COMP 321: Design of programming languages
Homework 5: Type-checker for a fragment of C

---

1. Written problems

There are no written problems for this assignment.

2. Coding problems (20 points)

For this project, you will write a type-checker for a fragment of C. This is a slightly smaller subset of C than that for which you wrote the lexer and parser. The primary difference is that the only allowed definitions are function prototypes and function definitions—i.e., there are no global variables. Following is an informal description of the valid programs; notice how wordy and imprecise it is, compared to a formal description!

Programs.

- A *program* is a sequence of definitions.
- A *definition* is either a function prototype or a function definition.
- A *function prototype* declares the name of the function, its return type, and the types of its parameters. It has no body. The two possible forms are

  τ `f(`τ_0 x_0`, ... ,`τ_{n-1} x_{n-1}`) ;`

  and

  τ `f(`τ_0`, ... ,`τ_{n-1}`) ;`

  Both declare a function `f` with return type τ and parameter types $\tau_0, \ldots, \tau_{n-1}$. The parameter names in the first form are ignored.
- A *function definition* declares the name of the function, its return type, the types of its parameters, and the body of the function. It has the form

  τ `f(`τ_0 x_0`, ... ,`τ_{n-1} x_{n-1}`)` $\{ss\}$

  which declares a function `f` with return type τ and parameters $x_0, \ldots, x_{n-1}$, where x_i is of type τ_i. The body of `f` is ss, which is a sequence of statements.

Validity.

1. No function may have more than one prototype; no function may have more than one definition.
2. There may be both a prototype and a definition for the same function, provided the definition follows the prototype and they both specify the same return type and types of parameters.
3. The function `g` may be used in the body of the function definition

   τ `f(`τ_0 x_0`, ... ,`τ_{n-1} x_{n-1}`)` $\{ss\}$

   only if there is a prototype for `g` before the definition for `f`; there is a definition for `g` before the definition for `f`; or `g` is `f` itself.
4. The body of the function definition

   τ `f(`τ_0 x_0`, ... ,`τ_{n-1} x_{n-1}`)` $\{ss\}$

is valid only if all the statements *ss* are valid in an environment that contains the type of `f` and the types of its parameters, and every `return` statement in *ss* returns a value of type τ. In particular, if $\tau =$ `void`, then there must be no `return` statement in *ss* (because there are no values of type `void`). *Note: this description does permit a function definition of non-*`void` *return type that has no* `return` *statements in its body. I will not test your code against such functions.*

Statement sequences.

Validity. Validity of a sequence of statements is as we have discussed in class.

Statements. Statements can only occur in the body of some function. Statement validity is only verified as part of verifying the validity of a statement sequence, which in turn can occur only as part of a block statement or function definition. We informally refer to the "current block" or "current function" to mean the block or function whose validity is being checked when the validity of the statement is being checked.

- A *declaration statement* consists of a type followed by one or more variables separated by commas, terminated with a semicolon; for example:

  ```
  int x ;
  bool x, y, z ;
  ```

  A declaration statement is valid if none of the variables have been previously assigned a type in the current block or function.
- A *initialization statement* consists of a type followed by one or more equalities separated by commas, terminated by a semi-colon. An equality consists of a variable followed by `=` followed by an expression. For example:

  ```
  int x = 5 ;
  bool x = true, y = true || false ;
  ```

  An initialization statement is valid if none of the variables have been previously assigned a type in the current block or function and the type of the expression assigned to the variable is the initialization type.
- An *expression statement* consists of an expression followed by a semicolon:

  e ;

  Such a statement is valid if *e* can be assigned some (any) type.
- A *return statement* consists of the keyword `return` followed by an expression and then a semicolon:

  `return` *e* ;

  A return statement is valid if the type of *e* is the return type of the current function.
- A *do-while statement* has the form

  `do` *s* `while (e) ;`

  where *e* is an expression and *s* is a statement. It is valid if *e* is of type `bool` and *s* is valid.
- A *while* statement has the form

  `while (e) s`

  where *e* is an exprssion and *s* is a statement. It is valid if *e* is of type `bool` and *s* is valid.
- A *for* statement has the form

`for` (τ x = e_0; e_1; e_2) s

where τ is a type, e_0, e_1, and e_2 are expressions, and s is a statement. It is valid if x has not been declared by an initialization or declaration statement in the current block or function; e_0 has type τ; e_1 and e_2 can be assigned some (any) type; and s is valid. Note that `x` may occur in e_1, e_2, and s.

