

Compilers: An Introduction

A compiler is a tool for translating one computer programming language to another. Most often, a compiler is thought of as translating a high-level language such as C to an executable low-level language (i.e. machine language). However, some compilers produce a target other than machine language. For example, the first C++ compilers translated to C, which in turn could be compiled by pre-existing C compilers. Java is typically compiled into a machine-independent intermediate representation called "byte code."

Whereas a compiler produces an output which is intended for later execution, an **interpreter** accepts a high-level language as input and then acts immediately to execute the program. Awk, Perl, Python, UNIX shell script, Javascript and BASIC are examples of languages which are typically interpreted.

If we need to define a distinction between compilers and interpreters, we can in this fashion: A compiler is a tool which operates in one time instance, known as **compile-time**, to translate the source code into a target code. This target will then be run at a later time, known as **run-time**. In a pure compiled language, there is no opportunity to modify or introduce new code into the running program, because the compiler is not part of the run-time environment. We also note that the compiler generates its target code with a presumption of what the run-time environment will look like, and what supporting code (e.g. shared libraries) will be present at run-time. If this presumption is incorrect, e.g. you take code compiled for a specific target and try to run it on an incompatible target, the result is failure to execute or erroneous execution.

In contrast, the interpreter exists in only one time. The source code is executed as it is seen. Because the interpreter engine is part of the execution environment, interpreted languages almost always have the ability to modify the running program. We can accept input as the program is running and interpret it, resulting in the input being executable. This mechanism is often called something like `eval`. This same mechanism can also be a computer security risk.

The differentiation between compiler and interpreter can be a gray area, and many of the same principles and techniques apply to both. Many clearly interpreted languages, such as Python, resort to a hybrid approach in which source code files are pre-compiled to an intermediate bytecode (e.g. `.py -> .pyc`). This addresses one potential problem with pure interpreters in that they can't detect errors until they reach that part of the program containing the error. It also improves the latency between interpreter invocation and the start of actual execution.

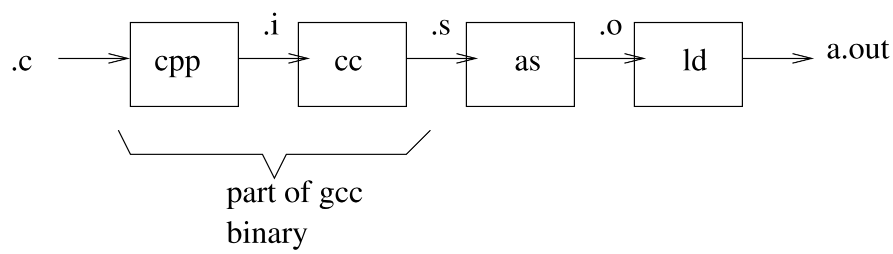
The Compiler Tool-chain

Strictly speaking, the compiler is just one component of a system for producing executable program files. The compiler translates the source language (e.g. C) into `assembly language`, which is a linear, textual, symbolic encoding of machine language. The **assembler** translates the assembly language into machine language. Compared to a compiler, an assembler is a fairly simple piece of code. It merely needs to recognize the symbolic opcodes and addressing modes, and generate the correct binary opcodes and bit fields.

Note, however, that the output of the assembler is still not an executable program. The assembler creates a **relocatable object file** in which those bytes or bit fields in the binary machine language which need to refer to specific memory addresses are left undefined. The object file also contains a **symbol table** containing globally visible symbolic names.

The **linker** pulls together all of the relocatable object files comprising the program, including object files which are part of the system libraries, resolves all symbol references, assigns absolute addresses to all symbols, and pokes the undefined locations with the now-defined values. The result is an absolute, executable file, which on UNIX systems is by default called `a.out`.

This multi-stage process is normally hidden from the casual user. When one invokes the `cc` command in UNIX, it takes care of invoking the compiler, assembler and linker. The UNIX utility `make` is also useful for coordinating the build of complicated executables, especially when the source tree is large and it is desirable to do the minimum amount of work necessary for each incremental design change. The following depicts the typical UNIX/gcc toolchain:



Inside the Compiler

The compiler is given a program in the source language and is charged with figuring out three basic things about it:

- What does it *say*?
- What does it *mean*?
- How can it be implemented in the target language?

We can make a dangerous analogy to the human brain's understanding of language. We hear or read the sentence "Pick that up and put it away". The first phase of understanding is to group the phonemes (hearing) or characters (reading) into words (tokens). Next comes "parsing" the sentence structure. In this example, the syntax is harder than what would typically be found in a computer language. The word "pick" could be either a noun or a verb. By iterative application of syntax rules, we understand that this phrase contains two subject/object/verb structures joined by the conjunction "and". Now that we understand what the sentence says, we have to figure out what it means. Here the context of what came earlier in the conversation is essential. Without that, there is no meaning. To what object does "that" refer? It must be something that was mentioned earlier. Finally, we need to figure out how to implement this directive. This includes knowing how to pick the object up, which might vary depending on the size and shape, and how to put it away, which seemingly requires knowledge of where it should go.

Because natural (human) language is very ambiguous, the techniques used to process it are very different from those used to process computer languages. In particular, computer languages were created by humans specifically to be easy for computers to understand and implement. For this reason, analogies between the two are best avoided.

