Lexical Analysis

ICS312
Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

Lexical Analysis

- **Lexical Analysis**, also called ‘scanning’ or ‘lexing’
- It does two things:
  - Transforms the input source string into a sequence of substrings
  - Classifies them according to their ‘role’
- The input is the source code
- The output is a list of **tokens**
- Example input:
  
  ```
  if (x == y) 
  z = 12;
  else 
  z = 7;
  ```
- This is really a single string:
  
  ```
  if ((x==y)z = 12;else if z = 7;)
  ```

Tokens

- A token is a **syntactic category**
- **Example tokens:**
  - Identifier
  - Integer
  - Floating-point number
  - Keyword
  - etc.
- In English we would talk about
  - Noun
  - Verb
  - Adjective
  - etc.
Lexeme

- A lexeme is the string that represents an instance of a token
- The set of all possible lexemes that can represent a token instance is described by a pattern
- For instance, we can decide that the pattern for an identifier is:
  - A string of letters, numbers, or underscores, that starts with a capital letter

Lexing output

```
if ((x==y)) z = 12;
else z = 7;
```

```
<key, 'if'> <openparen> <id, 'x'> <op, '=='<id, 'y'> <closeparen> <id, 'z'> <op, '='<integer, '12'> <semic>
```

```
<key, 'else'> <id, 'z'> <op, '='<integer, '7'> <semic>
```

- Note that the lexer removes non-essential characters
  - Spaces, tabs, linefeeds
  - And comments!
  - Typically a good idea for the lexer to allow arbitrary numbers of white spaces, tabs, and linefeeds

The Lookahead Problem

- Characters are read in from left to right, one at a time, from the input string
- The problem is that it is not always possible to determine whether a token is finished or not without looking at the next character
- Example:
  - Is character 'f' the full name of a variable, or the first letter of keyword 'for'?  
  - Is character '=' an assignment operator or the first character of the '==' operator?
- In some languages, a lot of lookahead is needed
- Example: FORTRAN
  - Fortran removes ALL white spaces before processing the input string
  - DO 5 I = 1.25 is valid code that sets variable DO5I to 1.25
  - But 'DO 5 I = 1.25' could also be the beginning of a for loop!

The Lookahead Problem

- It is typically a good idea to design languages that require 'little' lookahead
  - For each language, it should be possible to determine how many lookahead characters are needed
- Example with 1-character lookahead:
  - Say that I get an 'if' so far
  - I can look at the next character
  - If it's a ' ', ',', '.', then I don't read it; I stop here and emit a TOKEN_IF
  - Otherwise I read the next character and will most likely emit a TOKEN_ID
- In practice one implements lookahead/pushback
  - When in need to look at next characters, read them in and push them onto a data structure (stack/fifo)
  - When in need of a character get it from the data structure, and if empty from the file
A Lexer by Hand? You’re kidding!

Example: Say we want to write the code to recognizes the keyword ‘if’
```
c = readchar();
if (c == 'i') {
  c = readchar();
  if (c == 'f') {
    c = readchar();
    if (c not alphanumeric) {
      pushback(c);
      emit(TOKEN_IF)
    } else {
      // build a TOKEN_ID
    }
  } else {
    // something else
  }
} else {
  // something else
}
```

A Lexer by Hand?

There are many difficulties when writing a lexer by hand as in the previous slide
- Many types of tokens
  - fixed string
  - special character sequences (operators)
  - numbers defined by specific/complex rules
- Many possibilities of token overlaps
  - Hence, many nested if-then-else in the lexer’s code
- Coding all this by hand is very painful
  - And it’s difficult to get it right
- Nevertheless, some compilers have an implemented-by-hand lexer for higher speed

Regular Expressions

To avoid the endless nesting of if-then-else one needs a formalization of the lexing process
If we have a good formalization, we could even generate the lexer’s code automatically!

