(\* CSC201, Section 1, TEAM 1,

John Spaugh,

Taro Kumagai,

Niravkumar Tandel,

Charles Thomas \*)

(\*---- Sec1Proj1Team1Step1 ----------------------\*)

(\* Abastract Grammar of a Graal-like Language in BNF for Project \*)

type Integer\_Constant = int;

type Boolean\_Constant = bool;

datatype Variable = S of string;

datatype Arithmatic\_Op = Plus | Minus | Times | Div;

datatype Relational\_Op = Lt | Le | Eq | Ne | Ge | Gt;

datatype Boolean\_Op = And | Or;

datatype Operator = AOp of Arithmatic\_Op | ROp of Relational\_Op | BOp of Boolean\_Op;

datatype Expression = Var of Variable |

IC of Integer\_Constant |

BC of Boolean\_Constant |

EEO of Expression \* Expression \* Operator;

datatype Instruction = Skip |

VE of (Variable \* Expression) |

IfThenElse of (Expression \* Instruction \* Instruction) |

WhileLoop of (Expression \* Instruction) |

Seq of Instruction list;

datatype Type = TypeBool | TypeInt ;

type Declaration = Variable \* Type;

type DeclarationList = Declaration list;

type Program = DeclarationList \* Instruction;

(\* Val Variable Declarations \*)

val var\_n = S "n"

val var\_cur = S "cur"

val var\_prev1 = S "prev1"

val var\_prev2 = S "prev2"

val var\_i = S "i"

val var\_answer = S "answer"

val var\_temp = S "temp"

val declaration\_n = (var\_n, TypeInt)

val declaration\_cur = (var\_cur, TypeInt)

val declaration\_prev1 = (var\_prev1, TypeInt)

val declaration\_prev2 = (var\_prev2, TypeInt)

val declaration\_i = (var\_i, TypeInt)

val declaration\_answer = (var\_answer, TypeInt)

(\* Full Declaration List \*)

val allDeclarations = [

declaration\_n,

declaration\_cur,

declaration\_prev1,

declaration\_prev2,

declaration\_i,

declaration\_answer

]

(\* Relational Logic \*)

val n\_equals\_0 = EEO (Var var\_n, IC 0, ROp Eq) (\* n == 0 \*)

val n\_equals\_1 = EEO (Var var\_n, IC 1, ROp Eq) (\* n == 1 \*)

val i\_lessthan\_n = EEO (Var var\_i, Var var\_n, ROp Lt) (\* i < n \*)

(\* Assignment Instructions, FORMAT assign\_variable\_value \*)

val assign\_n\_15 = VE (var\_n, IC 15) (\* n = 15 \*)

val assign\_answer\_1 = VE (var\_answer, IC 1) (\* answer = 1 \*)

val assign\_answer\_2 = VE (var\_answer, IC 2) (\* answer = 2 \*)

val assign\_i\_1 = VE (var\_i, IC 1) (\* i = 1 \*)

val assign\_prev1\_1 = VE (var\_prev1, IC 1) (\* prev1 = 1 \*)

val assign\_prev2\_2 = VE (var\_prev2, IC 2) (\* prev2 = 2 \*)

val assign\_answer\_cur = VE (var\_answer, Var var\_cur) (\* answer = cur \*)

val assign\_prev2\_prev1 = VE (var\_prev2, Var var\_prev1) (\* prev2 = prev1 \*)

val assign\_prev1\_cur = VE (var\_prev1, Var var\_cur) (\* prev1 = cur\*)

(\* Arithmatic Expressions \*)

val add\_i\_1 = EEO (Var var\_i, IC 1, AOp Plus) (\* i + 1 \*)

val add\_prev1\_prev2 = EEO (Var var\_prev1, Var var\_prev2, AOp Plus) (\* prev1 + prev2 \*)

(\* Arithmatic Instructions \*)

val ipp = VE (var\_i, add\_i\_1) (\* i++ \*)

val cur\_prev1\_plus\_prev2 = VE (var\_cur, add\_prev1\_prev2) (\* cur = prev1 + prev2 \*)

(\* Instructions \*)

val insideWhile = [

ipp,

cur\_prev1\_plus\_prev2,

assign\_prev2\_prev1,

assign\_prev1\_cur

]

val whileLoop = WhileLoop(i\_lessthan\_n, Seq insideWhile)

val inner\_Else = [

assign\_prev2\_2,

assign\_prev1\_1,

whileLoop,

assign\_answer\_cur

]

val inner\_ifThenElse = IfThenElse(n\_equals\_1, assign\_answer\_1, Seq inner\_Else)

val outer\_Else = [

assign\_i\_1,

inner\_ifThenElse

]

val outer\_ifThenElse = IfThenElse(n\_equals\_0, assign\_answer\_2, Seq outer\_Else)

val allInstructions = [assign\_n\_15, outer\_ifThenElse]

(\* The Program LUCAS\*)

val lucas = (allDeclarations, Seq allInstructions)

(\* -------------------

cpp file

#include <cstdlib>

#include <iomanip>

#include <iostream>

int main(){ //int argc, char \*argv[]){

int n;

int cur;

int prev1;

int prev2;

int i;

int answer;

n=15;

if(n == 0){

answer = 2;

}else{

i=1;

if(n ==1){

answer=1;

