COM S 440/540 Project part 2

C parser

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. Error messages should be written to standard error, and have the format

> `Parser error in file` *filename* `line` *line number* `at text` *lexeme*
> *Description*

where the line number and offending text should be where the error occurs (or was detected). The description will typically indicate what the parser expected to appear next in input. 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. 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. These checks will instead be required for part 3 of the project.

If the input file is syntactically correct, your parser should create a single output file with extension `.parser`, containing a list of the global variables, function definitions, parameters, and local variables (see Section 4 for details).

2 C language (subset) definition: basic implementation

Rules for a minimalist implementation are given below; from these you should be able to define an appropriate grammar for our subset of the C programming language. The rules are numbered for later reference. Extra features, potentially requiring grammar modifications, are discussed in Section 3. 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 for one or more extra features.

1. A C program is‡ a sequence of zero or more (global) variable declarations or 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 literal, 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 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.
5. A *formal parameter* is† a type name, followed by an identifier, and optionally followed by a left and right bracket.

6. A *function definition* is‡ a function declaration followed by a left brace, a sequence of zero or more variable declarations or statements, and a right brace.

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

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

   - Nothing, 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 (yet).
   - 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. Then, optionally, the following: keyword **else**, followed by either a statement block, or a single statement.
   - 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.
   - Keyword **while**, followed by a left parenthesis, an expression, and a right parenthesis, and then either a statement block, or a single statement.
   - Keyword **do**, followed by either a statement block or a single statement, followed by keyword **while**, a left parenthesis, an expression, a right parenthesis, and a semicolon.

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

   - A literal 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.

10. 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.

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

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

13. 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

3 Extra features

3.1 Function prototypes

Modify rule 1 to allow function prototypes, as would appear in a header file or as "forward" declarations:

1'. A C program is‡ a sequence of zero or more (global) variable declarations, function prototypes, or function definitions, appearing in any order.

A *function prototype* is a function declaration followed by a semicolon.

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

3.2 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.

3.3 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.

3.4 User-defined structs

Modify rule 3 so that keyword `struct`, followed by an identifier, is also a valid type name. If you implement the `const` keyword (c.f. Section 3.3), make sure it may be applied to `struct` types and members.

Modify rules 1 and 6 to allow user-defined `struct` types. For a C program, global user-defined types may appear in any order. 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.

3.5 Struct member selection

Modify rule 10 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.

4 Output format

The output file should contain a line of the form

`File` *filename* `Line` *lineno*`:` *kind ident*

for each identifier, where the displayed *filename* and *lineno* should be the location of the identifier in the source file. The different "kinds" of identifiers are as follows.

- `global variable`, for global variables.
- `global struct`, for global struct declarations, assuming your parser can handle them.
- `function`, for function declarations. This includes function prototypes, if your parser can handle them.
- `parameter`, for formal parameters in a function declaration.
- `local variable`, for variable declarations within a function definition.
- `local struct`, for struct declarations within a function definition, assuming your parser can handle them.
- `member`, for members of a (local or global) struct, assuming you have implemented user-defined structs.

See Sections 5 and 6 for examples.

5 Basic example(s)

5.1 Input: `test1.c`

```
1  int x, y;
2  float z[50];
3
4  int foo(int z)
5  {
6    // Variable a is never declared? That's OK for now!
7    return a;
8  }
9
```

```
10  int bar(int a, int b)
11  {
12    // Local variable hides parameter, OK for now
13    int a;
14    // Incorrect parameter type is OK for now
15    a = foo(4.2);
16    for (i=0; i<10; i++) {
17      foo(i, 7);
18      // Incorrect number of parameters is OK for now
19    }
20    int foo;
21    // Incorrect assignment type is OK for now
22    foo = z * 2.5;
23    // break/continue not in a loop is OK for now
24    continue;
25    // Incorrect return type is OK for now
26    return 5.3;
27  }
28
29  int more, global[25], variables;
30
31  int test(int lots, int more, int useless)
32  {
33    char variables[15], just, to, show;
34    if (1) return;
35  }
36
37  int bar(int a, int b) // duplicate definition for bar? OK for now!
38  {
39    int d;
40    d = 0;
41    // Duplicate local variable is OK for now
42    int d;
43    while ( (d += ++a) < b);
44    return d;
45  }
46
47  int main()
48  {
49    float how, this, part, should, work;
50    return 7;
51  }
```

