COP-3402 Systems Software
09/12 Thu

Keep in mind that you will be writing a program that takes a source file and generates LLVM IR. The compiler acts like a mechanical programmer; it can read your source file and write a new version of that program in a different language.

The **git.md** tutorial under *syllabus/projects* was covered in the very first class of the semester. Make sure you gain those fundamentals about git and other environment setups and also how to submit projects.

Lexing:
A lexer groups characters into words.
Source files are text files.
Compilers use algorithms to recognize words.

char type
char c = 'a';
'\n' represents a new line character that is actually a single character
c = fgetc(file);
=> fgetc is a library function that takes a bite from the file and stores it into the memory that is referenced by the variable c.

Lexer reads and buffers characters:
Reads each character from a file
Buffer them into an array
Complete the word and go next
Each word is labeled with its name
=> 78 is a NUMBER

A **lexeme** is the actual string of characters that forms a single word/token in your language.
For the print; token is just the print keyword and the lexeme is p,r,i,n,t.
Lexer is able to account for pattern recognition to recognize a number.
The attribute is the value of a token.
The lexeme is the actual string of characters and the value of token is the actual number.
In order to represent symbols, we associate each one of those symbols with a particular number-ASCII.
ASCII is the encoding we will use for our compiler. We can represent 128 characters with ASCII.

command:
cd examples
hexyl helloworld.c
=> shows the ASCII codes for each of the text files.

Spaces are also characters, non-printable-20. We just ignore them in our projects.
Comments are whitespace:
everything after "//" is considered whitespace.

Keywords:

PRINT	“print”
INT	“int”
RETURN	“return”

Symbols:

SEMI	“;”
PLUS	“+”
MINUS	“-”
TIMES	“*”
DIVIDE	“/”
MOD	“%”

Whenever your lexer sees these characters, it should recognize them as being one of these tokens.

NUMBER	MINUS?	DIGIT	DIGIT*
IDENTIFIER	LETTER	(LETTER|DIGIT)*	

This is exactly a number looks like in our language: It is one or more digits and it can be proceeded by a minus sign.
This syntax is a pattern specification language called regular expressions.

Recognizing tokens:
Keywords have a single lexeme (ex: strncmp)
Punctuation has a single lexeme (ex: ispunct checks character equality)

Recognizing patterns of strings:
Numbers are any sequence of digits.
Keep in mind that minus can be lead a number or be a subtraction.
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

Suggested Architecture.1:
Treat the lexer (lex) as something that takes in a file and produces an array/stream of tokens:
Input: FILE*
○ Take a file and use fgetc
Output: struct token[]
○ Return a list of tokens
○ Tokens are a struct that pairs a token ID with its lexeme

```
#define PRINT 1
#define NUMBER 2
```

#define TIMES 3

Suggested Architecture.2:
Each token is a function:
Input: FILE*
Output: char * for the lexeme
○ NULL if the token was unmatched
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 processes by checking the next character
When we call fgetc(), it moves to the next character.
We can use *ungetc* to push the lookahead token back into the file. If the characters are not finished, just keep consuming tokens from the input.

Ex:
3 + - 5
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
...

=> fgetc actually reads in a buffer of characters. With ungetc you do not have to buffer it yourself and let the standard i/o library buffer it for you.

Using manpages:
man fgetc
=> gives the description of function prototype and the behavior.

Formal definition of languages:
An alphabet is a finite set of symbols, for our compiler, it is the set of symbols in our alphabet.
A string is a finite sequence of symbols over an alphabet
A language is a possibly infinite set of strings

Regular Expressions Describe String Patterns:
Concatenation ab: b must follow a
Alternation a|b: one or other character may appear
Closure a*: it can appear zero or more times, meaning infinite number of possible strings

Any language defined by regular expressions is a regular language.

Note order of operations: ((a|(bc))*)d = (a|bc)*d
Example strings in this language:
○ d, ad, bcd, abcbcabcd

We can define all regular languages with three operations: Concatenation, alternation, closure

Hand-coding lexers:

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

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

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

Ex:

```
//ab
c = fgetc(stdin);
assert('a'==c);
c = fgetc(stdin);
assert('b'==c);

//ab|D
c = fgetc(stdin);     //lookahead
if ('a' == c) {
    ungetc(c, stdin);
    c = fgetc(stdin);
    assert('a'==c);
    c = fgetc(stdin);
    assert('b'==c);
} else if ('D' == c) {
      ungetc(c, stdin);
      c = fgetc(stdin);
```

```
   assert(‘D’==c);
} else {
assert (0);
}


//(ab|D)*
c = fgetc(stdin);     //lookahead
While (‘a’ == c | ‘D’ == c){
ungetc(c, stdin);
c = fgetc(stdin);
if (‘a’ == c) {
    ungetc(c, stdin);
    c = fgetc(stdin);
    assert(‘a’==c);
    c = fgetc(stdin);
    assert(‘b’==c);
} else if (‘D’ == c) {
      ungetc(c, stdin);
      c = fgetc(stdin);
   assert(‘D’==c);
} else {
assert (0);
}
}
```