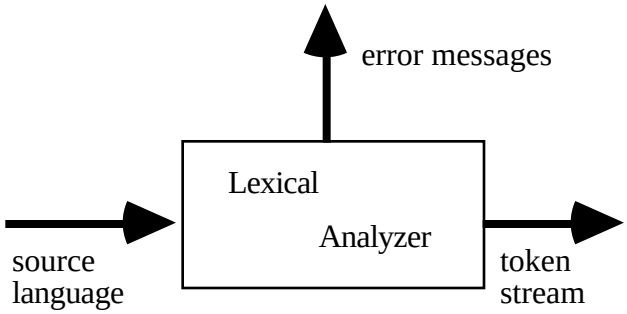
Notes on lexical analysis

Handout written by Maggie Johnson and revised by me.

The basics

Recall that *lexical analysis* or *scanning* is the process where the stream of characters making up the source program is read from left-to-right and grouped into tokens. *Tokens* are sequences of characters with a collective meaning. There are usually only a small number of tokens for a programming language: constants (integer, double, char, string, etc.), operators (arithmetic, relational, logical), punctuation, and reserved words.



The lexical analyzer takes as input a source program, and produces as output a stream of tokens that will serve as input to the syntax analyzer. The lexical analyzer might recognize particular instances of tokens such as:

> `3` or `255` for an integer constant token
> `"Fred"` or `"Wilma"` for a string constant token
> `numTickets` or `queue` for a variable token

Such specific instances are called *lexemes*. Think of a lexeme as the actual character sequence forming a token. The parser often needs access to both the token and the specific lexeme.

Note the scanner only is responsible for determining that the input stream can be divided into valid symbols in the source language. Few errors can be detected at the lexical level alone because the scanner has a very localized view of the source program without any context. The scanner can report about characters that are not valid tokens (e.g., an illegal or unrecognized symbol) and a few other malformed entities (illegal characters within a string constant, unterminated comments, etc.) It does not look for or detect garbled sequences, tokens out of place, undeclared identifiers, misspelled keywords, mismatched types and the like. For example, the following input will not generate any errors in the lexical analysis phase, because the scanner has no concept of the appropriate arrangement of tokens for a declaration. The syntax analyzer will catch this error later in the next phase.

```
int a for return double b[2];
```

Furthermore, the scanner has no idea how tokens should be grouped. In the above sequence, it returns `b`, `[`, `2`, and `]` as four separate tokens, having no idea that the collectively form an array declaration.

The lexical analyzer is often a convenient place to carry out some other chores like stripping out comments and white space between tokens and perhaps even some features like macros and conditional compilation (although if complex, this is usually handled by some sort of preprocessor).

Symbol table

Let's consider the data structures needed to support the scanning phase. The most important one is *symbol table*. The symbol table is used throughout the compiling process to build up information about identifiers used in the source program. At the lexical analysis stage, it may hold no more information than the text of the variable's name. However, during syntax and semantic analysis, information about the variable's type and scope will be added. Later in code generation, we will also store the address for the identifier. A simple structure for an entry in a symbol table might look like this:

```
struct entry
{
   char *name;          // the user-defined name of the identifier
   type_t type;         // its type
   scope_t scope;       // scoping information
   int location;        // address assigned by code generator
};
```

These entry structures are often called *declarations* because they are usually first entered into the symbol table when the variable or function is declared in the source code. As we progress through the additional stages of compiling, we add more information for each identifier and use that information to detect type or access errors and later generate the appropriate instructions for that type during the code generation stage. The identifier name serves as the key to look up the information stored as its associated value in the table.

Symbol tables are nearly always implemented with hashtables. What might be important about the hash function used for a symbol table? (Think about the properties of identifier names...)

Scanner implementation #1: loop & switch

There are two primary methods for implementing a scanner. The first is a program that is hard-coded to perform the scanning tasks. The second is the use of regular expressions and finite automata to model the scanning process.

