CSC 435: COMPILER CONSTRUCTION
ASSIGNMENT #2: TYPE CHECKING

1. Introduction

For Assignment 2, you will extend your compiler to typecheck the Abstract Syntax Trees (AST) produced by the parser.

Your compiler will be given a mark equal to the number of test cases that are successfully checked by it. Some testcases have been written to check correctly, others have been designed to fail. When a testcase fails, your compiler must produce an informative error message stating the location and reason for the failure.

Your compiler does not have to produce multiple error messages, but may terminate gracefully after the first error has been reported. If your compiler does produce multiple error messages, they must be ordered by line number.

2. Existence and Uniqueness Checks

2.1. Global.

(1) A program must contain at least one function.
(2) One function must be called `main`, and
   (a) must take no parameters, and
   (b) must have a return "type" of `void`.
(3) No two functions may have the same name.

2.2. Local.

(1) No two parameters of a function may have the same name.
(2) No two local variables declared in a function may have the same name.
(3) No parameter may have a "type" of `void`.
(4) No local variable may have a "type" of `void`.
(5) A function parameter may hide the name of the function. For example,

```
int
foo(int foo)
{
        return foo+1;
}
```

(6) A local variable may *not* hide the name of a parameter.
(7) Each local variable must be defined before being used.

3. Type Checks

Definition 1 (Type Equivalence)**.** Type equivalence is determined structurally. That is,

(1) The following (atomic) types must compare exactly:
   (a) `int`
   (b) `float`
   (c) `char`
   (d) `string`
   (e) `boolean`

(2) For compound types (i.e., array) the following must match:
   (a) The underlying atomic type (as specified above)
   (b) The length

We indicate type equivalence, as defined above, with the notation

$$\mathsf{type}(e_1) = \mathsf{type}(e_2)$$

where e_1 and e_2 are type expressions.

Definition 2 (Subtype)**.** Type t_1 is a subtype of a type t_2 if and only if all values represented by type t_1 can be represented by type t_2.

(1) To obtain **bonus marks**, consider the subtyping relationship ($<$) to be given by the single relation

$$\texttt{int} < \texttt{float}$$

(2) Otherwise, consider the subtyping relationship *empty*, so that no automatic conversions are allowed.

In combination with the $=$ equivalence defined above, the relationship $\leq$ is implied.

3.1. Statements.

(1) An expression e used as an `if`-statement condition must have

$$\mathsf{type}(e) = \texttt{boolean}$$

(2) An expression e used as an `while`-statement condition must have

$$\mathsf{type}(e) = \texttt{boolean}$$

(3) An expression e used in a `print`-statement must have

$$\mathsf{type}(e) \in \{\texttt{int}, \texttt{float}, \texttt{char}, \texttt{string}, \texttt{boolean}\}$$

(4) An expression e used in a `println`-statement must have

$$\mathsf{type}(e) \in \{\texttt{int}, \texttt{float}, \texttt{char}, \texttt{string}, \texttt{boolean}\}$$

(5) The type of an expression used in a `return` statement must be a subtype of the return type of the containing function.

(6) For an assignment e_1=e_2,

$$\mathsf{type}(e_1) \geq \mathsf{type}(e_2)$$

3.2. Expressions.

(1) For an array index expression e,

$$\mathsf{type}(e) = \texttt{int}$$

(2) For an expression $e_1 \oplus e_2$:
   (a) If $\oplus$ is `+`:

	`int`	`float`	`char`	`string`	`boolean`	`void`
`int`	`int`					
`float`		`float`				
`char`			`char`			
`string`				`string`		
`boolean`						
`void`						

   (b) If $\oplus$ is `-`:

	`int`	`float`	`char`	`string`	`boolean`	`void`
`int`	`int`					
`float`		`float`				
`char`			`char`			
`string`						
`boolean`						
`void`						

(c) If $\oplus$ is `*`:

	`int`	`float`	`char`	`string`	`boolean`	`void`
`int`	`int`					
`float`		`float`				
`char`						
`string`						
`boolean`						
`void`						

(d) If $\oplus$ is `<`:

	`int`	`float`	`char`	`string`	`boolean`	`void`
`int`	`boolean`					
`float`		`boolean`				
`char`			`boolean`			
`string`				`boolean`		
`boolean`					`boolean`	
`void`						

(e) If $\oplus$ is `==`:

	`int`	`float`	`char`	`string`	`boolean`	`void`
`int`	`boolean`					
`float`		`boolean`				
`char`			`boolean`			
`string`				`boolean`		
`boolean`					`boolean`	
`void`						

(3) For an invocation of function f,

(a) The number of arguments in the invocation must match the number of parameters n, in the function f.

(b) Denote the parameters of f as p_i, $i \leq n$. Similarly, denote the arguments of a call to f as a_i, $i \leq n$. For each i, we must have

$$\mathsf{type}(a_i) \leq \mathsf{type}(p_i)$$