ASSIGNMENT #3: Declarations, Symbols and Types

At the conclusion of this assignment, your compiler will be able to parse C language declarations and construct an internal representation of symbols and types and which will support scoping rules.

The Symbol Table

As discussed in lecture notes for Unit #4 (Semantic Analysis) a key component of your compiler is a symbol table. For this assignment, your symbol table should support:

- The ability to have different attributes for the different roles which IDENTIFIERs play in the C language. At this point in the course, some of these attributes don't make sense yet, but you should have the framework in place. In particular, for variables, you should be able to set a storage class.
- The four identifier namespaces (tags, labels, members, and "everything else")
- All four types of scopes (file, function, block and prototype) as well as arbitrary nesting of scopes to any depth. We will be making prototypes optional, but you should still provide a hook in the symbol table for supporting this type of scope.
- Keeping track of what the "current" scope is, and being able to look up a given identifier in a given namespace starting from a given scope and working up through the outer enclosing scopes up to and including global scope.
- Installing a new symbol into a given symbol table. Provide an option for handling the attempted installation of an identifier with the same name and same namespace: either allow duplicate definition (see lecture notes), or throw an error.

AST Representation of Types

As discussed in lecture notes and in class, the AST mechanism which you began to develop in assignment #2 can be readily extended to support the representation of arbitrarily complicated C types. You'll want to develop AST nodes representing scalar types, pointers, arrays and functions, as well as some mechanism (e.g. the "mini symbol table" approach) for representing structure and union types. You'll definitely want to write a system of "dumping" a given AST or portion of an AST to plain text, so you can determine if your declarations parsing code is producing the correct type representation! This should be a straightforward extension of the code which you've already developed in assignment 2.

Read the lecture notes carefully with regard to constructing type-representing AST during parsing and how the AST naturally comes out "backwards." The notes discuss a few potential ways to address this.

THINGS YOU DON'T HAVE TO DO:

This is probably the paragraph that you have read first! In order to simplify the compiler and make it feasible as a one-semester project, a number of C language features will be removed from the spec. You may attempt them for your own amusement, but in some cases they are so difficult that doing so is not advised. With regard to Assignment #3, the following items are optional:

- Initialized declarations
- complicated constant expressions used within an array declarator. e.g. you don't have to parse char a[2+2]; You may assume that array declarators are either [] or [NUMBER]
- C99 variable-length arrays (very difficult)

- Qualifiers such as const, volatile and restrict. however, you should still allow those keywords in the grammar
- C99 inline functions
- enums (not that difficult)
- bit fields in structure definitions
- prototypes/formal parameter lists. This is a major simplification. If you want to try it, it is moderately difficult, to extremely difficult if you want it to work 100% correctly with typedefs. Instead you are permitted to assume that all function declarators are just (), i.e. taking unknown arguments.
- K&R-style function definitions (very old form)
- typedefs: Getting typedef working in its simplest form, e.g. `typedef int iii; iii var;` is not too difficult, but handling all possible cases including ambiguity involving typedef names getting redfined in a declaration, is extremely difficult. The lecture notes contain a discussion of some pathological cases.

Grammar issues

The top-level of the language now consists of a list of

```
declaration_or_fndef: declaration
                    |     function_definition
                    ;
```

Declarations made at this top-level can be assumed to have a storage class of EXTERN. For now, we can ignore the distinction of what happens when the keyword `static` or `extern` appears. This will not be an issue until we get to the unit on target code generation and how the linker works.

Function *definitions* can only appear at this top level of the grammar. Note that a function *definition* also *declares* that function's name globally as a symbol of identifier class FUNCTION, having the type FUNCTION(returning X, taking P). A workable grammar rule for function definition is:

```
function_definition:
          decl_specifiers declarator compound_statement
```

This grammar rule does not permit the short-hand notation such as `f() { return;}` where the declaration specifiers are missing and the return value defaults to int. It also doesn't support K&R style declarations. Consult the C language standard or H&S for a fuller discussion of function definitions.

For our purposes at this time, we could define

```
compound_statement: '{' decl_or_stmt_list '}'

decl_or_stmt:
          declaration
          |statement
          ;

statement:
          compound_statement
          | expression ';'
          ;
```

Note that the semicolon is part of the grammar rule for a declaration (I am deliberately not providing this rule above, this is part of what you need to work out) and certain kinds of statements. At this point in the course, we

do not have control flow statements such as if/else, while, for, return, break, switch, etc. Therefore the only two statements we support are expression statements (you have those from assignmnent #2) and now compound statements (we need these to introduce inner block scopes)

In assignment #2, you had to leave out two things from the expression grammar: casts, and the sizeof operator taking an abstract type name. You will now have the ability to recognize abstract type names (the syntax is a simplification of the regular declarations syntax) and so you can circle back and add those two things to make the expression grammar essentially complete.

Program behavior / what you HAVE to do!

1) Your program will accept the output of cpp (gcc -E), just like in assignment #2.

2) Whenever you see a declaration of an identifier as a function, typedef or variable, you will print out the following:

- name of the identifier being declared
- file/line at which the declaration is effective
- the scope in which the declaration is rooted (give type of scope (e.g. global, prototype, function) and file/line of scope start)
- what the identifier is being used for (e.g. typedef,var,fn)
- storage class (if applicable)
- data type (more below)

3) In several places you will need to output a human-readable representation of whatever internal data structures you create to manipulate types. As types have a potentially complicated, nested structure, one possible choice is a series of indented lines, e.g.

```
pointer to
   pointer to
     array of 10 elements of type
       pointer to
         int
```

Another possibility is using braces, brackets or parentheses, e.g.

```
pointer(pointer(array[10] (pointer(int))))
```

You could also output the result in the style of a C type name:

```
(int *(**)[10])
```

The latter is more compact but also harder to generate and more confusing to read, especially if you, yourself, are having difficulty following C type name abstract declarator syntax!

