WinZig Complier

Implementation Report

CS4542 - Compiler Design

K.D.S.A. Chandrasiri
170081L

Table of Content

Introduction/Assumptions

Implemented all rules (complete WinZig language). Following are some assumptions made when implementing the language.

- Constants must be known in compile time.
  So they can either be char/integer literals or a constant that is defined earlier.
- All variables are initialized with 0.
  So for chars with chr(0), booleans with false and custom types with the first literal.
- In the output statement, a space will be added between items and will write new line at the end.
- Assigning to non-defined names is allowed, but the values will be discarded.
  Eg: d := func() is valid and it will simply run func() and discard the return value.

Program

E → <*program* <id:x> E E E E E <id:y>>

```
top ↓(2)    = 0
next ↓(2)   = 1
code ↓(2)   = Open
error ↓(2)  = if (x = y)
      then { Open }
      else { gen(Open, "Expected 'x' for the end token, but found 'y'.) }

newVarNames ↓(2)              = Empty List
newTypeLiteralNames ↓(2)      = Empty List
paramTypeSymbols ↓(2)         = Empty List
stringExpression ↓(2)         = ""
exprTypeSymbol ↓(2)           = UNDEFINED
activeFcnSymbol ↓(2)          = null
currentCaseVariableSymbol ↓(2)= null
nextCaseLabel ↓(2)            = null
isInLocalScope ↓(2)           = false
localSymbols ↓(2)             = Empty Table

code ↑(ε)   = close( gen(code ↑(6), "HALT"))
next ↑(ε)   = next ↑(6) + 1
error ↑(ε)  = close( error ↑(6) )
```

Consts

E → <*consts* E+>

~Defaults~

E → <*const* <id:x> <id:y>>

```
code ↑(ε)   = if (y is char literal)
            then { enterConstantSymbol(x, CHAR, y); code ↓(ε) }
            else if (y is integer literal)
            then { enterConstantSymbol(x, INTEGER, y); code ↓(ε) }
            else { yt = lookupConstant(y);
                   enterConstantSymbol(x, yt.typeSymbol, yt.value); code ↓(ε) }
error ↑(ε)  = if (definedInScope(x))
            then { gen( error ↓(ε), “Variable ‘x' is already defined”) }
            else if (y is not char/integer literal
                  && lookupConstant(y) == UNDEFINED)
            then { gen(error ↓(ε), “Constant 'y' is not defined”) }
            else { error ↓(ε) }
```

Types

E → <*types* E*>

```
~Defaults~
```

E → <*type* <id:x> E>

```
newTypeLiteralNames ↓(2)      = Empty List

code ↑(ε)                     = ts = enterTypeSymbol(x);
            for (i = 0 to len(newTypeLiteralNames ↑(2)) - 1)
            { enterConstantSymbol(newTypeLiteralNames ↑(2)[i], ts, i) };
            code ↑(2)
```

E → <*lit* <id:x>+>

```
newTypeLiteralNames ↑(ε)      = tln = newTypeLiteralNames ↓(ε);
                        for (each xi in x) { addItem(tln, xi) };
                        tln
```

SubProgs

E → <*subprogs* E*>

```
// Here N = len(E) - number of subprogs
// Here i ∈ [1, len(E)]
code ↓(1)   = if (N != 0)
            then { gen(code ↓(ε), “GOTO”, label(next ↑(N))) }
            else { code ↓(ε) }
next ↓(1)   = if (N != 0)
            then { next ↓(ε) + 1 }
            else { next ↓(ε) }
top ↓(i)    = 0
top ↑(ε)    = top ↓(ε)
```

E → <*fcn* <id:x> E <id:y> E E E E <id:z>>

```
error ↓(1)              = if (x = z)
                        then { error ↓(ε) }
                        else { gen(error ↓(ε) , "Expected ’x’ for the end
                        token, but found ’z’.) }
isInLocalScope ↓(1)     = true
localSymbols ↓(1)       = Empty Table
paramTypeSymbols ↓(1)   = Empty List
activeFcnSymbol ↓(7)    = enterFcnSymbol(x, label(next ↓(ε)),
                        paramTypeSymbols ↑(6), lookupType(y))
error ↓(7)              = if (lookupType(y) == UNDEFINED)
                  then { gen(error ↑(6), “Type 'y' is not defined”) }
                  else { error ↑(6) }
activeFcnSymbol ↑(ε)    = null
code ↑(ε)               = gen( gen(code ↑(7), “LIT 0”), “RTN 1”)
next ↑(ε)               = next ↑(7) + 2
isInLocalScope ↑(ε)     = false
localSymbols ↑(ε)       = Empty Table
```

E → <*params* E+>

```
newVarNames ↓(1)              = Empty List
newTypeLiteralNames ↑(ε)      = pts = paramTypeSymbols ↑(N);
                              for (each newVar in newVarNames  ↑(N))
                              { addItem(pts, newVar.typeSymbol) }; pts
```

DclIn

E → <*dclns* E+>

```
newVarNames ↓(1)  = Empty List
code ↑(ε)         = c = code ↑(N);
                  for (each newVar in newVarNames  ↑(N))
                  { c = gen(c, ”LIT 0”) };
                  c
next ↑(ε)         = next ↑(N) + len(newVarNames  ↑(N))
```

E → <*var* <id:xi>+ <id:y>>

```
error ↑(ε)        = if (lookupType(y) == UNDEFINED)
            then { gen(error ↓(ε), “Type 'y' is not defined”) }
            else if (for any xi: alreadyDefinedInScope(xi))
            then { gen(error ↓(ε), “Variable 'xi' already defined”) }
            else { error ↓(ε) }
code ↑(ε)         = for (i = 0 to len(xi) - 1)
            { enterVariableSymbol(xi[i], next ↓(ε) + i, lookupType(y)) };
            code ↑(ε)
next ↑(ε)         = next ↑(N) + len(xi)
newVarNames ↑(ε)  = nvn = newVarNames ↓(ε);
            for (each xi in xi nodes) { addItem(nvn, xi) };
            nvn
```

