Author:	**Elliot Carter**
Project Supervisor:	**Elizabeth Scott**
Project Title:	**Computer Language Design and Engineering**
Year:	**2020**
Course Module:	**Full Unit Project**

Final Report

Final word count: 46911

Aims and Objectives

My aim for this project is to develop a compiler which is capable of compiling a small subset of the C language. I intend to mimic the C language and its semantics as closely as possible.

Initially I plan for my C subset to support the following:

- Variable declaration
    - Initially this will only support integer types
- Evaluation of boolean expressions
    - This will the full set of equality and inequality checks which my current grammar includes
- Evaluation of arithmetic expression
    - Initially this will include addition and subtraction
- Flow control via conditional if statements and while loops
    - If statements are planned to include ‘else’ and ‘else if’ initially.

The set for functionality above should be enough to provide some interesting demonstrations of the compiler’s capabilities.

Further extensions include:

- Multiplication
- Bracketed expressions
- Printing
- Bitwise operators
- Comments
- Functions
- For loops and switch statements
    - break & continue statements
- Character and floating point types
- Constants
- Pointers
- Arrays & Strings
- Division / modulus

In the event that all of the above are implemented I could also look into adding structs and type definitions

Final word count: 46911

In this document I aim to explain my process of the development of my C compiler. I will begin by walking the reader through all of the necessary theory required to understand the approach I have taken in its development up to writing my initial set of grammars for the C language.This includes; a detailed explanation of Context Free Grammars and how they relate to software languages and parsing, an introduction to parsing in general and further details on LL(1) parsers and the restrictions placed on grammars which they permit, and a walkthrough of rdp, the language processor generation tool I am using to build the parser for this project. Aspects of rdp which are covered include; use of rdp's source language IBNF to generate parsers, implementing semantic actions and use of symbols tables. I will walk the reader through the design and implementation details of my initial set of grammars in order to demonstrate how my solution has evolved alongside my knowledge of the domain.

In the sections that follow, the document will switch to a heavy focus on code generation. This will include both the generation of intermediate language representations and how they can be used to generate machine code, and why this is the preferred approach to program compilation. I will begin with an introduction to intermediate representations before switching to a focus on three address code specifically as it is my chosen intermediate representation. I will also give details on how rdp can be used to generate three address code to prepare the reader for the following sections describing the furthered development of my grammar after switching to a strategy involving intermediate code generation.

I will give a brief introduction to code generation and the primary tasks associated with it before presenting my options for machine architectures my compiler could target and my reasons for choosing mips. I will follow this with an introduction to the mips assembly language in order to prepare the reader for the sections that follow, which will primarily focus on the design and implementation of my code generator and the steps I took to improving it.

Finally I will go into the particulars of my implementation of any specific language features not covered in previous iterations of the project's development. In particular this will cover my approach to developing functions, explaining the various components necessary to implement functions, why I designed them the way I did and exactly how my implementation works.

The document will close with my thoughts on how the project went, where I would have liked to take it next with regards to particular features, and what I would improve about the project.

Note to the Reader

It is assumed that the reader has a basic understanding of graphs and how a depth first traversal of a graph is performed. It is also assumed the reader has a least basic competency of the C programming language. It is also assumed the reader has at least a rudimentary understanding of assembly languages, and how programs are written in them. Knowledge of specific machine architectures is not required, and all other theory involved in this project is to be understood over the course of reading this document.

Final word count: 46911

Contents

Final word count: 46911

Final word count: 46911

A Brief Overview of Language Translation and Grammars

Before delving directly into context free grammars, I'd like to give a brief overview of grammars in general and the overall goal of language translation.

When translating a language, we begin by identifying individual words. From strings of words we form phrases and from phrases we derive meaning. There are three key features we can identify from the previously described process.

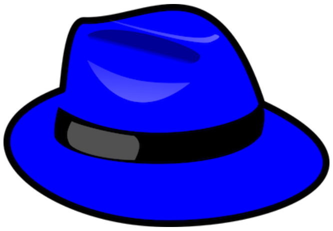
Vocabulary - The set of individual words for a given language

Grammar - The set of rules which determines how individual words can be combined to form legal phrases for a given language.

Semantics - An assigned or established meaning, perhaps derived from a set of rules.[1]

These three key features are consistent across both natural and computer languages.

As an example, let's take the simple phrase, *"Her blue hat"*, which clearly describes something similar to this image.



Looking at this sentence we can identify that the individual words are all part of the English vocabulary, and that the grammatical structure of the phrase is as follows:

[Possessive] [Adjective] [Noun]

We could take the same phrase written in another language, *"Son chapeau bleu"*. The words are taken the French vocabulary and the grammatical structure of the phrase is now:

[Possessive] [Noun] [Adjective]

Despite these differences however, the exact same meaning can be derived. This example also identifies one of the problems of language translation, in that we can not simply take the vocabularies of each language and substitute words individually.

In this case, if we were to do this by going from the French phrase to the English phrase, we would form the phrase "Her hat blue", which by the grammatical rules of the English language is an invalid sentence and subsequently holds no meaning.

Final word count: 46911

A similar case can be made for computer languages. Take the following below:

C: **Int** x[] = {4, 7, 3, 2, 9};

Java: **Int**[] x = {4, 7, 3, 2, 9};

In both cases we are forming an array of the same five integers, in the exact same order, in their respective languages. In both cases this will result in the allocation of five contiguous 32 bit words in memory which will store the values above in the order they are written.

Again, the semantics of these two statements are identical, and indeed in this case even the individual symbols are the same. The only difference here is the ordering, where in the case of C we have:

[Keyword] [Identifier] [Operator]

Whereas in the case of Java we have:

[Keyword] [Operator] [Identifier]

As with natural languages, computer languages are governed by a strict set of grammatical rules which govern what constitutes a legal statement, and indeed if we were to attempt to use Java like array initialization in C, the code simply would not compile.

Final word count: 46911

What is Context Sensitivity?

In the previous section we touched on how the ordering of words presents a challenge in language translation, and how this holds true for both natural and computer languages. However a far more complex problem faced in natural language translation, is the problem of context sensitivity.

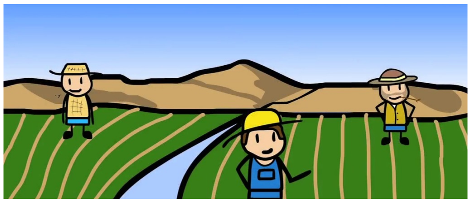
Take the following example

“...land beside the river…”

At first glance we may be led to believe a pilot is being instructed on where to land their helicopter.

“Try to land beside the river, so we may perform a fast extraction.”

And this would be perfectly reasonable. However without the surrounding context there is no guarantee that this was the intended meaning.



“You’ll find fertile land beside the river, where you can plant your crops.”

Same phrase, different context, entirely different meaning. This is what is meant by context sensitivity. It is when the meaning of a phrase is dependent on the wider context. This is a problem which is not only difficult to solve algorithmically, but also adds complexity to the translation algorithm, making it slower. For this reason, context sensitivity is typically avoided (although not entirely) in computer languages, and context-free grammars are used for computer language definitions.

Final word count: 46911

Understanding Context Free Grammars

Formally, a context free grammar is a 4-tuple (N, T, S, P) where

- **N** is a finite set of non terminal symbols
- **T** is a finite set of terminal symbols
- **S** is the start symbol and is an element of **N**
- **P** is a set of production rules of the form $A \rightarrow \alpha$, where $A$is a single non terminal symbol and $\alpha$is a string of terminals and non terminals. Also note that α can be empty.[2]

To define this more generally, think of **P**, the set of production rules, as your set grammatical rules which define legal sentences in your language.

Think of the set **T** as your vocabulary and **N** as your set of grammatical terms (noun, verb, preposition etc...)

Think of **S** as “sentence”, as in what everything combined eventually forms.

Terminals

Terminals are the basic symbols from which strings are formed, and are synonymous with the term “token”.

In the English language, your terminals would be grammatical classes (noun, verb, etc.). These grammatical classes can be matched with lexemes, which are the words in the English vocabulary, thus forming phrases and eventually sentences.

In a programming language, your terminals would be:

- Keywords (if, then, else, while, for, etc..)
- Identifiers
  - not the names of the identifiers themselves, but rather some token which matches with legal identifier names. The names themselves would be lexemes eg (x, var, _val, y32, etc...).
- Operators (+, -, *, /, =, &&, ||, >>, <<, etc...)
- String Literals
  - Again, not the literals themselves but rather some token which matches with the literals. The literals themselves would be lexemes eg. (3.141, 127, “some string”, etc…).

Terminals are symbols which appear in the outputs of the productions rules (the αin $A \rightarrow \alpha$) and they can not be rewritten using the rules of the grammar.

Non Terminals

Non terminals are syntactic variables that denote sets of strings.

Using the English language as an example, at the lowest level we would have our grammatical classes (noun, verb, adjective etc..), these are terminals which can be matched directly with words.

ADJ = adjective, DET = determiner, PREP = preposition

So our set of terminals is **T** = {NOUN, VERB, ADJ, DET, PREP}

On a higher level we have non terminals which can describe strings of terminals and other non terminals.

NP = noun phrase, PP = prepositional phrase

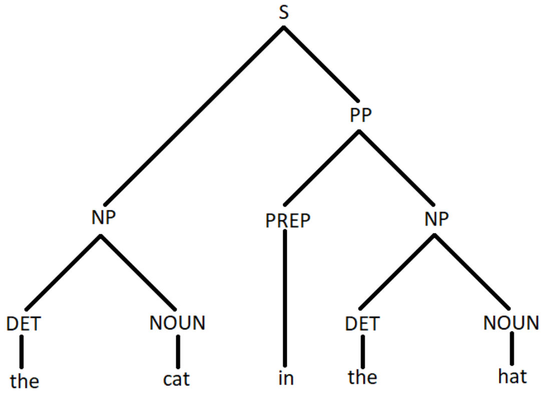
$$PP \ \rightarrow PREPOSITION \ NP$$
$$NP \ \rightarrow DETERMINER \ NOUN$$

At the highest level we have the start symbol S, which is itself a non terminal.

S = sentence

$$S \rightarrow NP \ PP$$

So our set **N** of non terminals is **N** = {S, PP, NP}



"The cat in the hat."

As the start symbol, the set of strings it denotes is the language generated by the grammar. In the case of the English language, **S** denotes all possible English sentences.[3]

Final word count: 46911

Production Rules

The production rules of a grammar specify the manner in which terminals and non terminals can be combined to form strings. Productions take the form $A \rightarrow \alpha$

- A is a non terminal, referred to as the “head” or “left side” of the production. This is the variable which denotes sets of strings
- α, referred to as the “body” or “left side” of the production, is a string of zero or more terminals and non terminals. The components of the body describe one way in which the head of the production can be constructed
- →is simply notation. :: =is sometimes used in place of the arrow.

