Name: Adwait Hegde
Roll No: 2019130019
TE Comp (Batch-A)

EXPERIMENT 6

Aim:

Syntax directed translation using YACC

Theory:

Syntax-directed Translation

Syntax-directed translation is essentially a formal treatment of two simple concepts:

- each node of a parse tree can have a set of associated values (attributes).
- the creation and traversal of parse trees can be automated.

Attributes are essentially equivalent to the %union value that yacc can calculate in the action associated with each grammar rule - we can use struct values if there is more than one attribute. However, the actions themselves are generalised - yacc values can only move from leaves of the parse tree towards the root, but in syntax-directed translation the values can move around the tree in any way the user requires. This is achieved by automatically creating a parse tree and traversing it as required to move the values around.

The automatic creation and use of parse trees also simplifies one more step in the translation process, as we often need to do this one way or another.

Systems like yacc and lex exist that can generate syntax-directed translators, but they are inevitably harder to learn than yacc and lex, although like any other high-level language, experienced users can achieve a lot more. However, just as with low-level languages, with sufficient ingenuity, yacc and lex do anything that the more complex systems can do. (Note that some of the tools being created today, such as JavaCC, if anything have less functionality than lex and yacc.) More importantly, by becoming aware of these more advanced concepts, we can make better use of lex and yacc.

When to use a tool or technique?

The questions relevant to the use of lex, yacc, and dictionaries are all about the form of the input text. By contrast, the most important questions relevant to the use of parse trees are about what you want to do with the input text once you have analysed it.

lex

Does the text you are processing have any structure at all?

If so, you will usually want to split it into small pieces (words), so you will almost always use lex (or something equivalent).

The only other situation I can imagine when it would not be useful would be e.g. trying to deal with FORTRAN, which has a structure, but the boundaries between words are decided by the context they appear in (spaces are not separators, so ``DO 1 I'' might be 3 words in one place and 1 word in another).

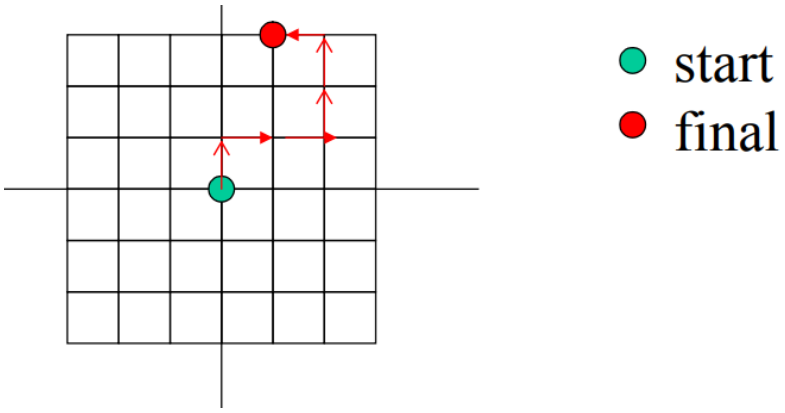
yacc
Does the text you are processing have a hierarchy of structures e.g. do the words combine into groups, which combine into larger groups, and so on?
Is this hierarchical structure well-defined (i.e. has a grammar), or can you create a grammar that accurately captures the required structure?
If so, then yacc should be able to help you.

However, one thing that yacc is not very good at is coping with mistakes in its input. Most modern systems have some facilities to automatically deal with errors, e.g. by skipping or inserting a word, or in more serious cases by jumping to the end of the current grammar rule and skipping input until it finds something that can follow it. In yacc (and in JavaCC) although there are some helpful facilities, they must inserted by hand at suitable points in the grammar.

Implementation

Given a starting location of 0,0 and a sequence of north, south, east, west moves (ex: NEENNW), find the final position on a unit grid.