Statements

E → *<block* E+>

~Defaults~

E → *<output* E+>

```
// Here i ∈ [2, len(E)]
code ↓(i)   = c = code ↑(i-1)
            if (exprTypeSymbol ↑(i-1) == INTEGER)
            then { gen(gen(gen(c, “SOS OUTPUT”), “LIT 32”), “OUTPUTC”) }
            else if (exprTypeSymbol ↑(i-1) == CHAR)
            then { gen(gen(gen(c, “SOS OUTPUTC”), “LIT 32”), “OUTPUTC”) }
            else { for (each ch in stringExpression ↑(i-1))
                  { c = gen(gen(c, “LIT ch”), “OUTPUTC”) };
                  gen(gen(c, “LIT 32”), “OUTPUTC”) }
top ↓(i)    = if (exprTypeSymbol ↑(i-1) == INTEGER)
            then { top ↑(i-1) - 1 }
            else if (exprTypeSymbol ↑(i-1) == CHAR)
            then { top ↑(i-1) - 1 }
            else { top ↑(i-1) }
next ↓(i)   = if (exprTypeSymbol ↑(i-1) == INTEGER)
            then { next ↑(i-1) + 3 }
            else if (exprTypeSymbol ↑(i-1) == CHAR)
            then { next ↑(i-1) + 3 }
            else { next ↑(i-1) + 2*len(stringExpression ↑(i-1)) + 2 }

code ↑(ε)   = c = code ↑(i-1)
            if (exprTypeSymbol ↑(N) == INTEGER)
            then { gen(gen(c, “SOS OUTPUT”), “OUTPUTL”) }
            else if (exprTypeSymbol ↑(N) == CHAR)
            then {gen(gen(c, “SOS OUTPUTC”), “OUTPUTL”) }
            else { for (each ch in stringExpression ↑(i-1))
                  { c = gen(gen(c, “LIT ch”), “OUTPUTC”) };
                  gen(c, “OUTPUTL”) }

top ↑(ε)    = if (exprTypeSymbol ↑(N) == INTEGER)
            then { top ↑(N) - 1 }
            else if (exprTypeSymbol ↑(N) == CHAR)
            then { top ↑(N) - 1 }
```

```
            else { top ↑(N) }
next ↑(ε)   = if (exprTypeSymbol ↑(N) == INTEGER)
            then { next ↑(N) + 2 }
            else if (exprTypeSymbol ↑(N) == CHAR)
            then { next ↑(N) + 2 }
            else { next ↑(N) + 2*len(stringExpression ↑(i-1)) + 1 }
```

E → <*if* E E E?>

```
error ↓(2)  = if (exprTypeSymbol ↑(1) != BOOLEAN)
            then { gen(error ↑(1), "Invalid type for if condition") }
            else { error ↑(1) }
code ↓(2)   = gen(code ↑(1), "COND label(next ↑(1)) label(next ↑(2))")
next ↓(2)   = next ↑(1) + 1
top ↓(2)    = top ↑(1) - 1

code ↓(3)   = if (has else clause)
            then { gen(code ↑(2), "GOTO label(next ↑(ε))") }
            else { code ↑(2) }
next ↓(3)   = if (has else clause)
            then { next ↑(2) + 1 }
            else { next ↑(2) }
```

E → <*while* E E>

```
error ↓(2)  = if (exprTypeSymbol ↑(1) != BOOLEAN)
            then { gen(error ↑(1), "Invalid type for while condition") }
            else { error ↑(1) }
code ↓(2)   = gen(code ↑(1), "COND label(next ↑(1)) label(next ↑(ε))")
next ↓(2)   = next ↑(1) + 1
top ↓(2)    = top ↑(1) - 1

code ↑(ε)   = gen(code ↑(2), "GOTO label(next ↓(ε))")
next ↑(ε)   = next ↑(2) + 1
```

E → <*repeat* E+ E>

```
// Here N = len(E) - the number of statements in repeat block
error ↑(ε)  = if (exprTypeSymbol ↑(N+1) != BOOLEAN)
            then { gen(error ↑(N+1), "Invalid type for repeat condition") }
            else { error ↑(N+1) }
code ↑(ε)   = gen(code ↑(N+1), "COND label(next ↑(ε)) label(next ↓(ε))")
```

```
next ↑(ε)   = next ↑(N+1) + 1
top ↑(ε)    = top ↑(N+1) - 1
```

E → <*for* E E E E>

```
error ↓(3)  = if (exprTypeSymbol ↑(2) != BOOLEAN)
            then { gen(error ↑(2), "Invalid type for loop condition") }
            else { error ↑(2) }
code ↓(3)   = gen(code ↑(2), "COND label(next ↑(2) + 1) label(next ↑(ε))")
next ↓(3)   = next ↑(2) + 1
top ↓(3)    = top ↑(2) - 1

// Note: visit Statement (4) and then ForStat (3)
// So the defaults for those 2 children will be set according to that order

code ↑(ε)   = gen(code ↑(4), "GOTO label(next ↑(1))")
next ↑(ε)   = next ↑(4) + 1
```

E → <*loop* E+>

```
// Here N = len(E) - the number of statements in loop block
code ↑(ε)   = gen(code ↑(N), "GOTO label(next ↑(N) + 1)")
next ↑(ε)   = next ↑(N) + 1
```

E → <*case* E CaseClause+ E?>

```
// Here N = len(CaseClause) - the number of case clauses in case block
// Here M = len(E) - the number of all children
// Here i ∈ [3, M]

currentCaseVariableSymbol ↓(2) = new temp local Variable named "~generated~"
nextCaseLabel ↓(i-1) = nextCaseLabel ↑(i-2)
code ↓(i) = gen(code ↓(i-1), "GOTO next ↑(ε)")
next ↓(i)   = next ↓(i-1) + 1

code ↑(ε)   = gen(code ↑(M), "POP 1")
next ↑(ε)   = next ↑(M) + 1
top ↑(ε)    = top ↑(M) - 1
currentCaseVariableSymbol ↑(ε) = currentCaseVariableSymbol ↓(ε)
```

