Lexical Analysis

Today we start

Optimization

Lexical Analysis

• What do we want to do? Example:
  if (i == j)
  z = 0;
else
  z = 1;

• The input is just a sequence of characters:
  \texttt{\textbackslash n\textbackslash n}\texttt{i\textbackslash n\textbackslash t\textbackslash n}\texttt{j = 0;\textbackslash n\textbackslash n}\texttt{else\textbackslash n\textbackslash t\textbackslash n}z = 1;

• Goal: Partition input string into substrings
  And classify them according to their role

The Structure of a Compiler

Source

Lexical analysis

Tokens

Parsing

Optimization

Intermediate Language

Code Gen.

Machine Code

Today we start
What's a Token?

- Output of lexical analysis is a stream of tokens
- A token is a syntactic category
  - In English: noun, verb, adjective, ...
  - In a programming language: Identifier, (Keyword) Integer, Float, IF, THEN ...
- Parser relies on the token distinctions:
  - E.g., identifiers are treated differently than keywords

Tokens

- A token corresponds to a string or a set of strings.
- Identifier: strings of letters or digits, starting with a letter
- Integer: a non-empty string of digits
- (Keyword) IF: string if
- Whitespace: a non-empty sequence of blanks, newlines, and tabs
- OpenPar: a left-parenthesis

Lexical Analyzer: Implementation

- An implementation must do two things:
  1. Recognize substrings corresponding to tokens
  2. Return the value or lexeme of the token
    - The lexeme is the substring
Example

- When reading the following:
\[
\text{if} (i == j) \text{n} \text{\n} \text{z} = 0; \text{n} \text{\n} \text{else} \text{n} \text{\n} \text{z} = 1;
\]
- **Token-lexeme** pairs returned by the lexer:
  - (Whitespace, "\n")
  - ((Keyword) IF, "if")
  - (LeftPar, "(")
  - (Identifier, "i")
  - (Relation, "==")
  - (Identifier, "j")

Lexical Analyzer: Implementation

- The lexer usually discards "uninteresting" tokens that don't contribute to parsing.
- Examples: Whitespace, Comments
- Question: What happens if we remove all whitespace and all comments prior to lexing?

Lookahead.

- Two important points:
  1. The goal is to partition the string. This is implemented by reading left-to-right, recognizing one token at a time.
  2. "Lookahead" may be required to decide where one token ends and the next token begins
- Even our simple example has lookahead issues
  
  i vs. if
  
  = vs. ==
Next

- We need
  - A way to describe the lexemes of each token
    - Identifier: strings of letters or digits, starting with a letter
    - Computer cannot understand the above informal description.
  - A way to resolve ambiguities
    - Is if two variables i and f?
    - Is == two equal signs ==

Regular Languages

- There are several formalisms for specifying tokens
  - “formalism” means a computer can recognize
- Regular languages are the most popular
  - Simple and useful theory
  - Easy to understand
  - Efficient implementations

Languages

**Def.** Let $\Sigma$ be a set of characters. A *language over* $\Sigma$ is a set of strings of characters drawn from $\Sigma$

($\Sigma$ is called the *alphabet*)
Examples of Languages

• Alphabet = English characters
• Language = English sentences
• Not every string on English characters is an English sentence

• Alphabet = ASCII
• Language = C programs
• Note: ASCII character set is different from English character set

Notation

• Languages are sets of strings.
• Need some notation for specifying which sets we want
• For lexical analysis we care about regular languages, which can be described using regular expressions.

Regular Expressions and Regular Languages

• Each regular expression is a notation for a regular language (a set of words)
• If $A$ is a regular expression then we write $L(A)$ to refer to the language denoted by $A$
Atomic Regular Expressions

- Single character: 'c' if c in Σ
  \[ L('c') = \{ "c" \} \]

- Concatenation: AB (where A and B are reg. exp.)
  \[ L(AB) = \{ ab | a \in L(A) \text{ and } b \in L(B) \} \]

- Example: \( L('i' 'f') = \{ "if" \} \)
  (we will abbreviate 'i' 'f' as 'if')

