Lexing

COP-3402 Systems Software

Paul Gazzillo

A Lexer Groups Characters into "Words"

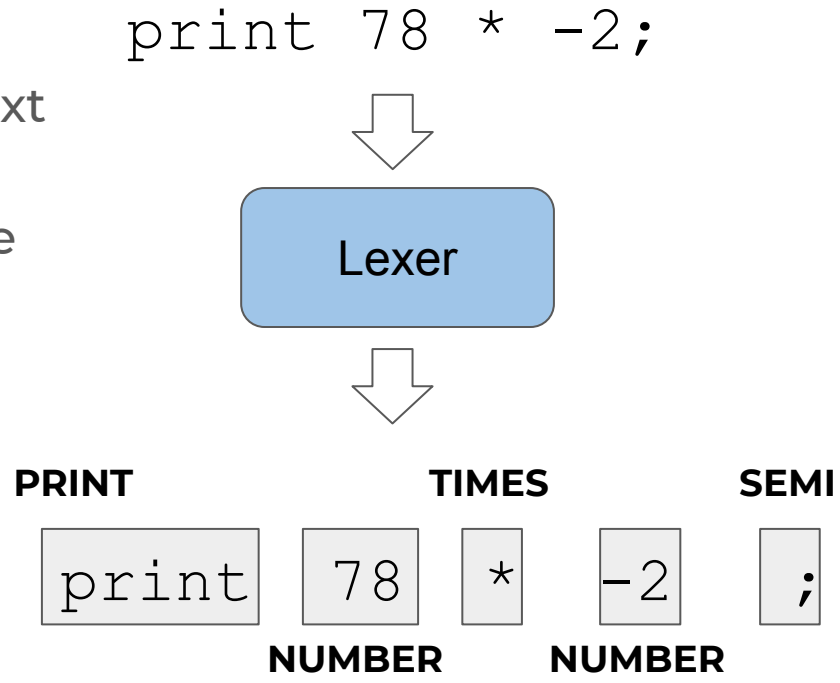
- Source files are text files
- Text files are streams of characters
- Humans automatically "lex", i.e., group characters into words
  - jljaojbdfik3v
  - hello, world!
  - ibetyoucanreadthis
  - 3+-2*10
- A computer does not do this recognition natively
- Compilers use an algorithm to recognize words

Characters in C

- **char** type
- Single quotes, one character at a time
  - char c = 'a';
- Escape sequence for non-printable characters
  - '\n' is a single character
- Read a single character at a time from a file
  - `c = fgetc(file);`

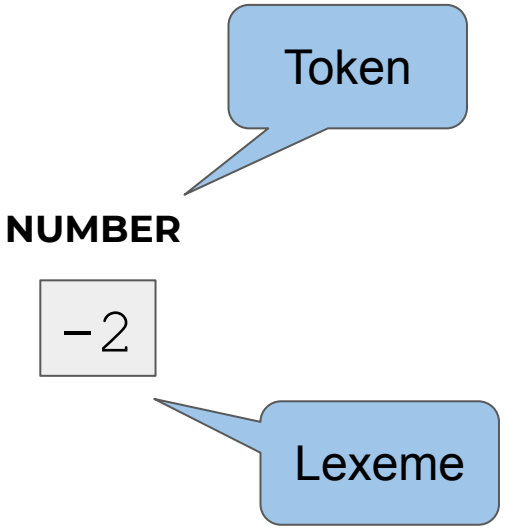
The Lexer Reads and Buffers Characters

- Read each character from a file
- Buffer each character into an array
  - Need max size or use dynamic array
- After a complete word, start on next
  - Need to know pattern of characters
- Each word is labeled with its name
  - 78 is a NUMBER



The Lexer Produces a Stream of Tokens

- Lexemes
  - The actual characters of the token
- Tokens
  - Abstract symbols representing words
- Token attributes
  - The value of the token
  - E.g., the number -2 vs the string
    - These look the same, but one is two ASCII bytes one is four integer bytes



UCF

Examples of Common Tokens

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but `"`, surrounded by `"`'s	`"core dumped"`

Figure 3.2: Examples of tokens

Examples of Tokens with Attributes

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
`<`	**relop**	`LT`
`<=`	**relop**	`LE`
`=`	**relop**	`EQ`
`<>`	**relop**	`NE`
`>`	**relop**	`GT`
`>=`	**relop**	`GE`

