Compiler Design

Assignment 2

Name: Richard Delaney
Number: 08479950

Design Brief
Components to Design and Implement

- *Constants*
- *Arrays*
- *For*
- *Conditional*
- *Switch Statement*
- *Chars*
- *Run-time Checking*

Constants

The brief said that constants had to be implemented into the system and I approached this problem by looking at how other objects were created and defining what should be similar or different. All objects are created in SymbolTable::NewObj method, the prototype looks like the following:

Obj* SymbolTable::NewObj (wchar_t* name, int kind, int type);

To create const, I first explored the area of having them as their own type, but I quickly disregarded this as it would require muddling up the atg with unneeded extra code. I decided to instead use a default parameter which for all existing code would make that object not constant and for any constant would set to an initial value 1.

The new Prototype looks like below

Obj* SymbolTable::NewObj (wchar_t* name, int kind, int type, int constant=0);

Thanks to this, I created an easy way to define constants in the language. The next problem was that Taste doesn't allow for in-line assignment, meaning that I had to come up with some sort of system to work out whether an Obj was a constant which was unset or a constant which was set. I decided on the following:

- Obj->constant = 1 represents an const object which has not been set
- Obj->constant = 2 represents an const object which has been set

The alteration I did to the ATG came in the VarDecl Production:

```
{
"const"             (. constant = 1;.)
 }
```

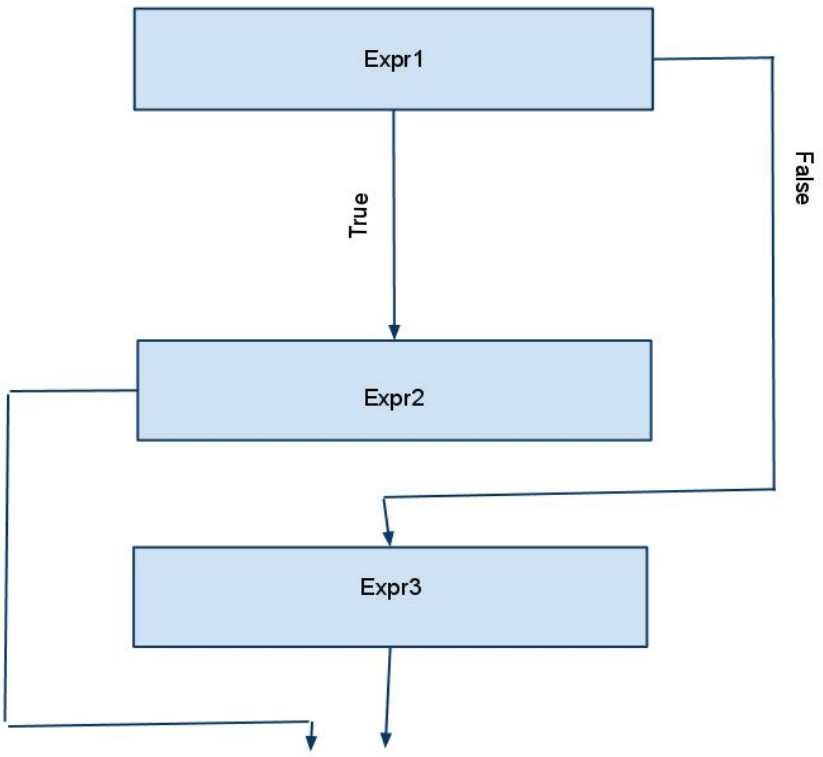
was added, and the constant value here is used in the creation of this new obj. Although this method is admittedly quite primitive, I think its effective and works very well for this example and ultimately allows for constants to be set and not applied.

Conditional

The Production I used was as follows:

<Stat> -> <Ident> '=' '(' <Expr1> ')' '?' <Expr2> ':' <Expr3> ';'

where Expr1 is of type bool, Expr2.type = Expr3.type = ident.type.
The flow control was implemented as shown below:



In words,

Type check each of the expressions.

