CRAFT DEVELOPMENT DIARY

Daniel McCarthy

Introduction

This is a development diary of the “Craft” compiler also known previously by the name “Goblin”.

The “Craft” compiler is designed to compile “Craft” code a language which I have designed.

Unfortunately, I have only documented the project since 25th July 2016 so there is months of work undocumented. Anyhow it is titled day 1 and onwards even though it was not day 1 of the project it was day 1 of the diary.

Throughout this document you will see all the struggles and design choices I have made to get the Craft compiler completed.

Information

Before continuing it is important that you understand the names used in this diary, I will help explain the ones an experienced reader may have trouble with.

For anything else such as trouble understanding what a Lexer, Parser, virtual method, or virtual pure method is. Please research on them to understand this dairy.

Code generator

The code generator is the base class for all code generators, it parses the abstract syntax tree from the root and invokes methods in its inheritor/child depending on the branch it finds in the tree. For example, if it should find an assignment such as “x = 50” then it would invoke a method on its inheritor for handling assignments.

Goblin Bytecode generator

The Goblin bytecode generator extends the “CodeGenerator” class its methods get invoked by the code generator depending on the branch the code generator has found. Depending on the method that was invoked the goblin bytecode generator would write data to the stream or do further iteration of the AST (abstract syntax tree) until it finds what it is looking for.

Craft bytecode generator

The Goblin bytecode generator was renamed to Craft bytecode generator so throughout this diary know that they are the same.

Day 1 - 25th July 2016

This is the first day of documenting the Craft compiler. Unfortunately, I left it a bit late so I am actually writing this on the 26th July 2016 however I will write about the 25th as if it is the 25th.

Today I was working more on implementing arrays, at this point arrays were coming along well, I could only support 1 dimensional arrays although up to 3 dimensions can be parsed. I managed to make array access in expressions possible, although not perfect it worked, I was in the process of fixing a bug where array access would only work in expressions if it was on the left hand side, this was because the "A" register was used and if something else was on the left hand side of the expression, let's say a number it would occupy register "A" so the array access on the right hand side of the expression would generate byte code that would have overwritten the number at run time. I was in the process of fixing this or I did fix it I don't remember and then all of a sudden my computer crashed. When I booted back up I found my entire goblin bytecode generator NULLED. I searched my git repository for the most recent commit I could find and unfortunately the commit I found was on the 7th of July. The code that was present at that time would be completely obsolete a lot of development has been done since then so I chose to write the goblin code generator again. I rewrote a basic template for the goblin bytecode generator and I worked a little bit on the code generator improving the design.

Day 2 - 26th July 2016

Today early hours in the morning I was focusing on improving the design of the code generator, as well as implementing more code on the goblin bytecode generator since the file was NULLED yesterday due to a system crash.

The work early hours in the morning today was continued from yesterday I continued into the 26th I usually do work late.

I decided that I should modify the code generator to use a “Scope” class instead of all the functionality defined in the code generator. This made sense as it is certainly possible to have child scopes which would have caused problems in my current design. So I created a “Scope” class and this class contains variables as well as methods for working with these variables such as creating new variables or getting variables by name. A lot of the related content was moved from the code generator into the “Scope” class and modified. The “Scope” class has the ability to have one parent and one child although this will be changed later to allow for multiple children.

I also removed a lot of obsolete methods from the code generator, the old design approach was to have a virtual pure method for every possible scenario, for example I would have a method for handling a number, and let the code generator invoke a virtual pure method every time a number was found. The child class would then be able to write the machine/byte code to the stream.

This design was not appropriate and my new design approach is to have a method for only more structured elements such as functions, and assignments.

At the moment only the assignment method is implemented and was implemented a few days ago, the function method does not exist yet, in its place is the “scope_start” method. This however also needs to be removed as every scope can act differently. This will be replaced with a method such as “function_start” and this would be for function declarations including the function scope.

I also fixed the “cleanBranch” method in the Parser class. Originally it was not working or was not working as expected. I made it clean the tree of branches who only have one child starting at the branch passed as the first argument. This meant that branches with only one child are replaced with its only child, leading to a cleaner tree with information that is no longer needed removed.