E → <*read* <id:x>+>

```
// Here i ∈ [2, len(E)]
// Here N = len(E) - number of all children
error ↓(i)  = if (lookupVariable(x[i-1]) == UNDEFINED)
            then { gen(error ↑(i-1), “Variable not defined”) }
            else if (type(lookupVariable(x[i-1])) ∉ {CHAR, INTEGER})
            then { gen(error ↑(i-1), “Invalid type for read statement”) }
            else { error ↑(i-1) }
code ↓(i)   = if (lookupVariable(x[i-1]) == UNDEFINED)
            then { code ↑(i-1) }
            else if (type(lookupVariable(x[i-1])) == INTEGER})
            then {
                  if (isGlobal(lookupVariable(x[i-1]))
                  then { gen(gen(code ↑(i-1), “SOS INPUT”), “SGV
                  address(lookupVariable(x[i-1])”) }
                  else { gen(gen(code ↑(i-1), “SOS INPUT”), “SLV
                  address(lookupVariable(x[i-1])”) }
            }
            else if (type(lookupVariable(x[i-1])) == CHAR})
            then {
                  if (isGlobal(lookupVariable(x[i-1]))
                  then { gen(gen(code ↑(i-1), “SOS INPUTC”), “SGV
                  address(lookupVariable(x[i-1])”) }
                  else { gen(gen(code ↑(i-1), “SOS INPUTC”), “SLV
                  address(lookupVariable(x[i-1])”) }
            }
            else { code ↑(i-1) }
next ↓(i)   = if (lookupVariable(x[i-1]) == UNDEFINED)
            then { next ↑(i-1) }
            else if (type(lookupVariable(x[i-1])) ∈ {CHAR, INTEGER})
            then { next ↑(i-1) + 2 }
            else { next ↑(i-1) }

error ↑(ε)  = if (lookupVariable(x[N]) == UNDEFINED)
            then { gen(error ↑(N), “Variable not defined”) }
            else if (type(lookupVariable(x[N])) ∉ {CHAR, INTEGER})
            then { gen(error ↑(N), “Invalid type for read statement”) }
            else { error ↑(N) }
code ↑(ε)   = if (lookupVariable(x[N]) == UNDEFINED)
            then { code ↑(N) }
            else if (type(lookupVariable(x[N])) == INTEGER})
            then {
```

```
                  if (isGlobal(lookupVariable(x[N]))
                  then { gen(gen(code ↑(N), “SOS INPUT”), “SGV
                  address(lookupVariable(x[N])”) }
                  else { gen(gen(code ↑(N), “SOS INPUT”), “SLV
                  address(lookupVariable(x[N])”) }
            }
            else if (type(lookupVariable(x[N])) == CHAR})
            then {
                  if (isGlobal(lookupVariable(x[N]))
                  then { gen(gen(code ↑(N), “SOS INPUTC”), “SGV
                  address(lookupVariable(x[N])”) }
                  else { gen(gen(code ↑(N), “SOS INPUTC”), “SLV
                  address(lookupVariable(x[N])”) }
            }
            else { code ↑(N) }
next ↑(ε)   = if (lookupVariable(x[N]) == UNDEFINED)
            then { next ↑(N) }
            else if (type(lookupVariable(x[N])) ∈ {CHAR, INTEGER})
            then { next ↑(N) + 2 }
            else { next ↑(N) }
```

E → <exit>

```
code ↑(ε)   = gen(code ↑(N), “HALT“)
next ↑(ε)   = next ↑(N) + 1
```

E → <return E>

```
error ↑(ε)  = if (activeFcnSymbol ↑(1) == NULL)
            then { gen(error ↑(1), “Return statement outside of function”) }
            else if (typeMismatch (returnTypeSymbol ↑(1), exprTypeSymbol ↑(1))
            then { gen(error ↑(1), “Type mismatch”) }
            else { error ↑(1) }
code ↑(ε)   = gen(code ↑(1), “RTN 1“)
next ↑(ε)   = next ↑(1) + 1
top ↑(ε)    = top ↑(1) - 1
```

E → <null>

~Defaults~

E → <integer E>

~Defaults~

E → <string <string:x>>

```
stringExpression ↑(ε)   = x
exprTypeSymbol ↑(ε)     = STRING
```

E → <*case_clause* CaseExpressions+ E>

```
// Here i ∈ [2, len(CaseExpressions)]
// Here N = len(CaseExpressions)
code ↓(i)   = if (CaseExpressions[i-1] is Identifier)
            then { gen(gen(gen(code ↓(i-1),
                  “LLV currentCaseVariableSymbol ↓(ε)”),
                  “LIT getConstantValue(CaseExpressions[i-1])”),
                  “BOP BEQ”) }
            else { code ↑(i-1) }
next ↓(i)   = if (CaseExpressions[i-1] is Identifier)
            then { next ↓(i-1) + 3 }
            else { next ↑(i-1) }
top ↓(i)    = if (CaseExpressions[i-1] is Identifier)
            then { top ↓(i-1) - 1 }
            else { top ↑(i-1) }

code ↓(N+1) = if (CaseExpressions[N] is Identifier)
            then {
                  c = gen(gen(gen(code ↓(N),
                  “LLV currentCaseVariableSymbol ↓(ε)”),
                  “LIT getConstantValue(CaseExpressions[N])”),
                  “BOP BEQ”);
                  for (each caseExpr in CaseExpressions) {
                        c = gen(c, “BOP BOR”)
                  }
                  gen(c, “COND next ↓(N+1) nextCaseLabel ↓(ε)”)
            }
            else { code ↑(N) }
next ↓(N+1) = if (CaseExpressions[i-1] is Identifier)
            then { next ↓(i-1) + 4 + N }
            else { next ↑(i-1) }
top ↓(N+1)  = if (CaseExpressions[i-1] is Identifier)
            then { top ↓(i-1) - 2 - N }
```

```
            else { top ↑(i-1) }
```

