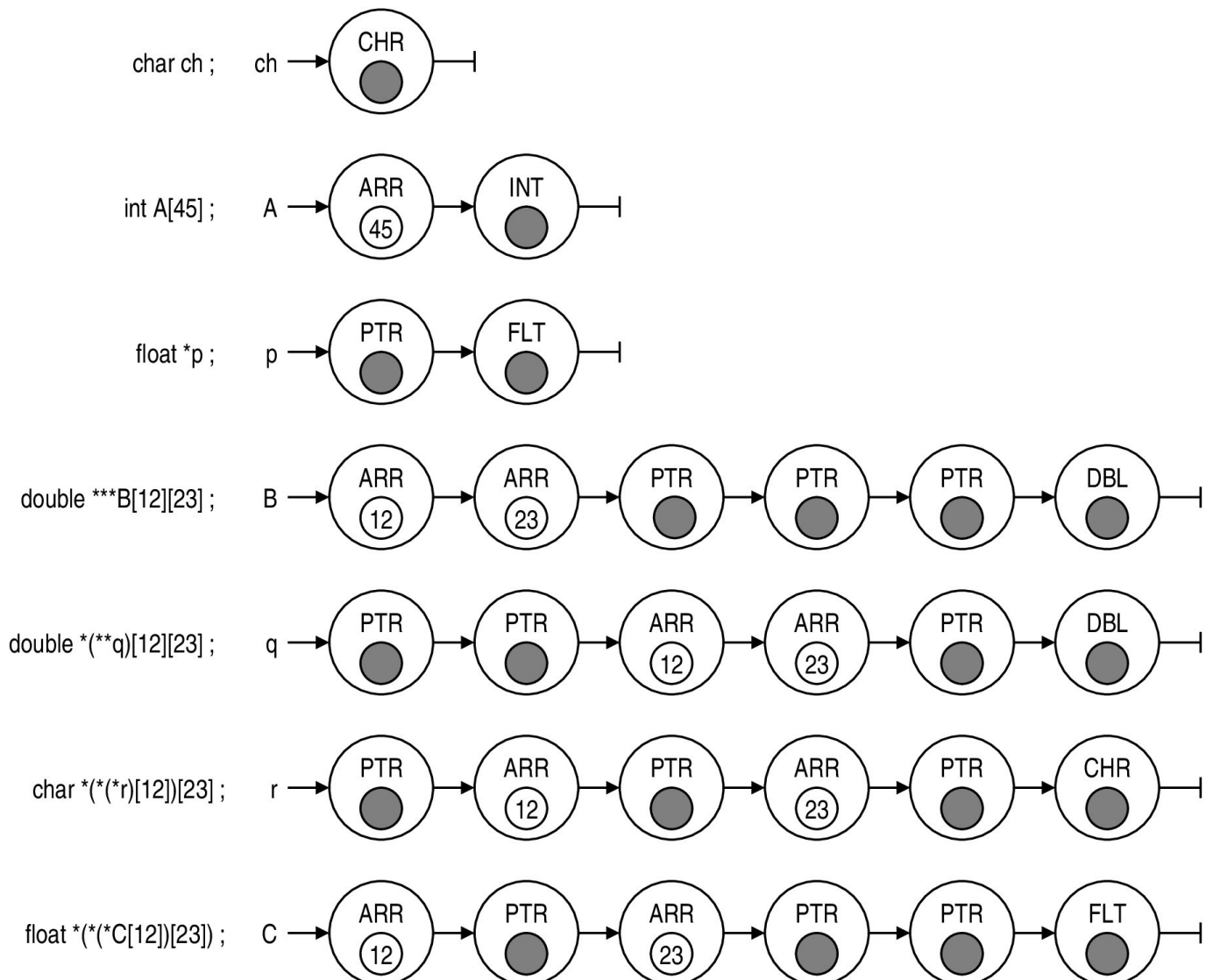


Syntax-directed translation

This assignment deals with the most general form of variable declarations using C-type pointer and array constructs. The yacc slides explain how variables of basic data types, pointers to basic data types, and one-dimensional arrays of basic data types can be defined. Examples: `int n`, `*p`, `A[10]`;

In a type expression, we may have multiple levels of pointer indirection. We may also have multi-dimensional arrays. Examples: `int n`, `***p`, `A[10][20][30]`; The pointer and the array constructs can be clubbed together. Example: `int **A[10][20][30]`; Let us follow the C-style precedence, that is, the array constructor has higher precedence over the pointer constructor. The last example defines `A` as a $10 \times 20 \times 30$ array of pointer to pointer to `int`. This default behavior can be altered by using parentheses. For example, `int (**A)[10][20][30]`; defines a pointer to a pointer to a $10 \times 20 \times 30$ array of `int`. The parentheses may be nested. For example, `int (*(A[10])[20])[30]`; defines an array of 10 pointers to an array of 20 pointers to an array of 30 `int`. It makes no sense to have parentheses within a sequence of `*`'s or a sequence of array constructors, so we assume that each (matched) parenthesis pair encloses a variable name.