I’ve already shown examples of production rules in the section describing non terminals however it would be prudent to work through a simpler example to illustrate exactly how production rules can be applied to form a grammar.[4]

The following grammar describes the syntax of simple arithmetic expressions

$T\ =\ \{0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ +,\ -\ \}$
$N\ =\ \{list,\ digit\}$
$S\ =\ list$
$P\ =\{$

$list \rightarrow list\ +\ digit$ **1**
$list \rightarrow list\ -\ digit$ **2**
$list \rightarrow digit$ **3**
$digit \rightarrow 0\,|\,1\,|\,2\,|\,3\,|\,4\,|\,5\,|\,6\,|\,7\,|\,8\,|\,9$ **4**

$\}$

Derivation

Now let’s take the expression $9\ -\ 5\ +\ 2$. Using the grammar described above, we can derive this expression by beginning with the start symbol and repeatedly replacing a non terminal by the body of a production for that non terminal.

Going from left to right as you normally would with arithmetic expression

1. 9 is a $list$ by production **3** **(**$list \rightarrow digit$**)**, since 9 is a $digit$
2. 9 - 5 is a $list$ by production **2** **(**$list \rightarrow list\ -\ digit$**)**, since 9 is a $list$and 5 is a $digit$
3. 9 - 5 + 3 is a $list$ by production **1** **(**$list \rightarrow list\ +\ digit$**)**, since 9 - 5 is a$list$and 2 is a$digit$ [5]

Final word count: 46911

To give a more precise definition, we can say that beginning with the start symbol, each rewriting step replaces a nonterminal by the body of one of its production rules. Specifically this derivational view corresponds to top down parsing, which is something we will discuss in the coming sections. For now our aim is to understand derivations in general, and the associated notation.

Each time we use a rule to perform a rewrite, we are performing a “derivation step”. Let us review once more our expression and grammar.[6]

$list \rightarrow list\ +\ digit$ **1**
$list \rightarrow list\ -\ digit$ **2**
$list \rightarrow digit$ **3**
$digit \rightarrow 0\,|\,1\,|\,2\,|\,3\,|\,4\,|\,5\,|\,6\,|\,7\,|\,8\,|\,9$ **4**

9 - 5 + 3

We begin with our start symbol $list$. From our start symbol we derive $list\ +\ digit$ from our first rule. This is our first derivation step and is denoted by the following:

$$list \Rightarrow list\ +\ digit$$

Our next derivation step will use the rule $list \rightarrow list\ -\ digit$ to replace the leftmost non terminal ‘$list$’. Note that each time we write out the whole derivation thus far.

$$list \Rightarrow list\ +\ digit \Rightarrow list\ -\ digit\ +\ digit$$

Next we replace our leftmost non terminal using the rule $list \rightarrow digit$**.**

$$list \Rightarrow list\ +\ digit \Rightarrow list\ -\ digit\ +\ digit \Rightarrow digit\ -\ digit\ +\ digit$$

We produce our first terminal symbol using the rule $digit \rightarrow 9$ to replace the leftmost non terminal.

$$list \Rightarrow list\ +\ digit \Rightarrow list\ -\ digit\ +\ digit \Rightarrow digit\ -\ digit\ +\ digit \\ \Rightarrow 9\ -\ digit\ +\ digit$$

And then $digit \rightarrow 5$for our next leftmost non terminal

$$list \Rightarrow list\ +\ digit \Rightarrow list\ -\ digit\ +\ digit \Rightarrow digit\ -\ digit\ +\ digit \\ \Rightarrow 9\ -\ digit\ +\ digit \Rightarrow 9\ -\ 5 + digit$$

Final word count: 46911

And then finally $digit \rightarrow 3$for our last remaining no terminal.

$$list \Rightarrow list + digit \Rightarrow list - digit + digit \Rightarrow digit - digit + digit$$
$$\Rightarrow 9 - digit + digit \Rightarrow 9 - 5 + digit \Rightarrow 9 - 5 + 3$$

We call such a sequence of replacements a derivation of $9 - 5 + 3$ from $list$. For a formal definition of derivation consider a non terminal A in the middle of a sequence of grammar symbols, as in $\alpha A \beta$where α and $\beta$are arbitrary strings of grammar symbols. Suppose $A \rightarrow \gamma$is a production rule, then we write $\alpha A \beta \Rightarrow \alpha\gamma\beta$as our derivation step. The symbol $\Rightarrow$means “derives in one step”. When a sequence of derivation steps $\alpha_1 \Rightarrow \alpha_2 \dots \Rightarrow \alpha_n$rewrites $\alpha_1$to α_n, we say α_1 derives α_n. We can say this as “derives in zero or more step”, and for this purpose we use the symbol $\Rightarrow^*$. Thus,

$\alpha \Rightarrow^* \alpha$ *for any string* α, *and*
If $\alpha \Rightarrow^* \beta$ *and* $\beta \Rightarrow^* \gamma$, *then* $\alpha \Rightarrow^* \gamma$

Likewise we can use $\Rightarrow^+$to say “derives in one or more steps”.[7]

In a typical case when dealing with programming languages, if a string can not be derived from the start symbol it would cause a syntax error. Remember that goal is to understand how these grammars can be applied in software language translation, or more specifically, compiling.

Parsing is one of the key stages of compilation. It is necessary to do so in order to verify that source code has been written correctly as defined by the language’s grammar. Additionally, parsing is necessary not just to verify grammatical integrity, but also to infer the semantic actions described by the code, so that these actions can be executed by the cpu. For now however we will focus on grammatical verification, and talk more on semantic evaluation later

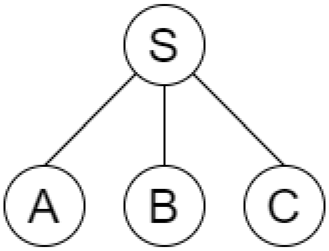
Final word count: 46911

Parse Trees

In the previous section I mentioned how the process of deriving the expression from the provided grammar was an example of parsing, and although the method used may suffice for simply illustrating an example, if we wish to perform these steps computationally, we are going to need something more rigorous and robust.

Parse trees, or derivation trees, are a graphical representation of how the start symbol of a grammar might derive a string in its language.

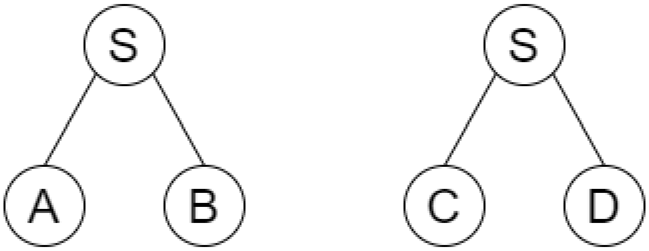
Take the following grammar $S \rightarrow ABC$, which can only result in the following parse tree.



We can see how for each of the associated symbols, a child node is connected to the start node S. Notice also how from left to right the order of the nodes in the tree matches the order in which the symbols appear in the grammar, this is important because the order in which the derivation is performed matters, particularly when dealing with the associated semantic actions a topic which again, I will cover in more detail later. Note also that the structure of a parse tree should always be as follows:

- The root of the tree is always the start symbol for the grammar.
- All leaf nodes should either contain a terminal or ε.
- All interior nodes should contain non-terminals.[8]

Let's take another example $S \ \rightarrow AB \mid CD$. Now we are dealing with two alternates, which means the grammar can form two possible trees.

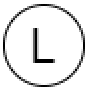


Final word count: 46911

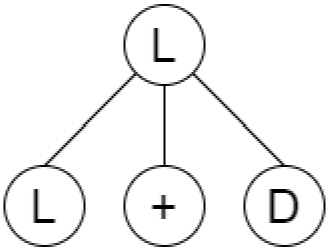
In practice these two trees would likely represent two separate derivations, of two separate input strings, under the same grammar. Note that I explicitly said this is likely however, not necessarily the case, a fact which I will explore in a later section.

Finally, to provide a more concrete example, let’s take the expression in from the previous section and perform a graphical derivation of it.

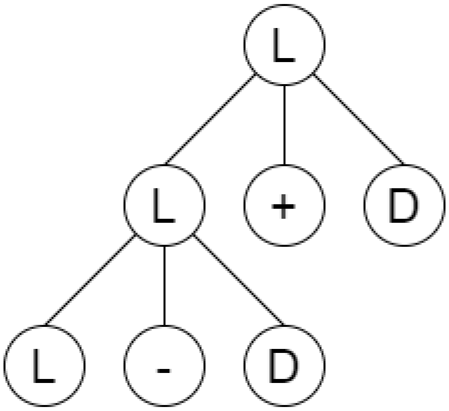
We begin by building a node for the start symbol $List$, which in the tree I will represent as a node containing the letter L.



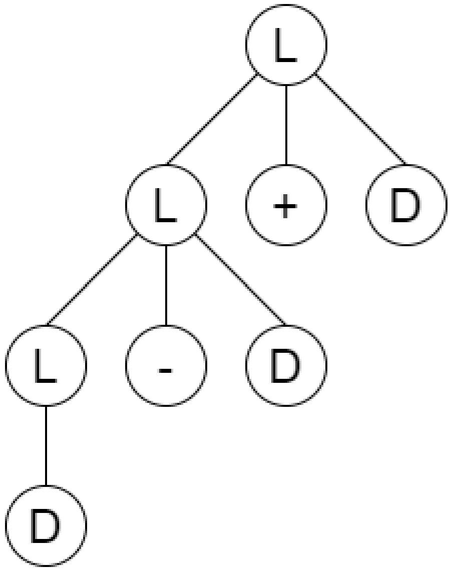
Next we take the rule $List \rightarrow List\ +\ Digit$, which for reasons which will become clear in a moment, will allow us to match the rest of the expression. Also note that I am using nodes containing the letter D to represent $Digit$.



