CENG 242
Homework # 3

(Due: April 2nd, 2006 Sunday 23:59)

Assume you have a hypothetical version of C which will allow:

- Nested function declarations, a function can be declared in the local scope of the other with arbitrary levels of nesting
- Declarations can be done at any place; they do not have to precede commands.
- Identifiers can have forward reference. They can be bound to declarations succeeding them in the code.
- Local scoping works as it is in standard C. A declaration cannot bind an applied occurrence in a higher (more global) scope or an occurrence in another local branch. Local declarations hide the others.
- Functions have no parameter for simplicity.

For example, the following is a valid Hyp.C code:

```
int f() {
     x=x+y;
     int x=0;
     x=x+g();
     int g() {
          int h() {
               t=t+x;
               return t+1;
          }
          return t+h()+y;
          int t=0;
     }
     return x;
}

int y;

int main() {
     x=f();
     x=x*x;
     return 0;
}

int x;
```

A compiler keeps track of identifiers in a data structure called *symbol table*. Symbol table maps identifier names to identifier data. In simplest form (ignore type checking etc.), for a variable “x” symbol table contains the address of the variable. All references to “x” are converted to address during compilation.

In this homework, you will implement an abstract data type `SymTable` in Haskell, which will simulate the manipulation of symbol table for our Hyp.C language:

- A value of `SymTable` gives the current state of the symbol table during compilation.
- You can use any internal representation as long as provide the proper interface.
- Commands are not actual commands but messages (function calls) to manipulate symbol table.
- Function names and the variable names are only bindables.
- Declaring a variable just mapping a symbol to an address, whereas declaring a function maps function name to an address and creates a local scope.
- A new table starts with current line number 0. Each message increments this number.
- Symbol table also keep track of the applied occurrences, at which line which identifier is referred and which identifier it is bound to.
- For simplicity, no identifier will be declared twice in the same scope. So you do not have to check for redefinitions.

You have the following messages:

- `newTable::SymTable`
  constructor. Empty symbol table.
- `declVar::SymTable -> String -> SymTable`
  `declVar table symbol` adds symbol to current scope in table and returns the new table. The address of the symbol is the current line number. Corresponds to:
  int symbol;
- `declFunc::SymTable -> String -> SymTable`
  `declFunc table symbol` adds symbol to current scope in table, creates a new local scope for the new function and returns the new table. Corresponds to:
  int symbol() {
- `exitFunc::SymTable -> SymTable`
  `exitFunc table` terminates the last local scope in table and returns the new table. Corresponds to:
  }
- `refIdentifier::SymTable -> String -> SymTable`
  `refIdentifier table symbol` adds a binding (an applied occurrence) to the symbol table and returns the new table.
- `link::SymTable -> SymTable -> Int -> SymTable`
  `link table1 table2 offset` merges two tables into one symbol table and returns this table. Unresolved symbols of one table can be bound to declarations of the other and the resulting table is returned with all binding and applied occurrences are combined. Assume there is no redefinition of symbols in the same scope. Offset is added to all line numbers of second table, so line numbers are not confused.
- `show::SymTable->String`
  you will implement your abstract type as an instance of Show so that your symbol table will be pretty printed. The format is:

```
Declarations:
name linenumber
fname linenumber
fname:name linenumber
fname:fname linenumber
fname:fname:name linenumber

Bindings:
name linenumber binding linenumber
name linenumber UNRESOLVED
```

For the above Hyp.C code, the corresponding calls will be:

```
t1 = newTable                       -- // creates a table
t2 = declFunc t1 "f"                -- int f() {
t3 = refIdentifier t2 "x"           -- x =
t4 = refIdentifier t3 "x"           -- x +
t5 = refIdentifier t4 "y"           -- y ;
t6 = declVar t5 "x"                 -- int x
t7 = refIdentifier t6 "x"           -- = 0;
t8 = refIdentifier t7 "x"           -- x =
t9 = refIdentifier t8 "x"           -- x +
t10 = refIdentifier t9 "g"          -- g();
t11 = declFunc t10 "g"              -- int g() {
t12 = declFunc t11 "h"              -- int h() {
t13 = refIdentifier t12 "t"         -- t =
t14 = refIdentifier t13 "t"         -- t +
t15 = refIdentifier t14 "x"         -- x;
t16 = refIdentifier t15 "t"         -- return t+1;
t17 = exitFunc t16                  -- }
t18 = refIdentifier t17 "t"         -- return t +
t19 = refIdentifier t18 "h"         -- h() +
t20 = refIdentifier t19 "y"         -- y;
t21 = declVar t20 "t"               -- int t
t22 = refIdentifier t21 "t"         -- = 0;
t23 = exitFunc t22                  -- }
t24 = refIdentifier t23 "x"         -- return x;
t25 = exitFunc t24                  -- }
t26 = declVar t25 "y"               -- int y;
t27 = declFunc t26 "main"           -- int main() {
t28 = refIdentifier t27 "x"         -- x =
t29 = refIdentifier t28 "f"         -- f();
t30 = refIdentifier t29 "x"         -- x =
t31 = refIdentifier t30 "x"         -- x *
t32 = refIdentifier t31 "x"         -- x;
t33 = exitFunc t32                  -- }
t34 = declVar t33 "x"               -- int x;
tablePrint = show t34               -- // to show the table t34
```

After the last call, the value of tablePrint should be:

```
"Declarations:
f 1
f:x 5
f:g 10
f:g:h 11
f:g:t 20
y 25
main 26
x 33
Bindings:
x 2 5
x 3 5
y 4 25
x 6 5
x 7 5
x 8 5
g 9 10
t 12 20
t 13 20
x 14 5
t 15 20
t 17 20
h 18 11
y 19 25
t 21 20
x 23 5
x 27 33
f 28 1
x 29 33
x 30 33
x 31 33"
```