- An *if* statement has the form

  `if` (e) s

  where e is an expression and s is a statement. It is valid if e has type `bool` and s is valid.
- An *if-else* statement has the form

  `if` (e) s_0 `else` s_1

  where e is an expression and s_0 and s_1 are statements. It is valid if e has type `bool` and s_0 and s_1 are valid.
- A *block* statement has the form

  { ss }

  where ss is a statement sequence. It is valid if ss is valid. Variable declarations in ss are local to the block. ss may refer to variables declared in an enclosing block.

Expressions. Just like statements, expressions only occur in the context of some function definition, and we use the the informal phrase "current function" just as for statements.

- Every literal is an expression; the type of a τ-literal is τ.
- A variable is an expression. Its type is the type that has been declared for it in the current function. It is an error to use a variable that has not been previously declared (as a parameter for the current function or with a declaration or initialization statement that has occurred previously in the current or an enclosing block or the current function).
- A function call expression has the form $f(e_0, \ldots, e_{n-1})$, where f is a function identifier and $e_0, \ldots, e_{n-1}$ are expressions. It has type τ if and only if f takes exactly n parameters of types $\tau_0, \ldots, \tau_{n-1}$, e_i is of type τ_i for each i, and f has return type τ.
- A post-increment expression has the form x`++` where x is a variable. It has type `int` if x has type `int` and does not type-check otherwise. Pre-increment expressions (`++`x), post-decrement expressions ($x--$) and pre-decrement expressions ($--x$) are similar.
- A logical negation statement has the form `!`e where e is an expression. It has type `bool` if e has type `bool` and does not type-check otherwise.
- Arithmetic operation expressions (using `+`, $-$, $*$, /) type-check if the operands type-check with the same numeric type (i.e., `int` or `double`), and do not type-check otherwise. %, `>>`, and `<<` expressions are similar, but must have `int`-type operands. An arithmetic expression that type-checks has the same type as its operands.
- Order-test expressions (`<`, `<=`, `>`, and `>=`) type-check if the operands type-check with the same numeric type (i.e., `int` or `double`), and do not type-check otherwise. An order-test expression that type-checks has type `bool`.
- (In)equality expressions (`==`, `!=`) type-check if the operands type-check with the same type, and do not type-check otherwise. An (in)equality expression that type-checks has type `bool`.
- Conjunction and disjunction expressions (&& and ||) type-check if the operands both type-check with type `bool` and do not type-check otherwise. Conjunction and disjunction expressions that type-check have type `bool`.

- Assignment expressions have the form x `=` e where x is a variable and e is an expression. An assignment expression type-checks if x has been previously declared in the current function to have type τ and e has type τ. The type of the assignment expression is τ.
- Conditional expressions have the form e ? e_0 : e_1 where e, e_0, and e_1 are all expressions. A conditional expression type-checks with type τ if e type-checks with type bool and e_0 and e_1 both type-check with type τ.

The type-checker. You are to implement two structures: `AnnAst` and `Typing`. The former defines types corresponding to annotated abstract syntax trees. In the latter you will define at least two functions:

- `checkPgm` : `Ast.program -> AnnAst.program`: `checkPgm`(p) returns the annotated AST corresponding to p.
- `inferExpNoEnv` : `Ast.exp -> AnnAst.exp`: `inferExp` e infers a type of e under the empty environment and returns the annotated AST of e corresponding to the inferred type. You will really need to implement a type-inference function that takes an environment as a parameter, and delegate to that.

