1 Overview

This document provides hints for the code generation phase of the project. I have written this in a rather informal way. However, you should be able to translate most of it in your code generation phase directly. This document assumes that your compiler constructs a parse tree for Java^{--} programs. However, I believe that it will help you with your implementation even if the compiler does not construct the parse tree directly.

2 Code generation approach

The idea is to start from the top level data structure and invoke a procedure, say `GenCode`, on it.

```
GlobalDataStructure::GenCode(){
   GlobalDataStructure::iterator it;

   if (it = begin(); it != end(); it++)
        (*it).GenCode(); // invoke GenCode on data structures with  classes.
}
```

The primary role of `GenCode` for a nonterminal or parse tree node is to generate code for that particular node. `GenCode` for the node is therefore defined in terms of `GenCode` of its children nodes and code that combines the codes of children nodes (thereby implementing the semantics associated with the node). Look at the lecture handouts to see how `GenCode` for various constructs are defined. If you have organized parse tree node classes into inheritance hierarchies, and define `GenCode` to be a virtual function, `GenCode` will propagate down the tree recursively generating code for the nodes.

3 Code generation of constructs

The following are the main concerns during code generation:

- How to allocate space for variables?

  You have two kinds of variables: i) variables of primitive types, and ii) variables of reference types. Space for primitive types will be allocated on the stack. Space for variables of a reference type will be stored in the heap, whereas reference to the data will stay on the stack. Let us look at the following method declaration:

  ```
  void func() {
     ClassC x = new ClassC;          // create an object
     x.c = 4;                        // store a value in x.c
               :
  }
  ```

  Data for variable x will be stored in the following manner:

  i) An allocation for a reference is made on the AR of `func`. This will store a pointer to the location on heap where the data will actually be stored.

  ii) Invoke `sbrk` to allocate space on heap for x. Use the size of `ClassC` (say, ClassCSize) to determine how much space to allocate. (Check how this can be done by looking at pages 6 and 7 of the spim hanout.) Store the address returned by `sbrk` on stack at offset for the reference (say, xoffset). The following shows the nature of generated code:

```
<save registers>
li   $4, ClassCSize     # ClassCSize = size of (ClassC),
                        # register 4 = input to malloc
sbrk                    # call system to allocate space
<restore registers>
sw $2, $fp(xoffset)     # returned address is in $2
                        # store it on stack at fp+ offset of x.
```

- *How to access variables?*

  If the variable is of primitive type, then data will exist on the stack during runtime. So it can be accessed through its offset in the activation record. For reference types, there will be two steps in every access to data: one for finding the address of data on heap, and another for accessing the data itself. For instance, translation of assignment of term $x.c$ in the above example will involve generating the code to do the following:

  1. Read the value of x into register r1 to get address on data associated with x on heap.
  2. Add offset of c to content of r1 to get address of x.c into register r2.
  3. Store value 4 into the address specified in register r2.

  So generated code for the assignment will look something like the following:

```
lw $16, $fp(xoffset)    # load reference for x in register 16
lw $17, 0x04            # load value 4 in register 17
sw $17, $16(coffset)    # store value of $17 in address specified by 16+coffset.
```

- *How to store values for expressions?*

  The values of expressions will be stored in temporary variables.

- How to generate the control aspects of control flow statements? (Through labels and gotos.)
- How to manage registers? (Use a very simple model.)

The following sections describe the specific constructs in detail.

3.1 Class declaration

The code generation for a class will involve generating code for its methods and constructors[1]. The following issues will need to be taken care of for each class:

- What is its size? You can find this by calculating the size of the member fields, and summing them.
- What is the offset for each member field within this class?
- What is the nature of code generated for each method and constructor?

`CodeGen` for ClassType may look like the following:

```
ClassType::CodeGen(){
  Fields *fl= Fields(); // find the fields of this class
  Methods *mt = Methods(); // methods of this class

  // assume that iterator fit will go over all fields
  int offset = 0; // initial offset
  for (fit = fl->begin(); fit!= fl->end(); fit++) {
```

[1]Note that we do not have any constructors in the project.

```
            (*fit).Offset = offset; // assign offset
            class_size = class_size + (*fit).Size(); // size of field
            offset = offset+(*it).Size(); // set offset for next field
    }
    // assume that iterator mit will go over all methods of class
    for (mit = mt->begin(); mit!= ml->end(); mit++) {
            (*mit)->GenCode(); // generate code for method
    }
  }
```

We discuss the generation of code for methods in great detail below:

3.1.1 Method declaration

Method definitions can be handled by generating an assembly routine, say `_p_C_asm`, for each method p of class C. For each method of the form:

```
public resultType p(T1 a1, T2 a2, ..., Tm am)
```

generate an assembly subroutine of the following form:

```
        .global  _p_C_asm
        .ent     _p_C_asm
_p_C_asm:
         <code for _p_C_asm>
```

In addition, you will need to do the following for methods:

1. For a method declaration of the form `r1 p(f1, f2, ..., fn)`, add one more parameter to p: this parameter will pass the reference to object on which p will be invoked. Hence, the declaration is of the form : r1 p(C this, f1, f2, .., fn)

2. Assign an offset for each parameter `fi` with respect to `this`. The first parameter, this, will be nearest to the frame pointer. Hence, you can use the offset to access `fi` on the stack through the frame-pointer.

