Josh A. Bosley
Computer Science
UNAS BASIC Interpreter
Lake Superior State University

Introduction

While at Lake Superior State University in the Computer Science program I was put through the Computer Organization and Architecture class. The primary function of this class was to give us a firm understanding of computers as a logical concept, and then to get our hands dirty with assembly. My professor has developed his own reduced instruction set computer (RISC) language, simply called “Unnamed Assembly Language” or “UNAS” for short. UNAS runs on a virtual processor, and while unconventional in that the language has 60 registers (a few reserved) is a fantastic transitional language when stepping down to assembly from C++. The class was riddled with programming assignments that were either solely mathematical in nature, or stress tested our understanding of stacks and main memory. I loved programming in UNAS, and when it was time to declare a final project I knew I wanted to take my learning to the next level. We got to pick from a list of projects, and the most challenging was to write an “UNAS BASIC Interpreter.” Now, we never actually covered Interpreters, or compilers in this class, as it was not an interpreter class. This is why I viewed it as such a challenge.

The Approach

Having never attempted a project like this I decided to break down the idea into a simple logic flow, and tried to maintain simplicity as unknown variables came into the picture. The first step was to analyze the input file, and assess the necessary size of the stack. This was a simple feat as UNAS has a fixed length instruction-set, each being 4 bytes. I essentially incremented a counter upon scanning one of the following BASIC instructions: LET, PRINT, GOTO, IF, and FOR. The assumption is that each one of the instructions in BASIC would have one variable needed to be stored into the stack. Once analyzing the file yielded the memory usage, the next step was to do the full lexical analysis, and parsing. For this I used a basic Flex (lex) and Bison (yacc) setup that passed the data tied to tokens to a C++ class (NUBasic) for code generation. For each of the BASIC statements that could be encountered, there was a matching NUBasic method.

BASIC Token	NUBasic Method
LET	assignVariable()
PRINT	doPrint()
GOTO	doGoto()
IF	doIF()
FOR	doForLoop()
NEXT	doFLNext()

Some of the code generation was fairly simple as the code was essentially 1-to-1 or 1-to-2, that is to say that for every one line of some BASIC code would translate nicely to one or two lines of UNAS. Some methods on the other hand would not be so easily written. A prime example would be in the instance of declaring a new variable set to a complex mixed expression such as: LET var1 = 5 * (A – 2) – 8 + C/D. In order to preserve the rules of operations I had to convert the infix notation to postfix conversion

for the UNAS calculation to take place correctly (More on this in the section: *Infix to Postfix conversion*.) While writing each method I discovered many small tasks to keep in mind. The following is a write-up of the methods should give a clear view of the NUBasic class.

NUBasic Method Functionality

doMemoryEvaluation()

Called before parsing to evaluate input file's memory requirements. This is necessary to determine stack size required.

assignVariable()

Called to create a variable as-per statement demands. Calls private class helper **analyzeDataType()** to determine if it should send statement to:

constructString(), **constructInt()**, or **constructFloat()**

constructString()

If the input was an expression, the first thing done is string concatenation to create a new string. Once complete or if it is a single string passed in, generate an UNAS .string label, and call private method setVariable().

constructInt()

If the input was an expression, convert to postfix by use of private class helper postFix(). When completed, or if it was a single integer passed in, call private method setVariable().

constructFloat()

If the input was an expression, convert to postfix by use of private class helper postFix(). It it was a single float, or an expression a .float label is created for each new float item before the call to private method setVariable()

setVariable()

As the primary code generator, setVariable() has many duties. It determines if the current action is making a new variable, or updating an old variable. If it is updating a variable, it locates the variable's location in the stack by way of the variable structure's mi item (memory index). If it is creating a new item, it handles memory assignment based on the current memory index (cmi). Once the type of operation is determined, and the code is generated for their respective load operations, it determines if a postfix calculation is required. If a calculation is not required, the item is stored by way of stf, or stw into the necessary stack location. However, if a calculation is required, a queue is built with expression items, and code is generated to calculate the items using registers starting at r10, or f10 depending on data type by way of a stack routine (More on this in 'Calculating and translating postfix notation'). If a calculation takes place, this method will retrieve the resulting value from the stack, and create a stw, or stf operation to store it in its stack location.

