Final Project Documentation - Group 17

COMP SCI 4TB3/6TB3, McMaster University

Author: Rickesh Mistry, Kenny Kim, Ryan Ticlo, April 2019

This collection of *jupyter notebooks* develops a modified compiler for P0, a subset of Pascal. The compiler is intended to generate WASM code. The objective of our project was to extend the P0 compiler by adding switch statements, for loops, and foreach loops. The new grammar for the P0 compiler follows:

The P0 Grammar

```
selector ::= {"." ident | "[" expression "]"}.
factor ::= ident selector | integer | "(" expression ")" | "not" factor.
term ::= factor {("*" | "div" | "mod" | "and") factor}.
simpleExpression ::= ["+" | "-"] term {("+" | "-" | "or") term}.
expression ::= simpleExpression
    {("=" | "<>" | "<" | "<=" | ">" | ">=") simpleExpression}.
compoundStatement = "begin" statement {";" statement} "end"
statement ::=
    ident selector ":=" expression |
    ident "(" [expression {"," expression}] ")" |
    compoundStatement |
    "if" expression "then" statement ["else"statement] |
    "while" expression "do" statement |
    "case" expression "of" case {";" case} [elsePart] [";"] "end" |
    "for" controlVariable ":=" initialValue ("to"|"downto") finalValue "do"
 statement |
    "for" controlVariable "in" "[" constList "]" "do" statement
case :: = constList ":" statement
elsePart ::= ("else"|"otherwise") statementlist
constList ::= expression {“,” expression}
statementlist ::= statement {";" statement}
controlVariable ::= ident
initialValue ::= expression
finalValue ::= expression
type ::=
    ident |
    "array" "[" expression ".." expression "]" "of" type |
    "record" typedIds {";" typedIds} "end".
typedIds ::= ident {"," ident} ":" type.
declarations ::=
    {"const" ident "=" expression ";"}
    {"type" ident "=" type ";"}
    {"var" typedIds ";"}
    {"procedure" ident ["(" [["var"] typedIds {";" ["var"] typedIds}] ")"]
 ";"
        declarations compoundStatement ";"}.
program ::= "program" ident ";" declarations compoundStatement.
```

Modifications to SC

The `FOR`, `IN`, `TO`, `DOWNTO`, `CASE`, and `OTHERWISE` symbols were added to the scanner to allow the modified compiler to recognize the new commands.

In [ ]:
```
TIMES = 1; DIV = 2; MOD = 3; AND = 4; PLUS = 5; MINUS = 6
OR = 7; EQ = 8; NE = 9; LT = 10; GT = 11; LE = 12; GE = 13
PERIOD = 14; COMMA = 15; COLON = 16; RPAREN = 17; RBRAK = 18
OF = 19; THEN = 20; DO = 21; LPAREN = 22; LBRAK = 23; NOT = 24
BECOMES = 25; NUMBER = 26; IDENT = 27; SEMICOLON = 28
END = 29; ELSE = 30; IF = 31; WHILE = 32; ARRAY = 33
RECORD = 34; CONST = 35; TYPE = 36; VAR = 37; PROCEDURE = 38
BEGIN = 39; PROGRAM = 40; EOF = 41; TILDE = 42; AMP = 43; BAR = 44
FOR = 45; IN = 46; TO = 47; DOWNTO = 48; CASE = 49; OTHERWISE = 50;
```

In [ ]:
```
KEYWORDS = \
    {'div': DIV, 'mod': MOD, 'and': AND, 'or': OR, 'of': OF, 'then': THE
N,
    'do': DO, 'not': NOT, 'end': END, 'else': ELSE, 'if': IF, 'while': W
HILE,
    'array': ARRAY, 'record': RECORD, 'const': CONST, 'type': TYPE,
    'var': VAR, 'procedure': PROCEDURE, 'begin': BEGIN, 'program': PROGR
AM, 'tilde':TILDE,
    'bar':BAR,'amp':AMP, 'for' : FOR, 'in' : IN, 'to': TO, 'downto' : DO
WNTO, 'case' : CASE,
    'otherwise' : OTHERWISE}
```

Modifications to P0

The new symbols were imported into P0.