I thought about the parser rules today and how I perhaps should change them, for example my expressions will have three branches the first one being the left operand, the second being the operator and the third being the right operand, this could be changed allowing the parent branch to be the operator and then its two children being the left operand and the right operand. This is how most designs look when I search for “Abstract syntax tree” on google images. There will be no definite change yet as I need to outline the pros of this change and cons if there is any.

Day 3 - 28th July 2016

Today I made the Exception class extend one of C++'s standard exceptions "logic_error". My reason for this is that should an exception be thrown and not caught I or others will be able to see the output in the console window.

I also fixed the design problems caused by the rule changes and the parser update where I fixed the "cleanBranch" method. I also made the ability to have multiple children in the Scope class which is required because scopes can have multiple scopes inside of them.

I have considered changing certain parser rules to output different types of branches than it does at the moment. Google image searches have given me many results with expressions that look like this:

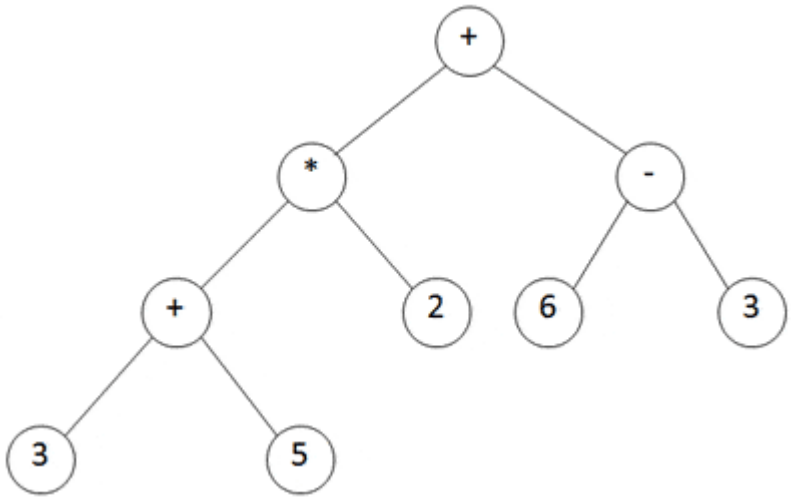


Fig 1.

My system does things a little differently and I need to consider whether or not the above image would be the best way to go about things. My tree is workable but I do believe the above image represents a better structure, the parser rules will need to be thought through carefully and not just the ones for expressions. I partially think that because I am not doing a similar structure as shown in Fig 1. May be partly the reason certain expressions cause parsing errors but I cannot be sure as of yet I am still in a bit of a grey area with this situation.

Another bug was discovered in function calls, a function call cannot have arguments that are also function calls, this is a parsing error.

In other news I attempted to update the parser to do a look ahead first before choosing a rule as the fact it does not do a look ahead has already caused me countless problems and requires rules to be placed in a pacific position to work correctly. Although today I have failed to implement the look ahead feature as the previous parser design would loop through all the rules and then apply it to every branch in the root, so in the first phase for example "3 + 3 + a" would become "E + E + E" and then as it goes to other rules would be broken down further. A look

ahead would not be possible with this design. I changed the design to apply the rules on a left to right basis without applying it to every branch at once, for example “3 + 3 + a” would become “E + 3 + a” now because there is no rule for “E + 3 + a” only “E + E” the parser will now fail to generate an abstract syntax tree, more thinking will need to be done but I plan to make the parser better before continuing the project as if I do not it may cause me even worse problems later on.

One possible solution that I would rather avoid because it might come across as a bad design perhaps, is the ability to state that a particular rule can be any of the following branches. E.g the “E” branch can be the “identifier” token or the “number” token.

Day 4 - 29th July 2016

Today I made the parser a look ahead parser which will help prevent rule conflictions, for example two rules may exist, “E:operator:E” and “E:operator:E:operator”, The second rule is a rule that you probably would never implement but I will use it as an example as it’s the perfect scenario. Now take the expression “E+E+” you would expect this expression to match the second rule but in the first parser this did not happen it would pick the first rule that it matched to so it would be “E:operator:E”. Now that the parser is changed it will now look ahead and find the most appropriate rule so it would pick “E:operator:E:operator” which is the correct rule.