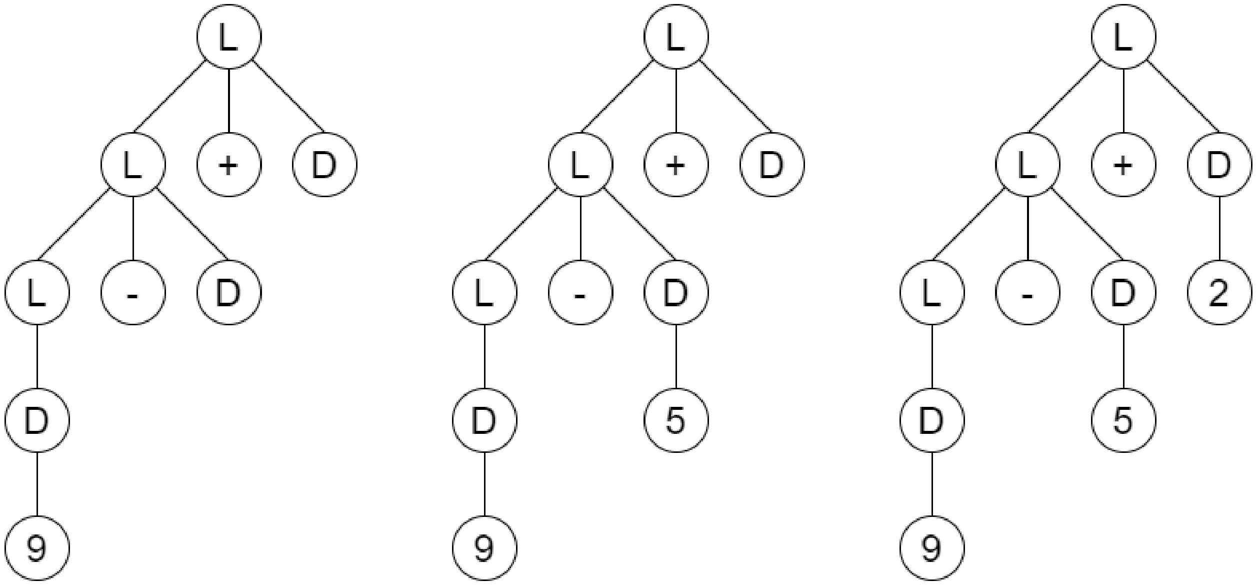
We follow by matching the leftmost leaf node containing L with the rule $List \rightarrow List\ -\ Digit$.



We then follow by matching the resulting leftmost leaf node containing L with the rule $List \rightarrow Digit$.



Finally we can complete the derivation by matching integers 9, 5 and 2 from left to right with their respective leaf nodes containing D.



Final word count: 46911

Thus the derivation is complete. There is however a point of discussion I wish to raise involving the nature of this derivation.

If you recall the steps which were taken to perform the derivation of this expression originally, you will notice that they were almost the reverse of what was performed to build this tree. This is because two different parsing methods were applied to perform the derivation for each example.

For the first example, a bottom up parsing approach was applied. In this approach, we started with the leaf node containing the terminal 9, and worked our way up to the start symbol.

To build the tree, a top down parsing approach was applied. In this approach, as illustrated, we begin with the start symbol and work our way down to the terminals.

In practice, there are reasons why one might implement either technique. To put it simply, top down parsers typically perform derivations faster and are easier to implement, however without backtracking these parsers are quite limited in the grammars which they can permit. Bottom up parsers require more setup and are harder to implement, however they permit a larger set of grammars.

From this point onwards however, I will largely be focusing on top down parsing, and specifically recursive descent parsing, as it is the parsing technique which is directly related to my implementation.From a practical perspective, particularly with respect to recursive descent parsing, trees are an ideal data structure for representing the process of parsing a string. Think of the nature of function calls and how they create branches in a program's structure. As you traverse the arcs of a parse tree, you can see how the interior nodes representing non-terminals could be interpreted as function calls, and how the leaf nodes representing terminals, could be interpreted as logic within the bodies of those functions. I will explore this idea more later when speaking about the tools which implement these parsers.

Ambiguity

Up until now all the derivations which have been performed were by hand and as humans, we are able to intuitively identify the correct path in order to successfully derive a given input string. Specifically, we are able to make choices, choices which a finite state automata, such as a parser, may be incapable of making.

I would like to preface this with the fact that there do exist parsers which can deal with most of the types of ambiguity I intend to discuss, however these parsers fall outside the scope of this document. Furthermore, ambiguity is one of the fundamental problems associated with parsing and in particular, with respect to top down parsing without backtracking, falls at the crux of why the set of possible grammars which they permit is so limited. With that being said, we will first look a type of ambiguity which effects all parsers.

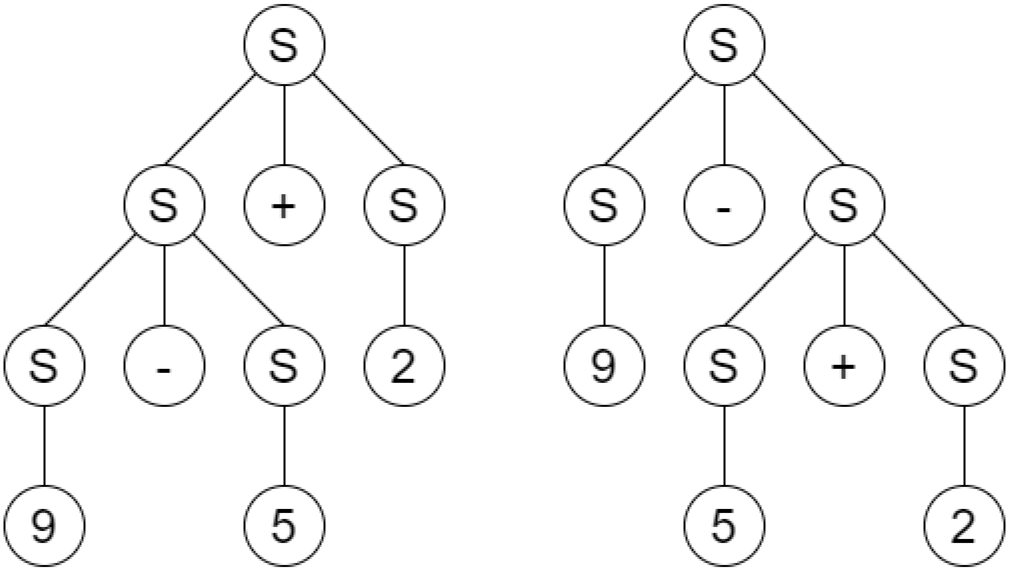
We have a grammar:

$$S \rightarrow S + S \mid S - S$$
$$S \rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

And a numerical expression:

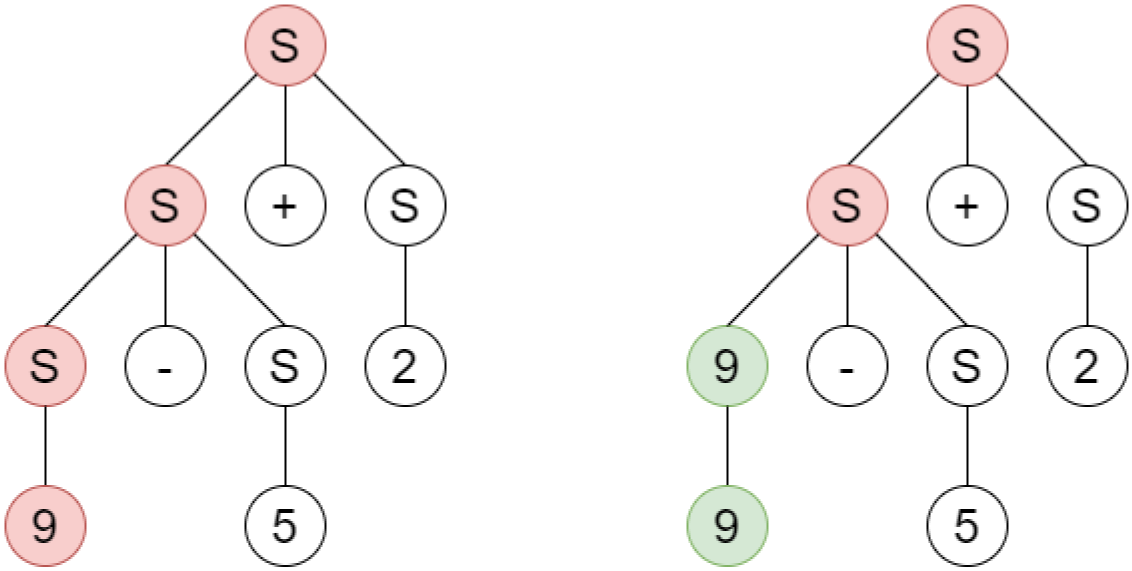
$$9 - 5 + 2$$

Now observe the following two derivation trees.

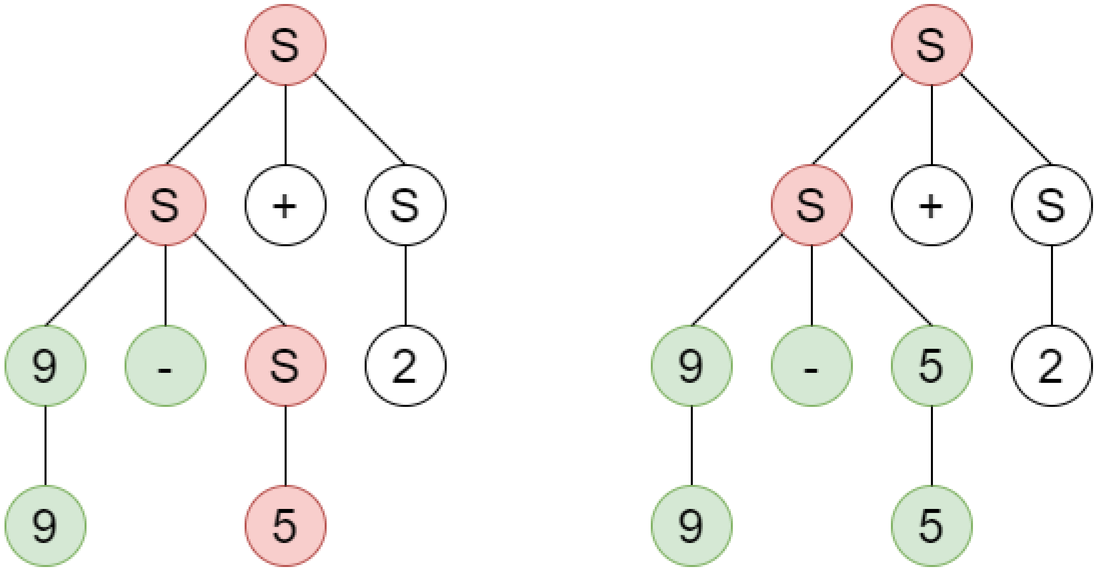


Both of these derivations are perfectly legal according to the grammar which means it is possible to derive the same string in more than one way. This would not be a problem if our only goal was to verify the validity of the string, however when we involve the semantics, we run into a problem.