In [ ]:
```
import nbimporter
nbimporter.options["only_defs"] = False
import SC  #  used for SC.init, SC.sym, SC.val, SC.error
from SC import TIMES, DIV, MOD, AND, PLUS, MINUS, OR, EQ, NE, LT, GT, \
    LE, GE, PERIOD, COMMA, COLON, RPAREN, RBRAK, OF, THEN, DO, LPAREN, \
    LBRAK, NOT, BECOMES, NUMBER, IDENT, SEMICOLON, END, ELSE, IF, WHILE,
\
    ARRAY, RECORD, CONST, TYPE, VAR, PROCEDURE, BEGIN, PROGRAM, EOF, \
    getSym, mark, TILDE, AMP, BAR, FOR, IN, TO, DOWNTO, CASE, OTHERWISE
import ST  #  used for ST.init
from ST import Var, Ref, Const, Type, Proc, StdProc, Int, Bool, Enum, \
    Record, Array, newDecl, find, openScope, topScope, closeScope, print
SymTab
```

The new symbols were added for recursive decent parsing.

The `IN` symbol was added to `FOLLOWFACTOR`.

The `FOR` and `CASE` symbols were added to `FIRSTSTATEMENT`.

The `Else`, `IN`, and `BECOMES` symbols were added to `FOLLOWSTATEMENT`.

```
In [ ]: FIRSTFACTOR = {IDENT, NUMBER, LPAREN, NOT, TILDE}
        FOLLOWFACTOR = {TIMES, DIV, MOD, AND, OR, PLUS, MINUS, EQ, NE, LT, LE, G
        T, GE,
                        COMMA, SEMICOLON, THEN, ELSE, RPAREN, RBRAK, DO, PERIOD,
        END, AMP, BAR, IN}
        FIRSTEXPRESSION = {PLUS, MINUS, IDENT, NUMBER, LPAREN, NOT, TILDE}
        FIRSTSTATEMENT = {IDENT, IF, WHILE, BEGIN, FOR, CASE}
        FOLLOWSTATEMENT = {SEMICOLON, END, ELSE, IN, BECOMES}
        FIRSTTYPE = {IDENT, RECORD, ARRAY, LPAREN}
        FOLLOWTYPE = {SEMICOLON}
        FIRSTDECL = {CONST, TYPE, VAR, PROCEDURE}
        FOLLOWDECL = {BEGIN}
        FOLLOWPROCCALL = {SEMICOLON, END, ELSE}
        STRONGSYMS = {CONST, TYPE, VAR, PROCEDURE, WHILE, IF, BEGIN, EOF}
```

Procedure `controlVariable()` parses

```
controlVariable ::= ident.
```

This is used in `for` and `for each` loop statements.

```
In [ ]: def controlVariable():
            if SC.sym == IDENT:
                x = find(SC.val);
                x = CG.genVar(x)
            else:
                mark('Ident expected!!')
            return x
```

Procedure `constList()` parses

```
constList ::= expression {“,” expression}
```

This is used in `for each` loop statements.

```
In [ ]: def constList():
            #empty list to add stuff
            xs = []
            #expression returns Var(Int)!! hopefully
            x = expression()
            #append it to the list
            xs.append(x)
            #while there are more elements in the list
            while SC.sym == COMMA:
                if SC.sym == COMMA: getSym()
                else: mark(", missing")
                #append it to the list
                y = expression()
                xs.append(y)
            #create Type(Array) with parameters (self, base, lower, length):
            #set lower to 0 since we are gonna access it starting from x[0]...
            x = Type(CG.genArray(Array(xs[0].tp, 0, len(xs))))
            #print(x)
            return x, xs
```

Procedure `initialValue()` parses

```
initialValue ::= expression.
```

This is used in `for` loop statements.

```
In [ ]: def initialValue():
            x = expression()
            return x
```

Procedure `finalValue()` parses

```
finalValue ::= expression.
```

This is used in `for` loop statements.

```
In [ ]: def finalValue():
            x = expression()
            return x
```

Procedure `case(x, counter_name, else_name)` parses

```
case :: = constList ":" statement.
```

This is used for `case` statements.