A "loop & switch" implementation consists of a main loop that reads characters one by one from the input file and uses a switch statement to process the character(s) just read. The output is a list of tokens and lexemes from the source program. The following program fragment shows a skeletal implementation of a simple loop and switch scanner. The main program calls `InitScanner()` and loops calling `ScanOneToken()` until EOF. `ScanOneToken()` reads the next character from the file and switches off that char to decide how to handle what is coming up next in the file. The return values from the scanner can be passed on to the parser in the next phase.

```
/* Each different token type has its own unique code */

#define T_SEMICOLON   ';'     // use ASCII values for single char tokens
#define T_COMMA       ','
#define T_LPAREN      '('
#define T_RPAREN      ')'
#define T_ASSIGN      '='
#define T_DIVIDE      '/'
 ...

#define T_WHILE       257     // reserved words
```

```
#define T_PRINT       258
#define T_RETURN      259
#define T_IF          260
 ...

#define T_IDENTIFIER  268     // identifiers, constants, etc.
#define T_INTEGER     269
#define T_DOUBLE      270
#define T_STRING      271

#define T_END         349     // code used when at end of file
#define T_UNKNOWN     350     // token was unrecognized by scanner

struct token {
   int type;                  // one of the token codes from above
   union {
      char stringValue[256];  // holds lexeme value if string/identifier
      int intValue;                 // holds lexeme value if integer
      double doubleValue;           // holds lexeme value if double
   } val;
};


int main(int argc, char *argv[])
{
   struct token token;

   InitScanner();
   while (ScanOneToken(stdin, &token) != END)
      ; // process token
   return 0;
}

static void InitScanner()
{
   create_reserved_table();   // table maps reserved words to token type
   insert_reserved("WHILE", T_WHILE)
   insert_reserved("PRINT", T_PRINT)
   insert_reserved("RETURN", T_RETURN)
   ....
}

static int ScanOneToken(FILE *fp, struct token *token)
{
   int i, ch, nextch;

   ch = getc(fp);       // read next char from input stream
   while (isspace(ch))
      ch = getc(fp);    // discard all white space

   switch(ch)
   {
      case '/':               // check for a comment or T_DIVIDE operator
         nextch = getc(fp);
         if (nextch == '/' || nextch == '*')
            ; /* code to skip over a comment */
         else
            ungetc(ch, fp); // fall-through to case for single-char token
```

```
            case ';': case ',': case '=':   // ... and other single char tokens
               token->type = ch; // ASCII value is used as token type
               return ch;

            case 'A': case 'B': case 'C':       // ... and other upper letters
               token->val.stringValue[0] = ch;
               for (i = 1; isupper(ch = getc(fp)); i++)
                  token->val.stringValue[i] = ch;
               ungetc(ch, fp);
               token->val.stringValue[i] = '\0';
               token->type = lookup_reserved(token->val.stringValue);
               return token->type;

            case 'a': case 'b': case 'c': // ... and other lower letters
               token->type = T_IDENTIFIER;
               token->val.stringValue[0] = ch;
               for (i = 1; islower(ch = getc(fp)); i++)
                  token->val.stringValue[i] = ch;
               ungetc(ch, fp);
               token->val.stringValue[i] = '\0';
               if (lookup_symtab(token->val.stringValue) == NULL)
                  add_symtab(token->val.stringValue);
               return T_IDENTIFIER;

            case '0': case '1': case '2': case '3':  //.... and other digits
               token->type = T_INTEGER;
               token->val.intValue = ch - '0';
               while (isdigit(ch = getc(fp)))   // convert digit char to number
                  token->val.intValue = token->val.intValue * 10 + ch - '0';
               ungetc(ch, fp);
               return T_INTEGER;

            case EOF:
               return T_END;

            default:          // anything else is not recognized
               token->val.intValue = ch;
               token->type = T_UNKNOWN;
               return T_UNKNOWN;
         }
      }
```

The mythical source language tokenized by the above scanner requires that reserved words must be in all upper-case and identifiers in all lower-case. This convenient feature makes it easy for the scanner to choose which path to pursue after reading just one character. It is sometimes necessary to design the scanner to “look-ahead” before deciding what path to follow— notice the handling for the '/' character which pulls the next character to check for another slash or star (which indicates the beginning of a comment), otherwise the extra character is pushed back onto the input stream and the token is interpreted as the single char operator for division.

