COM S 440/540 Project part 3

Compiler — type checking

1 Basic requirements

Your compiler at this point must be able to parse and type check input programs written in our subset of C. For every statement in the input file of the form "expression semicolon", you should display to standard output the filename, linenumber, and type of the expression in the following format:

```
Expression at file <filename> line <linenumber> has type <type>
```

(see examples later). Additionally, you must display appropriate error messages (including the file name and line number where the error occurs) to standard error, for lexer errors (cannot open files, bad tokens, unclosed comments), syntax errors, and type mismatch errors for:

- assignment statements `x = expr`.
- function calls (passed parameter values must match number and type of formal parameters).
- `return` statements (the return value type should match the function type).

See the discussion below for more details about our type system. You must also give errors for undeclared variables and functions. After the first error, you may make a "best effort" to continue processing the input file, or exit. As usual, your executable should be named `compile`, and the input file(s) should be specified on the command line, with the type checker invoked using the `-t` switch:

```
prompt$ ./compile -t infile.c
```

2 Our type system

Table 1 and Table 2 show how operators are defined for various types of operands, and the resulting type. It also shows which explicit coercions, or casts, are allowed. For example, in the table, the entry for operation "`char + char`" has result "`char`", indicating that operator `+` can be applied when the left and right operands each have type `char`, and the resulting expression has type `char`. If types are not listed for an operator, then the operator cannot be applied to those types. For example, we cannot apply `+` to two expressions of type `char[]` (character arrays). Note that a string literal should have type `char[]`.

2.1 Minimal implementation

All coercions are explicit; there are no automatic type widenings. This means that, for example, the assignment

```
float x;
x = 3;
```

will cause a type mismatch error.

Operation			Result
	-	char	char
	!	char	char
	~	char	char
	&	char	char[]
char[]	[]		char
char	+	char	char
char	-	char	char
char	*	char	char
char	/	char	char
char	%	char	char
char	|	char	char
char	&	char	char
char	==	char	char
char	!=	char	char
char	>	char	char
char	>=	char	char
char	<	char	char
char	<=	char	char
char	||	char	char
char	&&	char	char
char	++		char
	++	char	char
char	--		char
	--	char	char
char	=	char	char
char	+=	char	char
char	-=	char	char
char	*=	char	char
char	/=	char	char

Operation			Result
	-	int	int
	!	int	char
	~	int	int
	&	int	int[]
int[]	[]		int
int	+	int	int
int	-	int	int
int	*	int	int
int	/	int	int
int	%	int	int
int	|	int	int
int	&	int	int
int	==	int	char
int	!=	int	char
int	>	int	char
int	>=	int	char
int	<	int	char
int	<=	int	char
int	||	int	char
int	&&	int	char
int	++		int
	++	int	int
int	--		int
	--	int	int
int	=	int	int
int	+=	int	int
int	-=	int	int
int	*=	int	int
int	/=	int	int

Operation			Result
	-	float	float
	!	float	char
	&	float	float[]
float[]	[]		float
float	+	float	float
float	-	float	float
float	*	float	float
float	/	float	float
float	==	float	char
float	!=	float	char
float	>	float	char
float	>=	float	char
float	<	float	char
float	<=	float	char
float	||	float	char
float	&&	float	char
float	++		float
	++	float	float
float	--		float
	--	float	float
float	=	float	float
float	+=	float	float
float	-=	float	float
float	*=	float	float
float	/=	float	float

Table 1: Operations on types

Operation					Result
`char`	`?`	`char`	`:`	`char`	`char`
`int`	`?`	`char`	`:`	`char`	`char`
`float`	`?`	`char`	`:`	`char`	`char`
`char`	`?`	`int`	`:`	`int`	`int`
`int`	`?`	`int`	`:`	`int`	`int`
`float`	`?`	`int`	`:`	`int`	`int`
`char`	`?`	`float`	`:`	`float`	`float`
`int`	`?`	`float`	`:`	`float`	`float`
`float`	`?`	`float`	`:`	`float`	`float`
`char`	`?`	`char[]`	`:`	`char[]`	`char[]`
`int`	`?`	`char[]`	`:`	`char[]`	`char[]`
`float`	`?`	`char[]`	`:`	`char[]`	`char[]`
`char`	`?`	`int[]`	`:`	`int[]`	`int[]`
`int`	`?`	`int[]`	`:`	`int[]`	`int[]`
`float`	`?`	`int[]`	`:`	`int[]`	`int[]`
`char`	`?`	`float[]`	`:`	`float[]`	`float[]`
`int`	`?`	`float[]`	`:`	`float[]`	`float[]`
`float`	`?`	`float[]`	`:`	`float[]`	`float[]`

