i116: Basic of Programming

14. Programming language processor: compiler

Kazuhiro Ogata

Roadmap

- Compiler for Minila

Compiler for Minila

- Minila has three more statements than the assignment calculator:
  - Empty statement
  - Conditional (**if**) statement
  - Loop (**while**) statement
- The compiler for Minila needs to generate command lists for the three statements.

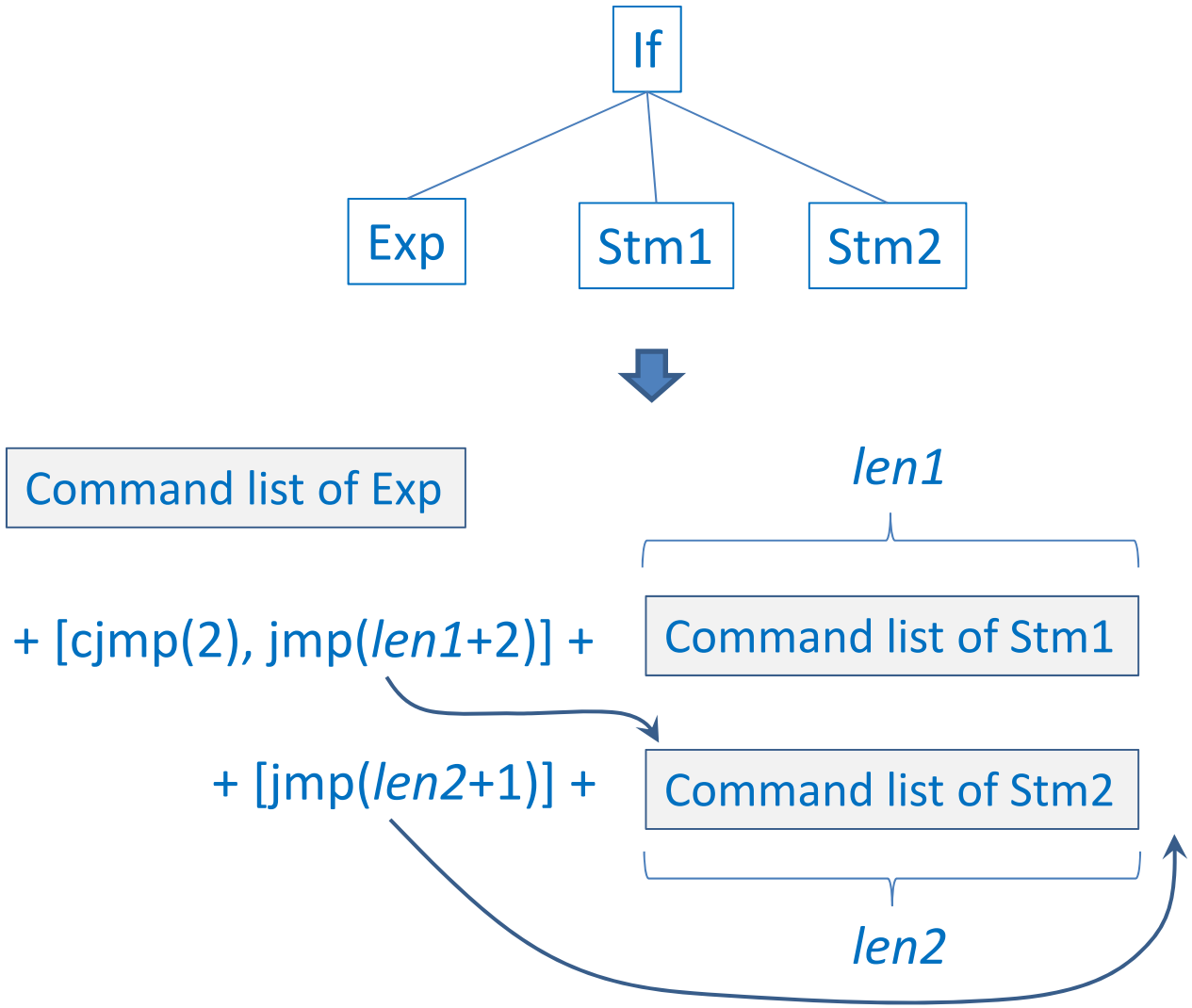
Compiler for Minila

EmptyStm

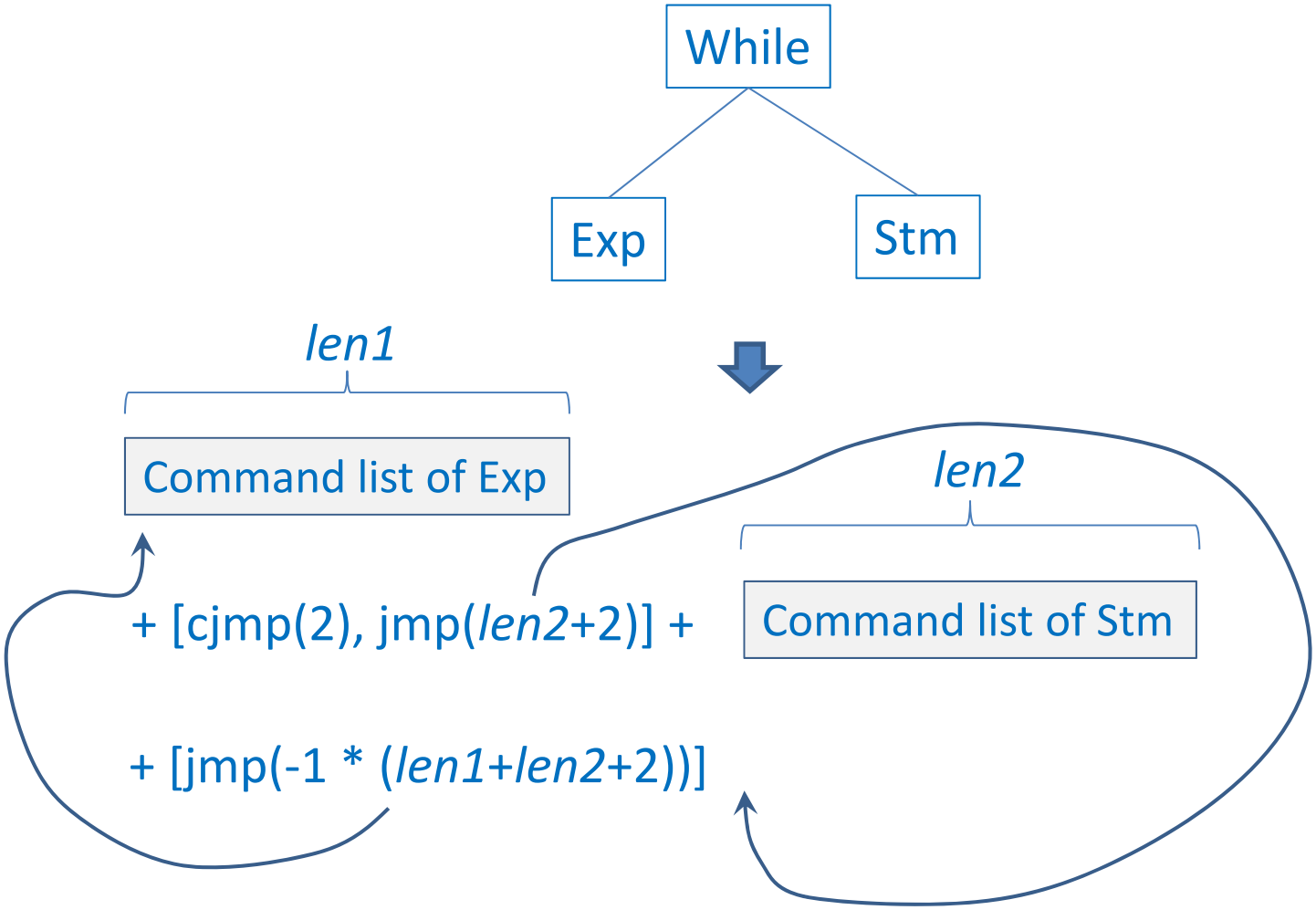
[ ]

