SET - 2

COMPILER DESIGN

ASSIGNMENT - 1

Short Questions

1. Enumerate Major Data structures of Compiler

A. Principal DS that are needed by the phases as part of their operation and that serve to communicate information among the phases

Tokens → Scanner collects char and generate tokens

Syntax tree → Each Syntax tree node itself may require diff attribut to be stored in symbol table

Symbol table

literal table

Intermediate Code

temporary files

Symbol Table :

1. Their DS keeps information associated with identifiers, function variables, constants and data types.

2. The symbol table interacts with almost every phase of the compiler

3. The scanner, parser or semantic analyzer may enter Identifi into the table.

4. The semantic analyzer will add data types and other info. provided by the symbol table to make appropriate object code choices.

Literal Table :

1. It stores constants and strings used in a program.

2. Literal table is important in reducing the size of a prog. in memory by allowing the reuse of constants and strings

3. It is also needed by the code generator to construct symbolic addresses for literals and for entering data definations in the target code file.

Intermediate Code :

1. Depending on the kind of Intermediate code (three-addr-code) and the kinds of optimizations performed, this code may be kept as an array of text strings, a temp text file,

or as a linked list of struct.

Temporary files :

1. Keeping only enough information from earlier parts of the source program to enable translation to proceed.

---

2. Depict General format for LEX program

A 1. Lex is a tool that allows one to specify a lexical Analyzer by specifying R.E to describe patterns for tokens.

2. I/p notation for the Lex tool is referred to as the Lex language and the tool itself is the Lex Compiler

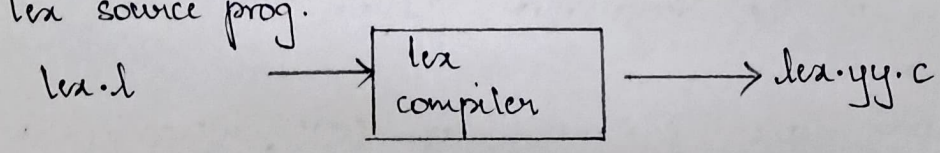
3. lex Compiler transforms the I/p patterns into a transition diagram and generate code, in a file called lex.yy.c

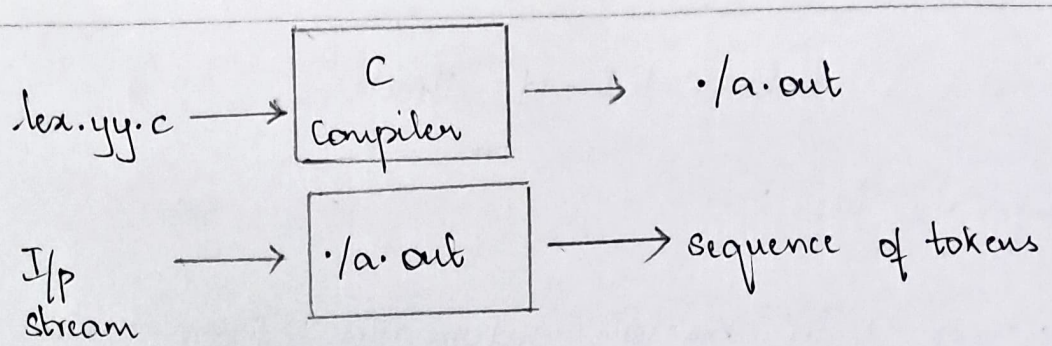
→ Use of lex

→ Structure of lex programs

→ Conflict Resolution in lex

Use of Lex :





Structure of lex programs :

1. A lex program has the following form :

```
%{
    declaration
%}
%%
    rules
%%
    C fns / auxiliary fns.
```

Declaration Section:

Includes declarations of variables, constant and regular definitions.

Transition rules :

pattern { Action }

1. Each pattern is a R.E, which may use the regular definitions of the declaration section

2. The actions are fragments of code, typically written in C

C functions / Auxiliary fns:

1. This sections holds whatever additional fns are used in the actions.
2. These fns can be compiled separately and loaded with the lexical Analyzer.

Conflicts Resolution in lex:

* Two rules that lex uses to decide on the proper lexeme to select, when several prefixes of the I/p match one or mor patterns.

1. Always prefer a longer prefix to a shorter prefix.

   <, <= ⇒ prefer <=

2. If the longest possible prefix matches two or more patter prefer the pattern listed first in the lex program.

a	then	b
↓	↓	↓
id	kw	id

---

3. Eliminate Left Recursion from

   E -> E+T | T

   T -> T*F | F

4. Consider the Grammar

$E \rightarrow E+T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

$E \rightarrow E+T \mid T$

$A \rightarrow A\alpha \mid \beta$

then

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

$A = E$

$\alpha = +T$

$\beta = T$

then rule becomes,

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

Similarly for the rule,

$T \rightarrow T * F \mid F$

We can eliminate left recursion as

$T \rightarrow FT'$

$T' \rightarrow * FT' \mid \varepsilon$

So the equivalent grammar will be

$E \rightarrow TE'$

$E \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow * FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Long Questions

4. Create Recursive Descent Parsing with an example.

$E \rightarrow num\ T$

$T \rightarrow * num\ T$

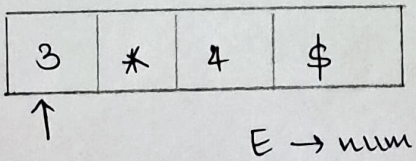
A: The parser should be activated by a procedure corresponding to the start symbol.

