Project Report

CS3071 Lab 7

Yasir Mohamed – 13318246

Arrays

In the attached code submission, I have successfully implemented working 2 Dimensional 0 indexed arrays. This was completed through modifying code only in the Tastier.ATG source file in the Tastier Compiler.

Array Initialisation

```
[ '[' Expr<out type>
  ( ']'
                (.
                    if(type != TastierType.Integer ){
                      SemErr("Error, Integer Expected");
                    }else{
                      xDimension = expression;
                      for(int i=0; i<xDimension;i++){
                        string temp = name+i;
                        int stackPos = currentScope.Count(s => (s.Item2 == (int)TastierKind.Var || s.Item2 == (int)TastierKind.Cons));
                        sym = new Symbol(temp, (int)TastierKind.Var, (int)typeStore, openScopes.Count-1, stackPos);
                        currentScope.Push(sym);
                      }
                    }
                .)

    | ','
                (.
                  if(type!=TastierType.Integer){
                    SemErr("Error, Integer Expected");
                  }else{
                    xDimension = expression;
                  }
                .)

    Expr<out type> ']'
                (.
                  if(type != TastierType.Integer){
                    SemErr("Error, Integer Expected");
                  }else{
                    int j, i;
                    yDimension = expression;

                    for(j=0; j<yDimension; j++){
                      for(i=0; i<xDimension;i++){
                        string temp = name+i+""+j;
                        int stackPos = currentScope.Count(s => (s.Item2 == (int)TastierKind.Var || s.Item2 == (int)TastierKind.Cons));
                        sym = new Symbol(temp, (int)TastierKind.Var, (int)typeStore, openScopes.Count-1, stackPos);
                        currentScope.Push(sym);
                      }
                    }
                  }
                .)
    )

]
```

Figure 1.1 – Code to implement array initialisation

The above code snippet shows how I allowed arrays to be initialised In Tastier. My approach was to allow array index bounds to be optionally included in variable declaration, and when found the array bounds were saved through the use of a global variable expression. This was then used to initialise all the indexes in the array by pushing a symbol to the stack (in the scope the array was declared in) for each index in the array. My reasoning behind this is that when an array is initialised, it is a contiguous block of memory allocated on array initialisation. So by initialising all variables in an array, I am allocating the appropriate space on the stack for the array. The limitation of the array implementation above is that it only works for integer and Boolean data types, and will only work in a maximum of two dimensions. The symbol name for each array is simply decided by the array name, and the index that the symbol represents.

Writing to Arrays

```
= Ident<out name>
  ['[' Expr<out type>          (.    xDimension = expression;    .)
    (']'
                               (.
                                 name = name+xDimension;
                                 array = true;
                               .)
    | ',' Expr<out type> ']'
                               (.
                                 name = name + xDimension + "" + expression;
                                 array = true;
                               .)
    )

  ]
                               (.
                                 try{
                                   sym = lookup(openScopes, name);
                                   if (sym == null) {
                                     sym = _lookup(externalDeclarations, name);
                                     isExternal = true;
                                   }
                                   if(sym == null){
                                     throw new System.NullReferenceException();
                                   }
                                 }catch (Exception){
                                   if (sym == null && array == false) {
                                       SemErr("reference to undefined variable " + name);
                                   }
                                   else if(sym == null && array == true){
                                     SemErr("Error, array index out of bounds");
                                   }
                                   System.Environment.Exit(100);
                                 }
                             .)
```

Figure 1.2 – Saving to an Array

The advantage of my array implementation is that values can be stored to arrays in much the same way that values are stored to single variables. Since an array index would be accessed in the form “array[2]” or “array[1,3]”, I can simply use the array indexes to form the symbol name. In this scenario, the symbol names would be either “array2” or “array13”. This then follows the same route that all other variables would follow when saving values: i.e. look up the symbol (array2/array13) and write to the stack position.

Accessing Array Indexes

```
( Ident<out name>
  ['[' Expr<out type>      (.
                             xDimension = expression;  //Get X-Dimension of array
                           .)
    (']'
                 (.
                   name = name+xDimension;
                   array = true;
                 .)
    | ',' Expr<out type> ']'
                 (.
                   name = name + xDimension + "" + expression; //Append X and Y dimansion of array
                   array = true;
                 .)
    )

  ]
```

Figure 1.3 – Accessing Array Indexes

Done in much the same way as writing to an array, the square brackets of an array index are made to be an optional parameter for an Identifier. This allows for the use of the same code used to access other Variables in Taster. Again, the array index values are appended to the arrays name in order to find the correct symbol name. This symbol name is then looked up and used to accordingly store values to the symbol representing the specified array index.