}else{

prev2 =2;

prev1 =1;

while (i < n){

i++; // i = i +1;

cur = prev1 + prev2;

prev2 = prev1;

prev1 = cur;

}

answer = cur;

}

}

std::cout << "Answer: " << answer << std::endl;

return 0;

}

---------------------------\*)

(\*------------step2 static semantics---------------\*)

(\* sml<Sec1Proj1Team1Step2.ml>Sec1Proj1Team1Step2Result.txt \*)

(\* 1 NoDuplicate: DeclarationList -> variable -> bool \*)

fun NoDuplicate((a: Variable, b: Type):: decListTail)(c: Variable) = (a <> c) andalso NoDuplicate decListTail c

| NoDuplicate [] c = true;

(\* 2 DecListVCheck: DeclarationList: DeclarationList -> bool \*)

val rec DecListVCheck =

(fn ( ((a: Variable, b: Type):: decListTail):DeclarationList) =>

DecListVCheck(decListTail) andalso

NoDuplicate(decListTail)(a) |

([]) => true

) ;

(\* Good testing step 2, input allDeclarations list with int into DecListVCheck

Gtest1 expected true \*)

val Gtest1 = DecListVCheck(allDeclarations);

(\* Bad testing step 2, New List using allDeclarations adding on a duplicate n to the head

Btest1 expected false \*)

val declarationslist\_bad = declaration\_n :: allDeclarations;

val Btest1 = DecListVCheck(declarationslist\_bad);

(\* 3-7 : DeclarationList -> AbsTypingTable \*)

(\* 3 \*)

datatype TypeValue = NoDeclaration | DeclaredInt | DeclaredBool ;

(\* 4 \*)

type AbsTypingTable = Variable -> TypeValue

(\* 5 AbsTypingTableNoDeclaration: variable -> NoDeclaration \*)

val AbsTypingTableNoDeclaration = (fn (x:Variable) => NoDeclaration)

(\* 6 NewAbsTypingTable: AbsTypingTable (old) -> Declaration -> AbsTypingTable (new) \*)

fun NewAbsTypingTable(oldatt: AbsTypingTable)(a:Variable, TypeBool)

=(fn (b:Variable)=> if b = a then DeclaredBool

else oldatt(b)) |

NewAbsTypingTable(oldatt: AbsTypingTable)(a:Variable, TypeInt)

=(fn(b:Variable)=> if b = a then DeclaredInt

else oldatt(b));

(\* Step 6 Testing

Had to add var bb to test for boolean since allDeclarations has none\*)

val myAbsTypingTable1 = NewAbsTypingTable(AbsTypingTableNoDeclaration)(declaration\_n);

myAbsTypingTable1 var\_n;

myAbsTypingTable1 var\_cur;

myAbsTypingTable1 var\_prev1;

myAbsTypingTable1 var\_prev2;

myAbsTypingTable1 var\_i;

myAbsTypingTable1 var\_answer;

val var\_bb = S "bb";

val declaration\_bb = (var\_bb, TypeBool);

val myAbsTypingTable2 = NewAbsTypingTable(myAbsTypingTable1)(declaration\_bb);

myAbsTypingTable2 var\_n;

myAbsTypingTable2 var\_cur;

myAbsTypingTable2 var\_prev1;

myAbsTypingTable2 var\_prev2;

myAbsTypingTable2 var\_i;

myAbsTypingTable2 var\_answer;

myAbsTypingTable2 var\_bb;

(\* 7 \*)

val rec wholeAbsTypingTable =

(fn ((decListhead:: decListTail):DeclarationList)=>

NewAbsTypingTable(wholeAbsTypingTable(decListTail) )(decListhead) |

([]) => AbsTypingTableNoDeclaration

);

(\*Step 7 Testing

Every var in allDecs should output its' type, any var not in allDecs should be no dec\*)

val myAbsTypingTable = wholeAbsTypingTable(allDeclarations);

myAbsTypingTable(var\_n);

myAbsTypingTable(var\_cur);

myAbsTypingTable(var\_prev1);

myAbsTypingTable(var\_prev2);

myAbsTypingTable(var\_i);

myAbsTypingTable(var\_answer);

myAbsTypingTable(var\_temp);

(\* 8 DetermineExpType: Expression -> AbsTypingTable -> TypeValue\*)

fun DetermineExpType(Var(x)) = (fn(y:AbsTypingTable) => y(x)) |

DetermineExpType(IC(x)) = (fn(y:AbsTypingTable) => DeclaredInt) |

DetermineExpType(BC(x)) = (fn(y:AbsTypingTable) => DeclaredBool) |

DetermineExpType(EEO(x1, x2, AOp(opa) ) ) =

(fn(y:AbsTypingTable) => DeclaredInt) |

DetermineExpType(EEO(x1, x2, ROp(opa) ) ) =

(fn(y:AbsTypingTable) => DeclaredBool) |

DetermineExpType(EEO(x1, x2, BOp(opa) ) ) =

(fn(y:AbsTypingTable) => DeclaredBool);