Upon implementing the new Parser, I removed some code that should be their this caused an issue and prevented branches from being excluded from the tree, this was a simple bug to fix and I fixed it quickly.

Since the parser is now more efficient I no longer need the hash tag to represent functions and function calls I will be considering soon weather to remove them or keep them the same just to make the language a little different.

In other news I am considering rewriting the parser rules all together to try and do a better job, or at least rewrite some of them, also another problem was found and that is with the cleaning system for the parser. This system does exactly what it is supposed to do but in certain situations this would cause problems.

For example, the fourth branch in the “FUNC” branch holds a bunch of statement branches, now because of this cleaning system if only one statement in the function body exists then the cleaning system will replace that “STMT” branch with its child branch, I do not like this at all as obviously the function body should have some sort of root branch for its self. Anyway more thought will need to be given before any change is made.

If I do make this change, then I would set it up a bit similar to the way you exclude branches in the parser rules. In the parser rules you exclude branches from the tree by using the quote character: “ ’ ” in the current rule.

Day 5 - 30th July 2016

Today I was working on fixing a bug with input such as “c = (50 + 19) * 8” or “c = 50 + 19 * 8”. This bug is a parser bug where the parser cannot break down the branches correctly.

This issue occurred because of the parser rules and possibly the parser its self. Take these rules for example.

E:identifier

E:number

E:E:operator:E

E:’symbol@(:E:’symbol@)

ASSIGN:E:’symbol@=:E

The following happens:

*C = 50 + 19 * 8*

*E = 50 + 19 * 8*

*E = E + 19 * 8*

*ASSIGN + 19 * 8*

As you can see the parser is left with “+ 19 * 8” unparsed, this is because the ASSIGN rule matched correctly while the other rules did not. A look ahead exists but it will not work with branches who have not yet become a child because of this the parser is not working correctly.

Possible solutions

1. Change the parser rules
2. Make the parser branch look what is ahead and then see if it matches a different rule

I would rather implement solution 2. Although I am not sure if it is the best solution yet.

Priority is now changing the parser to do just this.

Day 6 - 31th July 2016

Today I decided to start rewriting the majority of the parser, I will not rewrite the rule system but the actual AST(Abstract Syntax Tree) generator. The problem with the old design was that it did not follow standard procedures to generating AST's and because of this the code was a bit ugly. I will now be using stacks instead of vectors, this did cross my mind in the past but I chose to go with the vector.

I hope to implement a shift reduce parser, properly this time.

Day 7 - 9th August 2016

Today I worked more on rewriting the parser, the development was coming along well but unfortunately I have some segmentation faults that I need to fix. The parser code is not quite right but I feel that I am nearly there. Once complete this will be the 3rd time I have wrote the parser.

Day 8 - 11th August 2016

Today not much was accomplished, I fixed a bug where I was creating a new instance of an array with a size of zero, this was because I did not check that functions existed before creating the array, e.g “new function[total_functions]” if the “total_functions” variable is zero then it will attempt to create a new function array of 0 elements, this causes issues. I believe that similar issues still exist in the system which I plan to clean out once the parser is complete.

As for the parser its self it is not coming along well at all and I am considering going back to the old working design, although not the best design it did work rather well.

I appear to be having many segmentation faults while implementing the new parser design and it is holding me back.

Day 9 - 16th August 2016

Today I successfully created a shift reduce parser that uses a stack, at the moment it does not do a look ahead and is incomplete as it also is not recursive when it comes to reductions.

I hope to soon make the parser do a look ahead and do recursions on reductions until no more reductions can be done for the current stack state.

Day 10 - 17th August 2016

Today I made the shift reduce parser recursive when it comes to reductions, this was simple to do, I simply just recall the same method when a reduction has been successful, this in turn will attempt to reduct the stack further.

