Compiler Design Exercise 7 Report

By Kevin Hennessy – 11726665

Contents

Exercise 7 One Page Summary:

Features implemented:

Structs – Simple data structure, similar to C style structs. Can store members of all types, including other structs.

Arrays – Single and multidimensional. Can make a Struct array.

Switch Statement – Java style flow control statement with 'default' case.

Extra feature – N/A (Attempted to implement procedure parameters, could not get it fully working)

Documentaion + Comments – Language specification and explanatory comments

Sample Program and output:

Sample program:

```
program MyProgram {

/*Struct declaration*/

  struct date {
   int day;
   int month;
   int year;
   bool publicholiday;
  };

  struct person {
   int favouritenumber;
   struct date birthday;
  };

  struct vehicle {
   struct person owner;
   struct date registrationdate;
   int numberofwheels;
  };

  /*Attempt at parameter passing for extra feature. (writes 0 instead of 42 which is incorrect)*/
  _int myfun( int writeme2){
    write writeme2;
  }

  _void Main() {
     int  var;
     var := 21;
     int writeme;

    /*switch statement*/

     switch(var){
       case 0:
          writeme := 101;
          break;
       case 1:
          writeme := 102;
          break;
       default:
```

```
        writeme := 103;
        break;
    }
    write writeme;
    struct date mydate;

  /*Array declaration */
    int [2] myarray;
    int [2][2][2] ia;
    struct date [1][2] mydatearray;

   /*array member usage*/
        myarray[0] := 22 + mydatearray[0][1].day;
        write myarray[0];

            /*struct member usage*/
       mydate.day := 32;
       mydate.day := mydate.day + mydate.day;
       write mydate.day;

   /*attempt at passing parameters*/
    writeme := 42;
    myfun(writeme);
  }
}
```

Output:

```
["103","22","64","0"]
```

Record feature:

For my programming language, I chose to implement C style structs. Structs in my program have three main use cases: Definition, Instantiation and Member Retrieval.

Struct Definition:

```
struct date {
 int day;
 int month;
 int year;
 bool publicholiday;
};

struct person {
 int favouritenumber;
 struct date birthday;
};

struct vehicle {
 struct person owner;
 struct date registrationdate;
 int numberofwheels;
};
```

Struct definition is where you define what structs will be in your program. Above is a sample piece of code that defines three structs, two of which have struct members. The compiler knows you are defining a struct when you reach the '{' character after the keyword struct and an identifier.

At the start of my program we declare a dictionary called definedStructs whose Key references another dictionary which represents the struct member. (Their name is used as a Key to reference type and kind.) When we know we are defining a struct, we first check if it exists already in the definedStructs (cant have two "date" objects for example) and then if it doesn't we add the members to the dictionary one by one.

When we come across the keyword "struct" and an identifier during our struct definition (eg inside the person struct) we have come across a nested struct. How we handle this is to stop and check our dictionary to see if the struct exists, and if it does, we must have its members in the dictionary, so we run through them one by one and add them as attributes to the struct, except we add the <ident>. To the start of it. Here is the debugging output

of the compiler as it parses the person struct:

```
Struct 'person' defined.  Attributes=(
   favouritenumber, type = 1, kind = 1,
   birthday.day, type = 1, kind = 1,
   birthday.month, type = 1, kind = 1,
   birthday.year, type = 1, kind = 1,
   birthday.publicholiday, type = 2, kind = 1,
)
```

In this way we can have arbitrary nested structs, for example:

```
Struct 'vehicle' defined.  Attributes=(
   owner.favouritenumber, type = 1, kind = 1,
   owner.birthday.day, type = 1, kind = 1,
   owner.birthday.month, type = 1, kind = 1,
   owner.birthday.year, type = 1, kind = 1,
   owner.birthday.publicholiday, type = 2, kind = 1,
   registrationdate.day, type = 1, kind = 1,
   registrationdate.month, type = 1, kind = 1,
   registrationdate.year, type = 1, kind = 1,
   registrationdate.publicholiday, type = 2, kind = 1,
   numberofwheels, type = 1, kind = 1,
)
```

Struct Instantiation:

Struct instantiation is where we declare a struct instance that we can use. An example of this is:

```
Struct date mydate;
```

The compiler knows that we want to instantiate a struct when it sees the ";" symbol after "struct" and an identifier.

First we have to make sure that the struct has been defined already, so we use the struct name (date in this case) and search the dictionary.

