CS3071 – Exercise 5 Report

Edward Bergman
#15335633

Terminology:

constexpr – A compile time known expression.

Task:

Our tasks were to implement constants and arrays. The understanding of these terms according to the examples and descriptions given:

Constant – an object kind that stores a type and value from a constexpr that once assigned, can't be reassigned.

Array – A C-like array with bounds that must be initilized by constexpr values at definition. The syntax chosen for arrays follows the more traditional C-like style of *arrayName[index1][index2][index3]...* . These arrays are also 0 indexed.

Method:

General

Modifications were done to the Symbol Table 'CloseScope' method so that further debugging information could be gleaned while developing a solution to the task. Lists are used to implement N-dimensional arrays so imports are used in the ATG along with the new kinds that needed to be defined 'constant' and 'array'.

Constant

The first decision that had to be made was where is constant declaration allowed. The design decision taken was to allow constant declarations where ever variable declarations were allowed.

The second decision was whether to use "hot swapping", essentially replace all occurances of the const Identity with a LoadConstant, removing the need to have the constants as objects during runtime. This method, while simpler in the current state of the language specification, could lead to difficulties if more advanced constants are introduced so the decision was made to keep them as runtime objects during execution but with a value that is known at compile time.

To implement constants, we must define a new kind of object, 'constant', in the symbol table. Constants are something that can be evaluated at compile time and hence must have a value that can be reffered to at compile time. To do this, we also add a feild to the Obj type in SymTab.cs that specifies a compile time value that can be referenced.

With the decisions made and the information in the SymbolTable to represent a constant. We move to the modifications to the ATG.

The first part was to summarise all possible compile time evaluatable expressions which is what ConstExpr is as a production rule.

ConstExpr = (number | ‘true’ | ‘false’ | Ident)

With the condition that the ident itself is a constant so we can get its compile time value.

With this we can now write the production rule for our ConstDecl to declare a constant.

ConstDecl = “const” Type Ident “=” ConstExpr

While *Type* is not neccesary (its type can be infered from ConstExpr), the view of Tastier is that of a type safe language so the *Type* remains in place, possibly preventing an unintended type cast if the language was to be extended in that direction. Alternativly a keyword (C++’s *auto)* could be used to allow type inference but this is beyond the scope of this exercise and merely serves as a discussion as to why *Type* is kept over the example that was given for the assignment.

Other than it’s initilisation, const kinds behave similar to var kinds except that const’s can’t be assigned a value after initilisation.

Arrays

The first decision was whether to have dynamic or static arrays. In the interest of simplicity, static was chosen. This allows more information to be kept about the array in the symbol table and proper space allocation for the array on the stack. We are unsure if dynamic arrays could be possible as stack space is allocated compile time for the objects i.e.

int a; int b[x][y]; int c[b[2][3]];

These lines lead to an issue in that we don’t know where to place ‘c’ on the stack as we don’t know how much space ‘b’ will use up. It could be possible to allocate space for ‘b’ and ‘c’ to be a pointer to where the array start is (the extra layer of indirection allows us to be more flexible). Allowing dynamic arrays also needs us to keep runtime information on the array bounds which again need some form of stack storage, possibly before the array elements themselves. With all this to consider, static arrays were far simpler to implement and hence were chosen.

The second decision was to use the more traditional syntax that is described under the task description. This is for the simple reason that if expressions are allowed in the index, this provides a much clearer view of indexing into an array.

To implement arrays, we define another new kind in the symbol table, array. An array Obj’s type is still taken into account (interger or boolean). This means that var now signifies the information that it is of a scalar kind. As these arrays are static, we must know their bounds at compile time, meaning we must have a way of storing information of the array’s dimensions and bounds. This (ironically) is modelled best with a 1d int array, its length being the total dimensions and its values, the bounds of each dimension.
Let N*i* = bounds of a dimension *i*:

Stack Space Required = N*0* * N*1* * ... * N*i*

We then simply change the next available address in the current stack to be after the space required for the array.

The production rule for an array is relativly straightforward since we have defined a rule for what we call a ConstExpr

ArrayDecl = ‘[‘ ConstExpr ‘]’ { ‘[‘ ConstExpr ‘]’ }

As for the implementation behind this, we use a C# list to store the dimensions as the { ‘[‘ ConstExpr ‘]’ } part is recognized. This allows us to have the information needed to construct a N-Dimensional array in the Symbol Table.