Front End / Back End structure

Often, in discussing the architecture of a compiler, we speak of it having a **front-end** and a **back-end**. The front-end answers the first two questions above, i.e. it recognizes the source language in terms of the defined language constructs and determines the meaning based on the rules of the language. The back-end emits target language instructions which will accomplish what the source program says and means.

The Front End

For the sake of discussion, the functions of the front-end are usually divided into three parts or phases:
• Lexical analysis, or "scanning"
• Syntactical analysis, or "parsing"
• Semantic analysis.

We shall see that these are, in a sense, one-, two- and three-dimensional ways of looking at the source program.

Briefly, **lexical analysis** is a linear process of examining the characters of the source code one character at a time, and grouping them into fundamental language units called **tokens** or **lexemes**. The rules which describe how characters may be grouped into these tokens can be specified using **Regular Expression** notation.

Syntactical analysis looks at the input as a series of tokens, and discovers how the tokens are related in terms of higher-level constructs such as if-then-else statements, declarations, etc. This is a recursive process which (conceptually) places the tokens into a tree representing the grammatical structure of the input program. The syntax rules are expressed as a **Grammar**.

Both lexical and syntactical analysis are fairly mechanical. There are straightforward algorithms for constructing programs or subroutines which perform this analysis. In this course, we will make extensive use of two programs: **lex/flex**, which builds a lexical analyzer subroutine from a set of regular expressions, and **yacc/bison**, which builds a syntactical analyzer subroutine from a set of grammar rules.

After lexical and syntactical analysis, the compiler understands what the programmer **said**.

The third phase, **semantic analysis** discovers the meaning or intent of the syntactical constructs. In this phase, higher-level language concepts such as type checking, identifier scope, etc. are applied. Unfortunately, the third phase of the compiler front-end is not as easy to specify and implement mechanically. There is no *practical* system of specifying these rules nor an automated way of building a semantic analyzer, thus the code to implement semantic analysis must be developed by hand on a case-by-case basis. The structure of the semantic analysis phase will vary depending on the source language rules, but generally includes such major components as: the type system, the symbol table, the IR generator.

Although conceptually these three phases of the front end could be "pure" and isolated from each other, with the output of one phase becoming the input to the next, in actual compilers all three phases work together as co-routines. This results in a more efficient compiler.

At the conclusion of lexical, syntactical and semantic analysis, the compiler "understands" what the programmer wants to do. The job of the front end is then complete, and its product is an internalized **intermediate representation** (IR) of the program. The structure of this varies greatly from compiler to compiler. It could be a tree-like data structure, or a linear array of abstract pseudo-code, or some hybrid thereof. The IR is not the target language and is generally independent of any particular target architecture. During the compilation process, there may be several different forms of IR that are used and transformed.

The Back End

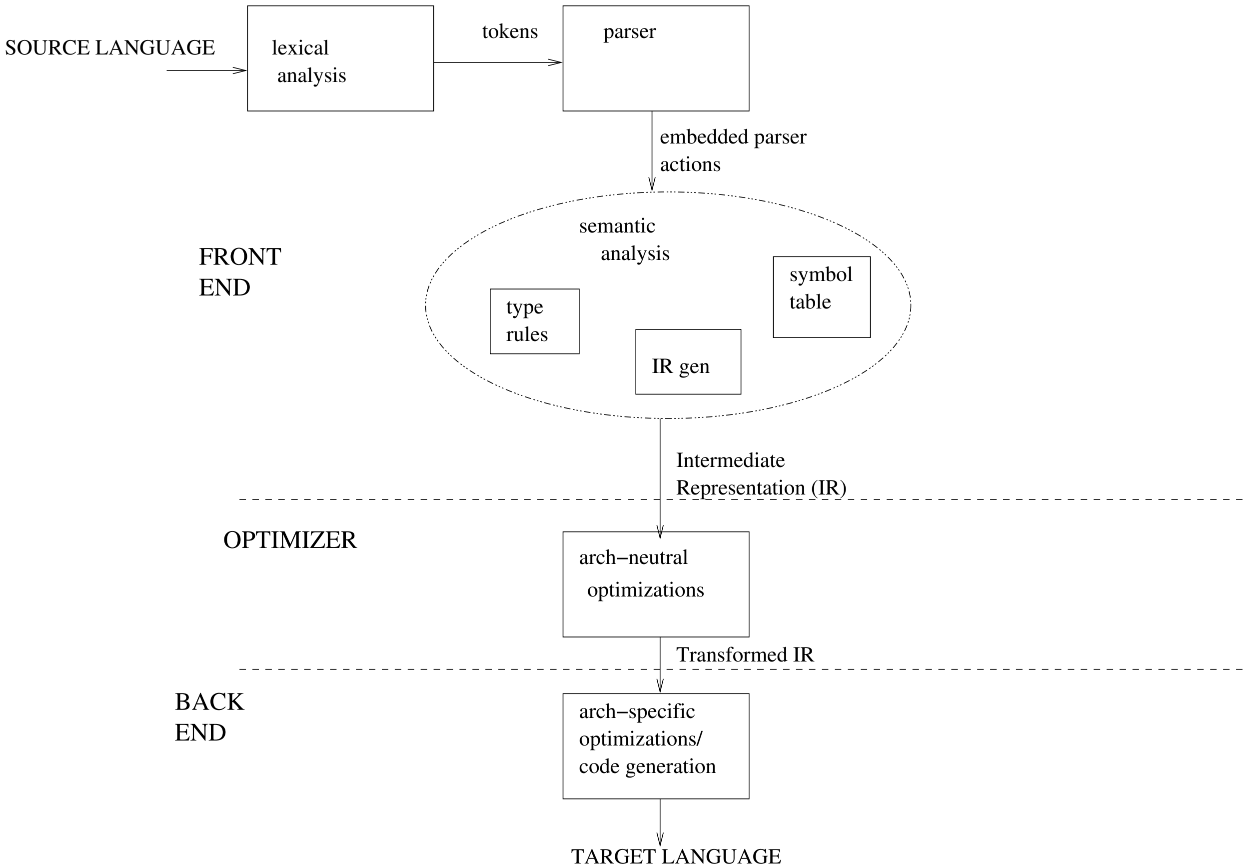
It is the job of the **back-end** of the compiler to translate the intermediate representation into the target language (e.g. assembly code). The back end must take into account the unique features of the target. When the target is assembly language, the back end must keep track of which registers to use, which instructions to select to implement the intent of the IR, the subroutine calling convention in effect, etc. The goal of the back end is to produce not only a target which correctly implements the IR, but one which implements it in an optimal fashion.