I also attempted to implement the look ahead but was unsuccessful.

Day 11 - 18h August 2016

Today I was attempting to implement a shift reduce parser and during my attempts I realised that I was popping from the back of the stack and not the front, I was curious to why the branches were getting created in order and it is because of that reason.

This setup worked fine for a stack that did not need a look ahead but in a real situation a look ahead is required.

I decided to have a cigarette and thought through the best approach I could do to get this working and the solution I came up with is to make the rule requirements also a stack, and make the start of the input get popped and then pushed to the parser stack so it's a left to right parser rather than a right to left. Then finally during checking for reductions I would pop from the back of the parser stack so its right to left.

Take the following example

Rule requirements: identifier:operator:number

Now take the following input

A + 9

Now what would happen is during every reduction attempt we pop from the right most symbol, and then we pop from the right most rule requirement. So as you can see the rule requirement is "number" and the right most input is also a "number".

This is why this sort of setup seems to work.

Now as for the look ahead symbol since we are now pushing data onto the parse stack left to right instead of the previous right to left, the look ahead will always be the next symbol to the right.

I started writing code that would check if there was a rule match with the parser stack plus the look ahead symbol and if it was it would shift the look ahead symbol to the parser stack and then reduce with the rule it matched with. However, if there was no match it would then try to match a rule with the parse stack alone.

There was a serious problem with this design. I fixed this problem by making it only choose the parser stack plus the look ahead symbol reduction should there be a match for the parser stack alone and also the parser stack plus the look ahead symbol. This solved the problem and my parser could work well with a look ahead symbol, however I then set the following rules:

E:identifier

E:number

E:E:operator:E

T:E:operator:E:E

and even tried

T:E:operator:E:identifier

I may have also tried

T:identifier:operator:identifier:identifier

Once I tried those rules everything went terribly wrong, it appears the reduction of “identifier” into branch “E” and “number” into branch “E” causes issues with the current parser logic.

I am also concerned about the cleanliness of the parser and the approaches I am taking.

It appears I am back to the drawing board.

Day 12 - 6th September 2016

Today I worked on the parser and made the parser work with a look ahead properly I also made the parser reduce the correct way around rather than the previous opposite. Due to the way a stack works previously the branches were being written backwards this is fixed now. I also fixed a few bugs in the parser and partially implemented the ability of custom branches being set based on rules. E.g the newly reduced branch based on rule “V_DEF” becomes the “V_DEF” branch in memory.

I also changed the ParserRule class and made a method that returns the total requirements of its self.

Day 13 - 7th September 2016

On the 7th of September I ran into problems trying to implement an infinite operator due to the design of the parser. The infinite operator would basically say this rule applies if there is one or infinite of a particular branch.

I then realised at that point that I am going to waste months more trying to get this parser to work so I decided to ditch the rule system all together and that's exactly what I did. I started rewriting the entire parser and this time the parser would generate branches through code rather than rules.

Day 14 - 8th September 2016

On the 8th of September I attempted to implement expressions in the newly designed parser but sadly failed even after 3 or 4 attempts.

Day 15 - 9th September 2016

Today I managed to implement expressions successfully. I also began work on “IF” statements but it did not work, I forgot to invoke the “process_if_stmt()” method and will do it tomorrow.

Day 16 - 10th September 2016

Today I managed to partially implement “IF” statements in the parser, although “ELSE IF” and “ELSE” statements are currently not present.

For the parser I plan to soon make a few more methods for tasks that I believe are repeated quite frequently. This will improve design and partially increase the speed of code development.

Day 17 - 11th September 2016

Today I implemented the “ELSE” statement but it does not feel right so I will rewrite it differently. Ideally the “ELSE” statement and “ELSE IF” should nest into other “IF” statement children. Currently because I implemented “ELSE” statements I did not consider nesting, nor would their need to be, but with an “ELSE IF” setup the final “ELSE IF” should have the “ELSE” branch.

Day 18 - 13th September 2016

Today I worked on the parser and everything written for this date is parser related.

I made the “IF”, and “ELSE IF” statements, I also changed the “ELSE” statement. All these statements nest within each other.

