Com S 541 — Programming Languages 1 May 17, 1996

Homework 6: Types

Due:
This is an individual homework.

1. (15 points) Using the type checking rules for SML given in class, write out a formal proof of the following type judgement, where the type environment H maps `3` and `1` to `int`, and maps `p` to `int -> int -> int`. (Hint, it may be helpful to use an edtior to make it easy to build the proof by copying.)

   $H \vdash$ `((`**let val** `one: int = 1` **in** `(`**fn** `x:int => ((p x) one))` **end**`) 3)`

2. (15 points) Use the type inference rules for SML given in class to infer the new type environment that results from the following declaration, starting from the empty type environment.

   ```
   val S = (fn f => (fn g => (fn x => ((f x) (g x)))))
   ```

   (a) You are to give resulting type environment, and a formal proof using the type inference rules. (b) Also answer the following question: is the type given to S in the resulting type environment polymorphic?

3. (30 points; extra credit only, semantics) In SML, there are no coercions allowed. What effect if any would allowing an int to be treated as a real have on the type inference rules for SML. That is, what would happen if `5 * 4.30` would type check and have type `real`? If you think this is fine, write any changes to the the type inference rules for SML that are needed. If you don't think this works, show why it doesn't, preferably from the type inference rules.

4. (40 points) Write type checking rules for the following abstract syntax, which is intended to be a fragment of C++. Comments (to you, the reader) in the grammar start with `%`.

   ```
   <Program> ::=
       <Empty>
     | <FunctionDefinition> <Program>
   <Empty> ::=
   <FunctionDecl> ::=
       <Type> <Identifier> ( <ParameterList> ) { <Body> }
   <ParameterList> ::=
       <Empty>
     | <FormalList>
   <FormalList> ::=
       <Formal>
     | <FormalList> , <Formal>
   <Formal> ::=
       <Type> <Identifier>
     | <Type> <Identifier> [ ]
   <Statement> ::=
       <Declaration>
     | { <Body> }
     | <Expr> ;
     | if ( <Expr> ) { <Body> }
     | if ( <Expr> ) { <Body> } else { <Body> }
     | while ( <Expr> ) { <Body> }
   ```

```
  | return ;
  | return <Expr> ;
<Body> ::=
    <Empty>
  | <Statement> <Body>
<Expr> ::=
    <Identifier>
  | <CharLiteral>
  | <StringLiteral>                   % think of it as an array of char
  | <IntLiteral>
  | <Expr> [ <Expr> ]                 % array subscripting
  | <Expr> + <Expr>
  | <Identifier> = <Expr>             % assignment statement
  | <Expr> == <Expr>                  % equality test
  | <Identifier> ( <Actuals> )
<Actuals> ::=
    <Empty>
  | <ExprList>
<ExprList> ::=
     <Expr>
  | <ExprList> , <Expr>
<Declaration> ::=
    <Type> <Identifier> ;
  | <Type> <Identifier> [ <Expr> ] ;  % array declaration
<Type> ::=
    int
  | char
  | void
```

Hints: you probably don't need to give type checking rules for `<Type>` or `<Empty>`. You don't have to deal with arithmetic coercions (such as the fact that in many contexts, a `char` can be coerced to an `int`). You might want to start with `<Expr>`. Statements won't have a type per se, but you have to make sure that all the subparts type check and that they are sensible. (For example, when is `return 3;` a legal statement?) Assignments are expressions in C++, so one can write: `i = (j = 3);`

5. (15 points; extra credit only) C++ also permits forward declarations of functions, which allows mutual recursion. Consider the following additional production for declarations.

```
<Declaration> ::=
    extern <Type> <Identifier> ( <ParameterList> ) ;
```

Can you extend your rules to deal with such declarations?