Note that the back end and front end can be cross-pluggable. The back end does not require any knowledge of the source language in order to do its job, as it works from the intermediate representation. Likewise, the front end needs to understand the source language, but does not need to know about the target language. As a result, multiple front ends (e.g. C, C++ and FORTRAN) can utilize the same back end, and a given front end can be attached to various back ends to create executable code for different targets. When this is done such that the compiler creates code for an architecture other than the one on which the compiler is running, that is called a **cross-compiler**.

Optimization

There are potentially infinite sequences of target language instructions which correctly implement the source language. Optimization is the process of arriving at a target which is "optimal" either in terms of speed of execution, memory consumption, or some other metric. Optimization is the most difficult part of compiler construction, and a detailed study of it is almost always left to a follow-up course in compiler theory.

We can divide optimization into two types: those which are independent of the target, and those which are target-specific. The latter optimizations are considered to be part of the back-end. On the other hand, architecture-neutral optimizations can be performed on the IR, thus adding another phase to the compiler which sits between the front end and the back end. Examples of architecture-neutral optimizations include eliminating code which can never be reached and avoiding computation of expressions whose values can be determined at compile time.



Lexical Analysis -- Overview

The source program is a sequence of characters which are part of the set of allowable source code characters. Different languages accept different character sets. For example, ANSI C allows only a subset of 7-bit ASCII characters. The job of lexical analysis (this act is often called "scanning" or "lexing") is to examine this continuous stream of characters and break it into groups of characters called **tokens**.

As a tangible example, consider the following fragment of C code:

```
a  =b+c-012;
```

The lexer, following the ANSI C language specification, sees this as the following sequence of tokens:

TOKEN CLASS	LEXEME	TOKEN VALUE
IDENT	a	a
EQ	=	n/a
IDENT	b	b
PLUS	+	n/a
IDENT	c	c
MINUS	-	n/a
NUM	012	10

Note that the space between the a and = in the original source code is not relevant and is not considered a token. This is part of the definition of the C language: spaces, tabs and newlines are **delimiters**, i.e. they may appear 0 or more times in between tokens, but are not themselves considered tokens. Another way of looking at this is that whitespace characters are tokens which are simply discarded and do not reach the syntactical stage. Other languages may have other rules in which whitespace is significant.

At the next level, of syntactical analysis, the parser will follow a series of rules, called a grammar, which defines the syntax of the language. Consider a grammar for a simple language that only understands arithmetic expressions and then consider the example above, again. In terms of the grammar, the tokens a, b and c occuring above are equivalent. The parser doesn't care what you call the variables identifiers, it just cares that it has seen a variable name.

Therefore we can say that the lexer consumes the input character stream which comprises the source program, and produces a stream of tokens. Each token is identified to the parser by a **token class** (e.g. IDENTIFIER or NUM), and where there is additional information, such as the name of the identifier or the value of the integer literal, that is known as the **token semantic value** and is also passed up to the parser. The literal string of characters which comprise the token is called the **lexeme**. The lexeme is discarded after recognition by the lexer and extraction of its semantic value. It is not carried further into the compilation process, except perhaps for error reporting.

The token value is not necessarily the same as the lexeme. In the example above, the number was given in octal. This same token, with the exact same token class and semantic value, may come from the lexeme `10`, `0x0A`, or in fact from an infinite number of equivalent lexemes (consider 000000012).

We will see that when we build a compiler using lex/flex and yacc/bison, the token class is represented as an integer return code from the lexer function `yylex()`. The lexeme is returned in the global `char *` variable `yytext`. The semantic value is returned in another global variable `yylval`, which may be of any data type (including struct or union) that the compiler writer chooses.