We represent the type of a variable as a linked list. Each node in the list consists of three items: the type of the node (basic type, or pointer type, or array type), size (relevant only for array-type nodes), and a pointer to the next node in the list. The following figure illustrates this concept. Later, we will add another component (width) to each node.



These lists grow at the end, that is, new nodes are appended to the end. Therefore we store both a head pointer and a tail pointer in a type list, that is, a list is a pair (head, tail) of node pointers. Use a pointer to such a pair as yacc's stack type for lists, so implement this as a C structure.

We define a declaration using the following grammar. For simplicity, we assume that each declaration statement defines a single variable (although the extension to handling multiple variables per declaration is straightforward).

```
DECL    →    TYPE TEXPR ;
TYPE    →    char | int | float | double
TEXPR   →    PTR TEXID ARR
PTR     →    ε | PTR *
ARR     →    ε | ARR [ num ]
TEXID   →    id | ( TEXPR )
```

Each type expression (TEXPR) starts with a (possibly empty) sequence of *****'s, and ends with a (possibly empty) sequence of array constructs **[num]**. Between these two, we have TEXID (type expression with an ID) which can be either a variable name (**id**) or a parenthesized type expression.

Both TEXPR and TEXID will compute partial lists, and will set \$\$ to a pointer to the list. Let us see how the reduction of TEXPR → PTR TEXID ARR would work. The count of *****'s is stored in PTR. Then TEXID creates a list (in its \$\$ which is \$2 in the handle); call this list *L*. If the production TEXID → **id** is used, *L* is the empty list (in this case, a symbol-table entry is arranged for the variable). If the production TEXID → (TEXPR) is used, then *L* is the list created by the inner TEXPR. Then, we proceed to the ARR section, and scan the **[num]**'s from left to right. For every such construct, an array-type node is appended to *L*. When we are done with ARR, we turn our attention to the count *c* of pointer constructs stored in PTR. We append *c* pointer-type nodes to *L*, and this *L* will be the final \$\$ for the head of the production (TEXPR). Eventually, the reduction DECL → TYPE TEXPR will be used. At that time, the basic-type node (the basic type is stored in \$1) will be appended to the list created by TEXPR and available as \$2.

We illustrate this construction in the following example.

Processing of **int **(**A[5])[6][7]**

The inner part **(**A[5])** is first fully processed.

1. Consume ******* from the input. Remember in PTR that three pointer operators are read.
2. **A** (an ID) is shifted from the input. This is followed by the reduction TEXID → **id**. A reference **stref** to the symbol-table entry for storing this variable is saved globally. Storing this in a marker will involve additional effort, so a global variable is a practical compromise. Note that this name will continue to make sense until the eventual reduction of DECL. Finally, the empty list is stored as \$. Unlike C, we will allow a variable to be redefined. In that case, the old type of the variable will be replaced by the new type.
3. Then there is the reduction ARR → ε. Nothing needs to be done.
4. Shift **[5]** from the input.
5. The reduction ARR → ARR **[5]** is carried out. The list created by TEXID is available as \$0. Append the node (ARR, 5) to this list. The list becomes (ARR, 5).
6. There are no more reductions involving ARR. It is now time for the reduction TEXPR → PTR TEXID ARR. At this point, \$2 = (ARR, 5). Recall from the count stored in PTR that three pointer nodes should be added. The list changes to (ARR, 5) → (PTR) → (PTR) → (PTR). This is the final \$\$ for the inner TEXPR.

Processing the outer part **int ** TEXID [6][7]**

1. Shift **int** from the input, and make the reduction TYPE → **int**.
2. Consume ****** from the input, and remember in PTR that two pointer nodes are to be appended later.

3. Encountering `(` in the input processes the inner part. As explained above, this gives the list $(ARR, 5) \rightarrow (PTR) \rightarrow (PTR) \rightarrow (PTR)$.
4. Then the reduction $ARR \rightarrow \epsilon$ does nothing.
5. Then there are two reductions $ARR \rightarrow ARR [6]$ and $ARR \rightarrow ARR [7]$ (after appropriate shifts). These reductions respectively append two array nodes to $\$0$. So the list now becomes $(ARR, 5) \rightarrow (PTR) \rightarrow (PTR) \rightarrow (PTR) \rightarrow (ARR, 6) \rightarrow (ARR, 7)$.
6. The count stored in `PTR` for this `TEXPR` is now consulted, and two pointer nodes are appended. The list becomes $(ARR, 5) \rightarrow (PTR) \rightarrow (PTR) \rightarrow (PTR) \rightarrow (ARR, 6) \rightarrow (ARR, 7) \rightarrow (PTR) \rightarrow (PTR)$. This is the $\$\$$ for the outer `TEXPR`.