(\* 9 ExpressionVCheck: Expression -> AbsTypingTable -> Bool\*)

fun ExpressionVCheck(Var(a)) = (fn(b:AbsTypingTable) => b(a) <> NoDeclaration) |

ExpressionVCheck(IC(a)) = (fn(b:AbsTypingTable) => true) |

ExpressionVCheck(BC(a)) = (fn(b:AbsTypingTable) => true) |

ExpressionVCheck(EEO(a1, a2, AOp(opa))) =

(fn(b:AbsTypingTable) => ExpressionVCheck(a1)(b) andalso

(DetermineExpType(a1)(b) = DeclaredInt ) andalso

ExpressionVCheck(a2)(b) andalso

(DetermineExpType(a2)(b) = DeclaredInt)) |

ExpressionVCheck(EEO(a1, a2, ROp(opa))) =

(fn(b:AbsTypingTable) =>

ExpressionVCheck(a1)(b) andalso

(DetermineExpType(a1)(b) = DeclaredInt ) andalso

ExpressionVCheck(a2)(b) andalso

(DetermineExpType(a2)(b) = DeclaredInt)) |

ExpressionVCheck(EEO(a1, a2, BOp(opa))) =

(fn(b:AbsTypingTable) =>

ExpressionVCheck(a1)(b) andalso

(DetermineExpType(a1)(b) = DeclaredBool ) andalso

ExpressionVCheck(a2)(b) andalso

(DetermineExpType(a2)(b) = DeclaredBool));

(\*Test 9 Good in order\*)

val and\_t\_t = EEO (BC true, BC true, BOp And);

val nExpression = Var (var\_n);

val myExpressionVCheck = ExpressionVCheck(nExpression)(myAbsTypingTable);

val iCExpression = IC (0);

val myExpressionVCheck = ExpressionVCheck(iCExpression)(myAbsTypingTable);

val bCExpression = BC (false);

val myExpressionVCheck = ExpressionVCheck(bCExpression)(myAbsTypingTable);

val myExpressionVCheck = ExpressionVCheck(add\_i\_1)(myAbsTypingTable);

val myExpressionVCheck = ExpressionVCheck(i\_lessthan\_n)(myAbsTypingTable);

val myExpressionVCheck = ExpressionVCheck(and\_t\_t)(myAbsTypingTable);

(\*Test 9 Bad (5 - 6) - (8 + t)\*)

val minus\_5\_6 = EEO (IC 5, IC 6, AOp Minus);

val add\_8\_t = EEO (IC 8, BC true, AOp Plus);

val minus\_56\_8t = EEO (minus\_5\_6, add\_8\_t, AOp Minus);

val myExpressionVCheck = ExpressionVCheck(minus\_56\_8t)(myAbsTypingTable);

(\*10 InstructionVCheck: AbsTypingTable -> Instruction -> Bool\*)

val rec InstructionVCheck =

(fn (a:AbsTypingTable) =>

(fn (Skip) => true |

( VE(x, y)) =>

(a(x) = DetermineExpType(y)(a) ) andalso

(a(x) <> NoDeclaration) andalso

ExpressionVCheck(y)(a) |

( IfThenElse(x,y,z)) =>

(DetermineExpType(x)(a) = DeclaredBool) andalso

ExpressionVCheck(x)(a) andalso

InstructionVCheck(a)(y) andalso InstructionVCheck(a)(z) |

( WhileLoop(x,y)) =>

(DetermineExpType(x)(a) = DeclaredBool) andalso

ExpressionVCheck(x)(a) andalso

InstructionVCheck(a)(y) |

( Seq([])) => true |

( Seq(InstListHead :: InstListTail)) =>

InstructionVCheck(a)(InstListHead) andalso

InstructionVCheck(a)(Seq(InstListTail))

)

);

(\* \*\*\*\*testing part 10, 4-good cases \*\*\*\* \*)

val ItestGood\_1 = InstructionVCheck myAbsTypingTable cur\_prev1\_plus\_prev2

val ItestGood\_2 = InstructionVCheck myAbsTypingTable inner\_ifThenElse

val ItestGood\_3 = InstructionVCheck myAbsTypingTable whileLoop

val ItestGood\_4 = InstructionVCheck myAbsTypingTable (Seq inner\_Else)

(\* \*\*\*\*testing part 10, 1-bad case\*\*\*\* \*)

val badInstruction = VE (var\_n, BC true)

val ItestBad\_1 = InstructionVCheck myAbsTypingTable badInstruction

(\* 11 ProgramVCheck: Program -> Bool \*)

fun ProgramVCheck(a,b) =

DecListVCheck(a) andalso

InstructionVCheck(wholeAbsTypingTable(a))(b);

(\* \*\*\*\*testing part 11, 1-good case, apply to sample Program

if false then wrong in (function validity) or (program code)\*)

val test11\_good1 = ProgramVCheck(lucas);

(\* \*\*\*\*testing part 11, 1-bad case\*)

val bad\_program = (allDeclarations, badInstruction);

val test11\_bad1 = ProgramVCheck(bad\_program);

(\*------------End step2 static sementics---------------\*)

(\*------------Start step3 Dynamic sementics---------------\*)

(\* sml<Sec1Proj1Team1Step3.sml>Sec1Proj1Team1Step3Result.txt \*)

(\* 1 ValueInAbsProgState \*)

datatype ValueInAbsProgState = ValueUnknown | ValueInt of int | ValueBool of bool

(\* 2 AbsProgState: Variable -> ValueInAbsProgState\*)

type AbsProgState = Variable -> ValueInAbsProgState

(\* 3 AbsProgStateUnknown: Variable -> ValueInAbsProgState \*)

val AbsProgStateUnknown = (fn(x:Variable) => ValueUnknown)