Loop-and-switch scanners are sometimes called *ad hoc* scanners, indicating their design and purpose of solving a specific instance rather a general problem. For a sufficiently reasonable set of token types, a hand-coded loop-and-switch scanner might be all that’s needed— they require no other tools and can be coded rather efficiently. The `gcc` front-end uses an ad hoc scanner, in fact.

Scanner implementation #2: regular expressions & finite automata

The other method of implementing a scanner is using regular expressions and finite automata. A quick detour for some background review and then let's see how we can generate a scanner drawing upon techniques from automata theory.

Regular expression review

We assume that you are well acquainted with regular expressions and all this is old news to you.

symbol	an abstract entity that we shall not define formally (such as "point" in geometry). Letters, digits and punctuation are examples of symbols.
alphabet	a finite set of symbols out of which we build larger structures. An alphabet is typically denoted using the Greek sigma ∑, e.g., ∑ = `{0,1}`.
string	a finite sequence of symbols from a particular alphabet juxtaposed. For example: `a`, `b`, `c`, are symbols and `abcb` is a string.
empty string	denoted ∂, is the string consisting of zero symbols.
formal language ∑*	the set of all possible strings that can be generated from a given alphabet.
regular expressions	rules that define exactly the set of words that are valid tokens in a formal language. The rules are built up from three operators:

concatenation	`xy`	
alternation	`x|y`	x or y
repetition (0)	`x*`	x repeated 0 or more times
repetition (1)	`x+`	x repeated 1 or more times

Formally, the set of regular expressions can be defined by the following recursive rules:

1) Every symbol of ∑ is a regular expression
2) ∂ is a regular expression
3) if $\mathbf{r_1}$ and $\mathbf{r_2}$ are regular expressions, so are
   $(\mathbf{r_1})$ $\mathbf{r_1 r_2}$ $\mathbf{r_1} \mid \mathbf{r_2}$ $\mathbf{r_1}^*$
4) Nothing else is a regular expression.

Here are a few practice exercises to get you thinking about regular expressions again. Give regular expressions for the following languages over the alphabet `{a,b}`

all strings beginning and ending in *a* _____
all strings with an odd number of *a's* _____
all strings without two consecutive *a's*_____

We can use regular expressions to define the tokens in a programming language. For example, here is a regular expression for an integer, which consists of one or more digits:

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

Here is one for an identifier that must begin with a letter and can be followed by any number of letters or digits:

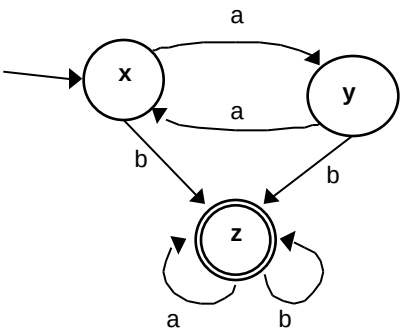
```
(a|b|c|...)(a|b|c|...|1|2|3|...)*
```

Finite automata review

Once we have all our tokens defined using regular expressions, we can create a finite automaton for recognizing them. This can be done by hand, or we can use an automatic scanner generator like `lex` to generate the finite automata.

To review, a *finite automata* has:

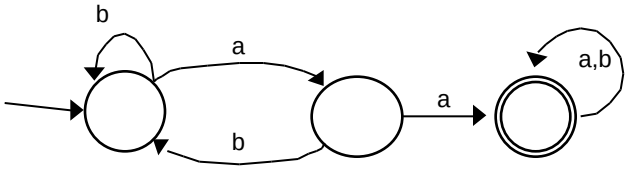
1) A finite set of states, one of which is designated the initial state or *start state*, and some (maybe none) of which are designated as final states**.**
2) An alphabet ∑ of possible input symbols.
3) A finite set of transitions that specifies for each state and for each symbol of the input alphabet, which state to go to next.



