COM S 440/540 Project part 2

Create a parser (with a very basic symbol table)

1 Requirements for part 2

When executed with a mode of 2, your compiler should read the specified input file and check that the file has correct C syntax (for our subset of C, discussed in Section 2). If the input file is not syntactically correct, then display an appropriate error message. Syntax error messages should be written to standard error, and should indicate the input file name, line number, and offending text around where the error occurs or is discovered. After the first syntax error, your parser may either make a "best effort" attempt to continue processing the input file, or exit. When tested on input files with syntax errors, your compiler will be considered correct if it catches the first syntax error.

If the input file is syntactically correct, then the output should be a report of the global type declarations and global variables. Additionally, for each function, the function parameters, local type declarations, and local variables should be reported. See Section 3 for more details. If the input file contains syntax errors, then the output should be any subset of the output (including no output at all) that would be obtained by ignoring the syntax errors. For a compiler, "exit cleanly with no output on a syntax error" is better behavior than "garbage output and segfault on a syntax error".

This part of the project checks **only** the input file **syntax**. You **should not** check for type consistency, for existence of called functions, for existence of used variables, for duplication of function definitions, or for duplication of variable declarations. All of these will be required instead for part 3 of the project.

2 C language (subset) definition

For a minimalist implementation, the input language is defined in Section 2.1. Extra credit options, potentially requiring grammar modifications, are discussed in Sections 2.2 through 2.5. Rules that are more restrictive than the C standard (for purposes of making the parser or compiler simpler) are marked with † or ‡, with ‡ indicating that the rule will need to be modified to complete an extra credit option.

2.1 Minimalist rules

From the rules below, you should be able to define an appropriate grammar for our subset of the C programming language. The rules are numbered for later reference.

1. A C program is‡ a sequence of zero or more (global) variable declarations, function prototypes, and function definitions, appearing in any order.
2. A *variable declaration* is‡ a type name, followed by a comma-separated list of one or more identifiers, each identifier optionally followed by a left bracket, an integer constant, and a right bracket. The list is terminated with a semicolon. Note that this restricts arrays to a single dimension.
3. A *type name* is‡ one of the simple types: `void`, `char`, `int`, `float`.
4. A *function prototype* is a function declaration followed by a semicolon.
5. A *function declaration* is a type name (the return type of the function), followed by an identifier (the name of the function), a left parenthesis, an optional comma-separated list of formal parameters, and a right parenthesis.

6. A *formal parameter* is† a type name, followed by an identifier, and optionally followed by a left and right bracket.

7. A *function definition* is‡ a function declaration followed by a left brace, a sequence of zero or more variable declarations, a sequence of zero or more statements, and a right brace. Note that this definition requires all variable declarations to appear before statements.

8. A *statement block* is a left brace, a sequence of zero or more statements, and a right brace.

9. A *statement* is† one of the following.

   - An empty string followed by a semicolon.
   - An expression followed by a semicolon.
   - Keywords **break** or **continue** followed by a semicolon. Note: you do **not** need to check that these statements are within a loop.
   - Keyword **return**, followed by an optional expression, and a semicolon.
   - Keyword **if**, followed by a left parenthesis, an expression, and a right parenthesis, followed by either a statement block or a single statement followed by a semicolon. Then, optionally, the following: keyword **else**, followed by either a statement block, or a single statement followed by a semicolon.
   - Keyword **for**, followed by a left parenthesis, an optional expression, a semicolon, an optional expression, a semicolon, an optional expression, a right parenthesis, and then either a statement block, or a single statement followed by a semicolon.
   - Keyword **while**, followed by a left parenthesis, an expression, and a right parenthesis, and then either a statement block, or a single statement followed by a semicolon.
   - Keyword **do**, followed by either a statement block or a statement and a semicolon, followed by keyword **while**, a left parenthesis, an expression, a right parenthesis, and a semicolon.

10. An *expression* is† one of the following.

    - A literal (constant) value.
    - An identifier, left parenthesis, a comma-separated list of zero or more expressions, and a right parenthesis.
    - An l-value.
    - An l-value, an *assignment operator*, and an expression.
    - An l-value, preceeded by or followed by the increment or decrement operator.
    - A *unary operator*, and an expression.
    - An expression, a *binary operator*, and an expression.
    - An expression, a question mark, an expression, a colon, and an expression.
    - A left parenthesis, a type name, a right parenthesis, and an expression.
    - A left parenthesis, an expression, and a right parenthesis.

11. An *l-value* is‡ an identifier, optionally followed by a left bracket, an expression, and a right bracket. Note that this restricts array access to a single dimension.

12. Unary operators (for any expression) are: `-`, `!`, `~`

13. Binary operators are: `==`, `!=`, `>`, `>=`, `<`, `<=`, `+`, `-`, `*`, `/`, `%`, `|`, `&`, `||`, `&&`

14. Assignment operators are: `=`, `+=`, `-=`, `*=`, `/=`

Operator precedence and rules for associativity are shown in Table 1.