Operation		Result
`(char)`	`char`	`char`
`(char)`	`int`	`char`
`(char)`	`float`	`char`
`(int)`	`char`	`int`
`(int)`	`int`	`int`
`(int)`	`float`	`int`
`(float)`	`char`	`float`
`(float)`	`int`	`float`
`(float)`	`float`	`float`

Table 2: More operations, and casts on types

2.2 Extra credit

Your compiler should automatically coerce types (by widening only) as necessary. The following widenings are allowed:

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

More than one widening is possible. For example, for operation `char + float`, the type `char` can be widened to `int` which can be widened to `float`, giving an overall type of `float`.

3 Function overloading

3.1 Minimal implementation

It is an error to define more than one function with the same name. (However, the same prototype may be given more than once.)

3.2 Extra credit

Two or more functions with the same name are allowed, if they have different numbers or types of parameters. If you implement implicit coercions (widenings), this can give rise to ambiguous function calls, which should generate an error. For example, if functions

```
float f(int a, float b);
float f(float a, int b);
```

are declared, then the function call

```
f(3, 5);
```

is ambiguous, and should generate an approprate error message.

4 Grading

Basic compiler for a single input file	60 points
Ease of building the executable	10 points
This includes your `README.txt` documentation and how well you use `make`.	
Developer documentation (`developers.tex`)	20 points
Implicit coercions (at most one of the following)	
NO: Type mismatch if types do not match perfectly	5 points
YES: Automatic type widening as needed	25 points
Function overloading (at most one of the following)	
NO: Error if same function name has different parameter types	5 points
YES: Functions can be overloaded based on parameter numbers / types	25 points
Total points for this part of the project (any excess earned is extra credit)	
Students in 440	100 points
Students in 540	120 points

5 Submitting

Create a tag named `Part3` and push it to your remote repository. Also, upload your `developer.pdf` documentation to Canvas.

6 Examples

6.1 Input file (`types.c`)

```
1
2  int global_a;
3
4  int f(int a)
5  {
6    return a+3;
7  }
8
9  void g(int a)
10 {
11   global_a = a;
12 }
13
14 int main()
15 {
16   int a, b[4], c;
17   float z;
```

```
18    int b;
19    4;
20    -6.4;
21    - "hello";
22    4 + 'c';
23    5 / 1.0;
24    "hello" + "world";
25    3 ? "hello" : "world";
26    f('c');
27    g(c+4);
28    h(3);
29    f(3, 4, 5);
30  }
```

6.2 Output, minimal implementation

```
Error in types.c line 18:
       local variable b already declared as:
         int b[4] (near line 16 in file types.c)
Error in types.c line 21:
       Unary operator - cannot be applied to expression of type char[]
Error in types.c line 22:
       Operation not supported: int + char
Error in types.c line 23:
       Operation not supported: int / float
Error in types.c line 24:
       Operation not supported: char[] + char[]
Error in types.c line 26:
       No match for function call f(char)
       Candidates are:
           int f(int) declared in types.c near line 4
Error in types.c line 28:
       No match for function call h(int)
       No functions with this name
Error in types.c line 29:
       No match for function call f(int, int, int)
       Candidates are:
           int f(int) declared in types.c near line 4
Expression at line 11 has type int
Expression at line 19 has type int
Expression at line 20 has type float
Expression at line 25 has type char[]
Expression at line 27 has type void
```

6.3 Output, maximal implementation

```
Error in types.c line 18:
       local variable b already declared as:
         int b[4] (near line 16 in file types.c)
Error in types.c line 21:
       Unary operator - cannot be applied to expression of type char[]
Error in types.c line 24:
       Operation not supported: char[] + char[]
Error in types.c line 28:
```

```
       No match for function call h(int)
       No functions with this name
Error in types.c line 29:
       No match for function call f(int, int, int)
       Candidates are:
           int f(int) declared in types.c near line 4
Expression at line 11 has type int
Expression at line 19 has type int
Expression at line 20 has type float
Expression at line 22 has type int
Expression at line 23 has type float
Expression at line 25 has type char[]
Expression at line 26 has type int
Expression at line 27 has type void
```