COM S 440/540 Project part 3

Type checking

1 Requirements for part 3

When executed with a mode of 3, your compiler should read the specified input file and check that the file has correct C syntax (as done in part 2), and perform type checking on all expressions. For any type errors (discussed in Section 2), display an appropriate error message. As usual, these 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 error, your compiler may either make a "best effort" attempt to continue processing the input file, or exit. Your compiler should continue to catch and report errors at the lexing and parsing stages (i.e., errors from Parts 1 and 2 should remain).

If the input file is syntactically correct, then the output should be a report for each statement of the form

expression ;

indicating the type of the expression. See Section 3 for more details. If the input file contains errors, then the output should be any subset of the output (including no output at all) that would be obtained by ignoring the errors.

2 Type checking

2.1 Literals, identifiers, and functions

Each expression will have a corresponding type. For literals, the type may be inferred based on the literal: character literals have type `char`, integer literals have type `int`, real literals have type `float`, and string literals have type `char[]`. These may be modified by `const`, see Section 2.6. For global variables, local variables, and parameters, the type should match how they are declared. The return type of a function specifies the type of any call to that function. Regarding functions and identifiers, an appropriate error message should be given for

- using a variable that has not been declared;
- declaring a local variable with the same name as another local variable or parameter of the same function;
- declaring a parameter with the same name as another parameter in the same prototype;
- declaring a global variable with the same name as another global variable;
- calling a function that has not been defined or declared as a prototype;
- calling a function with the incorrect number of parameters;
- calling a function with passed parameters whose types do not *exactly* match the parameters given in its prototype (see Section 2.4 to relax this slightly);
- declaring two prototypes for the same function name that are not exactly the same (ignoring parameter names);

#	Operation	Result type
R1	N `?` T `:` T	T

#	Left type	operator	Right type	Result type
R2		`~`	I	I
R3		`-`	N	N
R4		`!`	N	`char`
R5		`(char)`	N	`char`
R6		`(int)`	N	`int`
R7		`(float)`	N	`float`
R8	I	`%`, `&`, `|`	I	I
R9	N	`+`, `-`, `*`, `/`	N	N
R10	N	`==`, `!=`, `>`, `>=`, `<`, `<=`, `&&`, `||`	N	`char`
R11		`++`, `--`	N	N
R12	N	`++`, `--`		N
R13	N	`=`, `+=`, `-=`, `*=`, `/=`	N	N
R14	T`[]`	`[ int ]`		T

Table 1: Operations on types. $I \in \{\texttt{char}, \texttt{int}\}$ is any integer type, $N \in \{\texttt{char}, \texttt{int}, \texttt{float}\}$ is any numeric type, and T is *any* type.

- a `return` statement inside a function, that returns a value with a different type than the function's type (do not worry yet about *missing* `return` statements);
- giving more than one function definition for a prototype.

Extra credit options in Sections 2.4 and following may adjust these rules or (more likely) introduce additional rules, to bring type checking closer to the C standard.

2.2 Expressions

All expressions should be checked for type errors. Table 1 shows how operators are defined for various types of operands, and the resulting type. Rules are numbered for later reference. It also shows which explicit coercions, or casts, are allowed (c.f. rules R5, R6, and R7). Note that the variables I, N, and T represent the same type within a rule; for example, rule R9 says the result of `char + char` has type `char`. For a minimal implementation, you may give an error if the operand types do not *exactly* match. For example, rule R9 says the operands of `+` must be the same type; thus an operation `int + char` would produce a type error. Rule R14 is for array indexing, and states both that it is an error to index an item that is not an array type, and that array indices must be integers (specifically, `int`s).

2.3 Statements

Statements that require a condition must check that the expression given for the condition is a numerical type (one of `char`, `int`, or `float`). Specifically, this means to check the following.

- The expression inside an `if` condition is numerical.
- The expression inside a `while` or `do`–`while` condition is numerical.
- The second expression in a `for` loop (that determines if the loop continues or not), *if present*, must be numerical.

Original	Widened
`char`	`int`
`int`	`float`

Table 2: Allowed type widenings.

2.4 Extra credit: widening

Relax the type checking rules slightly by automatically coercing types, by widening only (according to Table 2) as necessary. Note that more than one widening may be possible, and desired. For example, for operation `char + float`, the left operand of type `char` can be widened to `int` which can then be widened to `float`. Once the left operand has automatically been widened to type `float`, rule R9 indicates that the resulting type is `float`.

For extra credit, your compiler should automatically widen types any place it is needed: for operations shown in Table 1, for assignments, for parameter passing, and for function return values. For example, if we have the following function defined:

```
1    float widen(int a)
2    {
3      a += 'x';
4      return a;
5    }
```

then the expression in line 3 would automatically widen the character constant `'x'` to type `int`, and the return statement in line 4 would automatically widen variable `a` from type `int` to type `float` as required for the return type of function `widen()`. Calling the function as `widen('q')` would be allowed, as the character `'q'` can be automatically widened to type `int` as required.