doPrint()

The print method starts off by creating a variable for its input information by way of **assignVariable()**. When creating the variable, reserved words are used to ensure that other information isn't accidentally overwritten. Once the variable is created, the variable is copied into its respective print register (r50, r59, or f59) and their respective function call to print is generated. The variable created for the print operation is then deleted.

doGoto()

The **doGoto()** method creates a unique label to jump to, some variant of JBGT*N* - Where '*N*' is the a stringed value of a private counter. Once created, **doGoto()** passes its information to private class helper **insertAtIndex()** which handles the insertion of unique labels into UNAS code. The code passed to **insertAtIndex()** is stored in a vector until END is reached, at which point it inserts all labels at their appropriate place in the UNAS output code.

doIF()

If the evaluated rhs, or lhs of the relational is a variable, or if is an integer, code is generated to load the item into r5, or r6 for the UNAS equivalent of given relational, and a unique label is generated for the location in UNAS code that the statement will branch to. Once the label is generated, it and the corresponding information is sent to **insertAtIndex()** for later insertion.

doForLoop()

Create code for the given loop var, or for the loading of its existing memory index, as-well-as a unique for loop label. After loading variable information, the limit (int given after TO) is loaded into a register based on the private counter *forLoopIndex*. Theis method places its own label within UNAS code, as where the for loop is called is where it will jump to. Once the loads, and label are placed into the UNAS code **doForLoop()** is complete.

doFLNext()

Expected after a for loop, **doFLNext()** accompanies the last **doForLoop()** call seen, thus it handles the relational testing between given variable and limit and generates the code for deciding if a jump to the current for loop's label is necessary.

doRead()

Given in a variable, **doRead()** bases the input method on the current value of variable. A simple method, all **doRead()** really does is generates UNAS code for input call, and the storing of its result to the variable memory index.

> ***Note: This method was one of the last written, so it varies slightly from true BASIC usage. In NUBASIC, the variable to be read in-to must be declared prior to the statement to ensure that the correct value type can be read in.***

Infix to Postfix conversion

Translating an expressions from infix to postfix was a necessary step to easily calculate large expressions in UNAS. The translation was done via implementing a simple shuntyard algorithm. Once the postfix expression is created, the code generation process can begin.

Tracking register use in code generation

Given a postfix expression: 10 2 +

Starting at register 10 (r10, or f10) *currentRegister*

Using an integer stack *registerLocations*

Iterate through each item in expression (10, 2, +)

If the current item is not an operator, generate the code to load the item (var or int/float) into the *currentRegister*, and then push *currentRegister* to *registerLocations*, and increment *currentRegister*. Do this until an operator is found (^, *, /, +, -).

If the current item is an operator pop the last two items to rhs, and lhs integers from *registerLocations*, and decrement *currentRegister* by 2 (this is to reuse old registers so we don't burn through all registers with large expressions). Now that lhs, and rhs have been gotten from the stack, the corresponding UNAS code for the operation is generated based on int or float values and inserted into the output file.

("addf *currentRegister*, lhs, rhs"). Once the code generation is complete, push *currentRegister* onto stack as the result stored from this operation is now a value we may need to use in the next calculation. Increment *currentRegister*, and continue.

Note : UNAS does not have a supporting call for the '^' operator. In the case that this item is evaluated, the code for ^ is added to the output by way of* ***loadSupportingCode()**. Once loaded, the values of lhs, and rhs are stored in registers for the power code, and the result copied to currentRegister.*

Once each item is evaluated, the last item is popped off of the stack and stored in the stack at its memory index.

This method of handling calculations has been thoroughly tested, and can easily be hand traced to ensure accuracy.