Due: June 2, 2011 11:59 PM.

1 Overview

This is the final phase of the compiler project. It involves extending the semantic analysis phase of the project in order to generate MIPS R2000 assembly code that can then be assembled and executed either on the MIPS simulator (`spim/xspim`).

2 Code Generation

The details of the target architecture and the assembly language are described in the handout on SPIM. You should carefully read this handout in order to understand the machine architecture and the assembly instruction set supported by the machine.

The primary goal of this phase is to map Java^{--} language abstractions such as classes, instance variables, logical and arithmetic expressions, control flow statements, and method declarations and activations to the primitive abstractions supported by MIPS R2000.

2.1 Data

Java^{--} supports primitive types such as *boolean*, *integer*, *char*, and *string*. An object of a primitive type can be mapped directly to a memory location of appropriate size and type supported directly by the MIPS assembler. Literals such as `integer_constant`, and `string_constant` can be mapped through the `.data`, and `.asciiz` assembly directives respectively.

The reference type (class) is more complex. Its implementation involves resolution of the following two issues:

1. *Where should the data associated with an object be stored? On stack or heap?* Data for 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) Allocate space on heap for x. Use the size of `ClassC` to determine how much space to allocate.

   (iii) Store 4 at (reference value of x + offset of x).

2. *How can data be accessed?* 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:

   (a) Read the value of `x` into register, say `r1`, to get the address of data associated with `x` on heap.

   (b) Add the offset of `c` to the content of `r1` to get the address of `x.c` into register, say `r2`.

   (c) Store the value 4 into the address specified in register `r2`.

2.2 Expressions

Java^{--} supports both logical and arithmetic expressions. You will need to implement these expressions by mapping them to primitive logical and arithmetic instructions supported by the target machine. One of the most important considerations in generating code for expressions is the usage of temporary storage for holding intermediate results. This is especially valid in a RISC-based machine where only load and store instructions access memory directly. For instance, implementation of `X + Y * Z` may involve (i) loading `Y` and `Z` in registers, (ii) performing multiplication, (iii) storing the result in a temporary storage, say `T1`, (iv) Loading `X` and `T1` in registers, (v) performing addition, and (vi) storing value in another temporary variable.

Note that some of the loads and stores are redundant, and can be optimized easily. However, we will not be concerned with optimization issues at all. The temporary storage can either be a machine register or a temporary variable generated by the compiler. Note that you will need to treat temporary variables like any other variable. That is, define their types and scopes by adding them in corresponding symbol tables. Note also that the need for temporary storage requires that the compiler make more than one pass in order to generate code for a procedure: one pass for generating temporary variables, and another for generating code and constructing the activation record that allocates storage space for the temporary variables as well.

You may want to define a class `TempNameGenerator` for generating a unique names for temporary variables.

2.3 Control statements

A control statement such as `if-then-else`, `while-do` or `for` is implemented by generating code that checks its control condition, and jumps to appropriate parts of the generated code. Note that since the MIPS assembly language supports ascii identifiers for labels, you can use ascii labels for implementing the control statements.
As in the case of temporaries, you may want to define a class `LabelNameGenerator` for generating unique labels. You are free to implement case statements by converting them into equivalent if-then-else statements.

2.4 Assignment

For an assignment statement, you will need to generate code for the right hand side that evaluates the right hand side expressions and stores the value of the expression in the storage location indicated by the left hand side.

2.5 Method declaration and invocation

Method declarations and activations form the most complex component of this phase. For a method declaration of the form:

```
class C {
   void func(T1 a, T2 b) {
       <local variables>
       <statements>
   }
       :
}
```

the following will need to be done:

i) Add an implicit parameter for `func`. This parameter is the object on which the method is invoked. Hence, the declaration is almost equivalent to

```
void func (C this, T1 a, T2 b) { ... }
```

This will be needed when you are pushing arguments for the function on the stack during a method invocation.

ii) Construct activation record for the procedure. This is done by going through the symbol table, allocating an offset for each local and temporary variable and determining the size of the activation record.

iii) Map local (both variable and temporary) declarations into the activation record. You will be using the offset of each variable for this purpose.

iv) Generate instructions for changing the stack. (See lecture handouts.)

v) Generate assembly code for the body procedure.

For a method activation of the form x.f(a, b), the following will need to be done:

i) Transform x.f(a, b) into f(x, a, b). Note that x is an implicit argument for f. This means that you will be pushing x along with a and b on stack.

ii) Devise a call-return protocol which builds appropriate mechanisms for putting arguments on the activation stack. Java^{--}.

I suggest that you implement the method invocation first without the dynamic method dispatching. You can add that later when you have time.

You should familiarize yourself thoroughly with the MIPS R2000 calling convention (Section 9 of the MIPS handout).

3 Details

The final phase of your compiler will read a Java^{--} program and generate an equivalent MIPS R2000 assembly program. The execution of the generated program should give the same results as the Java^{--} program. We will test this program by assembling and running the program directly on the CSIF machine. I suggest that you implement the different components of the compiler in the following order:

- Implement one of the primitive types, say integer.
- Class containing only the primitive type, method declarations with no arguments and no returns, and constants
- Expressions containing only two operands with the specified primitive type.
- Sequence and if-then-else
- Classes containing references
- Constructors, method declarations and invocations with any number of arguments
- Complex expressions
- Add more primitive types
- Other constructs