	a	b
(start) x:	y	z
y:	x	z
(final) z:	z	z

What is a regular expression for the FA above? _____

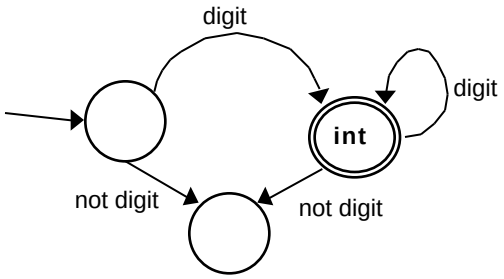
What is a regular expression for the FA below? _____



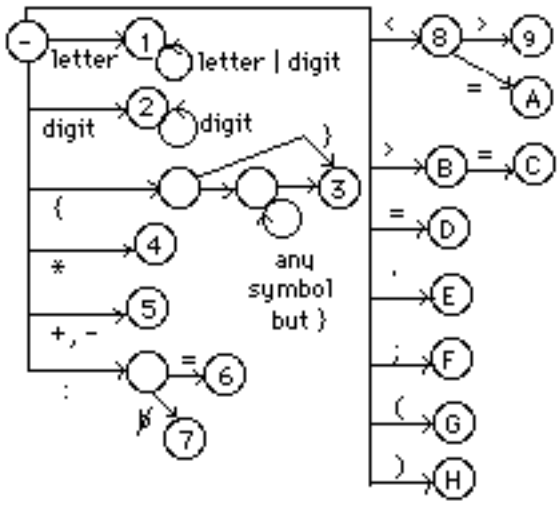
Define an FA that accepts the language of all strings that end in b and do not contain the substring aa. What is a regular expression for this language?

Now that we remember what FAs are, here is a regular expression and a simple finite automata that recognizes an integer.

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



Here is an FA that recognizes a subset of tokens in the Pascal language:



This FA is incomplete in that we are not handling "/", div, mod, real numbers, (* ... *), etc. But this should give you an idea of how an FA can be used to drive a scanner. The numbered/lettered states are final states. The loops on states 1 and 2 continue to execute until a character other than a letter or digit is read. For example, when scanning "`temp:=temp+1;`" it would report the first token at final state 1 after reading the "`:`" having recognized the lexeme "`temp`" as an identifier token.

What happens in an FA-driven scanner is we read the source program one character at a time beginning with the start state. As we read each character, we move from our current state to the next by following the appropriate transition for that. When we end up in a final state, we perform an action associated with that final state. For example, the action associated with state 1 is to first check if the token is a reserved word by looking it up in the reserved word list. If it is, the reserved word is passed to the token stream being generated as output. If it is not a reserved word, it is an identifier so a procedure is called to check if the name is in the symbol table. If it is not there, it is inserted into the table.

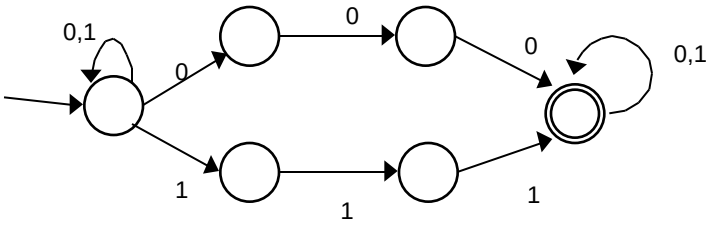
Once a final state is reached and the associated action is performed, we pick up where we left off at the first character of the next token and begin again at the start state. If we do not end in a final state or encounter an unexpected symbol while in any state, we have an error condition. For example, if you run "ASC@I" through the above FA, we would error out of state 1.

From regular expressions to NFA

So that's how FAs can be used to implement scanners. Now we need to look at how to create an FA given the regular expressions for our tokens. There is a looser definition of an FA that is especially useful to us in this process. A *nondeterministic finite automaton (NFA)* has:

1) A finite set of states with one start state and some (maybe none) final states.
2) An alphabet $\sum$ of possible input symbols.
3) A finite set of transitions that describe how to proceed from one state to another along edges labeled with symbols from the alphabet (but not ∂). We allow the possibility of more than one edge with the same label from any state, and some states for which certain input letters have no edge.

Here is an NFA that accepts the language `(0|1)*(000|111)(0|1)*`



Notice that there is more than one path through the machine for a given string. For example, `000` can take you to a final state, or it can leave you in the start state. This is where the non-determinism (choice) comes in.