Lexer Specification

Question: How do we formalize the job a lexer has to do to recognize the tokens of a specific language?
Answer: We need a language!
  - More specifically, we’re going to talk about the language of tokens!
  - What’s a language?
    - An alphabet (typically called Σ)
      - e.g., the ASCII characters
    - A subset of all the possible strings over Σ
  - We just need to provide a formal definition of a the language of the tokens over Σ
    - Which strings are tokens
    - Which strings are not tokens
  - It turns out that for all (reasonable) programming languages, the tokens can be described by a regular language
    - i.e., a language that can be recognized by a finite automaton
    - A lot of theory here that I’m not going to get into
Describing Tokens

- Most popular way to describe tokens: **regular expressions**
- Regular expressions are just **notations**, which happen to be able to represent regular languages
  - A regular expression is a string (in a meta-language) that describes a pattern (in the token language)
- L(A) is the language represented by regular expression A
  - Remember that a language is just a set of valid strings
- Basic: L('c') = \{ 'c' \}
- Concatenation: L(AB) = \{ ab | a in L(A) and b in L(B) \}
  - L('i' 'f') = \{ 'if' \}
  - L(('i')('f')) = \{ 'if' \}
- Union: L(A|B) = \{ x | x in L(A) or x in L(B) \}
  - L('if'|'then'|'else') = \{ 'if', 'then', 'else' \}
  - L(('0'|'1') ('1'|'0')) = \{ '00', '01', '10', '11' \}

REs for Keywords

- It is easy to define a RE that describes all keywords
  - Key = 'if' | 'else' | 'for' | 'while' | 'int' | ..
- These can be split in groups if needed
  - Keyword = 'if' | 'else' | 'for' | ..
  - Type = 'int' | 'double' | 'long' | ..
- The choice depends on what the next component (i.e., the parser) would like to see

Regular Expression Overview

<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\varepsilon)</td>
<td>empty pattern</td>
</tr>
<tr>
<td>a * *</td>
<td>Any pattern represented by 'a'</td>
</tr>
<tr>
<td>ab</td>
<td>Strings with pattern 'a' followed by pattern 'b'</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a * *</td>
<td>Zero or more occurrences of pattern 'a'</td>
</tr>
<tr>
<td>a * *</td>
<td>One or more occurrences of pattern 'a'</td>
</tr>
<tr>
<td>a? * *</td>
<td>(a \mid \varepsilon)</td>
</tr>
<tr>
<td>. * *</td>
<td>Any single character (not very standard)</td>
</tr>
</tbody>
</table>

Let's look at how REs are used to describe tokens

RE for Numbers

- Straightforward representation for integers
  - digits = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
  - integer = digits\*
- RE systems allow the use of '-' for ranges, sometimes with '[' and ']
  - digits = [0-9]+
- Floating point numbers are much more complicated
  - 2.00, .12e-12, 312.00001E+12, 4, 3.141e-12
- Here is one attempt
  - ('+'|'-'|\varepsilon)(digit\* '.'? | digits\* '.')\*(E'|'e')(\'+\'|'\-')(\varepsilon|digits\*)))
- Note the difference between meta-character and language-characters
  - '+' versus +, '-' versus -, '(' versus (, etc.
- Often books/documentations use different fonts for each level of language
RE for Identifiers

- Here is a typical description
  - letter = a-z | A-Z
  - ident = letter ( letter | digit | '_' )*
    - Starts with a letter
    - Has any number of letter or digit or '_' afterwards
- In C: ident = (letter | '_') (letter | digit | '_')*

RE for Phone Numbers

- Simple RE
  - digit = 0-9
  - area = digit digit digit
  - exchange = digit digit digit
  - local = digit digit digit digit
  - phononenumber = ('area') '?' exchange ('-'|' ') local

- The above describes the $10^{3+3+4}$ strings of the L(phononenumber) language

REs in Practice

- The Linux grep utility allows the use of REs
  - Example with phone numbers
    - grep '(\[0-9\]\{3\}) \{0,1\}\[0-9\]\{3\}\[\-|\ ]\[0-9\]\{4\}' file
  - The syntax is different from that we’ve seen, but equivalent
  - Sadly, there is no single standard for RE syntax
- Perl implements regular expressions
- (Good) text editors implement regular expressions
  - e.g., for string replacements
- At the end of the day, we often have built for ourselves tons of regular expressions
  - Many programs you use everyday use REs internally, including compilers

Now What?

- Now we have a nice way to formalize each token (which is a set of possible strings)
- Each token is described by a RE
  - And hopefully we have made sure that our REs are correct
  - Easier than writing the lexer from scratch
  - But still requires that one be careful
- Question: How do we use these REs to parse the input source code and generate the token stream?
- A little bit of ‘theory’
  - REs characterize Regular Languages
  - Regular Languages are recognized by Finite Automata
  - Therefore we can implement REs as automata
Finite Automata

- A finite automaton is defined by
  - An input alphabet: \( \sum \)
  - A set of states: \( S \)
  - A start state: \( n \)
  - A set of accepting states: \( F \) (a subset of \( S \))
  - A set of transitions between states: subset of \( S \times S \)

