CS 445 – Fall 2022

Homework 6

Due on or before 20 Nov at 11:59 PM Pacific Time.

1. The problem

In this assignment we will be preparing to generate code. Your compiler will need to compute the scope, size, and location of each symbol in an input program. You will add a new -M option to your compiler to print this information that your compiler has computed. The format that it will be printed in is shown later in this document.

1.1 Suggested Parser Changes

Your compiler should record the size of each variable encountered at parse time. The size of most variables is 1 "data word" except for arrays. The size of arrays is the number of items contained in the array plus 1 data word for holding the size. As a reminder, the data word containing the size of the array is held in a location before the first element of the array. This means that the size of an array x is effectively held in location x[-1]. Since arrays are passed by reference, the size of any array passed as a function argument is 1 data word.

1.2 Suggested Semantic Analysis Changes

Recall that the application binary interface (ABI) that we are using utilizes register R0 to hold the global pointer and register R1 to hold the current frame pointer at run-time. Your compiler should use two variables, `foffset` and `goffset`, to compute and update these values at compile-time in order to keep track of the location of symbols within the current input program. You should modify your TreeNode declaration to contain information about the size, location, and reference type (local, global, static, or parameter) of each symbol in the input program. The location of a symbol will be the offset from the appropriate pointer (global or frame) where memory is allocated for the symbol. This offset should be readily available in the variables `foffset` and `goffset`. You should use the symbol table to store this new information that is associated with each symbol. Once you do this, you will be able to use the symbols table to determine the location (offset), size, and reference type of all references to a variable when you encounter them.

You will need to reset the local frame offset `foffset` each time that a new function is entered so that the symbols for a particular function are offset from the top of the frame for that particular function. This will permit re-use of frame locations for different function calls. You should also be sure to leave room for the "return ticket" (the return frame pointer and the return address) that is located at the top of the frame for each called function. The result will be that each declaration that requires space will store the size of that space and the offset (location) of that space in the TreeNode associated with the declaration. The "reference type" field in your TreeNode will help in deciding where to allocate space and how to reference that space when generating code. The "reference type" field will hold one of the four values (local, global, static, or parameter) that will specify which offset register (R0 or R1) to use at code generation time.

Constants typically don't require that memory be allocated, as they are often just loaded from instruction memory (as discussed using the LDC instruction). In the case of constant arrays, however, you will need to allocate space for them in the global area. The current language only permits constant arrays of characters, so these will just be global character arrays, and their address is the address of the first character in the array. The size of these constant arrays is also stored in a data word immediately before the first item in the array.

Function symbols are global. The offset location of functions will be used later for storing the address of the first instruction in the function so that we can look up a function in the symbols table and jump to it when needed. Function offsets should be initialized to zero.

It is assumed that the `foffset` value will be saved when entering a new scope and restored when exiting a scope so that frame memory can be reclaimed as discussed in lectures. Be sure that your code does this. If you don't get this right, many of your locations will be incorrect. The size reported for scopes is actually the maximum extent of the `foffset` variable when allocating storage for that scope. For instance, in the example program below, the scope introduced by the compound statement associated with main (on line 23) is reported as having size -3. This is because the value of `foffset will` be -3 when allocating the storage necessary for this scope because the stack grows toward decreasing addresses and because the value of `foffset` starts at -2 in the function main due to its "return ticket". To continue the example, since the value of `foffset` is -3 in the scope associated with the compound statement on line 23, when the scope associated with the compound statement on line 25 is entered, space for 3 more variables is allocated, so the value of `foffset` now becomes -6 and so the size for the compound statement on line 25 is reported as -6.

2. An Example

Your code will be tested on the various inputs provided for this assignment. Each input file will be fed to your compiler with the command line "`c- -M filename`" where the -M option is specified to print an augmented containing tree information about the location and size of symbols. An example of this is shown below.

2.1 Example Input File

The following is an example C- input file. The lines in this file are numbered for reference. Comments contained in the file specify the location of symbols and are also for your reference.

```
 1 int g;                // loc G 0
 2 char h[10]:"dogs";    // loc G -2
 3
 4 beachDay(int p, q[])  // loc L -2, -3
 5 {
 6     int a:17;         // loc L -4
 7     bool b[10];       // loc L -6
 8     static int s;     // loc G -12
 9     static int t[10]; // loc G -14
10     static char u[10]:"corgi";  // loc G -25
11     int z;            // loc L -16
12     a;
13     b;
14     s;
```

```
 15     t;
 16     u;
 17     666;
 18     'x';
 19     "purple";
 20     true;
 21 }
 22
 23 main() {
 24    int a;           // loc L -2
 25    {
 26    int b, c, d;     // loc L -3, -4, -5
 27    }
 28    {
 29    int e, f;        // loc L -3, -4
 30    }
 31    for a = 1 to 10 do a;   // loc L -3
 32
 33 }
 34 // next free space in globals is -35
```