Operators	Associativity	Precedence
`()  []  .`	left to right	(highest)
`!  ~  -` (unary) `--  ++  (`*type*`)`	right to left	
`*  /  %`	left to right	
`+  -`	left to right	
`<  <=  >  >=`	left to right	
`==  !=`	left to right	
`&`	left to right	
`|`	left to right	
`&&`	left to right	
`||`	left to right	
`?:`	right to left	
`=  +=  -=  *=  /=`	right to left	
`,`	left to right	(lowest)

Table 1: Precedence and associativity of operators

2.2 Extra credit: variable initialization

Relax rule 2 to allow global and local variables to be declared and initialized at the same time. For example, this would allow variable declarations of the form

```
int a, b=3, c, d=b+1;
```

for both global and local variables. You may do this for simple types only; do not worry about initializing arrays or struct variables.

If you choose to implement this feature, you may have the option of earning additional extra credit for code generation in part 4.

2.3 Extra credit: constants

Relax rule 3 to allow type names to be modified with the keyword `const`, either before or after the type name. For example, this would enable the declaration of function parameters with type `const int` or `int const`; note that these are equivalent types.

If you choose to implement this feature, you will have the option of earning additional extra credit for type checking in part 3.

2.4 Extra credit: user-defined structs

Modify rule 2 so that keyword `struct`, followed by an identifier, is also a valid type name. If you implement the `const` keyword (c.f. Section 2.3), make sure it may be applied to `struct` types and members.

Modify rules 1 and 7 to allow user-defined `struct` types. For a C program, global user-defined types may appear in any order. While any type and variable declarations that are local to a function may be intermingled, you may assume (and define your grammar accordingly) that these must appear before any statements of the function. A *user-defined type declaration* is[†] the keyword `struct`, followed by an identifier, a left brace, zero or more variable declarations (without initializations), a right brace, and a semicolon.

Note that you must allow (1) arrays of structs, (2) array variables inside structs, and (3) arbitrary nesting of structs. If you choose to implement this feature, you will have the option of earning additional extra credit for type checking in part 3.

2.5 Extra credit: struct member selection

Modify rule 11 to allow member selection using the `.` operator. For example, `point.x` should be a valid l-value. Remember that you must allow (1) arrays of structs, (2) array variables inside structs, and (3)

arbitrary nesting of structs. For example,

```
window[3].upperleft.x = mouse.x.stack[mouse.x.top];
```

should be syntactically correct. The *type checking* of that expression falls under part 3, where you will have the option of earning additional extra credit. Also, you may be able to earn extra credit for code generation in part 4.

3 Format of output

The output should contain the following, in order.

1. For each user-defined global `struct` in declaration order (if you did not implement the extra credit discussed in Section 2.4, there will be none), write the line

   `Global struct` *structname*

   (with the correct *structname*), followed by a line containing a comma-separated list of the struct members (in the same order they were declared), followed by a blank line.

2. If there are any global variables, write the line

   `Global variables`

   followed by a line containing a comma-separated list of global variable names (in the same order they were declared), followed by a blank line.

3. For each function prototype or definition, in declaration order, write the line

   `Prototype` *funcname*

   if it is just a prototype, and write the line

   `Function` *funcname*

   if the function body is given (with the correct *funcname*). Then write the following, in order.

   (a) Write a line containing "`Parameters: `" and a comma-separated list of function parameter names in declaration order. If there are no parameters, omit this line.

   (b) If the function body is given, write a line containing "`Local structs: `" and a comma-separated list of user-defined structs in declaration order. If there are no local user-defined structs (or you did not implement the extra credit discussed in Section 2.4), omit this line.

   (c) If the function body is given, write a line containing "`Local variables: `" and a comma-separated list of local variable names in declaration order. If there are no local variables, omit this line.

Important note: every `struct` member, parameter, local variable, and global variable should be written with `[]` appended if it is an array. Students are encouraged to check the example outputs in Sections 4 and 5, and to test their code thoroughly using examples developed by the instructor and other students.