Figure 3.12: Tokens, their patterns, and attribute values

Characters Are Represented with Numbers

- Everything is a number (binary code)
- We use ASCII: http://www.asciitable.com
  - 7 bits
  - 2^7 = 128 characters
- Non-printable characters, e.g., newline '\n', tab '\t'
- Printable characters, e.g., 'a', 'A', '5', ';'
- Each character is given a unique number
  - 'a' is 0x41 (hex 41, decimal 65)
  - '\n' is 0x0A
- Text files are actually binary data
  - Use hex editor, e.g., hexyl, or other tool: od, hexdump

Whitespace and Comments

- Spaces are (nonprintable) characters too!
- Lexer must read each whitespace character as well
  - Spaces, newlines, tabs, etc
- Comments are whitespace
  - E.g., "// this is a comment"
  - Everything after the "//" is considered whitespace
- Need to know what characters comprise tokens
  - And what characters don't, i.e., those that end the token
- Whitespace ends (delimits) all tokens in our language

Demo: Inspecting Text Files

Lexical Specifications

- Our lexical specification
  - https://github.com/cop3402fall19/syllabus/blob/master/projects/lexical_specification.md
- Note that some punctuation is comprised of two characters
  - Can't compare single character to an entire string
  - Check one character at-a-time
- Note that there is special syntax for defining patterns

Recognizing Tokens

- How do we know what token is in our buffer?
- Keywords have a single lexeme
  - `strncmp`
- Punctuation also has a single lexeme
  - `ispunct`, check character equality
- Identifiers and numbers have many possible lexemes
  - Need to do pattern matching
- Keywords as special identifiers
  - Keywords have the same pattern as identifiers
  - Recognize an identifiers, then strncmp to determine if it's a keyword

Recognizing Patterns of Strings

- Numbers are any sequence of digits
  - The leading minus sign is optional
- Implementing this in pseudo-code:

```
clear buffer

if (c is a minus sign or c is a digit)
     // the number can optionally start with a minus sign
     add c to buffer
 while (c is a digit)
     add c to buffer
 // when we see anything other than a digit we know we are done
 return token  // make known constants for each token
```

- (Keep in mind: minus can be lead a number or be a subtraction)

Suggested Architecture #1: Token Stream

- The lexer is a single function `lex()`
- Input: **FILE***
  - Take a file and use fgetc
- Output: **struct token[]**
  - Return a list of tokens
  - Tokens are a struct that pair a token ID with its lexeme
- Token IDs are integer constants

```
#define PRINT   1
#define NUMBER  2
#define TIMES   3
// etc.
```

- The lexeme is a char*, i.e., the copied (strdup) contents of the buffer

Suggested Architecture #2: Token Functions

- Each token is a function
- Input: **FILE***
- Output: **char *** for the lexeme
  - NULL if the token was not matched
- For example:

```
char *identifier(FILE *file) {
    // if first character is not alpha, return NULL
    // otherwise buffer characters until non-alpha character
    // return buffer (strncpy if reusing the buffer)
}
```

Lookahead Character

- Lexing works by checking the *next* character
- The next character will eventually be a new token (or whitespace)
- Calling `fgetc()` moves to the next character in input
- For example: `3+-5`
  - How many tokens are there?
- + (plus) signals the end of number token 3
  - But we've already consumed the plus character!
- Solution: use `ungetc` or do your own input buffering

Lexing with Lookahead in Action

fgetc, see digit, must be NUMBER

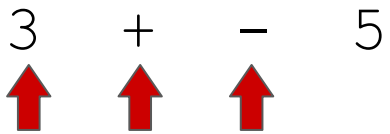
fgetc, see nondigit, NUMBER is over

recognized NUMBER 3, so ungetc

fgetc, see plus, must be PLUS token

fgetc, confirm end of PLUS token

recognized PLUS, so ungetc



Using manpages

- Documentation for library functions, e.g., fgetc(), isalpha()
- Just type, e.g., `man fgetc`

```
FGETC(3)                                          Linux Programmer's Manual
               FGETC(3)

NAME
       fgetc, fgets, getc, getchar, ungetc - input of characters and strings

SYNOPSIS
       #include <stdio.h>

       int fgetc(FILE *stream);

       char *fgets(char *s, int size, FILE *stream);

       int getc(FILE *stream);

       int getchar(void);

       int ungetc(int c, FILE *stream);

DESCRIPTION
       fgetc()  reads  the next character from stream and returns it as an unsigned char cast to an int,
or EOF on end of file or
       error.
```