E → <.. <id:x> <id:y>>

```
error ↑(ε)  = if (typeMismatch (currentCaseVariableSymbol ↓(ε), x)
            then { gen(error ↓(ε), “Type mismatch”) }
            else if (typeMismatch (currentCaseVariableSymbol ↓(ε), y)
            then { gen(error ↓(ε), “Type mismatch”) }
            else if (value(x) > value(y))
            then { gen(error ↑(1), “Case value range is not in order”) }
            else { error ↑(1) }
code ↑(ε)   = gen(gen(gen(gen(gen(gen(code ↑(1),
                  “LIT value(x)“),
                  “LLV address(currentCaseVariableSymbol ↓(ε))“),
                  “BOP BLE“),
                  “LLV address(currentCaseVariableSymbol ↓(ε))“),
                  “LIT value(y)“),
                  “BOP BLE“),
                  “BOP BAND“)
next ↑(ε)   = next ↑(1) + 7
top ↑(ε)    = top ↑(1) - 1
```

E → <*otherwise* E>

```
~Defaults~
```

E → <*assign* <id:x> E>

```
error ↑(ε)  = if (lookup(x) == UNDEFINED)
            then { error ↑(1) }
            else if (lookup(x) is not Variable)
            then {  gen(error ↑(1), “Expected a variable”) }
            else if (typeMismatch(type(lookup(x)), exprTypeSymbol ↑(1)))
            then {  gen(error ↑(1), “Type mismatch”) }
            else { error ↑(1) }
code ↑(ε)   = if (lookup(x) == UNDEFINED)
            then { gen(code ↑(1), “POP 1”) }
            else {
                  if (isGlobal(lookupVariable(x[N]))
                  then { gen(code ↑(1), “SGV address(lookupVariable(x[N])”) }
```

```
                  else { gen(code ↑(1), “SLV address(lookupVariable(x[N])”) }
            }
next ↑(ε)   = next ↑(1) + 1
top ↑(ε)    = top ↑(1) - 1
```

E → <*swap* <id:x> <id:y>>

```
error ↑(ε)  = if (lookupVariable(x) == UNDEFINED)
            then { gen(error ↓(ε), “Variable not defined”) }
            else if (lookupVariable(y) == UNDEFINED)
            then { gen(error ↓(ε), “Variable not defined”) }
            else if (typeMismatch(lookupVariable(x), lookupVariable(y)))
            then {  gen(error ↓(ε), “Type mismatch”) }
            else { error ↓(ε) }
code ↑(ε)   = if (lookupVariable(x) != UNDEFINED
                        && lookupVariable(y) != UNDEFINED)
            then {
                  if (isGlobal(lookupVariable(x)))
                  then { gen(code ↓(ε), “LGV address(lookupVariable(x))”) }
                  else { gen(code ↓(ε), “LLV address(lookupVariable(x))”) }
                  if (isGlobal(lookupVariable(y)))
                  then { gen(code ↓(ε), “LGV address(lookupVariable(y))”) }
                  else { gen(code ↓(ε), “LLV address(lookupVariable(y))”) }
                  if (isGlobal(lookupVariable(x)))
                  then { gen(code ↓(ε), “SGV address(lookupVariable(x))”) }
                  else { gen(code ↓(ε), “SLV address(lookupVariable(x))”) }
                  if (isGlobal(lookupVariable(y)))
                  then { gen(code ↓(ε), “SGV address(lookupVariable(y))”) }
                  else { gen(code ↓(ε), “SLV address(lookupVariable(y))”) }
            }
            else { code ↓(ε) }
next ↑(ε)   = if (lookupVariable(x) != UNDEFINED
                        && lookupVariable(y) != UNDEFINED)
            then { next ↓(ε) + 4 }
            else { next ↓(ε) }
```

E → <*true*>

```
code ↑(ε)               = gen(code ↓(ε), “LIT 1“)
next ↑(ε)               = next ↓(ε) + 1
exprTypeSymbol ↑(ε)     = BOOLEAN_TYPE
top ↑(ε)                = top ↑(1) + 1
```

Expressions

E → <<= E E>

```
error ↑(ε)  = if (typeMismatch (exprTypeSymbol ↑(1), exprTypeSymbol ↑(2)))
            then { gen(error ↑(2), “Type mismatch”) }
            else { error ↑(2) }
code ↑(ε)               = gen(code ↑(2), “BOP BLE“)
next ↑(ε)               = next ↑(2) + 1
exprTypeSymbol ↑(ε)     = BOOLEAN_TYPE
top ↑(ε)                = top ↑(2) - 1
```

E → << E E>

```
error ↑(ε)  = if (typeMismatch (exprTypeSymbol ↑(1), exprTypeSymbol ↑(2)))
            then { gen(error ↑(2), “Type mismatch”) }
            else { error ↑(2) }
code ↑(ε)               = gen(code ↑(2), “BOP BLT“)
next ↑(ε)               = next ↑(2) + 1
exprTypeSymbol ↑(ε)     = BOOLEAN_TYPE
top ↑(ε)                = top ↑(2) - 1
```

E → <>= E E>

```
error ↑(ε)  = if (typeMismatch (exprTypeSymbol ↑(1), exprTypeSymbol ↑(2)))
            then { gen(error ↑(2), “Type mismatch”) }
            else { error ↑(2) }
code ↑(ε)               = gen(code ↑(2), “BOP BGE“)
next ↑(ε)               = next ↑(2) + 1
exprTypeSymbol ↑(ε)     = BOOLEAN_TYPE
top ↑(ε)                = top ↑(2) - 1
```

