Compilers [Spring 2020]

Programming Assignment VI

Objectives

1. To learn basic rules for generating RISC-style machine code equivalent to high-level abstract-syntax trees (ASTs).
2. To implement a code generator that converts Diminished Java (DJ) ASTs into Diminished Instruction Set Machine (DISM) programs.
3. To test a complete dj2dism compiler.

Due Date: Wednesday, April 29, 2020 (at 11:59pm).

Machine Details

Complete this assignment by yourself on the cselx##.csee.usf.edu computers. You are responsible for ensuring that your programs compile and execute properly on these machines.

Assignment Description

This assignment asks you to complete your dj2dism compiler by implementing the code-generation phase of compilation.

Begin by downloading and studying the header file for the code-generation module. This file, *codegen.h*, is posted at http://www.cse.usf.edu/~ligatti/compilers/20/as6. Then implement the `generateDISM` method, which is declared in *codegen.h*, in a new file called *codegen.c*.

If you are completing this assignment by extending your Assignment V implementation (as is recommended), you will also have to make a few minor modifications to your *dj.y*:

1. The *codegen.h* header file needs to be included.
2. The main method in *dj.y*, which is the main method of the entire compiler, should now enforce that the compiler is invoked with the command "dj2dism s.dj", where "s.dj" is the source-program filename and "s" can be any string.
3. After typechecking the input file "s.dj", the main method of *dj.y* should open the file "s.dism", into which the target program will be output. (If "s.dism" already exists, your compiler should overwrite the existing "s.dism" file.)
4. Having opened and obtained a file pointer for "s.dism", the main method of *dj.y* should invoke the code generator by calling the `generateDISM` method.
5. Before exiting, the main method of *dj.y* should close the "s.dism" target-code file.

Notes on Code Generation in dj2dism

1. Your generated DISM-code file must contain exactly one instruction per line. This will make it easy to find the precise instruction at which your program halts. (The sim-dism virtual machine reports the instruction number at which the DISM program halted; if sim-dism reports that the program halted at instruction *n* and the code file contains exactly one instruction per line, then we can find the halting instruction at code-file line number *n+1*.)

2. In order to satisfy the constraint that every line of the generated ".dism" file must contain exactly one DISM instruction, you may sometimes find it convenient to write instructions of the form "#LABEL: mov 0 0" or "mov 0 0 ; comment". In such instructions, the "mov 0 0" is used as an empty operation (i.e., a *no-op*) so that we can output a DISM label and/or a comment on an otherwise empty line of code.
3. Your code generator must implement the conjunction operator (`&&`) in the standard short-circuit style. File *good6.dj* illustrates this operator's dynamic semantics.
4. Your code generator must check and halt before dereferencing null-valued objects. Files *good20.dj*, *good21.dj*, and *good22.dj* illustrate the required behavior.