I also fixed a bug in the parser where any keyword would be valid for a variable. You could literally type "else var_name" and the parser would register a variable declaration branch. This is completely illegal and should not be allowed.

I also fixed a bug in the parser where expressions with no valid token caused a segmentation fault. This bug fix did not go as planned and introduced another bug which I then fixed. I do not remember what the bug was.

I created methods that are designed to increase readability and faster development time, I then changed some code in the parser to make it use these methods. I decided to do this because I was writing two statements where it could have been done in one function call.

I implemented the ability to set declared variables to a value while declaring them. This allowed variables to be defined and set at the same time. For example: “uint8 x = 50;”

I also added some code to check for a semicolon after a variable declaration in body statements, this is important as otherwise we cannot guarantee there is a semicolon and the programmer would receive syntax errors that may not make sense.

I also implemented structures in the parser.

I also implemented pointers in the parser although they still do not work in the global scope yet as I want to think about the cleanest way to go about it.

I finally implemented the ability to get addresses of entities using the “&” operator. However, this currently does not work in the global scope and currently does not work when defining and setting the structure variable for example: “struct test a = &b”

Day 19 - 14th September 2016

On the 14th of September I implemented the “include” macro. This allows people to include other files in their source files. Currently it is possible to crash the program upon including the same file or including something that will lead to the same file being included again. I hope to fix this soon.

I also made the ability to load code generators from DLL files.

Day 20 - 15th September 2016

On the 15th of September I changed the design a bit and made the Craft compiler its own DLL rather than an executable file. I had to do this due to problems linking other DLL files with the compiler. Since the compiler its self is now a DLL file I have an executable file interfacing with it.

Obviously these DLL files could be Linux equivalents in the future but for now it will work with Windows only.

I also was implementing expressions in the newly created code generator called “8086CodeGen” its target code is for the 8086 processor. These expressions did not work as I did not implement the order of operations the parser.

Day 21 - 16th September 2016

On the 16th of September I made the parser prioritise multiplication and division before other forms of mathematics, this is known as the order of operations. Before this was implemented problems were present in the code generator when generating expressions, as no order of operations was acknowledged this lead to different results than you would get from a calculator.

Day 22 - 17th September 2016

On the 17th of September I was writing assembly language in an 8086 emulator to try and find the best code design for functions, this was proven difficult as a lot of variables come into play. For example, a function may have many variables but they may get assigned to either scope variables, argument variables or global variables. The way to access each one of these is completely different from each other. I did not work out the best way to do this on the 17th as it passed midnight.

Day 23 - 18th September 2016

Today early in the morning continuing from last night, I managed to successfully handle expressions, assignments and functions in the 8086 code generator. I spent a long time compiling code in other compilers and reading the assembly output to try and understand it. The “8cc” compiler assisted me as well as an online C, C++ compiler that pacifically shows you assembly output.

Eventually after studying these compilers I managed to implement a design of my own, although things are done a bit differently which I am not yet sure if it will be a problem in the future.

Should I want to access an argument variable of my own function I access these variables in 16 bit words relative to the base pointer. The offset is “+4” for argument one, “+6” for argument two and so on.

Should I wish to access a scope variable it’s the opposite way, I minus the base pointer starting at offset “-2”. So for example scope variable “a” our first variable would be located at “bp-2” and our second variable “b” would be at “bp-4”.

Global variable access is currently not permitted inside a scope. Functionality for this will be added shortly.

Here is example code as well as assembly offset of early this morning’s accomplishment

```
uint8 main(uint8 a, uint8 test)
{
  uint8 b;
  b = 10;

  uint8 c;
  b = (50 * b);
  return b;
}
```

Assembly output

```
_main:
push bp
mov bp, sp
mov ax, 10
mov [bp-2], ax
mov ax, 50
mov bx, [bp-2]
mul bx
mov [bp-2], ax
mov ax, [bp-2]
pop bp
ret
```

In other news while writing code for the code generator I wrote code that could turn a positive to a minus and I forgot that this was legal in most programming languages, Take the expression

```
-(50 + 10)
```

The result of that expression should be “-60” the parser currently requires that all operators have a left and right operand, I will need to tweak this design as I would have had to anyway due to the Boolean operator “!” used to turn a false to a true or a true to a false. I will attempt to implement this in the parser and code generator around the same time I implement the Boolean operator “!”