E → <> E E>

```
error ↑(ε)  = if (typeMismatch (exprTypeSymbol ↑(1), exprTypeSymbol ↑(2)))
            then { gen(error ↑(2), “Type mismatch”) }
            else { error ↑(2) }
code ↑(ε)               = gen(code ↑(2), “BOP BGT“)
next ↑(ε)               = next ↑(2) + 1
exprTypeSymbol ↑(ε)     = BOOLEAN_TYPE
top ↑(ε)                = top ↑(2) - 1
```

E → <= E E>

```
error ↑(ε)  = if (typeMismatch (exprTypeSymbol ↑(1), exprTypeSymbol ↑(2)))
            then { gen(error ↑(2), “Type mismatch”) }
            else { error ↑(2) }
code ↑(ε)               = gen(code ↑(2), “BOP BEQ“)
next ↑(ε)               = next ↑(2) + 1
exprTypeSymbol ↑(ε)     = BOOLEAN_TYPE
top ↑(ε)                = top ↑(2) - 1
```

E → <<> E E>

```
error ↑(ε)  = if (typeMismatch (exprTypeSymbol ↑(1), exprTypeSymbol ↑(2)))
            then { gen(error ↑(2), “Type mismatch”) }
            else { error ↑(2) }
code ↑(ε)               = gen(code ↑(2), “BOP BNE“)
next ↑(ε)               = next ↑(2) + 1
exprTypeSymbol ↑(ε)     = BOOLEAN_TYPE
top ↑(ε)                = top ↑(2) - 1
```

E → <+ E E>

```
error ↑(ε)  = if (typeMismatch (exprTypeSymbol ↑(1), exprTypeSymbol ↑(2)))
            then { gen(error ↑(2), “Type mismatch”) }
            else { error ↑(2) }
code ↑(ε)               = gen(code ↑(2), “BOP BPLUS“)
next ↑(ε)               = next ↑(2) + 1
exprTypeSymbol ↑(ε)     = exprTypeSymbol ↑(1)
top ↑(ε)                = top ↑(2) - 1
```

E → <- E E>

```
error ↑(ε)  = if (typeMismatch (exprTypeSymbol ↑(1), exprTypeSymbol ↑(2)))
            then { gen(error ↑(2), “Type mismatch”) }
            else { error ↑(2) }
code ↑(ε)               = gen(code ↑(2), “BOP BMINUS“)
next ↑(ε)               = next ↑(2) + 1
exprTypeSymbol ↑(ε)     = exprTypeSymbol ↑(1)
top ↑(ε)                = top ↑(2) - 1
```

E → <* E E>

```
error ↑(ε)  = if (typeMismatch (exprTypeSymbol ↑(1), exprTypeSymbol ↑(2)))
            then { gen(error ↑(2), “Type mismatch”) }
            else { error ↑(2) }
code ↑(ε)               = gen(code ↑(2), “BOP BMULT“)
next ↑(ε)               = next ↑(2) + 1
exprTypeSymbol ↑(ε)     = exprTypeSymbol ↑(1)
top ↑(ε)                = top ↑(2) - 1
```

E → </ E E>

```
error ↑(ε)  = if (typeMismatch (exprTypeSymbol ↑(1), exprTypeSymbol ↑(2)))
            then { gen(error ↑(2), “Type mismatch”) }
            else { error ↑(2) }
code ↑(ε)               = gen(code ↑(2), “BOP BDIV“)
next ↑(ε)               = next ↑(2) + 1
exprTypeSymbol ↑(ε)     = exprTypeSymbol ↑(1)
top ↑(ε)                = top ↑(2) - 1
```

E → <*mod* E E>

```
error ↑(ε)  = if (typeMismatch (exprTypeSymbol ↑(1), exprTypeSymbol ↑(2)))
            then { gen(error ↑(2), “Type mismatch”) }
            else { error ↑(2) }
code ↑(ε)               = gen(code ↑(2), “BOP BMOD“)
next ↑(ε)               = next ↑(2) + 1
exprTypeSymbol ↑(ε)     = exprTypeSymbol ↑(1)
top ↑(ε)                = top ↑(2) - 1
```

E → <*or* E E>

```
error ↑(ε)  = if (typeMismatch (exprTypeSymbol ↑(1), exprTypeSymbol ↑(2)))
            then { gen(error ↑(2), “Type mismatch”) }
            else { error ↑(2) }
code ↑(ε)               = gen(code ↑(2), “BOP BOR“)
next ↑(ε)               = next ↑(2) + 1
exprTypeSymbol ↑(ε)     = BOOLEAN
top ↑(ε)                = top ↑(2) - 1
```

E → <*and* E E>

```
error ↑(ε)  = if (typeMismatch (exprTypeSymbol ↑(1), exprTypeSymbol ↑(2)))
            then { gen(error ↑(2), “Type mismatch”) }
            else { error ↑(2) }
code ↑(ε)               = gen(code ↑(2), “BOP BAND“)
next ↑(ε)               = next ↑(2) + 1
exprTypeSymbol ↑(ε)     = BOOLEAN
top ↑(ε)                = top ↑(2) - 1
```

E → <- E>

```
error ↑(ε)  = if (exprTypeSymbol ↑(1) != INTEGER)
            then { gen(error ↑(1), “Negative is only defined for integer”) }
            else { error ↑(1) }
code ↑(ε)               = gen(code ↑(1), “UOP UNEG“)
next ↑(ε)               = next ↑(1) + 1
exprTypeSymbol ↑(ε)     = INTEGER
```

E → <not E>

```
error ↑(ε)  = if (exprTypeSymbol ↑(1) != BOOLEAN)
            then { gen(error ↑(1), “Not is only defined for boolean”) }
            else { error ↑(1) }
code ↑(ε)               = gen(code ↑(1), “UOP UNOT“)
next ↑(ε)               = next ↑(1) + 1
exprTypeSymbol ↑(ε)     = BOOLEAN
```

