**CASE STUDY: LEX**

Lex is officially known as a "Lexical Analyser". It's main job is to break up an input stream into more usable elements. Or in, other words, to identify the "interesting bits" in a text file.

For example, if you are writing a compiler for the C programming language, the symbols { } ( ); all have significance on their own. The letter a usually appears as part of a keyword or variable name, and is not interesting on it's own. Instead, we are interested in the whole word. Spaces and newlines are completely uninteresting, and we want to ignore them completely, unless they appear within quotes "like this"

All of these things are handled by the Lexical Analyser.

During the first phase the compiler reads the input and converts strings in the source to tokens. With regular expressions we can specify patterns to lex so it can generate code that will allow it to scan and match strings in the input. Each pattern specified in the input to lex has an associated action. Typically an action returns a token that represents the matched string for subsequent use by the parser. Initially we will simply print the matched string rather than return a token value.

The following represents a simple pattern, composed of a regular expression, that scans for identifiers. Lex will read this pattern and produce C code for a lexical analyzer that scans for identifiers.

letter(letter|digit)\*

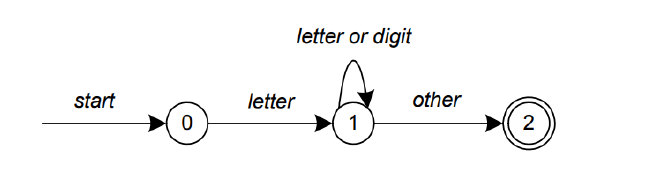
This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits. This example nicely illustrates operations allowed in regular expressions:

 repetition, expressed by the “\*” operator

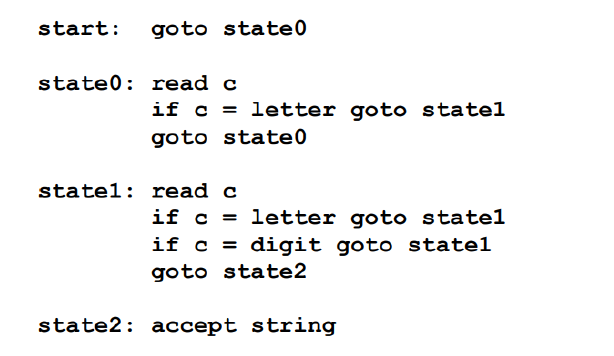
 alternation, expressed by the “|” operator

 concatenation

Any regular expression expressions may be expressed as a finite state automaton (FSA). We can represent an FSA using states, and transitions between states. There is one start state and one or more final or accepting states.



In above Figure state 0 is the start state and state 2 is the accepting state. As characters are read we make a transition from one state to another. When the first letter is read we transition to state 1. We remain in state 1 as more letters or digits are read. When we read a character other than a letter or digit we transition to accepting state 2. Any FSA may be expressed as a computer program. For example, our 3-state machine is easily programmed:



This is the technique used by lex. Regular expressions are translated by lex to a computer program that mimics an FSA. Using the next input character and current state the next state is easily determined by indexing into a computer-generated state table.

Now we can easily understand some of lex’s limitations. For example, lex cannot be used to recognize nested structures such as parentheses. Nested structures are handled by incorporating a stack. Whenever we encounter a “(” we push it on the stack. When a “)” is encountered we match it with the top of the stack and pop the stack. However lex only has states and transitions between states. Since it has no stack it is not well suited for parsing nested structures. Yacc augments an FSA with a stack and can process constructs such as parentheses with ease. The important thing is to use the right tool for the job. Lex is good at pattern matching. Yacc is appropriate for more challenging tasks.

**YACC**

Yacc is officially known as a "parser". It's job is to analyse the structure of the input stream, and operate of the "big picture". In the course of it's normal work, the parser also verifies that the input is syntactically sound.

Consider again the example of a C-compiler. In the C-language, a word can be a function name or a variable, depending on whether it is followed by a ( or a = There should be exactly one ) for each in the program. YACC stands for "Yet Another Compiler". This is because this kind of analysis of text files is normally associated with writing compilers.

However, as we will see, it can be applied to almost any situation where text-based input is being used.

For example, a C program may contain something like:

{

int i;

i = 33;

printf("int: %d\n",i);

}

In this case, the lexical analyser would have broken the input sream into a series of "tokens", like this:

{

int

i

;

i

=

33

;

printf

(

"int: %d\n"

,

i

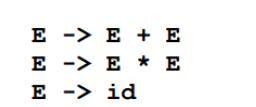
)

;

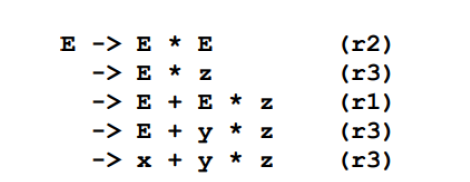
}

Note that the **lexical analyser** has already determined that where the keyword int appears within quotes, it is really just part of a litteral string. It is up to the **parser** to decide if the token int is being used as a keyword or variable. Or it may choose to reject the use of the name int as a variable name. The parser also ensures that each statement ends with a ; and that the brackets balance.

