CSc 453: Semantic Restrictions for Clike

Declarations

The following rules guide the processing of declarations. Here the definition of a function refers to the specification of its formals, locals, and its body.

1. An identifier may be declared at most once as a global, and at most once as a local in any particular function; however, an identifier may appear as a local in many different functions. (Even if there are multiple braces within a function; this is different than C where a new scope (braces) can define a new local with the same name. In clike, we only allow one local of a name per function.)

2. A function may have at most one prototype; a function may be defined at most once.

3. If the type of the return value of a function if not specified explicitly, it defaults to int.

4. If a function has a prototype, then the types of the formals at its definition must match, in number and order, the types of the arguments in the prototype; and the type of the return value must match the return value at its prototype.

   The prototype, if present, should precede the definition of the function.

5. An identifier can occur at most once in the list of formal parameters in a function definition.

6. There must be exactly one type declaration for each identifier listed as a formal definition: this is the list that comes before the function body (and locals). The formals may have their type declared in any order. Each declared formal must also appear in the list of formal parameters.

7. The formal parameters of a function have scope local to that function.

Type Consistency Requirements

Variables must be declared before they are used. Functions must have their argument types and return value specified (either via a prototype or via a definition) before they are called.

If an identifier is declared to have scope local to a function, then all uses of that identifier within that function refer to this local entity; if an identifier is not declared as local to a function, but it is declared as a global, then any use of of that identifier within that function refers to the entity with global scope. Note that local scope takes precedence over global scope.

The following rules guide the checking of type consistency. Here, a type $t_1$ is considered to be compatible with a type $t_2$ if and only if (i) $t_1=t_2$ or (ii) $t_1=int$ and $t_2=char$ or vice-versa. (Yes, this means doubles and ints are NOT compatible unless you explicitly will call a function to convert).

Function Definitions

1. Any function called within an expression must not have a return type void. Any function call that is a statement must have return type void.

2. A function whose return type is void can return a value; i.e., it may not contain a statement of the form:

   ```
   return expr;
   ```

A function whose return type is NOT void cannot contain a statement of the form:

```
return;
```

Such functions (i.e., whose return type is not void) must contain at least one statement of the form:
Expressions

The type of an expression $e$ is given by the following:

1. If $e$ is an integer constant, then its type is int. If $e$ is a float constant, then its type is float.
2. If $e$ is an identifier, then the type of $e$ is type of that identifier; if $e$ is an array element, then the type of $e$ is the type of the elements of that array.
3. If $e$ is a function call, then the type of $e$ is the return type for that function.
4. If $e$ is an expression of the form:

$$e_1 + e_2, \quad e_1 - e_2, \quad e_1 * e_2, \quad e_1 / e_2, \quad -e$$

Then $e_1$ and $e_2$ can only of char, int, or double types. (Using bool type here is an error). Also, $e_1$ and $e_2$ must be type compatible. If either $e_1$ and $e_2$ are of type char, then one or both are promoted to int and the result is an int. If both $e_1$ and $e_2$ are the same type $t$, (i.e., of type int and double), the resultant type is $t$. To clarify, int and double are not type compatible in this context.
5. If an expression is of the form:

$$e_1 \geq e_2, \quad e_1 \leq e_2, \quad e_1 > e_2, \quad e_1 < e_2, \quad e_1 \neq e_2, \quad e_1 == e_2$$

Then the type of the expression is bool. $e_1$ and $e_2$ must be type-compatible.
6. If $e$ is an expression of the form:

$$e_1 \&\& e_2, \quad e_1 || e_2, \quad ! e$$

Then the type of $e$ is bool.

The rules for type checking expression are given by the following:

1. The type of the index in an array reference must be compatible with int.
2. Each actual parameter of a function call must be compatible with its corresponding formal parameter.
3. In the subexpressions associated with the operators:

$$+, -, *, /, <=, >=, <, >, ==, !=$$

the types must be compatible (and bool is not allowed).

The subexpression associated with the operators:

$$\&\&, ||, !$$

must have types compatible with bool.
Statements

1. Only variables of type `char`, `int`, and `double` or elements of arrays, can be assigned to; the type of the right hand side of an assignment must be compatible with the type of the left hand side of that assignment.

2. The type of the expression in a `return` statement must be compatible with the return type of the function.

3. The type of a conditional in a `if`, `for` or `while` must have type `bool`.

4. Each actual parameter of a function call must be compatible with the corresponding formal parameter.