The parameter `counter_name` is used to initialize the counter for arrays in each `case` statement.

The parameter `else_name` is used to track if no cases have run and the elsePart should run.

In [ ]:
```
def case(x, counter_name, else_name):
    global array_num
    y, inputList = constList()
    #inorder to push it to the stack; get the ST.Array
    array_tp = y.val;
    #array_name starting from for_array_0
    array_name = "for_array_"+str(array_num)
    #declare it, and will create global variable in genForArray()
    newDecl(array_name, Var(array_tp))
    #increment array number
    array_num += 1
    #call genForArray with name of the array, user input array
    CG.genCaseArrayLoopInit(x, array_name, inputList, counter_name, else
_name)
    if SC.sym == COLON:
        getSym()
        a = statement()
        CG.genCaseArrayLoopEnd(counter_name)
    else:
        mark("colon (:) expected from case function")
```

Procedure `elsePart (else_name)` parses

```
elsePart ::= ("else"|"otherwise") statementlist.
```

This is used in `case` statements.

In [ ]:
```
def elsePart(else_name):
    if (SC.sym == ELSE or SC.sym == OTHERWISE):
        getSym()
        CG.genCaseElseInit(else_name)
        x = statementList()
        CG.genCaseElseEnd()
    else:
        mark("else or otherwise expected from elsePart function")
```

Procedure `statementList()` parses

```
statementlist ::= statement {";" statement}.
```

This is used in `elsePart` for `case` statements.

```
In [ ]: def statementList():
            xs = []
            x = statement()
            xs.append(x)
            while SC.sym == SEMICOLON:
                if SC.sym == SEMICOLON:
                    getSym()
                y = statement()
                if y == None:
                    break
                xs.append(y)
            return xs
```

Procedure `statement()` was modified to include `if\elif` clauses which parse

```
"case" expression "of" case {";" case} [elsePart] [";"] "end" |
"for" controlVariable ":=" initialValue ("to"|"downto") finalValue "do" stat
ement |
"for" controlVariable "in" "[" constList "]" "do" statement.
```

The `elif` clause which checks `SC.sym == FOR` is used for both `for` and `for each` loop statements.