E → <eof>

```
code ↑(ε)               = gen(code ↑(1), “SOS EOF“)
next ↑(ε)               = next ↑(1) + 1
exprTypeSymbol ↑(ε)     = BOOLEAN
top ↑(ε)                = top ↑(1) + 1
```

E → <call <id:x> E+>

```
// Here i ∈ [2, len(E)]
// Here N = len(E) Number of all parameters

error ↓(1)  = if (lookupFunction(x) == UNDEFINED)
            then { gen(error ↓(ε), “Function not defined”) }
            else if (len(params(lookupFunction(x))) != len(E))
            then { gen(error ↓(ε), “Function number of params mismatch”) }
```

```
            else { error ↓(ε) }
code ↓(1)   = gen(code ↓(ε), “LIT 0”)
next ↓(1)   = next ↓(ε) + 1
top ↓(1)    = top ↓(ε) + 1


error ↓(i)  = if (!isFunctionAssignable(y, i-1, exprTypeSymbol  ↑(i-1)))
            then { gen(error ↑(i-1), “Parameter i-1 invalid for function x”) }
            else { error ↑(i-1) }

error ↑(ε)  = if (!isFunctionAssignable(y, N, exprTypeSymbol  ↑(N)))
            then { gen(error ↑(N), “Parameter N is invalid for function x”) }
            else { error ↑(N) }

code ↑(ε)   = gen(
                  gen(code ↑(N), “CODE label(lookupFunction(x))”),
            “CALL top ↓(ε) + 1”)
next ↑(ε)   = next ↑(N) + 2
top ↑(ε)    = top ↓(ε) + 1
exprTypeSymbol ↑(ε)     = returnType(lookupFunction(x))
```

E → <*succ* E>

```
error ↑(ε)  = if (exprTypeSymbol ↑(1) == BOOLEAN)
            then { gen(error ↑(1), “Succ is not defined for boolean”) }
            else { error ↑(1) }
code ↑(ε)               = gen(code ↑(1), “UOP USUCC“)
next ↑(ε)               = next ↑(1) + 1
```

E → <*pred* E>

```
error ↑(ε)  = if (exprTypeSymbol ↑(1) == BOOLEAN)
            then { gen(error ↑(1), “Pred is not defined for boolean”) }
            else { error ↑(1) }
code ↑(ε)               = gen(code ↑(1), “UOP UPRED“)
next ↑(ε)               = next ↑(1) + 1
```

E → <*chr* E>

```
error ↑(ε)  = if (exprTypeSymbol ↑(1) != INTEGER)
            then { gen(error ↑(1), “Chr is only defined for integer”) }
            else { error ↑(1) }
```

```
exprTypeSymbol ↑(ε)     = CHAR
```

E → *<ord* E>

```
error ↑(ε)  = if (exprTypeSymbol ↑(1) != CHAR)
            then { gen(error ↑(1), “Ord is only defined for char”) }
            else { error ↑(1) }
exprTypeSymbol ↑(ε)     = INTEGER
```

Generated Code

Example 1: Echo-print the first ten numbers on the input

```
program copy:
{ Echo-prints the first ten numbers on the input }
var count, f: integer;
begin
count := 1;
 while (count <= 10) do
    begin
       read(f);
       output(f);
       count := count + 1
    end
end copy.
```

```
         LIT 0
         LIT 0
         LIT 1
         SGV 0
L0       LGV 0
         LIT 10
         BOP BLE
         COND L1 L2
L1       SOS INPUT
         SGV 1
         LGV 1
         SOS OUTPUT
         SOS OUTPUTL
         LGV 0
         LIT 1
         BOP BPLUS
         SGV 0
         GOTO L0
L2       HALT
```

Example 2: Program to compute the factors of entered numbers

```
{
       This is a program to compute the factors of entered
numbers.
       It tests:
              procedures
              repeat loop
              if statement
              arithmetic
}
program factors:

var
    i : integer;

function Factor ( i : integer ):integer;
var
    j : integer;
begin
    if i > 0 then
       for (j := 1; j <= i; j:=j+1)
           if i mod j = 0 then output ( j )
end Factor;

begin
    repeat
       read(i);
       d:=Factor ( i )
    until i <= 0
end factors.
```

```
        LIT 0
        GOTO L9
L1      LIT 0
        LLV 0
        LIT 0
        BOP BGT
        COND L2 L3
L2      LIT 1
        SLV 1
L4      LLV 1
        LLV 0
        BOP BLE
        COND L5 L6
L5      LLV 0
        LLV 1
        BOP BMOD
        LIT 0
        BOP BEQ
        COND L7 L8
L7      LLV 1
        SOS OUTPUT
        SOS OUTPUTL
        GOTO L8
L8      LLV 1
        LIT 1
        BOP BPLUS
        SLV 1
        GOTO L4
L6      GOTO L3
L3      LIT 0
        RTN 1
L9      SOS INPUT
        SGV 0
        LIT 0
        LGV 0
        CODE L1
        CALL 2
        POP 1
        LGV 0
        LIT 0
        BOP BLE
        COND L10 L9
L10     HALT
```

