CS39003 Compilers Laboratory
Autumn 2025
Assignment 6
Date of posting: 13-Oct-2025

Yacc programming with inherited attributes (continued)

This assignment is a continuation of Assignment 5. Here, you add structures to the list of variable types. The definition of a structure involves the specification of the component fields. This specification may be optionally followed by a list of variables. One may also use a structure type defined earlier in the program to introduce new variables of that structure type. We reproduce the grammar of Assignment 5 below. The three new productions that deal with structures are highlighted. The terminal symbols are in color red.

```
PROG     →  DECLIST
DECLIST  →  DECLIST DECL | DECL
DECL     →  BASIC VARLIST ; | struct id { DECLIST } ; | struct id { DECLIST } VARLIST ; | struct id VARLIST ;
BASIC    →  void | char | unsigned char | short | short int | unsigned short | unsigned short int |
            int | unsigned | unsigned int | long | long int | unsigned long | unsigned long int | float | double
VARLIST  →  VARLIST , VAR | VAR
VAR      →  * VAR | id DIM
DIM      →  [ num ] DIM | ε
```

Here is an example of a set of declarations generated by PROG (or DECLIST).

```
char letter, alphabet[26];
unsigned long int a, *p, **q, ***r, A[5], B[6][7], **C[2][3][4];
void *vptr;

struct stud {
   char roll[10];
   char name[101];
   short int yob;
   float CGPA;
   char gender;
} FooBar;
struct stud BTech[200], Dual[100], MTech[150], MS[100], PhD[250];

struct listnode {
   struct coordinates {
      int x, y, z;
   } point;
   struct listnode *next, *prev;
};

struct listnode *head, *tail;
```

Modify your code for Assignment 5 to incorporate structures. The incremental work to be carried out is explained below.

Lex file

Use your last `ctype.l` with two additions. Now, struct is a new keyword introduced. Moreover, you need to handle the new punctuation symbols: { and }

Type table

The global type table `TT` stores the basic data types, and the array and pointer types as in Assignment 5.

The declaration of each structure introduces a new name space (a new symbol table). We use only named structures. The type table will store an entry for each structure. The category of that entry is `STRUCTURE`, and the reference is the serial number of the symbol table for that structure. Moreover, we need to store the structure name in that `TT` entry. This name is not to be confused with the variable names of that structure type. Consider `struct stud` as defined above. There will be a single entry in `TT` for this structure, and the name field will store `stud`. The type of the variable `FooBar` will be this entry in `TT`. The subsequent declarations of the student arrays will insert the following new array types in `TT`:

`array(200,struct stud)`, `array(100,struct stud)`, `array(150, struct stud)`, and `array(250,struct stud)`.

The types of the arrays `Dual` and `MS` are the same: `array(100,struct stud)`.

Allow structure definitions to be nested. In the above example, `struct coordinates` is declared inside `struct listnode`. Any structure declaration is to be treated globally, that is, `struct coordinates` can be used to declare variables outside the scope of `struct listnode`. Allow pointers to a structure being defined, inside that structure (like `next` and `prev` in the example). However, non-pointer references (variables or arrays) to a structure cannot be allowed until that structure is fully defined.

The calculation of the width of a structure uses the same ideas as that of the symbol table storing global variables. We continue to enforce that all variables are 4-byte aligned, that is, start at locations in their respective name spaces (the global data segment or the memory of a structure), that are multiples of 4. As an example, consider the following structure declared earlier.

```
struct stud {
   char roll[10];
   char name[101];
   short int yob;
   float CGPA;
   char gender;
} FooBar;
```

The variable `FooBar` will start at an address which is a multiple of 4. The individual components of the structure will also be 4-byte aligned. A 9-letter roll number with a trailing null character is stored in a char array `roll` of size 10, and occupies memory locations 0 – 9 relative to the beginning of the structure. The next array `name` starts at the next multiple of 4, that is, from offset 12 inside the structure, and will occupy memory locations 12 – 112. The third entry is a `short int` requiring 2 bytes. It will occupy memory locations 116 – 117 (not 113 – 114 immediately after `name`). Likewise, `CGPA` will occupy locations 120 – 123, and `gender` 124 – 124. The structure uses (relative) memory locations 0 – 124, that is, 125 bytes. But its width must be a multiple of 4, that is, the width stored in `TT` against `struct stud` will be 128.