When we find the members that for date that we defined, what we do is run through them one by one and add the string "mydate." to the front. As we do this we allocate space for the variables one by one by adding them to the stack of symbols in the program.

To make sure that the names we are generating for struct members are unique is to ban the

use of the '.' symbol for any variable instantiation done by the programmer. This way if the programmer want to make an integer called a.b he will be stopped, however if he makes a struct 'a' with int 'b' as a member he is free to modify it as he wishes.

Member retrieval:
As explained in the previous section struct members are just regular variables they can be used just like anything else in the language. Here is an example of this:

```
struct person kevin;
struct person eimear;
kevin.birthday.day := 21;
write kevin.birthday.day;
kevin.birthday.day := kevin.birthday.day + 21;
write kevin.birthday.day;
eimear.birthday.day := kevin.birthday.day -1;
write eimear.birthday.day;
```

Arrays:
There are two kinds of array in my program: struct arrays and non-struct arrays.

Array declaration:

```
int [2] myarray;
int [4][2][2] ia;
struct vehicle [5][5] da;
```

Arrays are implemented in a very similar way to structs in my program, in that an array is basically a naming scheme for variables, and instantiating an array is like instantiating a lot of variables at once.

When we see the declariations above, I use an algorithim to generate all the possible elements and allocate space for all of them. I took advantage of the regular pattern that emerges when you list all array elements.
For example when we have "int [1][2][1] ia;" we get the pattern:

```
ia[0][0][0]
ia[0][0][1]
ia[0][1][0]
ia[0][1][1]
ia[0][2][0]
ia[0][2][1]
ia[1][0][0]
ia[1][0][1]
ia[1][1][0]
ia[1][1][1]
ia[1][2][0]
ia[1][2][1]
```

I used the three numbers multiplied to get a total, and used the below piece of code to generate the variable names:

```
 /*Algorithim to generate all array members*/
Dictionary<int, string> Dict = new Dictionary<int, string>();
for (int i =0; i< total; i++){
   Dict.Add(i, name);
}
int beforeincrement = total;
for(int i =0; i< dimensions; i++){
     int mynum = numElementsQueue.Dequeue();
      beforeincrement = (beforeincrement / (mynum+1));
```

```
                int writevalue = 0;
                int count =0;
                Console.WriteLine(">   beforeincrement = "+beforeincrement+", mynum = "+mynum$

                for(int j =0; j< total; j++){
                  if(count == beforeincrement){
                    writevalue++;
                    writevalue = writevalue % (mynum+1);
                    count = 0;
                  }
                  count++;
                  Dict[j] =  Dict[j] + "["+writevalue+"]";
                }
            }
```

When we have done this, we interate through the array element dictionary and we allocate space for all the variabes by creating symbols for them in the currentScope. (Similar to structs, the programmer is not allowed use the '[' or ']' character when declaring variables to stop them abusing this.) In this way declaring an integer array with 12 elements is just like declaring 12 integers at once.

Here is the debugging output for the above array declaration:

```
    Array 'ia' declared. Type = 'int' Number of dimensions = 3. Elements = (
    Variable ia[0][0][0] declared. = (kind = 1, type = 1, stackframe = 1, stackframeoffset = 0),
    Variable ia[0][0][1] declared. = (kind = 1, type = 1, stackframe = 1, stackframeoffset = 1),
    Variable ia[0][1][0] declared. = (kind = 1, type = 1, stackframe = 1, stackframeoffset = 2),
    Variable ia[0][1][1] declared. = (kind = 1, type = 1, stackframe = 1, stackframeoffset = 3),
    Variable ia[0][2][0] declared. = (kind = 1, type = 1, stackframe = 1, stackframeoffset = 4),
    Variable ia[0][2][1] declared. = (kind = 1, type = 1, stackframe = 1, stackframeoffset = 5),
    Variable ia[1][0][0] declared. = (kind = 1, type = 1, stackframe = 1, stackframeoffset = 6),
    Variable ia[1][0][1] declared. = (kind = 1, type = 1, stackframe = 1, stackframeoffset = 7),
    Variable ia[1][1][0] declared. = (kind = 1, type = 1, stackframe = 1, stackframeoffset = 8),
    Variable ia[1][1][1] declared. = (kind = 1, type = 1, stackframe = 1, stackframeoffset = 9),
    Variable ia[1][2][0] declared. = (kind = 1, type = 1, stackframe = 1, stackframeoffset = 10),
    Variable ia[1][2][1] declared. = (kind = 1, type = 1, stackframe = 1, stackframeoffset = 11),
)
```

