Skywalker

TDP019 – VT 2012

Anders Ydremark – andyd643@student.liu.se
Joakim Kvarnström – joakv409@student.liu.se

Linköpings universitet

Innehållsförteckning

1. Inledning

Programmeringsspråket Skywalker är ett simpelt språk, konstruerat för att göra det enkelt att skriva rutiner för att manipulera motorer och servon.

1.1 Bakgrund

När vi började fundera på vad vi skulle skapa för språk hittade vi en artikel där det visades hur man kunde koppla en kontroll från radiostyrda flygplan till datorn. Eftersom vi båda tycker det är kul med radiostyrda flygplan föddes iden om ett språk där man kunde skriva intruktioner till flygplanet som det sedan utför. Då vi inte kunde hitta något redan befintligt språk för att göra detta tyckte vi att det skulle vara roligt att genomföra det.

1.2 Syfte

Syftet med Skywalker är att få bort den mer avancerade bakomliggande koden och skapa ett gränssnitt i form av ett språk där det ska vara lättare att skriva instruktioner.Språket ska även kännas mer avsett för just styrning av motorer och servon så att man slipper lära sig ett större och mer avancerat programmeringsspråk.

1.3 Målgrupp

Skywalkers målgrupp är användare i alla åldrar med intresse för radiostyrning, automation och enkel programmering. Det är skapat för att vara enkelt och anpassat för just radiostyrda fordon och kräver ingen större förkunskap inom programmering.

2. Användarhandledning

I denna del av rapporten kommer vi att förklara programspråket mer i detalj, dess målgrupp och ge användarexempel.

2.1 Konstruktioner

Skywalker är uppbyggt runt konceptet av en tidslinje. Då man skriver sitt program ska man ha i åtanke att då programmet startas kommer en tidslinje skapas. Man kommer t.ex. att kunna använda sig utav funktionen ”Wait”. Denna funktion tar ett argument som bestämmer hur länge programmet ska vänta innan det utför nästa kommando.

2.2 Extern fil

För att få programspråket Skywalker att fungera behövs en extern fil. Denna fil används för att göra en simulering av vad som händer när man kör programmet. Denna fil går att skriva om så att man t.ex. kan använda den för att styra en riktig radiokontroll.Exempel:

```
1  @@control = {}
2
3  def scheduler
4  lista = [Main]
5  scope = [Main]
6  i = 0
7
8  loop do
9  if (scope.empty?)
10 break
11 end
12
13 con = lista[i].resume
14
15 if (con =~ /(wait) (.*)/)
16 # Kod som kärs när "wait" kallas.
17 elsif (con =~ /(call) (.*)/)
18      # Kod som körs när man kalla på en rutin.
19 temp = eval($2)
20 lista.insert(temp)
21 scope<< temp
22 elsif (con == :end)
23      # Kod som körs i slutet av en rutin.
24 scope.pop
25 end
26 lista.insert(i+1, scope.last)
27 i += 1
28   end
29 end
```

2.3 Första steget - Interfaces

Det första steget I skapandet av ett Skywalker program är att defineraett ”interface”. Ett interface deklarerar vilka motorer i den inkluderade rubyfilen som ska användas. De läggs i en lista där man kan plocka ut den specifika motorn och manipulera just denna. Man ska även inkludera den externa rubyfilen med definerade motorer och servon i detta interface. Exempel:

```
interface
     external(def.rb);
     Controls = {mainEngine, secondEngine, mainServo};
end
```

Vi har här skapat en lista som heter “Controls” som innehåller namnen på de olika motorerna och servona som kan kommas åt.

