3 Systematic lexical analysis

Generation of lexical analysers is usually a (semi-)automatic process. This is possible because the design of modern programming languages allows compilers to define their lexical units, known as tokens or atoms, in terms of sets of strings, known as lexemes, which are capable of easy description by regular expressions.

Older languages, such as Fortran, do not allow such means to be employed. Only ad hoc, i.e. language specific, solutions can be used for some parts of such languages.

Regular expressions

Regular expressions can be used to build grammars, which define a set of strings. Any such set of strings constitutes a language. The set of languages which can be defined using regular expressions is a sub-set of those which may be defined by a context free grammar.

The following rules define the possible regular expressions over a given alphabet, S. An alphabet is the set of atomic symbols (characters) allowed by the grammar of a language.

1. ϵ is a regular expression denoting $\{\epsilon\}$, i.e. the set containing just the empty (null) string.
2. If α is a symbol in S then α is also a regular expression denoting $\{\alpha\}$, i.e. the set containing just the string α. (Note the three, technically distinct, uses of α.)
3. If ρ and σ are regular expressions denoting languages $\mathcal{L}(\rho)$ and $\mathcal{L}(\sigma)$ respectively then the following are also regular expressions:

   Alternative combination: $\rho|\sigma$ is a regular expression denoting $\mathcal{L}(\rho) \mid \mathcal{L}(\sigma)$.

   Concatenation: $\rho\ \sigma$ is a regular expression denoting $\mathcal{L}(\rho)\mathcal{L}(\sigma)$.

   Repetition: $\rho*$ is a regular expression denoting $(\mathcal{L}(\rho))*$.

   Parentheses: (ρ) is a regular expression denoting $\mathcal{L}(\rho)$.

This uses the operators

| *or*

$*$ *zero or more repetitions*

and *concatenation* is implied by juxtaposition.

Parentheses around sub-expressions affect the order of evaluation, as is normal. To reduce the need for parentheses, the following precedence is normal, along with left association.

1	repetition,	$*$
2	concatenation,	*implied*
3	alternatives,	|

The set of strings defined by a regular expression, is called a regular set.

N.B. these operators are very similar to those used in GNU EMACS. Consider the character patterns of EMACS as a regular set, awaiting expansion.

Algebraic properties

For completeness, here are the algebraic rules for regular expressions, taken from Aho, Sethi and Ullman p.96.

Axiom			Description
$\rho\|\sigma$	$=$	$\sigma\|\rho$	$\|$ is commutative
$\rho\|(\sigma\|\tau)$	$=$	$(\rho\|\sigma)\|\tau$	$\|$ is associative
$(\rho\sigma)\tau$	$=$	$\rho(\sigma\tau)$	concatenation is associative
$\rho(\sigma \mid \tau)$	$=$	$\rho\sigma \mid \rho\tau$	concatenation distributes
$\sigma \mid \tau)\rho$	$=$	$\sigma\rho \mid \tau\rho$	over $\|$
$\epsilon\rho$	$=$	ρ	ϵ is the identity element
$\rho\epsilon$	$=$	ρ	for concantentation
$\rho*$	$=$	$(\rho \mid \epsilon)*$	relation between $*$ and ϵ
$\rho**$	$=$	$\rho*$	$*$ is idempotent

Regular definitions

A complete grammar defined by regular expressions may use a sequence of inter-related simpler definitions. It then takes the form

$$\begin{aligned} \delta_1 &\rightarrow re_1 \\ \delta_2 &\rightarrow re_2 \\ &\cdots \\ &\cdots \\ \delta_n &\rightarrow re_n \end{aligned}$$

where each δ_i is a name to be given to the regular expression, re_i, and $\rightarrow$ is the production operator. Each re_i must contain only the basic symbols of the language and names defined in preceding definitions. That is, each re_i is a regular expression defined over the alphabet

$$S \Rightarrow \{\delta_1, \delta_2, \ldots, \delta_{i-1}\}$$

Examples

(a) Pascal identifiers

$$\begin{aligned} \mathit{letter} &\rightarrow A \mid B \mid ... \mid Z \mid a \mid b \mid ... \mid z \\ \mathit{digit} &\rightarrow 0 \mid 1 \mid ... \mid 9 \\ \mathit{id} &\rightarrow \mathit{letter}\ (\ \mathit{letter} \mid \mathit{digit}\ )* \end{aligned}$$

(b) Pascal unsigned numbers.