In order to evaluate each expression represented by these trees I will perform a depth first traversal through each where I will only perform numerical operations after all the child nodes of a parent have been visited. Observe the following:

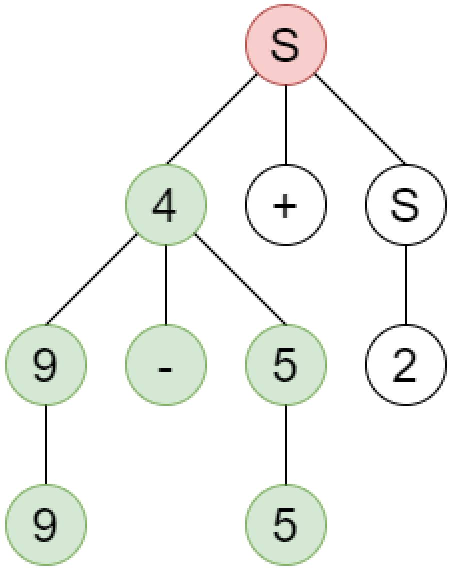


As you might expect, we begin by traversing each leftmost node until we reach a leaf (as depicted by the tree on the left). Since the node containing 9 is its parent's only child, we immediately assign this value to the parent node (as depicted by the tree on the right).

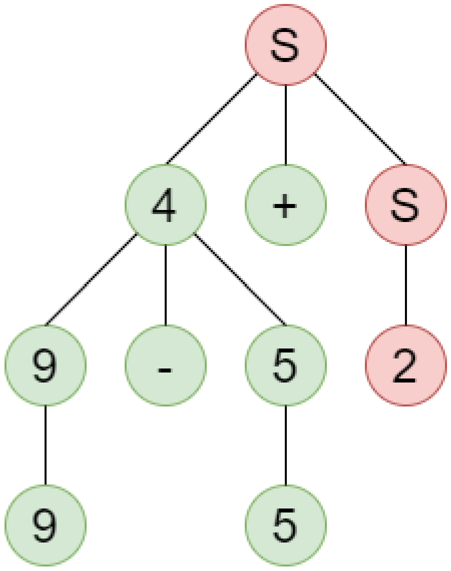


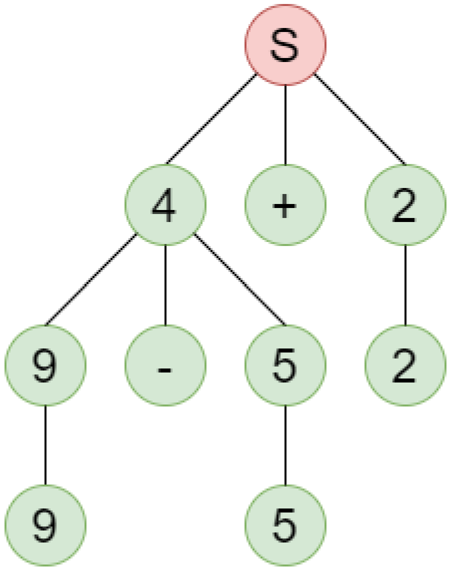
We continue traversing the tree, visiting the node containing the minus and the node containing the 5. Once again the value is immediately assigned to its parent node.

Final word count: 46911



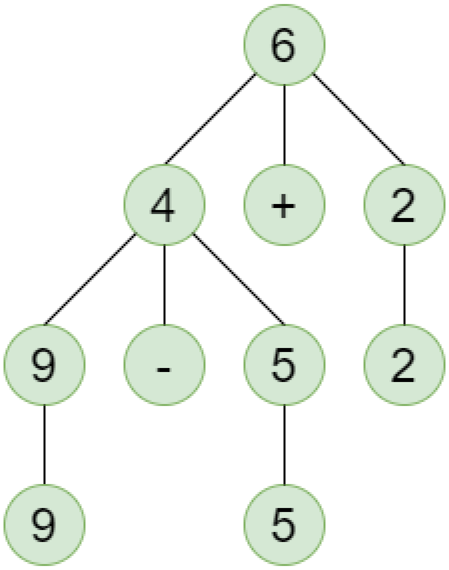
Now that the entire left side has been visited, we evaluate the expression 9 - 5, and assign the resulting value, 4, to the parent node.



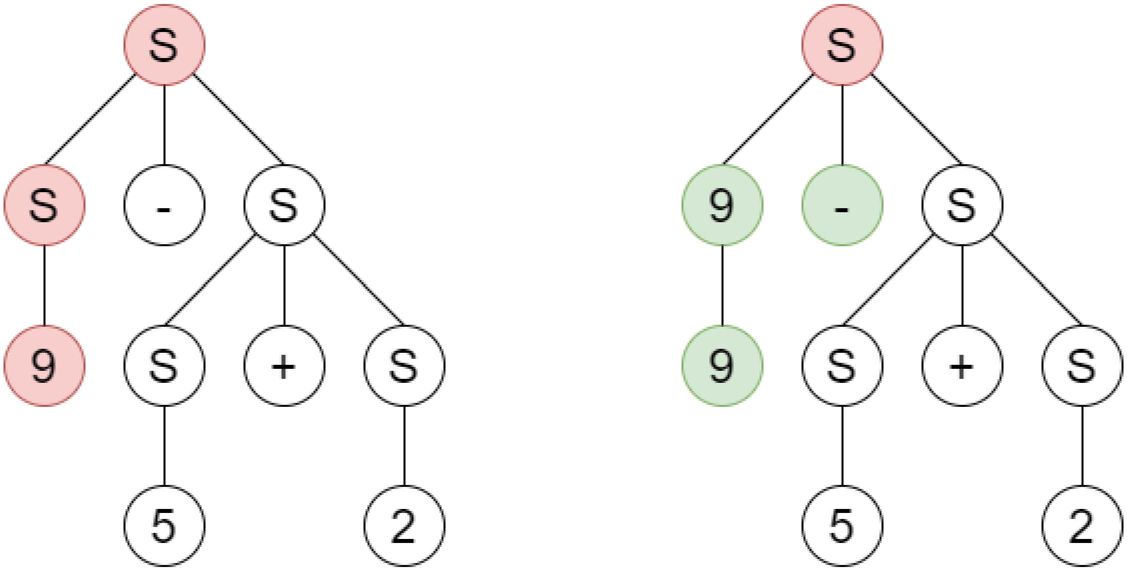


We continue down the right side as we did with the left side, visiting each node and immediately assigning 2 to the parent node.

Final word count: 46911

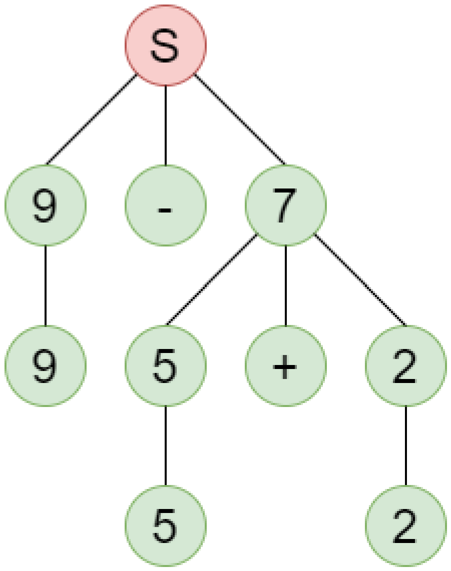


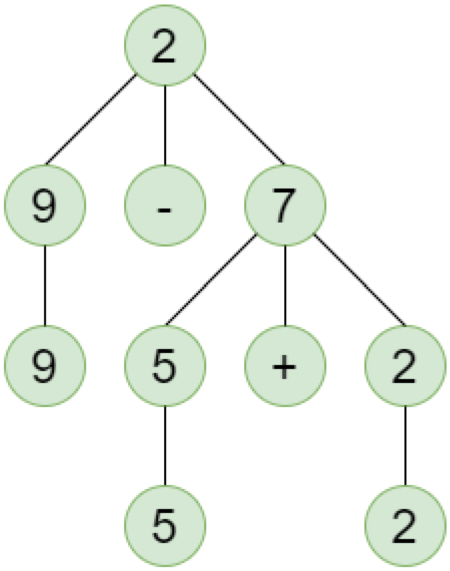
Now we've finally arrived back at the start node after visiting every other node and we can finally perform the final evaluation and assign the expected result of 6 to the start node. Now this might all seem fine, but let's observe what happens when we apply the same process with the other tree.



This time the entire left side traversal results in the value 9, instead of 4 as we might expect.

Final word count: 46911





Now if we traverse the entire right side, it results in the value 7, instead of 2, making the final calculation 9 - 7. This is incorrect according to the normal semantics of arithmetic expressions.

The plus and minus operators are left associative, meaning that the evaluation should be performed from left to right as if the left side were bracketed. This however was only the case for the first tree and in fact for the second the exact opposite occurred, where the operators were treated as if they were right associative, resulting in an incorrect evaluation.

This example serves to demonstrate the fact that if a grammar is not structured correctly to match the intended semantics of the language, this problem will occur regardless of the limitations of the parser, assuming the parser is capable of handling this grammar at all. Note also, that this problem is not limited to the evaluation of numerical expressions.

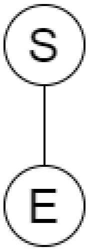
`Final word count: 46911`

Reducing Ambiguity

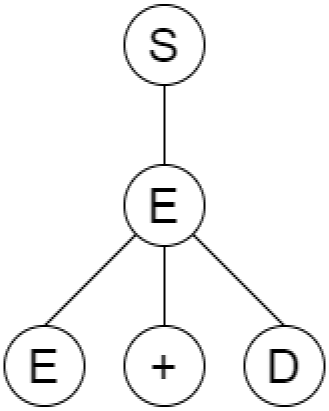
Now examine the following grammar.

$S \rightarrow E$
$E \rightarrow E + D \mid E - D \mid D$
$D \rightarrow 1 \mid 2 \mid 3 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Now let's observe how the tree would be built for the same expression with this grammar.

