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), write an error message to standard error, in the format

> `Type checking error in file` *filename* `line` *line number*
> *Description*

where the *Description* should be as detailed as possible. 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). After the first error, your compiler may either make a "best effort" attempt to continue processing the input file, or exit.

If there are no errors, then your compiler should create an output file with extension `.types`, that gives, for each statement of the form

> *expression* ;

the type of the expression. See Section 4 for more details.

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 3.4. 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

- declaring a variable, parameter, or struct member of type `void`;
- 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 3.2 to relax this slightly);

#	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	`=`	T	T
R15	T`[]`	`[ int ]`		T

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

- 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 definition for the same function name.

Note that these type checking rules are more strict than standard C. Extra features in Section 3 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 R15 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).

3 Extra features

3.1 Prototype checking

Relax the type checking rules to allow function prototypes. Note that this requires your parser to support function prototypes, which was extra for part 2.

Specifically, your compiler should allow any number of function prototypes for a given function, as long as the return types, number of parameters, and parameter types are consistent. In other words, prototypes

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

Table 2: Allowed type widenings.

for the same function name must match perfectly, except for parameter names. A type checking error should be given otherwise.

In terms of type checking, presence of a function prototype does not require that a function definition also appears. Relative to function prototypes, the corresponding function definition may appear before or after any of its prototypes, or not appear at all.

3.2 Automatic type 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 this extra feature, 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.

3.3 Variable initialization

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

```
void foo(int a)
{
  int b = a,     /* OK */
      c,
      d = 0,     /* OK */
      e = d,     /* OK */
      f = 3.14,  /* ERROR, type mismatch */
      g;
}
```

3.4 Enforcing constants

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

For this extra feature, an appropriate error message should be generated for attempted modifications of any `const` item. This says the left side of any assignment operator (rules R13 and R14) cannot be `const`, and items with an increment or decrement operator (rules R11 and R12) cannot be `const`. The only exception is for variable initializations during declaration (if this has been implemented):

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

Additionally, when determining (and displaying) the type of an expression, the following adjustments should be made.

- The type of a string literal should be `const char[]`.
- The type displayed for a variable should be `const` if it was declared `const`.
- The type displayed for an array element should be `const` if the array is `const`.
- All other operator rules (R1 through R10) should discard `const` information. For example, it is legal to add a `const int` to a `const int` or to an `int`, and in both cases the resulting expression should have type `int`.

3.5 User-defined structs

Implement type checking for user-defined structs. Note that this requires your parser to support user-defined structs, which was extra 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.

3.6 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 for part 2. Practically, it also requires you to implement user-defined structs as discussed in Section 3.5, otherwise this will be impossible to test.

This means to implement the following type checking rule

#	Operation	Result type
R16	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".
- If a struct variable is `const`, then all of its members become `const` (for students who implemented `const`).

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

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

4 Format of output

For each statement of the form

expression ;

write a line of the form

`File` *filename* `Line` *linenum*`:` `expression has type` *type*

to the output stream. The *filename* and *linenum* given should be the location of the expression statement (specifically, the location of the ";") and *type* should be the type of the expression, as one of the four basic types `void`, `char`, `int`, or `float`, or a user-defined `struct structname` (if that extra feature has been implemented), followed by "`[]`" if it is an array. For example, the source lines from a file `input.c`

```
23      putchar(72);
24      3.14159265 / 2.0;
25      "Hello, world!";
```

should generate the lines

```
File input.c Line 23: expression has type int
File input.c Line 24: expression has type float
File input.c Line 25: expression has type char[]
```

in the output file `input.types`.

