Answers to the final examination on Compilers

Christian Rinderknecht

13 December 2005

Question. Consider the following Lex specification:

```
%{
#include<string.h>
char* keyword[] = {"else", "if", "return", "then"};
%}
num  (\.[0-9]+|[+\-]?[0-9]+(\.[0-9]+)?)(E[+\-]?[0-9]+)?
id   [A-Za-z]([_]*[A-Za-z0-9])*
ws   [ \n\t]+
%%
{num}  { printf ("num<%s>\n",yytext); }
{id}   { int index = 0;
         while (index <= 3
                && strcmp(keyword[index],yytext)) index++;
         if (index == 4) printf("id<%s>\n",yytext);
         else printf("kwd<%s>\n",yytext);
       }
{ws}   {}
"+"    { printf ("plus<>\n"); }
%%
```

Some possible transition diagrams corresponding to the regular expressions `num`, `id` and `ws` are respectively given in figure 1, 2 and 3.

Define the meaning of the pointers ⤴, ↾ and ⇑ presented in class and show how the input

```
return␣␣x+␣.5E2+y_0
```

where ␣ represents a blank character, is analysed using the transition diagrams. Make clear of which diagram you are referring to.

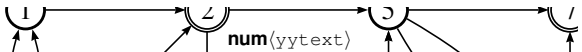
Answer.

- The pointer ⤴ points to the last character in the input buffer which was the current character when the current state was initial.
- The pointer ↾ points to the current available character in the input buffer.

Figure 1: Transition diagram for regular expression `num`

Figure 2: Transition diagram for regular expression `id`

Figure 3: Transition diagram for regular expression `ws`



- The pointer ⇑ points to the last character in the input buffer which was the current character when the current state was final.

The transition diagram of figure 1 is tried first.

The first available character is `r`, which cannot be read by `num`. Then the next regular expression, `id`, is tried. The processing is summarised in the following array, where the changes made at each step are shown in bold font.

States	r	e	t	u	r	n	␣	␣	x	+	␣	.	5	E	2	+	y	_	0
1 *initial*	↾																		
	↿↾																		
2 final	↿	↾																	
	↿	**⇑↾**																	
2 final	↿	⇑	↾																
	↿		**⇑↾**																
2 final	↿		⇑	↾															
	↿			**⇑↾**															
2 final	↿			⇑	↾														
	↿				**⇑↾**														
2 final	↿				⇑	↾													
	↿					**⇑↾**													
2 final	↿					⇑	↾												
	↿						**⇑↾**												

We are stuck in state 2, so we have recognised the lexeme `return` with the regular expression `id`. In order to determine the corresponding token, we need to check the keyword list `keyword`: `return` is in the list, so the token is `kwd`. We reset the ↿ to the value of ↾, i.e. it points to the first ␣, and start again the analysis by trying to match the regular expression `num` against the input. This fails. Therefore, the second regular expression, `id`, is tried. It fails to match the current character. Then the third regular expression, `ws`, is tried. One transition diagram for this regular expression, i.e. one recognising the same words, is

Therefore the analysis goes on in the following manner (states are taken in the transition diagram for regular expression `ws`).

States	r	e	t	u	r	n	␣	␣	x	+	␣	.	5	E	2	+	y	_	0
1							↿↾												
2 final							↿	↾											
							↿	**⇑↾**											
2 final							↿	⇑	↾										
							↿		**⇑↾**										

At that point, the analysis get stuck. There is no action associated to state 2 in the transition diagram of regular expression `ws`, therefore we restart the analysis

without emitting any lexeme. The pointer ⌊ is set to the value of pointer ⌈, i.e. it points now to the character `x`, and the regular expression `num` is tried first. It fails to recognise the current character, therefore the analysis restarts with the next regular expression, `id`. The analysis then proceeds as follows (states refers to the transition diagram of `id`).

States	r	e	t	u	r	n	␣	␣	x	+	␣	.	5	E	2	+	y	_	0
1									⌊⌈										
2									⌊	⌈									
final									⌊	⇑⌈									

Then the analysis gets stuck. Hence we emit the lexeme `x` and check for it in the keyword table. It is not present in the table, so it is to interpreted as an identifier, i.e. the token and lexemes are written **id**⟨`x`⟩. The analysis restarts again with resetting ⌊ to the same value as ⌈ and tries the first regular expression, `num`. This fails because ␣ (the current character) is not recognised by `num`. Therefore, the nex regular expression, `id`, is tried. Same problem.

The third expression, `ws`, succeeds: **TO BE FINISHED**