Compound Regular Expressions

- Union: A | B
  \[ L(A \mid B) = \{ s | s \in L(A) \text{ or } s \in L(B) \} \]

- Examples:
  \[ L('if' \mid 'then' \mid 'else') = \{ "if", "then", "else" \} \]
  \[ L('0' \mid '1' \mid \ldots \mid '9') = \{ "0", "1", \ldots, "9" \} \]
  (note the \ldots \ are just an abbreviation)

- Another example:
  \[ L(('0' \mid '1') ('0' \mid '1')) = \{ "00", "01", "10", "11" \} \]

More Compound Regular Expressions

- So far we do not have a notation for infinite languages
- Iteration: A^*
  \[ L(A^*) = \{ \epsilon \} \cup L(A) \cup L(AA) \cup L(AAA) \cup \ldots \]

- Examples:
  \[ L('0'^*) = \{ \epsilon, "0", "00", "000", \ldots \} \]
  \[ L('1' '0'^*) = \{ \text{strings starting with 1 and followed by 0's} \} \]
- Epsilon: \( \epsilon \)
  \[ L(\epsilon) = \{ \epsilon \} \]
Example: Keyword

- Keyword: "else" or "if" or "begin" or ...

  'else' | 'if' | 'begin' | ...

(Recall: 'else' abbreviates 'e' 'l' 's' 'e')

Example: Integers

Integer: a non-empty string of digits

digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

number = digit digit*

Abbreviation: \( A^+ = A A^* \)

Example: Identifier

Identifier: strings of letters or digits, starting with a letter

letter = 'A' | ... | 'Z' | 'a' | ... | 'z'

identifier = letter (letter | digit) *

Is (letter* | digit*) the same?
Example: Whitespace

Whitespace: a non-empty sequence of blanks, newlines, and tabs

( ' ' | \t | \n )+

Example: Phone Numbers

• Regular expressions are all around you!
• Consider (510) 643-1481
  \Sigma = \{ 0, 1, 2, 3, ..., 9, (, ), - \}
  area = digit^3
  exchange = digit^3
  phone = digit^4
  number = '(' area ')' exchange '-' phone

Example: Email Addresses

• Consider wushen@cs.wmich.edu

  \Sigma = letters U \{ ., @ \}
  name = letter^*
  address = name '@' name ('.' name)*
Regular Expressions => Lexical Spec. (1)

1. Select a set of tokens
   • Number, Keyword, Identifier, ...

2. Write a R.E. for each token
   • Number = digit+
   • if_keyword = 'if'
   • Identifier = letter (letter | digit)*
   • OpenPar = '('
   • ...

Regular Expressions => Lexical Spec. (2)

3. Construct R, representing a large token category:
   \[ R = R_1 | R_2 | R_3 | ... \]

Facts: If \( s \) in \( L(R) \) then \( s \) is a lexeme
- Furthermore \( s \) in \( L(R_i) \) for some "i"
- This "i" determines the token that is reported

Regular Expressions => Lexical Spec. (3)

4. Let the input be \( x_1...x_n \)
   \( (x_1...x_n \) are characters in the language alphabet)
   • For \( 1 \leq i \leq n \) check
     \[ x_{i-1}x_i \in L(R) \]?

5. It must be that
   \[ x_{i-1}x_i \in L(R) \] for some \( i \) and \( j \)

6. Remove \( x_{i-1}x_i \) from input and go to (4)
Lexing Example

R = Whitespace | Integer | Identifier | '+'

• Parse "f +3 +g"
  - "f" matches R, more precisely Identifier
  - "+" matches R, more precisely '+'
  - ...
  - The token-lexeme pairs are
    (Identifier, "f"), ('+', '+'), (Integer, '3')
    (Whitespace, ' '), ('+', '+'), (Identifier, "g")

• We would like to drop the Whitespace tokens
  - after matching Whitespace, continue matching

Ambiguities (1)

• There are ambiguities in the algorithm
• Example:
  R = Whitespace | Integer | Identifier | '+'
• Parse "foo+3"
  - "f" matches R, more precisely Identifier
  - But also "fo" matches R, and "foo", but not "foo+
• How much input is used? What if
  - x1...xi ∈ L(Rj) and also x1...xk ∈ L(Rk)
  - "Maximal munch" rule: Pick the longest possible substring that matches R