2.4 Andra steget – Routine Main

Det andra steget i att skapa ett Skywalker-program är att skapa en main-rutin. Main-rutinen är själva kärnan i programmet då det är här alla ”anrop” sker. Man kan även skapa subrutiner med instruktioner som sedan går att anropa från main-rutinen. I exemplet nedan har vi skapat en subrutin som heter ”Roll”. Denna subrutin kallar vi sedan på med hjälp av kommandot ”call”. Rutinen ”Roll” kommer då att köras tills den är färdig. Därefter kommer main-rutinen att fortsätta där den var. Exempel:

```
routine Roll
     Controls[mainEngine] = 55;
     Controls[mainServo] = 20;
     Wait(1);
     Controls[mainServo] = 25;
     Wait(1);
     Controls[mainServo] = 30;
     Wait(1);
     Controls[mainServo] = 35;
end
```

När ett program körs, utförs kommandona i rutinen uppifrån och ned. Om vi t.ex. skriver:

```
routine Main
     Controls[mainEngine] = 35;
     wait(10);
     call(Roll);
     Controls[mainServo] = -35;
     wait(10);
end
```

så sker följande stegvis förklarat:

1. Programmet exikveras vid tid 0.
2. Vid tid 0, Motorn med namnet ”mainEngine” sätter thrust till 35.

3. Programmet väntar i 10 sekunder. (”mainEngine” är fortfarande 35).
4. Vid tid 10 kallar vi på subrutinen ”Roll” som kommer att köras i 3 sekunder.
5. Vid tid 13, sätts ”mainServo” till -35.
6. Programmet väntar i ytterligare 10 sekunder (”mainEngine” och ”mainServo” är fortfarande satta till 35 respektive -35)
7. Vid tid 23, programmet avslutas och alla värden sätts tillbaka till ursprunglig variabel.

I main-rutinen sätts motorernas och servonas variablar alltid tillbaka till sitt ursprungliga ”nollvärde” efter att rutinen är avslutad. Detta är inte fallet i en subrutin, då värdena som sätts i en subrutin kommer att stå kvar även när man ”kommer tillbaka” till main-rutinen.

2.5 Relationsoperatorer

Det finns ett antal relationsoperatorer i programspråket Skywalker. Dessa används för att kontrollera om något t.ex. är större än något annat, mindre än något annat, mindre än eller lika med något annat osv. De relationsoperatorer som finns är:

<	Mindre än
>	Större än
<=	Mindre än eller lika med
>=	Större än eller lika med
==	Lika med
!=	Skilt ifrån

2.6 Styrsatser

Det finns två olika sorters kontrollsatser i Skywalker. Dessa används för att kontrollera om jämförelser är sanna eller falska eller för att utföra operationer medan något är sant.

2.6.1 Villkorssatser

Det går att använda sig av ”if”- och ”else”-satser i Skywalker. En if sats används för att säga; ”om något är sant, gör detta”.Exempel:

```
a = 10;         # Tilldelar ”a” värdet 10.
if(a == 5)      # Säger att om ”a” är lika med 5
     b = 20;    # så ska vi sätta variabeln ”b” till 20.
else            # Om det inte är sant
     b = 25;    # Ska vi sätta variabeln ”b” till 25.
```

2.6.2 Repetitionssats

En while-loop används för att göra något medan någonting är sant. Ett bra exempel är att ta exemplet vi använde oss av i del ”2.5 Andra steget – Routine Main” där vi skapade en subrutin. I denna subrutin upprepade vi ett kommando flera gånger och gjorde hela tiden små ändringar. Ett smidigare sätt att skriva denna kod på kan vara:

```
routine Roll
     Controls[mainEngine] = 55;
     Controls[mainServo] = 20;
     a = 0;
     while(a <= 5)
          wait(1);
          Controls[mainServo] =a;
          a += 1;
     end
end
```

2.7 Aritmetiska uttryck

Det går även att använda sig utav olika aritmetiska uttryck i programspråket. Det går att utföra additioner, subtraktioner, multiplikationer och divisioner. Operationerna multiplikation och division har högst prioritet, dvs, de kommer att beräknas före addition och subtraktion som har lägre prioritet. Ett uttryck med parenteser går före både addition, subtraktion, multiplikation och division. Exempel:

```
1 + 2 * 3       # Beräknas 2 * 3 = 6 och sedan 6 + 1 = 7
(1 + 2) * 3     # Beräknas 1 + 2 = 3 och sedan 3 * 3 = 9
```

2.8 Tilldelning

Det går att tilldela en variabel ett värde. Detta görs genom att skriva namnet på variabeln följt av lika med tecken följt av ett värde. Detta värde kan antingen vara ett tal, en annan variabel eller ett aritmetiskt uttryck. Exempel:

```
a = 3           # Variabeln ”a” tilldelas värdet 3
b = 2           # Variabeln ”b” tilldelas värdet 2
c = a + b       # Variabeln ”c” tilldelas världet av ”a” plus ”b”
```

3. Systemdokumentation

Systemdokumentationen kommer att förklara hur språket är byggt mer tekniskt vad gäller både grammatik och kod.

