
rs	matches a string that can somehow be split into two parts, the first matching r, the second matching s
$r*$	matches a string that consists of n parts where each part matches r, for any natural number $n \geq 0$

Checkpoint 1

Define a regular expression that matches: a valid L1 identifier; a hexadecimal constant; a typedef.

Implementation: Finite Automata

Regular expressions lend themselves naturally to finite automata. Implementation is mostly straightforward, and there are some fun proofs that say regular expressions and finite automata are interchangeable.

When constructing an automata for a regular expression, we need to be careful and accept the longest possible valid token. If we did not have that guarantee, `formula` might lex as the keyword `for` followed by an identifier, rather than as a single identifier.

We use nondeterministic finite automata to match the longest possible expression. At a state where you need later information to determine if the token is complete, we simply consider all possibilities and choose later.

In lecture, we discussed details of how we can efficiently convert a NFA into a DFA, but in your compiler you will use a lexer generator that will handle the implementation for you. You simply define tokens in a regular expression-like manner and the lexer will generate a function that consumes a string input and produces a list of tokens.

Checkpoint 2

Create a DFA that accepts valid hexadecimal constants and rejects all other strings.

Lab 1 Tip: Spilling Temps

We can't fit all of our data in registers, so we spill into memory. But we need at least one operand in a register for most arithmetic operations. This is getting into the software engineering part of the course, but we will outline one strategy that you can use.

You will need to reserve a register, typically `%r11d`. Perform register allocation, then scan through your instructions looking for memory-memory operations. You then insert a `mov` from the destination to `%r11d`, perform the operation, then move `%r11d` back to memory.

In a functional language, you can implement this in a pass similar to code generation, where you case on instruction type and produce either a list with the input instruction, or a list with the moves into and out of `%r11d`.