Finally, the reduction $DECL \rightarrow TYPE\ TEXPR$ will take place. A node storing the basic data type `int` is added to $\$2$. So we get the final list $(ARR, 5) \rightarrow (PTR) \rightarrow (PTR) \rightarrow (PTR) \rightarrow (ARR, 6) \rightarrow (ARR, 7) \rightarrow (PTR) \rightarrow (PTR) \rightarrow (INT)$. This list is stored as the type in the global `stref` created when the name `A` was seen.

Write a `lex` file to return only the relevant tokens: `id`, `num`, the keywords `char`, `int`, `float`, and `double`, single-letter constructs and punctuation symbols `*`, `[`, `]`, `(`, and `)`, `;`, and `\n`. Ignore white spaces (white spaces are not allowed inside `id`, `num`, and the keywords). Also ignore empty lines (this can be done at the yacc level).

Write a `yacc` file to implement the above construction and storage of linked lists. This file will not implement any list-related function or symbol-table utility. Yacc actions will only call the relevant functions. When a type is stored in the symbol table, the name and the type of the variable are printed in the format given in the sample output.

We also need to calculate the width of each variable. The type of a variable is calculated as a list. Write a recursive function to calculate the width of each node in the list. If the node is of a basic data type, use the following widths. `char` : 1, `int` : 4, `float` : 4, and `double` : 8. If the node is a pointer, its width is 8 (irrespective of what it is pointing to). Finally, if the node is of the array type with size s , then the width of the node is s times the width of the next node in the list. At the time the list is stored in the symbol table, the width at the nodes should be calculated (you cannot do it beforehand because the list becomes complete only after the addition of the last basic-type node).

Write the list-related definitions and functions in a separate source file, and the symbol-table definitions and functions in yet another source file. You may use other header files if you need them. The `main()` function may be in the yacc file or in a separate file that combines all the required source files (`lex.yy.c`, `y.tab.c`, and your source files).

In the `main()` function, first process the input file. Then, print the final contents of the symbol table (see the sample output). During the printing, assume that the variables are stored consecutively in a data segment starting from the offset 0. The width of a variable is the width stored in the first node of its type list. Calculate the offset of the next variable by adding this width to the current offset and rounding it up to the next multiple of 4. This happens typically, because variables are kept word-aligned, that is, start at offsets which are multiples of the word size (assumed to be 4 here). The next multiple of a (non-negative) integer w can be computed by the integer-only expression $4*((w+3)/4)$. For instance, if a `char` array of size 21 starts at offset 456, then the next variable will start at offset $456 + 24 = 480$.

Include all the source files and a makefile (with `compile`, `run` (on `input.txt`), and `clean` targets) in a `tar/tgz/zip` archive. Submit that archive.

Sample Output

Input

```
char ch;
int A[45];
float *p;

double * ARR [ 77 ] ;
double ( * PTR ) [ 77 ] ;

double ***B[12][23];
double * (**q) [12][23];
char *(*r)[12][23];
float *(*C[12])[23];

int *(*(**(B))[11][22])[33][44])[55][66];
```

Output

```
+++ Creating new entry for ch
+++ Stored ch with type char

+++ Creating new entry for A
+++ Stored A with type array(45,int)

+++ Creating new entry for p
+++ Stored p with type pointer(float)

+++ Creating new entry for ARR
+++ Stored ARR with type array(77,pointer(double))

+++ Creating new entry for PTR
+++ Stored PTR with type pointer(array(77,double))

+++ Creating new entry for B
+++ Stored B with type array(12,array(23,pointer(pointer(pointer(double))))))

+++ Creating new entry for q
+++ Stored q with type pointer(pointer(array(12,array(23,pointer(double))))))

+++ Creating new entry for r
+++ Stored r with type pointer(array(12,pointer(array(23,pointer(char))))))

+++ Creating new entry for C
+++ Stored C with type array(12,pointer(array(23,pointer(pointer(float))))))

+++ Changing entry for B
+++ Stored B with type pointer(array(11,array(22,pointer(pointer(array(33,array(44,pointer(array(55,array(66,pointer(int)))))))))))

+++ Symbol table
name      width  offset  type
----
ch         1       0      char
A          180     4      array(45,int)
p          8       184     pointer(float)
ARR        616     192     array(77,pointer(double))
PTR         8       808     pointer(array(77,double))
B           8       816     pointer(array(11,array(22,pointer(pointer(array(33,array(44,pointer(array(55,array(66,pointer(int)))))))))))
q           8       824     pointer(pointer(array(12,array(23,pointer(double))))))
r           8       832     pointer(array(12,pointer(array(23,pointer(char))))))
C          96       840     array(12,pointer(array(23,pointer(pointer(float))))))
```