UCF

Formalizing Patterns of Strings

Formal Definition of Languages

- An ***alphabet*** is a finite set of symbols
- A ***string*** is a *finite* sequence of symbols over an alphabet
- A ***language*** is a possibly infinite set of strings
- All languages can have the *empty string* ε (epsilon)

Regular Expressions Define String Patterns

- Concatenation **ab**
  - One string follows another
  - E.g., **ab** means the string must start with **a** and be followed by **b**
- Alternation **a|b**
  - One or the other string appears, but not both
  - E.g., **(a|bc)** means the string is either **a** or **bc** but not both
- Closure **a***
  - The string is repeated zero or more times
  - E.g., **(ab)*** means **ab** appears zero of more times
  - Zero times is equivalent to empty string Ɛ (epsilon)
- Use parentheses to enforce order of operations
  - Default order, from highest to lowest: closure, concatenation alternation
  - Analogous to exponentiation, multiplication, and addition, respectively

Regular Languages

- Any language defined by regular expressions is a *regular language*
- Regular languages are *closed* under regular expressions
  - Combing languages with regular expressions results in another regular language

OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

Figure 3.6: Definitions of operations on languages

UCF

Combine Expressions to Define a Language

- Note order of operations: **((a|(bc))*)d = (a|bc)*d**
- Example strings in this language
  - **d**, **ad**, **bcd**, **abcbcabcd**
- What is the regular expression for numbers?
  - Hint: use ε (epsilon)

(0|1|2|3|4|5|6|7|8|9**)(**0|1|2|3|4|5|6|7|8|9**)***

- What is the regular expression for identifiers?

- How about "every second character is a 0"?

Regular Expression Syntactic Sugar

- All regular languages can be described with three operations
  - Concatenation, alternation, closure
- **(a|ε)** is tedious to write
  - Many regex processors use **a?**
- Character ranges: **[0-9]** = **(0|1|2|3|4|5|6|7|8|9)**
  - Relies on ASCII character codes for digits being sequential
- Character classes: **[x2\n]** = **(x|2|\n)**
  - Can do negation as well: [^aL0] = all symbols in language except a, L, or 0
- Regular expression for identifiers:

  [A-Za-z][A-Za-z0-9]*

EXPRESSION	MATCHES	EXAMPLE
c	the one non-operator character c	`a`
$\backslash c$	character c literally	`\*`
`"`s`"`	string s literally	`"**"`
`.`	any character but newline	`a.*b`
`^`	beginning of a line	`^abc`
`$`	end of a line	`abc$`
$[s]$	any one of the characters in string s	`[abc]`
$[\hat{}\,s]$	any one character not in string s	`[^abc]`
$r*$	zero or more strings matching r	`a*`
$r+$	one or more strings matching r	`a+`
$r?$	zero or one r	`a?`
$r\{m, n\}$	between m and n occurrences of r	`a{1,5}`
$r_1 r_2$	an r_1 followed by an r_2	`ab`
$r_1 \mid r_2$	an r_1 or an r_2	`a|b`
(r)	same as r	`(a|b)`
r_1/r_2	r_1 when followed by r_2	`abc/123`

Figure 3.8: Lex regular expressions

Demo: `egrep`

Hand-Coding Lexers

- Regular expressions are equivalent to code
  - Concatenation is a sequence of statements
  - Alternation is an if statement
  - Closure is a while loop

```
// concatenation
c = fgetc(file);
assert('a' == c);
c = fgetc(file);
assert('b' == c)
```

```
// closure
c = fgetc(file);
while (...) {
  ungetc(c, file);
  // repeated pattern
}
```

```
// alternation
c = fgetc(file);
if (...) {
  ungetc(c, file);
  // first alternative
} else if (...) {
  ungetc(c, file);
  // second alternative
}
```

Demo: Lexing (ab|c)*

Conclusion

- The lexer groups characters into words
- It buffers characters to save the lexeme
- Regular expressions describe patterns of strings
- Regular expression can be hand-implemented in a lexer