Lecture 2

CS 473: COMPILER DESIGN

Adapted from slides by Steve Zdancewic
Lexical analysis, tokens, regular expressions, automata, lexer generators

LEXING
Compilation in a Nutshell

Source Code
(Character stream)
if (b == 0) { a = 1; }

Token stream:
if ( b == 0 ) { a = 0 ; }

Lexical Analysis

Parsing

Analysis & Transformation

Backend

Assembly Code

11:
   cmpq %eax, $0
   jeq 12
   jmp 13
12:
   ...

...
First Step: Lexical Analysis

• Change the character stream “if (b == 0) a = 0;” into tokens:

```
if ( b == 0 ) { a = 0 ; }
```

```
IF; LPAREN; ID(“b”); EQEQ; NUM(0); RPAREN; LBRACE; ID(“a”); EQ; INT(0); SEMI; RBRACE
```

• Token: data type that represents indivisible “chunks” of text:
  – Identifiers: a y11 elsex _100
  – Keywords: if else while
  – Integers: 2 200 -500 5L
  – Floating point: 2.0 .02 1e5
  – Symbols: + * ` { } ( ) ++ << >> >>>
  – Strings: "x" "He said, "Are you?"
  – Comments: (* CIS341: Project 1 ... *) /* foo */

• Often delimited by whitespace (‘ ’, \t, etc.)
  – In some languages (e.g. Python or Haskell) whitespace is significant
How hard can it be?
handlex.c

**DEMO: HANDLEX**
Lexing By Hand

• How hard can it be?
  – Tedious and painful!

• Problems:
  – Precisely define tokens
  – Matching tokens simultaneously
  – Need look ahead
  – Error handling
  – Hard to maintain
PRINCIPLED SOLUTION TO LEXING
Regular Expressions

• Regular expressions precisely describe sets of strings.

• A regular expression $R$ has one of the following forms:
  - $\varepsilon$  
    Epsilon stands for the empty string
  - 'a'  
    An ordinary character stands for itself
  - $R_1 \mid R_2$  
    Alternatives, stands for choice of $R_1$ or $R_2$
  - $R_1R_2$  
    Concatenation, stands for $R_1$ followed by $R_2$
  - $R^*$  
    Kleene star, stands for zero or more repetitions of $R$

• Useful extensions:
  - "foo"  
    Strings, equivalent to 'f' 'o' 'o'
  - $R+$  
    One or more repetitions of $R$, equivalent to $RR^*$
  - $R?$  
    Zero or one occurrences of $R$, equivalent to ($\varepsilon \mid R$)
  - ['a'-'z']  
    One of a or b or c or ... z, equivalent to (a | b | ... | z)
  - [^'0'-'9']  
    Any character except 0 through 9
  - .  
    Any character
Example Regular Expressions

• Recognize the keyword “if”: "if"
• Recognize a digit: ['0'-'9']
• Recognize an integer literal: '-?'['0'-'9']+
• Recognize an identifier:
  (['a'-'z'] | ['A'-'Z']) ('0'-'9' | '_') | ['a'-'z'] | ['A'-'Z'])*
How to Match?

• Consider the input string: \textit{if}x = 0
  – Could lex as: \texttt{if x = 0} or as: \texttt{ifix = 0}

• Regular expressions alone are ambiguous, need a rule for choosing between the options above

• Most languages choose “longest match”
  – So the 2\textsuperscript{nd} option above will be picked
  – Note that only the first option is “correct” for parsing purposes

• Conflicts: arise due to two tokens whose regular expressions have a shared prefix
  – Ties broken by giving some matches higher priority
  – Example: keywords have priority over identifiers
  – Usually specified by order the rules appear in the \texttt{lex} input file
Lexer Generators

• Reads a list of regular expressions: $R_1, \ldots, R_n$, one per token.
• Each token has an attached “action” $A_i$ (just a piece of code to run when the regular expression is matched):

```
'-'?digit+ { return NUM; }
'+' { return PLUS; }
'if' { return IF; }
[a-z]([0-9]|[a-z]|')* { return ID; }
whitespace+ { /* do nothing */ }
```

• Generates scanning code that:
  1. Decides whether the input is of the form $(R_1 | \ldots | R_n)^*$
  2. Whenever the scanner matches a (longest) token, runs the associated action
Questions

Top
Finite Automata

- Every regular expression can be recognized by a finite automaton
- Consider the regular expression: "" " [^"" ] * ""
- An automaton (DFA) can be represented as:
  - A transition table:

<table>
<thead>
<tr>
<th></th>
<th>&quot;&quot;</th>
<th>Non-&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>ERROR</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>ERROR</td>
<td>ERROR</td>
</tr>
</tbody>
</table>

  - A graph:

```
0 -> "" -> 1
      "" -> 2 (Error)
```
• Every regular expression can be recognized by a finite automaton

• Strategy: consider every possible regular expression:

'a'

ε

R₁R₂

What about?