3.1 Implementation

Skywalker är implementerat med hjälp av rdparser.rb som följde med kursen och som även använts i den parallella ruby-kursen TDP007. Koden läses in först med en lexikal analys (tokens), sedan skapas noder utifrån konstruktioner beskrivet i grammatiken. Sist kompilerar vi alla noder med hjälp av compile-funktionen i varje nod.

3.2 Lexikal analys

Källkoden läses in som en sträng som bryts ner i tokens med hjälp av reguljära uttryck. Vi plockar bort alla blanksteg och sedan läser vi in alla flyttal, integers, strängar och operatorer för sig. Därefter har vi en token som matchar ett tecken och används för resterande tecken, t.ex parenteser.

3.3 Parsning

Tokens skapade från den lexikala analysen matchas sedan mot regler specificerade i grammatiken för språket. Utifrån dessa regler skapas ett abstrakt syntaxträd med “noder” eller klasser som lagrar uttryck. De lagrar i sin tur underliggande noder och bygger på så sätt upp ett träd av noder.

När hela källkoden är parsad och trädet är klart exekveras compile-funktionen på rotnoden. Rotnoden exekverar i sin tur alla underliggande noder vilket “rekurserar” ner till de understa noderna, vars returvärden sedan skickas upp till noden över. När rekursionen har nått tillbaka till rotnoden har hela programmet exekverats och resultatet skrivs ut till out.rb.

3.4 Kompilering

Alla noder har en ”compile”-funktion vars uppgift är att skriva ut ruby-kod. En del noder returnerar bara ut ett värde medan andra kallar på underliggande noder som t.ex kan gå igenom en lista och sedan returnera resultatet som en sträng.

Varje rutin i Skywalker kompileras till en ”Fiber”. Fördelen med Fibers gentemot exempelvis funktioner är att vi kan ”pausa” blocket och återuppta det senare. På så sätt kan vi köra en del av en rutin, bryta körningen av den rutinen, göra någonting annat och sedan återuppta körningen där vi slutade.

Call- och Wait-funktionerna kompileras till “yields”. Det innebär att de pausar kodblocket och returnerar ut en sträng. Call returnerar “call <rutin>” och Wait returnerar “wait <sek>”. Det är sedan upp till exekveringslistan i den externa filen att göra något med dessa anrop.

Ett exempel på hur kod kompileras från Skywalker till Ruby:

routine Main	Main = Fiber.new do loop do	Kompilerar “routine” till Fiber samt använder loop för att den inte ska “dö” efter första körningen
Controls[servo2] = 0;	@@control["servo2"] = 0	Controls ska vara samma i alla rutiner och blir därför en global variabel i ruby.
a = 0;	a = 0	a är en lokal variabel och blir det även i ruby då variabler är lokala i Fibers
while(a <= 10)	while(a<=10)	while-satser har samma syntax som ruby
Controls[servo] = a;	@@control["servo"] = a	Globala Controls[servo] sätts till lokala a.
call(SubRoutine);	Fiber.yield "call SubRoutine"	Ett anrop till en subrutin genererar Fiber.yield för att pausa rutinen som körs. Det är sedan upp till exekveringslistan att ta vara på returvärdet och göra något med det.
a += 1;	a += 1	Iterering av variabler ser likadant ut.
end	end	end markerar slut på styrsatser i både ruby och Skywaker
wait(5);	Fiber.yield "wait 5"	wait genererar också Fiber.yield. Fungerar på samma sätt som call
end	end	För att säga åt exekveringslistan att rutinen kört färdigt skickar vi symbolen :end. Detta måste också hanteras i den externa filen.

3.5 Omgivningshantering eller räckvidd

Då Skywalker kompilerar till ruby-kod använder vi oss utav rubys omgivningshantering. Varje rutin har dock en egen räckvidd då varje ruting kompileras till ett ”Fiber”. Då vi vill komma åt alla kontroller i alla rutiner hanteras dessa med hjälp av en global hash-tabell.

