Project: Code generation hints (2)

This document describes the nature of final code generation for your project. I have made a number of assumptions here, mostly in terms of availability of information regarding various aspects of the parse tree and the symbol tables. They are specified in terms of method calls to various data structures. You will need to implement them so that they are specific to your data structures. First, assume that the following are defined (and implemented by you):

- Symbol table: A symbol table contains a set of entries (one for each entity such as method, variable, class, etc.). Each entity will be represented by a pointer to its entry in the table. We can use the pointer to access various information about the entity (for instance, its identifier, offset, size, type etc.).
- E.code: code sequence for evaluating nonterminal E. This can be stored as part of the data structure associated with nonterminal E.
- E`.GenCode()`: a method that generates code for E.

  E`.GenCode(label1, label2)`: A method that generates code for expression E with labels `label1` and `label2`.

  Labels can be simple strings that you pass to GenCode.

  This document will show how GenCodes for various language constructs are specified.
- `GenLabel()`: a method used for generating labels. You can define a global method that will return a unique label every time it is called. Generated labels will be used as targets of assembly jump instructions.
- `GenTemp()`: A method for creating temporary variables. In addition to creating the temporary, it will put the temporary in current symbol table. It returns a pointer to temporary variable's entry in the symbol table.
- `id.SymTab`: Defines the pointer to id's entry into a symbol table. This pointer will be used to determine id's size, type, offset etc.
- `E.symtab`: Defines the pointer to the symbol table entry which will hold the value of E at runtime. This entry may be a constant, a variable or a temporary generated for E. For instance, if value of E will be be stored in a temporary variable, say T1, E.symtab will point to the symbol table entry for T1.
- `CurrentSymbolTable`: Pointer to the top symbol table on stack.

1 Expressions

The primary goal in generating code for expression is to generate code for evaluating simple expressions (expressions involving one or two operands), and store them into temporaries.

- E $\rightarrow$ **id**

```
  // find the symbol table entry for the identifier, and make
  // it symbtab of expression.
  E.symtab = CurrentSymbolTable->FindId(id);
// assumption: FindId returns a pointer to symbol table entry for id.
  E.type = (E.symtab)->type(); // get type of identifier.
```

- E → integer_constant:

```
E.symtab = NULL; // This E is  a constant.
E.type = INTEGER; // type of expression
E.val = integer_constant.val; // get the value of literal
// assumption: integer_constant.val contains the value
// associated with the token. Also, E.val stores the constant
// value directly.
```

  Similar code will be generated for other kinds of constants.

- E → E1 op E2:

```
E1.GenCode(); // generate code for E1
E2.GenCode(); // generate code for E2
E.code = E1.code + E2.code; // add generated code of E1 and E2.
// E1.symtab points to entry where value for E1 will be stored;
// similarly for E2.symtab
if ((E1.symtab != NULL) && (E1.symtab->PrimitiveType()) {
   // assumption: E1.symtab->SimpleType() returns true if
   // E1.symtab is of primitive type.
   // E1.symtab is simple local variable or temporary, which means
   // it will be on stack at a fixed offset
   // generate code for loading E1.symtab in a register, say 19
    sprintf(buffer, "lw $19, %d($fp)\n", (E1.symtab)->offset());
   // assumption: (E1.symtab)->offset() returns offset of entry.
} else if ((E1.symtab != NULL) && ((E1.symtab)->ReferenceType()) {
  // E1.symtab is a reference;
  // generate code for accessing variable from heap. Look at
  // lecture 14 handout how that can be done.
 } else { // E1 is a constant.
    // use type of E1 to determine what constant to load
    sprintf(buffer, "li $19, %d\n", E1.val);
}
E.code = E.code + buffer; // add buffer to E.code
// similar code for E2
// add code from E2 to E.code
// generate a temporary variable that will store value computed
// by E.
E.symtab = GenTemp();  // this puts temporary in symbol table
(E.symtab)->type = E.type;
// after putting in symbol table, calculate offset for temporary.
CurrentSymbolTable->EvalOffset(E.symtab);
// assumption: CurrentSymbolTable->EvalOffset evaluates offset of
// the passed entry pointer.
// generate code for evaluating operation:
switch (op) {
case ADDITION:
    sprintf(buffer, "add $19, $19, $20\n");
    break;
case ...
         :
}
E.code = E.code+buffer;
// now generate code for storing value back
sprintf(buffer, "sw $19, %d($fp)\n", E.symtab.offset);
E.code = E.code+buffer;
```

E.code will contain the assembly code for these set of instructions.

- For assignment expression E $\rightarrow$ id = E1, GenCode:

```
E1.GenCode();  // generate code for right hand side
// note E.symtab will contain pointer to entry that will store
// its value.
// code for loading E's value. Same as above.
// assume that register 19 contains E's value
if ((id.symtab)->SimpleType()) {
   // id is simple local variable or temporary, which means
   // it will be on stack at a fixed offset
   // generate code for storing value in register 19 at fixed offset
    sprintf(buffer, "sw $19, %d($fp)\n", (id.symtab)->offset());
} else {
  // variable is a reference;
  // generate code for storing information on heap.
}
E.code = E1.code + buffer;
```

- Other kinds of expressions can be implemented similarly.

2 Boolean expressions

Boolean expressions are evaluated by generating jmps.