Here we extend the notation to use the operators

$+$ (one or more repetitions) and

? (zero or one occurrence).

Each has the same precedence as $*$.

$$\begin{aligned} \mathit{digit} &\rightarrow 0 \mid 1 \mid \ldots \mid 9 \\ \mathit{digits} &\rightarrow \mathit{digit}\ + \\ \mathit{optional\_fraction} &\rightarrow (\mathit{digits})\ ? \\ \mathit{optional\_exponent} &\rightarrow (\ E(\ +\mid -\ )\ ?\ \mathit{digits}\ )\ ? \\ \mathit{number} &\rightarrow \mathit{digits}\ \mathit{optional\_fraction}\ \mathit{optional\_exponent} \end{aligned}$$

Restrictions

As stated before, regular expressions can define only a sub-set of all context free grammars. They are not capable of describing most complete languages. In essence, they describe languages composed of sets of strings of the form

$$S \rightarrow \alpha B$$

where α is a basic symbol and B is a regular expression.

1. They cannot describe balanced nesting or embedding of constructs, such as parenthesised expressions. This would require a repetition of opening brackets to be matched by the same number of closing brackets, with possibly intervening strings within these repetitions. Note that, in Pascal, *begin* and *end* are brackets.

2. Repetition of the same string cannot be described, although in fact no context free grammar can be made to describe it. An example of such a set of strings is:

$$\{wcw \mid w \text{ is a string of } a\text{'s and } b\text{'s }\}$$

3. Finally constructs where the number of repetitions is fixed by the value of a part of the string cannot be described. Symbols have identity, but no other value, in regular expressions. The definition of some types of literal may not form a regular expression. Aho, Sethi and Ullman cite the, now nonstandard, Fortran Hollerith literal, with the form

   $nHa_1a_2\ldots a_n$

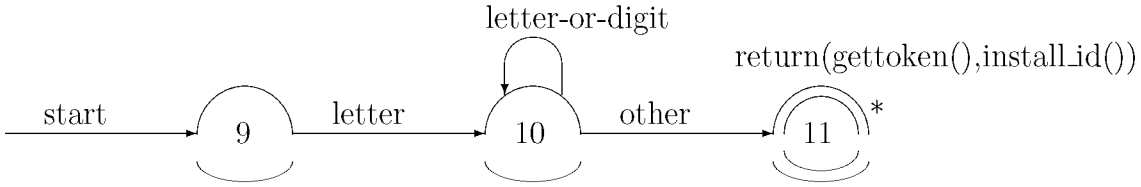
   Again, no context free grammar could describe this set of strings.

Analysing regular expressions

The systematic way to perform lexical analysis is to define the tokens as a regular set generating all the possible lexemes. This is usually straightforward for modern languages, whose definition will be based on such a notation. This definition can then be used to generate a finite state automaton whose accepting states indicate success in finding a token.

Let us follow Aho, Sethi and Ullman's use of transition diagrams to describe such automata. These show states as circles or nodes and possible transitions as arcs or edges. Double circles indicate accepting states.

Thus, a transition diagram for a Pascal identifier is as follows.



There is always a start state, possibly several. Loops represent repetitions.