3.6 Fibers

Grunden i vår “runtime” är Fibers. Fibers är en klass i ruby som tar ett kodblock som sedan kan “pausas” och återupptas. Fibern kommer inte att köras av sig själv utan behöver exekveras med hjälp av metoden resume. När man kör resume på en fiber kommer den att exekvera all kod fram till yield eller slutet av Fibern. När all kod i Fibern är slut kommer den att vara “död” och kan inte exekveras igen. Exempel:

```
myFiber = Fiber.new do
   puts "hej"
   Fiber.yield
   puts "då"
end
myFiber.resume
=> hej
myFiber.resume
=> då
myFiber.resume
=> FiberError: dead fiber called
```

Då vi har rutiner som man ska kunna anropa flera gånger sätter vi hela blocket i en loop. Exempel:

```
myFiber = Fiber.new do
   loop do
      puts "Hej"
      Fiber.yield
   end
end
```

Detta gör att vi kan kalla på fibern hur många gånger vi vill.

Problemet med att göra så är att det inte går att se när fibern har kört all kod och börjar om igen. Detta har vi löst genom att returnera :end från vår sista Fiber.yield. På så sätt så vet vi när blocket är klart och inte längre ska finnas i exekveringsistan.

För mer information om Fibers se: http://www.ruby-doc.org/core-1.9.3/Fiber.html

3.7 Kodstandard

Skywalker är ganska löst typat och kräver inte så mycket struktur. I likhet med många andra imperativa språk kräver Skywalker dock att du har en Main-rutin. Det är den som exekveras först och som sedan kan komma att kalla på subrutiner.

Värt att nämna också är att bara sekvenser av bokstäver får användas som variabelnamn. Vi har inte någon speciell kodkonvention som vi rekommenderar men vi tror att språket ska vara så pass enkelt att det blir överflödigt.

3.8 Exekveringslista

När det kompilerade programmet körs så körs första posten i exekveringslistan. Det är den som hanterar vilken ”Fiber” som ska återupptas. När en Fiber pausas lägger vi till en ny post i exekveringslistan som återupptar den när den körs. När vi nästlar anrop sorteras dessa i listan så att de kallas i rätt ordning. När Fibern returnerar symbolen :end läggs det inte längre till någon post i listan och programmet kör vidare. När Main-rutinen returnerar :end-symbolen läggs det inte till någon mer post i listan och programmet avslutas.

3.9 Grammatik

<program>	::=	<interface><routine_list>
<routine_list>	::=	<routine> | <routine_list><routine>
<routine>	::=	“routine” [A-Z][a-zA-Z] <stmt_list> <terminator>
<interface>	::=	“interface” <interface_stmt_list> <terminator>
<interface_stmt_list>	::=	<interface_stmt> | <interface_stmt_list><interface_stmt> | <stmt_list>
<stmt_list>	::=	<stmt> | <stmt_list><stmt>
<stmt>	::=	<assign_stmt> “;” | <while_stmt> | <if_else_stmt> | <call_stmt> “;” | <wait_stmt> “;” | <control_assign_stmt> “;” | <addition> “;”
<call_stmt>	::=	“call” “(“ [A-Z][a-zA-Z] “)”
<wait_stmt>	::=	“wait” “(“ <addition> “)”
<control_assign_stmt>	::=	“Controls” “[“ [a-zA-Z] “]” “=” <addition>
<interface_stmt>	::=	<interface_assign_stmt> | <include_stmt>
<include_stmt>	::=	“external” “(“ *filename* “)”
<interface_assign_stmt>	::=	“Controls” “=” “{“ <interface_var_list> “}”
<interface_var_list>	::=	<interface_var> | <interface_var_list> “,” <interface_var>
<inteface_var>	::=	[a-zA-Z]
<assign_stmt>	::=	<identifier> “=” <addition>
<addition>	::=	<multi> | <addition><addition_oper><multi>

