

Phase1 Report

<i>Mahmoud Abdel Latif</i>	<i>#64</i>
<i>Karim Nasr</i>	<i>#48</i>
<i>Shady Abdel Aziz</i>	<i>#28</i>
<i>Hassan Khalil</i>	<i>#22</i>

Data Structures:

Automata:

the structure of state is called vertex:

```
struct vertex {  
    typedef pair<string, vertex*> ve;  
    vector<ve> adj;           //symbol of transition, destination state  
    int name;                 //number of state  
    bool is_final;            //if final state  
    string id;                //id of automata of this state is final  
    vertex(int s) : name(s), is_final(false) {}  
}
```

the structure of the automata is called graph and it's a class:

```
class graph  
{  
    typedef map<int, vertex *> vmap;  
    vmap work;  
}
```

the work map holds the number of state and pointer to this state object

Parsing Lexical Rules:

map<string, string> : to hold all regular **definitions**.

map<string, string> : to hold all regular **expressions**
(tokens as keys and postfix as value)

vector<string> : to hold all **keywords**.

vector<string> : to hold all **punctuations**.

Building NFA:

stack<pair<int, int>>: to deal with postfix come from lexical rules parser.

Building DFA:

vector<DFA_state>

A list holding final states of the automaton, to be used during minimization.

vector<string>

A list holding all possible inputs of the automaton, to be used during minimization

-Some other data structures were used, like a stack to run DFS on the graph to find e-closure, a queue used during subset construction algorithm and some other maps and sets helping around.

Minimizing DFA:

int trans[][]: to hold transition table.

map<int, string>: the key is the number of final states and the value is type of this state

Matching:

set<string>

A set holding identifiers. This data structure represents the symbol table.

Algorithms and Techniques:

Parsing Lexical Rules:

parsing:

we are using regex to match every line in the file to distinguish between definitions, expression, keywords and punctuations. For every expression we build its postfix.

postfix algorithm

All variables, constants or numerals are added to the output expression. Left parentheses are always pushed onto the stack. When a right parenthesis is encountered, symbols on top of the stack are popped and copied to the resulting expression until the symbol at the top of the stack is a left parenthesis. At that time, parentheses are discarded.

If the symbol being scanned has a higher precedence than the symbol at the top of the stack, the symbol being scanned is pushed onto the stack and the scan pointer is advanced. If the precedence of the symbol being scanned is lower than or equal to the precedence of the symbol at the top of the stack, one element of the stack is popped to the output and the scan pointer is not advanced. So, the symbol being scanned will be compared with the new top element on the stack. When the end of the expression is encountered, the stack is popped to the output. If the top of the stack is a left parenthesis and the end of the expression is encountered, or a right parenthesis is scanned when there is no left one in the stack, the parentheses of the original expression were unbalanced and an unrecoverable error has occurred

Building NFA:

The main idea here is to create a graph called NFA and add the first state with number 0

```
NFA.addvertex(0)
```

makeAuto function is called for every regular expression to build its small NFA, then combine all small NFAs to one big NFA, we used this pseudocode:

for each expression e in the regular expression map came from lexical rules parser

```
int num_of_first_vertex = makeAuto(e.postfix, e.token)
NFA.addedge(0, num_of_first_vertex, epsilon)
```

makeAuto function returns the first vertex number to be used in combining NFAs. We used this pseudocode:

```
makeAuto(postfix, id)
```

```
stack<pair<int, int>> stk
```

```
stk.push(make_new_transition(postfix[0]))
```

```
for each symbol s in the postfix
```

```
if(s=="*")
```

```
    make automata of (*) and push it in the stk
```

```
else if(s=="?")
```

```

    make automata of (?) and push it in the stk
else if(s=="+")
    make automata of (+) and push it in the stk
else if(s=="@")
    make automaton of (concatenation) and push it in the stk
else if(s=="|")
    make automaton of (|) and push it in the stk
else
    stk.push(make_new_transition(s))
make last vertex final and save id in its object.
Return the number of first vertex

```

Building DFA:

Subset construction algorithm

Subset Construction (III)

```

initially,  $\epsilon$ -closure( $s_0$ ) is the only state in  $Dstates$  and it is unmarked;
while there is an unmarked state  $T$  in  $Dstates$  do begin
    mark  $T$ ;
    for each input symbol  $a$  do begin
         $U := \epsilon$ -closure(move( $T, a$ ));
        if  $U$  is not in  $Dstates$  then
            add  $U$  as an unmarked state to  $Dstates$ ;
         $Dtran[T, a] := U$ 
    end
end

```

Fig. 3.25. The subset construction.