(\* 4 NewAbsProgState: Variable x ValueInAbsProgState -> AbsProgState -> Variable -> ValueInAbsProgState \*)

fun NewAbsProgState (a:Variable, b:ValueInAbsProgState) (oldaps:AbsProgState) (c:Variable) =

if c = a then b else oldaps c

(\* Testing 1-4 \*)

val MyAbsProgState1 = NewAbsProgState (var\_n , ValueInt 15) AbsProgStateUnknown;

MyAbsProgState1 var\_n;

MyAbsProgState1 var\_cur;

MyAbsProgState1 var\_prev1;

MyAbsProgState1 var\_prev2;

MyAbsProgState1 var\_i;

MyAbsProgState1 var\_answer;

val MyAbsProgState2 = NewAbsProgState (var\_cur, ValueInt 0) MyAbsProgState1;

MyAbsProgState2 var\_n;

MyAbsProgState2 var\_cur;

MyAbsProgState2 var\_prev1;

MyAbsProgState2 var\_prev2;

MyAbsProgState2 var\_i;

MyAbsProgState2 var\_answer;

(\*\*\* 5 - 8 Exceptions \*\*\*)

exception WrongDivision

exception WrongOpForValueInt

exception WrongOpForValueBool

exception WrongExpression

(\* 9 ExpCalculation: ValueInAbsProgState x ValueInAbsProgState -> operator -> ValueInAbsProgState \*)

val ExpCalculation =

(fn(ValueInt v1, ValueInt v2) =>

(fn(AOp Plus) => ValueInt(v1 + v2)

| (AOp Minus) => ValueInt(v1 - v2)

| (AOp Times) => ValueInt(v1 \* v2)

| (AOp Div) => if v2 = 0

then raise WrongDivision

else ValueInt(v1 div v2)

| (ROp Gt) => ValueBool(v1 > v2)

| (ROp Lt) => ValueBool(v1 < v2)

| (ROp Eq) => ValueBool(v1 = v2)

| (ROp Ne) => ValueBool(v1 <> v2)

| (ROp Ge) => ValueBool(v1 >= v2)

| (ROp Le) => ValueBool(v1 <= v2)

| (\_) => raise WrongOpForValueInt

)

| (ValueBool v1, ValueBool v2) =>

(fn(BOp And) => ValueBool(v1 andalso v2)

| (BOp Or) => ValueBool(v1 orelse v2)

| (\_) => raise WrongOpForValueBool

)

| (\_, \_) => (fn(\_) => raise WrongExpression)

)

(\* 10 ExpressionValue: Expression -> AbsProgState -> ValueInAbsProgState \*)

fun ExpressionValue (Var x) (aps:AbsProgState) = aps(x)

| ExpressionValue (IC x) (aps:AbsProgState) = ValueInt(x)

| ExpressionValue (BC x) (aps:AbsProgState) = ValueBool(x)

| ExpressionValue (EEO (a, b, c)) (aps:AbsProgState) =

ExpCalculation (ExpressionValue a aps, ExpressionValue b aps) (c)

(\*\*\* Testing 5 - 10 \*\*\*)

(\* Good Base Cases \*)

val Good\_BaseCase1 = ExpressionValue (Var var\_n) MyAbsProgState2;

val Good\_BaseCase2 = ExpressionValue (IC 0) MyAbsProgState2;

val Good\_BaseCase3 = ExpressionValue (BC true) MyAbsProgState2;

(\* Good Binary Cases \*)

val Good\_BinaryCase1 = ExpCalculation (ValueInt 2, ValueInt 1) (AOp Div);

val Good\_BinaryCase2 = ExpCalculation (ValueInt 2, ValueInt 1) (ROp Ge);

val Good\_BinaryCase3 = ExpCalculation (ValueBool true, ValueBool false) (BOp And);

(\* BAD Case \*)

(\*val Bad\_WrongExpression = ExpCalculation (ValueInt 0, ValueBool true) (AOp Plus);\*)

(\* 11 MeaningInstruction: Intruction -> AbsProgState (Old) -> AbsProgState (New) \*)

val rec MeaningInstruction =

(fn(Skip) => (fn(aps:AbsProgState) => aps)

| (VE (a, b)) => (fn(aps:AbsProgState) => NewAbsProgState (a, ExpressionValue b aps) (aps))

| (IfThenElse (a, b, c)) => (fn(aps:AbsProgState) => if ExpressionValue a aps = ValueBool(true)

then MeaningInstruction b aps

else MeaningInstruction c aps)

| (WhileLoop (a, b)) => (fn(aps:AbsProgState) => if ExpressionValue a aps <> ValueBool(true)

then aps

else MeaningInstruction (WhileLoop (a, b))

(MeaningInstruction b aps))

| Seq [] => (fn(aps:AbsProgState) => aps)

| Seq (InstListHead::InstListTail) => (fn(aps:AbsProgState) => MeaningInstruction (Seq InstListTail)

(MeaningInstruction InstListHead aps))

)

(\*\*\* Testing 11 \*\*\*)

