Updated on March 20th 2009, Friday 01:30

CENG 242

Homework #2
(Due: March 30th 2009, Monday 11:55)

In this homework you are going to implement *dynamic binding* by writing a program to trace bindings of global, local variables.

```
data Decl = Var String | Func { name::String, locals::Decls, body::[Command] }


type Decls = [Decl]

data Command = UseVar String | CallFunc String
         deriving Show
```

`Decls` type is a list of `Decl` denoting a group of variable and function declarations. For this homework, types of variables are not significant so we only mention names of the variables.

`Func` is a triple denoting a function declaration. First element is the name of the function. Second is the local declarations in the function and the last one is a list of `Command`'s denoting the executable commands in the function. '`{ ... }`' notation is a special syntax where the functions `name, locals, body` that maps `Func` values to their fields are automatically generated by the interpreter.

You can observe this below:
Main> name(Func "a" [Var "b"] [UseVar "b"])
"a"
Main> locals(Func "a" [Var "b"] [UseVar "b"])
[Var "b"] (displayed as "[var b;]" by show function)
Main> body(Func "a" [Var "b"] [UseVar "b"])
[UseVar "b"]

You can use two notations equivalently:
`Func "main" [Var "x",Var "y"] [UseVar "x", CallFunc "f"]` , or
`Func{name="main",locals=[Var "x", Var "y"],body=[UseVar "x", CallFunc "f"]}`

Note that the custom *show* function defined in Hw2_Data module displays declarations in this form:

Main> [Var "b"]
[var b;]

Main> [Var "x",Var "y",Func{name="h",locals=[Var "a"],body=[UseVar "x"]}]
[var x;,var y;,function h() {
  var a;

Updated on March 20th 2009, Friday 01:30

```
  x;
}]
```

A `Command` denotes an executable command accessing variables. Values of this type mean a reference to variable and calling another function respectively (`UseVar, CallFunc`).

The definitions above are included in Hw2_Data module (from "Hw2_Data.hs" file) and you're going to import this module in **hw2.hs** file that you are going to submit, with this line:

```
import Hw2_Data
```

Given these definitions, you can define a program as a group of declarations:

```
[Var "x",Var "y",Func{name="h",locals=[Var "a",Func{name="g",locals=[Var
"x",Func{name="f",locals=[Var "x"],body=[UseVar "x"]}],body=[CallFunc "f",UseVar
"x"]}],body=[CallFunc "g"]},Func{name="main",locals=[Var "x",Var "y"],body=[UseVar
"x",CallFunc "h"]}]
```

The program above corresponds to the following C program:

```
int x,y;
void h() {
    int a;

    void g(){
      int x;

      void f(){
            int x;
            x++;
      }

      f();
      x++;
    }

    g();
}
int main() {
    int x,y;
    x++;
    h();
}
```

You will write an Haskell function `trace` with the following signature:

```
trace:: Decls -> [String]
```

Updated on March 20th 2009, Friday 01:30

This function will take a list of declarations and trace this program for binding of variables. If `main` function is given in the function list, it is the first function to call. Depending on the `CallFunc` commands executed, the other functions are called.

`trace` function will return a list of strings. After each action in the command trace, corresponding string from the strings below will be appended to this list.

When you encounter an error, you will just append the error string (defined below), stop appending any other strings to the list and return the list.

`Use Variable` *`scope`*`:`*`var`*
: This output is generated when a variable is used by a `Usevar` *`var`* command.

`Call` *`scope`*`:`*`func`*
: This output is generated when a function is called by a `CallFunc` *`func`* command.

`Return` *`scope`*`:`*`func`*
: This output is generated when a function is ended.

`Invalid reference` *`var`*
: This output is generated when an invalid reference to a non-existing (not alive) variable. This might be due to a `UseVar` *`var`* command.

`Error: main is not found`
: This output is generated when there is no `main` function is declared in the function declaration list given to `trace` function. Trace simply outputs this and ends.

`Function` *`func`* `is not found`
: This output is generated when a function required by `CallFunc` *`func`* call is not found.

In the list above, *scope* is a string is defined as "global" or name of the function where the used variable or called/returned function is bound. Use "global" as the scope name for global variables.

Consider this example program:

```
int x,y;
void f(){
      int c;
      c++;
}

void h() {
    int a;
    void g(){
      int x;

      void f(){
            int x;
            x++;
```

```
        }

        f();
        x++;
    }
    f();
    g();
}
int main() {
    int x,y;
    x++;
    h();
}
```

Observe that g() calls local f() with dynamic binding. The output for this program will be like this:

[“Call global:main”,
“Use Variable global:main:x”,
“Call global:h”,
“Call global:f”,
“Use Variable global:f:c”,
“Return global:f”,
“Call global:h:g”,
“Call global:h:g:f”,
“Use Variable global:h:g:f:x”,
“Return global:h:g:f”,
“Use Variable global:h:g:x”
“Return global:h:g”,
“Return global:h”,
“Return global:main”]

Another example program:

```
int x;

void f(){
        int k;
        k * x;
}

void v(){
        int x;
        x++;
        f();
}
```

Updated on March 20th 2009, Friday 01:30

```
int main(){
        x++;
        f();
        v();
}
```

Observe that variable “x” binds to global:x in f() function call from main() and binds to global:v:x in f() function call from v(), with dynamic binding. The output for this program will be like this:

[“Call global:main”,
“Use Variable global:x”,
“Call global:f”,
“Use Variable global:f:k”,
“Use Variable global:x”,
“Return global:f”,
“Call global:v”,
“Use Variable global:v:x”,
“Call global:f”,
“Use Variable global:f:k”,
“Use Variable global:v:x”,
“Return global:f”,
“Return global:v”,
“Return global:main”
]

An example with error strings:
Main> trace [Var "x",Func{name="empty",locals=[],body=[UseVar "a",UseVar "y",UseVar "x"]},Func{name="main",locals=[Var "x",Var "y"],body=[CallFunc "empty"]}]

[”Call global:main”,”Call global:empty”,”Invalid reference a”]

Observe that we stopped appending any string to the output after encountering *invalid reference* error.

Assumptions:

- Declaration order is NOT important. Forward references are possible.
- Name of a function and a variable will NOT conflict in test data.
- No recursive calls allowed. Test data will be recursion free.
- There is no repeated declarations in the same declaration list (in global and local). No function declaration will include local declaration like: [Var "x",Var "a",Var "x"]. Variable names can be re-used in different blocks (i.e. global "x" can be hidden by a local "x" of "main" and it can be hidden by local "x" of "f". It makes three copies of same variable name however in different blocks)

Updated on March 20th 2009, Friday 01:30

Specifications:

- All the work should be done **individually**.
- Your codes should be written in Haskell and have the name "**hw2.hs**"
- In evaluation, black box method will be used. So be careful about the name of functions, data structures etc.
- You will submit your code through **Cow** system.
- You should test your codes in **inek** machines with **hugs** before submitting.