DFS for E-closure

Subset Construction (IV)

(ϵ -closure computation)

```

push all states in  $T$  onto  $stack$ ;
initialize  $\epsilon$ -closure( $T$ ) to  $T$ ;
while  $stack$  is not empty do begin
    pop  $t$ , the top element, off of  $stack$ ;
    for each state  $u$  with an edge from  $t$  to  $u$  labeled  $\epsilon$  do
        if  $u$  is not in  $\epsilon$ -closure( $T$ ) do begin
            add  $u$  to  $\epsilon$ -closure( $T$ );
            push  $u$  onto  $stack$ 
        end
    end
end

```

*Minimizing
DFA:*

Fig. 3.26 Computation of ϵ -closure

Algorithm: Minimizing the number of states of a DFA

- **Input.** A DFA M with set of states S , set of inputs Σ , transitions defined for all states and inputs, start state s_0 , and a set of accepting states F .
- **Output.** A DFA M' accepting the same language as M and having as few states as possible.
- **Method.**
 1. Construct an initial partition Π of the set of states with two groups: the accepting states F and non-accepting states $S - F$.
 2. Partition Π to Π_{new} .
 3. If $\Pi_{new} = \Pi$, let $\Pi_{final} = \Pi$ and go to step (4). Otherwise, repeat step (2) with $\Pi := \Pi_{new}$.
 4. Choose one state in each group of the partition Π_{final} as the *representative* for that group.
 5. Remove dead states.

Construct New Partition

```
for each group  $G$  of  $\Pi$  do begin
    partition  $G$  into subgroups such that two states  $s$  and  $t$ 
    of  $G$  are in the same subgroup if and only if for all
    input symbols  $a$ , states  $s$  and  $t$  have transitions on  $a$ 
    to states in the same group of  $\Pi$ ;
    /* at worst, a state will be in a subgroup by itself */
    replace  $G$  in  $\Pi_{\text{new}}$  by the set of all subgroups formed
end
```

Fig. 3.45. Construction of Π_{new} .

Matching:

The general algorithm is: tokens are read one by one from the source file. For each token read, it's scanned character by character, and each character moves the DFA from one state to another. The algorithm tries to find the longest possible match, when it's found, it's returned, then the whole process starts again.

Error recovery: If no match is found, one character is skipped at a time, trying to find a match greedily. All characters skipped during matching are held together to be included in an error message generated by the analyzer.

Transition Table:

It's printed in file called trans.txt in project folder.

The resultant stream of tokens for the example test program:

```
int  
id  
,  
id  
,  
id  
,  
id  
;  
while  
(  
id  
relop  
num  
)  
{  
id  
assign  
id  
addop  
num  
;  
}
```

Assumptions:

The input source file must not contain spaces, tabs or empty lines in the end of file.

Lex Tool (Bonus Part)

Lex is short for "lexical analysis". Lex takes an input file containing a set of lexical analysis rules or regular expressions. For output, Lex produces a C function which when invoked, finds the next match in the input stream.

1. Format of lex input:

(beginning in col. 1) declarations

%%

token-rules

%%

user program

2. Declarations:

a) string sets; name character-class

b) standard C; %{ -- c declarations --
 %}

3. Token rules: regular-expression { optional C-code }

a) if the expression includes a reference to a character class, enclose the class name in brackets { }

b) regular expression operators;

*, + --closure, positive closure

" " or \ --protection of special chars

| --or

^ --beginning-of-line anchor

() --grouping

\$ --end-of-line anchor

? --zero or one

. --any char (except \n)

{ref} --reference to a named character class (a definition)

[] --character class

[^] --not-character class

4. Match rules: Longest match is preferred. If two matches are equal length, the first match is preferred. Remember, lex partitions, it does not attempt to find nested matches. Once a character becomes part of a match, it is no longer considered for other matches.

