EECS 337

Compiler Design

2009 Fall Semester

Homework 4

Due on Tuesday, November 3, 2009

Objectives: The objectives of this assignment are to (1) generate intermediate code for expressions and array references, and (2) to execute type checking and type conversion.

Refactor: 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)

Symbol Table: Modify the id_type structure that you wrote in homework 3 so that it can be arranged in a doubly-linked list:

```
struct id_type {
        void *type; /* The type of this variable, as per Hw2 */
        unsigned int dimension; /* Vector dimension */
        unsigned int size; /* Size along the last dimension */
        id_type_t *subtype; /* Type of the previous dimensions */
        id_type_t *supertype; /* Type of the parent dimension */
};
```

Modify the parser so that it correctly sets the `supertype`. Modify the function that prints out the hash table to demonstrate that both `subtype` and `supertype` are set correctly. 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 newly written dump of the hash table.

Size calculation: Write a function

```
unsigned int sizeofidtype(id_type_t *t);
```

which returns the amount of space required by `id_type_t`. For example:

Type	**`*t`**	**Return value**
Scalar	{&int_var, 0, 0, NULL, NULL}	sizeof(int)
Scalar	{&float_var, 0, 0, NULL, NULL}	sizeof(double)
Array	{&float_var, 1, 10, NULL, NULL}	10*sizeof(double)
Array	{&float_var, 2, 50, &{&float_var, 1, 10, NULL, &supertype}, NULL }	50*10*sizeof(double)

In addition to the other material specified in the Web page, hand in a printout of the 4 test cases for function `sizeofidtype` specified in the table above.

Quadruples: Define an intermediate-code address as:

```
typedef struct {
        enum {symbol, int_const, float_const, bool_const, code} type;
        union {
                list_entry_t *entry_ptr;
                int *int_const_ptr;
                double *double_const_ptr;
                int *bool_const_ptr;
                quadruple_t *instr_ptr;
        } addr;
} intmdt_addr_t;
```

Define a quadruple as:

```
typedef struct quadruple quadruple_t;
struct quadruple {
        char *op;
        intmdt_addr_t *arg1;
        intmdt_addr_t *arg2;
        intmdt_addr_t *result;
};
```

Define intermediate code as an array of quadruples

```
#define MAXCODELEN 8192

typedef struct {
        quadruple_t code[MAXCODELEN];
        unsigned int n; /* Number of instructions in code */
} intmdt_code_t;
```

Write a function to print out an intermediate code address (a symbol is printed as its lexeme, a constant as its value, and an instruction as its index in `intmdt_code_t.code`.) Write a function to initialize `intmdt_code_t`, another function to print out `intmdt_code_t`, and a `gen` function

```
int gen(intmdt_code_t *intermediate_code,
        char *op, intmdt_addr_t *arg1, intmdt_addr_t *arg2,
        intmdt_addr_t *result);
```

which increments `n` and appends the instruction defined by `op, arg1, arg2, result` to the end of the quadruple array. The gen function should return 0 in case of error, 1 otherwise. Create your test cases (at least 4) for function `gen`, and hand them in with their output, in addition to the other material requested on the Web site.

Temporary Variables: Implement a function to create a new temporary variable and insert it into the symbol table:

```
intmdt_addr_t *newtemp(env_t *top);
```

Modify the symbol table so as to make it possible to distinguish between source code variables and compiler-generated temporary variables. In addition to everything else, create your 4 test cases, and hand them in.

Array References: Design and implement the translation of array references (*loc*) in the grammar in Appendix A.1. You can use the `sizeofidtype` function.

Type checking and conversion: Design and implement a function:

```
list_entry_t *widen(intmdt_addr_t *a, id_type_t *t);
```

that attempts to generate code to convert types. The `widen` function can use `gen` and `newtemp`. The function returns a pointer to the symbol table entry for the variable containing the converted value, or NULL in case of failure (e.g, convert a float to an integer, convert an array into a basic type). Assume that quadruple addresses and Boolean variables cannot be converted.

Expressions: Design and implement the translation of assignment statements (*loc=bool*). Your design should include type conversion, type checking, and type error handling. The compiler should generate errors if (among other cases) `gen` or `widen` fail. Hand in a print out the code generated on the program at the bottom of page 986.