221 Compilers

In this exercise, you should see how to translate a simple high-level language into assembly code for a stack machine. Most recent past papers have included a question on code generation (C244 2014Q2, C221 2016Q2, C221 2015Q2)- this exercise sets the foundations.

Exercise 2.1: Code Generation for Statements

Input: assume the following abstract syntax tree data type for statements:

```
data Statement = Assign Name Expression |
               Compound [Statement] |
               IfThen Expression Statement |
               IfThenElse Expression Statement Statement |
               While Expression Statement
```

Here Expression is a Haskell data type for the abstract syntax tree of expressions.

Output: your translator should return a list of assembly language instructions. The instructions are represented as the following Haskell data type:

```
data Instruction =
  Add | Sub | Mul | Div  (as before)
  | PushImm num    (push constant onto stack)
  | PushAbs Name   (push variable at given
                    location onto stack)
  | Pop Name       (remove value at top of stack
                    and store it at given loc'n)
  | CompEq         (subtract top two elements
                    of stack, and replace with
                    1 if the result was zero,
                    0 otherwise)
  | JTrue label    (remove top item from stack;
                    if 1 jump to label)
  | JFalse label   (jump if stack top is 0)
  | Jump label     (jump unconditionally
  | Define label   (set up destination for jump)
```

Note that `Define` is an assembler directive, not an executable instruction. What to do

- Write a function `translate_statement` which generates assembly code for a statement. You are given some of the cases below: your job is to supply the rules for `Compound`, `IfThenElse` and `While`.

  Assume that you have been given a function `translate_expression`, which takes as input the AST for an expression, and produces assembly code for a stack machine which when executed leaves the value of the expression on the top of the stack (a simple version of such a function is given in Chapter 2 of the lecture notes).

Hint 1: Assignments For example, here is the rule for translating assignment statements:

```
translate_statement :: Statement -> [Instruction]

translate_statement (Assign var exp)
 = translate_expression exp ++
   [Pop var]
```

The `Pop` instruction takes the value at the top of the stack and stores it at the named location. The "`++`" operator is used to join lists of instructions. For example,

```
translate_statement (Assign "y" (Binop Plus (Number 12)(Ident "x")))
```

would yield the following list of assembly language instructions for some simple stack-based machine:

```
[PushImm 12,
 PushAbs "x",
 Add,
 Pop "y"]
```

This could then be printed out in the proper syntax for some real processor. For example, for Intel's IA32 instruction set you would get something like this:

```
pushl $12
pushl x
popl  %eax
addl  %eax,(%esp)
popl y
```

It turns out that the `Add` pseudo-instruction can't be done with just one IA32 instruction, you need two.

Hint 2: If-then statements As another example, here is the rule for the if-then statement:

```
translate_statement (IfThen cond_exp body)
 = translate_expression cond_exp ++
   [JFalse label] ++
   translate_statement body ++
   [Define label]
   where
   label = a new label name which has not been used before in the program
```

The instruction `JFalse` removes the element at the top of the stack, and jumps to the given label if the value represents false — for example we could encode True as 1 and False as 0. There is a similar instruction `JTrue`.

For example the statement `if a=100 then a:=b` translates to the stack machine sequence

```
[PushVar "a",
 PushConst 100,
 CompEQ,
 JFalse "L1234",
 PushVar "b",
 Pop "a",
 Define "L1234"]
```

This could then be printed out in IA32 assembler as follows:

```
   mov.w a,-(sp)
   mov.w #100,-(sp)
   mov.w (sp)+,d0    ; These two IA32 instructions achieve the effect
   sub.w d0,(sp)     ; of the stack machine ‘‘CompEQ’’ instruction.
   tst.w (sp)+       ; These two IA32 instructions achieve the effect
   beq   L1234       ; of the stack machine ‘‘JTrue’’ instruction.
   mov.w b,-(sp)
   mov.w (sp)+,a
L1234:
```

The expression “`Define "L1234"`” does not correspond to an actual instruction to be executed. Instead it makes the destination of the `JFalse` instruction. Thus if the conditional expression evaluates to the representation of False, the body is not executed, and control transfers directly to the end of the sequence.