Specifying Lexical Rules with Regular Expressions

Both lexical and syntactical analysis can be performed mechanically. Specialized languages are used to specify the lexical and grammatical rules. We shall see in the next unit that grammar rules are a superset of lexical rules. However, in terms of practical compiler construction, it is easier and more efficient to specify the lexical rules using a different, more limited language called **Regular Expressions.**

Using regular expressions gives us a standardized way to talk about lexical rules, as opposed to using ad-hoc methods or resorting to narrative descriptions. Furthermore, regular expressions lend themselves easily to automatic generation of programs to recognize them.

The reader should consult one of the many freely available and comprehensive resources on regular expressions. The *flex* manual clearly explains the regular expressions that are used by flex and how to apply them, including order of operations and escaping issues. The following is a brief overview and not a complete list:

REGEXP	MATCHES
`x`	The single character x
`.`	Any single character (except newline)
`\t`	ASCII TAB. All C-style single character escapes are recognized
`[abc]`	single character a, b or c
`[A-Z]`	single uppercase letter
`[ab-hq]`	the single letter a, b,c,d,e,f,g,h or q
`[^0-9]`	any single character (including newline) except a digit
`RE*`	zero or more instances of the expression RE
`RE+`	one or more instances of the expression RE
`RE?`	zero or one instance of the expression RE
`RE{m,n}`	between m and n instances of RE
`RE{m,}`	m or more instances of RE
`RE{n}`	exactly n instances of RE
`(RE)`	expression RE (parens used for order of operations)
`AB`	expression A followed by expression B (concatenation)
`A|B`	either expression A or expression B (alternation)
`^RE`	expression RE, but only at the start of a line
`RE$`	expression RE, but only at end of line

Note that to match patterns that literally include special characters such as plus, one needs to either escape them with the backslash, or place the pattern inside double quotes. The latter escapes all regular expression operator characters. Some later versions of flex include additional regular expression pattern operators which are derived from the Extended Regular Expression language of Perl. These operators may not be backwards-compatible with earlier Flex versions or with lex.

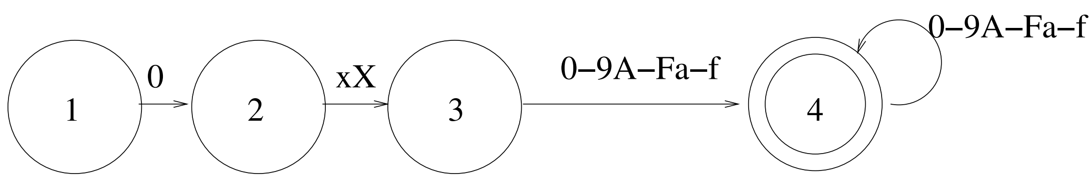
Recognizing Tokens with Finite Automata

Let us say that we wish to examine a string of characters and determine if it is a valid hexadecimal literal. We can write the regular expression:

```
0[xX][0-9A-Fa-f]+
```

for this. It means, in narrative: "the input must consist of the digit 0, the letter x or X, followed by one or more hexadecimal digits."

The typical structure of a program for lexical analysis is a **state machine** which is also known as a **finite automaton**. An FA has a finite number of states and a specific initial state. At each state, it considers the next input character and then transitions to a new state based on the current state and the character (the new state may be the same as the current state). One or more states are **accepting states** in which we conclude that the pattern has been matched. We can diagram an FA for the above RE:



The initial state is 1 and accepting states are denoted by a double circle. Drawing the FA follows fairly intuitively from the RE.

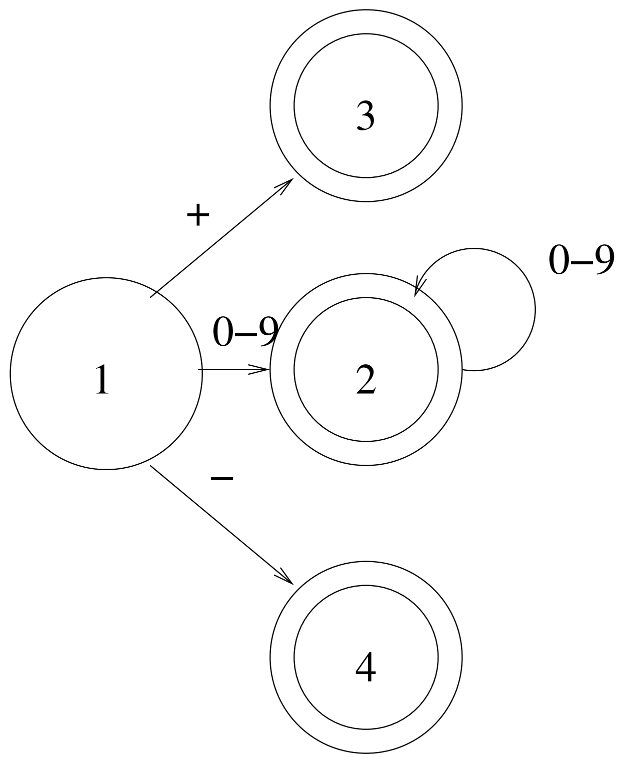
Consider the string `0X1`. Starting at state 1, we would match the 0 character and go to state 2. Then we would match the uppercase X as one of the allowable transitions to state 3. The character 1 is an allowable transition to state 4. We have now reached an accepting state, and the string 0X1 is recognized by the FA.