Perhaps the most difficult part of the array was indexing into an N-Dimensional array. Using this website as a resource we obtained a formula for indexing from an N-Dimensional array to a 1d array, which is how it is actually stored.

N*i* = Size of Dimension *i*;
d*i* = index into Dimension *i*;

1d index = ((d*0* * N*1* + d*1*) * N*2* + d*2*) * N*3* + d*3* ....

This calculation is done in its own production rule

ArrayIndex = ‘[‘ Expr ‘]’ { ‘[‘ Expr ‘]’ }

With each iterartion of *‘[‘ Expr ‘]’* we generate ARM code to do bounds checking between 0 and N*i*, that it’s dimensions match and accumulate the result that is the 1d index into the array. This index can be used in conjuction with LoadIndexed... and StoreIndexed... functions available in the CodeGen which effectivly turns an index into an address of an array.

Finally we have two parts left to deal with, storing into an array and retrieving a value from an array.

To retrieve a value from an array we can use our previous production rule ArrayIndex. Accessing a value was put in the Primary production rule as a Primary is a single valued object which is also the result of any value in an array.

Primary = (Ident [ArrayIndex] | ...)

Without explaining all the code, we simply check that if the ArrayIndex is recognized, we check that the Ident is an array, otherwise it is a compile time error. Likewise if the ArrayIndex is not recognized, we make sure the object specified by Ident is of kind var or const.

To store a value into an array we also use our previous production rule ArrayIndex. We insert this into the Stat production rule as that is where assignment is dealt with for identities and could be expanded for arrays.

Stat = (Ident (‘:=’ Expr | ‘()’ | ArrayIndex ‘:=’ Expr) | ...)

As we can see, we have two seperate kinds of assignments, one for a var/const and one for an array. This was done to more easily seperate code segments, rather than having optional Production rules and introducing more layers of complexity. As for the implementation, it mostly follows the above for loading a value from an array except with a slight difference in that we are now using

StoreIndexedLocal and we push the calculated Index from ArrayIndex on to the stack. This is done to prevent it being overwritten during the evaluation of the Expr on the right hand side.

Remarks:

Most of the important remarks are made during the design decisions aspects of both tasks. We would have liked to have tried to implement dynamic arrays but due to time constraints, we felt static arrays satisfied the task adequately. Another slight remark is on the definition of ‘const’. With only primitive types, they didn’t need the functionality of the C++ ‘const’ qualifier but we felt having them be the equavilant of C macros where they are pure compile time details was not sufficient enough to hold up if more future functionality is needed.

Seperating ConstExpr into another production rule served two purposes. One is to capture what can be evaluated during compile time. The second was for possible future extensibilty of the language. Taking inspiration fron C++’s ‘constexpr’ keyword, it effectivly acts as a peice of code that can be evaluated during compile time. While a lot more featured than our version, it does show the power of a Compiler to reduce runtime expressions such as 3 + 4 to a compile time value which is 7. This opens the door to a lot of easy optimization, we can now evaluate these ConstExpr if they are defined in terms of of other ConstExpr segments. This is of course not done in this assignment as this is not a small task but arises as an interesting avenue for later development.

We were also not in a position to run the generated .s code. The ARM code was verified to the best of our ability, using a more complex CloseScope() function to output scope information as well as include more comments in the .s file using Console.WriteLine() in the ATG itself to help break up the segments. Hopefully this report along with the provided code is sufficient enough to show an informed thought process to the decisions made and the code written.

Learnt:

Firstly, we learn’t that compilers have a powerful ability to have compile time values. This allows us to check for issues at compile time instead of runtime, an obvious advantage. i.e.

```
const var x = -10;
int buf[x]; //compile time error
```

Secondly, the concept of heapstorage could have allowed for an easier implementation of dynamic arrays, as we do not have to be worried about the stack pointer moving about. Further research suggested the heap is an OS detail and hence is not directly available in base ARM. (At least at our current knowledge).

Lastly, to reiterate on the topic of C++’s ‘constexpr’, this assignment has provided a much greater understanding of how compilers can simplify any compile time known values or segments to reduce the binary size produced and also possbily improve runtime performace.