2.5 Extra credit: initialization

Implement type checking for variable initialization. Note that this requires your parser to support variable initialization, which was extra credit for part 2. Note that several variables may be declared and initialized at once:

```
1    void foo(int a)
2    {
3      int b = a,     /* OK */
4          c,
5          d = 0,     /* OK */
6          e = d,     /* OK */
7          f = 3.14,  /* ERROR, type mismatch */
8          g;
9    }
```

2.6 Extra credit: constants

Implement type checking for the `const` modifier. Note that this requires your parser to support the `const` modifier, which was extra credit for part 2.

For this extra credit, an appropriate error message should be generated if any `const` item is used as an "l-value"; i.e., the left side of any assignment opperator in rule R13, and any item with an increment or decrement operator (rules R11 and R12) has the `const` property. The only exception is for variable initializations during declaration (if this has been impemented):

```
1     void bar(int a)
2     {
3       const int b = a; /* OK, declare and initialize */
4       const int c;
5       c = 4;           /* ERROR, assignment to a const */
6     }
```

Additionally, character, integer, real, and string literals should have their ordinary types but modified with `const`, and rules R1 through R10 in Table 1 should be updated so that, if all operands are `const`, then the result type should also be `const`. For example, the expressions `17`, `3+(4*5)`, and `(1>2)?3:4` should all have type `const int`.

2.7 Extra credit: user-defined structs

Implement type checking for user-defined structs. Note that this requires your parser to support user-defined structs, which was extra credit for part 2. This means that appropriate error messages should be given for the following.

- It is an error to define two global structs with the same name.
- It is an error to define two local structs with the same name.
- When declaring a global variable, local variable, or parameter that is a struct type, it is an error if that struct type has not been defined.

2.8 Extra credit: struct member selection

Implement type checking for member selection of structs. Note that this requires your parser to support member selection with the " . " operator, which was extra credit for part 2. Practically, it also requires you to implement user-defined structs as discussed in Section 2.7, otherwise this will be impossible to test.

This means to implement the following type checking rule

#	Operation	Result type
R15	S . m	T

where T is the type of member "m" in struct S. That is, for any expression of the form "item.m":

- It is a type error if "item" is not a `struct` type.
- It is a type error if the struct "item" does not have a member named "m".
- Otherwise, "item.m" has type as given by the type of member "m", as defined in the struct corresponding to "item".

Note that this is complicated by the fact that "item" may also involve member selection and array indexing. This portion of extra credit is to check expressions of the form

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

3 Format of output

The output is similar to Part 2, but with type information. Specifically, the output should contain the following, in order.

1. For each user-defined global `struct` in declaration order, write the line

   `Global struct` *structname*

(with the correct *structname*). Then write the type and name of each struct member, each on a separate line, in the same order they were declared, followed by a blank line. If you did not implement user-defined structs as discussed in Section 2.7, then there will be no output here.

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

   `Global variables`

   followed by the type and name of each global variable, each on a separate line, in the same order they were declared, followed by a blank line. If variable initialization is implemented for extra credit (see Section 2.5), write "`(initialized)`" after each global variable with an initial value.

3. For each function definition (i.e., with an implementation), write the line

   `Function` *funcname*, returns *rtype*

   with the correct function name *funcname* and return type *rtype*. Then write the following, in order.

   (a) Write a line containing "`Parameters`". Then write the type and name of each global variable, each on a separate line, in the same order they were declared, followed by a blank line.

   (b) For each user-defined local `struct` (none, if you did not implement the extra credit discussed in Section 2.7), in declaration order, write the line "`Local struct` *structname*", with the correct *structname*. Then write the type and name of each struct member, each on a separate line, in the same order they were declared, followed by a blank line.

   (c) Write a line containing "`Local variables`". Then write the type and name of each local variable, each on a separate line, in the same order they were declared, followed by a blank line. If variable initialization is implemented for extra credit (see Section 2.5), write "`(initialized)`" after each local variable with an initial value.

   (d) Write a line containing "`Statements`". Then, for each statement in the body of the function of the form

   expression ;

   in order, write a line containing "Expression on line *lineno* has type *type*", where *lineno* is the line number of the "`;`" in the input file, and *type* is the name of the type for the expression.