(\* Creating Full Initial ProgState \*)

val MyAbsProgStatePartial1 = NewAbsProgState (var\_n, ValueInt 0) AbsProgStateUnknown

val MyAbsProgStatePartial2 = NewAbsProgState (var\_cur, ValueInt 0) MyAbsProgStatePartial1

val MyAbsProgStatePartial3 = NewAbsProgState (var\_prev1, ValueInt 0) MyAbsProgStatePartial2

val MyAbsProgStatePartial4 = NewAbsProgState (var\_prev2, ValueInt 0) MyAbsProgStatePartial3

val MyAbsProgStatePartial5 = NewAbsProgState (var\_i, ValueInt 0) MyAbsProgStatePartial4

val MyAbsProgStateFull = NewAbsProgState (var\_answer, ValueInt 0) MyAbsProgStatePartial5

(\* 4 Good Cases \*)

val MeaningInstructionTest\_VE = MeaningInstruction assign\_n\_15 MyAbsProgStateFull

val MeaningInstructionTest\_IfThenElse = MeaningInstruction inner\_ifThenElse MyAbsProgStateFull

val MeaningInstructionTest\_WhileLoop = MeaningInstruction whileLoop MyAbsProgStateFull

val MeaningInstructionTest\_Seq = MeaningInstruction (Seq insideWhile) MyAbsProgStateFull

(\* ResultingProgState VE \*)

val ResultProgStateVE\_n = MeaningInstructionTest\_VE var\_n

val ResultProgStateVE\_cur = MeaningInstructionTest\_VE var\_cur

val ResultProgStateVE\_prev1 = MeaningInstructionTest\_VE var\_prev1

val ResultProgStateVE\_prev2 = MeaningInstructionTest\_VE var\_prev2

val ResultProgStateVE\_i = MeaningInstructionTest\_VE var\_i

val ResultProgStateVE\_answer = MeaningInstructionTest\_VE var\_answer

(\* ResultingProgState IfThenElse \*)

val ResultProgStateIfThenElse\_n = MeaningInstructionTest\_IfThenElse var\_n

val ResultProgStateIfThenElse\_cur = MeaningInstructionTest\_IfThenElse var\_cur

val ResultProgStateIfThenElse\_prev1 = MeaningInstructionTest\_IfThenElse var\_prev1

val ResultProgStateIfThenElse\_prev2 = MeaningInstructionTest\_IfThenElse var\_prev2

val ResultProgStateIfThenElse\_i = MeaningInstructionTest\_IfThenElse var\_i

val ResultProgStateIfThenElse\_answer = MeaningInstructionTest\_IfThenElse var\_answer

(\* ResultingProgState While \*)

val ResultProgStateWhile\_n = MeaningInstructionTest\_WhileLoop var\_n

val ResultProgStateWhile\_cur = MeaningInstructionTest\_WhileLoop var\_cur

val ResultProgStateeWhile\_prev1 = MeaningInstructionTest\_WhileLoop var\_prev1

val ResultProgStateWhile\_prev2 = MeaningInstructionTest\_WhileLoop var\_prev2

val ResultProgStateWhile\_i = MeaningInstructionTest\_WhileLoop var\_i

val ResultProgStateWhile\_answer = MeaningInstructionTest\_WhileLoop var\_answer

(\* ResultingProgState Seq \*)

val ResultProgStateSeq\_n = MeaningInstructionTest\_Seq var\_n

val ResultProgStateSeq\_cur = MeaningInstructionTest\_Seq var\_cur

val ResultProgStateSeq\_prev1 = MeaningInstructionTest\_Seq var\_prev1

val ResultProgStateSeq\_prev2 = MeaningInstructionTest\_Seq var\_prev2

val ResultProgStateSeq\_i = MeaningInstructionTest\_Seq var\_i

val ResultProgStateSeq\_answer = MeaningInstructionTest\_Seq var\_answer

(\* 12 Exception \*)

exception InvalidProgram

(\* 13 MeaningProgram: Program -> AbsProgState \*)

val MeaningProgram =

(fn((a,b): Program) => if ProgramVCheck(a,b)

then MeaningInstruction b AbsProgStateUnknown

else raise InvalidProgram

)

(\*\*\* Testing Program Lucas \*\*\*)

val lucasResultProgState = MeaningProgram lucas;

(\* ResultingProgState Lucas \*)

val ResultLucas\_n = lucasResultProgState var\_n

val ResultLucas\_var\_cur = lucasResultProgState var\_cur

val ResultLucas\_var\_prev1 = lucasResultProgState var\_prev1

val ResultLucas\_var\_prev2 = lucasResultProgState var\_prev2

val ResultLucas\_var\_i = lucasResultProgState var\_i

val ResultLucas\_var\_answer = lucasResultProgState var\_answer

val ResultLucas\_var\_temp = lucasResultProgState var\_temp

(\*------------End step3 Dynamic sementics---------------\*)

Standard ML of New Jersey [Version 110.99.8; 32-bit; April 25, 2025]