4) Whenever a struct or union is *defined* (i.e. with a pair of braces and a list of members inside), print out the file/line at which this definition is happening, and the scope in which it is rooted (remember, nested struct/union definitions are not rooted in the struct/union scope itself). Then print out the declaration of each member. Depending on how you handle the table of members, it may not be convenient to list the members in the order of their declaration, so you don't have to do that...just make sure to cover all of the members.

5) When a type involves a struct/union reference, print out the tag and where that struct/union type is defined, or (incomplete) if the definition has not been seen yet. E.g.:

```
              INPUT:
struct foo {
int a;
};


struct foo *p;

struct bar *p2;

              OUTPUT:
struct foo definition at <stdin>:1{
a is defined at <stdin>:2 [in struct/union scope starting at <stdin>:1]
     as a field of struct foo  off=0 bit_off=0 bit_wid=0, type:
  int
}

p is defined at <stdin>:4 [in global scope starting at <stdin>:1] as a
variable with stgclass extern  of type:
  pointer to
   struct foo (defined at <stdin>:1)

p2 is defined at <stdin>:8 [in global scope starting at <stdin>:1] as a
variable with stgclass extern  of type:
  pointer to
   struct bar (incomplete)
```

6) Output reasonable error messages for *semantically* invalid things, such as attempted re-declarations, invalid types (e.g. a function returning a function), etc. Error messages must give the file and line at which the error is detected. You should be able to recover from semantic errors and continue processing the input, but this is not mandatory. You are not expected to recover from syntax errors.

SUGGESTIONS AND ISSUES

The BNF grammar which appears in the C99 standard or in Harbison & Steele is a good starting point. Both use the notation `_opt` to indicate an "optional" symbol. This, of course, is not something which yacc/bison provides. foo_opt is equivalent to:

```
foo_opt:        /*empty*/
        |foo
        ;
```

You might find that it is better to "inline" some of these opt cases to avoid trouble with the LALR(1) algorithm, e.g.:

```
array_decl: decl '[' const_expr_opt ']'
```

could be re-written as

```
array_decl:     decl '[' const_expr ']'
                |decl '[' ']'
                ;
```

Note that some copies of Harbison & Steele had an error in the printing of the grammar. In the rule for declaration, `declarator_init_list` should be optional, otherwise

```
                struct a {int a;};
```

is not recognized. The C99 standard has it right. Your professor took another approach and explicitly added the rules for struct/union def ';' and struct/union ref ';'

Don't worry too much about efficient memory allocation. While we'd like to develop a memory management strategy that is neat, given the choice between completing the compiler project and having great memory allocators, we really need to favor the former.

IN CASE OF EMERGENCY

If you are running seriously behind and are looking for something to cut to get back on schedule, consider eliminating support for structures and unions, since we will not strictly need them to test the compiler later. This doesn't mean this is an optional part of the assignment, but if there is a part that you can't finish, this would have the least impact.

You could also skip abstract type names. This will prevent you from being able to handle the full C expression grammar (needed for cast expressions and one form of sizeof operator). Again, not optional, but good triage material. As we advance toward the final production compiler, we could live without these two things.

COMPLETELY OPTIONAL STUFF

If you plan on tackling prototypes/parameter lists at this time, there are some things you should know. If you wish to skip this optional part, stop reading now and just get started on this assignment.

The distinction between function DECLARATION and function DEFINITION is subtle and tricky. The following

```
int f(char *p,int a);
```

declares the identifier f to be the name of a function which returns int and has an argument prototype of (char *,int). This declaration is rooted in whatever the current scope is. It could appear inside of a function definition, block, or even a struct/union definition (except for the slight problem that functions can not be members of struct/union, pointers to function are OK).

Now, consider

```
int f(char *p,int a)
{
        /*...*/
}
```

this contains both a DECLARATION of f in the current scope and begins a DEFINITION of f too. The declaration takes effect, according to the C standard, at the end of the declarator of which f is a part, i.e. at the closing parenthesis. The function DEFINITION also begins a new scope, and the arguments p and a are implicitly declared within that scope. That declaration of the arguments along with their names disappears at the closing brace, but the declaration of f's return type and argument prototype survives. This is because the above form, a function DEFINITION, is only valid at the top-level of the grammar. Therefore the symbol f would be installed in the global scope.

Unfortunately for you, the compiler writer, both declarations and definitions of functions begin with a declarator. You don't know, when seeing the opening parenthesis, what you are ultimately going to get, because the parser is only looking one token ahead. The approach which your professor took is to save both the types and the names with the prototype, and then when it is apparent that a function definition is starting, to re-process that prototype to declare each formal parameter in the function's scope.

Your professor is also a fan of the older K&R syntax:

```
int f(p,a)                    /* Or even just f(p,a) */
char *p;            /* a is implicitly declared int */
{
        /*...*/
}
```

If you want to support K&R syntax, it will be necessary to insert a semantic action just before the opening brace to re-examine the formal parameter list and match the parameters up with declarations which were made in K&R style. It is also necessary to do the implicit int declarations.

Formal parameters to functions which are of type "array of X" or "function returning Y" are automatically converted to "pointer to X" or "pointer to function returning Y", as discussed in class, H&S and the C99 standard. You should apply those conversions when you process the prototype.