- E $\rightarrow$ true

```
sprintf(buffer, "%s:\n", E.true);
E.code = buffer;
```

- E $\rightarrow$ false

```
sprintf(buffer, "%s:\n", E.false);
E.code = buffer;
```

- E $\rightarrow$ E1 or E2

```
E1.true = E.true;   E1.false = GenLabel();
E2.true = E.true;   E2.false = E.false;
E1.GenCode(E1.true, E1.false);
sprintf(buffer, "%s:\n", E1.false);
E2.GenCode(E2.true, E2.false);
E.code = E1.code + buffer + E2.code;
```

- E → E1 and E2

```
  E1.true = GenLabel();   E1.false = E.false;
  E2.true = E.true;       E2.false = E.false;
  E1.GenCode(E1.true, E1.false);
  sprintf(buffer, "%s:\n", E1.true);
  E2.GenCode( E2.true, E2.false);
  E.code = E1.code + buffer + E2.code;
```

- E → id1 relop id2

```
sprintf(buffer, "lw $19, %d($fp)\n", (id1.symtab)->offset());
sprintf(buffer1, "lw $20, %d($fp)\n", *id2.symtab)->offset());
sprintf(buffer2, "ble $19, $20 %s\n", E.true );
sprintf(buffer3, "jmp %s\n", E.false);
E.code = buffer+buffer1+buffer2+buffer3;
```

3 Statements

Statements can be evaluated by generating code for sub-statements, and proper jumps.

- S → if E then S1 else S2

```
E.true = GenLabel(); E.false = GenLabel();
S.next = GenLabel();
E.GenCode( E.true, E.false);
sprintf(buffer, "%s:\n", E1.true);
S1.GenCode();
sprintf(buffer2, "jmp %s:\n", S.next);
sprintf(buffer3, "%s:\n", E1.false);
S2.GenCode();
sprintf(buffer4, "%s:\n", S.next);
s.code = E.code + buffer + S1.code + buffer2 + buffer3 + S2.code + buffer4;
```

- S → while E do S1

```
S.begin = GenLabel();
E.true = GenLabel(); E.false = GenLabel();
sprintf(buffer, "%s:\n", S.begin);
E.GenCode(E.true, E.false);
sprintf(buffer1, "%s:\n", E1.true);
S1.GenCode();
sprintf(buffer2, "jmp %s:\n", S.begin);
sprintf(buffer3, "%s:\n", E1.false);
S.code = buffer + E.code + buffer1 + S1.code + buffer2 + buffer3;
```

- MethodDecl → ResultType MethodName (ArgList) Block

```
// generate code for block first. This will allow us to
// generate all temporaries.
Block.GenCode();
ArSize = Block.FindARsize(); // find activation record size
//assumption: FindArSize() returns size of AR associated with block
VarSize = Block.FindVarSize();
// assumption: FindVarSize() returns total space occupied by variables on AR
ArgList.EvalParameterOffsets(); // evaluate offsets for params
// generate prologue
sprintf(beg, ".text\n");
sprintf(beg1, ".ent %s\n", MethodName.AssemblyName());
sprintf(beg2, "%s:\n", MethodName.AssemblyName());
sprintf(beg3, "subu $sp, %d:\n", ArSize);
sprintf(beg4, "sw $31, %d($sp):\n", ArSize-VarSize);
sprintf(beg5, "sw $fp, %d($sp):\n", ArSize-VarSize-4);
sprintf(beg6,"addu $fp, $sp, %d\n", ArSize);
// now generate epilogue
sprintf(end, "lw $31, %d($sp)\n", ArSize-VarSize);
sprintf(end1, "lw $fp, %d($sp):\n",ArSize-VarSize-4);
sprintf(end2, "addu \$sp, %d\n", ArSize);
sprintf(end3, "j $31 \n");
sprintf(end4, ".end %s\n", MethodName.AssemblyName());
//combine epilogue, body code, and prologue
MetodDecl.code = sum of all beg buffers + Block.code + sum of all end buffers.
```

- Method invocations:

  `MethodInvocation` $\rightarrow$ `MethodName(ParameterList) ParameterList` $\rightarrow$ `Param ParameterList1 |` ϵ

  We look at GenCode for `ParameterList` first:

```
Param.GenCode(); // generate code for evaluating parameters
// generate code for rest of parameters
ParamList1.GenCode(); // generate code for rest of list
// generate code for pushing this parameter on stack
sprintf(buf, "subu $sp, %d\n", (Param.symtab)->size());
// assumption: (Param.symtab)->size() returns size of parameter
sprintf(buf1, "lw $19, %d($fp)\n", (Param.symtab)->offset());
sprintf(buf2, "sw $19, %d($sp)\n",(Param.symtab)->size());
// order of addition of buffers is important to push parameters
// in right order on stack.
ParamList.code = Param.Code + ParamList1.code + buf + buf1 + buf2;
```

  Now for method invocation expression is:

```
 ParameterList.GenCode();
// generate code for jumping to function
sprintf(buf, "jal %s\n", MethodName.AssemblyName());

// generate code for fixing stack
```

```
int paramsize = MethodName.GetParamSize();
// assumption: GetParamSize returns size of parameters of method

sprintf(buf1, "%s:\n", MethodName.AssemblyName());
sprintf(buf2, "addu $sp, %d\n", paramsize);
MethodInvocation.code = ParameterList.code + buf + buf1 + buf2;
```