We now tabulate the fields of each entry in `TT`. Each blank entry in the following table means that that field is not used. The last row is added to the same table of Assignment 5.

Category	Dimension	Reference	Width	Name
VOID			0	
UCHR			1	
…				
DBL			8	
ARRAY	Number of elements	Index of the element type	Calculated	
POINTER		Index of the element type	Pointer size (usually 8)	
STRUCTURE		Number of symbol table	Calculated	Structure name

Design your own implementation (array of records) for the global type table `TT`. Use linear search in that array. There is no need to go for an efficient data structure to be designed by you or available in any ready-made library.

Symbol tables

Maintain several symbol tables. The main symbol table stores all the names (and associated information) of the variables declared outside any structure. Each structure, on the other hand, has its own symbol table storing the variables declared in the list for that structure. Each entry in each symbol table will again be a triple (`name`, `type`, `offset`), where `name` is the user-given name of the component (field), `type` is a reference to (index in) the entry in the type table `TT`, storing the type of that component, and `offset` is the starting location (a multiple of 4) of the component in its symbol table.

Design your own implementation for the symbol tables. Again you do not have to go for any efficient data structures. Instead use a two-dimensional array of (`name`, `type`, `offset`) triples. The outer dimension (row) stores the number of the symbol table. The main symbol table is at row 0, and the subsequent rows are for the structures. Each row will store information of the variables appearing in that name space. Whenever needed, make linear search (for a `name`) in each symbol table.

Also modify the function to insert a variable in a symbol table. This function would now require three arguments: the name of the variable, the number (row index) of the symbol table, and the type of the variable (an index in `TT`). The offset of that variable to be stored as a triple (`name`, `type`, `offset`) is to be computed by the insertion function. A global array (no longer a single variable) is to be used to store the current widths (maintained as multiples of 4) of all the symbol tables. Disallow duplicate names of two variables in each symbol table. Names may however be shared by variables in different symbol tables.

Yacc file

Implement the grammar given at the beginning of this assignment (and no other grammar). You only need to insert appropriate marker non-terminals in the appropriate places of the productions. The types and the widths of types, and the offsets of the variables are calculated and stored globally as in Assignment 5.

Now you need to use additional inherited attributes (and the associated marker non-terminals). Each DECLIST must be associated with the number of the symbol table where all variables declared in that name space must be inserted. A DECL may involve the definition of a struct type, which itself has its own DECLIST. This DECLIST must receive the next symbol-table number. A DECL using a struct type defined earlier, uses the symbol-table number for that structure (stored in the TT entry for that structure). Use a marker non-terminal before each DECLIST to store this symbol-table number. This symbol-table number must move down the parse tree to all the variables declared in that name space. You need this number in the insert function for that symbol table. It is illegal to use an undefined structure name to declare variables of that structure type. A search in TT can detect this error.

There is another situation where you need a marker non-terminal. At the beginning of the DECLIST in a struct definition, an entry for that structure type is created in TT. This DECLIST is stored in a new symbol table. However, the width of that symbol table is also to be stored in the TT entry for that structure. It is correctly computed after that DECLIST is completely read, and shifted to the top of the parse stack. Use a marker non-terminal after the DECLIST, to store the final width of the structure in its entry in TT.

Printing the tables

After all declarations are read, print the type table TT, and the details of the variables stored in **all** the symbol tables. Use the format as explained in the section “Sample Output” below. Write (update) suitable functions for doing these tasks.

What to submit

Write a makefile with compile, run, and clean targets. Pack you lex file, your yacc file, the makefile, and any other source or header file that you use (**not** `lex.yy.c`, `y.tab.h` or `y.tab.c`) in a single zip/tar/tgz archive. Submit that archive only.

