Regular Expressions

Regular expressions provide a notation for describing the tokens used in modern programming languages.

A regular expression specifies a pattern. The set of character strings that match a regular expression is called a regular set.

For example,

$(underscore|letter)(underscore|letter|digit)*$

is a regular expression that specifies the set of Java identifiers – provided ...

Regular expressions and what they match

Conventional notation	Unix notation	matches
ϕ		no string
ϵ		the empty string
s	s	the string s
$A\|B$	$[AB]$	any string that matches A or B
AB	AB	any string that can be split into a part that matches A followed by a part that matches B
$A*$	$A*$	any string that consists of zero or more A's

s is any character string

A and B are regular expressions

Shorthands

- $A+$ one or more A's
  equivalent to $AA*$
- $A?$ zero or one A
  equivalent to $[\epsilon A]$
- $[0-9] = [0123456789]$

Using Regular Expressions

- Unix applications
  - substitution using gvim
    $< range > s/ < regexp > / < substitution >$
  - searching files for patterns
    $grep \ < regexp > < path >$
- Specifying valid input ...
- perl pattern specifications
  $t.e$ matches the or tee but not $tale$

Constructing regular expressions

Write a regular expression that matches the Java keyword **class**

Constructing regular expressions

Using conventional notation, extend your regular expression so that it matches any of the Java keywords: **abstract class extends**

Write an equivalent regular expression using Unix notation.

Constructing regular expressions

Using conventional notation, write a regular expression that matches unsigned integers

Do the same, this time using Unix notation.

Constructing regular expressions

Extend each of your previous regular expressions so that they match signed integers.

More information

To find out all about regular expressions in Unix, see

```
man -s 5 regexp
```


Finite State Automata

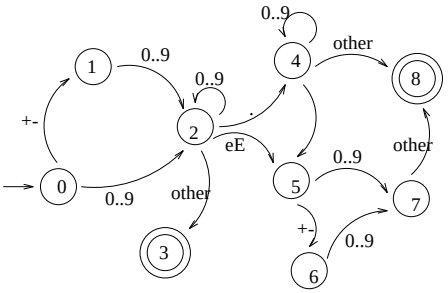
A finite state automaton (FSA) recognises strings that belong to a regular set.

It consists of

- a finite set of states,
- a set of transitions from state to state,
- a start state and
- a set of final states, called "accept states".

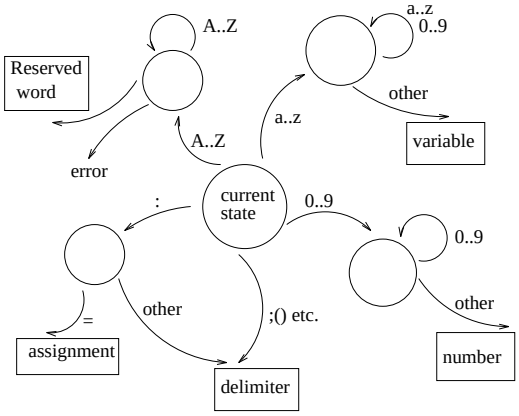
A FSA can be represented by a transition diagram – a directed graph whose nodes are labelled with state symbols, and whose edges are labelled with characters.

FSA that recognises numbers



This FSA is deterministic – for a given state and input character, the next state is uniquely determined.

Fragment FSA for a typical programming language



Pseudo code for a scanner

Assume one character, 'ch', has been read by 'getch', which reads a character, skips white space and produces a listing.

```
case ch of:
    ';' : token := semi
          getch
    ':' : getch; if ch = '='
                  then token := assign
                       getch
                  else token := colon
    '0' .. '9' : token := number
                 while ch in ['0' .. '9']
                       getch
                 endwhile
    ... &c.
```


Transition Tables

In general, a deterministic FSA can be programmed as a language independent driver that interprets a transition table. The transition table for the FSA that recognises numbers looks like this.

State	Character						
	+	-	.	e	E	0 ... 9	other
0	1	1				2	
1						2	
2			4	5	5	2	3
3							
4				5	5	4	8
5	6	6				7	
6						7	
7						7	8
8							

A scanner driver

```
State:= InitialState;
Read(CurrentChar);
loop
    NextState := TT(State,CurrentChar);
    exit when NextState = Error or
              CurrentChar = EOF;
    State := NextState;
    Read(CurrentChar);
end loop
if State in FinalStates
    then return valid token
    else lexical error
end if;
```


Using a Scanner Generator

There are algorithms for translating regular expressions to nondeterministic finite state automata (NFA), and for translating NFA to deterministic finite state automata (DFA).

These are used in scanner generators like lex, developed by M.E. Lesk and E. Schmidt of Bell labs.

Programming effort is limited to describing lexemes – an example of declarative programming.

Given input consisting of character class definitions, regular expressions with corresponding actions, and subroutines, lex generates a scanner called yylex.

Symbol Tables

The compiler uses symbol tables to collect and use information about names that appear in a source program.

Each time the scanner encounters a new name, it makes a symbol table entry for the name. Further information is added during syntactic analysis, and this information is used (and further extended) during semantic analysis and code generation.

Symbol Tables

The kind of information stored in a symbol table entry includes items like

- the string of characters denoting the name,
- the block or procedure in which the name occurs,
- attributes of the name (e.g. type and scope),
- the use to which the name is put (e.g. formal parameter, label ...)
- parameters, such as the number of dimensions in an array and their upper and lower limits, and
- the position in storage to be allocated to the name (perhaps).

The Symbol Table as an Abstract Data Type

Conceptually, a symbol table is a collection of pairs:

(name,information)

with operations to

- determine whether or not a name is in the table,
- add a new name,
- access the information associated with a given name,
- add new information for a given name, and
- delete a name, or a group of names.

Symbol Table Implementation

Symbol tables are typically implemented using

- linear lists,
- hash tables, or
- various tree structures.

In selecting an implementation, speed of access is traded against structural complexity.