The empty list of commands

Compiler for Minila



Compiler for Minila



Compiler for Minila

```
class EmptyParseTree(StmParseTree):
    ...
    def compile(self):
        return []
```

EmptyStm .compile()

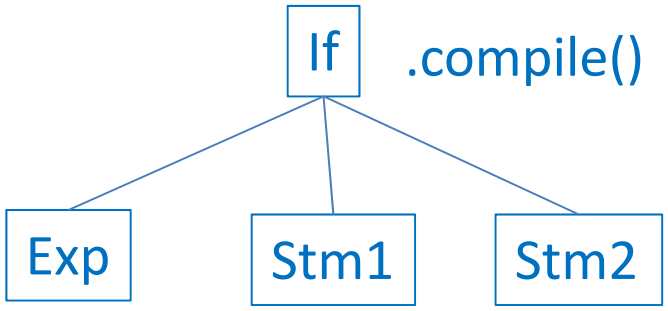
generates the empty list of commands

[ ]

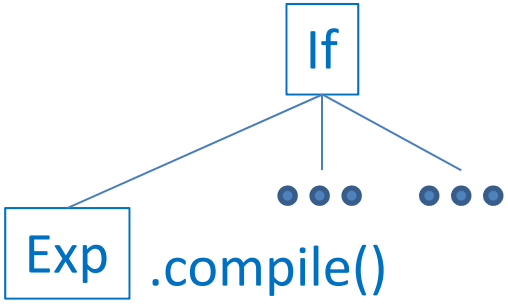
Compiler for Minila

```
class IfParseTree(StmParseTree):
    exp = ExpParseTree()
    stm1 = StmParseTree()
    stm2 = StmParseTree()
    ...
    def compile(self):
        cl1 = self.exp.compile()
        cl2 = self.stm1.compile()
        cl3 = self.stm2.compile()
        cl4 = [Command(CName.CJMP,2), Command(CName.JMP,len(cl2) + 2)]
        cl5 = [Command(CName.JMP,len(cl3) + 1)]
        return cl1 + cl4 + cl2 + cl5 + cl3
```