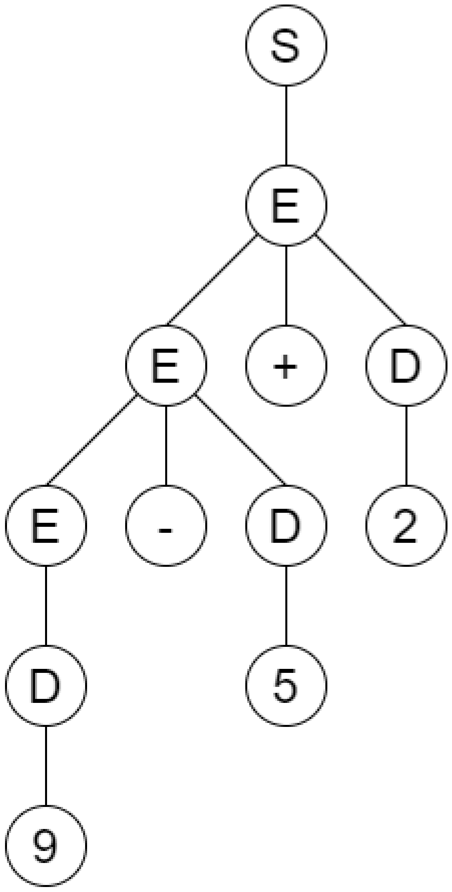


As usual, we begin by building a node with the start symbol, and we follow by building a node containing E, since there are no other choices to be made.



We continue by using the rule $E \rightarrow E + D$ to derive the tree above. We can not use $E \rightarrow D$ because it would not allow use to match more than a single digit, and we can not use $E \rightarrow E - D$, because there would be no way to place the addition on the correct side of the expression after doing so.

Final word count: 46911



It is no longer necessary that I step through each point of the derivation process, as it is clear from the previous step that with this grammar, it is no longer possible to produce more than a single tree with this input string. It also just so happens that this tree would correctly evaluate the expression if the same method as before were applied. This is less so the point however (although not entirely irrelevant), but more so the fact that with this grammar, there is only one possible way to interpret this expression, meaning regardless of the semantics associated with it, there would only ever be one outcome. Indeed, it has been made unambiguous, at least in this regard.[9]

Final word count: 46911

Top Down Parsing

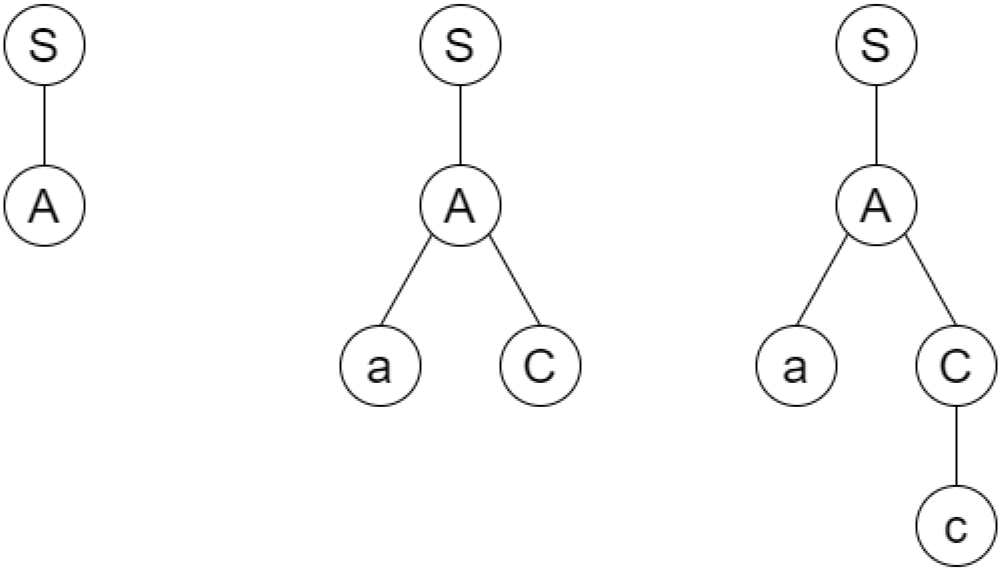
As previously mentioned, a top down parser works by first starting at the start symbol of a grammar and gradually producing the set of terminals matching the input string.

This process involves the construction of a tree from top to bottom where for each node containing a non-terminal symbol, an alternate for that symbol is chosen, which in turn produces child nodes, which themselves may either contain terminal or non-terminal symbols. The process is complete when each leaf of the tree contains a terminal symbol, indicating that the input string has been produced.

Here is an example of this:

$S \ \rightarrow A$
$A \rightarrow$aB | aC
$B \ \rightarrow b$
$C \rightarrow c$

$Input\ String: ac$

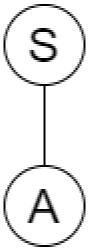


1. We begin with the start symbol S and produce a child node containing A with the only alternate of S, $S \ \rightarrow A$.
2. We have a choice between the two alternates of A at this point. Our aim is to produce the string ‘ac’, so we choose $A \rightarrow aC$.
3. At this final step of the derivation we simply take the only remaining optio $C \rightarrow c$**,** which ends up producing the desired string, ‘ac’.

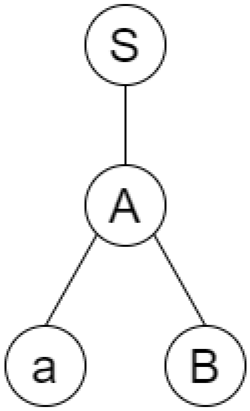
If a parse of an input string which is not permissible by the grammar is attempted, there will come a point in the tree's construction where no alternate of any production rule can be applied to continue the process, meaning it has failed.

Given the same grammar, let's attempt to parse the following input string:

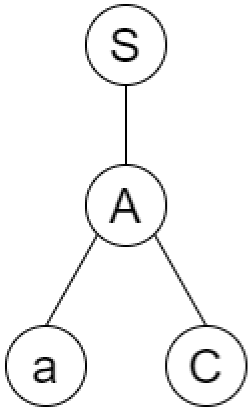
$Input\ String: ad$



1. We begin in much the same manner as before.



2. Now we run into our first problem where we select the alternate $A \rightarrow aB$. Although this accounts for the 'a', from this point we will be unable to match the 'd'.



3. We run into the same problem using $A \rightarrow aC$, where once again the 'a' is accounted for but the 'd' can not be matched.

After attempting both to produce the desired string with both possible alternates, we have run out of choices, indicating that this is not a valid sentence within this grammar, thus failing the parse.

Besides demonstrating the process of a failed parse, this example also highlights another issue which we've encountered previously but somewhat glossed over. The element of choice was presented here, where since $A \rightarrow aB$ would not produce the desired result, $A \rightarrow aC$was attempted instead.

As humans we are able to make these choices purely by observing the grammar as a whole and making an intuitive decision on which alternates to choose immediately. A machine may not have this luxury, and instead may have to make individual observations of all possible paths before reaching a decision on whether the parse can continue or if it must fail. For this form of decision making to be performed, it may not just require trying each alternate of a single

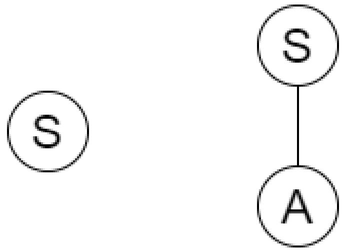
Final word count: 46911

production, but the alternates of previous productions also, and this means performing some form of backtracking. Here is an example to more clearly demonstrate this idea.

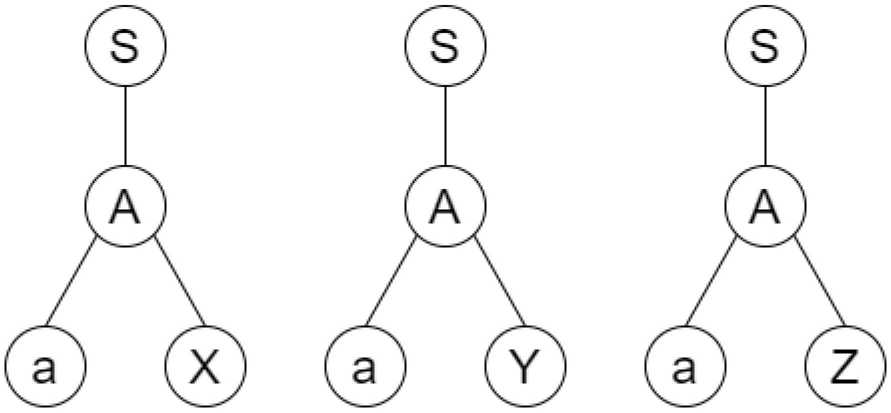
$S \rightarrow A \mid B$
$A \rightarrow$ aX | aY | aZ
$B \rightarrow bX \mid bY \mid bZ$
$X \rightarrow x$
$Y \rightarrow y$
$Z \rightarrow z$

$Input\ String\colon\ bz$

Once again as humans faced with such a simple grammar and a short input string, we can clearly see the set of steps necessary to produce the desired result, but let's observe how a machine might perform the same task.

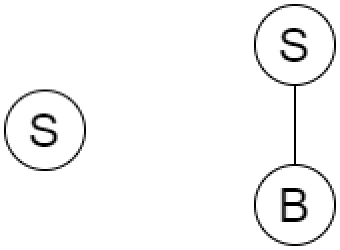


As usual we begin with the start symbol, and thereafter, are immediately presented with a choice of either $S \rightarrow A$ or $S \rightarrow B$. Since we are a methodical machine, we will simply work our way through these alternates from left to right until we either produce the desired string, or exhaust all possible routes and conclude the task to be impossible.

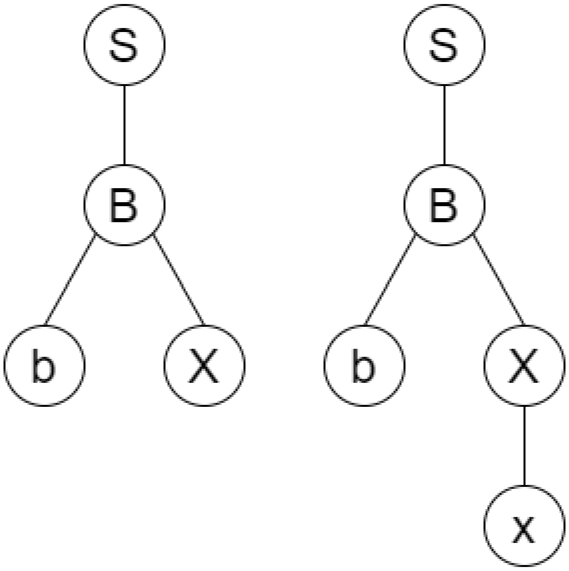


Continuing with our task, we try all possible alternatives of A in turn before concluding that it will not be possible to match the character 'b' given our previous step. Thus, we must backtrack.