We have constructed a FA which can answer the question: does a string of characters match the pattern? But it is not very useful as a lexical analyzer. We need to be able to match a series of lexemes, not just one.

Now let us consider another example in which there are several regular expression patterns, each of which describes an allowable token. Our language will consist of decimal integers and the plus and minus operators. (The backslash in the second pattern below escapes the special meaning of the + character):

```
[0-9]+
\+
-
```

The job of the lexer will be to recognize one of these three patterns. Therefore, our overall FA will have three accepting states, corresponding to these three patterns:



Consider the behavior of this FA when the input stream is 12+34. Should this be tokenized as 1/2/+/3/4, or 12/+/34, or perhaps some other way? We need some rules to disambiguate:

• (1) Regular expressions are "greedy." They always try to match the longest valid string of characters. When the FA is in an accepting state and the next character forms a valid transition, that transition is taken and the FA continues.

• (2) If the FA is in an accepting state but the next character does not form a valid transition, then the string is accepted. The next character is left on the table, and the lexical analysis will resume at the initial state with that character.

• (3) If the FA is not in an accepting state and the next character does not form a valid transition, then an error has been detected. There is no token pattern which matches the lexeme. We will consider error conditions later.

Now, Let us write a C function which when called will consume input until a token is matched, and then return that token.

```
#define INIT 1
#define DIGITS 2

int lex()
{
  int state,c;

        state=INIT;
        for(;;)
        {
                c=getc(stdin);
                switch(state)
                {
                 case INIT:
                        if (isdigit(c))
                        {
                                state=DIGITS;
                                break;
                        }
                        if (c=='+') return PLUS;
                        if (c=='-') return MINUS;
                        if (c==EOF) return EOF;
                        ERROR;
                 case DIGITS:
                        if (!isdigit(c))
                        {
                                ungetc(c,stdin);
                                return NUMBER;
                        }
                }
        }
}
```

The structure of the lexer function in C is an endless loop which consumes a character at a time, and advances from state to state based on each character, until an accepting condition is reached and the function returns. When writing a lexer by hand, it is better to give the states meaningful names as opposed to state 1, state 2, etc. Note that states 3 and 4 of the FA are not defined explicitly as states but rather the return happens as soon as the appropriate character is seen. This shortcut is allowable since states 3 and 4 are accepting states with no transitions out.

It is assumed that `PLUS`, `MINUS` and `NUMBER` have been defined as integer constants elsewhere in the program.

Note that we return EOF when we have reached the end of input. Note that a macro ERROR (not defined here) is needed for the case of seeing input which does not match any rule.

As we are reading in digits, when we encounter a non-digit, we must use the `ungetc` function which pushes the character we had previously read back to standard input, so that the next time the lexer is called, it picks up again at that character which is the beginning of the next token (rule #2 above). This is called “pushback”. Another way of doing the same thing is to keep a global variable, say `curchar`, seed it initially with a call to `getc`, and then only call `getc` again when we have consumed it. This is often called “look-ahead.” These methods are essentially equivalent.

In general, it is necessary for the lexer to use push-back / look-ahead. The standard C library function `ungetc` is only guaranteed to work for one character. We can come up with examples of lexical rules, however, which require more than one character of backing up:

```
clue
clueless
lest
```

Let us say the input stream is "cluelest". After having seen "clue", we do not know if we should match rule 1 or rule 2. We must continue to accept characters. Now, after having seen "clueles", things are looking pretty good for pattern #2, but then we see "cluelest" which obviously doesn’t match.

In order to fix this, we must introduce more rules which refine our previous set of three rules on FA operation:

• (4) Whenever we reach an accepting state, we "remember" that state and our place in the input stream at the time. Only the very last accepting state is remembered, i.e. this is not a stack of states.

• (3A) If there are no valid transitions out of a non-accepting state, but we are remembering a previous accepting state, then we return to that state and accept the pattern. The input stream is reset to the remembered position, and we "forget" that we have seen the subsequent characters which led to the dead-end. Otherwise previous rule (3) applies and the input string is rejected.

In the example above, upon seeing the ’t’ in "cluelest", we back up to the accepting state for the pattern "clue", and the "les" characters are put back on the table. The state machine resumes the next time in the initial state, and the characters "lest" are recognized by the third pattern.