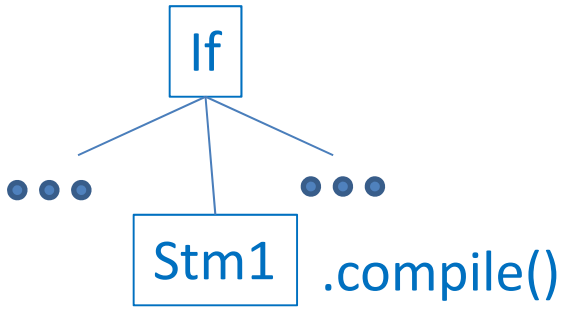
Compiler for Minila

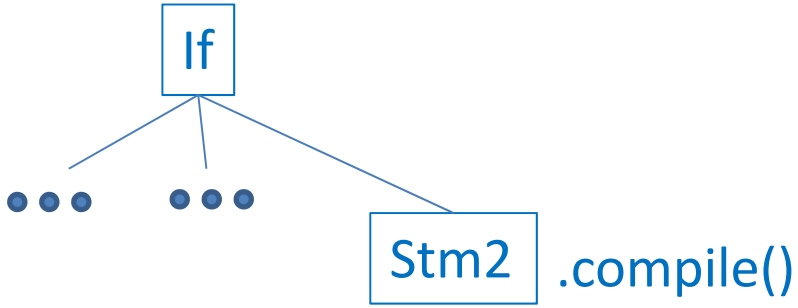


Command list of Exp

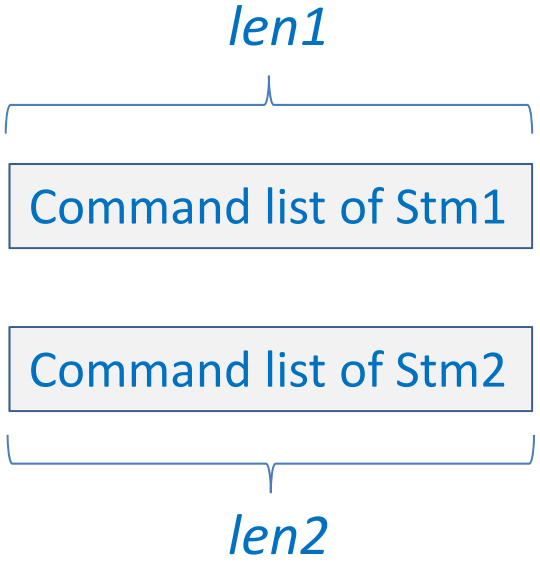


Compiler for Minila

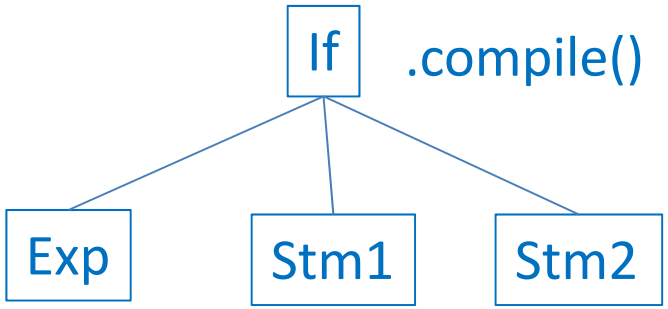




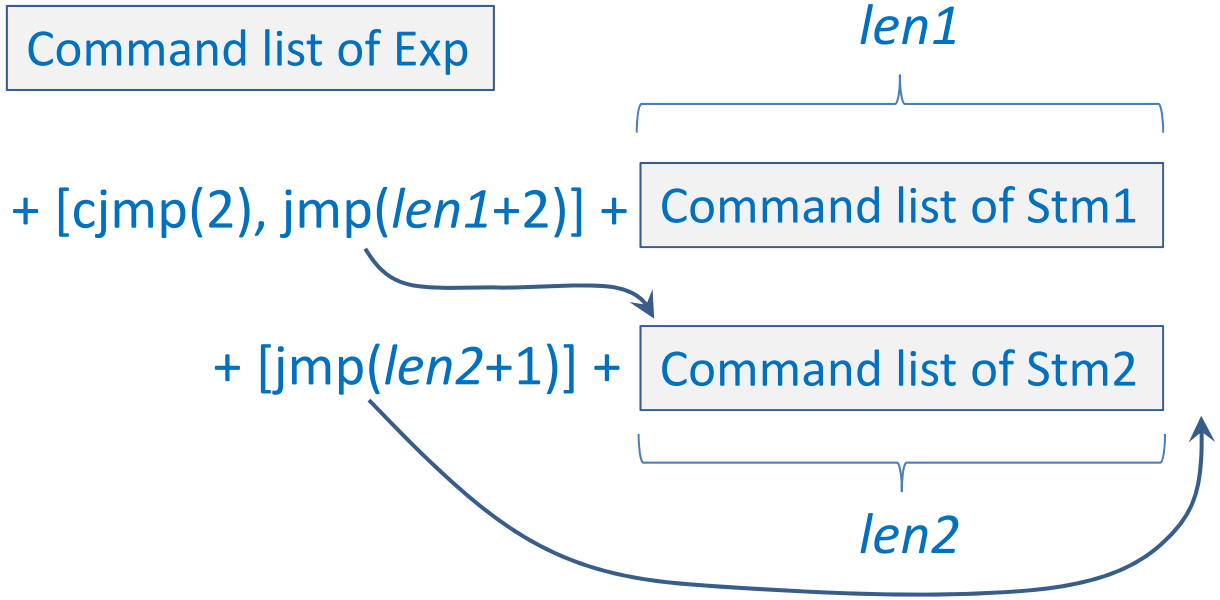
Command list of Exp



Compiler for Minila



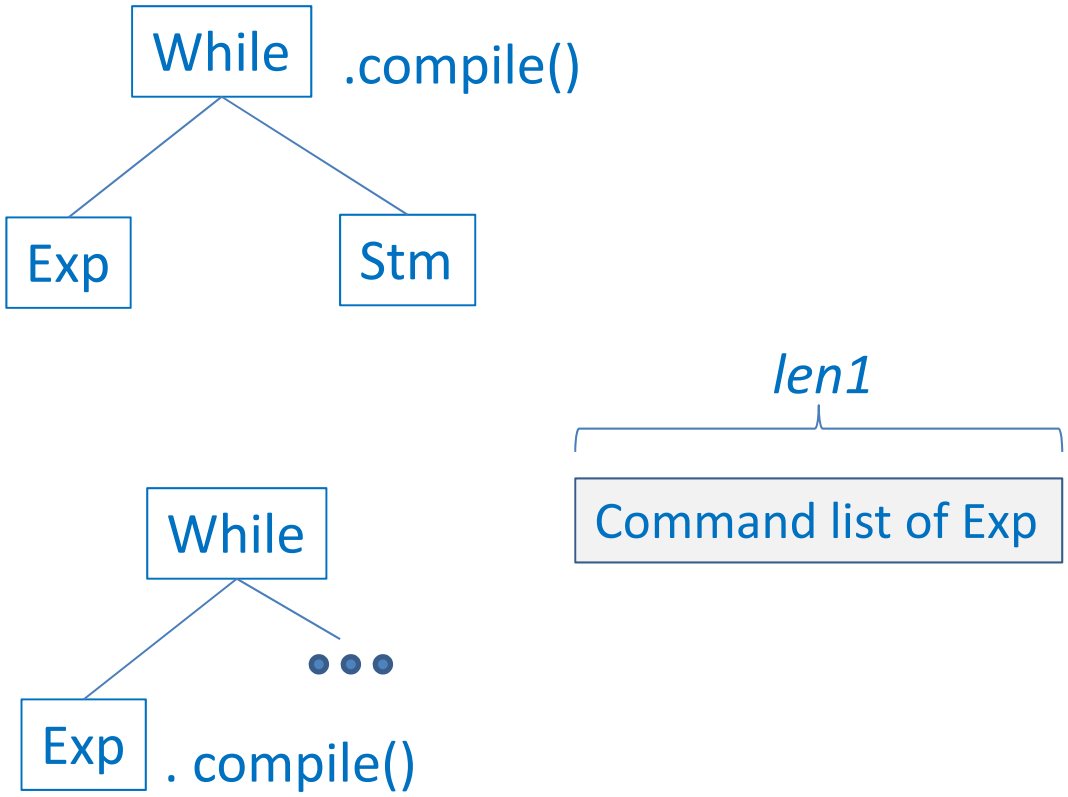
It generates the following:



Compiler for Minila

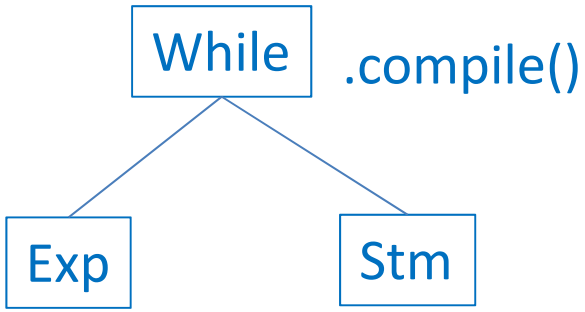
```
class WhileParseTree(StmParseTree):
    exp = ExpParseTree()
    stm = StmParseTree()
    ...
    def compile(self):
        cl1 = self.exp.compile()
        cl2 = self.stm.compile()
        size1 = len(cl1)
        size2 = len(cl2)
        cl3 = [Command(CName.CJMP,2), Command(CName.JMP,size2 + 2)]
        cl4 = [Command(CName.JMP,-1 * (size1 + size2 + 2))]
        return cl1 +  cl3 + cl2 + cl4
```

Compiler for Minila

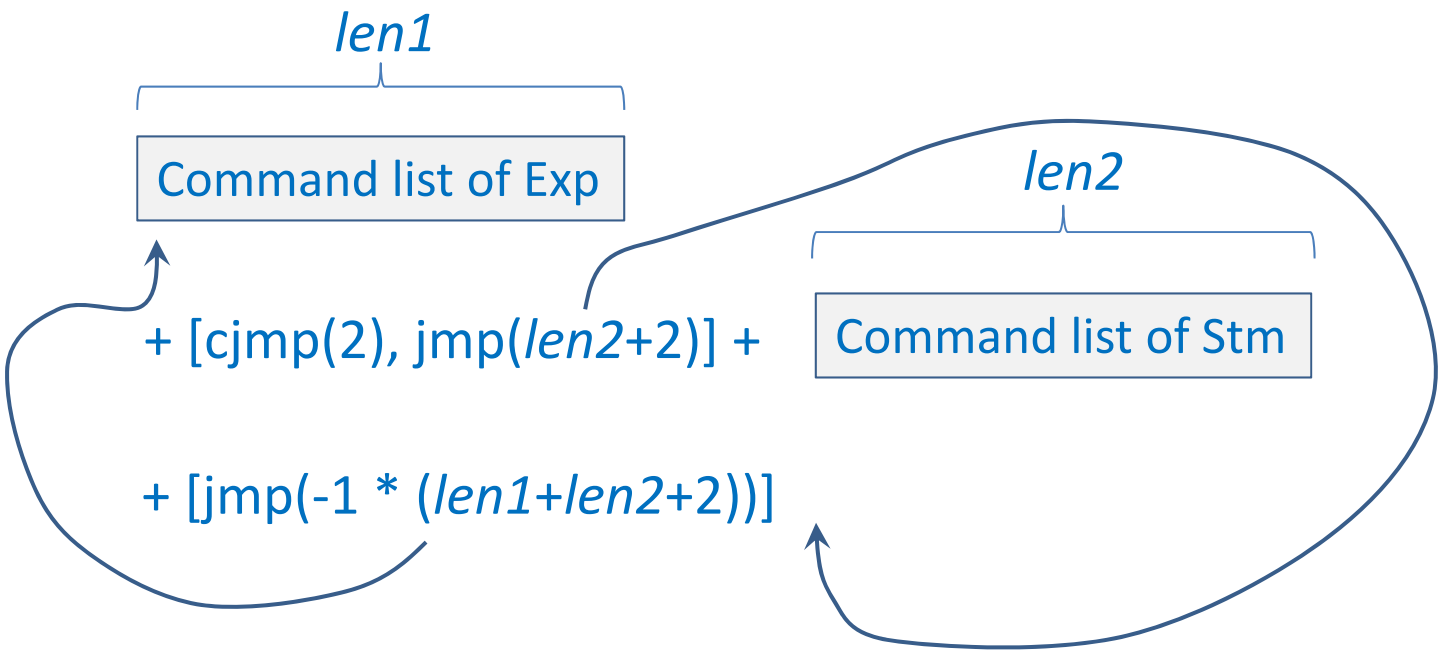


Compiler for Minila

Compiler for Minila



It generates the following:



Compiler for Minila

- Let *pgm* be a program in Minila.
- After generating the command list from *pgm*, the command quit is finally generated.

Command list of *pgm* + [quit]

- To this end, we add the method genCode() to the class StmParseTree.

Compiler for Minila

```
class StmParseTree(object):
    ...
    def genCode(self):
        return self.compile() + [Command(CName.QUIT,None)]
```

pgm .genCode()

It invokes the method genCode() in StmParseTree because genCode() is not defined in each of the five statement classes.