R₁ | R₂
Nondeterministic Finite Automata

- A finite set of states, a start state, and accepting state(s)
- Transition arrows connecting states
  - Labeled by input symbols
  - Or $\varepsilon$ (which does not consume input)
- **Nondeterministic**: two arrows leaving the same state may have the same label
Converting regular expressions to NFAs is easy.
Assume each NFA has one start state, unique accept state.
RE to NFA (cont’d)

• Sums and Kleene star are easy with NFAs
• Construct an NFA for the following regular expression:

\((a^*b^*) \mid (b^*a^*)\)
Deterministic Finite Automata

• An NFA accepts a string if there is any way to get to an accepting state
  – To implement, we either have to try all possibilities or get good at guessing!

• A deterministic finite automata never has to guess: two arrows leaving the same state must have different labels, and never $\varepsilon$

• This means that action for each input is fully determined!

• We can make a table for each state: “if you see symbol X, go to state Y”

• Fortunately, we can convert any NFA into a DFA!
NFA to DFA conversion (Intuition)

- Idea: Run all possible executions of the NFA “in parallel”
- Keep track of a set of possible states: “finite fingers”
- Consider: – ? [ 0 – 9 ] +

NFA representation:

DFA representation:
Summary of Lexer Generator Behavior

• Take each regular expression $R_i$ and its action $A_i$

• Compute the NFA formed by $(R_1 | R_2 | \ldots | R_n)$
  – Remember the actions associated with the accepting states of the $R_i$

• Compute the DFA for this big NFA
  – There may be multiple accept states (why?)
  – A single accept state may correspond to one or more actions (why?)

• Compute the minimal equivalent DFA
  – There is a standard algorithm due to Myhill & Nerode

• Produce the transition table

• Implement longest match:
  – Start from initial state
  – Follow transitions, remember last accept state entered (if any)
  – Accept input until no transition is possible (i.e. next state is “ERROR”)
  – Perform the highest-priority action associated with the last accept state; if no accept state there is a lexing error
Homework 1: Lexing

• Do exercises 1-4 from chapter 2 of the textbook
• Practice writing regular expressions and finite automata and converting between them
• Grad students have extra problems
• Due next Wednesday at the start of class
• Submit via Gradescope
Questions

Top
Compilation in a Nutshell

Source Code
(Character stream)
```
if (b == 0) { a = 1; }
```

Token stream:
```
if ( b == 0 ) { a = 0 ; } 
```

Assembly Code
```
l1:
    cmpq %eax, $0
    jeq 12
    jmp 13
l2:
    ...
```
Lexer Generators

• Reads a list of regular expressions: $R_1, \ldots, R_n$, one per token.
• Each token has an attached “action” $A_i$ (just a piece of code to run when the regular expression is matched):

```
'-'?digit+  { return NUM; }  
'+'         { return PLUS; }  
'if'        { return IF; }     
[a-z]([0-9]|[a-z]|'_')* { return ID; }     
whitespace+ { /* do nothing */ }  
```

• Generates scanning code that:
  1. Decides whether the input is of the form $(R_1 | \ldots | R_n)^*$
  2. Whenever the scanner matches a (longest) token, runs the associated action
lex1.lex, lex2.lex

**DEMO: FLEX**

HTTPS://SOURCEFORGE.NET/PROJECTS/FLEX/
MANUAL: HTTPS://WESTES.GITHUB.IO/FLEX/MANUAL/
Sometimes we want to use different lexers for different parts of a program.

For instance, comments:

```
abc /* 123 */ 123
```

**Start states** let us specify multiple sets of lexing rules and switch between them.

```flex
%s COMMENT     // define a new ruleset for comments

// INITIAL is the default lexer
<INITIAL>[a-z]+    { return ID;  }
<INITIAL>[0-9]+    { return NUM;  }

// switch to the comment lexer
<INITIAL>"/*"      { BEGIN COMMENT;  }
<COMMENT>.
    { continue;  }

// switch back when we’re done
<COMMENT>"*/"      { BEGIN INITIAL;  }
```

Demo: states.lex
Questions

Top