I217: Functional Programming

10. A Programming Language Processor – Compiler

Kazuhiro Ogata, Canh Minh Do

Roadmap

- Compiler

Compiler

The compiler translates programs written in Minila into lists of instructions that can be executed by the virtual machine.

We first describe how to generate lists of instructions for expressions and then how to generate lists of instructions for statements.

Compiler

Given an expression *e*, genForExp generates a list of instructions for *e* such that executing the instruction list makes the result of *e* left at the top of the stack.

```
op genForExp : Exp -> IList .
```

In what follows, the following variables are used:

```
vars E E1 E2 : Exp .
vars S S1 S2 : Stm .
var V : Var .
var N : Nat .
var IL : IList .
```

Compiler

the empty list of instructions

eq genForExp(n(N)) = push(N) | iln .

Executing push(N) leaves N at the top of the stack.

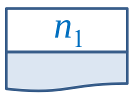
eq genForExp(V) = load(V) | iln .

Executing load(V) leaves the natural number associated with V in a given environment at the top of the stack. If the environment does not have any entry whose key is V, then errNat is pushed onto the stack, which will become errStack.

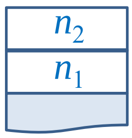
Compiler

eq genForExp(E1 + E2)
  = genForExp(E1) @ genForExp(E2) @ (add | iln) .

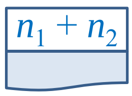
Executing the instruction list generated by genForExp(E1) leaves the result n_1 of calculating E1 at the top of the stack.



Executing the instruction list generated by genForExp(E2) leaves the result n_2 of calculating E2 at the top of the stack.



Executing add leaves the result of calculating $n_1 + n_2$ at the top of the stack.



Compiler

For the remaining expressions, equations can be described likewise for genForExp.

Compiler

Given a program in Minila, compile generates the list of instructions for the program (a statement) with generate and adds quit to the list at the end.

```
op compile : Stm -> IList .
eq compile(S) = generate(S) @ (quit | iln) .

op generate : Stm -> IList .

eq generate(estm) = iln .
```

If the program is estm, then it generates the empty list of instructions.

Compiler

If the program is V := E ;, then it generates the list of instructions for E and adds store(V) to the list at the end.

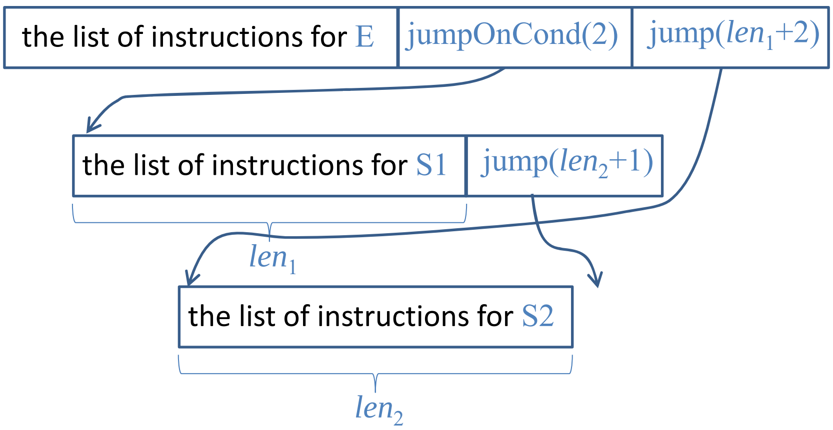
the list of instructions for E	store(V)

generate(x := n(3) + n(4) * n(5) ;) generates the following:

$$\underbrace{\text{(push(3) | (push(4) | (push(5) | (multiply | (add}}_{\text{n(3) + n(4) * n(5)}}\text{ | (store(x) | iln)))))):IList}$$

Compiler

If the program is if E {S1} else {S2}, then it generates the following list of instructions:



Compiler

You can use len to calculate the length of an instruction list *il*:

len(*il*)

If you want to know the length of the instruction list that is generated from a statement *s*, then all you have to do is as follows:

len(generate(*s*))

Compiler

generate(if tmp * tmp > x {z := tmp - n(1) ;} else {y := tmp ;})
generates the following:

(load(tmp) | (load(tmp) | (multiply | (load(x) | (greaterThan |

tmp * tmp > x

(jumpOnCond(2) | (jump(6) |

(load(tmp) | (push(1) | (minus | (store(z) | (jump(3) |

z := tmp - n(1) ;

(load(tmp) | (store(y) | iln)))))))))))))):IList

y := tmp ;

Compiler

generate(if ((z - y) % n(2)) === n(0) {tmp := y + (z - y) / n(2) ;}
else {tmp := y + ((z - y) / n(2)) + n(1) ;})

generates the following:

(load(z) | (load(y) | (minus | (push(2) | (mod | (push(0) | (equal |

((z - y) % n(2)) === n(0)

(jumpOnCond(2) | (jump(10) |

(load(y) | (load(z) | (load(y) | (minus | (push(2) | (divide | (add | (store(tmp) | (jump(11) |

tmp := y + (z - y) / n(2) ;

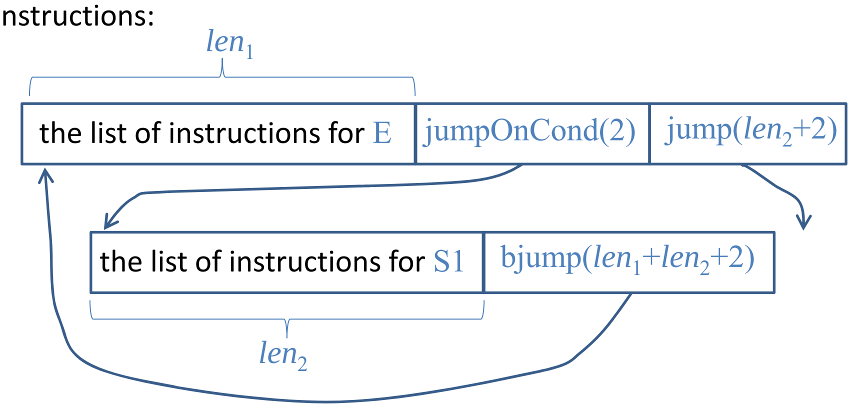
(load(y) | (load(z) | (load(y) | (minus | (push(2) | (divide | (add | (push(1) | (add | (store(tmp) |

tmp := y + ((z - y) / n(2)) + n(1) ;

iln)))))))))))))))))))))))))):IList

Compiler

If the program is while E {S1}, then it generates the following list of instructions:



Compiler

generate(while y < n(10) || y === n(10) {x := x * y ; y := y + n(1) ;})
generates the following:

(load(y) | (push(10) | (lessThan | (load(y) | (push(10) | (equal | (or |

y < n(10) || y === n(10)

(jumpOnCond(2) | (jump(10) |

(load(x) | (load(y) | (multiply | (store(x) |

x := x * y ;

(load(y) | (push(1) | (add | (store(y) | (bjump(17) | iln)))))))))))))))))):IList

y := y + n(1) ;

Compiler

for V E1 E2 {S1}

is equivalent to

```
V := E1 ;
while V < E2 || V === E2 {
  S1
  V := V + n(1) ;
}
```

in that the result (the environment) obtained by interpreting the former is exactly the same as the one obtained by interpreting the latter.

This can be used to describe the equation to generate the list of instructions for the for statement.

Compiler

Given two statements s_1 and s_2 that are equivalent,

the instruction list $\mathrm{generate}(s_2)$ generated from s_2 can be used as the one $\mathrm{generate}(s_1)$ generated from s_1.

Namely,
$\mathrm{eq\ generate}(s_1) = \mathrm{generate}(s_2)$.