Hints on Organizing the Code Generator

You may find it helpful to organize your *codegen.c* such that it contains the following (in addition to the definition of the `generateDISM` method declared in *codegen.h*):

```
#define MAX_DISM_ADDR 65535

/* Global for the DISM output file */
FILE *fout;

/* Global to remember the next unique label number to use */
unsigned int labelNumber = 0;

/* Declare mutually recursive functions (defs and docs appear below) */
void codeGenExpr(ASTree *t, int classNumber, int methodNumber);
void codeGenExprs(ASTree *expList, int classNumber, int methodNumber);

/*  Using the global classesST, calculate the total number of
    (non-static) fields, including inherited fields, in an object of
    the given type. */
int getNumObjectFields(int type)

/* Generate code that increments the stack pointer */
void incSP()

/* Generate code that decrements the stack pointer */
void decSP()

/* Returns the address of the given static field in the given class. */
int getStaticFieldAddress(int staticClassNum, int staticMemberNum)

/* Output code to check for a null value at the top of the stack.
   If the top stack value (at M[SP+1]) is null (0), the DISM code
   output will halt. */
void checkNullDereference()

/* Generate DISM code for the given single expression, which appears
   in the given class and method (or main block).
   If classNumber < 0 then methodNumber may be anything and we assume
   we are generating code for the program's main block. */
void codeGenExpr(ASTree *t, int classNumber, int methodNumber)
```

```
/* Generate DISM code for an expression list, which appears in
     the given class and method (or main block).
   If classNumber < 0 then methodNumber may be anything and we assume
   we are generating code for the program's main block. */
void codeGenExprs(ASTree *expList, int classNumber, int methodNumber)

/* Generate DISM code as the prologue to the given method or main
   block.  If classNumber < 0 then methodNumber may be anything and we
   assume we are generating code for the program's main block. */
void genPrologue(int classNumber, int methodNumber)

/* Generate DISM code as the epilogue to the given method or main
   block.  If classNumber < 0 then methodNumber may be anything and we
   assume we are generating code for the program's main block. */
void genEpilogue(int classNumber, int methodNumber)

/* Generate DISM code for the given method or main block.
   If classNumber < 0 then methodNumber may be anything and we assume
   we are generating code for the program's main block. */
void genBody(int classNumber, int methodNumber)

/* Map a given (1) static class number, (2) a method number defined
   in that class, and (3) a dynamic object's type to:
   (a) the dynamic class number and (b) the dynamic method number that
   actually get called when an object of type (3) dynamically invokes
   method (2).
   This method assumes that dynamicType is a subtype of staticClass. */
void getDynamicMethodInfo(int staticClass, int staticMethod,
   int dynamicType, int *dynamicClassToCall, int *dynamicMethodToCall)

/* Emit code for the program's vtable, beginning at label #VTABLE.
   The vtable jumps (i.e., dispatches) to code based on
   (1) the dynamic calling object's address (at M[SP+4]),
   (2) the calling object's static type (at M[SP+3]), and
   (3) the static method number (at M[SP+2]). */
void genVTable()

/* Emit code for the program's instanceof-table (itable).
   The itable stores a zero on the stack (at M[SP+1]) if the
   object being tested is an instanceof the required class;
   otherwise the itable stores a nonzero value on the stack
   (at M[SP+1]). */
void genITable()
```

More Hints

My *codegen.c* file is 704 lines of code (106 of which are comments/whitespace). As with Assignment V, this is a relatively challenging assignment, requiring you to write several hundred lines of code. Please budget your time accordingly.

Many examples of valid DJ programs, on which you can test your compiler, are posted at http://www.cse.usf.edu/~ligatti/compilers/20/as1/dj/examples/good/. A complete solution to this assignment will correctly compile all of the valid DJ programs into equivalent, valid DISM code that can be executed in the sim-dism virtual machine. As usual, though,

we will grade your compiler's performance on DJ programs that have not been distributed to the class.

Grading
Your grade on this assignment is determined by the level to which you implement the desired functionality:

- *Level I*: `generateDISM` correctly generates DISM code for all DJ programs that (1) declare no classes and (2) declare no local variables. Expect to write about 250-300 lines of code for this level (or about 200-250 without counting comments and whitespace).
- *Level II*: `generateDISM` correctly generates DISM code for all DJ programs that declare no classes. For this level, expect to write about 70 lines of code beyond that of Level I.
- *Level III*: `generateDISM` correctly generates DISM code for all DJ programs.

Undergraduate students will earn: 75% credit for reaching Level I, 85% credit for reaching Level II, and 105% credit for reaching Level III.

Graduate students will earn: 65% credit for reaching Level I, 75% credit for reaching Level II, and 100% credit for reaching Level III.

All students will earn extra credit (+15% for undergraduates and +10% for graduate students) for having implemented a complete Level-III dj2dism compiler from scratch (i.e., without using any of the provided object-code files) during this semester.

Compilation of the Compiler
Use the following commands to compile the complete dj2dism compiler from scratch:

```
> flex dj.l
> bison -v dj.y
> sed -i '/extern YYSTYPE yylval/d' dj.tab.c
> gcc dj.tab.c ast.c symtbl.c typecheck.c codegen.c -o dj2dism
```

Alternatively, you may download working versions of the lexer, parser, AST, symbol-table, and type-checker modules as object-code files *dj.tab.o*, *ast.o*, *symtbl.o*, and *typecheck.o* from http://www.cse.usf.edu/~ligatti/compilers/20/as6/. With these object-code files, and the header files *ast.h*, *symtbl.h*, *typecheck.h*, and *codegen.h*, you can compile dj2dism with:

```
> gcc dj.tab.o ast.o symtbl.o typecheck.o codegen.c -o dj2dism
```

Example Executions
Your compiler should continue to report lexical, syntactic, and semantic errors in the source program before exiting. As with previous assignments, you need not report multiple errors. If there are no lexical, syntactic, or semantic errors, your compiler generates DISM code.