- Evaluate Expr1
    - Jump on False to Expr3
    - Expr2 on True
- Assign the value to the object pointed to by Ident

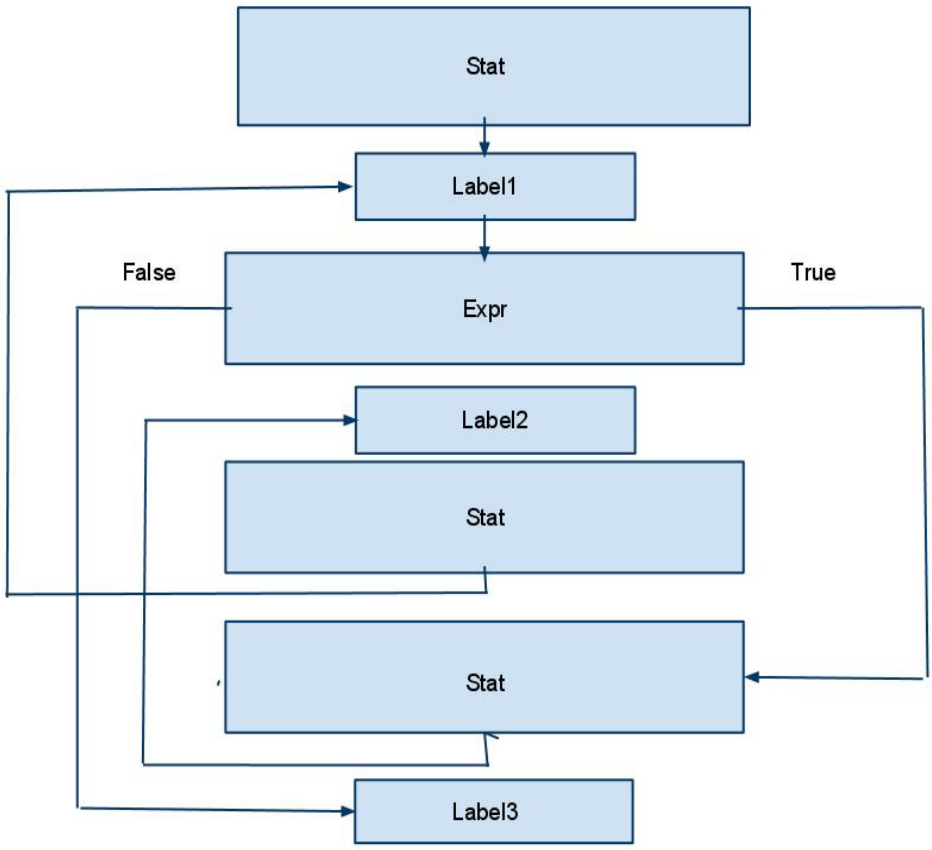
This was achieved using gen->Emit and gen->Patch in the same way as the if statement and while loops.

For

My for loop is built in the production:

Stat -> for “(“ Stat1 “;” Expr “;” Stat2 “)” Stat3

The flow of the for loop is defined below, although I struggled initially to get it to loop back up to Stat2.



This shows the flow of my for loop, and I have implemented the above in the same way using

jumps and on false jumps. I met a problem with the stat in that the for loop must end with a ; This is only a minor difference between the conventional c/++ Java For loop which I attempted to implement.

Chars

To outline how I implemented chars I'll just go through the necessary changes in the atg file:

I first added "char" as a type in the Production <Type>, I also added a new int type at the top of the atg file called character.

This dealed with making a new type char, this dealed with the variable declaration part of a new type.

Next I altered Factor, and allowed the following production:

<Factor> -> ' letter '

This allowed easily for assignments, I then moved onto character arithmetic, this proved to be a lot less trouble than I imagined as I just had to change the type checking to allow for characters and the rest of it was just implied as I was storing my characters as integers basically.

