CS39003 Compilers Laboratory
Autumn 2025
Assignment 5
Date of posting: 06-Oct-2025

Yacc programming with inherited attributes

This assignment deals with C-type declaration of variables, arrays, and pointers. We use the following grammar for this purpose. The start symbol is PROG which is used to generate a (non-empty) sequence of declarations (DECLIST). Each declaration, generated by the non-terminal DECL, starts with a basic type (like int or double) followed by a non-empty comma-separated list of variables and then by a semicolon. Each variable in a list of variables contains one ID preceded optionally by one or more stars (indicating pointers) and followed optionally by one or more dimensions within square brackets (indicating arrays). The terminal symbols below are in color red.

```
PROG     →  DECLIST
DECLIST  →  DECLIST DECL | DECL
DECL     →  BASIC 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 roll[10], name[101];
unsigned short int yob;
float CGPA;
char gender;

long int a, *p, **q, ***r, A[5], B[6][7], **C[2][3][4];

double points[100][3], *P;

void *vptr;

unsigned char letters[26];
```

This assignment deals with the following tasks.

- Write a lex file `ctype.l` to identify the tokens (identifiers, numbers, and punctuation symbols).
- Write a yacc file `ctype.y` to do the following tasks associated with parsing.
    - Maintain a global type table `TT` for storing all basic types and other types (arrays and pointers) that the user defines in the input file.
    - Maintain a global symbol table `ST` to store all the names (and associated information) about the variables declared in the program.
- The yacc grammar should only call the functions for handling the type table `TT` and the symbol table `ST`. These functions may be defined in the same yacc file, or in another separate source file. Also write functions to print the contents of the type table and the symbol table.
- A `main` function that first sets `yyin` (if needed), then calls `yyparse()` to process all the declarations in the input file, and finally prints the contents of the type table and the symbol table by calling appropriate functions.

Lex file

As usual, the lex file (`yylex()` to be more precise) will supply the stream of input tokens to `yyparse()`. For each basic data type, use a single token. For example, both unsigned short and unsigned short int should return the same token USRT. You may use the following eleven token names for the basic data types (to be declared in the yacc file): VOID, UCHR, CHR, SRT, USRT, LNG, ULNG, UINT, INT, FLT, and DBL. In general, white spaces should be ignored by lex. But special care needs to be taken here, because some basic data types contain white spaces. In the example above, there must be any non-empty sequence of white spaces between unsigned and short, and also between short and int. In this assignment, treat the new-line character as a white space. In earlier

assignments, the new-line character marks the end of an assignment statement. Now, semicolon is used as the explicit statement delimiter, so the new-line character no longer needs to play this role.

This assignment does not deal with any keyword.

Identifiers are generated by the token id. We follow the C-style naming convention for variables. Positive integers are generated by the token num for use as array dimensions.

The punctuation symbols to be returned are: * ; , [ ]

Type table

The type table TT is at the heart of type checking. You need to build a careful data structure to store all the types appearing in the input program. Use an array of structures to implement TT. The components of each element of TT are explained shortly.

You start by populating TT by the eleven basic types. New types are generated by array and pointer constructs. These types should be added to TT as soon as they are generated. These new types are usually defined recursively. For example, consider the following declarations.

```
long int x, A[10], B[5][10], *p, **q, **C[2][3][4];
```

The types of the four variables introduced by this declaration are as follows.

```
x   long
A   array(10,long)
B   array(5,array(10,long))
p   pointer(long)
q   pointer(pointer(long))
C   array(2,array(3,array(4,pointer(pointer(long)))))
```

These types must be stored in the type table TT. This table must never contain duplicate types. TT is already loaded with the basic types. Assume that the remaining types are currently not residing in TT.

Since x is of basic type, no new type needs to be inserted to TT. For A, the new type to be inserted in TT will be array(10,long). The entry for this type will have category ARRAY, dimension 10, and element type long (reference to the entry of TT storing this basic type). B uses the same data type for A in order to represent each row, so the type array(10,long) must not be inserted again in TT. However, the new type to be added to TT will be array(5,array(10,long)). This is again of category ARRAY, but of dimension 5, and the element type is a reference to (that is, index in TT of) the type array(10,long). Note that two array types are the same if and only if they have the same dimension and the same reference type. That is, array(10,long) and array(20,long) are of different types. Likewise, array(10,long), array(10,double), array(2,array(5,long)) and array(10,array(8,char)) are all of different types.

The new type to be inserted for p is an entry in TT of category POINTER, and with a reference to the basic type long. Since q is a pointer to this pointer type, only a single entry is to the added to TT. This entry is of category POINTER, and with a reference to the type pointer(long) that is already inserted in TT.

The declaration for C uses many new types. The element type of this three-dimensional array is pointer(pointer(long)). This type is already introduced in TT by the declaration of q. For a complete definition of the type of C, the following new types are to be added to TT.

```
array(4,pointer(pointer(long)))
array(3,array(4,pointer(pointer(long))))
array(2,array(3,array(4,pointer(pointer(long)))))
```

All these entries are of category ARRAY, but have different dimensions and different reference types.

Alongside the type, each entry in TT should store the width (memory requirement or size) of (each variable of) that type. For a basic type, this width can be obtained using the sizeof operator of C. For example, width(CHR) = sizeof(char) = 1, width(INT) = sizeof(int) = 4, and width(DBL) = sizeof(double) = 8. We assume that all pointers are of the same size that can be obtained as sizeof(void *). The width of the array ARRAY(DIM,element_type) is DIM times the width of element_type. Since element_type is a reference to an entry in TT, the width of that entry is to be used for calculating the width of the ARRAY entry in TT.