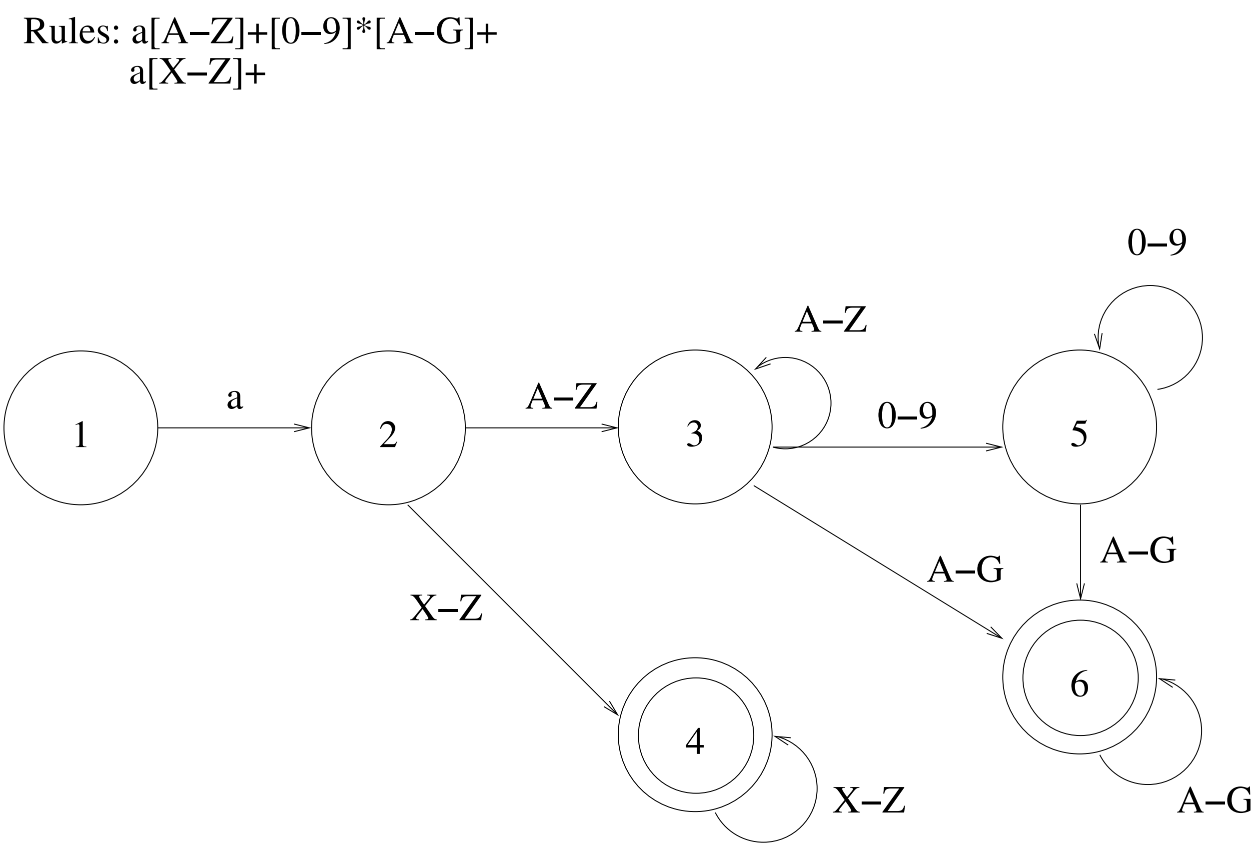
Backing up creates challenges if the lexer is meant to accept input from a non-seekable file (e.g. the keyboard). The lexer must introduce a buffering layer with a buffer large enough to back up out of the deepest mess it can get itself into. Note that a programming language is usually designed so that writing a compiler for it is not an exercise in masochism. Examples such as clue/clueless/lest above are fairly rare, and most languages can be lexically analyzed with just a single character of pushback and no backing-up.

NFA vs DFA

Now let us consider another contrived example which will motivate a discussion of deterministic vs non-deterministic finite automata.

```
a[A-Z]+[0-9]*[A-G]+
a[X-Z]+
```

Our first, intuitive stab at drawing the FA for this system of regular expression patterns might be:



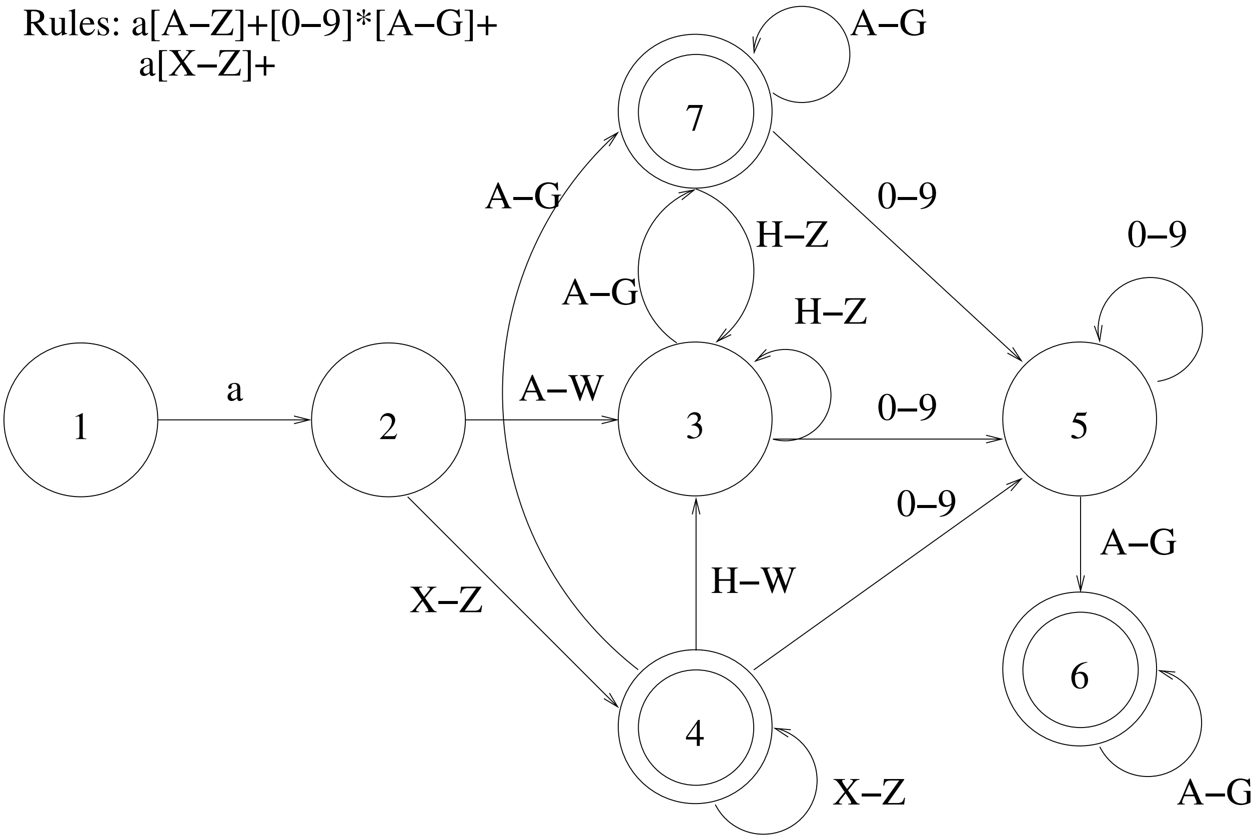
If we were to try to implement this FA as a C function we would stumble upon a problem. In states 2 and 3, there are multiple transitions for a given character. This type of FA is known as a **Non-deterministic Finite Automaton (NFA)** and does not lend itself to efficient implementation. We could follow this NFA and implement it using a path searching algorithm, i.e. try one path and if it leads nowhere, back up and try the other path. To do this requires a stack and potentially wastes a lot of time exploring dead-ends.