5 Basic examples

5.1 Input: `test1.c`

```
1
2  float A;
3
4  int f(int A[], int n)
5  {
6      A;
7      A[3];
8      A[n] = A[n-1] + A[n-2];
9
10     return n;
11 }
12
13 int printf(char msg[])
14 {
15     return 0;
16 }
17
18 void g()
```

```
19 {
20     A;
21     int b[50];
22     f(b, 5) + printf("Hello world!\n");
23     int c;
24     c = b[b[0]++]++;
25     return;
26 }
```

5.2 Output: test1.types

```
File test1.c Line 6: expression has type int[]
File test1.c Line 7: expression has type int
File test1.c Line 8: expression has type int
File test1.c Line 20: expression has type float
File test1.c Line 22: expression has type int
File test1.c Line 24: expression has type int
```

5.3 Input: test2.c

```
1
2  int main()
3  {
4      printf("Hello, world!\n");
5      return 0;
6  }
```

5.4 Error for test2.c

```
Type checking error in file test2.c line 4
  Unknown function 'printf'
```

5.5 Input: test3.c

```
1
2  int foo(int x, float y)
3  {
4      x;
5      y;
6      x+3;
7      2.5-y;
8      x*(int)y;
9      return x ? foo(x-1, y) : --x;
10 }
11
12 int bar()
13 {
14     foo(3); /* type error */
15 }
```

5.6 Error for test3.c

```
Type checking error in file test3.c line 14
  Parameter mismatch in call to foo(int, float)
  Expected 2, received 1 parameters
```

6 Extra feature examples

6.1 Input: extra3.c

```
1  int foo(char a);
2
3  int foo(char x)
4  {
5      int a, b=x;
6      int const c=0;
7      return c++;
8  }
9
10 int foo(char b);
```

6.2 Error for extra3.c

```
Type checking error in file extra3.c line 7
  Invalid operation: const int ++
```

6.3 Input: extra4.c

```
1
2  struct pair {
3      int x, y;
4  };
5
6  struct triangle {
7      char color;
8      struct pair vertex[3];
9  };
10
11 void foo()
12 {
13     struct pair p, q;
14
15     p;
16     p.x;
17     q.y / p.x;
18
19     struct ornament {
20         struct pair hook;
21         int length;
22         struct triangle faces[24];
23     };
24
25     const struct ornament t;
26
27     t;
28     t.hook;
29     t.length;
30     t.faces;
31     t.faces[5];
32     t.faces[5].color;
```

```
33      t.faces[5].vertex[1];
34      t.faces[5].vertex[1].x;
35      "Bye!";
36  }
```

6.4 Output: `extra4.types`

```
File extra4.c Line 15: expression has type struct pair
File extra4.c Line 16: expression has type int
File extra4.c Line 17: expression has type int
File extra4.c Line 27: expression has type const struct ornament
File extra4.c Line 28: expression has type const struct pair
File extra4.c Line 29: expression has type const int
File extra4.c Line 30: expression has type const struct triangle[]
File extra4.c Line 31: expression has type const struct triangle
File extra4.c Line 32: expression has type const char
File extra4.c Line 33: expression has type const struct pair
File extra4.c Line 34: expression has type const int
File extra4.c Line 35: expression has type const char[]
```

7 Grading

Our testing script, `TypesTest.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 scripts, `LexTest.sh` and `ParseTest.sh`.

Points	Description
15	**Documentation**
3	`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.
6	**Ease of 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.
8	**Still works in modes 0, 1, and 2** We will test earlier modes only using source files that are syntactically correct, with no type errors.
66	**Type checking**
4	Literals
10	Identifiers (global variables, local variables, parameters)
10	Function calls
4	Function returns

	4	Unary operators
	4	Casts
	6	Binary operators: arithmetic
	6	Binary operators: comparison and logic
	6	Assignment and update operators
	4	Increment and decrement
	4	Array indexing
	4	Ternary operator
35		**Extra features**
	5	Prototype checking (c.f. Section 3.1)
	5	Automatic widening (c.f. Section 3.2)
	5	Variable initializations (c.f. Section 3.3)
	5	Constants (c.f. Section 3.4)
	5	User-defined structs (c.f. Section 3.5)
	5	Struct member selection (c.f. Section 3.6)
	5	`const` with `struct`
100		**Total for students in 440 (max points is 120)**
115		**Total for students in 540**

8 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.