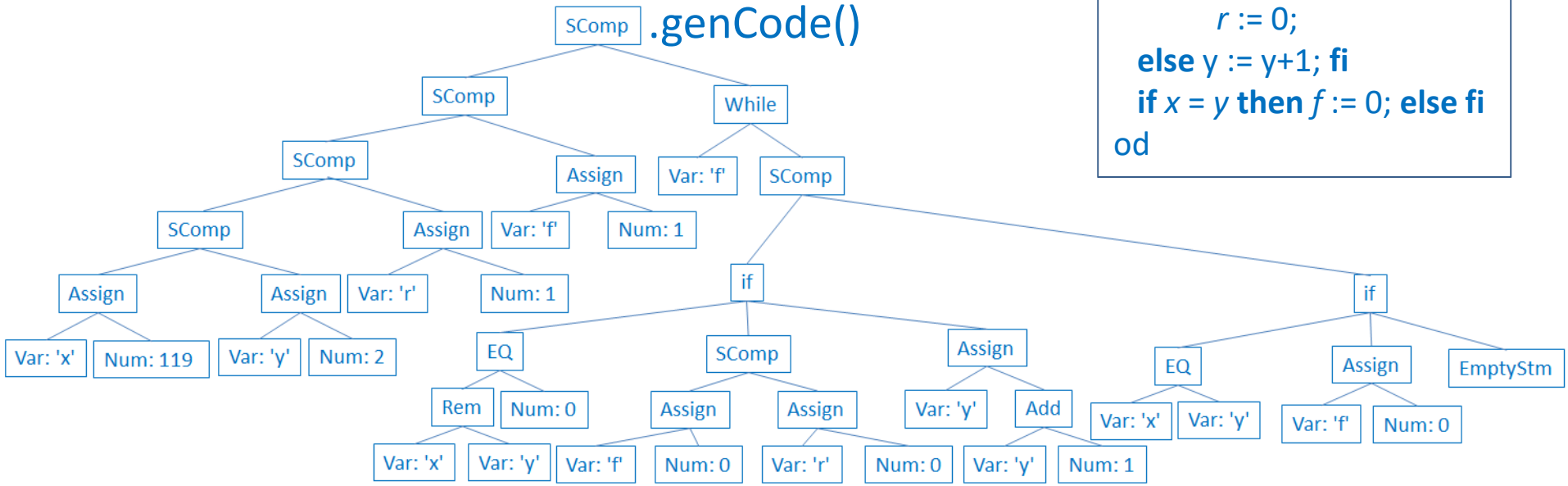
Command list of *pgm* + [quit]

generated by

pgm .compile()

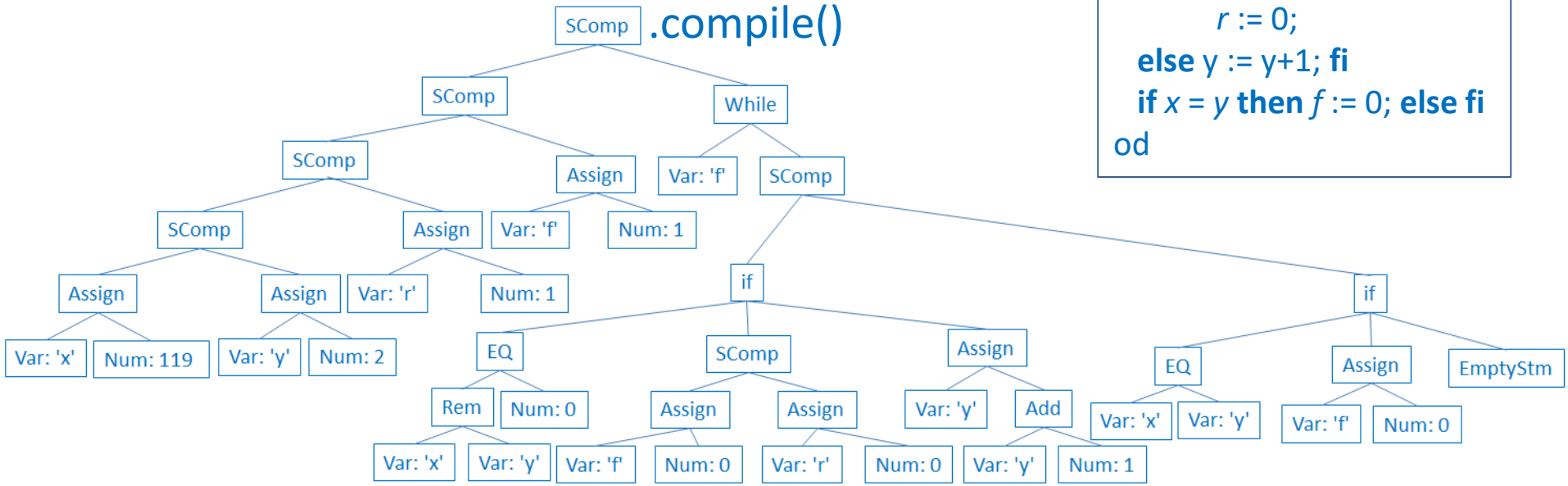
Compiler for Minila

```
x := 119;
y := 2;
r := 1;
f := 1;
while f do
   if x % y = 0
   then f := 0;
        r := 0;
  else y := y+1; fi
  if x = y then f := 0; else fi
od
```



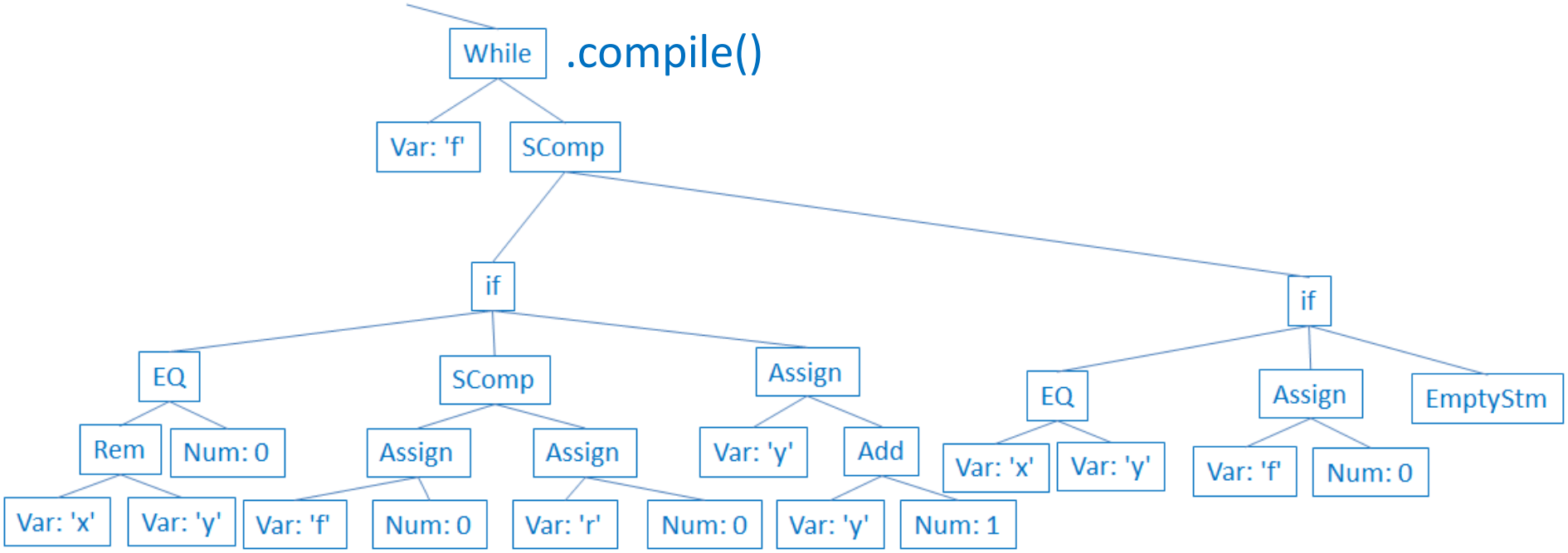
Compiler for Minila

```
x := 119;
y := 2;
r := 1;
f := 1;
while f do
   if x % y = 0
   then f := 0;
        r := 0;
  else y := y+1; fi
  if x = y then f := 0; else fi
od
```