2.2 Example Output Tree

The following is an example of the augmented tree that will be printed if the input file shown above is handed to your compiler with the -M command line option. Output lines that are too long to fit within the dimensions of this document wrap around. Your output lines should not wrap around.

```
WARNING(13): Variable 'b' may be uninitialized when used here.
WARNING(4): The parameter 'p' seems not to be used.
WARNING(4): The parameter 'q' seems not to be used.
WARNING(11): The variable 'z' seems not to be used.
WARNING(26): The variable 'b' seems not to be used.
WARNING(26): The variable 'c' seems not to be used.
WARNING(26): The variable 'd' seems not to be used.
WARNING(29): The variable 'e' seems not to be used.
WARNING(29): The variable 'f' seems not to be used.
WARNING(24): The variable 'a' seems not to be used.
WARNING(4): The function 'beachDay' seems not to be used.
WARNING(1): The variable 'g' seems not to be used.
WARNING(2): The variable 'h' seems not to be used.
Var: g of type int [mem: Global loc: 0 size: 1] [line: 1]
Sibling: 1  Var: h of array of type char [mem: Global loc: -7 size: 11] [line: 2]
.   Child: 0  Const "dogs" of array of type char [mem: Global loc: -2 size: 5] [line: 2]
Sibling: 2  Func: beachDay returns type void [mem: Global loc: 0 size: -4] [line: 4]
.   Child: 0  Parm: p of type int [mem: Parameter loc: -2 size: 1] [line: 4]
.   Sibling: 1  Parm: q of array of type int [mem: Parameter loc: -3 size: 1] [line: 4]
.   Child: 1  Compound [mem: None loc: 0 size: -17] [line: 5]
.   .   Child: 0  Var: a of type int [mem: Local loc: -4 size: 1] [line: 6]
.   .   .   Child: 0  Const 17 of type int [line: 6]
.   .   Sibling: 1  Var: b of array of type bool [mem: Local loc: -6 size: 11] [line: 7]
.   .   Sibling: 2  Var: s of static type int [mem: LocalStatic loc: -17 size: 1] [line: 8]
.   .   Sibling: 3  Var: t of static array of type int [mem: LocalStatic loc: -19 size: 11]
[line: 9]
.   .   Sibling: 4  Var: u of static array of type char [mem: LocalStatic loc: -36 size: 11]
[line: 10]
.   .   .   Child: 0  Const "corgi" of array of type char [mem: Global loc: -30 size: 6]
[line: 10]
.   .   Sibling: 5  Var: z of type int [mem: Local loc: -16 size: 1] [line: 11]
.   .   Child: 1  Id: a of type int [mem: Local loc: -4 size: 1] [line: 12]
```

```
.   .   Sibling: 1  Id: b of array of type bool [mem: Local loc: -6 size: 11] [line: 13]
.   .   Sibling: 2  Id: s of static type int [mem: LocalStatic loc: -17 size: 1] [line: 14]
.   .   Sibling: 3  Id: t of static array of type int [mem: LocalStatic loc: -19 size: 11]
[line: 15]
.   .   Sibling: 4  Id: u of static array of type char [mem: LocalStatic loc: -36 size: 11]
[line: 16]
.   .   Sibling: 5  Const 666 of type int [line: 17]
.   .   Sibling: 6  Const 'x' of type char [line: 18]
.   .   Sibling: 7  Const "purple" of array of type char [mem: Global loc: -47 size: 7]
[line: 19]
.   .   Sibling: 8  Const true of type bool [line: 20]
Sibling: 3  Func: main returns type void [mem: Global loc: 0 size: -2] [line: 23]
.   Child: 1  Compound [mem: None loc: 0 size: -3] [line: 23]
.   .   Child: 0  Var: a of type int [mem: Local loc: -2 size: 1] [line: 24]
.   .   Child: 1  Compound [mem: None loc: 0 size: -6] [line: 25]
.   .   .   Child: 0  Var: b of type int [mem: Local loc: -3 size: 1] [line: 26]
.   .   .   Sibling: 1  Var: c of type int [mem: Local loc: -4 size: 1] [line: 26]
.   .   .   Sibling: 2  Var: d of type int [mem: Local loc: -5 size: 1] [line: 26]
.   .   Sibling: 1  Compound [mem: None loc: 0 size: -5] [line: 28]
.   .   .   Child: 0  Var: e of type int [mem: Local loc: -3 size: 1] [line: 29]
.   .   .   Sibling: 1  Var: f of type int [mem: Local loc: -4 size: 1] [line: 29]
.   .   Sibling: 2  For [mem: None loc: 0 size: -4] [line: 31]
.   .   .   Child: 0  Var: a of type int [mem: Local loc: -3 size: 1] [line: 31]
.   .   .   Child: 1  Range [line: 31]
.   .   .   .   Child: 0  Const 1 of type int [line: 31]
.   .   .   .   Child: 1  Const 10 of type int [line: 31]
.   .   .   Child: 2  Id: a of type int [mem: Local loc: -3 size: 1] [line: 31]
Offset for end of global space: -53
Number of warnings: 13
Number of errors: 0
```

Please study the input file and its associated output and make your compiler mimic this output as closely as possible.

3. Deliverables and Submission

Your homework will be submitted as an *uncompressed* .tar file that contains no subdirectories. Your .tar file must contain all files and code necessary to build and test your compiler. If you use ourgetopt, *be sure to include the ourgetopt files in your .tar archive***.**

The .tar file is submitted to the class submission page. You can submit as many times as you like. **The last file you submit before the deadline will be the only one graded.** No late submissions are accepted for this assignment. For all submissions you will receive email at your uidaho address showing how your file performed on the pre-grade tests. The grading program will use more extensive tests, so thoroughly test your program with inputs of your own. The error messages that your c- emits will be sorted before comparison with the expected output so that the order in which your messages are printed is not as important as it otherwise would be.

Your program must run with no runtime errors such as segmentation violations (SIGSEGVs).