Day 24 - 19th September 2016

I do not remember exactly what I done on the 19th of September but I know it was either the 19th or 20th that I successfully implemented function calls.

I have decided that I am going to write diary entries as soon as possible for now on, to prevent forgetting important information.

Day 25 - 20th September 2016

On the 19th or 20th of September 2016 I successfully implemented function calls in the code generator. Any function arguments are pushed to the stack before the function is called.

I also started working on making pointers and accessing memory with these pointers. I did not manage to finish this on the 20th as it passed midnight.

Day 26 — 21st September 2016

On the 21st of September I was just still trying to implement pointers.

Day 27 – 23rd September 2016

19:33:

I managed to implement pointer access in the parser this allows one to do things like “return *a;”, “*a = 50;” and more.

While implementing this pointer access I realised that the current design of the parser especially in the expressions must be improved and by the looks of it I do not believe it is compatible with structure access. I hope to improve this and implement structure access in expressions.

20:44:

I implemented structure access in the parser, it turns out the current parser design is actually compatible with structure access. The design of the parser is not too bad but improvements could be made in the future.

Day 28 – 24th September 2016

03:18:

In the 8086 code generator I implemented the ability to access a value that a pointer is pointing to and I also implemented the ability to set the value of the memory a pointer is pointing to.

I have still not made the appropriate classes for some of the branches created for this process.

I am also considering soon making a method for setting registers and the 8086 code generator will store these registers in memory so should a register be set to the exact same thing again it will ignore it, although more thought needs to be taken as if I am not careful I may not reset it during appropriate instructions resulting in problems. It may be best to just finish the code generator and make these amendments later on.

Source code

```
uint8 test(uint8* a)
{
  uint8 c;
  *a = *a + 5;
  return *a;
}

uint8 main(uint8 a)
{
  uint8 b;
  b = test(&a) + 2;
  return b;
}
```

Assembly output:

```
_test:
push bp
mov bp, sp
mov bx, [bp+4]
push bx
mov bx, [bp+4]
```

```
mov ax, [bx]
pop bx
mov cx, 5
add ax, cx
mov [bx], ax
push bx
mov bx, [bp+4]
mov ax, [bx]
pop bx
pop bp
ret
_main:
push bp
mov bp, sp
mov ax, bp
add ax, 4
push ax
call _test
add sp, 2
mov cx, 2
add ax, cx
mov [bp-4], ax
mov ax, [bp-4]
pop bp
ret
```

Day 29 – 25th September 2016

23:09:

In the 8086 code generator I have been trying to implement compare expressions in the compiler as these are required before “if” statements will be possible.

I managed to make compare expressions possible such as “a == 5” this will set register “AX” to “1” if variable “a” is equal to five or “0” if variable “a” is not equal to five. “1” can be expressed as “true” and “0” can be expressed as “false”.

Source code

```
uint8 main(uint8 a)
{
  a = 5;
  a = a == 5;
  return a;
}
```

Assembly code

```
_main:
push bp
mov bp, sp
mov ax, 5
mov [bp+4], ax
mov ax, [bp+4]
mov cx, 5
cmp ax, cx
jne _0
mov ax, 1
jmp _1
_0:
mov ax, 0
_1:
```

mov [bp+4], ax

mov ax, [bp+4]

pop bp

ret

Currently multiple compare expressions in sequence are not implemented, e.g “a == 5 && b == 6”.

There is a slight problem with the order of operations as it does not prioritise the order of operations for these compare operators. This leads expressions such as “a > 5 + 6” to check if variable “a” is above “5” and then add “6” this causes the compiler to generate pointless instructions that are overwritten at the end of an assignment.

I hope to allow multiple compare expressions in a sequence and fix the order of operations soon so that I can move onto developing the “if” statement.