7. Example lex file

```
/* scanner for a toy Pascal-like language */

%{
/* need this for the call to atof() below */
#include <math.h>
%}

DIGIT    [0-9]
ID       [a-z][a-z0-9]*

%%

{DIGIT}+ {
    printf( "An integer: %s (%d)\n", yytext,
            atoi( yytext ) );
}

{DIGIT}+"."{DIGIT}* {
    printf( "A float: %s (%g)\n", yytext,
            atof( yytext ) );
}

if|then|begin|end|procedure|function {
    printf( "A keyword: %s\n", yytext );
}

{ID}      printf( "An identifier: %s\n", yytext );

"+"|"-"|"*"|"|"  printf( "An operator: %s\n", yytext );

"{"[^}\n]*"}"    /* eat up one-line comments */

[ \t\n]+        /* eat up whitespace */

.              printf( "Unrecognized character: %s\n", yytext );

%%
```

8. Execution of lex: (to generate the yylex() function file and then compile a user program)

(MS) c:> flex sample.lex

c:> gcc lex.yy.c -lfl

c:> ./a.out

flex produces lex.yy.c

(Linux) \$ lex sample.lex

\$ gcc lex.yy.c -lfl

\$./a.out

lex produces lex.yy.c

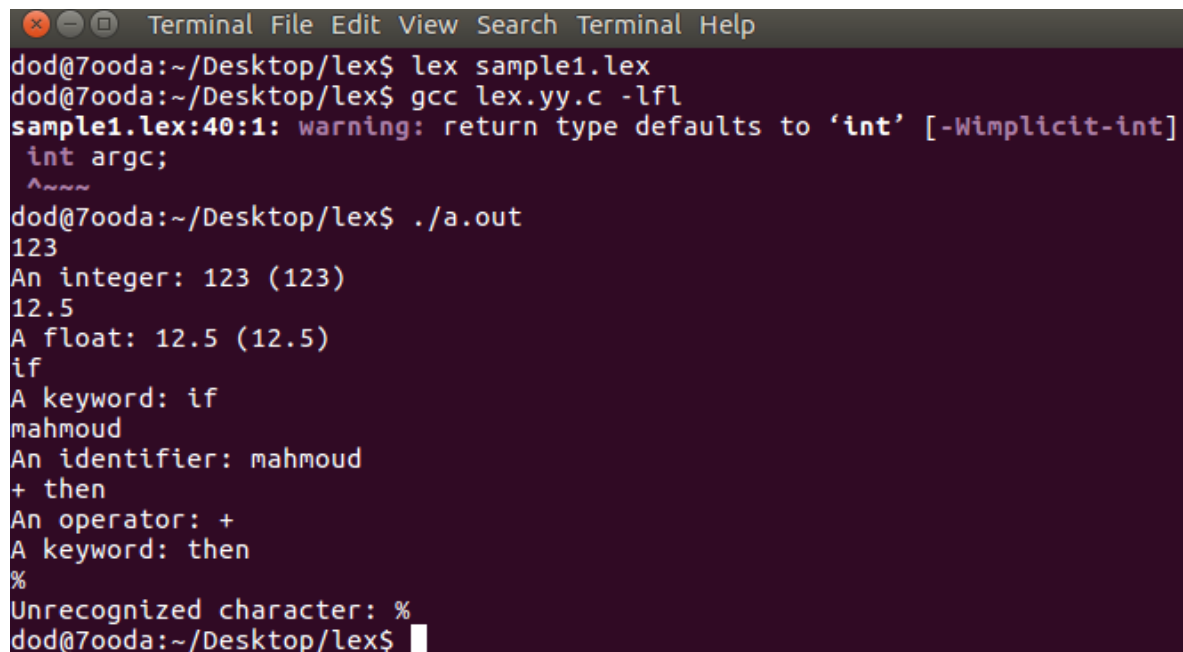
The produced .c file contains this function: `int yylex()`

9. User program:

(The above scanner file must be linked into the project)

```
%%  
  
main( argc, argv )  
{  
    int argc;  
    char **argv;  
    {  
        ++argv, --argc; /* skip over program name */  
        if ( argc > 0 )  
            yyin = fopen( argv[0], "r" );  
        else  
            yyin = stdin;  
    }  
    yylex();  
}
```

10. Test cases and screenshots:



```
Terminal File Edit View Search Terminal Help  
dod@7ooda:~/Desktop/lex$ lex sample1.lex  
dod@7ooda:~/Desktop/lex$ gcc lex.yy.c -lfl  
sample1.lex:40:1: warning: return type defaults to 'int' [-Wimplicit-int]  
    int argc;  
    ^~~~~  
dod@7ooda:~/Desktop/lex$ ./a.out  
123  
An integer: 123 (123)  
12.5  
A float: 12.5 (12.5)  
if  
A keyword: if  
mahmoud  
An identifier: mahmoud  
+ then  
An operator: +  
A keyword: then  
%  
Unrecognized character: %  
dod@7ooda:~/Desktop/lex$
```