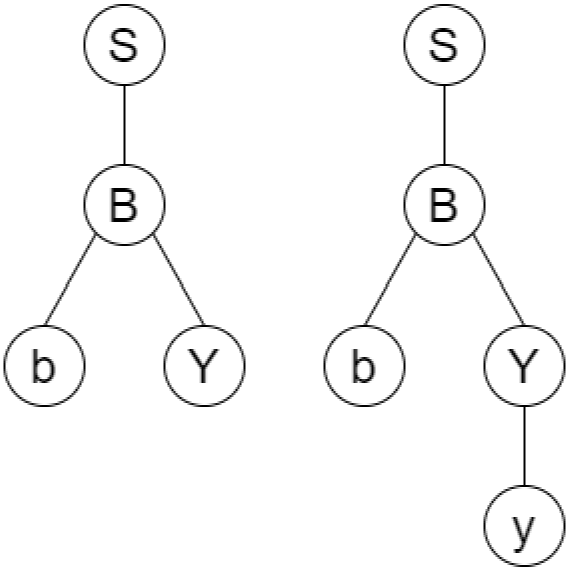
Final word count: 46911



We have now backtracked into a previous state and are now ready to attempt the possible choices presented via the second alternate of S, $S \to B$.

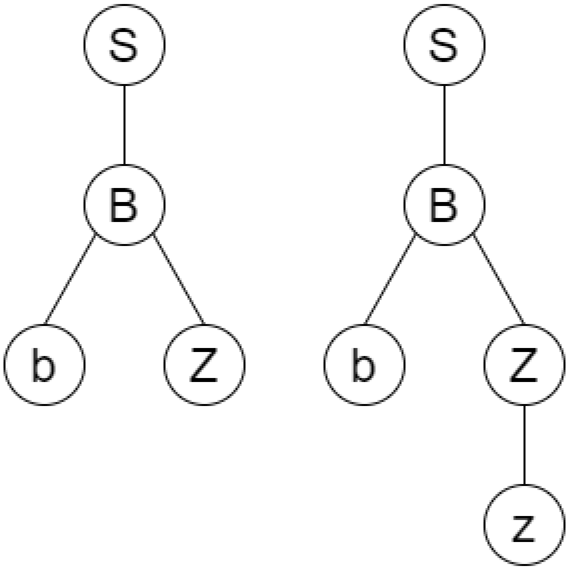


We have finally found a choice which matches the ‘b’, however the step which follows does not produce the desired result, thus the next alternate of B must be checked.



Again the ‘b’ is matched but the followup step does not produce the desired result.

Final word count: 46911



The last alternate of B is checked, and it appears the entire string can be produced in this way, thus the parse has been completed successfully.

To be clear, this is a perfectly acceptable method of parsing a string, and indeed in the end we arrived at the correct result, however this required a lot of processing, resulting in a much higher time complexity for this algorithm.

I am gradually moving towards how we can parse strings in linear time, however first I will demonstrate a lot of the processing in the previous example could have been avoided.

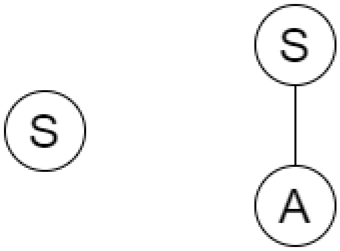
$S \rightarrow A \mid B$
$A \rightarrow$ aX | aY | aZ
$B \rightarrow bX \mid bY \mid bZ$
$X \rightarrow x$
$Y \rightarrow y$
$Z \rightarrow z$

If we examine this grammar once more, we can identify a clear pattern in the alternates of A and B.

$A \rightarrow$ aX | aY | aZ

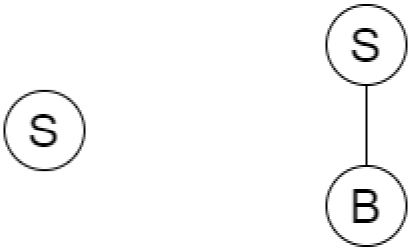
We can observe that for A the first terminal symbol which appears in each of its alternates is the character ‘a’. This is an important fact because it means we have a method of grouping these alternates in a way which can determine whether or not we need to check any of them at all.

Let’s step through the same parsing process as before, only this time we will take this fact into account

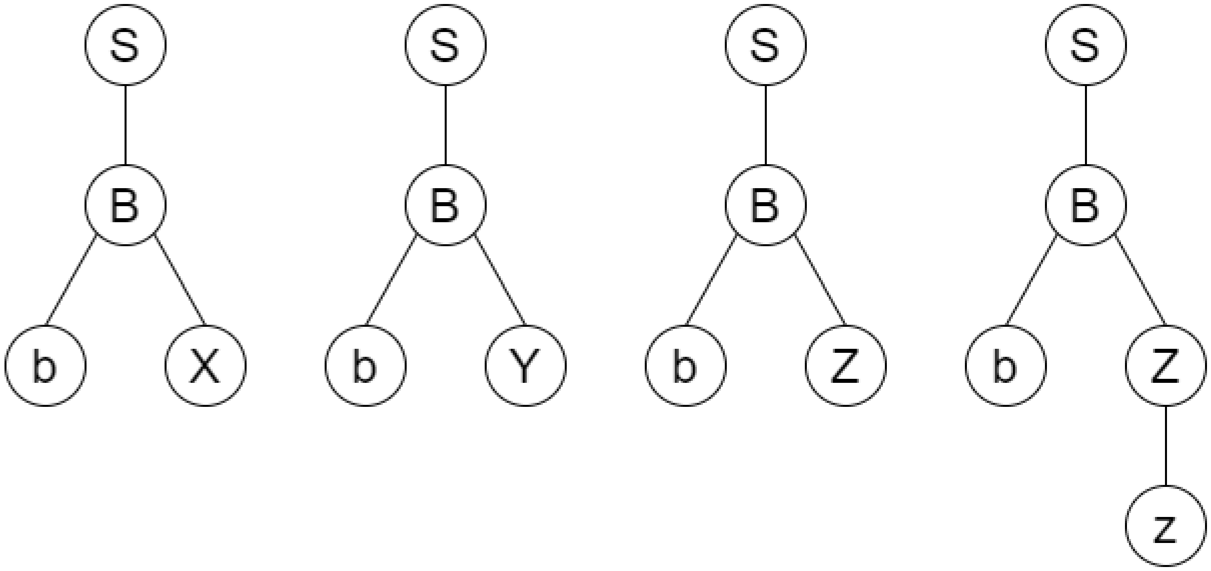


We begin the process in the exact same way as before, checking the alternates of S in turn, only this time we know not to check the individual alternates of A because we know ahead of time that the only character we could possibly match against first if is ‘a’ and the first character we intend to match against is ‘b’.

$B \to bX \mid bY \mid bZ$



Having immediately concluded that $S \to A$ has no chance of producing the desired result, we immediately turn to the second alternate of S, $S \to B$. As with A, we have the handy fact of knowing each of the alternates of B begins with a ‘b’, thus we know to check them.



Additional to avoiding the checks associated with $S \to A$, we also avoid explicitly checking

Final word count: 46911

against the terminals ‘x’ and ‘y’. This is because we know ahead of time that the first symbols we’ll have to match against if we were to explore these non-terminals are ‘x’ and ‘y’ respectively. Thus it is enough to simply cycle through each of the alternates of B before we find $B \rightarrow bZ$, which forces us to check $Z \rightarrow z$ which ends up producing the desired result.

With this simple concept, a lot of the additional processing was avoided. This is a concept known as the FIRST set, which I will explore in more detail in the next section.

First Sets

Formally $First(\alpha)$ is given by the following definition.

Where G is a grammar and $\alpha$is any string of terminals and non terminals within G:

$$First(\alpha) \ = \ \{t,\ where\ t\ is\ a\ terminal\ and\ \alpha \Rightarrow^{*} t\delta\}$$

If $\alpha \Rightarrow^{*} \varepsilon$ then we also include ε in $First(\alpha)$, also $First(\varepsilon) \ = \ \{\varepsilon\}$[10]

As demonstrated by the previous section, FIRST sets are used to guide the choice of which alternates are taken into consideration when parsing a string. We know only to consider any alternate if and only if the character we are currently matching appears in the first set of that alternate and if it does not, we can simply ignore it.

We can compute FIRST sets by the following:

```
FIRST_OF_A = $\emptyset$
For each alternate $\alpha$in a production A:
    A = nullable
    For each symbol s in $\alpha$:
        If s is a non terminal:
            If s is nullable:
                FIRST_OF_A += (FIRST(s)/{$\epsilon$})
            Else:
                FIRST_OF_A += FIRST(s)
                A != nullable
                break
        Else if s is a terminal:
            FIRST_OF_A += s
            A != nullable
            Break
    If A is nullable:
        FIRST_OF_A += $\varepsilon$
```

If you were to write a program which required the use of FIRST sets, the above pseudo code would be useful in order to guide your implementation, however when writing your own grammars, it is particularly useful to be able to recognise what the FIRST sets for your alternates are from pure observation, since being aware of them beforehand, will help to guide the structure of your grammar, and help you avoid any conflicts which you may have run into had you not been aware of this (conflicts which I will identify in a later section). Below is a method more suitable for calculating first sets by hand.

$First(\alpha\delta) \;=\; \{\alpha\}$
$First(\varepsilon) \;=\; \{\varepsilon\}$

$If\ A\ is\ a\ non\ terminal\ then\ for\ any\ \delta$:

$If\ A \overset{*}{\Rightarrow} \varepsilon,\ First(A\delta) \;=\; (First(A)\backslash\{\varepsilon\}) \;\cup\; First(\delta)$
$Else\ \ First(A\delta) \;=\; First(A)$

$If\ A\ \rightarrow \alpha \mid \beta \mid \gamma\ then,\ First(A) \;=\; First(\alpha) \;\cup\; First(\beta) \;\cup\; First(\gamma)$

Here is an example of how you might apply this:

Here is a grammar:

$S\ \rightarrow\ aBA \mid BB \mid Bd$
$A\ \rightarrow Ad \mid d$
$B\ \rightarrow\ \varepsilon$[11]

We’ll start from the bottom and work our way up, as it is usually easier this way, since the rules at the top tend to make use of the rules beneath them.

For the rule $B\ \rightarrow\ \varepsilon$, we only have to worry about a single alternate. Recall that $First(\varepsilon) \;=\; \{\varepsilon\}$. This is the only rule which we must apply here and thus:

$First(B) \;=\; First(\varepsilon) \;=\; \{\varepsilon\}$

For the rule $A\ \rightarrow Ad \mid d$,we must make a union of multiple alternates, Recall that $If\ A\ \rightarrow \alpha \mid \beta \mid \gamma\ then,\ First(A) \;=\; First(\alpha) \;\cup\; First(\beta) \;\cup\; First(\gamma)$. We can approach calculating the First set of this production rule in much the same way:

$First(A) \;=\; First(Ad) \;\cup\; First(d)$

For the first alternate, $A \rightarrow Ad$ recall:

$If\ A \overset{*}{\Rightarrow} ε,\ First(Aδ)\ =\ (First(A)\backslash\{ε\})\ \cup First(δ)$
$Else\ \ First(Aδ)\ =\ First(A)$

Since A is not nullable, $First(Ad)\ =\ First(A)$, however since we are calculating the FIRST set of A, $First(A)$ won't add anything new, since we'd essentially be adding a set to itself, which would result in the same set, thus we can ignore this term.