Thankfully, it is a property of any NFA that it can be transformed into a **Deterministic Finite Automaton (DFA)** (i.e. the sort of FA which we have been using already). The distinction between the two is that:

• A NFA may have multiple transitions from a given state for a given character. A DFA can have only one transition per character.

• A NFA may have a transition on ε, the empty character. This is not allowed for a DFA.

We just need to introduce some additional states which encode information about where we have already been. Here is the DFA for the above set of two rules:



There are only two rules, but there are 3 accepting states. This DFA is not intuitive, and it might be difficult to understand by casual examination the set of regular expressions which it was intended to match. Thankfully, this is a contrived example, and most lexical rules of most programming languages lend themselves to straightforward DFA construction.

Lex/Flex

Sometimes it is desirable to hand-code the lexical analyzer, especially if the source language is simple. But in more complex, real-world languages such as C, a hand-coded lexer is tedious and prone to error.

Fortunately, there are algorithms to take a set of regular expressions and transform it into a DFA. The "Dragon" textbook shows that *any* set of regular expressions can be easily translated into an NFA (section 3.6), and further shows that any NFA thus constructed can be transformed into a DFA (section 3.7), regardless of the number or complexity of the patterns.

Therefore, a program can be written which takes the set of regular expressions of the source language as an input, and constructs a DFA state machine in the form of a function which can be called. This program is an example of a code generator in that it outputs a high-level language (C) which is then intended to be compiled in conjunction with the rest of the compiler.

The de-facto choice is the `lex` program, which has been available on UNIX systems for many years. An open-source, enhanced version called `flex` is used on most modern, open-source UNIX-like operating systems. Again, the reader is referred to the man pages for flex and online tutorials for a complete discussion. Below is presented a brief overview.

Flex accepts as input a specification file with a `.l` extension. The meat of this file is a series of rules (expressed as regular expressions) and associated actions to execute when the rule matches. Here is an example of a simple lexer:

```
%option noyywrap
%{
#define NUMBER 1
#define PLUS 2
#define MINUS 3

int yylval;
%}
%%
[0-9]+              {yylval=atoi(yytext);return NUMBER;}
\+                  {return PLUS;}
-                   {return MINUS;}
 .                  {fprintf(stderr,"Error: unknown token %s\n",yytext);}



%%

main()
{
 int t;
          while (t=yylex())
          {
           switch(t)
           {
                    case NUMBER: printf("NUMBER: %d\n",yylval);
                                        break;
                    case PLUS: printf("PLUS\n");break;
                    case MINUS:printf("MINUS\n");break;
           }
          }
          printf("EOF\n");
}
```

Lex/flex generates a file, which is by default called `lex.yy.c`. It contains a single function named `yylex` and several static, global variables all of which begin with the prefix `yy`. These names can be changed to avoid conflict if needed, but as there is usually just one lexical analyzer in the compiler program, it is usually not necessary. It is possible to include C code in the `.l` file which will be inlined ahead of the definition of the `yylex()` function. We see an example of that above surrounded by `%{` and `%}` delimiters.

The function `yylex()` consumes input characters until a token is matched (or EOF is reached). `yylex()` returns the integer token code, or 0 on EOF. Because `yylex()` maintains hidden static global variables, the next time it is called, it simply resumes where it left off.