- type Integer\_Constant = int

type Boolean\_Constant = bool

datatype Variable = S of string

datatype Arithmatic\_Op = Div | Minus | Plus | Times

datatype Relational\_Op = Eq | Ge | Gt | Le | Lt | Ne

datatype Boolean\_Op = And | Or

datatype Operator

= AOp of Arithmatic\_Op

| BOp of Boolean\_Op

| ROp of Relational\_Op

datatype Expression

= BC of bool

| EEO of Expression \* Expression \* Operator

| IC of int

| Var of Variable

datatype Instruction

= IfThenElse of Expression \* Instruction \* Instruction

| Seq of Instruction list

| Skip

| VE of Variable \* Expression

| WhileLoop of Expression \* Instruction

datatype Type = TypeBool | TypeInt

type Declaration = Variable \* Type

type DeclarationList = Declaration list

type Program = DeclarationList \* Instruction

stdIn:173.73-173.75 Warning: calling polyEqual

val var\_n = S "n" : Variable

val var\_cur = S "cur" : Variable

val var\_prev1 = S "prev1" : Variable

val var\_prev2 = S "prev2" : Variable

val var\_i = S "i" : Variable

val var\_answer = S "answer" : Variable

val var\_temp = S "temp" : Variable

val declaration\_n = (S "n",TypeInt) : Variable \* Type

val declaration\_cur = (S "cur",TypeInt) : Variable \* Type

val declaration\_prev1 = (S "prev1",TypeInt) : Variable \* Type

val declaration\_prev2 = (S "prev2",TypeInt) : Variable \* Type

val declaration\_i = (S "i",TypeInt) : Variable \* Type

val declaration\_answer = (S "answer",TypeInt) : Variable \* Type

val allDeclarations =

[(S "n",TypeInt),(S "cur",TypeInt),(S "prev1",TypeInt),(S "prev2",TypeInt),

(S "i",TypeInt),(S "answer",TypeInt)] : (Variable \* Type) list

val n\_equals\_0 = EEO (Var (S "n"),IC 0,ROp Eq) : Expression

val n\_equals\_1 = EEO (Var (S "n"),IC 1,ROp Eq) : Expression

val i\_lessthan\_n = EEO (Var (S "i"),Var (S "n"),ROp Lt) : Expression

val assign\_n\_15 = VE (S "n",IC 15) : Instruction

val assign\_answer\_1 = VE (S "answer",IC 1) : Instruction

val assign\_answer\_2 = VE (S "answer",IC 2) : Instruction

val assign\_i\_1 = VE (S "i",IC 1) : Instruction

val assign\_prev1\_1 = VE (S "prev1",IC 1) : Instruction

val assign\_prev2\_2 = VE (S "prev2",IC 2) : Instruction

val assign\_answer\_cur = VE (S "answer",Var (S "cur")) : Instruction

val assign\_prev2\_prev1 = VE (S "prev2",Var (S "prev1")) : Instruction

val assign\_prev1\_cur = VE (S "prev1",Var (S "cur")) : Instruction

val add\_i\_1 = EEO (Var (S "i"),IC 1,AOp Plus) : Expression

val add\_prev1\_prev2 = EEO (Var (S "prev1"),Var (S "prev2"),AOp Plus)

: Expression

val ipp = VE (S "i",EEO (Var (S "i"),IC 1,AOp Plus)) : Instruction

val cur\_prev1\_plus\_prev2 =

VE (S "cur",EEO (Var (S "prev1"),Var (S "prev2"),AOp Plus)) : Instruction

val insideWhile =

[VE (S "i",EEO (Var (S "i"),IC 1,AOp Plus)),

VE (S "cur",EEO (Var (S "prev1"),Var (S "prev2"),AOp Plus)),

VE (S "prev2",Var (S "prev1")),VE (S "prev1",Var (S "cur"))]

: Instruction list

val whileLoop =

WhileLoop

(EEO (Var (S "i"),Var (S "n"),ROp Lt),

Seq

[VE (S "i",EEO (Var (S #),IC 1,AOp Plus)),

VE (S "cur",EEO (Var (S #),Var (S #),AOp Plus)),

VE (S "prev2",Var (S "prev1")),VE (S "prev1",Var (S "cur"))])

: Instruction

val inner\_Else =

[VE (S "prev2",IC 2),VE (S "prev1",IC 1),

WhileLoop

(EEO (Var (S "i"),Var (S "n"),ROp Lt),

Seq

[VE (S "i",EEO (Var #,IC #,AOp #)),

VE (S "cur",EEO (Var #,Var #,AOp #)),VE (S "prev2",Var (S "prev1")),

VE (S "prev1",Var (S "cur"))]),VE (S "answer",Var (S "cur"))]

: Instruction list

val inner\_ifThenElse =

IfThenElse

(EEO (Var (S "n"),IC 1,ROp Eq),VE (S "answer",IC 1),

Seq

[VE (S "prev2",IC 2),VE (S "prev1",IC 1),

WhileLoop

(EEO (Var (S #),Var (S #),ROp Lt),

Seq [VE (#,#),VE (#,#),VE (#,#),VE (#,#)]),

VE (S "answer",Var (S "cur"))]) : Instruction

val outer\_Else =

[VE (S "i",IC 1),

IfThenElse

(EEO (Var (S "n"),IC 1,ROp Eq),VE (S "answer",IC 1),

Seq

[VE (S "prev2",IC 2),VE (S "prev1",IC 1),

WhileLoop (EEO (Var #,Var #,ROp #),Seq [VE #,VE #,VE #,VE #]),

VE (S "answer",Var (S "cur"))])] : Instruction list

val outer\_ifThenElse =

IfThenElse

(EEO (Var (S "n"),IC 0,ROp Eq),VE (S "answer",IC 2),

Seq

[VE (S "i",IC 1),

IfThenElse

(EEO (Var (S #),IC 1,ROp Eq),VE (S "answer",IC 1),

Seq [VE (#,#),VE (#,#),WhileLoop (#,#),VE (#,#)])]) : Instruction

val allInstructions =

[VE (S "n",IC 15),

IfThenElse

(EEO (Var (S "n"),IC 0,ROp Eq),VE (S "answer",IC 2),

Seq

[VE (S "i",IC 1),

IfThenElse

(EEO (Var #,IC #,ROp #),VE (S #,IC #),

Seq [VE #,VE #,WhileLoop #,VE #])])] : Instruction list

val lucas =

([(S "n",TypeInt),(S "cur",TypeInt),(S "prev1",TypeInt),(S "prev2",TypeInt),

(S "i",TypeInt),(S "answer",TypeInt)],

Seq

[VE (S "n",IC 15),

IfThenElse

(EEO (Var (S "n"),IC 0,ROp Eq),VE (S "answer",IC 2),

Seq [VE (S #,IC #),IfThenElse (EEO #,VE #,Seq #)])])