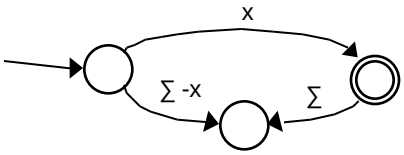
There is a third type of finite automata called NFA-∂ which have transitions labeled with the empty string. The interpretation for such transitions is one can travel over an empty-string transition without using any input symbols.

A famous proof in formal language theory (Kleene's Theorem) shows that FAs are equivalent to NFAs which are equivalent to NFA-∂s. And, all these types of FAs are equivalent in language-generating power to regular expressions. In other words,

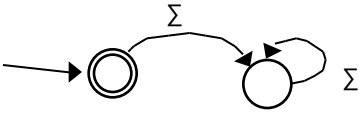
> If R is a regular expression, and L is the language corresponding to R, then there is an FA that recognizes L. Conversely, if M is an FA recognizing a language L, there is a regular expression R corresponding to L.

The reason why NFAs are helpful is it is typically quite easy to take a regular expression and convert it to an NFA or NFA-∂, thanks to the simple rules of Thompson's construction:

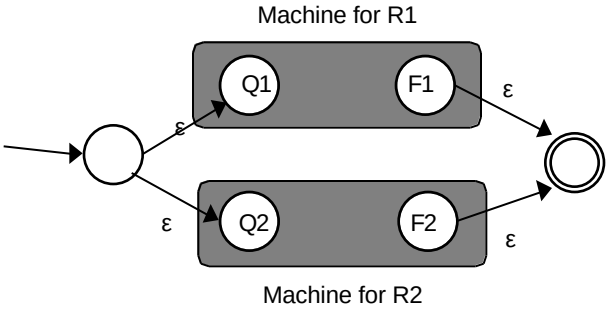
Rule 1: There is an NFA that accepts any particular symbol of the alphabet:



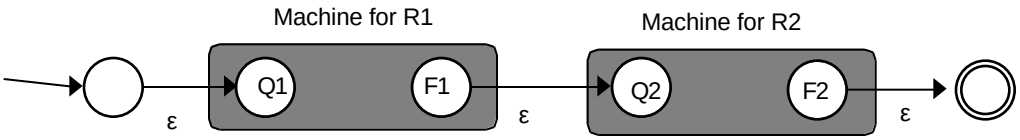
Rule 2: There is an NFA that accepts only ∂:



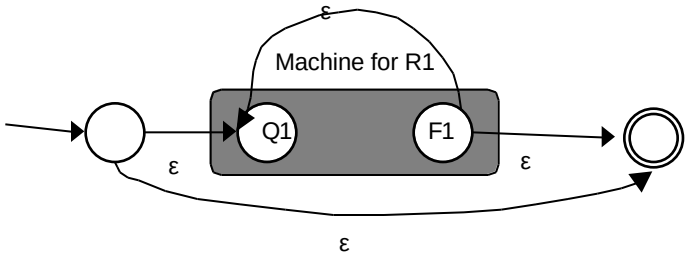
Rule 3: There is an NFA-∂ that accepts `r1|r2`:



Rule 4: There is an NFA-∂ that accepts `r1r2`:



Rule 5: There is an NFA-∂ that accepts `r1*`:

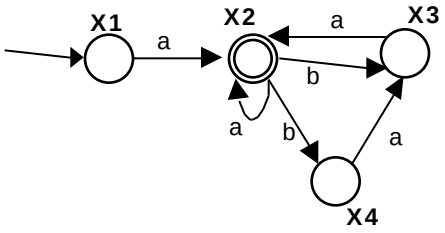


From NFA to FA via subset construction

Once we have our NFAs defined from the regular expressions, we use *subset construction* to convert the NFAs to FAs. Subset construction is an algorithm for constructing the deterministic FA that recognizes the same language as the original nondeterministic FA. Each state in the new DFA is made up of a set of states from the original NFA. The start state of the DFA will be the start state of NFA. The alphabet for both automata is the same.