Testing

```
program arrayTest{
        void Main(){
                int oneD[10];
                int twoD[10,4];
                int i;
                oneD[1] := 100;
                oneD[7] := 200;
                twoD[4,3] := 300;

                write oneD[1], oneD[7], twoD[4,3];
        }
}
```

Figure 1.4 – Array Test Program

```
mohamey@macneill:~/TastierCompiler$ ~/.cabal/bin/tvm arraytest.bc test/Inputs/test.IN
["100","200","300"]
mohamey@macneill:~/TastierCompiler$
```

Figure 1.5 – Array Test Results

As can be see from the tests above, arrays are successfully initialised, written to and accessed by Tastier.

Limitations

There are some limitations associated with my array implementation when compared to implementations done through modifications to the Tastier Machine.

1. Array Indexes cannot be accessed through the use of variables. This is a major limitation as it means arrays cannot be effectively used in loops, and accessed to array indexes can only be hardcoded.
2. This implementation is not fully multi-dimensional – It can only accommodate two dimensions. This is in part to unsuccessfully modifying the Tastier Machine and the implementation within TastierCompiler.

Switch Statements

In the attached code submission, I have successfully implemented a Switch-case statement with an optional default case. This was done through modifying code only in Tastier.ATG in the Tastier Compiler Source Directory.

Code

```
| "switch"                          (.      //Create a boolean to symbol to track whether or not to use the default case
                                            Scope currentScope = openScopes.Peek();
                                            sym = new Symbol("switch", (int)TastierKind.Var, 1, 0, currentScope.Count(s => (s.Item2 == (int)TastierKind.Var || s.Item2 == (int)TastierKind.Cons)));
                                            currentScope.Push(sym);
                                            program.Add(new Instruction("", "Const " + 0));
                                            program.Add(new Instruction("", "StoG " + (sym.Item5+3)));
                                    .)
  '(' Expr<out type>
')''{'                              (.
                                            openLabels.Push(generateLabel());
                                    .)
    {
        "case" Expr<out type> ':'   (.   program.Add(new Instruction("", "Dup"));   .)
                                    (.
                                        if((TastierType)type != TastierType.Integer){
                                            SemErr("Error, Integer type expected");
                                        }
                                        else{
                                            program.Add(new Instruction("", "Equ"));
                                            program.Add(new Instruction("", "FJmp "+openLabels.Peek()));
                                        }
                                    .)
                    Stat            (.
                                       Instruction startOfOptCase = new Instruction(openLabels.Pop(), "Nop");
                                       openLabels.Push(generateLabel());
                                       program.Add(new Instruction("", "Const " + 1));
                                       program.Add(new Instruction("", "StoG " + (sym.Item5+3)));
                                       program.Add(new Instruction("", "Jmp " + openLabels.Peek()));
                                       program.Add(startOfOptCase);
                                    .)
    }

    [
                    (.                      //Check if we should enter default case
                                            program.Add(new Instruction("", "Const " + 0));
                                            program.Add(new Instruction("", "LoadG " + (sym.Item5+3)));
                                            program.Add(new Instruction("", "Equ"));
                                            program.Add(new Instruction("", "FJmp "+openLabels.Peek()));
                    .)
        "default":'
                Stat
    ]
  '}'                               (.
                                        program.Add(new Instruction(openLabels.Pop(), "Nop"));
                                    .)
```

Figure 2.1 – Switch Statement Implementation

The switch statement implementation can accept multiple cases and an optional default statement. It works by first of all saving a symbol to the stack which will be used by the switch statement to check if the program needs to enter the default case. It makes use of the program instruction “Dup”, which duplicates the value of the switch statement and allows it to be reused in comparing its value with the value of cases. After the value of the switch statement is duplicated, it is compared with the value of the case. If the two values are not equal, and Fjmp instruction is added and the program jumps over the case to the next one. However, if it is found to be equal, the code block in stat is executed, the program stores that it has been evaluated in the switch symbol, and continues on through the remaining cases.

Finally, when the optional default statement is reached, the compiler evaluates whether the default statement should be executed based on the result of comparing the symbol for the switch with 0 (false). If the equation returns false, the program jumps over the default. Otherwise, it executes the default statement.

Testing

```
program SwitchTest{
        void Main(){
                int temp[10];
                temp[7] := 14;
                switch(temp[7]){
                        case 10:
                                temp[2]:= 100;
                        case 11:
                                temp[2] := 200;
                        case 12:
                                temp[2] := 300;
                        case 13:
                                temp[2] := 400;
                        case 14:
                                temp[2] := 500;
                        case 15:
                                temp[2] := 600;
                        default:
                                temp[2] := 1000;
                }
                write temp[2];
        }
}
```

Figure 2.2 – Test Case 1

```
program SwitchTest{
        void Main(){
                int temp[10];
                temp[7] := 14;
                switch(temp[7]){
                        case 10:
                                temp[2]:= 100;
                        case 11:
                                temp[2] := 200;
                        case 12:
                                temp[2] := 300;
                        case 13:
                                temp[2] := 400;
                        case 140:
                                temp[2] := 500;
                        case 15:
                                temp[2] := 600;
                        default:
                                temp[2] := 1000;
                }
                write temp[2];
        }
}
```