The `elif` clause which checks `SC.sym == CASE` is used for `case` statements.

```
In [ ]: def statement():
            global array_num
            if SC.sym == END:
                return None
            if SC.sym not in FIRSTSTATEMENT:
                mark("statement expected"); getSym()
                while SC.sym not in FIRSTSTATEMENT | FOLLOWSTATEMENT | STRONGSYM
        S : getSym()
            if SC.sym == IDENT:
                #ORIGINAL CODE
                #...

            #MODIFIED CODE
            elif SC.sym == FOR:
                getSym();
                #x = ident
                x = controlVariable()
                getSym()
                ##for controlVariable "in"
                if SC.sym == IN:
                    getSym()
                    #if '['
                    if SC.sym == LBRAK:
                        getSym()
                        #from constList, get Type(Array) and array of input
                        y, inputList = constList()
                        #inorder to push it to the stack; get the ST.Array
                        array_tp = y.val;
                        #array_name starting from for_array_0
                        array_name = "for_array_"+str(array_num)
                        #declare it, and will create global variable in genForAr
        ray()
                        newDecl(array_name, Var(array_tp))
                        #call genForArray with name of the array, user input arr
        ay
                        CG.genForArray(array_name, inputList)
                        #open the scope to store local variable
                        openScope()
                        #temp variable name starting from counter_0
                        var_name = "counter_"+str(array_num)
                        #Var int to initialize
                        temp_var = Var(Int)
                        #declare it, will create local variable in genForInit()
                        newDecl(var_name, temp_var)
                        #call genForInit with controlVariable(ident, array_name,
                        #var_name, length of input Array)
                        CG.genForInit(x, array_name, var_name, len(inputList))
                        #increment array number so it doesn't declare same array
        name
                        #if we have more than 1 array / variable
                        array_num += 1
                        #if ]
                        if SC.sym == RBRAK:
                            getSym()
                            if SC.sym == DO: getSym()
                            else: mark("'do' expected from for loop")
                    #statement() prints all the stuff b/w begin and end
                    statement()
                    #genForEnd() to close the loop
                    CG.genForEnd()
                    #closeScope -> popping the local variable after the
 loop
                    closeScope()

                else: mark("']' expected from for loop")
            else: mark("'[' expected from for loop")
        ###for controlVariable :=
        elif SC.sym == BECOMES:
            getSym()
            #init_value = initialValue; int value hopefully
            init_value = initialValue().val
            if (SC.sym == TO or SC.sym == DOWNTO):
                #set goes up to True if "to"; set to False if "downto"
                if (SC.sym == TO): goes_up = True;
                else: goes_up = False;
                getSym()
                #final_value = finalValue()
                final_value = finalValue().val
                #####setting up the array
                #input List having init_value to final_value
                inputList = []
                ####create list according goes_up
                if (init_value <= final_value and goes_up):
                    #create the list
                    while (init_value <= final_value):
                        inputList.append(Const(Int, init_value))
                        init_value = init_value + 1
                elif(init_value >= final_value and not goes_up):
                    #create the list
                    while (init_value >= final_value):
                        inputList.append(Const(Int, init_value))
                        init_value = init_value - 1
                #if user gives wrong combination of ("to/downto") and in
itialValue  and finalValue
                else: mark("can't go upto "+str(final_value)+" from "+st
r(init_value)+" or vise versa")
                #make Type(Array) so we can pass it to the function
                y = Type(CG.genArray(Array(inputList[0].tp, 0, len(input
List)-1)))
                #inorder to push it to the stack; get the ST.Array
                array_tp = y.val;
                #array_name starting from for_array_0
                array_name = "for_array_"+str(array_num)
                #declare it, and will create global variable in genForAr
ray()
                newDecl(array_name, Var(array_tp))
                #call genForArray with name of the array, user input arr
ay
                CG.genForArray(array_name, inputList)
                #open the scope to store local variable
                openScope()
                #temp variable name starting from counter_0
                var_name = "counter_"+str(array_num)
                #Var int to initialize
                temp_var = Var(Int)
                #declare it, will create local variable in genForInit()
                newDecl(var_name, temp_var)
                #call genForInit with controlVariable(ident, array_name,
                #var_name, length of input Array)
                CG.genForInit(x, array_name, var_name, len(inputList))
                #increment array number so it doesn't declare same array
name
                #if we have more than 1 array / variable
                array_num += 1
                if SC.sym == DO: getSym()
                else: mark("'do' expected from for loop")
                #statement() prints all the stuff b/w begin and end
                statement()
                #genForEnd() to close the loop
                CG.genForEnd()
                #closeScope -> popping the local variable after the loop
                closeScope()

            else:
                mark("to or downto expected from for loop")
        else: mark("in or := expected from for loop")

    ###case statement
    elif SC.sym == CASE:
        getSym()
        #x = expression
        x = expression()
        #open the scope because all the stuff will be a local Var
        openScope()
        #Counter variable for indexing array returned by constList
        counter_name = "counter_"+str(array_num)
        #Var int to initialize
        temp_var = Var(Int)
        #declare counter variable
        newDecl(counter_name, temp_var)
        #else variable to track if any case matches expression.
        else_name = "else_"+str(array_num)
        #Var int to initialize
        temp_var = Var(Int)
        #declare else variable
        newDecl(else_name, temp_var)
        #increment array_num
        array_num += 1
        #call CG.genCaseInit()
        CG.genCaseInit(counter_name, else_name)
        if SC.sym == OF:
            getSym()
            #call case where array for each case will be initialized
            #it needs counter name and else name in order to make a loop
            case(x, counter_name, else_name)
            while SC.sym == SEMICOLON:
                if (SC.sym == SEMICOLON):
                    getSym()
                case(x, counter_name, else_name)
            #elsePart needs variable else to check if else is set to
            #0 or 1. if set to 0, execute statementList else not
            elsePart(else_name)
            ###the last semicolon is taken care from elsePart -> stateme
ntlist
            if SC.sym == END:
                getSym()
                #for setting current level back to original
                CG.genCaseEnd()
                #close the scope
                closeScope()
            else:
                mark("end expected from case statement")
        else:
            mark("of expected from case statement")

    else: x = None
    return x
```