Common problems

- **Haskell's algebraic data types**

  Note that `Statement` and `Instruction` are examples of Haskell's algebraic data types. This is a compact representation, which would be implemented using a `union` in C. The symbols `Assign`, `Compound`, `IfThen`, `Add`, `Pop` etc. are called *constructors*. For example, a statement can have one of five forms; the constructor indicates which form is present and introduces the elements of the structure, such as (e.g. for `While`, the ASTs for the conditional expression and the loop body.

- **Haskell's lists**

  The translator returns a list of instructions. Haskell has three operators for building lists:

[x]	given an element x, make α into a list with one element	e.g. [1]
x : A	given a list A and an element x, make a new list starting with x and ending with A	e.g. 2:[1]=[2,1]
A++B	join the two lists A and B	e.g. [1,2]++[3,4]=[1,2,3,4]

If you have spare time, you might like to think about how to implement this code generator in Java, using a Visitor pattern.

Extension - boolean operators

Consider adding an AST type for Boolean expressions, such as comparisons. How would you design a "transBExp" function for Boolean expressions?

How about we also add And, Or and Not operators. The trick, of course, would be to avoid executing the second operand of an And expression if the first operand turns out to be false (since the result is already guaranteed to be false). An idea for doing this might be to give transBExp the labels of the statements to jump to if the result is true, and, respectively, false.

Paul Kelly October 2016

Exercise 2.1: Code generation for Statements - notes on possible solutions

Here is a complete solution:

```
translate_statement :: Statement -> [Instruction]

translate_statement (Assign var exp)
 = translate_expression exp ++
   [Pop var]

translate_statement (Compound statlist)
 = translate_statement_list statlist

translate_statement (IfThen exp body)
 = translate_expression exp ++
   [JFalse skiplabel] ++
   translate_statement body ++
   [Define skiplabel]

translate_statement (IfThenElse exp thenbody elsebody)
 = translate_expression exp ++
   [JFalse elselabel] ++
   translate_statement thenbody ++
   [Jump endlabel] ++
   [Define elselabel] ++
   translate_statement elsebody ++
   [Define endlabel]

translate_statement (While exp body)
 = [Define startlabel] ++
   translate_expression exp ++
   [JFalse endlabel] ++
   translate_statement body ++
   [Jump startlabel] ++
   [Define endlabel] ++

translate_statement_list :: [Statement] -> [Instruction]

translate_statement_list [] = []
```

```
translate_statement_list (fst:rest)
 = translate_statement fst ++
   translate_statement_list rest
```

Doing it in Java using a Visitor

Doing this in Java is very similar but slightly more laborious. Let's start with the AST for statements. Let's just do assignment and if-then for brevity:

```
public abstract class StatementTree {
    public abstract void Accept(StatementTreeVisitor v);
}
public class AssignNode extends StatementTree {
    String lhs; ExpressionTree rhs;
    AssignNode(String _lhs, ExpressionTree _rhs) {
        lhs = _lhs; rhs = _rhs;
    }
    public void Accept(StatementTreeVisitor v) {
        v.visitAssignNode(lhs, rhs);
    }
}
public class IfThenNode extends StatementTree {
    ExpressionTree cond; StatementTree body;
    IfThenNode(ExpressionTree _cond, StatementTree _body) {
        cond = _cond; body = _body;
    }
    public void Accept(StatementTreeVisitor v) {
        v.visitIfThenNode(cond, body);
    }
}
public abstract class StatementTreeVisitor {
    abstract void visitAssignNode(String lhs, ExpressionTree rhs);
    abstract void visitIfThenNode(ExpressionTree cond, StatementTree body);
}
```

We could get the Java translator to return a list of instructions, or perhaps assemble them in an array. For simplicity let's just print them out:

```
public class TranslateVisitor extends StatementTreeVisitor {
    void visitAssignNode(String lhs, ExpressionTree rhs) {
        // print instructions which, when executed, will leave
```

```
        // expression value at top of stack
        rhs.Accept(new TranslateExpVisitor());
        System.out.println("pop "+lhs);
    }
    void visitIfThenNode(ExpressionTree cond, StatementTree body) {
        // print instructions which, when executed, will leave
        // expression value at top of stack
        UniqueLabel skiplabel = new UniqueLabel();
        cond.Accept(new TranslateExpVisitor());
        System.out.println("JFalse "+skiplabel.toString());
        body.Accept(this);
        System.out.println("Define "+skiplabel.toString());
    }
}
```

You can find working Java and Haskell code at

`http://www.doc.ic.ac.uk/~phjk/Compilers/SampleCode.`