Regular Languages

Lembit Jürimägi

Grammar

G = {S, N, T, R}

where

- G is the grammar,
- S is the nonterminal starting symbol,
- N is the set of all nonterminals:
  for example: {S, A, B},
- T is the set of terminals:
  for example: {a, b}
- R is the set of production rules,
  for example:

    S -> AB

    AB -> aABb

    A -> a

    B -> b

Chomsky Hierarchy

- Type 3: regular grammar
- Type 2: context-free grammar
- Type 1: context-sensitive grammar
- Type 0: unrestricted grammar

Regular Grammar

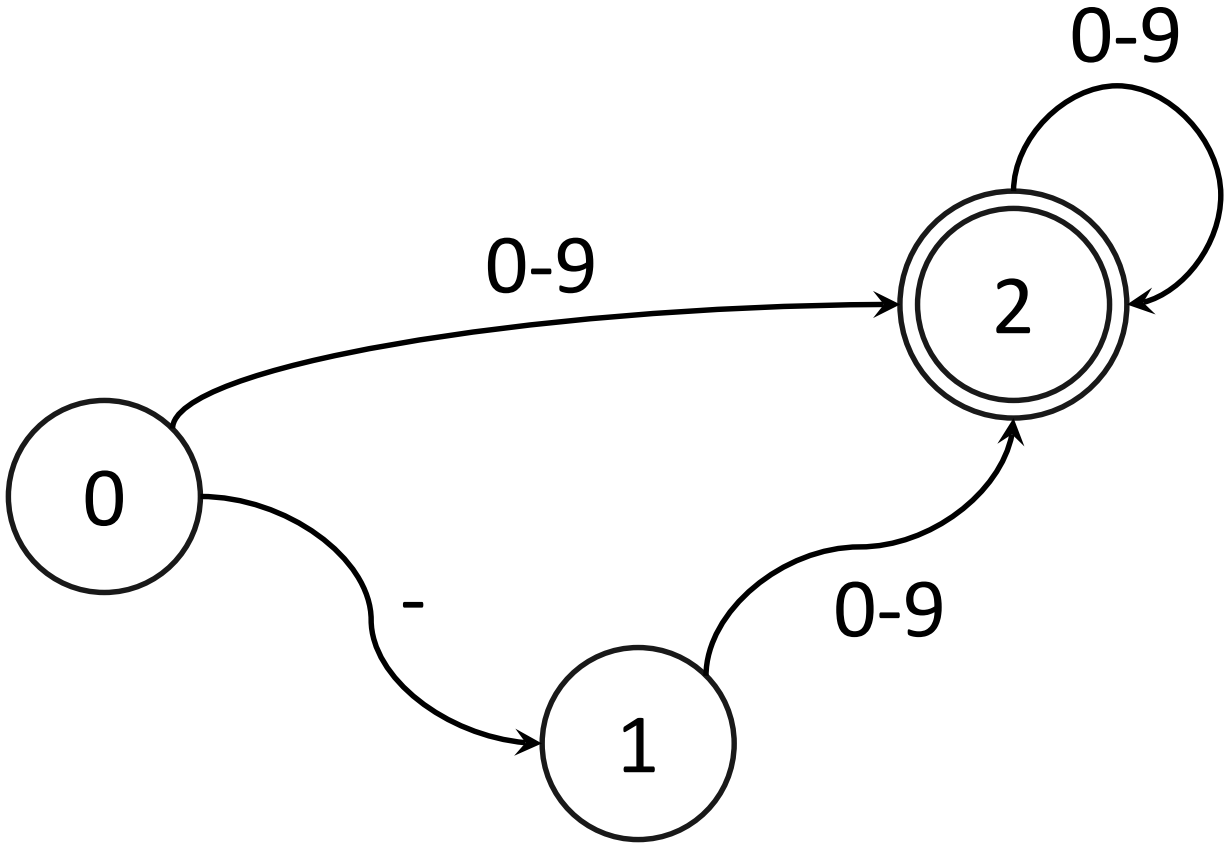
Regular grammar places severe restrictions on the production rules:

- A single nonterminal element allowed on the left side
- On the right side a rule is allowed to have:
  - nothing
  - a single terminal element
  - a single nonterminal element
  - a single terminal element and a single nonterminal element
- The order of nonterminal and terminal elements must remain the same within the grammar, it could be either:
  - nonterminal followed by terminal (left linear)
  - terminal followed by nonterminal (right linear)

Regular Language

- Language generated by regular grammar
- Language parser can be implemented using FSM
- Fast parsing
- Rules too simplistic (very poor support for nesting statements)
- Can be used as a scanner

Finite State Machine



Scanner / Lexer

- Scanner is a tool for matching the input for described patterns
- Lexer is a scanner for lexical elements of (some) language
- For example
  - 111 – numeric constant (integer)
  - 1.11 – numeric constant (real)
  - a11 – variable
  - "a11" – string constant

Regular Expressions (REGEXP)

- Wildcard characters instead of rules
- "+" – previous character (or character class) repeated 1 or more times
  - a+ – a, aa, aaa, etc
- "*" – previous character (or character class) repeated 0 or more times
  - ab* – a, ab, abb, etc
- "[]" – for describing character class
  - [abc] – a, b, c
- "-" – for character range
  - [a-z] – any lowercase latin character
- "^" – any character except the following character
  - [^abc] – any character except a, b, c

Regular Expressions (REGEXP)

- "." – any character except newline
- "|" – choice
  - a|b – a, b
- "{}" – string length
  - a{3} – aaa
  - a{2,4} – aa, aaa, aaaa
  - a{3,} – aaa, aaaa, aaaaa, etc
- to match a wildcard character it needs to be escaped
  - \+
  - \-
  - *

Flex

- Open source rewrite of AT&T tool lex
- Generates a scanner based on rules section
- Uses regular expressions for pattern matching
- Can be used as a lexer in conjunction with a parser
- Offers C functions and global variables for operating the scanner

Flex source file structure

- 4 sections:

  top

  definitions / options

  %%

  patterns / rules

  %%

  code

- Some sections may be empty

Flex file

```
%top{
#include "stdio.h"
}
%option case-insensitive
%option noyywrap
NUM    [0-9]
%x     STR
%%
{NUM}+       { printf("This looks like an integer: %d\n", atoi(yytext)); }
"\""         { BEGIN(STR); }
<STR>[^\"]+  { printf("quoted text: %s", yytext); }
<STR>"\""    { BEGIN(INITIAL); }
%%
int main(void){
       return yylex();
}
```

Flex: useful stuff

Options

- %option noyywrap – stops after scanning current file
- %option prefix="smth" – renames functions and variables from yy* to smth*
- %option case-insensitive – scanner is case insensitive
- %option yylineno – counts linenumbers, must have a rule for \n to function properly
- %option warn – prints warnings
- %option stack – allows explicit manipulation of states via a stack

Variables / Functions

- FILE *yyin – input file, by default set to stdin
- char *yytext – the currently matched input
- int yylineno – the line number in current input file
- int yylex() – starts the scanning process, returns when return statements present in rules or end of input
- BEGIN(); – macro for explicit machine state
- INITIAL – starting state
- unput() – puts character back into stream to be scanned next

Flex: more useful stuff

Explicit states

%x NAME – creates exclusive state NAME

%s NAME2 – creates inclusive state NAME2

<NAME>a – specifies that a rule is to be applied only when in state NAME

Pushdown stack manipulation

- yy_push_state() – pushes current state to top of stack and switches to supplied state
- yy_pop_state() – pops state from stack and switches to it
- yy_top_state() – returns the top state without altering stack

Flex: some command-line options

- -h – help
- -v – gives some information about the generated scanner
- -f – generates full character tables for each state
- -o – lets you specify filename for generated scanner
- -I – generates interactive scanner to be used from command-line
- -B – generates batch scanner to be used with files
- -7 – use 7 bit characters
- -8 – use 8 bit characters

Example:

```
flex -v -f test.l
```