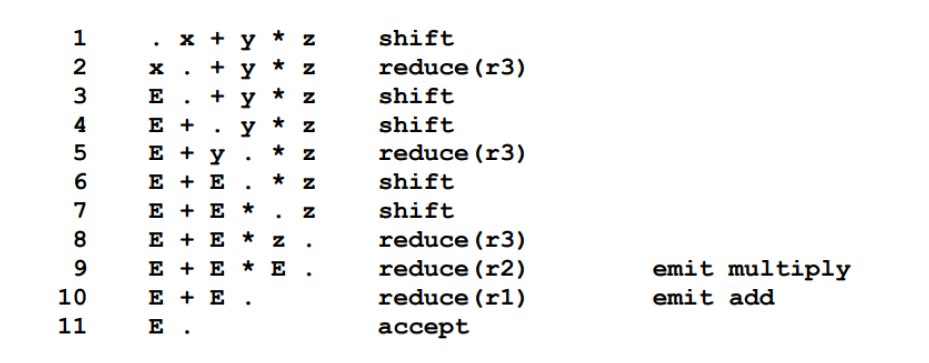
Grammars for yacc are described using a variant of Backus Naur Form (BNF). This technique, pioneered by John Backus and Peter Naur, was used to describe ALGOL60. A BNF grammar can be used to express context-free languages. Most constructs in modern programming languages can be represented in BNF. For example, the grammar for an expression that multiplies and adds numbers is



Three productions have been specified. Terms that appear on the left-hand side (lhs) of a production, such as E (expression) are nonterminals. Terms such as id (identifier) are terminals (tokens returned by lex) and only appear on the right-hand side (rhs) of a production. This grammar specifies that an expression may be the sum of two expressions, the product of two expressions, or an identifier. We can use this grammar to generate expressions:



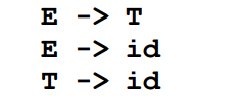
At each step we expanded a term and replace the lhs of a production with the corresponding rhs. The numbers on the right indicate which rule applied. To parse an expression we need to do the reverse operation. Instead of starting with a single nonterminal (start symbol) and generating an expression from a grammar we need to reduce an expression to a single nonterminal. This is known as bottom-up or shift-reduce parsing and uses a stack for storing terms. Here is the same derivation but in reverse order:



Terms to the left of the dot are on the stack while remaining input is to the right of the dot. We start by shifting tokens onto the stack. When the top of the stack matches the rhs of a production we replace the matched tokens on the stack with the lhs of the production. In other words the matched tokens of the rhs are popped off the stack, and the lhs of the production is pushed on the stack. The matched tokens are known as a handle and we are reducing the handle to the lhs of the production. This process continues until we have shifted all input to the stack and only the starting nonterminal remains on the stack. In step 1 we shift the x to the stack. Step 2 applies rule r3 to the stack to change x to E. We continue shifting and reducing until a single nonterminal, the start symbol, remains in the stack. In step 9, when we reduce rule r2, we emit the multiply instruction. Similarly the add instruction is emitted in step 10. Consequently multiply has a higher precedence than addition.

Consider the shift at step 6. Instead of shifting we could have reduced and apply rule r1. This would result in addition having a higher precedence than multiplication. This is known as a shift reduce conflict. Our grammar is ambiguous because there is more than one possible derivation that will yield the expression. In this case operator precedence is affected. As another example, associativity in the rule:

**E -> E + E** is ambiguous, for we may recurse on the left or the right. To remedy the situation, we could rewrite the grammar or supply yacc with directives that indicate which operator has precedence.



Yacc takes a default action when there is a conflict. For shift-reduce conflicts yacc will shift. For reduce-reduce conflicts it will use the first rule in the listing. It also issues a warning message whenever a conflict exists. The warnings may be suppressed by making the grammar unambiguous.

Online Notes

<https://www.idexcel.com/resources/case-studies/Amazon-Lex-Case-Study.pdf>

<https://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf>

<https://iith.ac.in/~ramakrishna/Compilers-Aug14/doc/flex.pdf>

<https://courses.cs.vt.edu/~cs1104/Compilers/Lex.Yacc.html>

<http://tinf2.vub.ac.be/~dvermeir/courses/compilers/yacc.pdf>

<https://www3.diism.unisi.it/~maggini/Teaching/TEL/slides%20EN/04%20-%20YACC.pdf>

<https://www.epaperpress.com/lexandyacc/download/yacc.pdf>

Video Links

<https://www.youtube.com/watch?v=VweEoX3PVPY>

<https://www.youtube.com/watch?v=912hi4nLp0E>

<https://www.youtube.com/watch?v=y29NaFR8h2g>

<https://www.youtube.com/watch?v=87K0nHA_F5Y>

<https://www.youtube.com/watch?v=fo4TLclA3B4>

<https://www.youtube.com/watch?v=bnPxdBUkGpc>