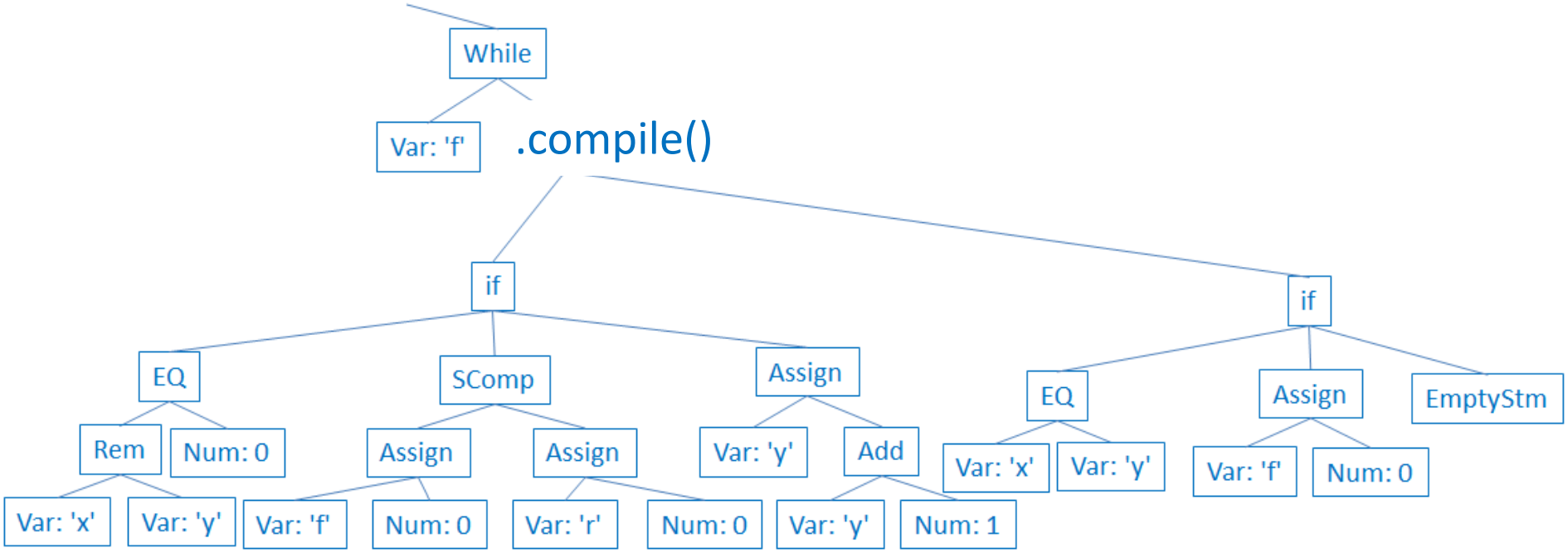
Interpreter for Minila

[push(119), store(x), push(2), store(y), push(1), store(r), push(1), store(f),



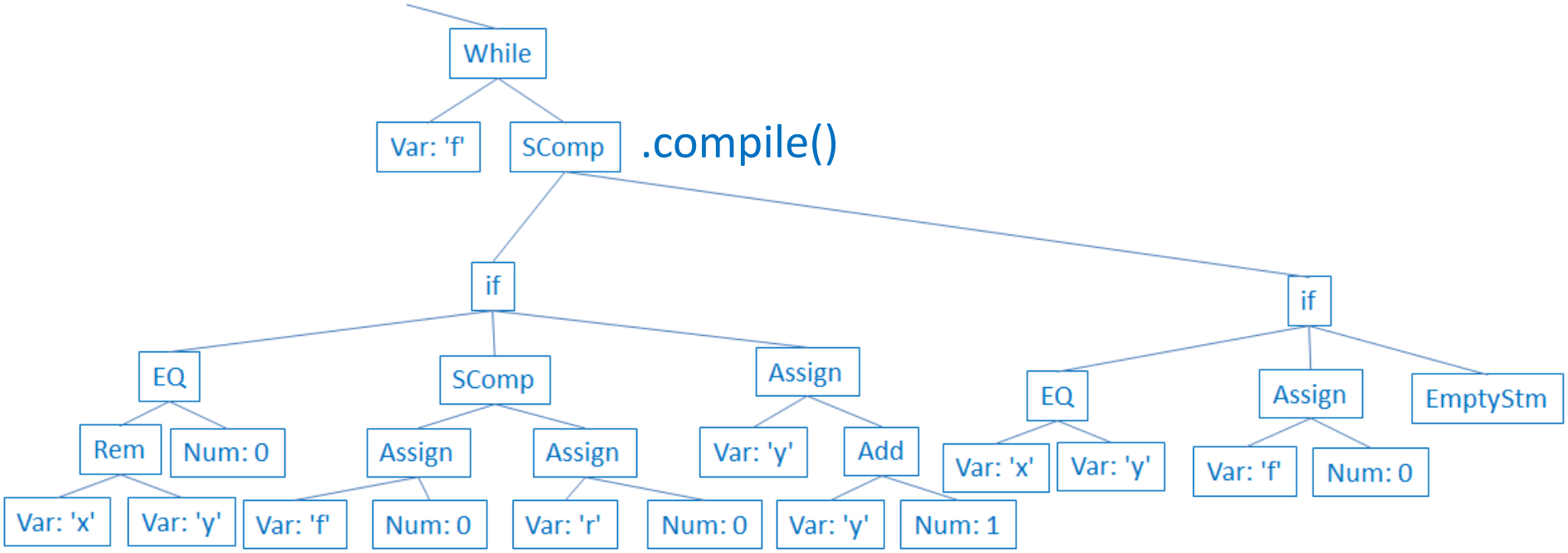
Interpreter for Minila

[push(119), store(x), push(2), store(y), push(1), store(r), push(1), store(f),
load(f), cjmp(2), jmp(???),



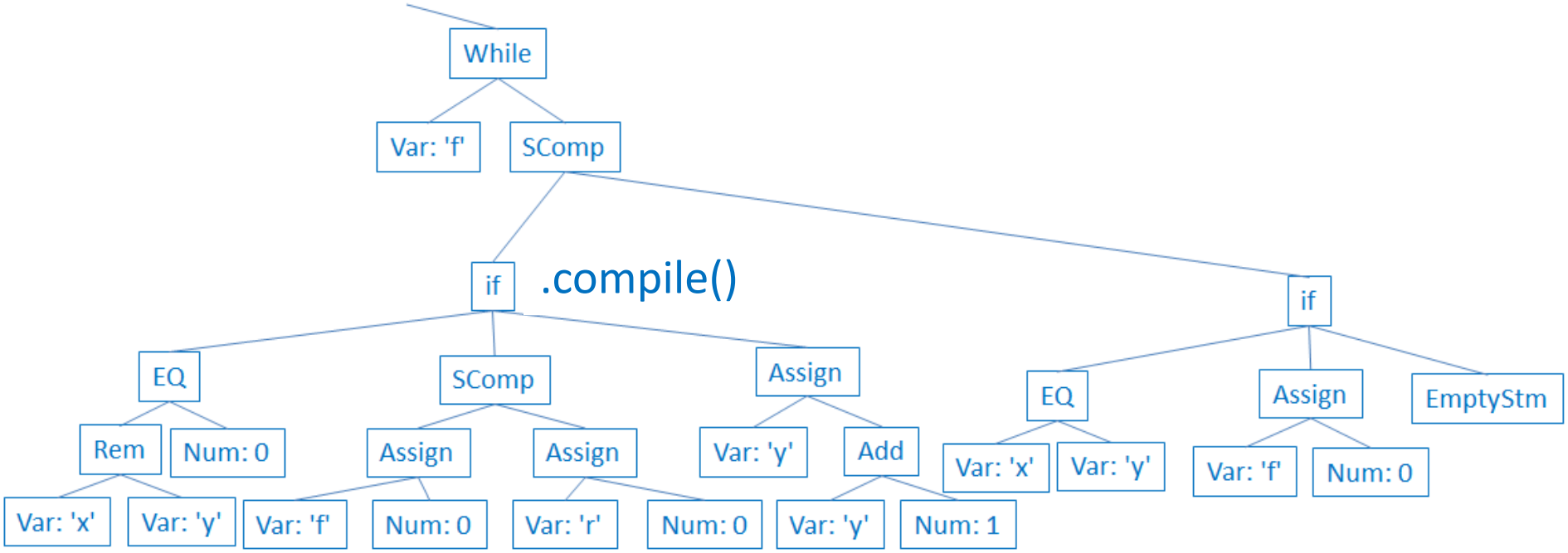
Interpreter for Minila

[push(119), store(x), push(2), store(y), push(1), store(r), push(1), store(f),
load(f), cjmp(2), jmp(???),