Figure 2.3 – Test Case 2

```
mohamey@macneill:~/TastierCompiler$ ~/.cabal/bin/tvm SwitchTest.bc test/Inputs/test.IN
["500"]
mohamey@macneill:~/TastierCompiler$
```

Figure 2.4 – Results of Test Case 1 (Successful)

```
mohamey@macneill:~/TastierCompiler$ ~/.cabal/bin/tvm SwitchTest.bc test/Inputs/test.IN
["1000"]
mohamey@macneill:~/TastierCompiler$
```

Figure 2.5 – Results of Test Case 2 (Successful)

As can be seen from the above test cases, the switch statement functions as required when either a case or the default statement needs to be executed.

Extra Feature – Runtime Array bounds Checking

In the code submission, I've successfully implemented runtime array bounds checking for the compiler. When a program is compiled, it checks the indexes of all array accesses to ensure they are all within the bounds of a specified array.

Code

```
(.
  bool isExternal = false; //CS3071 students can ignore external declarations, since they only deal with compilation of single files.
  try{
    sym = lookup(openScopes, name);
    if (sym == null) {
      sym = _lookup(externalDeclarations, name);
      isExternal = true;
    }
    if(sym == null){
      throw new System.NullReferenceException();  //Variable was not found
    }
    type = (TastierType)sym.Item3;
    if ((TastierKind)sym.Item2 == TastierKind.Var || (TastierKind)sym.Item2 == TastierKind.Cons) {
        if (sym.Item4 == 0) { //If it is a global variable
            if (isExternal) {
              program.Add(new Instruction("", "LoadG " + sym.Item1));
              // if the symbol is external, we load it by name. The linker will resolve the name to an address.
            } else {
              program.Add(new Instruction("", "LoadG " + (sym.Item5+3)));
            }
        } else {
            int lexicalLevelDifference = Math.Abs(openScopes.Count - sym.Item4)-1;
            program.Add(new Instruction("", "Load " + lexicalLevelDifference + " " + sym.Item5));
        }
    } else
      SemErr("variable expected");
  }catch (Exception){
    if (sym == null && array == true) {
      SemErr("Error, array index out of bounds");    //Array Index was out of bounds
    }
    else{
      SemErr("reference to undefined variable " + name);
    }
    System.Environment.Exit(100);
  }
.)
```

Figure 3.1 – Array bounds check on Variable Load

Since array indexes are stored as symbols on the stack, an array index can be checked to see if it is in bounds by looking up the symbol. If the array index is confirmed to be a variable and its symbol is not present on the stack then the Array must be out of bounds. This approach for checking array indexes is also used when storing values in array indexes, as can be seen in the code snippet below.

```
(.
  try{
    sym = lookup(openScopes, name);
    if (sym == null) {
      sym = _lookup(externalDeclarations, name);
      isExternal = true;
    }
    if(sym == null){
      throw new System.NullReferenceException();
    }
  }catch (Exception){
    if (sym == null && array == false) {
        SemErr("reference to undefined variable " + name);
    }
    else if(sym == null && array == true){
      SemErr("Error, array index out of bounds");
    }
    System.Environment.Exit(100);
  }
.)
```

Figure 3.2 – Checking Array indexes before variable assignment

In this code snippet, the code in the first bubble is only executed when the symbol name cannot be found. It throws a Null Reference Exception which is handled by the catch block. If the symbol being referenced cannot be found and has been deemed to be a valid variable then it's index must be out of bounds, so an exception is thrown and handled.

Testing

```
program BoundsTest{
        void Main(){
                int oneD[10];
                oneD[2] := 210;
                int twoD[21,14];

                twoD[21,14] := 213;
                write oneD[2];
        }
}
```

Figure 3.3 – Testing out of bounds index for variable assignment

```
mohamey@macneill:~/TastierCompiler$ mono bin/tcc.exe test/Programs/BoundsTest.TAS BoundsTest.asm
-- line 7 col 13: Error, array index out of bounds
mohamey@macneill:~/TastierCompiler$
```

Figure 3.4 – Results of first test case (line 7 is "twoD[21,14] := 213;")

```
program BoundsTest{
        void Main(){
                int oneD[10];
                oneD[2] := 210;
                int twoD[21,14];

                twoD[20,13] := 213;
                write oneD[10];
        }
}
```

Figure 3.5 – Testing accessing an out of bounds array index

```
mohamey@macneill:~/TastierCompiler$ mono bin/tcc.exe test/Programs/BoundsTest.TAS BoundsTest.asm
-- line 8 col 16: Error, array index out of bounds
mohamey@macneill:~/TastierCompiler$
```

Figure 3.6 – Results of Test case above (Line 8 is the Write Statement)