For the second alternate, $A \rightarrow d$ recall $First(αδ)\ =\ \{α\}$. Although there is no followup delta in this case, the same rule may still be applied, therefore:

$First(d)\ =\ \{d\}$

Therefore:

$First(A)\ =\ \{d\}$

Finally, for the rule $S\ \rightarrow\ aBA \mid BB \mid Bd$, we get the following:

$First(S)\ =\ First(aBA)\ \cup\ First(BB)\ \cup\ First(Bd)$

For the first alternate, $S\ \rightarrow\ aBA$, again we make use of $First(αδ)\ =\ \{α\}$, where this string begins with a terminal, thus we need not take anything further than the first symbol into account.

$First(aBA)\ =\ first(a)\ =\ \{a\}$

For the first alternate, $S\ \rightarrow\ BB$, recall once again:

$If\ A \overset{*}{\Rightarrow} ε,\ First(Aδ)\ =\ (First(A)\backslash\{ε\})\ \cup First(δ)$
$Else\ \ First(Aδ)\ =\ First(A)$

Because B is nullable, we do the following:

$First(BB)\ =\ (First(B)\backslash\{ε\})\ \cup\ First(B)$

Now, because $First(B)\ =\ \{ε\}$this makes $(First(B)\backslash\{ε\})\ =\ \emptyset$, so we can ignore this term,which leaves us with:

$First(BB)\ =\ First(B)\ =\ \{e\}$

Final word count: 46911

Finally, for the last alternate $S \rightarrow Bd$, we get:

$First(Bd) = (First(B) \backslash \{\varepsilon\}) \cup First(d)$

Once again $(First(B) \backslash \{\varepsilon\}) = \emptyset$, so we can simply ignore, which leaves us with

$First(S) = First(d) = \{d\}$

Therefore:

$First(S) = \{a\} \cup \{\varepsilon\} \cup \{d\} = \{a, d, \varepsilon\}$

Note: placing the epsilon at the end is a personal choice, the order doesn't actually matter.

Besides reducing the time complexity of our parsing algorithm, FIRST sets can also be used as a tool to help us restrict our grammars, the method by and reasons for why I shall explore in the coming section.

LL(1) Parsers

An LL parser is a top-down parser which parses input from left to right, performing leftmost derivations of the sentence.[12] This means an LL parser would parse the string "abc", by matching the characters in order from 'a' to 'c'.

A tree beginning with the start symbol of the grammar is built, and the production rules of the grammar are applied in order to extend the tree, until a set of leaf nodes containing tokens which match the input string are produced. Whilst building the tree, the leftmost nodes containing nonterminals are always expanded first, hence why the derivation is leftmost. There are several examples of this process in previous sections.

An LL parser is called an LL(k) parser if it uses k tokens of lookahead whilst parsing the input. This means the parser is able to use k consecutive tokens from the input string before deciding which rule to use.[13]

For example, with the string "abcdef", if we had a parser with 3 tokens of lookahead, from 'a' the parser would be able to look at tokens 'a', 'b' and 'c' before necessarily having to make a decision on which rule must be applied to continue to derivation. Similarly, from 'c', it'd be able to look at tokens 'c', 'd' and 'e'.

We are specifically going to be looking at LL(1) parsers, which are parsers which only have a single token of look ahead (so only one character may be taken into account before deciding which rule to use). LL(1) parsers are extremely practical in terms of their simplicity to implement, and they run in linear time with respect to the input string, making them fast, however the main drawback of LL(1) parsers, are the restriction which must be placed on the grammar in order for an LL(1) grammar to be able to parse it.

LL(1) Grammars

An LL(1) Grammar is one which can be accepted by an LL(1) parser. In order for this to be possible, several restrictions are placed on the grammar. Formally, an LL(1) grammar must:

- Be Left Factored (contain no FIRST / FIRST conflicts)
- Be Follow Deterministic (contain no FIRST / FOLLOW conflicts)
- Contain no left recursion (immediate or otherwise)

I will now go into depth into what each of these restrictions impose on a grammar, and to some extent how they can be dealt with and avoided.

FIRST / FIRST Conflicts & Left Factored Grammars

Previously I have mentioned how a FIRST set can be used as a tool to help place restrictions on a grammar. This is one of those restrictions (the other being follow determinism).

A FIRST / FIRST conflict occurs when any pair of alternates in a production rule contain the same token. To put this formally:

$$If\ for\ any\ grammar\ G,\ which\ contains\ a\ rule\ S\ \rightarrow \alpha \mid \beta\ ,\ where\ First(\alpha)\ \cap\ First(\beta) \neq \emptyset$$
$$Then\ G\ is\ not\ left\ factored$$ [14]

To give an example, we can take the grammar:

$S\ \rightarrow Ab \mid ac$
$A\ \rightarrow a \mid \varepsilon$

We take the FIRST set of A as:

$First(A)\ =\ First(a)\ \cup First(\varepsilon)\ =\ \{a,\ \varepsilon\}$

Now looking at the alternates of the rule $S\ \rightarrow Ab \mid ac$, we get FIRST sets $First(Ab)$ and $First(ac)$.

$First(Ab)\ =\ (First(A)/\{\varepsilon\})\ \cup First(b)\ =\ \{a,\ b\}$
$First(ad)\ =\ First(a)\ =\ \{a\}$

Now we can see that if we were to take the intersection of these FIRST sets, this would not result in the empty set.

$\{a,\ b\}\ \cap\ \{a\}\ =\ \{a\}$

The reason this is a problem, is because given our single token of lookahead, we don’t want to have to make choices on which alternate of a particular production rule we’re to use, since this can result in backtracking, which could slow down the process of parsing the input string significantly. Take the following example.

$S\ \rightarrow\ aB \mid aC \mid aD$
$B\ \rightarrow b$
$C\ \rightarrow c$
$D\ \rightarrow d$

$Input\ String$: $"ad"$

In order to make our first choice of alternate, we have the token ‘a’, which we are able to make with three separate alternates of the rule $S\ \rightarrow\ aB \mid aC \mid aD$, therefore, if any attempt fails, we must backtrack in order to check our other possible choices. This means we end up having to make the following derivations before we produce the desired string.

$S\ \Rightarrow aB\ \Rightarrow ab$ **fails!**
$S\ \Rightarrow aC\ \Rightarrow ac$ **fails!**
$S\ \Rightarrow aD\ \Rightarrow ad$ **Success!**

In the interest of speed, we would prefer to be able to avoid doing anything like this. I will also mention briefly that through the use of a technique known as left factoring we can eliminate this issue, like so:

$S\ \rightarrow aS'$
$S'\ \rightarrow\ B \mid C \mid D$
$B\ \rightarrow b$
$C\ \rightarrow c$
$D\ \rightarrow d$

Now if we were to take the same input string as before, we would be able to make the correct derivation in significantly less steps.

$S\ \Rightarrow aS'\ \Rightarrow aD\ \Rightarrow ad$

I won’t be covering exactly how to perform this technique however as it falls outside of the scope of this document.Simply knowing why a grammar must be left factored in order for it to be LL(1) is sufficient is sufficient for understanding the design choices presented in the grammar displayed later in this document.

Follow sets

In order to describe the next restriction, we must first go over what a FOLLOW set actually is. To give a formal definition:

$For\ a\ non\ terminal\ A\ we\ define$:

$$Follow(A)\ =\ \{t,\ where\ t\ is\ a\ terminal\ and\ S \Rightarrow^{*} \beta A t \alpha,\ for\ some\ \beta,\ \alpha\}$$

$Furthermore,\ If\ a\ derivation\ of\ the\ form\ \ S \Rightarrow^{*} \beta A\ exists$

$$\$\ \in\ Follow(A),\ where\ \$\ is\ a\ special\ token,\ indicating\ "End\ -\ Of\ -\ File"$$

$Since\ S\ is\ the\ start\ token,\ and\ subsequently\ always\ immediately\ precedes\ the\ end\ of\ a\ file$

$$\$\ \in\ \ Follow(S),\ with\ no\ exceptions$$[15]

To explain this simply, the FOLLOW set of a non terminal is the set of tokens which may follow that non terminal symbol during a derivation. For example, if we take the following grammar:

$S\ \rightarrow A$
$A\ \rightarrow xDa\ |\ B$
$B\ \rightarrow yDb\ |\ zDc$
$D\ \rightarrow\ d$

Now by observing each alternate where the D appears in, we can see which tokens may follow D in a derivation.

$A\ \rightarrow xDa$
$B\ \rightarrow yDb$
$B\ \rightarrow zDc$

We can clearly see tokens ‘a’, ‘b’ and ‘c’ follow D in these rules, therefore

$Follow(D)\ \ =\ \{a,\ b,\ c\}$

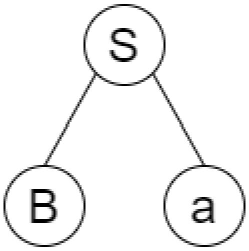
The reason the we sometimes needs to know which token follows a certain non terminal, is because when that non terminal is nullable, we need to know if we can match the current token of lookahead with not only the the set of tokens in the FIRST set of that non terminal, but also the set of tokens which can follow it, in the event that it is nullified. Let’s look at the following example:

$S\ \rightarrow Ba$

$B \rightarrow b \mid \varepsilon$

With this grammar, it should be possible to parse the string “a”, however let’s observe what happens when FOLLOW sets are not used.

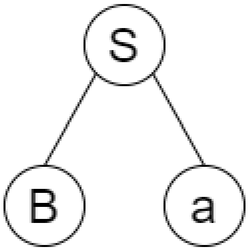
We begin with our start token S and our first and only lookahead token ‘a’. We know $First(Ba)$ to be {a, b}, which contains our lookahead token, so we can expand using this alternate.



We have expanded our tree, and proceed to check our leftmost node, however, $First(b)$is {b}, neither of which match our lookahead token ‘a’.

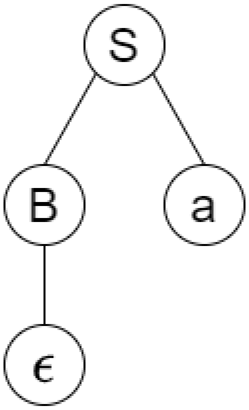
Because the FOLLOW set of B isn’t taken into consideration, whilst parsing B we have no way of verifying that “a” is meant to be a legal matching symbol. Now we shall do the same only this time, we consider both the FIRST and FOLLOW sets of B, since it is nullable.