You may indent lines with varying amount of whitespace, to make the output easier to read. 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: `test1.c`

```
1  int global, A[50];
2
3  void print(int x);
4
5  int putc(int a);
6
7  int f(int a)
8  {
9    a = b;              /* error */
10   a + "nope";         /* error */
11   a += putc(10+a/2);
12   print(42);
```

```
13    return a+3;
14  }
15
16  float g(int x, int y)
17  {
18    int z;
19    z = f(x) + A[putc(f(y))];
20    while (print(3)) {  /* error */
21      z++;
22      return 4.5;
23    }
24    if (z < 12) {
25      return;           /* error */
26    }
27    g(x-1, y-1);
28    return 7;           /* error */
29  }
30
31  void h()
32  {
33    f('c');             /* error for minimal implementation */
34    f(3, 4, 5);         /* error */
35    g(6);               /* error */
36    h(7);               /* error */
37    i("not defined");   /* error */
38    A[g(1,2)];          /* error */
39  }
```

4.2 Output for `mycc -3 test1.c`

```
Global variables
       int global
       int[] A

Function f, returns int
    Parameters
       int a

    Local variables

    Statements
       Expression on line 11 has type int
       Expression on line 12 has type void

Function g, returns float
    Parameters
       int x
       int y

    Local variables
       int z

    Statements
       Expression on line 19 has type int
```

```
        Expression on line 21 has type int
        Expression on line 27 has type float

Function h, returns void
    Parameters

    Local variables

    Statements
```

4.3 Errors for `mycc -3 test1.c`

```
Error near test1.c line 9
       Undeclared identifier: b
Error near test1.c line 9
       Operation not supported: int = error
Error near test1.c line 10
       Operation not supported: int + char[]
Error near test1.c line 20
       Condition of while loop has invalid type: void
Error near test1.c line 25
       Returning void in a function of type float
Error near test1.c line 28
       Returning int in a function of type float
Error near test1.c line 33
       Parameter mismatch in function call
       f(char)
Error near test1.c line 34
       Parameter mismatch in function call
       f(int, int, int)
Error near test1.c line 35
       Parameter mismatch in function call
       g(int)
Error near test1.c line 36
       Parameter mismatch in function call
       h(int)
Error near test1.c line 37
       Function i has not been declared.
Error near test1.c line 38
       Array index should be an integer (was: float)
```

4.4 Discussion for `test1.c`

For this implementation, any expression statement with an error is omitted from the output. For example, the statements on lines 9 and 10 would be displayed in the output for function `f()` if they did not contain errors. You may either omit these from output, or include them with a type of "`error`". Also, note that the first error (missing identifier `b`) causes the second error (assuming `b` would have been the correct type), as a single error can have a cascade effect. Your implemenation may stop after the first N errors, where N can be a fixed positive integer of your choice, or infinity.

5 Examples: all extra credit implemented

5.1 Input: `widen.c`

```
1
2  int A[100];
3
4  float fact(int n)
5  {
6    float f;
7    if (n<2) {
8      return 1;      // 1 is an int, widened to float here
9    } else {
10     f = n * fact(n-1);  // n should be widened to float here
11     return f;
12   }
13 }
14
15 void tests()
16 {
17   char c;
18   int i;
19   float f;
20
21   c = 'a';
22   i = 'b';   // char -> int
23   f = i;     // int -> float
24   f = 'c';   // char -> float
25
26   fact(c);          // char -> int
27   fact(f ? i : c);  // char -> int
28   fact(A['4']);     // array index char -> int
29 }
```

5.2 Output for `mycc -3 widen.c`

```
Global variables
    int[] A

Function fact, returns float
    Parameters
        int n

    Local variables
        float f

    Statements
        Expression on line 10 has type float

Function tests, returns void
    Parameters

    Local variables
        char c
        int i
        float f

    Statements
```

```
Expression on line 21 has type char
Expression on line 22 has type int
Expression on line 23 has type float
Expression on line 24 has type float
Expression on line 26 has type float
Expression on line 27 has type float
Expression on line 28 has type float
```

5.3 Errors for `mycc -3 widen.c`

There are no errors.

5.4 Input: `consts.c`

```
1
2  const float pi = 3.14159265358979323846264338327950288419 71; /* approximately */
3
4  void zero(int A[], const int N)
5  {
6    int i = 0;
7
8    N;
9    N+1;
10   i >= N;
11   N > 3;
12
13   for (;;) {
14     A[i] = 0;
15     i++;
16     if (i>=N) break;
17   }
18 }
19
20 void OK()
21 {
22   int primes[100];
23   int size = 100;
24   zero(primes, size);
25 }
26
27 void illegal()
28 {
29   const int i;
30   const int j = 4;
31
32   i = 7; /* nope */
33   j++;   /* nope */
34
35   pi /= 2.0; /* nope */
36 }
```