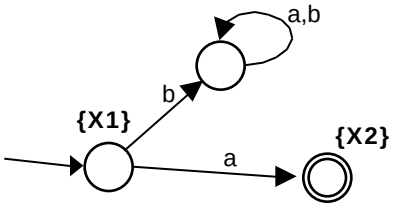
So, given a state of from the original NFA, an input symbol x takes us from this state to the union of original states that we can get to on that symbol x. We then have to analyze this new state with its definition of the original states, for each possible input symbol, building a new state in the DFA. The states of the DFA are all subsets of S, which is the set of original sets. There will be a max of 2^n of these (because we might need to explore the entire power set), but there are usually far fewer. The final states of the DFA are those sets that contain a final state of the original NFA.

Here is an example:

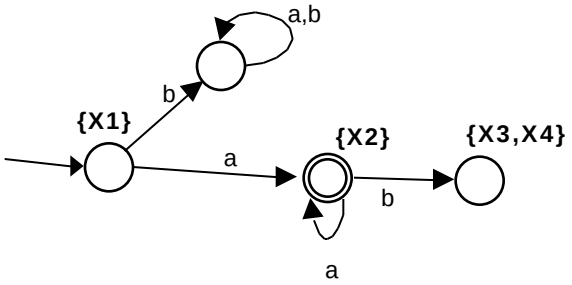


This is non-deterministic for several reasons. For example, the two transitions on 'b' coming out of the final state, and no 'b' transition coming out of the start state. To create an equivalent deterministic FA, we begin by creating a start state, and analyzing where we go from the start state in the NFA, on all the symbols of the alphabet. We create a set of states where applicable (or a sink hole if there were no such transitions). Notice if a final state is in the set of states, then that state in the DFA becomes a final state.

(after creating start state and filling in its transitions)

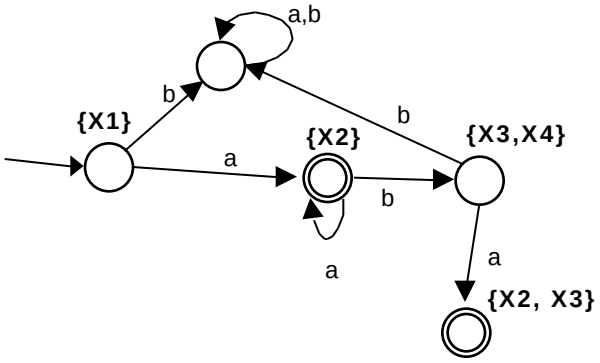


(after filling in transitions from {X2} state)

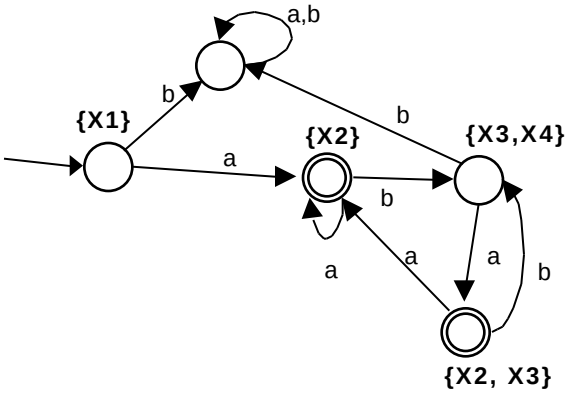


We continue with this process analyzing all the new states that we create. We need to determine where we go in the NFA from each state, on all the symbols of the alphabet.

(after filling in transitions from {X3, X4} state)



And finally, filling in the transitions from {X2, X3} state brings us full circle. This is now a deterministic FA that accepts the same language as the original NFA. We have 5 states instead of original 4, a rather modest increase all in all.



The process then goes like this: from a regular expression for a token, we construct an NFA that recognizes them using Thompson's algorithm. NFAs are not real useful as drivers for programs because non-determinism implies choices. So, we use subset construction to convert that NFA to a DFA. Once we have the DFA, we can use it as the basis for a scanner.