4 Examples: minimal implementation

4.1 Input: `p2test.c`

```c
int x, y;
float z[50];

int foo(int z)
{
  // Variable a is never declared? That's OK for now!
  return a;
}

int bar(int a, int b)
{
  // Local variable hides parameter, OK for now
  int a;
  // Incorrect parameter type is OK for now
  a = foo(4.2);
  for (i=0; i<10; i++) {
    foo(i, 7);
    // Incorrect number of parameters is OK for now
  }
  // Incorrect assignment type is OK for now
  a = z * 2.5;
  // Incorrect return type is OK for now
  return 5.3;
}

int more, global[25], variables;

float proto_only(char x, int y[], float z);

int test(int lots, int more, int useless)
{
  char variables[15], just, to, show;
}

int bar(int a, int b) // duplicate definition for bar? OK for now!
{
  int d;
  d = 0;
  while ( (d += ++a) < b);
  return d;
}

int main()
{
  float how, this, part, should, work;
  return 7;
}
```

4.2 Output for `mycc -2 p2test.c`

```
Global variables
   x, y, z[], more, global[], variables

Function foo
```

```
    Parameters: z

Function bar
    Parameters: a, b
    Local variables: a

Prototype proto_only
    Parameters: x, y[], z

Function test
    Parameters: lots, more, useless
    Local variables: variables[], just, to, show

Function bar
    Parameters: a, b
    Local variables: d

Function main
    Local variables: how, this, part, should, work
```

4.3 Errors for `mycc -2 p2test.c`

There are no errors. Syntactically, `p2test.c` is correct.

4.4 Input: `p2error.c`

```
1  int ok()
2  {
3    /* Empty functions are allowed! */
4  }
5
6  int printf(int n) // Because we can
7  {
8    int i;
9    i = 0;
10   for (;;) {
11     i++;
12     n/=2;
13     if (n) continue;
14     return i;
15   }
16 }
17
18 int too_many_elses(int a, int b)
19 {
20   if (a<b) {
21     return 1;
22   } else {
23     return 2;
24   } else {
25     return 3;
26   }
27 }
```

4.5 Errors for `mycc -2 p2error.c`

```
Error near p2error.c line 24 text 'else'
       syntax error
```

4.6 Output for `mycc -2 p2error.c`

```
Function ok

Function printf
    Parameters: n
    Local variables: i
```

4.7 Discussion for `p2error.c`

The input file has a syntax error, which is caught by the compiler. In this case, the output is the required information for the functions that were parsed correctly before the first syntax error.

5 Examples: all extra credit implemented

5.1 Input: `extra.c`

```
1  const float pi = 3.14159265358979323846264433; /* approximately */
2
3  struct point {
4    int x, y;
5  };
6
7  struct rectangle {
8    struct point upperleft;
9    struct point lowerright;
10 };
11
12 struct window {
13   struct rectangle area;
14   char text[1024];
15 };
16
17 void display(const struct window W[], int n);
18
19 struct point strange(int z)
20 {
21   /*
22     Syntactically correct.
23     Will fail type checking in part 3.
24   */
25   int y;
26   struct mything {
27     float a, b, c;
28   };
29   struct other A;
30
31   display(y, A.b.c[15].d.e, F[15].g);
32
```

```
33    for (;;) {
34      if (y.x == y) return;
35      y++;
36      --y.x;
37      break;
38    }
39    return A.x;
40  }
```

5.2 Output for `mycc -2 extra.c`

```
Global struct point
    x, y

Global struct rectangle
    upperleft, lowerright

Global struct window
    area, text[]

Global variables
    pi

Prototype display
    Parameters: W[], n

Function strange
    Parameters: z
    Local structs:    mything
    Local variables:  y, A
```

5.3 Errors for `mycc -2 extra.c`

There are no errors. Syntactically, `extra.c` is correct.

6 Grading

Points	Description
20	**Documentation**
5	`README.txt` How to build the compiler and documentation. Updated to show which part 2 features are implemented.
15	`developers.pdf` New section for part 2, that explains the purpose of each source file, the main data structures used, and gives a high-level overview of how the various features are implemented.
10	**Ease of grading**
5	Building How easy was it for the graders to build your compiler and documentation? For full credit, simply running "`make`" should build both the documentation and the compiler executable, and running "`make clean`" should remove all generated files.

	5	Works with script Does the `-o` switch work? Is your output formatted correctly?
10		**Still works in modes 0 and 1**
60		**Basic Parser**
	5	Global variables
	5	Function prototypes / parameter lists
	5	Function local variables and body
	10	For, while, do loops
	5	if then else
	5	break / continue / return / expression stmts
	10	Expressions with unary/binary/ternary operators
	5	Assignment operators; increment and decrement
	5	Identifiers and arrays
	5	Function calls and parameters
5		**Extra credit: Variable initializations** See Section 2.2
5		**Extra credit: Constants** See Section 2.3
10		**Extra credit: User-defined structs** See Section 2.4
5		**Extra credit: Struct member selection** See Section 2.5
100		**Total for students in 440 (max points is 120)**
115		**Total for students in 540**

7 Submission

Part	Penalty applied
Part 0	20% off
Part 1	10% off

Table 3: Penalty applied when re-grading

Be sure to commit your source code and documentation to your git repository, and to upload (push) those commits to the server so that we may grade them. In Canvas, indicate which parts you would like us to re-grade for reduced credit (see Table 3 for penalty information). Otherwise, we will grade only part 2.