3. Lay out space for local variables:

   - Find a relative address or offset for each variable: Implement by traversing the symbol table or when constructing the symbol table and assigning each variable an offset. You will need the type attribute of each variable to determine how much space it will need.

     Note: Since your code will generate temporaries, allocation of space for temporaries will have to be done as you are going along, that is, every time a temporary is generated, it is added to the symbol table along with its offset.

   ```
   lvarsize := size of local + temp variables
   ```

4. Find the registers that you want to save.

   ```
   rsize := size of registers.
   ```

5. Determine stack frame size:

   ```
   stack frame size := lvarsize + rsize;
   ```

6. Use stack frame size to generate code that will allocate space on stack:

   Generate code of the form `sp := sp - stack frame size;`

7. Generate code for saving registers: for each register, find an offset on the stack, and generate code for storing it with respect to that. (check the lecture handout).
8. Generate code for the body.
9. Generate code for loading registers from the current activation record.

   Use offsets of registers
10. Generate code for jumping to routine that may call this routine.

    Implementation notes: Most of the above code can be generated by writing a routine that generates the prologue and epilogue for a method.

One important note: Since temporary variables are not known until the code for the body has been generated, the size of the activation record cannot be generated until then. We cannot therefore generate the prologue for a function until the body of the code has been generated. You may therefore want to do the following for generating code for a method:

1. Generate code for body. Store it in a buffer.
2. We now have complete symbol table for the method. Find size of AR and offset of variables. Generate prologue and epilogue of the method.
3. Output Prologue, code of body, and epilogue now.

3.2 Variable access

Assume that a variable `v` is accessed somewhere in a program. Our main concern is: how is `v` accessed? Note that `v` is somewhere in an activation record on the control stack. Here is what you need to do in order to generate code for finding `v`:

```
if (v is local variable and defined in local method) then
   if (v is a primitive type)
      find offset for v from local symbol table.
      v is located at frame-pointer($fp) + offset.
else if (v is an expression of form x.c)
      search for x in local symbol table
      find offset for x from local symbol table
      find offset for c from x's symbol table
      generate code which will do the following:
         i) load content of ($fp)+offset of x into register r1
        ii) content of r1+offset of c points to location of x.c
else if (v is a parameter and is the ith parameter) then
    locate v in the stack using the its offset and framepointer.
```

3.3 Method call

For a method activation `x.p(f1, f2, ..., fn)`, do the following:

1. generate code for evaluating each parameter `fi`
2. generate code for pushing value of `fi` on stack. Note: you should generate code for evaluating and pushing the parameters on stack in such a way that `fn` is pushed first, and `f1` last.
3. generate code for pushing value of `x` on stack.
4. generate code for jumping to the assembly routine generated for `p`.

3.4 Expressions

Use the mechanism that I presented in the class for generating code for expressions. Here is an example. let us assume that we are generating code for an expression of kind `E -> E1 + E2`. The various steps are:

1. Generate code for `E1`.
2. Generate code for `E2`.
3. Say `E1.symtab`, `E2.symtab` contain pointers to entries in the symbol table.
4. Create a temporary variable, say `T1`, for `E`.
   - type of `T1` = type of `E`.
   - define offset of `T1`.
   - `E.symtab` is a pointer to an entry to `T1`.
5. if `E.type` == integer then `a`, `b`, `c` are general registers, `load` is integer load, `add` is integer add
   else `a`, `b`, `c` are double/float registers, `load` is float/double load, `add` is float add
   Generate code in the following manner:
   - load value stored in location defined by `E1.symtab` in register `a`.
   - load value stored in location defined by `E2.symtab` in register `b`.
   - add `a` and `b` into register `c`.
   - store `c` into location defined by `E.symtab`.

   Note: This assumes a very simple register model: that is, they are temporary storage used only for computation.

You can extend this idea to other expressions, including boolean expressions.

3.5 Control statements

For control statements you will have to generate labels and appropriate jumps. Again, look at the lecture handouts to get a good sense of what is going on. I will give an example that will show how you can generate code for control statements. For rule `S -> if E then S1 end`, assume that `E.symtab` contains pointer to an entry in symbol table cthat will hold the value of `E` at runtime.

1. Generate label `L1`.
2. Generate code for `E`.
3. Generate "load value from location for `E.symtab` into register `a`"
4. Generate "`if (register a = 0) jump to L1`"
5. Generate code for `S`.
6. Generate "`L1`:"

You can apply the same idea to if-then-else, while, repeat-until etc. Note: in the above scheme, we are treating boolean expressions just like arithmetic expression. That is, an evaluation of a boolean expression return 1 if true and 0 if false. This is different from what we did in the class. You can choose either implementation.