Práctica 6

Part 1: Compiling Arithmetic Expressions

In last Práctica you wrote the definition for compiling the assignment expression as something like:

\[
\text{encodestatement} \left( \text{assign} \left( \text{name}(X), \text{Expr} \right), D, \left( \text{E_code} ; \text{instr} \left( \text{store}, \text{Addr} \right) \right) \right) :\neg \\
\text{lookup} \left( X, D, \text{Addr} \right), \\
\text{encodeexpr} \left( \text{Expr}, D, \text{Exprcode} \right).
\]

This is, when compiling an assignment \( X := \text{Expr} \) you produce the code for the expression \( \text{E_code} \) which leaves the result in the accumulator, followed by \( \text{instr} \left( \text{store}, \text{Addr} \right) \) (where \( \text{Addr} \) is the memory location where the value for \( X \) is stored).

Here, we have to compile the arithmetic expressions.

Compiling Arithmetic Expressions

The task here is to define the predicate:

\[
\text{encodeexpr} \left( \text{Expr}, D, \text{E_code} \right).
\]

The clauses for \text{encodeexpr} must translate the different types of arithmetic expressions:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Const</td>
<td>LOADC Const</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>LOAD Addr</td>
<td>where Addr is the address of Name</td>
</tr>
<tr>
<td>Expr1 Op Expr2</td>
<td>Expr1code Instr</td>
<td>where Expr1code is the translation of Expr1 and Instr which applies Op</td>
</tr>
</tbody>
</table>

Here we will assume that \( \text{Expr2} \) is either a constant or a variable (not a composite expression). The translations for the different operators is (the first one is applied to a constant and the second applied to a name):

<table>
<thead>
<tr>
<th>Operator</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>ADDC, ADD</td>
</tr>
<tr>
<td>-</td>
<td>SUBC, SUB</td>
</tr>
<tr>
<td>*</td>
<td>MULC, MUL</td>
</tr>
<tr>
<td>/</td>
<td>DIVC, DIV</td>
</tr>
</tbody>
</table>

As an example: \( \text{Expr} + 7 \) translates to \( \text{Exprcode} \ ADDC 7 \).

Sample outputs of the program so far may look like this:

?- encodestatement(assign(name(x),const(a)),D,X).
D = dic(x, _G521, _G525, _G526) 
X = instr(loadc, a);instr(store, _G521)
Part 2: Compiling the Other Statements

The If Statement: let us start by considering the if statement which has the form:

    if  <test> then <then> else <else>

The code for this take the form:

    testcode;  
    thencode;  
    JUMP label2; 
label1:  
    elsecode;  
label2: 

where testcode causes a jump to label1 if the test proves false. We have used labels to indicate the instructions whose addresses are label1 and label2. The Prolog formulation of this is:

    encodestatement(if(Test,Then,Else),D,  
                   (Testcode; Thencode; instr(jump,L2); label(L1); Elsecode; label(L2)) ) :-  
        encodetest(Test,D,L1,Testcode),  
        encodestatement(Then,D,Thencode),  
        encodestatement(Else,D,Elsecode).  

Notice that the clause does not fix the addresses L1 and L2 but only constraints its values through labelinf the object code. One can think of the output from 'encodestatement' as being relocatable code. The output will contain free variables L1 and L2 whose values will not be fixed until the linking stage of compilation.

This is an example of the use of the logical variable to delay specifying certain parts of a data structure. In the code for encodetest(Test,D,L,Testcode), The test has the form test(Op,Arg1,Arg2) and L is the address to jump to if the test fails. Op is either =, < or >.

Thus, the definition of the predicate encodetest/4 is as follows:
The **While Statement**: Write the code for compiling the while statement.

The **Read and Write Statement**: write the code for the definition handling the read (of a variable) and the clause handling the write (of an expression) statement. Assume the instruction `instr(write, 0)` outputs the data in the accumulator into the screen. This is:

```
encodestatement(read(name(X)), D, instr(read, Addr) ) :- lookup(...).
encodestatement(write(Expr), D, (Ecode; instr(write, 0)) ) :- encodeexpr(...).
```

The **Sequence Statement**: write the code for handling the sequence statement `Stat1; Stat2` which which is treated as just another statement type. This is write code defining the clause

```
encodestatement( (S1;S2), D, (Code1;Code2) ) :- ...
```

**Target Language**

Remember the target language instructions are as follows:

- **Arithmetic literal**: `ADDC, SUBC, MULC, DIVC, LOADC`
- **Arithmetic memory**: `ADD, SUB, MUL, DIV, LOAD, STORE`
- **Control transfer**: `JUMPEQ, JUMPNE, JUMPLT, JUMPGT, JUMPGE, JUMP`
- **Input output**: `READ, WRITE, HALT`.

For an example (to compute factorials) of the compiler function see Practica 3.

Sample outputs of the program so far may look like this:

```
?- encodestatement(if(test(=, name(x), const(5)), assign(name(x), const(1)), assign(name(x), const(2))), D, X).
D = dic(x, _G921, _G925, _G926)
X = ((instr(load, _G921); instr(subc, 5)); instr(jumpne, _G897)); ((instr(load, _G921); instr(addc, 1)); instr(store, _G921)); instr(jump, _G892); label(_G897); (instr(load, 2); instr(store, _G921)); label2(_G892)

?- encodestatement(while(test(=, name(x), const(5)), assign(name(x), expr(+, name(x), const(1)))), D, X).
D = dic(x, _G878, _G882, _G883)
X = label(_G845); ((instr(load, _G878); instr(subc, 5)); instr(jumpne, _G859)); ((instr(load, _G878); instr(addc, 1)); instr(store, _G878)); instr(jump, _G845); label(_G859)

?- encodestatement((read(name(x)); write(expr(+, name(x), const(1)))), D, X).
D = dic(x, _G682, _G686, _G687)
X = instr(read, _G682); ((instr(load, _G682); instr(addc, 1)); instr(store, _G682)); instr(jumpne, _G897);
```

```