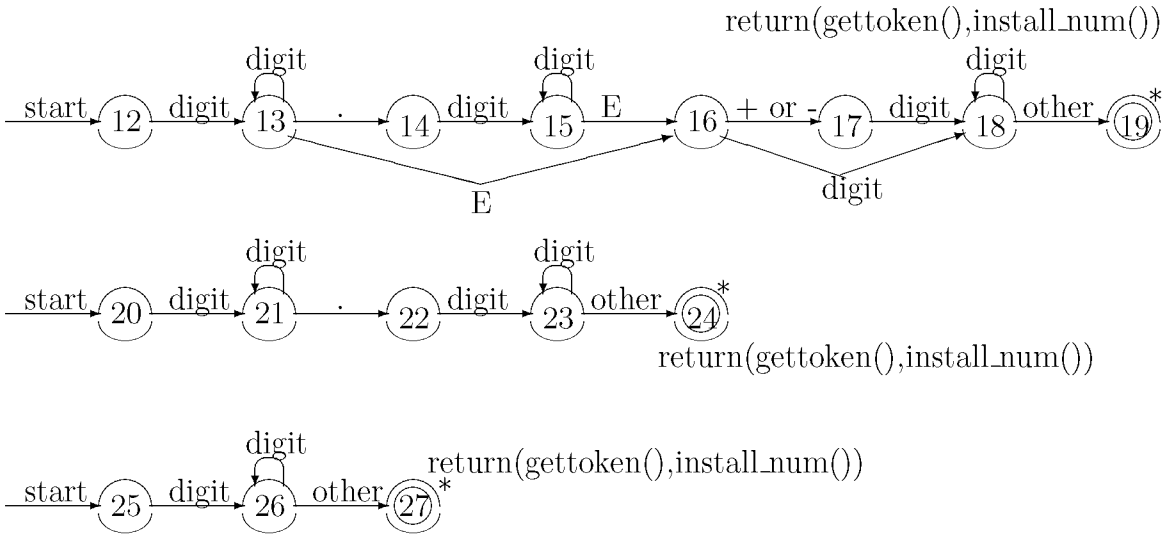
At the accepting state the appropriate lexical value to describe the lexeme must be generated, probably as a tuple containing (lexical type, appropriate value).

The question of *look ahead* (or *put back* in some implementation languages) must also be dealt with. Often the accepting state is reached after examining the first character of the next string in the program. If look ahead is used (Pascal's `Input^`) this may not matter. In languages which do not support look ahead, the current input must be backspaced (C's `ungetch()`) and this may mean deliberate buffering in low level implementations.

More difficult examples

The analysis of an identifier is very simple. Only one accepting state is reached. The regular expression contains no unique zero or more repetitions and so the longest possible string is found by the automaton as soon as other is reached.

The unsigned number's expression requires more complex analysis. It is capable of generating three kinds of number, i.e. exponentiated real (floating point), unexponentiated real (fixed point) and integer. Each has a different transition diagram as below.



This creates a problem, as we must first scan for the longest possibility and then rewind before checking for the next possibility. It may be more efficient to combine the diagrams and merely note that 27 and 24 are possible terminating states. Only when we fail to reach 19 would we use 24 as the accepting state. If we fail to reach 24, 27 is used. This requires the values at the last possible accepting state to be restored when the next is not reachable.

Implementation

It is possible to code finite state automata by hand or to write a program to generate them from a set of regular definitions. Let us first consider hand coding, using a systematic approach. From this we can derive the algorithm for an automatic generator.

Basically we translate the transition diagram by defining the actions of each state. These actions must either lead to another state along an edge in the diagram or be terminating actions, when the current state is an accepting state. One way to visualise this is as a state table, which specifies, for each current state, what the new state is for each basic symbol which may follow. Where no legal basic symbol is found, the match has failed, i.e. the current string is not compatible with that automaton.

A failure to match may be an error in some states or for some symbols. Alternatively, it may be a signal to rewind the input and try to match another alternative. It is always an error to find a symbol which is not in the alphabet for that language.

State transition table for floating point number

	Current State							
	12	13	14	15	16	17	18	19
digit	13	13	15	15	18	18	18	accepts
.	fail	14	fail	16	fail	fail	19	accepts
E	fail	16	fail	fail	fail	fail	19	accepts
+/-	fail	fail	fail	fail	17	fail	19	accepts
other	fail	fail	fail	fail	fail	fail	19	accepts

Coding consists of a *loop*, `while state <> 19`, and a *case statement*. Since each selector of the main case, which selects on the current state, has comparatively few legal branches within it representing edges from that state to a successor, we can choose an *if statement*, but in general we should use a *subsidiary case statement*.