More Ambiguities

R = Whitespace | 'new' | Integer | Identifier

• Parse "new foo"
  - "new" matches R, more precisely 'new'
  - but also Identifier, which one do we pick?
• In general, if x1...xi ∈ L(Rj) and x1...xj ∈ L(Rk)
  - Rule: use rule listed first (j if j < k)

• We must list 'new' before Identifier
Automatic Generation of Lexers

- 2 programs developed at Bell Labs in mid 70's for use with UNIX
  - Lex - transducer, transforms an input stream into the alphabet of the grammar processed by yacc
  - Flex = fast lex, later developed by Free Software Foundation
- Yacc/bison - yet another compiler/compiler (next week)

Input to lexer generator
- List of regular expressions in priority order
- Associated action with each RE

Output
- Program that reads input stream and breaks it up into tokens according the the REs

Lex/Flex

Lex Specification

- Definition section
  - All code contained within "%#" and "/$#" is copied to the resultant program. Usually has token defns established by the parser
  - User can provide names for complex patterns used in rules
  - Any additional lexing states (states preaced by "% directive)
  - Pattern and state definitions must start in column 1 (All lines with a blank in column 1 are copied to resulting C file)

Lex file always has 3 sections:
  - definition section
  - rules section
  - user functions section
Lex Specification (continued)

- Rules section
  - Contains lexical patterns and semantic actions to be performed upon a pattern match. Actions should be surrounded by {} (though not always necessary)
  - Again, all lines with a blank in column 1 are copied to the resulting C program

- User function section
  - All lines in this section are copied to the final .c file
  - Unless the functions are very immediate support routines, better to put these in a separate file

Partial Flex Program

```
D [0-9]
%{ 
  if printf("IF statement\n");
  [a-z]+ printf("tag, value %s\n", yytext);
  {D}+ printf("decimal number %s\n", yytext);
  "++" printf("unary op\n");
  "++" printf("binary op\n");
}
```

Flex Program

```
%{ 
  #include <stdio.h>
  int num_lines = 0, num_chars = 0;
  
  %{ 
    printf("# of lines = %d, # of chars = %d \n", num_lines, num_chars);
  }
  
  main() { 
    yylex();
    printf( "# of lines = %d, # of chars = %d \n", num_lines, num_chars );
  }
%
```

Running the above program:

```
[17] copy02 < flex count.l
[18] copy02 < gen lex yy.c -o
[19] copy02 < a.out < count.l
# of lines = 16, # of chars = 245
```
Another Flex Program

```c
/* recognize articles a, an, the */
#include <stdio.h>

%
[ \t\n]+ /* skip white space - action: do nothing */
a | /* | indicates do same action as next pattern */
an | the {printf("%s: is an article\n", yytext);
[a-zA-Z]+ {printf("%s: ???\n", yytext);
%
main()
{
yylex();
}
Note: yytext is a pointer to first char of the token
yyleng = length of token
```

Token Definitions
( Extended Regular Expression )

• Elementary Operations
  – single characters
    • except " \ . $ ^ \[ \] - ? * + | ( ) / { } % < >
  – concatenation (put characters together)
  – alternation (a|b|c)
    • [ab] == a|b
    • [a-k] == a|b|c|...|i|j|k
    • [a-zA-Z] == any letter or digit
    • [^a] == any character but a

• Elementary Operations (cont.)
  . matches any character except the newline
  * — Kleene Closure
  + — Positive Closure

• Examples:
  – [0-9]* [0-9]+*  
    • note: without the quotes it could be any character
  – [\t\f] — is whitespace
    • (except CR)
    • There is a blank space character before the \t
Token Definitions (Cont)

• Special Characters:
  . -- matches any single character (except newline)
  * and \ -- quote the part as text
  \t -- tab
  \n -- newline
  \b -- backspace
  " -- double quote
  \ -- \n  ? -- this means the preceding was optional
     \b? == a\b
     (ab)? == ab\n
7/5/2006   CS485, Summer II, 2006