Finally, a bug was found, the ability to declare a variable and set it at the same time is currently not possible: “uint8 b = 8;” I probably just forgot to add this functionality and will add it soon.

Day 30 – 28th September 2016

20:48:

Today I have found a bug in the parser where parsing pointers causes the expressions that use the multiplication symbol to cause an infinite loop.

The solution may be to keep track of pointer variables in the parser and let that determine how the tree is generated.

20:56

I have decided to just use the "@" symbol to represent pointer access for now as this will fix the problems I am facing. I do intend to eventually change it back to the "*" operator and fix this issue properly. I believe that similar issues will exist with the "&" operator as this can be used as a bitwise "&" therefore it is relevant for expressions so this too will be changed to a question mark symbol for now "?". Again I do intend to fix both of these issues in the future.

21:27

Successfully implemented compare expressions in the parser.

22:32

I have successfully implemented compare expressions in the 8086 code generator, the only problem which has now been fixed appeared to be the use of the ">", "<", ">=" and the "<=" operations. It did not perform well when it was its opposite so ">" would check if it is below it did not like that at all. I have since made them and only them use the correct instructions for their operators although I believe this will cause problems in the future when multiple compare expressions are present, for example "a > 4 && b < 6"

Day 31 – 29th September 2016

05:23:

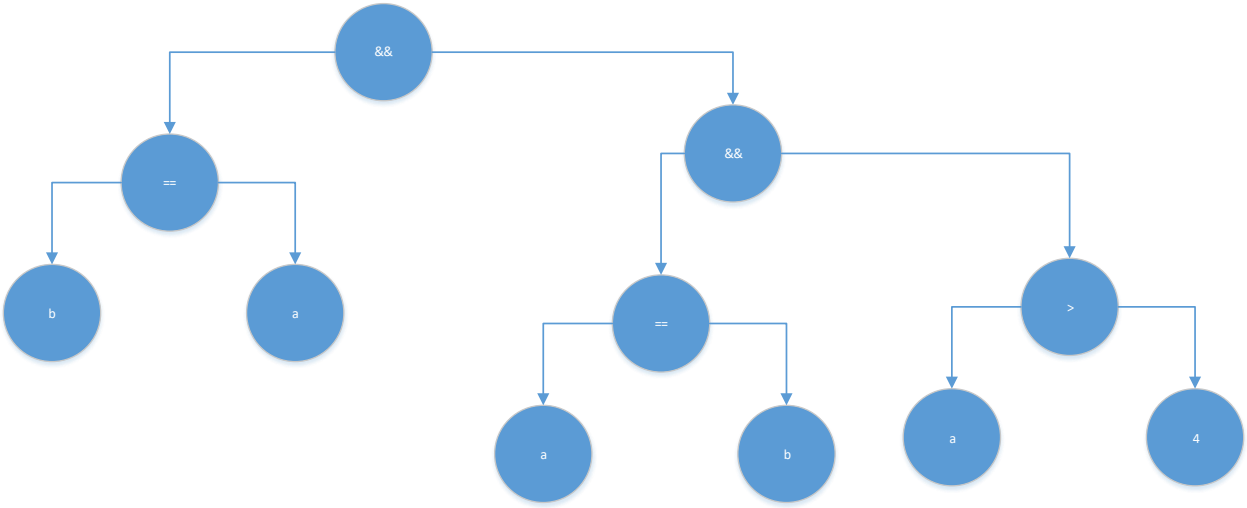
In the parser I have successfully implemented the ability for multiple conditions in an expression e.g “a == b && c > 1”.

I ran into many problems implementing this and it lead to me having to rename the “process_expression” method to “process_expression_part”. I then created a new “process_expression” method that would call “process_expression_part” if it needs to.

In the “process_expression” method the system first calls the “process_expression_part” method after it has returned it will check if the next token is a logic operator if it is then it will then need to invoke the “process_expression_part” method again and then put it together creating the “E” branch with the value being that of the logical operator. This new “E” branch will contain the children of the left and right expressions.