Sample Output

For the declarations given on the first page (stored in the file `decl.c`), the output follows.

```
$ make run
yacc -d ctype.y
lex ctype.l
gcc y.tab.c lex.yy.c
./a.out decl.c
+++ All declarations read

+++ 32 types
    Type   0:        0    void
    Type   1:        1    unsigned char
    Type   2:        1    char
    Type   3:        2    unsigned short
    Type   4:        2    short
    Type   5:        8    unsigned long
    Type   6:        8    long
    Type   7:        4    unsigned int
    Type   8:        4    int
    Type   9:        4    float
    Type  10:        8    double
    Type  11:       26    array(26,char)
    Type  12:        8    pointer(unsigned long)
    Type  13:        8    pointer(pointer(unsigned long))
    Type  14:        8    pointer(pointer(pointer(unsigned long)))
    Type  15:       40    array(5,unsigned long)
    Type  16:       56    array(7,unsigned long)
    Type  17:      336    array(6,array(7,unsigned long))
    Type  18:       32    array(4,pointer(pointer(unsigned long)))
    Type  19:       96    array(3,array(4,pointer(pointer(unsigned long))))
    Type  20:      192    array(2,array(3,array(4,pointer(pointer(unsigned long)))))
    Type  21:        8    pointer(void)
    Type  22:      128    struct stud with symbol table 1
    Type  23:       10    array(10,char)
    Type  24:      101    array(101,char)
    Type  25:    25600    array(200,struct stud with symbol table 1)
    Type  26:    12800    array(100,struct stud with symbol table 1)
    Type  27:    19200    array(150,struct stud with symbol table 1)
    Type  28:    32000    array(250,struct stud with symbol table 1)
    Type  29:       28    struct listnode with symbol table 2
    Type  30:       12    struct coordinates with symbol table 3
    Type  31:        8    pointer(struct listnode with symbol table 2)

+++ Symbol table 0 [main]
    letter               0 - 0          type =    2 = char
    alphabet             4 - 29         type =   11 = array(26,char)
    a                   32 - 39         type =    5 = unsigned long
    p                   40 - 47         type =   12 = pointer(unsigned long)
    q                   48 - 55         type =   13 = pointer(pointer(unsigned long))
    r                   56 - 63         type =   14 = pointer(pointer(pointer(unsigned long)))
    A                   64 - 103        type =   15 = array(5,unsigned long)
    B                  104 - 439        type =   17 = array(6,array(7,unsigned long))
    C                  440 - 631        type =   20 = array(2,array(3,array(4,pointer(pointer(unsigned long)))))
    vptr               632 - 639        type =   21 = pointer(void)
    FooBar             640 - 767        type =   22 = struct stud with symbol table 1
    BTech              768 - 26367      type =   25 = array(200,struct stud with symbol table 1)
    Dual             26368 - 39167      type =   26 = array(100,struct stud with symbol table 1)
    MTech            39168 - 58367      type =   27 = array(150,struct stud with symbol table 1)
    MS               58368 - 71167      type =   26 = array(100,struct stud with symbol table 1)
    PhD              71168 - 103167     type =   28 = array(250,struct stud with symbol table 1)
    head            103168 - 103175     type =   31 = pointer(struct listnode with symbol table 2)
    tail            103176 - 103183     type =   31 = pointer(struct listnode with symbol table 2)
    Total width = 103184

+++ Symbol table 1 [struct stud]
    roll                 0 - 9          type =   23 = array(10,char)
    name                12 - 112        type =   24 = array(101,char)
    yob                116 - 117        type =    4 = short
    CGPA               120 - 123        type =    9 = float
    gender             124 - 124        type =    2 = char
    Total width = 128

+++ Symbol table 2 [struct listnode]
    point                0 - 11         type =   30 = struct coordinates with symbol table 3
    next                12 - 19         type =   31 = pointer(struct listnode with symbol table 2)
    prev                20 - 27         type =   31 = pointer(struct listnode with symbol table 2)
    Total width = 28

+++ Symbol table 3 [struct coordinates]
    x                    0 - 3          type =    8 = int
    y                    4 - 7          type =    8 = int
    z                    8 - 11         type =    8 = int
    Total width = 12
$
```