Compiler

If the program is for V E1 E2 {S1}, then it generates the following list of instructions:

the list of instructions for V := E1 ;

the list of instructions for

while V < E2 || V === E2 { S1 V := V + n(1) ;}

Compiler

generate(for y n(1) n(10) {x := y * x ;})
generates the following:

(push(1) | (store(y) |
y := n(1) ;

while y < n(10) || y === n(10)
{ x := y * x ; y := y + n(1) ; }

(load(y) | (push(10) | (lessThan | (load(y) | (push(10) | (equal | (or |
y < n(10) || y === n(10)

(jumpOnCond(2) | (jump(10) | (load(y) | (load(x) | (multiply | (store(x) | (load(y) |
x := y * x ;

(push(1) | (add | (store(y) | (bjump(17) | iln)))))))))))))))))):IList
y := y + n(1) ;

Compiler

If the program is S1 S2, then it generates the following list of instructions:

the list of instructions for S1	the list of instructions for S2

You can use _@_ to combine two instruction lists il_1 and il_2:

$$il_1 \ @ \ il_2$$

Compiler

generate(x := y * x ; y := y + n(1) ;) generates the following:

(load(y) | (load(x) | (multiply | (store(x) |

x := y * x ;

(load(y) | (push(1) | (add | (store(y) | iln))))))):IList

y := y + n(1) ;

Compiler

generate(y := n(1) ; while y < n(10) || y === n(10) { x := y * x ; y := y + n(1) ; }) generates the following:

(push(1) | (store(y) |

y := n(1) ;

while y < n(10) || y === n(10)
{ x := y * x ; y := y + n(1) ; }

(load(y) | (push(10) | (lessThan | (load(y) | (push(10) | (equal | (or |

y < n(10) || y === n(10)

(jumpOnCond(2) | (jump(10) | (load(y) | (load(x) | (multiply | (store(x) | (load(y) |

x := y * x ;

(push(1) | (add | (store(y) | (bjump(17) | iln))))))))))))))))))):IList

y := y + n(1) ;

Exercises

1. Complete the compiler and do some tests for the compiler.
2. Extend the compiler so that arrays of natural numbers can be supported.
3. Extend the compiler so that functions (or procedures) can be supported.
4. Extend the compiler so that many other program constructs can be supported.

You may consult the Java compiler.

Appendices

The compiler translates the program

```
x := n(1) ;
for y n(1) n(10) {
  x := y * x ;
}
```

into the lists of instructions

```
(push(1) | (store(x) | (push(1) | (store(y) | (load(y) | (push(10) |
(lessThan | (load(y) | (push(10) | (equal | (or | (jumpOnCond(2) |
(jump(10) | (load(y) | (load(x) | (multiply | (store(x) | (load(y) | (push(1)
| (add | (store(y) | (bjump(17) | (quit | iln))))))))))))))))))))))):IList
```

Appendices

The compiler translates the program

```
x := n(24) ;  y := n(30) ;
while y =!= n(0) {
  z := x % y ;  x := y ;  y := z ;
}
```

into the lists of instructions

(push(24) | (store(x) | (push(30) | (store(y) | (load(y) | (push(0) | (notEqual | (jumpOnCond(2) | (jump(10) | (load(x) | (load(y) | (mod | (store(z) | (load(y) | (store(x) | (load(z) | (store(y) | (bjump(13) | (quit | iln)))))))))))))))))))):IList

Appendices

The compiler translates the program

```
x := n(20000000000000000) ;
y := n(0) ;
z := x ;
while y =!= z {
  if ((z - y) % n(2)) === n(0) {
    tmp := y + (z - y) / n(2) ;
  } else { tmp := y + ((z - y) / n(2)) + n(1) ;  }
  if tmp * tmp > x {  z := tmp - n(1) ; }
  else {  y := tmp ;  }
}
```

Appendices

into the lists of instructions

(push(20000000000000000) | (store(x) | (push(0) | (store(y) | (load(x) | (store(z) | (load(y) | (load(z) | (notEqual | (jumpOnCond(2) | (jump(44) | (load(z) | (load(y) | (minus | (push(2) | (mod | (push(0) | (equal | (jumpOnCond(2) | (jump(10) | (load(y) | (load(z) | (load(y) | (minus | (push(2) | (divide | (add | (store(tmp) | (jump(11) | (load(y) | (load(z) | (load(y) | (minus | (push(2) | (divide | (add | (push(1) | (add | (store(tmp) | (load(tmp) | (load(tmp) | (multiply | (load(x) | (greaterThan | (jumpOnCond(2) | (jump(6) | (load(tmp) | (push(1) | (minus | (store(z) | (jump(3) | (load(tmp) | (store(y) | (bjump(47) | (quit | iln))):IList