Writing chars was done using a new opcode I called "WRITEC", like "WRITE" all this does is pop off the stack and write it out as a character rather than a integer.

Using arrays with chars allows for the introduction of strings to the language.

Switch Statement

I designed the switch statement with a C/Java style switch statement in mind, I'll go through how I implemented the statement and what exactly happens to the flow of my production.

Firstly, I've restricted mine so that it only can use Ident as the switch parameter, this would be the normal operation of a switch statement in my experience. Each case statement consists of:

```
case Expr :
        Stat
        break;
```

I decided to implement my case's to have an expected break. This doesn't allow statements to roll through. The flow of the case statement took some work to get it to work the way that is expected, I've outlined the flow of the program below:

- Evaluate the switch variable,

- Push the value onto the stack,
- Push the value of the first case expression onto the stack,
- Emit an EQU opcode which will put a 1 or 0 onto the stack depending on the comparison
- On false jump to the next case statement.
- If no case was equal, go to the default case

Some issues I ran into where the break statement making each break statement jump immediately to the end, this was deceptive as patching each address meant that the case before that didn't get patched. Instead I made a label before the first break statement, Every other break statement jumps to that break statement which then jumps to the end. This is more efficient and a simpler solution than keeping track of many addresses to patch.

Arrays

I tried to implement arrays in many different ways and the one I had to settle for in the end was the only one that I could get to work to a desired level, The way I have arrays implemented now is messy and inefficient but ultimately it works in the space its given. It has a number of drawbacks and a lot of these problems are extremely difficult to overcome from what I've tried:

- For an array of size 10, 10 different objects are created
- You cannot reference or declare an arrays size using a variable, it must be a number.

These limitations stem from the fact that at compile time I need to have the size of the array, and I can't get this value from a variable at the time so I've had to go from number and just parse the number and pass that in as the size.

I'll go through the different steps on how arrays are created, referenced and interacted with:

To create an array I use the following production:

<type> <array name> [<number>]

The array name and the number are then concatenated and passed up to the NewObj Method, the NewObj Method then strips the numbers off and makes them the size of the array it then iterates over the NewObj method creating size elements with names that are the name of the array with the current element index at the end. The first element is returned from the method.

To assign, write or do any operation on an array element you simply call array_name [index] as you would in other languages. The compiler then references that object in the symbol table and performs the operations on that memory address.

What I've done to make arrays more generic and let it so that the actual array elements act just as variables act, I've included the array brackets into the definition of the Ident production, when my compiler sees this, it appends these to the name of the object and passes this up.

One thing that was glaringly wrong about my arrays when I started was the fact that the names of the array in the symbol table were actually valid identifiers for objects which means there easily could be conflicts between a integer variable count1 and element 1 of integer array count, count[1], this was resolved by giving each array a question mark suffix, this made them invalid identifiers and made it easy for the compiler to separate the two pairs.

Run-time checking of Array Bounds

Due to my primitive implementation of arrays, run-time checking of array bounds was actually rather easy to implement, If the array was out of bounds, the compiler would return an undefined object message allowing the user to find out what was wrong, I didn't think this was informative enough so instead of this, I made an array check in my SymbolTable::Find() method which when given an array element that was undeclared would produce an array index out of bounds error for the user.

Extra Things Implemented

Increment, Decrement Operators

As I was testing the other elements, I just really felt like the language was missing the increment and decrement operations found in most popular languages (++ and --), I only implemented simple one's which increment and decrement, but I just made it so that it loads a const 1 onto the stack and also loads the object address to increment and gives a simple add or sub expression.

This was a small addition but I like to have this functionality in my compiler.

Unless Statement

This unless statement was unneeded but its one feature of the perl language that I like a lot, its just a if not statement and effectively was easy to implement, but the flow was slightly different that the if statement, instead of jumping on false, I needed to jump on true so what I did was jump on false over a jump statement which jumped to the end.