<multi>	::=	<primary> | <multi><multi_oper><multi>
<primary>	::=	"(" <addition> ")" | <atom>
<terminator>	::=	"end"
<identifier>	::=	[a-z]
<addition_oper>	::=	"+" | "-" | "+=" | "-="
<multi_oper>	::=	"*" | "/" | "*=" | "/="
<atom>	::=	<integer> | <float> | <identifier> | "Controls" "[" [a-zA-Z] "]"
<while_stmt>	::=	"while" "(" <bool_expr> ")" <stmt_list> <terminator>
<boolean>	::=	"true" | "false"
<bool_expr>	::=	<addition><rel_oper><multi> | <boolean>
<if_stmt>	::=	"if" "(" <bool_expr> ")" <stmt_list>
<else_stmt>	::=	"else" <stmt_list>
<if_else_stmt>	::=	<if_stmt><terminato> | <if_stmt><else_stmt><terminator>
<rel_oper>	::=	"= =" | "<" | ">" | ">=" | "<="

		| ”!=”
<integer>	::=	[0-9]
<float>	::=	[0.00-9.99]

4 Kod

I det här avsnittet finns koden för språket, förutom detta behövs rdparsern för att kunna köra programmet.

4.1 skywalker.rb

Här är koden från filen ”skywalker.rb”:

```
require './parser.rb'
require './nodes.rb'
require 'fiber'

class Skywalker
def initialize
    @skywalker = Parser.new("skywalker") do
token(/\s+/)
token(/\d+\.\d+/) {|float| float.to_f }
token(/\d+/) {|int| int.to_i }
token(/\w+/) {|str| str }
token(/(!=|<=|>=|==|\+=|-=|\*=|\/=)/) {|op| op }
token(/./) {|wldcrd| wldcrd }

start :program do
match(:interface, :routine_list) {
          |int, rout| @@interface = int, @@code = rout }
end

rule :routine_list do
match(:routine_list, :routine) {
          |lst, rout| lst<<rout }
match(:routine) {
          |rout| RoutineListNode.new<<rout }
end

rule :interface do
match("interface", :interface_stmt_list, :terminator) {
          |_, stmt_lst, _, _| stmt_lst }
end

rule :routine do
        match("routine", /^[A-Z][a-zA-Z]+/, :stmt_list,
:terminator) {
          |_, name, stmt_list, _| RoutineNode.new(name,
stmt_list) }
end