Take the following expression “a == b && c > 1” the left expression is “a == b” and the right expression is “c > 1”.

Graphical representation: b == a && a == b && a > 4



Day 32 – 1st October 2016

18:58:

Fixed a bug in the 8086 code generator where function calls did not push function parameters on backwards, before this patch function parameters would be accessed the wrong way around from within a function body.

The other possible solution would be to change how the function body accesses data passed to the function.

The reason pushing function parameters on backwards fixed the problem is because of how the 8086 stack works.

The “gcc” compiler also pushes function parameters on backwards to achieve the same thing.

19:31

In the 8086 code generator I implemented the ability to declare variables and assign them at the same time, previously this was not possible. You can now do “uint8 a = 50;"

21:23

In the 8086 code generator the ability for multiple conditions in an expression e.g “a == b && c == 1” is now possible. However, the expression “a == b || c == 1” is currently not possible.

In other news due to the fact that multiple expressions cause a “push” and a “pop” to happen should the statement be “false” the value will never get popped back off the stack which could be a problem but I don’t know yet.

23:41

It turns out the “push” and “pop” statements generated talked about at “21:23” has become a problem, functions will not return to the correct address in memory due to the old value not being popped off. I plan to fix this soon; at the moment I have just commented them out but they are required or another alternative will need to be factored.

23:48

Multiple compare expressions in the 8086 code generator with use of the “||” logical operator are now possible. This means both logical operators “&&” and “||” are implemented.

23:56

Fixed a bug in the 8086 code generator, the bug was where I forgot to turn the compare expression flag off once it was complete. I also changed a bit of code and tided things up a bit.

Day 33 – 2nd October 2016

00:25:

I have just fixed a bug in the 8086 code generator where multiple expressions would always default to the “add” instruction. This is wrong as an expression such as “(80 + 20) * (20 + 40)” would act as if it was “(80 + 20) + (20 + 40)”.

00:34

I have fixed a bug that was introduced to the previous bug fix at “00:25”. Essentially the “make_math_instruction” method should not be called for logical operators.

00:42

I have fixed a bug where “push” and “pop” instructions were generated during compare expressions. This was explained in detail yesterday.

03:22

In the 8086 code generator I forgot to fully implement all of the possible compare expressions correctly this is fixed now.

03:46

I managed to implement “IF” statements in the 8086 code generator, currently “ELSE” and “ELSE IF” statements are not implemented.

Day 34 – 4th October 2016

01:43:

Successfully implemented “ELSE IF” statements in both the parser and the 8086 code generator. The “ELSE IF” statements do not use a branch class of their own as they are no different from “IF” statements property wise.

02:01:

Successfully implemented “ELSE” statements in both the parser and the 8086 code generator. The “IF” statement, “ELSE” and the “ELSE IF” statements were fairly easy to implement as the hard part was compiling an expression, the expression is a part of these statements but does not define them.

02:10:

Currently while providing negative values in “IF” statements the Lexer does not notice them, they do become a “number” token but it is positive. I need to fix this and perhaps look into making expressions default to signed numbers.

Day 35 – 7th October 2016

03:18:

Successfully implemented pointer assignments in the 8086 code generator, you can now do something such as “*ptr = 50;” to set the memory that the variable “ptr” is pointing to.

03:29:

I have realised a few weeks ago that in the parser accessing a pointer inside an expression can be a problem as how does the compiler know if you are multiplying or not, e.g “a = b *a;” Now imagine variable “a” being a pointer this should be a syntax error as “*a” is used to access the value that variable “a” is pointing at.

My point is the parser does not know if it is doing a multiplication or accessing a pointer value, the solution is to keep a variable or pointer list in the parser to describe the variables it is accessing. Should the variable it is accessing be a pointer then we know for sure we are accessing its value.

05:52

My previous diary entry at “03:29” was incorrect. I have successfully made the system work without needing to check if a variable is a pointer, I will remove the variable map and methods created with it shortly.

Pointers in expressions currently do not work as the assembly language generated will not get the value pointed to by an address stored in a particular register. I will get onto this soon and fix it.