

# Project 4: Activation Records

CSC 4351, Spring 2008

Due: 4 April 2008

Augment the `Semant` package to allocate locations for local variables, and to keep track of the nesting level as described on pp. 150–151, including the options for finding escapes and handling functions with more than  $k$  formal parameters. For a detailed description of MIPS Frames, see the SPIM Manual on the course web page. The necessary changes are the following:

## 1. Escape Analysis

Implement the methods `traverseVar`, `traverseExp` and `traverseDec` in `FindEscape.FindEscape`. The idea is to maintain a depth counter for the depth of each function declaration. For escape analysis, we will again construct a symbol table as for Project 3 and tear it down when we're done with escape analysis. When traversing a function, traverse the formals and the function body in a new `depth + 1` escape scope. When you traverse a variable or parameter declaration put a `VarEscape` or `FormalEscape` entry for it into the escape environment. When you traverse a variable reference (`SimpleVar`) get its escape entry from the environment, check if the entry depth is less than the current depth. If so, then set the escape. While we're at it we also find out which functions are leaf functions and which ones are not.

For remembering the result of escape analysis, there is a boolean field `escape` in `Absyn.FieldList` and `Absyn.VarDec`. There's also a boolean field `leaf` in `Absyn.FunctionDec`.

The files you'll need to modify are these three classes from `Absyn`. You'll need to initialize the escape fields to false instead of true and the leaf field to true instead of false.

In the `FindEscape` package, you only need to modify class `FindEscape`. The best strategy would be to have one method for each type of parse tree node as we did in `Semant.Semant`. Most of these methods don't do anything interesting other than calling the traversal method for the subtrees and maybe entering and exiting scopes. The interesting work is in `traverseVar(int, SimpleVar)`, `traverseExp(int, CallExp)` (for finding leaf functions), `traverseDec(int, VarDec)`, and `traverseDec(int, FunctionDec)`. This is quite a bit of code to write, but it's mostly simple code.

## 2. Finish the Machine Dependent Frame Data Structure

For this you need to implement a method `allocLocal` in `Mips.MipsFrame` and write code for allocating formal parameters.

Dealing with formals is a little tricky. First, the way in which actual parameters are passed has nothing to do with the way formal parameters are to be allocated. In its prologue a callee will copy its actual parameters into its formal parameter locations. If a formal parameter escapes then it must be allocated in the frame, otherwise it can be allocated in a temporary. On the MIPS, space for the formal is always allocated in the outgoing argument area of the caller (even if the actual parameter is passed in a register), so an `InFrame` formal will have an offset in this area. An `InReg` formal will refer to a new temporary.

You only need to change class `Mips.MipsFrame`. The amount of code to add is only about 25 lines, but you need to understand the frame layout and what to allocate in registers and what on the stack.

### 3. Allocate Variables and Construct the Frames

Most of this work is in `Semant.Semant`.

In `transProg` you must first find all variables that escape, then allocate a new level for translation of the program body, before the call to `transExp`. Then, wherever a variable entry is constructed you must first allocate the variable before constructing an entry for it with the result. Similarly, when constructing an entry for a function you must first construct a new level for it with information as to which parameters to the function escape, before constructing the function entry with the new level. Naturally, the function body must be translated in this new function level. `Semant.VarEntry` and `Semant.LoopVarEntry` will both need an additional `Access` parameter added to their respective constructors.

You need minor changes to `Semant.VarEntry` and `Semant.LoopVarEntry`. Most of the work is in `Semant.Semant`. You would simply add or modify a few lines in strategic locations. The number of lines to add/modify is again only in the order of 25 lines, but spread out through the code.

### Putting it Together

You can do these three tasks in any order. Task 3 depends on 1 and 2, but you can implement 3 assuming that all variables escape (the escape fields in the tree nodes are already set appropriately) and add 1 later. Similarly, you can produce some dummy result for 2, implement 3, and then finish task 2.

### Environment and Support Files

For working on this project, change the environment variable `PROG` in your `.bashrc` file to `chap6`. As usual, you can find support code in `${TIGER}/${PROG}`.

The main files for you to work with are the files in packages `FindEscape`, `Mips`, and `Semant`. In particular, you'll need to modify `FindEscape.FindEscape`, `Mips.MipsFrame`, and `Semant.Semant`.

### Compilation

Since we don't use any non-Java source files anymore, it's easiest to use `'javac -g *.java'` manually for compiling.

For running your type checker on an input file `test.tig`, execute

```
java Semant.Main test.tig
```

in your working directory. The result is what we got for project 3 with frame information added. You can still run the parser without semantic analysis using

```
java Parse.Main test.tig
```

### Submission

To make sure, you don't forget to submit anything, submit all the Java packages in which you modify files. Delete your class files first and then submit the entire directory structure:

```
cd prog4; rm */*.class; ~cs435100/bin/r_copy 4
```

In the `README` file, provide any information that will help the grader to give you partial credit. Explain what's implemented and what's not. Explain important design decisions in your code, e.g., if you add new classes.