We have used without definition, in the following outline code fragment, the procedures and functions:

char NextChar()	returns the next item, using an appropriate buffering technique.
Fail()	returns us to the state prior to start, with a failure flag. Unwinds the input.
ReturnVal()	sets up a tuple for a successful match.
int NumVal(char*)	gives a value or table index for a number string.
char* CurString()	gives the string from current start to this state

```
   While (State!=19){
      switch (State){
case 12: switch (NextChar()){
      case digit: State=13; break;
      default: Fail();
         }                          /*-------------------------*/
case 13: switch (NextChar()) {      /*|                         |*/
      case digit: ;                 /*| Basic Code for          |*/
      case dot: State=14; break;    /*| lexical analysis of     |*/
      case E:   State=16; break;    /*| an unsigned floating    |*/
      default:  Fail();             /*| point number            |*/
         }                          /*|                         |*/
case 14: switch (NextChar()){       /*-------------------------*/
      case digit: State=15; break;
      default:    Fail();
         }
case 15: switch (NextChar()){
      case digit: ;
      case dot: State=16;
      default:  Fail();
         }
case 16: /* etc.*/
case 17: /* etc.*/
      }
   }
   Number=ReturnVal(NumVal(CurString()));
```

DFAs and NFAs

The previous example assumed that the regular expressions produced a deterministic finite state automaton. This is not always true. Many examples exist where a non-deterministic automaton results. There are two basic approaches to handling NFAs. They can be converted to DFAs with multiple-substate states, effectively tracing the NFA's multiple paths in parallel. They can be simulated as NFAs directly, which is more CPU intensive, but maybe less space intensive than the DFA approach.

Completing the analyser

The fragment given, and the diagram and table which helped to generate it, will deal with one regular expression (or one alternative within such an expression in this case). To deal with the whole language, all the rules for it must be combined. This is difficult to describe in a transition diagram, but can be done using a combined state diagram.

In essence, each state marked *fail* should be considered as a candidate for leading to the initial state of an alternative. Where this is so we can write that state with a star, to indicate the need to rewind before moving to it. Fail is then reserved for definite errors.

The accepting state should also be labelled according to whether it needs to put back the last character read. The asterisk on an accepting state is used to show that one input symbol beyond the last one of this lexeme has had to be read to match an ϵ transition.

After acceptance, the state should return to 0, indicating that a new token is to be scanned.

Automatic generation

The above rules are so automatic that the only need is to generate the program following them. Lex is a fairly simple example, where you must specify:

- declarations - any variables to be used, plus regular definitions.
- translation rules - code fragments to be enacted when each match is found.
- auxiliary procedures - NumVal() etc.

An example of Lex input

The following example is taken from Aho, Sethi and Ullman, p109.

Note that in Lex the translation rules are evaluated in order and so the placing of the rules for keywords before the rule for *id* avoids their being confused. This works with a strategy of placing keywords in the symbol table before reading any input.

```
%{
      /* definitions of manifest constants
      LT, LE, EQ, NE, GT, GE,
      IF, THEN, ELSE, ID, NUMBER, RELOP
%}

/* regular definitions */
delim      [ \t\n]
ws         {delim}+
letter     [A-Za-z]
digit      [0-9]
id         {letter}({letter}|{digit})*
number      {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?

%%

{ws}       {/* no action and no return */}
if         {return(IF);}
then       {return(THEN);}
else       {return(ELSE);}
{id}       {yylval=install_id(); return(ID);}
{number}  {yylval=install_num(); return(NUMBER);}
"<"        {yylval=LT; return(RELOP);}
"<="       {yylval=LE; return(RELOP);}
"="        {yylval=EQ; return(RELOP);}
"<>"       {yylval=NE; return(RELOP);}
">"        {yylval=GT; return(RELOP);}
">="       {yylval=GE; return(RELOP);}

%%

install_id() {
   /* procedure to install the lexeme, whose
   first character is pointed to by yytext and
   whose length is yyleng, into the symbol table
   and return a pointer thereto */

}

install_num() {
   /* similar procedure to install a lexeme that
   is a number */
}
```