CS 302 : Implementation of Programming Languages
TUTORIAL 1 (Lexical Analysis); Jan 16, 2017

P0. Identify manually the first 20 lexemes that make up the tokens in the following program. Give reasonable attribute values for the tokens.

```
int hashpjw(string s)
{
  unsigned h =0, g;
  for ( int i = 0; i < s.length() ; i++ )
  { cout << " before " << h << endl;
    h = ( h << 4 ) + ( s[i] );
    if ( g = h&0xf0000000 )
    {
      h = h ^ ( g << 24 );
      h = h ^ g;
    }
  }
  return h % PRIME;
 }
```

P1. Find the errors in the following Lex specification for floating point numbers, where digit denotes the ten decimal digits 0 through 9.

```
{digit}*\. {digit}*(e-{digit}+)?
```

Give a correct specification for floating point numbers.

P2. Write lex specification for recognizing single line comments in C/C++. The action required is to count the number of single line comments in a file.

P3. Lex specification for comments is given below.

```
"/*" [^*]* (\*( [^/ ]  [^*]* )? )* "*/"
```

Is the specification correct for all cases ? If your answer is yes, justify why it is so. Otherwise provide an instance(s) of a comment that is/are not captured by this specification. Make minimal changes in the given specification (with justification) to make it work for all cases.

P4. Strings are characters enclosed within " and ". A " enclosed in a string is represented as "". Write a regular expression for strings. You must produce argument to show why your regular expression is correct.

P5. Consider the following Lex-like specification of three distinct tokens :

```
(aba)+      Token 1
(a(b)*a)    Token 2
a(a | b)    Token 3
```

Manually construct a dfa that is able to recognize all the tokens. Indicate the token(s) found against each accepting state. Are there any clashes ?

P6. Consider the Lex-like specification of three distinct tokens of P5,

(aba)+ Token 1
(a(b)*a) Token 2
a(a | b) Token 3

(a) Draw the syntax tree and find nullable(), firstpos() and lastpos() sets
(b) Compute followpos() and then construct the DFA

P7. Consider the following lex-like description of the following two tokens :

a b Token 1
(a b)* c Token 2

(a) Construct a DFA, using any method of your choice, that recognizes both the tokens.
(b) Find the cost for complete tokenization (finding all tokens in the string) of the input string $(ab)^{10}$, assuming that each edge transition in the DFA incurs unit cost.

P8. This question concerns the four array representation scheme for a DFA. The function used is reproduced below. The minimal number of array accesses to find nextstate (s,a) for a state s appears to be 2 (assuming such a transition exists), which is required for accessing arrays CHECK and NEXT.

```
function nextstate (s, a)
{ if CHECK[BASE[s] + a] = s
  then  NEXT[BASE[s] + a]
  else return (nextstate( DEFAULT[s], a))
}
```

Construct a minimal DFA for which determination of nextstate(s,a) for some state s, requires at least 6 array accesses, if such is possible, else justify why such a DFA can not exist.

P9. Construct manually the 4 array based representation for the following DFA in terms of Default, Base, Next and Check such that array sizes are as low as possible.

States	**a**	**b**	***c***	***d***
0	5	4	-	-
1	1	-	-	1
2	-	-	6	5
3	-	-	-	2
4	5	4	-	1
5	2	3	4	4
6	5	4	-	-

P10. Construct its minimal dfa for the dfa of P9. The states {1, 2 } are given to be final states.

***** End of Tutorial Problem Sheet 1 *****

Support Material from Course Notes

1. Extended Regular Expressions

Expression	Describes	Example
c	Any non-metalinguistic character c	a
\c	Character c literally	*
"s"	String s literally	"**"
.	Any character but newline	a.*b
^	Beginning of a line	^abc
$	End of line	abc$
[s]	Any character in s	[abc]
[^s]	Any character not in s	[^abc]
[A-E]	Any character A through E	[a-c]
r*	Zero or more r's	a*
r+	One or more r's	a+
r?	Zero or one r	a?
r1 r2	r1 then r2	ab
r1 | r2	r1 or r2	a|b
(r)	r	(a | b)
r1 / r2	r1 when followed by r2	abc/123

2. Minimizing states of a dfa

Outline of an algorithm that minimizes the states of a given dfa is given below.