Print the final position using using **YACC**. Use concept of Syntax directed translation



in this case output can be : (1 , 3)

Consider the grammar as: S --> S D

S --> START

D --> n | s | e | w

If input string is : START n e e n n w

o/p : Final position is (1 , 3)

If input is : START n e b b b

o/p : Invalid string

Code:

e5.l

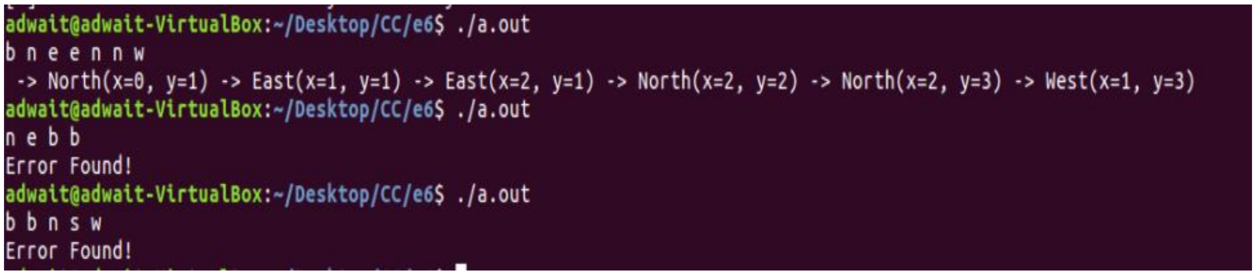
```
%{
#include <stdio.h>
#include "y.tab.h"

struct Loc{
    int x;
    int y;
};

%}


%%

[b]     {return B;}
[n]     {return N;}
[e]     {return E;}
[s]     {return S;}
[w]     {return W;}
[ \t]   {;}
\n      {return 0;}
.       {yyerror("Unexpected Input from the user"); }

%%


int yywrap(void){return 1;}
```

e5.y

```
%{
#include<stdio.h>
#include<stdlib.h>
//void yyerror(char *s);

struct Loc{
    int x;
    int y;
};

int yylex();
void yyerror();

%}

%token N S E W B
%start ST
%union {struct Loc *point;}
%type <point> ST D


%%

ST: ST D    {$$=(struct Loc*)malloc(sizeof(struct Loc));
             $$->x = $1->x + $2->x;
             $$->y = $1->y + $2->y;
             printf("(x=%d, y=%d)", $$->x, $$->y);}
| B         {$$=(struct Loc*)malloc(sizeof(struct Loc));
             $$->x = 0; $$->y = 0;}
;

D: N        {$$=(struct Loc*)malloc(sizeof(struct Loc));
             $$->x = 0; $$->y = 1;
             printf(" -> North");}
| S         {$$=(struct Loc*)malloc(sizeof(struct Loc));
             $$->x = 0; $$->y = -1;
             printf(" -> South");}
| W         {$$=(struct Loc*)malloc(sizeof(struct Loc));
             $$->x = -1; $$->y = 0;
             printf(" -> West");}
| E         {$$=(struct Loc*)malloc(sizeof(struct Loc));
             $$->x = 1; $$->y = 0;
             printf(" -> East");}
;
```

```
%%

int main() {
    yyparse();

    return 0;
}
void yyerror(char *s) {
    printf("Error Found!\n");
    exit(0);
}
```

Result:



```
adwait@adwait-VirtualBox:~/Desktop/CC/e6$ ./a.out
b n e e n n w
 -> North(x=0, y=1) -> East(x=1, y=1) -> East(x=2, y=1) -> North(x=2, y=2) -> North(x=2, y=3) -> West(x=1, y=3)
adwait@adwait-VirtualBox:~/Desktop/CC/e6$ ./a.out
n e b b
Error Found!
adwait@adwait-VirtualBox:~/Desktop/CC/e6$ ./a.out
b b n s w
Error Found!
```

Conclusion:
In this Experiment, I learned about the concept of Syntax directed translation using YACC. Was able to recollect the concept about structures in C

Ref.:
https://docs.google.com/document/d/1q8aWfuOkrUj_bVeONr3HnqXuw5hTtlS_FRFiyBlhKxs/edit