```

```
rule :stmt_list do
match(:stmt) {
          |stmt| StmtListNode.new<<StmtNode.new(stmt) }
match(:stmt_list, :stmt) {
          |lst, stmt| lst<<StmtNode.new(stmt) }
end

rule :interface_stmt_list do
match(:interface_stmt) {
          |stmt| StmtListNode.new<<StmtNode.new(stmt) }
match(:interface_stmt_list, :interface_stmt) {
          |lst, _, stmt| lst<<StmtNode.new(stmt) }
end

rule :interface_stmt do
match(:include_stmt, ";") {|stmt, _| stmt }
match(:interface_assign_stmt, ";") {|stmt, _| stmt }
end

rule :include_stmt do
match("external", "(", /.+/, ".", /.+/, ")") {
          |_, _, name, dot, ext, _|
IncludeNode.new("#{name}#{dot}#{ext}") }
end

rule :interface_assign_stmt do
match("Controls", "=", "{", :interface_var_list, "}") {
|_, _, _, var_lst, _| ControlsAssignNode.new(var_lst, 0) }
end

rule :interface_var_list do
match(:interface_var) {
          |var|
InterfaceVarListNode.new<<InterfaceVarNode.new(var) }
match(:interface_var_list, ",", :interface_var) {
          |lst, _, var| lst<<InterfaceVarNode.new(var) }
end

rule :interface_var do
match(/[a-zA-Z]/) {|var| var }
end

rule :control_assign_stmt do
match("Controls", "[", /[a-zA-Z]/, "]", "=", :addition) {
          |_, _, name, _, _, expr| ControlsAssignNode.new(name,
expr) }
end
```

```
86
87  rule :stmt do
88  match(:if_else_stmt) {|stmt| stmt }
89  match(:while_stmt) {|stmt| stmt }
90  match(:control_assign_stmt, ";") {|stmt, _| stmt }
91  match(:assign_stmt, ";") {|stmt, _| stmt }
92  match(:call_stmt, ";") {|stmt, _| stmt }
93  match(:wait_stmt, ";") {|stmt, _| stmt }
94  match(:addition, ";") {|expr, _| expr }
95  end
96
97  rule :addition do
98  match(:multi)
99  match(:addition, :addition_oper, :multi) {
100           |left, op, right| AdditionNode.new(op, left, right) }
101 end
102
103 rule :multi do
104 match(:primary)
105 match(:multi, :multi_oper, :multi) {
106           |left, op, right| MultiNode.new(op, left, right) }
107 end
108
109 rule :primary do
110 match("(", :addition, ")") {
111           |_, expr, _| ParenthesesNode.new(expr) }
112 match(:atom)
113 end
114
115 rule :atom do
116 match(Float) {
117           |float| FloatNode.new(float) }
118 match(Integer) {
119           |int| IntegerNode.new(int) }
120 match(:identifier)
121 match("Controls" , "[", /[a-zA-Z]+/, "]") {
122           |_, _, name, _|
123 IdentifierNode.new("@@control[\"#{name}\"]") }
124 end
125
126 rule :boolean do
127 match("true") {
128           |bool| bool }
129 match("false") {
130           |bool| bool }
131 end
132
```

```
rule :terminator do
match("end") {
          |a| a }
end

rule :addition_oper do
match(/\+|-|\+=|-=/) {
          |op| op }
end

rule :multi_oper do
match(/\*|\/|\*=|\/=/) {
          |op| op }
end

rule :rel_oper do
match("==")
match("!=")
match("<=")
match(">=")
match("<")
match(">")
end

rule :bool_expr do
match(:boolean) {
          |bool| BooleanNode.new(Dummy.new, bool, Dummy.new) }
match(:addition, :rel_oper, :multi) {
          |left, op, right| BooleanNode.new(left, op, right) }
end


rule :if_stmt do
match("if", "(", :bool_expr, ")", :stmt_list) {
          |_, _, bool, _, stmt| IfNode.new(bool, stmt) }
end

rule :else_stmt do
match("else", :stmt_list) {
          |_, stmt| ElseNode.new(stmt) }
end

rule :if_else_stmt do
match(:if_stmt, :terminator) {
          |stmt, _| IfElseNode.new(stmt, "") }
match(:if_stmt, :else_stmt, :terminator) {
```

```
          |if_stmt, else_stmt, _| IfElseNode.new(if_stmt,
else_stmt) }
end

rule :while_stmt do
match("while", "(", :bool_expr, ")", :stmt_list, :terminator) {
          |_, _, bool, _, stmt, _| WhileNode.new(bool, stmt) }
end

rule :call_stmt do
match("call", "(", /^[A-Z][a-zA-Z]+/, ")") {
          |_, _, routine, _| CallNode.new(routine) }
end

rule :wait_stmt do
match("wait", "(", :addition, ")") {
          |_, _, time, _| WaitNode.new(time)}
end

rule :identifier do
match(/^[a-z]+/) {
          |id| IdentifierNode.new(id) }
end

rule :assign_stmt do
match(/^[a-z]+/, "=", :addition) {
          |var, _, expr| AssignNode.new(var, expr) }
end

end
end

def done(str)
    ["quit","exit","bye",""].include?(str.chomp)
end

def run
print "[skywalker] "
str = gets
if done(str) then
puts "Bye."
else
      @@res = (@skywalker.parsestr)
puts "=> #{@@res.evaluate}"
      # @skywalker.parse(str)
run
end
```

```
end

defrunfile(filename)
code = File.read(filename)
    @@res = (@skywalker.parse code)
puts "=> #{@@res.evaluate}"
end

def compile(filename)
code = File.read(filename)
    @@res = (@skywalker.parse code)
external = File.read(@@external)
File.open("out.rb", 'w') {|f| f.write(external +
                                      "\n" +
                                      @@code.compile +
                                      "\nscheduler") }
end

def log(state = false)
if state
@skywalker.logger.level = Logger::DEBUG
else
@skywalker.logger.level = Logger::WARN
end
end
end
```