Lex basics

The reason we have spent so much time looking at how to go from regular expressions to finite automata is because this is exactly the process that lex goes through in creating a scanner. Lex is a lexical analysis generator that takes as input a series of regular expressions and builds a finite automaton and a driver program for it in C. We'll tell you a little bit about lex now, and you can find out a lot more from the links on the "Other materials" section of the course web page. You'll need to get pretty cozy with lex in order to complete pp1.

A lex input file mostly consists of a listing of rules. A rule has a regular expression (called the *pattern*) and an associated set of C statements (called the *action*). The idea is that whenever lex finds an input sequence that matches the pattern, it executes that action to process it. Running lex on such an input file uses process described earlier to build a FA that recognizes those regular expressions and then hooks up the actions to be performed when it hits the various accept states while processing the input.

Here are some of the operators used in lex regular expressions. Notice it goes far beyond our formal language definition of a regular expression.

Character classes	`[0-9]`	This means alternation of the characters in the range listed (in this case: `0|1|2|3|4|5|6|7|8|9`). More than one range may be specified, e.g. `[0-9A-Za-z]` as well as specifying individual characters `[aeiou0-9]`.
Character exclusion	`^`	The first character in a character class may be `^` to indicate the complement of the set of characters specified. For example, `[^0-9]` matches any non-digit character.
Arbitrary character	`.`	The period matches any single character except newline.
Single repetition	`x?`	This means 0 or 1 occurrence of `x`.
Non-zero repetition	`x+`	This means `x` repeated one or more times; equivalent to `xx*`.
Specified repetition	`x{n,m}`	`x` repeated between `n` and `m` times.
Beginning of line	`^x`	Match `x` at beginning of line only
End of line	`x$`	Match `x` at end of line only.
Context-sensitivity	`ab/cd`	Match `ab` but only when followed by `cd`. The lookahead characters are left in the input stream to be read for the next token.
Literal strings	`"x"`	This means `x` even if `x` would normally have special meaning. Thus, `"x*"` may be used to match x followed by an asterisk.
Definitions	`{varname}`	Substitutions may be defined. This means replace with the predefined pattern called `varname`.

Example #1
Here is a simple and complete lex program that capitalizes all lowercase letters and echo all others unchanged:

```
%%
[a-z]     putchar(toupper(yytext[0]));
.         ECHO;
```

The first `%%` marks the beginning of the rules section, the only section required in a lex file. The pattern for the first rule matches any single lowercase letter and the associated action prints the letter after uppercasing it. The second rule matches any remaining character and uses the lex action `ECHO` (which just prints the character unchanged).

Lex input files are traditionally named using a .l extension. You invoke lex on a .l file and it creates lex.yy.c which is a generated C source file containing a FA encoding all the rules from the input file. You compile that C file normally, link with the lex library, and you have built a scanner! The scanner reads from stdin and writes to stdout by default.

```
% lex uppercase.l
% gcc -o uppercase lex.yy.c –ll
% uppercase
... at this point anything you type, the scanner echos in UPPPERCASE
```

Example #2
The following lex program has all three sections: a definitions section (when you can define substitutions, set up global variables, etc.), the rules section, and the user subroutines section (where you can define helper functions). What does this program do?

```
%{
   int numchar = 0, numword = 0, numline = 0;
%}

%%

\n          {numline++; numchar++;}
[^ \t\n]+   {numword++; numchar+= yyleng;}
.           {numchar++;}

%%

main()
{
   yylex();
   printf("%d\t%d\t%d\n", numchar, numword, numline);
}
```

To build and run this program:

```
% lex count.l
% gcc -o count lex.yy.c –ll
% count <count.l
% 216   32   17
```

Programming Language Case Study: FORTRAN I

The first version of FORTRAN is interesting for us for a couple reasons. First of all, the language itself violates just about every principle of good design that we specified in a previous handout. Secondly, the process of developing the first FORTRAN compiler laid the foundation for the development of modern compilers.

Here is a brief description of FORTRAN I:

- Variables can be up to five characters long and must begin with a letter. Variables beginning with i, j, k, l, m and n are assumed by the compiler to be integer types, all others are reals. (This restriction may very well be the reason why programmers continue to use i as the name of the integer loop counter variable…)
- Variable declarations are not required, except for arrays. Arrays are limited to 3 dimensions.

