CS 473: COMPILER DESIGN

Adapted from slides by Steve Zdancewic, UPenn
Lexical analysis, tokens, lexer generators, regular expressions, automata
Compilation in a Nutshell

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

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

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:

\[
\begin{array}{c}
\text{IF; LPAREN; ID(“b”); EQEQ; NUM(0); RPAREN; LBRACE;}
\text{ID(“a”); EQ; INT(0); SEMI; RBRACE}
\end{array}
\]

• 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: (* CS476: 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
  – Multiple tokens may match
  – Each case’s behavior depends on other cases
  – Error handling is tricky
  – Hard to maintain
LEXER GENERATOR (LEX)
Regular Expressions: Refresher

• The key is a compact, checkable way of writing down what all the tokens are

• We can use regular expressions!

• Exercise: Write a regular expression for “strings of one or more a’s followed by one or more b’s.”

• Example solutions:

  a+b+
  aa*bb*
Lexer Generators

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

\[
\begin{align*}
\text{'-'}?\text{digit}+ & \quad \{ \text{return NUM;} \} \\
\text{'+'} & \quad \{ \text{return PLUS;} \} \\
\text{'if'} & \quad \{ \text{return IF;} \} \\
[a-z](0-9)[a-z]|'_'\} & \quad \{ \text{return ID;} \} \\
\text{whitespace}\} & \quad \{ /* do nothing */ \}
\end{align*}
\]

- 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
DEMO: LEX
HTTPS://SOURCEFORGE.NET/PROJECTS/FLEX/
MANUAL:
HTTP://DINOSAUR.COMPILEERTOOLS.NET/LEX/INDEX.HTML
Questions

Top
Anatomy of a Lex file

```c
%{

typedef union { ... } YYSTYPE;
YYSTYPE yylval;

}%

%%

"if"   { return IF; }
";"    { return SEMICOLON; }
[0-9]+ { yylval.ival = atoi(yytext);
         return NUM; }

%%

int main(){
    t = yylex();
}
```

prelude: definitions and helper functions, written in C

body: regular expressions and associated actions (again, written in C)

end: arbitrary C code, can call the lexing function yylex
keywords and special chars: just return the token

tokens with content: set yylval, return token type

if it doesn’t affect the program, don’t make a token

use a wildcard to catch input that isn’t a token
Running the Lexer

• Running `lex <filename>.lex` generates a file `lex.yy.c`

• The file defines a function called `yylex`, which looks for matches to the regular expressions, and runs the associated action when it finds one
  – If there are multiple possible matches, it chooses the `longest match`

• If lexer has a main function, we can just compile and run `lex.yy.c`

• Otherwise, we can use the lexer as a library, and call the generated `yylex` function in other files (the rest of the compiler)
Program 1: Lexer

- Posted on the course website ([https://www.cs.uic.edu/~mansky/teaching/cs473/sp21/program1.html](https://www.cs.uic.edu/~mansky/teaching/cs473/sp21/program1.html))
- Extend a simple lexer with support for more features
- Due next Wednesday at the start of class
- Submit via Gradescope

Extending a lexer:
1. What do we want the syntax to look like?
2. What tokens does it use that we don’t already have?
3. Add cases and regexps for those tokens
4. Add actions for those regexps, returning appropriate values
Questions

Top