4.2 nodes.rb

Här är koden från filen ”nodes.rb

```
classAdditionNode
def initialize(op, left, right)
    @op, @left, @right = op, left, right
end

def compile
    "#{@left.compile} #{@op} #{@right.compile}"
end
end

classCallNode
def initialize(name)
    @name = name
end

def compile
    "Fiber.yield \"call #{@name}\""
end
end

classMultiNode
def initialize(op, left, right)
    @op, @left, @right = op, left, right
end

def compile
    "#{@left.compile} #{@op} #{@right.compile}"
end
end

classIntegerNode
def initialize(a)
    @value = a
end

def compile
    "#{@value}"
end
end

classParenthesesNode
def initialize(add)
    @add = add
```

```
end

def compile
    "(#{@add.compile})"
end
end

classFloatNode
def initialize(a)
    @value = a
end

def compile
    "#{@value}"
end
end

classAssignNode
def initialize(var, expr)
    @var = var
    @expr = expr
end

def compile
    "#{@var} = #{@expr.compile}"
end
end

classControlsAssignNode
def initialize(ctrl, expr)
    @ctrl = ctrl
    @expr = expr
end

def compile
    "@@control[\"#{@ctrl}\"] = #{@expr.compile}"
end
end

classIncludeNode
def initialize(ext)
    @@external = ext
end

def compile
    ""
end
```

```
91  end
92
93  classInterfaceVarNode
94  def initialize(var)
95      @var = var
96  end
97
98  def compile
99      "#{@var}"
100 end
101 end
102
103 classInterfaceVarListNode< Array
104 def initialize
105 end
106
107 def compile
108 self.each {|a| @@control[a] = 0 }
109 end
110 end
111
112 classRoutineNode
113 def initialize(name, stmt)
114     @name = name
115     @stmt = stmt
116 end
117
118 def compile
119     "#{@name} = Fiber.new do\nloop
120 do\n#{@stmt.compile}Fiber.yield :end\nend\nend"
121 end
122 end
123
124 classRoutineListNode< Array
125 def initialize
126 end
127
128 def compile
129     temp = ""
130 self.each {|a| temp += a.compile + "\n" }
131 temp
132 end
133 end
134
135
136
137 classIfNode
```

```
138 def initialize(bool, stmt)
139     @bool = bool
140     @stmt = stmt
141 end
142 
143 def compile
144     "if (#{@bool.compile})\n#{@stmt.compile}"
145 end
146 end
147 
148 classElseNode
149 def initialize(stmt)
150     @stmt = stmt
151 end
152 
153 def compile
154     "else\n#{@stmt.compile}"
155 end
156 end
157 
158 classIfElseNode
159 def initialize(if_,else_)
160     @if = if_
161     @else = else_
162 end
163 
164 def compile
165     "#{@if.compile}\n#{@else.compile}\nend"
166 end
167 end
168 
169 classWhileNode
170 def initialize(bool, stmt)
171     @bool = bool
172     @stmt = stmt
173 end
174 
175 def compile
176     "while(#{@bool.compile})\n#{@stmt.compile}\nend"
177 end
178 end
179 
180 classWaitNode
181 def initialize(time)
182     @time = time
183 end
184 
```

```
185 def compile
186     "Fiber.yield \"wait #{@time.compile}\""
187 end
188 end
189
190 classIdentifierNode
191 def initialize(id)
192     @id = id
193 end
194
195 def compile
196     "#{@id}"
197 end
198 end
199
200 classBooleanNode
201 def initialize(left, op, right)
202     @left = left
203     @op = op
204     @right = right
205 end
206
207 def compile
208     "#{@left.compile}#{@op}#{@right.compile}"
209 end
210 end
211
212 class Dummy
213 def initialize
214 nil
215 end
216
217 def compile
218     ""
219 end
220 end
221
222 classStmtNode
223 def initialize(stmt)
224     @stmt = stmt
225 end
226
227 def compile
228     "#{@stmt.compile}\n"
229 end
230 end
231
```

```
232 classStmtListNode< Array
233 def initialize
234 end
235
236 def compile
237 str = ""
238 self.each {|a| str += a.compile}
239 str
240   end
241 end
```