EECS 337

Compiler Design

2009 Fall Semester

Homework 5

Revision 1, November 19, 2009

Due at the beginning of class, Thursday, November 19, 2009

Objectives: The objective of this assignment is to (1) implement and use a hierarchical environment, (2) generate code for control flow statements, and (3) calculate the address of all variables.

Refactoring: Make all modifications and improvements suggested in homework 3. Hand in the revised code and any supporting material to demonstrate your code refactoring (for example, test cases and their output)

Hierarchical environment: Implement a hierarchical environment `env_t` based on linked lists (or hash tables), as per Figure 2.37. Create a function to print the contents of a hierarchical symbol table. In addition to everything else, create your 4 test cases, and hand them in. Then, modify:

1. The grammar and associated actions upon entering and exiting each block (Figure 2.38).
2. Function `newtemp` and `widen` to use `env_t` instead of `list_head_t`
3. The translation scheme (Chapter 6) to use the "top" environment in a hierarchical symbol table.

In addition to source code and the other material specified on the Web page, hand in the revised code output on the program at the bottom of page 986, clearly showing the new dump of the hierarchical symbol table. For extra credit, implement `env_t` as a stack (page 91).

Truelist and Falselist: Implement the following functions, as defined in pages 410-411.

- `int quadruple_cmp(const void *, const void *)` to establish equality among two quadruple pointers
- `list_head_t *list_makelist(quadruple_t *instr_ptr)` creates a new list of quadruple pointers containing only `instr_ptr`.
- `list_head_t *list_merge(list_head_t *p1, list_head_t *p2)` concatenates the lists pointed to by `p1` and `p2` and return the concatenated list.
- `int backpatch(list_head_t *p, quadruple_t *i)` inserts `i` as the target for each of the instructions on the list pointed to by `p`. The backpatch function return 0 if successful, 1 in case of error.

Create test cases for each of these functions. (In particular, test the erroneous cases corresponding to NULL arguments.)
Define

```
typedef struct {
        list_head_t *truelist;
        list_head_t *falselist;
} boolean_list_t;
```

Create a function to print a `boolean_list_t`.

Code generation for control flow statements: Generate 3 address code for the control flow statements in the grammar of Appendix A.1 via backpatching. You can modify the grammar as needed to use backpatching. Hand in a print out the code generated on the program at the bottom of page 986.

Blocks: In the source language of Appendix A.1, each block can have *decls* to declare variables that are local to that block. For example,

```
{ float[5][7] i; int j; int[3] k; { int i; float[3][3] j; } }
```

is a valid program in this language. Discuss:

- An adaptation of activation records to allocate memory for the variables in each block
- A method to calculate the location of each variable at compile-time, given a hierarchical environment. In the case of vectors and arrays, your method needs only calculate the base of the array. Locations should be supplied for all variables, regardless of whether the variable appears in the original code or is a compiler temporary.
- A method to compute the length of each activation record given the answer to the previous questions.
- Test cases for the above.

Hand in a design document covering these points.

Address generation: Modify the values in the hierarchical symbol table so that it stores the location of each variable. Then, implement your method and test cases to calculate the location of each variable and store the result in the symbol tables. Also, implement your method to calculate the length of each activation record. Hand in your source code, the location of all the variables in the program at the bottom of page 986, and the location of all variables in the following program:

```
{ float[5][7] i; int j; int[3] k; { int i; float[3][3] j; } }
```