We assume that **all variables are 4-byte aligned**, that is, start at locations in the data segment, that are multiples of 4. As an example, consider the first four declarations of the example on the first page. Assume that no declarations appear before them. We start placing the variables at offset zero in the data segment.

```
char roll[10], name[101];
unsigned short int yob;
float CGPA;
char gender;
```

We use a 9-letter roll number as in IITKGP. If you need to store this as a string, you need to have a trailing null character, so the array `roll` will be of size 10, and occupies memory locations 0 – 9 inside the data segment. The next array `name` (capable of storing a name of maximum length 100, as a string) cannot start from memory location 10 immediately after the end of the array `roll`. It will start at the next multiple of 4, that is, from offset 12 inside the data segment, and will occupy memory locations 12 – 112. The third entry is an `unsigned 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 next variable (`a` in the example on the first page) will start from memory location 128, and so on. The total data segment size will be rounded up to the next multiple of 4.

We now tabulate the fields of each entry in `TT`. Each blank entry in the following table means field is not used.

Category	Dimension	Reference	Width
VOID			0
UCHR			1
…			
DBL			8
ARRAY	Number of elements	Index of the element type	To be calculated
POINTER		Index of the element type	Pointer size (usually 8)

Design your own implementation (array of records) of the 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 table

In earlier assignments, we have used a symbol table consisting of (`name`, `value`) pairs. From this assignment onward, each entry in each symbol table will be a triple (`name`, `type`, `offset`). Here, `name` is the user-given name of the variable, `type` is a reference to (index in) the entry in the type table `TT`, storing the type of that variable, and `offset` is the starting location (a multiple of 4) of the memory allocated to the variable (the offset in the data segment).

Design your own implementation for the symbol table. Again you do not have to go for any efficient data structures. Instead use an array of (`name`, `type`, `offset`) triples. Make linear search (for a `name`) in the symbol table.

Also write a function to insert a variable in the symbol table. This function would require two arguments: the name of the variable, and the type of the variable (an index in `TT`). The offset of that variable to be stored in the triple (`name`, `type`, `offset`) of the symbol table is to be computed by the insertion function. A global variable is used to store the current width (always maintained as a multiple of 4) of the symbol table. Disallow duplicate names of two variables in the symbol table.

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.

Width and offset calculations can be carried out using the global type table `TT` and the global width of the symbol table. Use the token num for calculating the width of an array. Store that width in `TT` against that array type. There is no need to use synthesized attributes in these calculations.

There is however a need to use inherited attributes that demand marker non-terminals. Each DECL begins with a basic data type. This should move as it is, to each outermost VAR in the rest of the declaration. Whenever a VAR encounters the production VAR → * VAR, the type of the child VAR changes to pointer(type of the parent VAR). Eventually, VAR vanishes by the production VAR → id DIM. The type of VAR then passes down as it is, to the subtree under DIM. Eventually, DIM → ε is used. Subsequently, the type of DIM undergoes a sequence of array constructions in a synthesized manner. At the end of these synthesized computations, id gets its type. This type is stored in `TT` (if not present already), and id is added to the symbol table with a reference to this type. The offset of id is obtained from the current width of the symbol table. The width of the symbol table is incremented by the width of (the type of) id, and if needed, rounded up to the nearest multiple of 4 for the placement of the next variable.

Notice that each use of VAR → * VAR introduces a pointer type. The type table TT is searched for that pointer type, and if not found, that type is inserted to TT. Moreover, each reduction using DIM → [num] DIM uses an array construction. This array type, if not already present in TT, should be inserted there.

Use suitable marker non-terminals for the above parent-to-child transfers of types (indices in TT).

Printing the tables

After all declarations are read, print the type table TT, and the details of the variables stored in the symbol table ST. Use the format as explained in the Sample Output section below. Write 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

+++ 27 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:       10    array(10,char)
    Type  12:      101    array(101,char)
    Type  13:        8    pointer(long)
    Type  14:        8    pointer(pointer(long))
    Type  15:        8    pointer(pointer(pointer(long)))
    Type  16:       40    array(5,long)
    Type  17:       56    array(7,long)
    Type  18:      336    array(6,array(7,long))
    Type  19:       32    array(4,pointer(pointer(long)))
    Type  20:       96    array(3,array(4,pointer(pointer(long))))
    Type  21:      192    array(2,array(3,array(4,pointer(pointer(long)))))
    Type  22:       24    array(3,double)
    Type  23:     2400    array(100,array(3,double))
    Type  24:        8    pointer(double)
    Type  25:        8    pointer(void)
    Type  26:       26    array(26,unsigned char)

+++ Symbol table
    roll                0 - 9          type =   11 = array(10,char)
    name               12 - 112        type =   12 = array(101,char)
    yob               116 - 117        type =    3 = unsigned short
    CGPA              120 - 123        type =    9 = float
    gender            124 - 124        type =    2 = char
    a                 128 - 135        type =    6 = long
    p                 136 - 143        type =   13 = pointer(long)
    q                 144 - 151        type =   14 = pointer(pointer(long))
    r                 152 - 159        type =   15 = pointer(pointer(pointer(long)))
    A                 160 - 199        type =   16 = array(5,long)
    B                 200 - 535        type =   18 = array(6,array(7,long))
    C                 536 - 727        type =   21 = array(2,array(3,array(4,pointer(pointer(long)))))
    points            728 - 3127       type =   23 = array(100,array(3,double))
    P                3128 - 3135       type =   24 = pointer(double)
    vptr             3136 - 3143       type =   25 = pointer(void)
    letters          3144 - 3169       type =   26 = array(26,unsigned char)
    Total width = 3172
$
```