A struct array is similar, but you must allocate space for the members aswell. Eg for

```
    struct person [1][1] da;
```

The output will be:

```
    Struct Array 'da' declared. Type = 'struct_person' Number of dimensions = 2. Members(
    Variable 'da[0][0].favouritenumber' declared. = (kind = 1, type = 1, sframe = 1, offset = 12),
```

```
    Variable 'da[0][0].birthday.day' declared. = (kind = 1, type = 1, sframe = 1, offset = 13),
    Variable 'da[0][0].birthday.month' declared. = (kind = 1, type = 1, sframe = 1, offset = 14),
    Variable 'da[0][0].birthday.year' declared. = (kind = 1, type = 1, sframe = 1, offset = 15),
    Variable 'da[0][0].birthday.publicholiday' declared. = (kind = 1, type = 2, sframe = 1, offset = 16),
    Variable 'da[0][1].favouritenumber' declared. = (kind = 1, type = 1, sframe = 1, offset = 17),
    Variable 'da[0][1].birthday.day' declared. = (kind = 1, type = 1, sframe = 1, offset = 18),
    Variable 'da[0][1].birthday.month' declared. = (kind = 1, type = 1, sframe = 1, offset = 19),
    Variable 'da[0][1].birthday.year' declared. = (kind = 1, type = 1, sframe = 1, offset = 20),
    Variable 'da[0][1].birthday.publicholiday' declared. = (kind = 1, type = 2, sframe = 1, offset = 21),
    Variable 'da[1][0].favouritenumber' declared. = (kind = 1, type = 1, sframe = 1, offset = 22),
    Variable 'da[1][0].birthday.day' declared. = (kind = 1, type = 1, sframe = 1, offset = 23),
    Variable 'da[1][0].birthday.month' declared. = (kind = 1, type = 1, sframe = 1, offset = 24),
    Variable 'da[1][0].birthday.year' declared. = (kind = 1, type = 1, sframe = 1, offset = 25),
    Variable 'da[1][0].birthday.publicholiday' declared. = (kind = 1, type = 2, sframe = 1, offset = 26),
    Variable 'da[1][1].favouritenumber' declared. = (kind = 1, type = 1, sframe = 1, offset = 27),
    Variable 'da[1][1].birthday.day' declared. = (kind = 1, type = 1, sframe = 1, offset = 28),
    Variable 'da[1][1].birthday.month' declared. = (kind = 1, type = 1, sframe = 1, offset = 29),
    Variable 'da[1][1].birthday.year' declared. = (kind = 1, type = 1, sframe = 1, offset = 30),
    Variable 'da[1][1].birthday.publicholiday' declared. = (kind = 1, type = 2, sframe = 1, offset = 31),
)
```

Switch Statements:

I was able to implement switch statements into the program, for example:

```
Int var;
int writeme;
writeme := 0;
var := 0
switch(var){
  case 0:
      writeme := 100;
      write writeme;
      break;
   case 1:
      writeme := 99;
      write writeme;
      break;

  default:
     writeme := 101;
     struct date mydate;
     break;
 }
```

These were implemented using assembly language loops, very similar to the "if" statements.

```
"switch" '('  Expr<out type,  varname>   ')' '{'   (.

                                        openLabels.Push(generateLabel());
                                       .)
{
 "case"  Expr<out type1,  varname>   ':'              (.

                                        if (type != type1){
                                          SemErr("incompatible types in switch statement");
                                        }
                                        openLabels.Push(generateLabel());
                                        program.Add(new Instruction("", "Equ"));
                                         program.Add(new Instruction("", "FJmp " + openLabels.Peek()));

                                       .)
```

```
Stat [{ VarDecl<external>  | StructDecl<external>}]  (.
                                        string label = openLabels.Pop();
                                         program.Add(new Instruction("", "Jmp " + openLabels.Peek() ));
                                         program.Add(new Instruction(label, "Nop"));

                                        .)
 "break;"
}
[
 "default" ':'

 Stat [{ VarDecl<external>  | StructDecl<external>}]

 "break;"

'}' (.  program.Add(new Instruction(openLabels.Pop(), "Nop"));  .)
```

The switch statement checks the type of the variable passed in and give an error if the case conditions do not match. The default case is required in the case statement, wheras the cases are not. Because of the way case statements are implemented (as part of Stat), you can have nested case statements if desired.