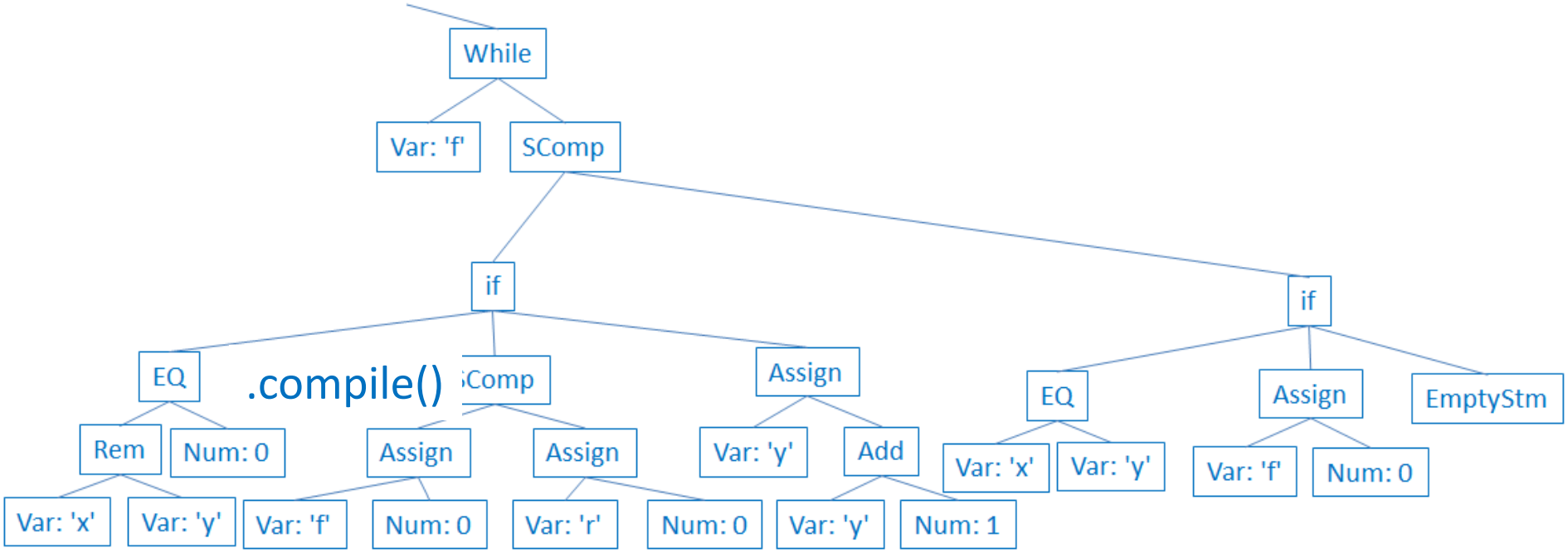
Interpreter for Minila

[push(119), store(x), push(2), store(y), push(1), store(r), push(1), store(f),
load(f), cjmp(2), jmp(???),



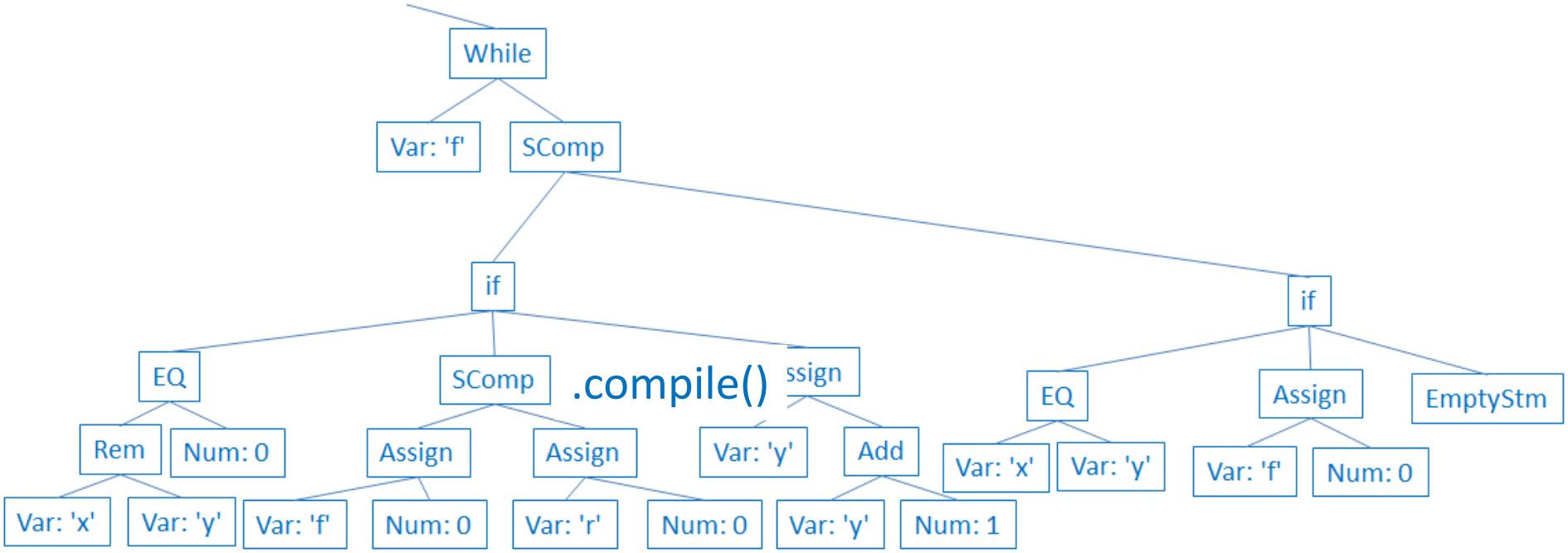
Interpreter for Minila

[push(119), store(x), push(2), store(y), push(1), store(r), push(1), store(f),
load(f), cjmp(2), jmp(???),
load(x), load(y), rem, push(0), eq, cjmp(2), jmp(???),



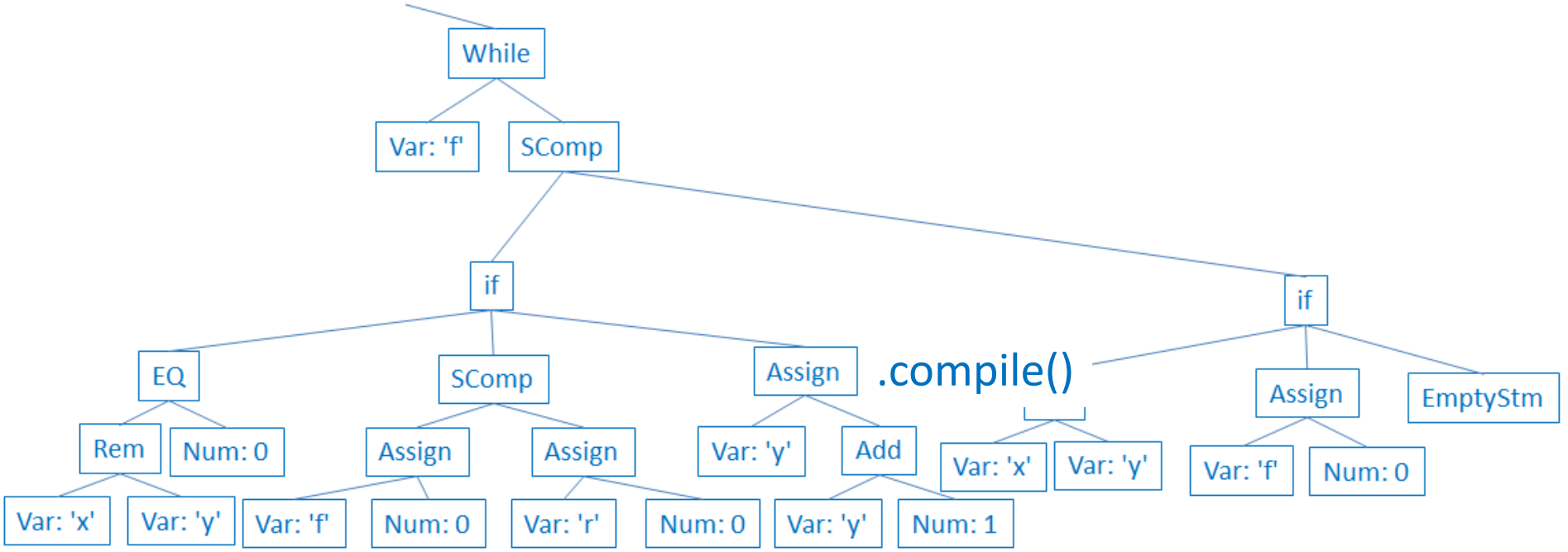
Interpreter for Minila

[push(119), store(x), push(2), store(y), push(1), store(r), push(1), store(f),
load(f), cjmp(2), jmp(???),
load(x), load(y), rem, push(0), eq, cjmp(2), jmp(???),
push(0), store(f), push(0), store(r), jmp(???),