5.2 Output: `test1.parser`

```
File test1.c Line 1: global variable x
File test1.c Line 1: global variable y
File test1.c Line 2: global variable z
File test1.c Line 4: function foo
File test1.c Line 4: parameter z
File test1.c Line 10: function bar
File test1.c Line 10: parameter a
File test1.c Line 10: parameter b
File test1.c Line 13: local variable a
```

```
File test1.c Line 20: local variable foo
File test1.c Line 29: global variable more
File test1.c Line 29: global variable global
File test1.c Line 29: global variable variables
File test1.c Line 31: function test
File test1.c Line 31: parameter lots
File test1.c Line 31: parameter more
File test1.c Line 31: parameter useless
File test1.c Line 33: local variable variables
File test1.c Line 33: local variable just
File test1.c Line 33: local variable to
File test1.c Line 33: local variable show
File test1.c Line 37: function bar
File test1.c Line 37: parameter a
File test1.c Line 37: parameter b
File test1.c Line 39: local variable d
File test1.c Line 42: local variable d
File test1.c Line 47: function main
File test1.c Line 49: local variable how
File test1.c Line 49: local variable this
File test1.c Line 49: local variable part
File test1.c Line 49: local variable should
File test1.c Line 49: local variable work
```

5.3 Input: `test2.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    for (i=0;;i++) {
10     n/=2;
11     if (n) continue;
12     return i;
13   }
14   for (;;) {
15     return;
16   }
17 }
18
19 int too_many_elses(int a, int b)
20 {
21   if (a<b) {
22     return 1;
23   } else {
24     return 2;
25   } else {     /* This should cause a syntax error */
26     return 3;
27   }
28 }
```

5.4 Error stream for `test2.c`

```
Parser error in file test2.c line 25 at text else
       Expected identifier (within expression)
```

6 Extra credit example

6.1 Input: `test3.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 later.
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 }
```

6.2 Errors for basic implementation, on `test3.c`

```
Parser error in file test3.c line 1 at text const
       Expected function or global declaration
```

6.3 Output: `test3.parser` for extra credit implementation

```
File test3.c Line 1: global variable pi
File test3.c Line 3: global struct point
File test3.c Line 4: member x
File test3.c Line 4: member y
File test3.c Line 7: global struct rectangle
File test3.c Line 8: member upperleft
File test3.c Line 9: member lowerright
File test3.c Line 12: global struct window
File test3.c Line 13: member area
File test3.c Line 14: member text
File test3.c Line 17: function display
File test3.c Line 17: parameter W
File test3.c Line 17: parameter n
File test3.c Line 19: function strange
File test3.c Line 19: parameter z
File test3.c Line 25: local variable y
File test3.c Line 26: local struct mything
File test3.c Line 27: member a
File test3.c Line 27: member b
File test3.c Line 27: member c
File test3.c Line 29: local variable A
```

7 Grading

The grading script, `ParseTest.sh`, is published on the course git repository, along with LaTeX source for these specifications. Students are *strongly* encouraged to test their code using the script, especially on `pyrite.cs.iastate.edu`, before submission. The script works in the same way as the previous grading script, `LexTest.sh`.

Points	Description
17	**Documentation**
5	`README.txt` How to build the compiler and documentation. Updated to show which part 2 features are implemented.
12	`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.
8	**Ease of building** How easy was it for the graders to build your compiler and documentation from the `README` file. You are encouraged to use `Makefile`s under the `Source/` and `Documentation/` directories so that running "`make`" will build everything and running "`make clean`" will remove all generated files.
5	**Still works in mode 0**
5	**Still works in mode 1**
60	**Basic Parser**

5	Global variables
5	Function declarations / 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
30	**Extra features**
5	Function prototypes (c.f. Section 3.1)
5	Variable initializations (c.f. Section 3.2)
5	Constants (c.f. Section 3.3)
10	User-defined structs (c.f. Section 3.4)
5	Struct member selection (c.f. Section 3.5)
100	**Total for students in 440 (max points is 120)**
115	**Total for students in 540**

8 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.