Transition Example
- \( s_1: a \rightarrow s_2 \)
- If the automaton is in state \( s_1 \), reading a character ‘a’ in the input takes the automaton in state \( s_2 \)
- Whenever reaching the ‘end of the input,’ if the state the automaton is in is in a accept state, then we accept the input
- Otherwise we reject the input

Automaton Examples

- This automaton accepts input ‘if’

Automaton as Graphs

- A state
- The start state
- An accepting state
- A transition

Automaton Examples

- This automaton accepts strings that start with a 0, then have any number of 1’s, and end with a 0
- Note the natural correspondence between automata and REs: \( 01^*0 \)
- Question: can we represent all REs with simple automata?
- Answer: yes
- Therefore, if we write a piece of code that implements arbitrary automata, we have a piece of code that implements arbitrary REs, and we have a lexer!
  - Not _this_ simple, but close
Non-deterministic Automata

- The automata we have seen so far are called Deterministic Finite Automata (DFA)
  - At each state, there is at most one edge for a given symbol
  - At each state, transition can happen only if an input symbol is read
    - Or the string is rejected
- It turns out that it's easier to translate REs to Non-deterministic Finite Automata (NFA)
  - There can be 'ε-transitions'!
    - Taken arbitrarily without consuming an input character
  - There can be multiple possible transitions for a given input symbol at a state
    - The automaton can take them all simultaneously (see later)

Example REs and DFA

- Say we want to represent RE 'a*b*c*d*e' with a DFA

Example REs and NFA

- 'a*b*c*d*e': much simpler with a NFA

- With ε-transitions, the automaton can ‘choose’ to skip ahead, non-deterministically

Example REs and NFA

- 'a*b+c+d+e': easy modification

- But now we have multiple choices for a given character at each state!
  - e.g., two ‘a’ arrows leaving n
NFA Acceptance

- When using an NFA, one must constantly keep track of all possible states.
- If at the end of the input (at least) one of these states is an accepting state, then accept, otherwise reject.

Input string: 010

REs and NFA

- So now we're left with two possibilities:
  - Possibility #1: design DFAs
    - Easy to follow transitions once implemented
    - But really cumbersome
  - Possibility #2: design NFAs
    - Really trivial to implement REs as NFAs
    - But what happens on input characters?
      - Non-deterministic transitions
      - Should keep track of all possible states at a given point in the input!
- It turns out that:
  - NFAs are not more powerful than DFAs
  - There are systematic algorithms to convert NFAs into DFAs and to limit their sizes
  - There are simple techniques to implement DFAs in software quickly.
Implementing a Lexer

- Implementing a Lexer is now straightforward
  - Come up with a RE for each token category
  - Come up with an NFA for each RE
  - Convert the NFA (automatically) to a DFA
  - Write a piece of code that implements a DFA
    - Pretty easy with a decent data-structure, which is basically a transition table
  - Implement your lexer as a ‘bunch of DFAs’
  - No nested if-then-else ad infinitum :)
- The above has been understood for decades and we now have automatic lexer generators!
- Well-known examples are lex and flex
- Let’s look at ANTLR

ANTLR

- ANTLR: A tool to generate lexer/parsers
- Let’s look on the course Web site for how to download/install/run ANTLR...
- Say we want to define a language with the following:
  - Reserved keywords: int, if, endif, while, endwhile, print
  - An addition operator: ‘+’
  - An assignment operator: ‘=’
  - An equal operator: ‘==’
  - A not-equal operator: ‘!=’
  - Integers
  - Variable names as strings of lower-case letters
  - Semicolons for terminating statements
  - Left and right parentheses
  - The ability to ignore white spaces, tabs, carriage returns, etc.

ANTLR

- Basics of Regular Expressions in ANTLR:
  - Regular expression name (chosen by you)
  - Colon
  - Regular expression
  - Semicolon
- Example:
  - DIGIT : [0-9] ;
  - VARIABLE: [a-z]+ ;
  - EQUAL: ‘==’ ;
- Let’s look at the full example on the Web site, and run it...
  - Not that this example has some “parser stuff” at the beginning, but we’re ignoring that for now

Conclusion

- 20,000 ft view
  - Lexing relies on Regular Expressions, which rely on NFAs, which rely on DFAs, which are easy to implement
  - Therefore lexing is ‘easy’
- Lexing has been well-understood for decades and lexer generators are known
  - We’ve seen and will use ANTLR
- The only motivation to write a lexer by hand: speed