Example 3: Program to test a series of numbers if prime

```
{
       This is a program to test a series of numbers entered
to see if
       they are prime.  It ouputs 1 for primes and 0 for
composites.
       It tests:
              functions
              while loop
              case statement
              repeat loop
              if statement
              arithmetic
}
program TestPrimes:

var n:integer;

function IsPrime ( n : integer ) : boolean;
var
    i : integer;
    Prime : boolean;
begin

    Prime := (n=2) or (n mod 2 = 1);  # either 2 or an odd
number

    i := 3;
    while Prime and (i*i <= n) do
        if n mod i = 0 then Prime := false
                       else i := i + 2;
    return (Prime)

end IsPrime;

begin

    repeat
       read(n);
       case IsPrime ( n ) of
           true:  output(1);
           false: output(0);
       end
    until eof

end TestPrimes.
```

```
         LIT 0
         GOTO L8
L1       LIT 0
         LIT 0
         LLV 0
         LIT 2
         BOP BEQ
         LLV 0
         LIT 2
         BOP BMOD
         LIT 1
         BOP BEQ
         BOP BOR
         SLV 2
         LIT 3
         SLV 1
L2       LLV 2
         LLV 1
         LLV 1
         BOP BMULT
         LLV 0
         BOP BLE
         BOP BAND
         COND L3 L4
L3       LLV 0
         LLV 1
         BOP BMOD
         LIT 0
         BOP BEQ
         COND L5 L6
L5       LIT 0
         SLV 2
         GOTO L7
L6       LLV 1
         LIT 2
         BOP BPLUS
         SLV 1
L7       GOTO L2
L4       LLV 2
         RTN 1
         LIT 0
         RTN 1
L8       SOS INPUT
         SGV 0
         LIT 0
         LGV 0
         CODE L1
         CALL 2
L11      LLV 1
         LIT 1
         BOP BEQ
         COND L13 L12
L13      LIT 1
         SOS OUTPUT
         SOS OUTPUTL
         GOTO L10
L12      LLV 1
```

	LIT 0 BOP BEQ COND L14 L10 L14 LIT 0 SOS OUTPUT SOS OUTPUTL GOTO L10 LLV 1 LIT 0 BOP BEQ COND L15 L10 L15 LIT 0 SOS OUTPUT SOS OUTPUTL L10 POP 1 SOS EOF COND L9 L8 L9 HALT

Example 4: Program to calculate factorial

```
program factorial:
var m,n:integer;

function fact (n:integer):integer;
begin
   m := m + 1;
   if n>0 then
        return (n * fact (n-1))
   else return (1)
end fact;

begin
     m := 0;
     read(n);
     output (  fact(n) , m)

end factorial.
```

```
         LIT 0
         LIT 0
         GOTO L0
L1       LGV 0
         LIT 1
         BOP BPLUS
         SGV 0
         LLV 0
         LIT 0
         BOP BGT
         COND L2 L3
L2       LLV 0
         LIT 0
         LLV 0
         LIT 1
         BOP BMINUS
         CODE L1
         CALL 3
         BOP BMULT
         RTN 1
         GOTO L4
L3       LIT 1
         RTN 1
L4       LIT 0
         RTN 1
L0       LIT 0
         SGV 0
         SOS INPUT
         SGV 1
         LIT 0
         LGV 1
         CODE L1
         CALL 3
         SOS OUTPUT
         LIT 32
         SOS OUTPUTC
         LGV 0
         SOS OUTPUT
         SOS OUTPUTL
         HALT
```

Example 5: Program to calculate fibinacci

```
program Fibonacci:
var i:integer;

function fibonacci(n:integer):integer;
begin
   if n=0 then return (0)
   else if n = 1 then return (1)
   else return (fibonacci ( n-1 ) + fibonacci ( n-2 ))
end fibonacci;

begin
for (i:=1; i<=7; i:=i+1)
begin
  output (fibonacci ( i ))
  end
end Fibonacci.
```

```
        LIT 0
        GOTO L0
L1      LLV 0
        LIT 0
        BOP BEQ
        COND L2 L3
L2      LIT 0
        RTN 1
        GOTO L4
L3      LLV 0
        LIT 1
        BOP BEQ
        COND L5 L6
L5      LIT 1
        RTN 1
        GOTO L4
L6      LIT 0
        LLV 0
        LIT 1
        BOP BMINUS
        CODE L1
        CALL 2
        LIT 0
        LLV 0
        LIT 2
        BOP BMINUS
        CODE L1
        CALL 3
        BOP BPLUS
        RTN 1
L4      LIT 0
        RTN 1
L0      LIT 1
        SGV 0
L8      LGV 0
        LIT 7
        BOP BLE
        COND L9 L10
L9      LIT 0
        LGV 0
        CODE L1
        CALL 2
        SOS OUTPUT
        SOS OUTPUTL
        LGV 0
        LIT 1
        BOP BPLUS
        SGV 0
        GOTO L8
L10     HALT
```