$E \rightarrow num\ T$

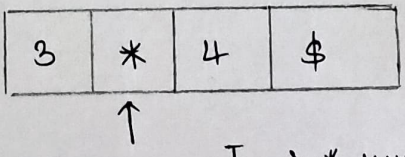
$T \rightarrow * num\ T \mid \varepsilon$

```
procedure E
{
   if lookhead = num then
      {
         match (num);
          T;
      }
   else
         error;
   if lookhead = '$'
     {
        declare success;
     }
   else
        error;
procedure T
{
    if lookhead = '*'
       {
        match ('*');
   if lookhead = 'num'
      {
       match ('num');
         T;
      }
```

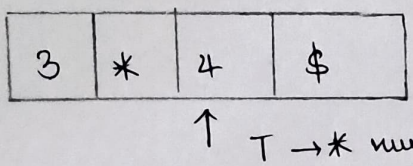
```
        else
                Error;
    }
    Else
                NULL;
}
Procedure match (token t)
{
    if lookhead = t
    lookhead = next_token;
    else
        Error
}
Procedure Error
{
  print ("Error!!");
}
```



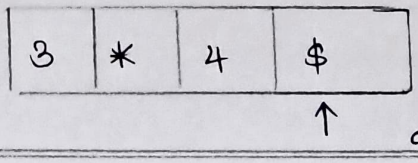
E → num T



T → * num T



T → * num T



declare Success

5. Construct SLR parse table for the following grammar

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

A. STEP 1:- CONSTRUCTION OF CANONICAL COLLECTION OF SET OF ITEM:

In this grammar we will add the Augmented grammar $E' \rightarrow .E$ in the I then we have to apply closure (I)

I_0 :

$E' \rightarrow .E$

$E \rightarrow .E+T$

$E \rightarrow .T$

$T \rightarrow .T*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

Applying goto on I_0

goto (I_0, E)

I_1 :

$E' \rightarrow E.$

$E \rightarrow E.+T$

goto (I_0, T)

I_2 :

$E \rightarrow T.$

$T \rightarrow T.*F$

goto (I_0, F)

I_3 :

$T \rightarrow F.$

goto $(I_0, ()$

I_4 :

$F \rightarrow (.E)$

goto (I_0, id)

I_5 :

$F \rightarrow id.$

Applying goto on I_0 is completed.

→ Now applying goto on I_1

goto $(I_1, +)$

I_6 :

$E \rightarrow E + .T$

$T \rightarrow .T * F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

Applying goto on I_2

goto $(I_2, *)$

I_7 :

$T \rightarrow T * .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

Applying goto on I_3

The production in I_3 is a kernal item so, we can't apply goto

Applying goto on I_4

goto (I_4, E)

I_8 :

$F \rightarrow (E \cdot)$

$E \rightarrow E \cdot + T$

(I_4, T)	$E \rightarrow T.$	I_2
(I_4, T)	$E \rightarrow T \cdot * F$	I_2
(I_4, F)	$T \rightarrow F.$	$\rightarrow I_3$
$(I_4, ()$	$F \rightarrow (\cdot E)$	$\rightarrow I_4$
(I_4, id)	$F \rightarrow id$	$\rightarrow I_5$

→ Repetition of states

Applying goto on I_5

the production in I_5 is a kernel item so, cant apply got

Applying goto on I_6

goto (I_6, T)

I_9 :

$E \rightarrow E + T \cdot$

$T \rightarrow T * * F$

(I_6, F)	$T \rightarrow F.$	$- I_3$
$(I_6, ()$	$F \rightarrow (\cdot E)$	$- I_4$
(I_6, id)	$F \rightarrow id.$	$- I_5$

→ Repetition of states.

Applying goto on I_7

goto (I_7, F)

I_{10} :

$T \rightarrow T * F.$

$(I_7, ()$	$F \rightarrow (.E)$	$- I_4$
(I_7, id)	$F \rightarrow id.$	$- I_5$

→ repetition of states

Applying goto on I_8

goto $(I_8,))$

I_{11} :

$F \rightarrow (E).$

$(I_8, +)$	$E \rightarrow E + .T$	$- I_6$

→ repetition of states.

STEP-2 : CONSTRUCTION OF SLR PARSING TABLE

States	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Accept			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S5			S4			8	2	3
5		r6	r6		r6	r6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

STEP 3: PARSING THE I/P STRING

Stack	I/p buffer	Parsing Action
$0	id * id + id	Shift id
$0id5	* id + id	reduce F → id
$0F3	* id + id	reduce T → F
$0T2	* id + id	Shift *
$0T2*7	id + id	Shift id
$0T2*7id5	+ id	reduce F → id
$0T2*7F10	+ id	reduce T → T * F
$0T2	+ id	reduce E → T
$0E1	+ id	Shift +
$0E1+6	id	Shift id
$0E1+6id5	$	reduce F → id
$0E1+6F3	$	reduce T → F
$0E1+6T9	$	reduce E → E+T
$0E1	$	Accept.