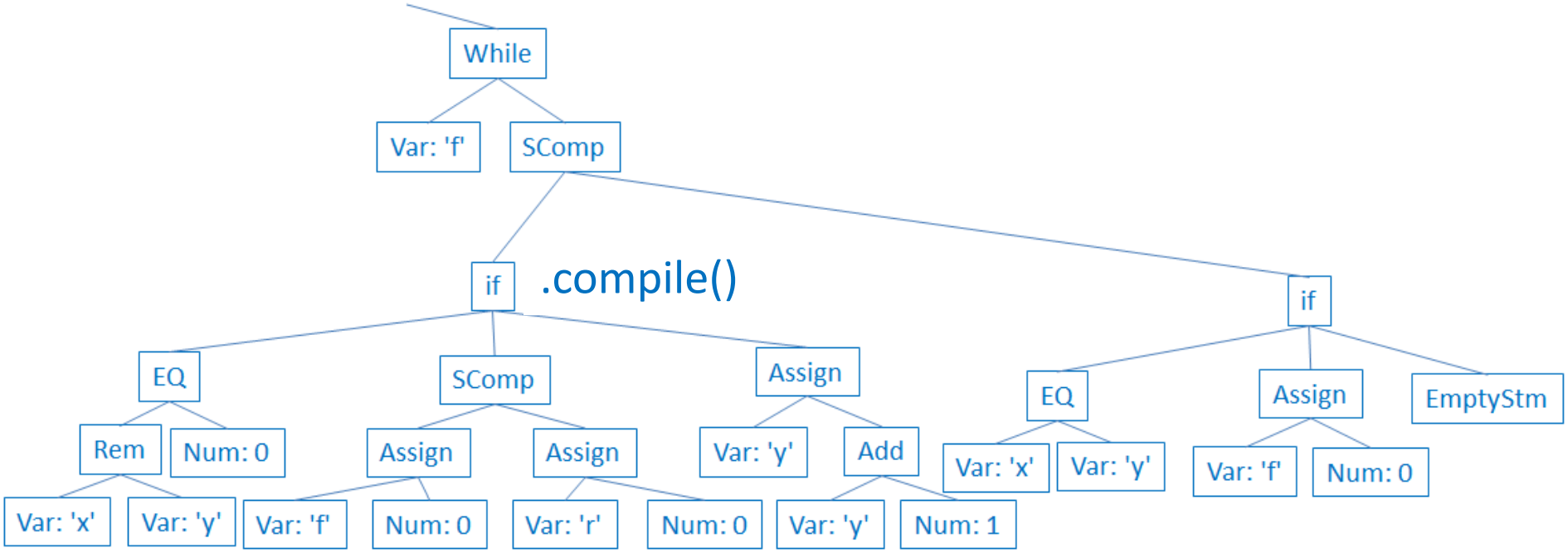
Interpreter for Minila

[push(119), store(x), push(2), store(y), push(1), store(r), push(1), store(f),
load(f), cjmp(2), jmp(???),
load(x), load(y), rem, push(0), eq, cjmp(2), jmp(???),
push(0), store(f), push(0), store(r), jmp(???),
load(y), push(1), add, store(y),



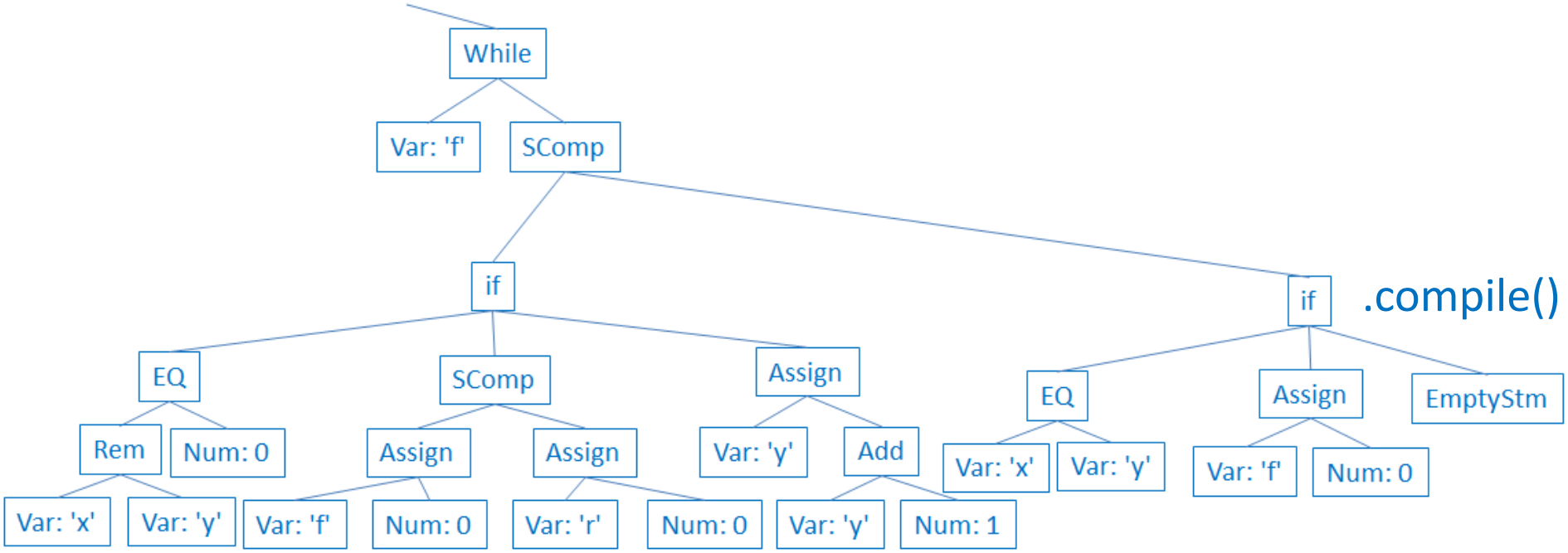
Interpreter for Minila

[push(119), store(x), push(2), store(y), push(1), store(r), push(1), store(f),
load(f), cjmp(2), jmp(???),
load(x), load(y), rem, push(0), eq, cjmp(2), jmp(6),
push(0), store(f), push(0), store(r), jmp(5),
load(y), push(1), add, store(y),