: (Variable \* Type) list \* Instruction

val NoDuplicate = fn : (Variable \* Type) list -> Variable -> bool

val DecListVCheck = fn : DeclarationList -> bool

val Gtest1 = true : bool

val declarationslist\_bad =

[(S "n",TypeInt),(S "n",TypeInt),(S "cur",TypeInt),(S "prev1",TypeInt),

(S "prev2",TypeInt),(S "i",TypeInt),(S "answer",TypeInt)]

: (Variable \* Type) list

val Btest1 = false : bool

datatype TypeValue = DeclaredBool | DeclaredInt | NoDeclaration

stdIn:206.31 Warning: calling polyEqual

stdIn:209.30 Warning: calling polyEqual

type AbsTypingTable = Variable -> TypeValue

val AbsTypingTableNoDeclaration = fn : Variable -> TypeValue

val NewAbsTypingTable = fn

: AbsTypingTable -> Variable \* Type -> Variable -> TypeValue

val myAbsTypingTable1 = fn : Variable -> TypeValue

val it = DeclaredInt : TypeValue

val it = NoDeclaration : TypeValue

val it = NoDeclaration : TypeValue

val it = NoDeclaration : TypeValue

val it = NoDeclaration : TypeValue

val it = NoDeclaration : TypeValue

val var\_bb = S "bb" : Variable

val declaration\_bb = (S "bb",TypeBool) : Variable \* Type

val myAbsTypingTable2 = fn : Variable -> TypeValue

val it = DeclaredInt : TypeValue

val it = NoDeclaration : TypeValue

val it = NoDeclaration : TypeValue

val it = NoDeclaration : TypeValue

val it = NoDeclaration : TypeValue

val it = NoDeclaration : TypeValue

val it = DeclaredBool : TypeValue

val wholeAbsTypingTable = fn : DeclarationList -> AbsTypingTable

val myAbsTypingTable = fn : AbsTypingTable

val it = DeclaredInt : TypeValue

val it = DeclaredInt : TypeValue

val it = DeclaredInt : TypeValue

val it = DeclaredInt : TypeValue

val it = DeclaredInt : TypeValue

val it = DeclaredInt : TypeValue

val it = NoDeclaration : TypeValue

val DetermineExpType = fn : Expression -> AbsTypingTable -> TypeValue

val ExpressionVCheck = fn : Expression -> AbsTypingTable -> bool

val and\_t\_t = EEO (BC true,BC true,BOp And) : Expression

val nExpression = Var (S "n") : Expression

val myExpressionVCheck = true : bool

val iCExpression = IC 0 : Expression

val myExpressionVCheck = true : bool

val bCExpression = BC false : Expression

val myExpressionVCheck = true : bool

val myExpressionVCheck = true : bool

val myExpressionVCheck = true : bool

val myExpressionVCheck = true : bool

val minus\_5\_6 = EEO (IC 5,IC 6,AOp Minus) : Expression

val add\_8\_t = EEO (IC 8,BC true,AOp Plus) : Expression

val minus\_56\_8t =

EEO (EEO (IC 5,IC 6,AOp Minus),EEO (IC 8,BC true,AOp Plus),AOp Minus)

: Expression

val myExpressionVCheck = false : bool

val InstructionVCheck = fn : AbsTypingTable -> Instruction -> bool

val ItestGood\_1 = true : bool

val ItestGood\_2 = true : bool

val ItestGood\_3 = true : bool

val ItestGood\_4 = true : bool

val badInstruction = VE (S "n",BC true) : Instruction

val ItestBad\_1 = false : bool

val ProgramVCheck = fn : DeclarationList \* Instruction -> bool

val test11\_good1 = true : bool

val bad\_program =

([(S "n",TypeInt),(S "cur",TypeInt),(S "prev1",TypeInt),(S "prev2",TypeInt),

(S "i",TypeInt),(S "answer",TypeInt)],VE (S "n",BC true))