5.5 Output for `mycc -3 consts.c`

```
Global variables
```

```
    const float pi (initialized)

Function zero, returns void
    Parameters
        int[] A
        const int N

    Local variables
        int i (initialized)

    Statements
        Expression on line 8 has type const int
        Expression on line 9 has type const int
        Expression on line 10 has type char
        Expression on line 11 has type const char
        Expression on line 14 has type int
        Expression on line 15 has type int

Function OK, returns void
    Parameters

    Local variables
        int[] primes
        int size (initialized)

    Statements
        Expression on line 24 has type void

Function illegal, returns void
    Parameters

    Local variables
        const int i
        const int j (initialized)

    Statements
```

5.6 Error for `mycc -3 consts.c`

```
Error near consts.c line 32
       Cannot assign to item of type const int
Error near consts.c line 33
       Cannot increment lvalue of type const int
Error near consts.c line 35
       Cannot assign to item of type const float
```

5.7 Input: `struct1.c`

```
1
2  struct pair {
3    int x, y;
4  };
5
6  void func1()
```

```
 7  {
 8    struct triple {
 9      struct pair p;
10      int z;
11    };
12
13    struct pair P;
14    struct triple T;
15
16    P;
17    T;
18  }
19
20  struct pair { int x, y; }; // Error
21
22  void func2()
23  {
24    struct pair P;
25    struct triple T;  // Error
26
27    P;
28    T;
29  }
```

5.8 Output for `mycc -3 struct1.c`

```
Global struct pair
    int x
    int y

Global variables

Function func1, returns void
    Parameters

    Local struct triple
        struct pair p
        int z

    Local variables
        struct pair P
        struct triple T

    Statements
        Expression on line 16 has type struct pair
        Expression on line 17 has type struct triple

Function func2, returns void
    Parameters

    Local variables
        struct pair P

    Statements
```

```
        Expression on line 27 has type struct pair
```

5.9 Error for `mycc -3 struct1.c`

```
Error near struct1.c line 20
       struct pair already defined near struct1.c line 4
Error near struct1.c line 25
       No definition for 'struct triple'
Error near struct1.c line 28
       Undeclared identifier: T
```

5.10 Input: `struct2.c`

```
1
2  struct pair {
3    int x, y;
4  };
5
6  struct triple {
7    struct pair p;
8    int z;
9  };
10
11 void test()
12 {
13   struct triple T;
14   T;
15   T.p;
16   T.z;
17   T.p.x;
18   T.p.y;
19   T.nope;    // Error
20   T.p.nope;  // Error
21   T.p.x.nope; // Error
22 }
```

5.11 Output for `mycc -3 struct2.c`

```
Global struct pair
    int x
    int y

Global struct triple
    struct pair p
    int z

Global variables

Function test, returns void
    Parameters

    Local variables
        struct triple T

    Statements
```

```
Expression on line 14 has type struct triple
Expression on line 15 has type struct pair
Expression on line 16 has type int
Expression on line 17 has type int
Expression on line 18 has type int
```

5.12 Error for `mycc -3 struct2.c`

```
Error near struct2.c line 19
       Base type struct triple has no member named nope
Error near struct2.c line 20
       Base type struct pair has no member named nope
Error near struct2.c line 21
       Base type int is not a struct
```

6 Grading

Points	Description
16	**Documentation**
4	`README.txt` How to build the compiler and documentation. Updated to show which part 3 features are implemented.
12	`developers.pdf` New section for part 3, that explains the purpose of each source file, the main data structures used (or how they were updated), and gives a high-level overview of how the various features are implemented.
8	**Ease of grading**
4	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.
4	Works with script Does the `-o` switch work? Is your output formatted correctly?
10	**Still works in modes 0, 1, and 2** Take care with mode 2, so that you can display local variables even if they have the same names, and functions with the same names but different prototypes. You might need completely separate symbol tables for modes 2 and 3.
66	**Type checking**
4	Literals
8	Identifiers (global variables, local variables, parameters)
6	Function calls
4	Function returns
4	Detects duplicate or mismatched function prototypes
4	Unary operators
4	Casts
6	Binary operators: arithmetic

6	Binary operators: comparison and logic
4	Assignment and update operators
4	Increment and decrement
4	Array indexing
4	Ternary operator
4	Conditions are numerical (see Section 2.3)
10	**Extra credit: widening** See Section 2.4.
5	**Extra credit: initializations** See Section 2.5.
5	**Extra credit: constants** See Section 2.6.
10	**Extra credit: user-defined structs** See Section 2.7
5	**Extra credit: struct member selection** See Section 2.8.
100	**Total for students in 440 (max points is 120)**
115	**Total for students in 540**

7 Submission

Part	Penalty applied
Part 0	30% off
Part 1	20% off
Part 2	10% off

Table 4: 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 4 for penalty information). Otherwise, we will grade only part 3.