Normally, the lexer is used in conjunction with a parser which expects integers representing token classes as the return value. In the example above, we are using token class values 1, 2 and 3 to represent the three possible tokens. In general it is bad practice to hard-code numbers like this, and in reading the documentation for lex/flex and yacc/bison (the parser generator) you'll see how to automatically assign symbolic names for these token class integers. In the example above, we have included #define constants to create symbolic names for these token class numbers.

Normally, `lex.yy.c` is compiled and linked against other C source code, which contains main() and other support functions. In the example above, the code for main() has been placed in the `.l` file. Everything after the second `%%` delimiter will be copied verbatim into `lex.yy.c`, after the definition of `yylex()`.

Between the first and second `%%` delimiters are the regular expression patterns to be recognized. After each pattern, C code may be specified within curly braces. This code will be placed into `lex.yy.c` such that it is executed when the pattern is matched.

The automatically generated lexer is thousands of lines long and not meant to be edited by hand. A pseudo-code outline of the yylex function follows, with a lot of stuff removed:

```
int yylex()
{

 AGAIN:
        int yy_current_state=yy_start;          //1 unless start conditions
                yy_cp=yy_last_accepting_cpos;   //Resume from last position
                c=get_next_character_from_buffer();
                while ( (yy_current_state = yy_nxt[yy_current_state][c]) > 0 )
                        {
                        if ( yy_accept[yy_current_state] )
                                {
                                        /* Remember last accepting state */
                                yy_last_accepting_state = yy_current_state;
                                yy_last_accepting_cpos = yy_cp;
                                }

                        c=get_next_character_from_buffer();
                        }

                yy_current_state=yy_last_accepting_state;
                yy_act = yy_accept[yy_current_state];
                /*yytext has been set up with NUL terminator from start
                  of match to yy_last_accepting_cpos */
                switch ( yy_act )       //Which rule
                {
                 case 1:
                        {...embedded action for rule 1....}
                        break;
                 case 2:
                        {...embedded action for rule 2...}
                // etc
                 default:
                        //fall-through
                }
  // No explicit return in action, or no valid action and need to back up
 goto AGAIN;
}
```

The embedded actions are copied verbatim from the `.l` file to `lex.yy.c`. flex doesn't really know anything about C syntax, and will not catch any C language errors here. They will not be reported until one tries to compile `lex.yy.c`. Note that a `return` of the token class code is expected in each action which recognizes a valid token (but read the documentation about error recovery, and more advanced features such as *start conditions*).

The string `yytext` is available in an action and contains the (nul-terminated) contents of the actual characters matched (the lexeme).

The example above illustrates the processing of token semantic values. We will see that yacc/bison expect these values to be passed in a global variable called `yylval`. In this example, it is of type `int`, but in a practical compiler, it will be a much more complicated struct or union.

When lex/flex sees a character or characters which do not form a valid lexeme, the default behavior is to ignore them and continue accepting input. The invalid characters are printed to stdout. This is a very annoying default, and thus the last pattern in the example above is there to match any character and report it as an error. lex/flex specifies that when multiple patterns match the input, the longest match wins. If that doesn't settle the matter, then the pattern defined first in the `.l` file wins. Thus the single dot above, which matches any character, has the least priority, and there is no danger that the rule will fire erroneously, e.g. when the plus character should have been recognized.

There are many flags to control the level of optimization which flex performs, and to trade-off between table size and efficiency. For example, in most real-world languages, there are broad ranges of characters which are equivalent in terms of the lexer (such as A-Z). One option (which is enabled by default) causes the generated lexer to first pass each character through an equivalency class table. The resulting, smaller integer is then used for the tables.

The user of flex can enable options which give verbose debugging output and this can be useful in understanding the algorithm which flex uses to construct its tables.

The student is encouraged to try several examples such as the one above in lex/flex, and to examine the `lex.yy.c` output file.

Some compiler writers prefer to write the lexer by hand because it offers finer control, especially in error cases, and because for many programming languages, the lexer isn't really that bad to code correctly. Flex generates fairly efficient lexers, and it is unlikely that hand-coding the lexer will generate any significant performance improvement.

Symbol Tables and Reserved Words

Most programming languages have **reserved words**, such as `if`, `break`, etc. These reserved words also, generally, match the same regular expression patterns as identifiers (or are a subset of them). To the parser, which will receive a token stream from the lexer, it is rather important to know whether a certain string of characters is a particular reserved word, or just an identifier.

When the lexer is coded by hand, it is often more convenient to build the DFA to recognize identifiers (e.g. `[_A-Za-z][_A-Za-z0-9]+` in C) and then check to see if that string matches a reserved word. An efficient way of doing this is to maintain a **hash table** of reserved words. Because the list of words is known in advance, this is a good application of **perfect hashing**. The reader is referred to the textbook, for a good discussion of hashing methods.

When the lexer is generated by lex, it is easier and more efficient to include rules which match each reserved word individually and return a corresponding token code. Many more DFA states are generated this way, but not so many that the increased memory consumption is even close to being a problem.

In some compilers, the lexer does further work on identifiers, keeping a **symbol table** of identifiers which it has already seen. Others return simple strings to the parser and let it, or higher-level code, take care of the rest. Since this is really a semantic issue, it is premature to discuss it now. We will also defer discussion of how to handle numeric, character and string literals with respect to their lexical values.