You should probably also define a type-checking function variant of your type-inference function, along with many other auxiliary functions. Many of the cases of your type-inference function will be nearly identical; factor out the common parts into their own functions. You must also define the following exceptions, which will be raised by `checkPgm` and `inferExpNoEnv` (and the functions upon which they depend):

- `UndeclaredError`: raised when a variable is used, but has not been declared (by either a declaration or initialization statement).
- `MultiplyDeclaredError`: raised when a variable has multiple declarations (including initializations) in the same block, or when a function has multiple prototypes or definitions, or when a function definition does not agree with its prototype, or when a function prototype follows its definition.
- `ReturnTypeError`: raised when the type of the expression in a return statement does not match the return type of the current function.
- `TypeError`: raised when there is any other kind of error inferring or checking the type of an expression or validity of a program.

You will need to implement some form of environment type, which should be some form of map or dictionary. I recommend making use of the `SplayMapFn` functor of the SML/NJ library. The base environment under which any program will be checked must contain the following functions:

- `readInt` : void `->` int and `printInt` : int `->` void.
- `readDouble` : void `->` double and `printDouble` : double `->` void.
- `readBool` : void `->` bool and `printBool` : bool `->` void.

You are not to provide definitions of these functions, but any program you type-check must be able to use them. In effect, your type-checker must act as though every program starts with prototypes for these six functions. No program that I test your code on will in fact have such prototypes.

Strategy. Spend a little while understanding the `Ast` structure and how it relates to the grammar, because the `AnnAst` structure that you must write is based on it. Your `AnnAst` structure should be almost a copy of the `Ast` structure. Although it is tempting to try to re-use types from `Ast`, I recommend against it. The places where there must be differences can be nested rather deeply in the type definitions, and it is more effort than it is worth to try to separate them out.

I suggest getting the type engine working for the language without prototypes and blocks first with our first notion of an environment as a map (dictionary) from identifiers to types. Then add blocks, then add prototypes. If you have properly abstracted your operations on environments (such as adding entries and lookups) into functions, then it should not be too difficult to modify your code to handle the change to the environment type from a single map to a stack (list) of maps. Handling prototypes will require yet more changes to some of the structure of your code. There are many possible ways you might make changes; in my solutions, I was able to isolate the changes to primarily the function responsible for type-checking a sequence of definitions. Needless to say, you are best off first developing appropriate type-checking rules on paper, testing them on some sample programs to see if they result in accepting correct programs and rejecting incorrect programs, then implementing those rules.

The text discusses a "two-pass" type-checker. That is because the language it describes does not have function prototypes, but does still permit mutually recursive function definitions. In that setting, the type-checker must make one pass to collect all the types of the functions, then make a second pass to type-check the bodies of all the functions. With function prototypes, a single pass is the correct approach; a function cannot be used unless it has been previously defined or declared (via a prototype).

3. Code distribution and submission

I have provided the lexer and grammar specifications as well as an module `Ast` that defines the abstract syntax in the files `cpp.lex`, `cpp.grm`, and `ast.sml`. You must use this front end (lexer/parser); that means understanding the types in the module `Ast`. As mentioned above, you must implement a module called `AnnAst`, in which you define the types for annotated abstract syntax along with functions for string representations of annotated abstract syntax (to be used by the driver), and `Typing`, in which you implement the typing engine.

I have provided both a driver program and a test suite for your code. The driver program (structure `Driver` in `driver.sml`) is an extension of the driver program in the lexer/parser project. The default behavior is now to type-check the argument and print the (string representation of) the annotated abstract syntax tree that is produced. The `make` target is `driver`. Documentation is in the header comment in `driver.sml`.

The test suite attempts to type-check a number of files, which are located in the `good` and `bad` subdirectories of `testfiles/programs` (for type-checking complete programs using `Typing.checkPgm`) and `testfiles/expr` (for type-checking expressions using `Typing.inferExpNoEnv`). The test suite just verifies that the files in `good` successfully type-check and the files in the directories in `bad` do not (i.e., raise the correct exception). Each directory in `bad` corresponds to the type of exception that should be raised in the files in that directory (e.g., the files in `bad/mult` should raise `MultiplyDeclaredError` exceptions). I will only check that exceptions are raised by type-checking programs. The `make` target is `tests` and the CM build file is `tests.cm`.

As a point of comparison, my implementation of `Typing` requires about 400 lines of code (including comments and blank lines).

You will submit `ann_ast.sml` (your annotated abstract syntax implementation) and `typing.sml` (your type engine implementation).