- Computations are done with the assignment statement where the left side is a variable and the right-side is an equation of un-mixed types.
- The control structures consisted of:
  - unconditional branch: `GOTO statement_number`
  - conditional branch: `GOTO (30, 40, 50) I1` where `I1` has a value of 1, 2 or 3 to indicate which position in the list to jump to.
  - IF statement: `IF (A+B-1.2) 7, 6, 9` transfers control to statement labeled 7, 6, or 9 depending on whether `(A+B-1.2)` is negative, zero or positive.
  - DO statement: `DO 17 I = 1, N, 2` which specifies that the set of statements following, up to and including that labeled 17, is to be executed with I first assigned value 1, incrementing by 2, up through N.
- No user-defined subroutines allowed, although several built-ins were provided including `READ` and `WRITE` for I/O.
- Blanks are ignored in all statements, e.g., all the following are equivalent:

```
DIMENSION IN DATA(10000)
DIMENSIONINDATA(10000)
D I M E N S I O N I N D A T A ( 1 0 0 0 0 )
```

- Reserved words are valid variable names, e.g., `DIMENSION IF(100)` declares an array called `IF`.

The task of building the first compiler was a huge one compared to what we face today. First of all, we can see there were considerable challenges in the definition of the language itself. Allowing variables to have the same name as reserved words makes the scanning process much more difficult (although they didn't really do "scanning" in the first compiler). Not requiring variable declarations meant the compiler had to remember lots of rules, and try to determine usage from context. In addition, these early compiler-writers had no formalisms to help as we do today, e.g., regular expressions and context-free grammars. Everything had to be designed and written from scratch. These formal techniques and the automatic tools based on them, allow us to build a fairly involved compiler over the course of 10 weeks, which is a far cry from 18 person-years.

The first FORTRAN compiler itself started as three phase but by the end, it had expanded to six. The first phase read the entire source program, filing all relevant information into tables, and translating the arithmetic expressions into IBM 704 machine code. The second phase analyzed the entire structure of the program in order to generate optimal code for the DO statements and array references. Phase 3 was the code generator as we know it. After a certain amount of work on phase 2, additional phase were added (3, 4, 5 with 6 becoming the code generator) to do further optimizations. Recall that optimization was key to the acceptance of the compiler since early FORTRAN programmers were skeptical of a machine creating truly optimal assembly code.

Comparing this structure to our "modern" compiler structure, we see that it does not map too well. In the original FORTRAN compiler, expressions are scanned, parsed and code-generated in the first and immensely complicated kitchen-sink of a first phase. One could view almost all of their inner four phases as just one great big optimization phase. So this compiler looks very different from what we know. Still, the same tasks were performed although in some cases less efficiently than what we can do today. The most important achievement, besides creating the first working compiler (which was a considerable achievement), was the foundations laid in optimization techniques. We will see that many of these techniques are still used in modern compilers today.

Bibliography

A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools. Reading, MA: Addison-Wesley, 1986.

J. Backus, “The History of FORTRAN I, II, III.”, SIGPLAN Notices, Vol. 13, no. 8, August, 1978, pp. 165-180.

J.P. Bennett, Introduction to Compiling Techniques. Berkshire, England: McGraw-Hill, 1990.

D. Cohen, Introduction to Computer Theory, New York: Wiley, 1986.

C. Fischer, R. LeBlanc, Crafting a Compiler. Menlo Park, CA: Benjamin/Cummings, 1988.

J. Hopcroft, J. Ullman, Introduction to Automata Theory, Languages, and Computation, Reading MA: Addison-Wesley, 1979.

S. Kleene, "Representation of Events in Nerve Nets and Finite Automata," in C. Shannon and J. McCarthy (eds), Automata Studies, Princeton, NJ: Princeton University Press, 1956.

A. McGettrick, The Definition of Programming Languages. Cambridge: Cambridge University Press, 1980.

T. Sudkamp, Languages and Machines: An Introduction to the Theory of Computer Science, Reading, MA: Addison-Wesley, 1988.

R.L.Wexelblat, History of Programming Languages. London: Academic Press, 1981.