Modifications to CGwat

The procedures `genForArray`, `genForInit`, and `genForEnd` generates code for the `for` and `for each` loop statements.

In [1]:
```
#genForArray for generating array for global variable,
#initializing array with values from the input array
def genForArray(array_name, inputList):
    #generate global variables (array) starting from
    #len(topScope)-1 because array is the only one added
    genGlobalVars(topScope(), len(topScope())-1)
    #find array with ST.find which returns Var(Array)
    array = find(array_name)
    #setting up the values in the array
    for i in range (len(inputList)):
        #genVar(array) to copy
        array_copy = genVar(array)
        #index Const with tp = Int, value = i
        index = Const(Int, i)
        #value from the user input list
        value = Const(Int, inputList[i])
        #item_in_array = address of array[i];
        item_in_array = genIndex(array_copy,index)
        #put value in the array; array[i] := user_input[i]
        genAssign(item_in_array, value.val)
```

```
In [ ]: #genForInit generates while loop before the expression() stuff
        def genForInit(x, array_name, var_name, ArrayLength):
            global curlev
            global asm
            #increment current level because why not
            curlev = curlev + 1
            #generate local variable
            genLocalVars(topScope(), len(topScope())-1)
            #######################################
            ##adding local var declaration to right after func call;;;
            ###it works for fn with procedure too
            ##############################
            array= asm
            local_decl = array[-1]
            array.pop(-1)
            temp=[]
            for value in array[::-1]:
                if "func $" in value:
                    index = array.index(value)
                    temp.extend(array[:index + 1])
                    #append because it is single array
                    temp.append(local_decl)
                    temp.extend(array[index + 1:])
                    break;
            asm = temp

            #setting temp_var to 0; temp_Var := 0
            index = Const(Int, 0)
            temp_val = find(var_name)
            temp_val = genVar(temp_val)
            genAssign(temp_val, index)
            #loop
            asm.append('loop')
            ###index = length of list
            index = Const(Int, ArrayLength)
            ##temp_var < length of list
            genRelation(LT, temp_val, index)
            #if
            asm.append('if')
            ###ident := tempArray[tempindex]
            array = find(array_name)
            array_copy = genVar(array)
            #array[temp_val]
            item_in_array = genIndex(array_copy,temp_val)
            #x := array[temp_val]
            genAssign(x, item_in_array)
            ####tempindex = tempindex + 1
            one = Const(Int, 1)
            genAssign(temp_val, genBinaryOp(PLUS, temp_val, one))
```

In [ ]:
```
#genForEnd for ending the loop
def genForEnd():
    global curlev
    #current level goes down;
    curlev = curlev - 1
    asm.append('br 1')
    asm.append('end')
    asm.append('end')
```

The procedures `genCaseInit`, `genCaseArrayLoopInit`, `genCaseArrayLoopEnd`, `genCaseElseInit`, `genCaseElseEnd`, and `genCaseEnd` generates code for `case` statements.