Example 6: Simple calculator program

```
program cow:

# A Simple Calculator Program

var i : integer;
    c : char;
    flag : boolean;

function GetNext(d:integer):integer;
begin
    if (flag = true) then
        read(i)
    else
       read(c);
    flag := not(flag);
end GetNext;

function P(d:integer) : integer;
var v: integer;
begin
    v := i;
    d:=GetNext(3);
    return (v);
end P;

function T(d:integer) : integer;
var v: integer;
begin
   v := P(3);
   while ((c = '*') or (c = '/')) do begin
        if (c = '*') then begin
            d:=GetNext(3);
           v := v * P(3);
        end
        else begin { c = '/' }
            d:=GetNext(3);
            v := v / P(3)
       end;
   end;
   return (v);
end T;

function E(d:integer) : integer;
var v: integer;
begin
   v := T(3);
   while ((c = '+') or (c = '-')) do begin
        if (c = '+') then begin
            d:=GetNext(3);
           v := v + T(3);
        end
        else begin { c = '-' }
            d:=GetNext(3);
            v := v - T(3)
       end;
   end;
   return (v);
```

```
        LIT 0
        LIT 0
        LIT 0
        GOTO L0
L1      LGV 2
        LIT 1
        BOP BEQ
        COND L2 L3
L2      SOS INPUT
        SGV 0
        GOTO L4
L3      SOS INPUTC
        SGV 1
L4      LGV 2
        UOP UNOT
        SGV 2
        LIT 0
        RTN 1
L5      LIT 0
        LGV 0
        SLV 1
        LIT 0
        LIT 3
        CODE L1
        CALL 3
        SLV 0
        LLV 1
        RTN 1
        LIT 0
        RTN 1
L6      LIT 0
        LIT 0
        LIT 3
        CODE L5
        CALL 3
        SLV 1
L7      LGV 1
        LIT 42
        BOP BEQ
        LGV 1
        LIT 47
        BOP BEQ
        BOP BOR
        COND L8 L9
L8      LGV 1
        LIT 42
        BOP BEQ
        COND L10 L11
L10     LIT 0
        LIT 3
        CODE L1
        CALL 3
        SLV 0
        LLV 1
        LIT 0
        LIT 3
        CODE L5
```

```
end E;


begin
    flag := true;
    d:=GetNext(3);
    output(E(3));
end cow.
```

```
          CALL 4
          BOP BMULT
          SLV 1
          GOTO L12
L11       LIT 0
          LIT 3
          CODE L1
          CALL 3
          SLV 0
          LLV 1
          LIT 0
          LIT 3
          CODE L5
          CALL 4
          BOP BDIV
          SLV 1
L12       GOTO L7
L9        LLV 1
          RTN 1
          LIT 0
          RTN 1
L13       LIT 0
          LIT 0
          LIT 3
          CODE L6
          CALL 3
          SLV 1
L14       LGV 1
          LIT 43
          BOP BEQ
          LGV 1
          LIT 45
          BOP BEQ
          BOP BOR
          COND L15 L16
L15       LGV 1
          LIT 43
          BOP BEQ
          COND L17 L18
L17       LIT 0
          LIT 3
          CODE L1
          CALL 3
          SLV 0
          LLV 1
          LIT 0
          LIT 3
          CODE L6
          CALL 4
          BOP BPLUS
          SLV 1
          GOTO L19
L18       LIT 0
          LIT 3
          CODE L1
          CALL 3
          SLV 0
          LLV 1
          LIT 0
          LIT 3
```

	CODE L6 CALL 4 BOP BMINUS SLV 1 L19 GOTO L14 L16 LLV 1 RTN 1 LIT 0 RTN 1 L0 LIT 1 SGV 2 LIT 0 LIT 3 CODE L1 CALL 4 POP 1 LIT 0 LIT 3 CODE L13 CALL 4 SOS OUTPUT SOS OUTPUTL HALT

Abstract Machine Instruction Set

Note: Abstract machine instruction set used is the same as the original WinZig specification. None of the instructions were changed. The instructions included below are same as the original instructions.

Instructions

```
NOP     :                                  # Do nothing.
HALT    : halt                             # Stop.
LIT  v  : Push v on Lf                     # Literal v.
LLV  i  : Push (Lf i) on Lf                # Load Local Value i.
LGV  i  : Push (Gf i) on Lf                # Load Global Value i.
SLV  i  : Lf i <- Pop Lf                   # Store Local Value i.
SGV  i  : Gf i <- Pop Lf                   # Store Global Value i.
LLA  i  : Push (Local_Address i) on Lf     # Load Local Address i.
LGA  i  : Push (Global_Address i) on Lf    # Load Global Address i.
UOP  i  : const X = Pop Lf                 # Unary Operation i.
          Push (Unop(i,X)) on Lf
BOP  i  : const Xr,Xl = Pop Lf, Pop Lf     # Binary Operation i.
          Push (Binop(i,Xl,Xr)) on Lf
POP  n  : Pop n off Lf                     # Pop n values.
DUP     : Push (Top Lf ) on Lf             # DUPlicate top of stack.
SWAP    : const One,Two = Pop Lf, Pop Lf   # SWAP top two values.
          Push One on Lf
        Push Two on Lf
CALL n  : Push I on Return_Stack           # Save current instruction.
          I <- Pop Lf                      # Entry point.
          Open_Frame n                     # Bump LBR (see below).
          repeat                           # back to top of loop.
RTN  n  : const Start = Depth Lf - n       # Return top n values
          if Start > 0 then                #    to caller
             for j=0 to n-1 do             # Move values to bottom of
               LF j <- Lf (Start+j)        #    frame.
             Pop Start off Lf              # Get rid of the rest.
          fi
          I <- Pop Return_Stack            # Branch to return address.
          Close_Frame i                    # Un-bump (De-bump?) LBR
            where I is CALL i              #    (see below).
GOTO L  : I <- L                           # branch to L
          repeat                           # Back to top of loop.



COND L M: I <- if Pop Lf = True            # Pop Stack.  If value is:
```

```
                    then L                  #   true,  go to L
                    else M                  #   false, go to M.
                fi
            repeat                          # Back to top of loop.
      CODE F  : Push F on Lf                # Push entry point.
      SOS i   : Operating_System i          # May change Lf.
```

Unary/Binary Operators

```
    Unop(i,X) means
    case i of
          UNOT   : not(X)
          UNEG   : -(X)
          USUCC  : X+1
          UPRED  : X-1
    endcase

    Binop(i,Xl,Xr) means
    case i of
          BAND   : Xl and Xr
          BOR    : Xl or  Xr
          BPLUS  : Xl +   Xr
          BMINUS : Xl -   Xr
          BMULT  : Xl *   Xr
          BDIV   : Xl div Xr
          BMOD   : Xl mod Xr
          BEQ    : Xl =   Xr
          BNE    : Xl <>  Xr
          BLE    : Xl <=  Xr
          BGE    : Xl >=  Xr
          BLT    : Xl <   Xr
          BGT    : Xl >   Xr
    endcase
```

Operating System Operators

```
    Operating_System i means
    case i of
          TRACEX : Trace_Execution <- not TraceExecution
          DUMPMEM: Dump_Memory
          INPUT  : readln(i)
                   Push i on Lf
```

```
      INPUTC : readln(ch)
               Push Ord(ch) on Lf
      OUTPUT : write (Pop Lf)
      OUTPUTC: write (Chr(Pop(Lf)))
      OUTPUTL: writeln
      EOF    : if eof(input)
                 then Push True  on Lf
                 else Push False on Lf
endcase
```