1. Construct an initial partition $\prod = \{ S - F, F_1, F_2, \ldots\ldots, F_n\}$, where $F = F_1 \cup F_2 \cup \ldots\ldots \cup F_n$ and each F_i is the set of final states for some token i.
2. for each set G in $\prod$ do
   partition G into subsets such that two states s and t of G are in the same subset if and only if
   for all input symbols a, states s and t have transitions onto states in the same set of $\prod$;
   replace G in $\pi\prod_{new}$ by the set of all subsets formed;
3. If $\pi\prod_{new} = \prod\pi$, let $\pi\prod_{final} := \prod$ and continue with step 4. Otherwise repeat step 2 with $\pi \prod := \pi\prod_{new}$.
4. Merge states in the same set of the partition.
5. Remove any dead (unreachable) states.

3. Four Arrays representation scheme for dfas

If s is a state and a is the numeric representation of a symbol, then

1. BASE[s] gives the base location for the information stored about state s.
2. NEXT[BASE[s] + a] gives the next state for s and symbol a, only if
   CHECK[BASE[s] + a] = s.
3. If CHECK[BASE[s] + a] ≠ s, then the next state information is associated with
   DEFAULT[s].

```
function nextstate (s, a)
{ if CHECK[BASE[s] + a] = s
  then  NEXT[BASE[s] + a]
  else
        return (nextstate( DEFAULT[s], a))
}
```

A heuristic, which works well in practice, to fill up the four arrays, is to find for a given state, the lowest BASE, so that the special entries of the state can be filled without conflicting with the existing entries.

4. Direct construction of deterministic finite automata from a regular expression

Let ∑ be the underlying alphabet. The rules for construction of the sets firstpos, lastpos and followpos is summarized in the following table. It is assumed that followpos(i) is initialized to ∅, ∀ i.

node n	***nullable(n)***	***firstpos(n)***	***lastpos(n)***	***followpos(i)***
Leaf labeled **ϵ**	**true**	∅	∅	
Leaf labeled with **a ∈** ∑ at position i	**false**	**{i}**	**{i}**	
$\mathbf{c_1 \mid c_2}$	nullable(c_1) **or** nullable (c_2)	firstpos(c_1) ∪ firstpos(c_2)	lastpos(c_1) ∪ lastpos(c_2)	
$\mathbf{c_1 \bullet c_2}$	nullable(c_1) **and** nullable (c_2)	**if** nullable(c_1) **then** firstpos(c_1) ∪ firstpos(c_2) **else** firstpos(c_1)	**if** nullable(c_2) **then** lastpos(c_1) ∪ lastpos(c_2) **else** lastpos(c_2)	**if** i ∈ lastpos(c_1) **then** followpos(i) += firstpos(c_2)
$\mathbf{c^*}$	**true**	firstpos()	lastpos(c)	**if** i ∈ lastpos(c) **then** followpos(i) += firstpos(c)
$\mathbf{c^+}$	nullable(c)	firstpos(c)	lastpos(c)	**if** i ∈ lastpos(c) **then** followpos(i) += firstpos(c)

Algorithm

1. Construct the tree for r# for the given regular expression r

2. Construct functions *nullable()*, *firstpos()*, *lastpos()* and *followpos()*
3. Let *firstpos(root)* be the start state. Push it on top of a stack.
   While (stack not empty)
   do begin
      pop the top state U off the stack; mark it;
      for each input symbol **a** do
      begin
         let p_1, p_2, p_3, ... p_k be the positions in U corresponding to symbol **a;**
         Let V = *followpos*(p_1) ∪ *followpos*(p_2) ∪ **.....** ∪ *followpos*(p_k);
         place V on stack if not marked and not already in stack;
         make a transition from U to V labeled **a**;
      end
   end
4. Final states are the states containing the positions corresponding to #.