: (Variable \* Type) list \* Instruction

val test11\_bad1 = false : bool

stdIn:374.12 Warning: calling polyEqual

datatype ValueInAbsProgState

= ValueBool of bool

| ValueInt of int

| ValueUnknown

type AbsProgState = Variable -> ValueInAbsProgState

val AbsProgStateUnknown = fn : Variable -> ValueInAbsProgState

val NewAbsProgState = fn

: Variable \* ValueInAbsProgState

-> AbsProgState -> Variable -> ValueInAbsProgState

val MyAbsProgState1 = fn : Variable -> ValueInAbsProgState

val it = ValueInt 15 : ValueInAbsProgState

val it = ValueUnknown : ValueInAbsProgState

val it = ValueUnknown : ValueInAbsProgState

val it = ValueUnknown : ValueInAbsProgState

val it = ValueUnknown : ValueInAbsProgState

val it = ValueUnknown : ValueInAbsProgState

val MyAbsProgState2 = fn : Variable -> ValueInAbsProgState

val it = ValueInt 15 : ValueInAbsProgState

val it = ValueInt 0 : ValueInAbsProgState

val it = ValueUnknown : ValueInAbsProgState

val it = ValueUnknown : ValueInAbsProgState

val it = ValueUnknown : ValueInAbsProgState

val it = ValueUnknown : ValueInAbsProgState

exception WrongDivision

exception WrongOpForValueInt

exception WrongOpForValueBool

exception WrongExpression

val ExpCalculation = fn

: ValueInAbsProgState \* ValueInAbsProgState

-> Operator -> ValueInAbsProgState

val ExpressionValue = fn : Expression -> AbsProgState -> ValueInAbsProgState

val Good\_BaseCase1 = ValueInt 15 : ValueInAbsProgState

val Good\_BaseCase2 = ValueInt 0 : ValueInAbsProgState

val Good\_BaseCase3 = ValueBool true : ValueInAbsProgState

val Good\_BinaryCase1 = ValueInt 2 : ValueInAbsProgState

val Good\_BinaryCase2 = ValueBool true : ValueInAbsProgState

val Good\_BinaryCase3 = ValueBool false : ValueInAbsProgState

val MeaningInstruction = fn : Instruction -> AbsProgState -> AbsProgState

val MyAbsProgStatePartial1 = fn : Variable -> ValueInAbsProgState

val MyAbsProgStatePartial2 = fn : Variable -> ValueInAbsProgState

val MyAbsProgStatePartial3 = fn : Variable -> ValueInAbsProgState

val MyAbsProgStatePartial4 = fn : Variable -> ValueInAbsProgState

val MyAbsProgStatePartial5 = fn : Variable -> ValueInAbsProgState

val MyAbsProgStateFull = fn : Variable -> ValueInAbsProgState

val MeaningInstructionTest\_VE = fn : AbsProgState

val MeaningInstructionTest\_IfThenElse = fn : AbsProgState

val MeaningInstructionTest\_WhileLoop = fn : AbsProgState

val MeaningInstructionTest\_Seq = fn : AbsProgState

val ResultProgStateVE\_n = ValueInt 15 : ValueInAbsProgState

val ResultProgStateVE\_cur = ValueInt 0 : ValueInAbsProgState

val ResultProgStateVE\_prev1 = ValueInt 0 : ValueInAbsProgState

val ResultProgStateVE\_prev2 = ValueInt 0 : ValueInAbsProgState

val ResultProgStateVE\_i = ValueInt 0 : ValueInAbsProgState

val ResultProgStateVE\_answer = ValueInt 0 : ValueInAbsProgState

val ResultProgStateIfThenElse\_n = ValueInt 0 : ValueInAbsProgState

val ResultProgStateIfThenElse\_cur = ValueInt 0 : ValueInAbsProgState

val ResultProgStateIfThenElse\_prev1 = ValueInt 1 : ValueInAbsProgState

val ResultProgStateIfThenElse\_prev2 = ValueInt 2 : ValueInAbsProgState

val ResultProgStateIfThenElse\_i = ValueInt 0 : ValueInAbsProgState

val ResultProgStateIfThenElse\_answer = ValueInt 0 : ValueInAbsProgState

val ResultProgStateWhile\_n = ValueInt 0 : ValueInAbsProgState

val ResultProgStateWhile\_cur = ValueInt 0 : ValueInAbsProgState

val ResultProgStateeWhile\_prev1 = ValueInt 0 : ValueInAbsProgState

val ResultProgStateWhile\_prev2 = ValueInt 0 : ValueInAbsProgState

val ResultProgStateWhile\_i = ValueInt 0 : ValueInAbsProgState

val ResultProgStateWhile\_answer = ValueInt 0 : ValueInAbsProgState

val ResultProgStateSeq\_n = ValueInt 0 : ValueInAbsProgState

val ResultProgStateSeq\_cur = ValueInt 0 : ValueInAbsProgState

val ResultProgStateSeq\_prev1 = ValueInt 0 : ValueInAbsProgState

val ResultProgStateSeq\_prev2 = ValueInt 0 : ValueInAbsProgState

val ResultProgStateSeq\_i = ValueInt 1 : ValueInAbsProgState

val ResultProgStateSeq\_answer = ValueInt 0 : ValueInAbsProgState

exception InvalidProgram

val MeaningProgram = fn : Program -> AbsProgState

val lucasResultProgState = fn : AbsProgState

val ResultLucas\_n = ValueInt 15 : ValueInAbsProgState

val ResultLucas\_var\_cur = ValueInt 1364 : ValueInAbsProgState

val ResultLucas\_var\_prev1 = ValueInt 1364 : ValueInAbsProgState

val ResultLucas\_var\_prev2 = ValueInt 843 : ValueInAbsProgState

val ResultLucas\_var\_i = ValueInt 15 : ValueInAbsProgState

val ResultLucas\_var\_answer = ValueInt 1364 : ValueInAbsProgState

val ResultLucas\_var\_temp = ValueUnknown : ValueInAbsProgState