Your compiler should enforce that the given source-program filename ends in ".dj":

```
> ./dj2dism
Usage: dj2dism file.dj
> ./dj2dism good99.dj
ERROR: could not open file good99.dj
> ./dj2dism good1.dism
Error: Input filename does not match *.dj
```

The compiler produces no output (besides the target-code file) when compiling a valid DJ program:

```
> ./dj2dism good1.dj
>
```

After compiling *good1.dj*, we can find the dj2dism-generated code in file *good1.dism*:

```
            mov 7 65535  ; initialize FP
            mov 6 65535  ; initialize SP
            mov 5 1  ; initialize HP
            mov 0 0  ; ALLOCATE STACK SPACE FOR MAIN LOCALS
            mov 0 0  ; BEGIN METHOD/MAIN-BLOCK BODY
            mov 1 0
            str 6 0 1  ; M[SP] <- R[r1] (a nat literal)
            mov 1 1
            sub 6 6 1  ; SP--
            bgt 6 5 #0  ; branch if SP>HP
            mov 1 77 ; error code 77 => out of stack memory
            hlt 1  ; out of stack memory! (SP <= HP)
     #0: mov 0 0
            hlt 0  ; NORMAL TERMINATION AT END OF MAIN BLOCK
            mov 0 0; BEGIN INSTANCEOF TABLE
              [ itable omitted :) ]
            mov 0 0; END INSTANCEOF TABLE
```

(Of course, the DISM files your compiler generates must only be functionally equivalent—rather than identical to—the DISM files my compiler generates. Regarding the codes that DISM halts with, my compiler uses the codes shown above, plus error code 66 to indicate that insufficient heap memory exists for a new object, 88 to indicate a null-pointer dereference, and 99 to indicate that a vtable entry was not found).

Debugging Hint: When trying to understand and correct the behavior of the DISM code your compiler generates, you may find it helpful to insert *ptn* instructions into the generated DISM code. For example, to see the address to which the SP refers at some point, you could insert *lod 4 6 0* and *ptn 4* at that point. You may also find it helpful to execute some of your DISM programs in sim-dism with the debug-mode turned on.

As expected, we can execute file *good1.dism* in the sim-dism virtual machine:

```
> ./sim-dism good1.dism
Simulation completed with code 0 at PC=13.
```

Of all the provided *good*.dj* files, my dj2dism compiler generates the most lines of DISM code for *good9.dj*.

```
> ./dj2dism good9.dj
> ./sim-dism good9.dism
```

```
1
2
4
5
6
1
2
4
4
0
33
Simulation completed with code 0 at PC=2070.
```

We have waited all semester to compile and run our *binProd.dj* from Assignment I.

```
> ./dj2dism binProd.dj
> ./sim-dism binProd.dism
Enter a natural number: 1
Enter a natural number: 2
Enter a natural number: 1
Enter a natural number: 3
Enter a natural number: 1
Enter a natural number: 4
Enter a natural number: 0
Enter a natural number: 1
24
Enter a natural number: 0
Simulation completed with code 0 at PC=560.
```

Notice how much smaller (and more efficient) our hand-written *binProduct.dism*s were! My hand-written *binProd.dism* is 28 instructions, but my compiled *binProd.dism* is 571 instructions.

Submission Notes

- Type the following pledge as an initial comment in your *codegen.c* file: "I pledge my Honor that I have not cheated, and will not cheat, on this assignment." Type your name after the pledge. Not including this pledge will lower your grade 50%.
- Upload and submit your *codegen.c* file on Canvas, or if you have implemented the entire Level-III compiler by yourself, upload the full compiler as a zip or tar file.
- ***Include a comment with your submission indicating at which level we should grade your assignment: I, II, or III. Otherwise we'll grade your submission at Level III.***
- If you submit your assignment multiple times, we'll grade your latest submission.
- For every day that your assignment is late (up to 2), your grade reduces 10%.
- To make it easier for our teaching assistant to read and evaluate your code, use spaces rather than tabs in your code and avoid long lines of code (I try to limit lines to 80 characters in width).
- For full credit, your submissions must compile on the cselx machines without warnings and without errors.
- Your programs will be graded on both correctness and style, so include good comments, well-chosen variable names, etc. in your programs.
- *codegen.c*—and the DISM code it outputs—must be well commented and formatted.