In [ ]:
```
#genCaseInit for initializing counter variable and else variable
def genCaseInit(counter_name, else_name):
    global asm
    global curlev
    curlev = curlev + 1
    #len(topScope())-2 because generating two variables
    genLocalVars(topScope(), len(topScope())-2)
    #######################################
    ##adding local var declaration to right after func call;;;
    ###it works for fn with procedure too
    ##############################
    array= asm
    local_decl = array[-2:]
    array.pop(-1)
    array.pop(-1)
    temp=[]
    for value in array[::-1]:
        if "func $" in value:
            index = array.index(value)
            temp.extend(array[:index + 1])
            #extend it because it is an array
            temp.extend(local_decl)
            temp.extend(array[index + 1:])
            break;
    asm = temp
    #set the else variable to 1, which states it has to go to else loop
    #Const one for integer
    one = Const(Int, 1)
    #find the local Var with else_name
    temp_val = find(else_name)
    temp_val = genVar(temp_val)
    genAssign(temp_val, one)
```

```
In [ ]: #genCaseArray with input (x.Var)
        def genCaseArrayLoopInit(x, array_name, inputList, counter_name, else_na
        me):
            #len(topScope())-1 because generating one variables(array)
            genGlobalVars(topScope(), len(topScope())-1)
            ####generating the array with values from the inputList
            #find array with ST.find which returns Var(Array)
            array = find(array_name)
            #setting up the values in the array
            for i in range (len(inputList)):
                #genVar(array) to copy
                array_copy = genVar(array)
                #index Const with tp = Int, value = i
                index = Const(Int, i)
                #value from the user input list
                value = Const(Int, inputList[i])
                #item_in_array = address of array[i];
                item_in_array = genIndex(array_copy,index)
                #put value in the array; array[i] := user_input[i]
                genAssign(item_in_array, value.val)
            #set the counter variable to 0.
            #Const one for integer
            zero = Const(Int, 0)
            #find the local Var with else_name
            temp_val = find(counter_name)
            counter_var = genVar(temp_val)
            genAssign(counter_var, zero)
            #######################starting loop
            #outer loop
            asm.append('loop')
            ###index = length of list
            index = Const(Int, len(inputList))
            ##temp_var < length of list
            genRelation(LT, counter_var, index)
            #if temp_var < length of list, get the array[temp_var] and
            #compare it with x and if it is equal, do the statements
            asm.append('if')
            ###ident := tempArray[tempindex]
            array = find(array_name)
            array_copy = genVar(array)
            item_in_array = genIndex(array_copy,temp_val)
            #if array[temp_var] == x
            genRelation(EQ, item_in_array, x)
            #if statement to check array[temp_var] == x
            asm.append('if')
            ## if it is equal, case statement has executed, and else part does
        n't
            ## needs to be executed
            #Const one for integer
            zero = Const(Int, 0)
            #find the local Var with else_name
            temp_val = find(else_name)
            temp_val = genVar(temp_val)
            genAssign(temp_val, zero)
```

In [ ]:
```
##genCaseArrayLoopEnd for ending if statement and loop
##also incrementing counter up 1
def genCaseArrayLoopEnd(counter_name):
    ##break the if statement
    asm.append('end')
    ####tempindex = tempindex + 1
    temp_val = find(counter_name)
    counter_var = genVar(temp_val)
    one = Const(Int, 1)
    genAssign(temp_val, genBinaryOp(PLUS, counter_var, one))
    ##break the while loop
    asm.append('br 1')
    asm.append('end')
    asm.append('end')
```

In [ ]:
```
#genCaseElseInit for checking if any of the case statement has been
#executed, and if so, var else should be set to 0
def genCaseElseInit(else_name):
    #Const one for integer
    one = Const(Int, 1)
    #if else var is still set to 1, execute the else part
    #find the local Var with else_name
    temp_val = find(else_name)
    temp_val = genVar(temp_val)
    genRelation(EQ, temp_val, one)
    asm.append('if')
```

In [ ]:
```
#genCaseElseEnd for closing if statement generated by genCaseElseInit
def genCaseElseEnd():
    asm.append('end')
```

In [ ]:
```
def genCaseEnd():
    global curlev
    curlev = curlev - 1
```