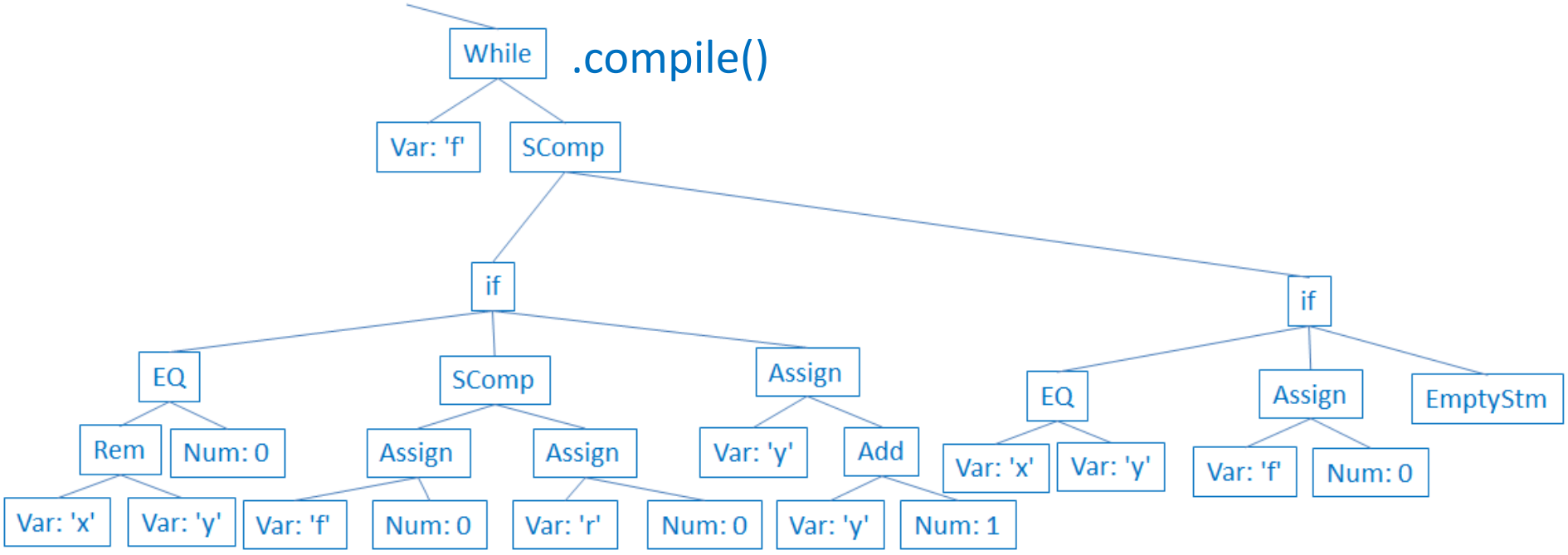
Interpreter for Minila

[push(119), store(x), push(2), store(y), push(1), store(r), push(1), store(f),
load(f), cjmp(2), jmp(???),
load(x), load(y), rem, push(0), eq, cjmp(2), jmp(6),
push(0), store(f), push(0), store(r), jmp(5),
load(y), push(1), add, store(y),
load(x), load(y), eq, cjmp(2), jmp(4), push(0), store(f), jmp(1),



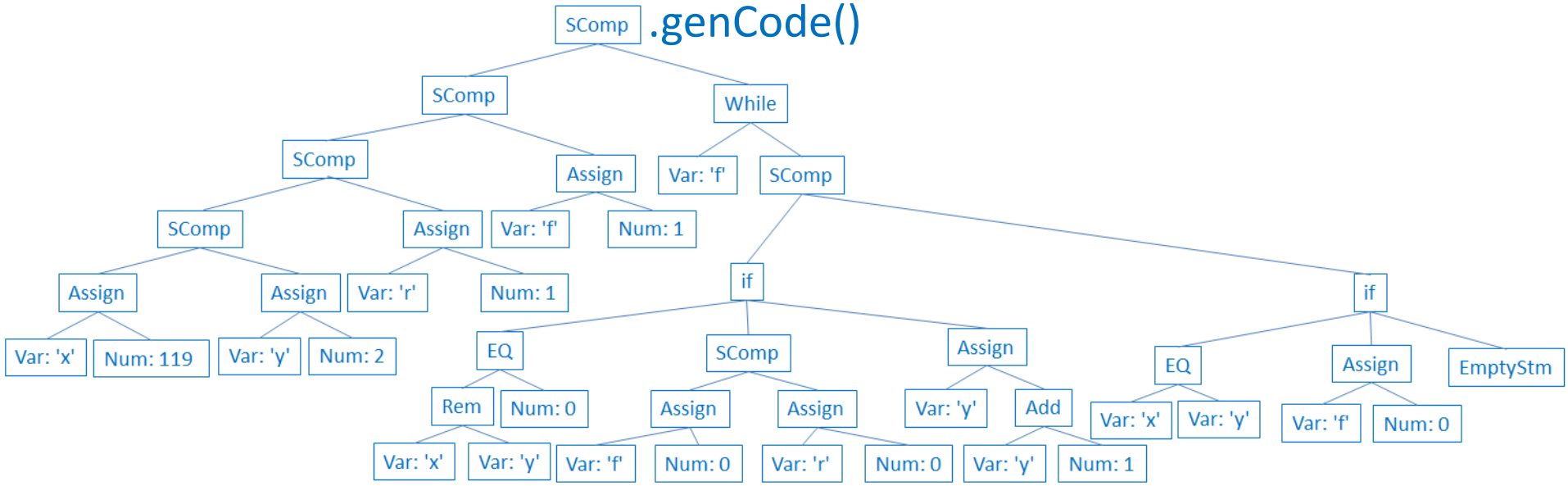
Interpreter for Minila

[push(119), store(x), push(2), store(y), push(1), store(r), push(1), store(f),
load(f), cjmp(2), jmp(26),
load(x), load(y), rem, push(0), eq, cjmp(2), jmp(6),
push(0), store(f), push(0), store(r), jmp(5),
load(y), push(1), add, store(y),
load(x), load(y), eq, cjmp(2), jmp(4), push(0), store(f), jmp(1), jmp(-27),



Interpreter for Minila

[push(119), store(x), push(2), store(y), push(1), store(r), push(1), store(f),
load(f), cjmp(2), jmp(26),
load(x), load(y), rem, push(0), eq, cjmp(2), jmp(6),
push(0), store(f), push(0), store(r), jmp(5),
load(y), push(1), add, store(y),
load(x), load(y), eq, cjmp(2), jmp(4), push(0), store(f), jmp(1), jmp(-27),
quit]



Interpreter for Minila

```
from scan import *
from parse import *
from vm import *
fact = ' '\
'    x := 1;'\
'    y := 1;'\
'    while y < 5'\
'    do'\
'        x := x * y;'\
'        y := y + 1;'\
'    od'
tl = scan(fact)
tlo = TokenList(tl)
pt = tlo.parse()
cl = pt.genCode()
print(l2s(cl))
print(VM(cl).run())
```

```
from scan import *
from parse import *
from vm import *
fact = ' '\
'    x := 1;'\
'    y := 1;'\
'    while y < 10 || y = 10'\
'    do'\
'        x := x * y;'\
'        y := y + 1;'\
'    od'
tl = scan(fact)
tlo = TokenList(tl)
pt = tlo.parse()
cl = pt.genCode()
print(l2s(cl))
print(VM(cl).run())
```

Interpreter for Minila

```
from scan import *
from parse import *
from vm import *
gcd = ' '\
'    x := 19110; '\
'    y := 17850; '\
'    while y != 0 do '\
'      tmp := x%y; '\
'      x := y; '\
'      y := tmp; '\
'    od '
tl = scan(gcd)
tlo = TokenList(tl)
pt = tlo.parse()
cl = pt.genCode()
print(l2s(cl))
print(VM(cl).run())
```

Interpreter for Minila

```
from scan import *
from parse import *
from vm import *
isPrime = ' '\
'   x := 119; '\
'   y := 2; '\
'   r := 1; '\
'   f := 1; '\
'   while f do '\
'      if x % y = 0 '\
'      then f := 0; '\
'               r := 0; '\
'     else y := y+1; fi '\
'     if x = y then f := 0; else fi '\
'   od '
```

```
tl = scan(isPrime)
tlo = TokenList(tl)
pt = tlo.parse()
cl = pt.genCode()
print(l2s(cl))
print(VM(cl).run())
```

Interpreter for Minila

```
from scan import *
from parse import *
from vm import *
sr = ' '\
'   v0 := 2000000000000000; '\
'   v1 := 0; '\
'   v2 := v0; '\
'   while v1 != v2 do '\
'    if (v2-v1)%2 = 0 '\
'     then v3 := v1+(v2-v1)/2; '\
'     else v3 := v1+(v2-v1)/2+1; '\
'    fi '\
'    if v3*v3 > v0 '\
'     then v2 := v3-1; '\
'     else v1 := v3; '\
'    fi '\
'   od '
```

```
tl = scan(sr)
tlo = TokenList(tl)
pt = tlo.parse()
cl = pt.genCode()
print(l2s(cl))
print(VM(cl).run())
```