As before we begin with our start token S and our first and only lookahead token ‘a’. We know $First(Ba)$ to be {a, b}, which contains our lookahead token, so we can expand using this alternate.



We have expanded our tree, and proceed to check our leftmost node, and this time, we know $First(b) \cup Follow(B) = \{a, b\}$, thus indicating to use that our lookahead token is valid.

Final word count: 46911



Now that we've verified using the FOLLOW set that this is in fact a legal string, we continue the parse and find 'a' to not match with 'b', thus ε is assigned. We then move back up to the root and on to the rightmost node where our lookahead token is matched with 'a' and the parse is completed.

The FOLLOW set is useful as it allows us to deal with nullable non terminals, however this also exposes us to another potential conflict.

FIRST / FOLLOW Conflicts & Follow Determined Grammars

Previously I discussed how if there is the potential for FIRST / FIRST conflicts to occur within a grammar, then a parser can not reliably parse strings for that grammar without including some form of backtracking to account for where these conflicts occur. If a grammar is not follow determined however, a similar problem occurs. We define follow determinism as follows:

$A\ grammar\ is\ said\ to\ be\ follow\ determined\ if\ for\ any\ non\ terminal\ A,\ strings\ \gamma,\ \delta\ and\ any\ terminal\ \alpha$

$$A \Rightarrow^{*} \gamma\ and\ A \Rightarrow^{*} \gamma\alpha\delta\ imply\ that\ \alpha \notin\ Follow(A)$$

However, it turns out that if a grammar is left factored, then the following properly is enough to guarantee that it is also follow determined:

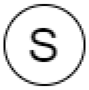
$$For\ any\ non\ terminal\ A,\ if\ A \Rightarrow^{*} \varepsilon\ then\ First(A)\ \cap\ Follow(A)\ =\ \emptyset$$ [14]

We can examine the following grammar in order to understand how this affects the parser:

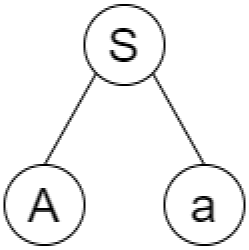
$S\ \rightarrow\ Aa$
$A\ \rightarrow a\ |\ \varepsilon$[15]

The two sentences which exist in this grammar are "a" and "aa". Let's observe what happens if we attempt to parse "aa".

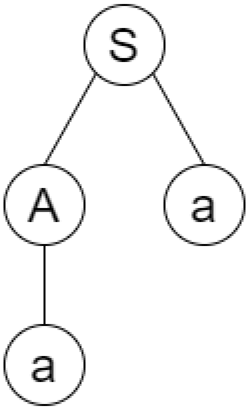
Final word count: 46911



We begin with the start symbol, and our first token of lookahead ‘a’. We know $First(Aa)$to be {a}, thus we know to expand using this alternate.

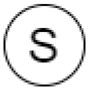


After expanding the tree, we move to our leftmost node. We know A is nullable and $First(a) \cup Follow(A)$ to be {a}. Our lookahead token is still ‘a’, so we match it with the rule $A \rightarrow a$and move onto our next lookahead token, which is also ‘a’.

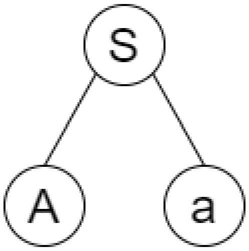


After moving back to the root node, we explore the rightmost node, and find that our second token of lookahead matches ‘a’, thus we have successfully produced our input string.

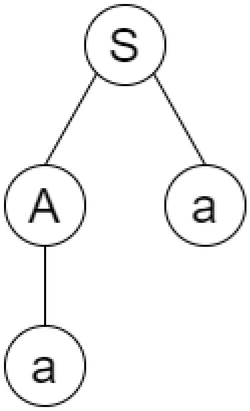
Clearly there are no issues here, however let’s observe what happens when we attempt to parse ‘a’.



As before we begin with S and our lookahead token ‘a’. We know $First(Aa)$to be {a}, thus we know to expand using this alternate.



After expanding the tree, we move to our leftmost node and examine the available alternates. As before We know A is nullable and $First(a) \cup Follow(A)$ to be {a}. Our lookahead token is once again ‘a’, so naturally we match against the rule $A \rightarrow a$and move onto our next lookahead token, ‘$’ or “End-Of-File”.



Now we move to the root node and explore the rightmost node, however this time we’ve already reached the end of the string, but we still have a token to match against. This is clearly erroneous.

Final word count: 46911

As demonstrated above, the problem revolves around the fact that there was no way of knowing whether to take the ‘a’ or εwhen parsing A because in both cases we had the same lookahead token, so naturally the exact same action was taken in both of the scenarios.

In this case, this problem can be simply avoided by modifying this grammar to be:

$S \rightarrow aA$
$A \ \rightarrow a \mid \varepsilon$

Thus fixing the problem, since now when parsing the sentence “a”, the first lookahead token is matched before parsing the non terminal A. because there is no longer an intersection between A’s FIRST and FOLLOW sets, the ambiguous situation posed when having the same lookahead token in two separate scenarios no longer occurs.

Final word count: 46911

Left Recursion

The last restriction placed on LL(1) grammars is that they must not contain any left recursion. The reason for this is because if a non terminal is able to replicate before the lookahead token is ever matched, the parser will simply be stuck with the same lookahead token, never advancing any further into the input string, meaning the parser will continue to loop indefinitely.

$$A\ grammar\ is\ said\ to\ be\ left\ recursive\ if\ it\ has\ a\ non\ terminal\ A\ such\ that$$
$$there\ is\ a\ derivation\ sequence\ A \Rightarrow^{+} A\alpha\ for\ some\ string\ \alpha$$

$$A\ production\ shows\ immediate\ left\ recursion\ if\ A\ \rightarrow A\alpha$$ [16]

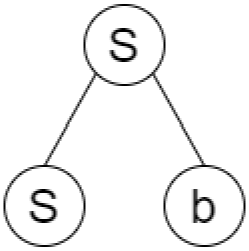
Consider the grammar:

$S\ \rightarrow Sb \mid a$

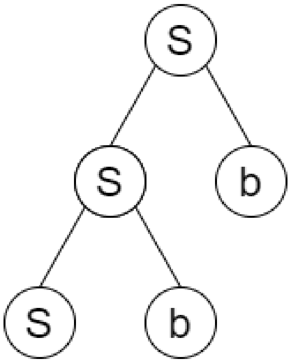
This grammar allows any sentence beginning with ‘a’, followed by an infinite number of b’s. But let’s observe what would happen if we attempted to parse any valid string.

$Input\ String\colon\ ab$

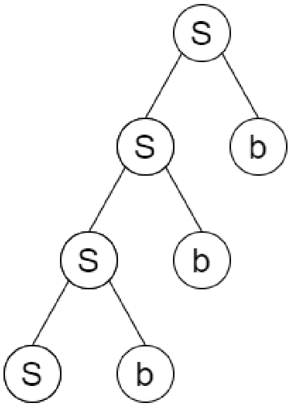
As with any derivation, we begin with the start symbol. Our lookahead token is currently ‘a’. $S \rightarrow Sb$is the leftmost alternate, so this is the alternate which we interact with first. $First(Sb)\ =\ \{a\}$so we proceed with this alternate.



The tree has been expanded and our lookahead token is still ‘a’. The exact same steps occur as before, taking us into the next state...



...and again...

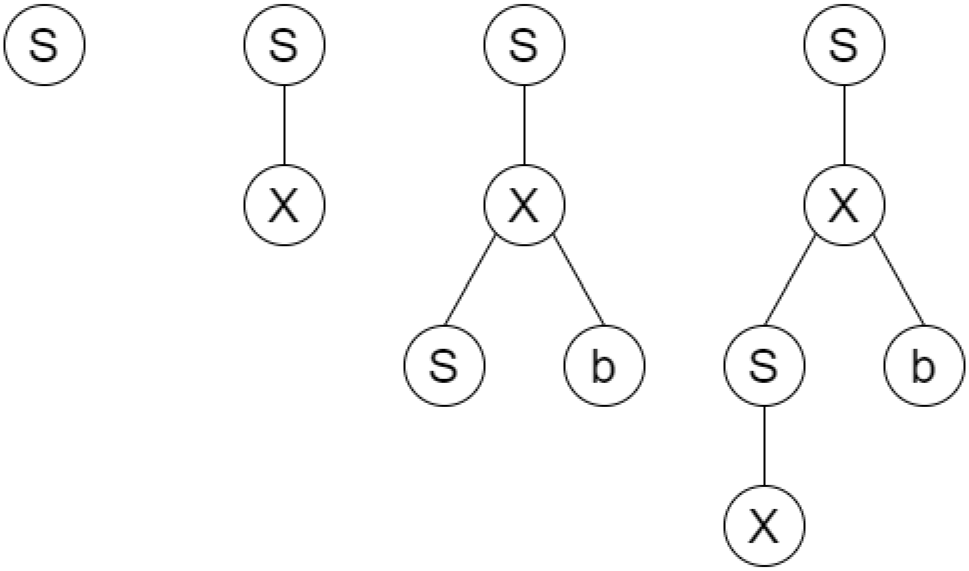


...and thus it never terminates.

There are two forms of left recursion. The grammar above is immediately left recursive, however a grammar can contain indirect left recursion, where the recursion occurs due one non terminal deriving another, rather than the same non terminal deriving itself.

$S \ \rightarrow X$
$X \ \rightarrow Sb \mid a$

This is effectively the same grammar, however the left recursion now has one level of indirection.



Through a technique known as substitution, it is possible to remove left recursion, both immediate and indirect. The first example above which contains immediate left recursion can be modified like so:

$S \ \rightarrow aS'$
$S' \ \rightarrow bS' \mid \varepsilon$

This effectively makes the grammar tail recursive, so the lookahead tokens can be matched before the non terminals are parsed.

I will not be going into any further detail on how substitution is performed as it falls outside the scope of this document. Knowing why a grammar must not contain any left recursion, immediate or otherwise, in order for it to be LL(1) is sufficient for understanding the design choices presented in the grammar displayed later in this document.

Recursive Descent Parsing

At this point we have covered all the underlying theory necessary to understand an implementation. We understand the concept of parse trees, how they are built using the production rules of a context free grammar, and how they produce our desired input string through left to right leftmost derivation (LL Parsing). We know the restrictions which must be placed on our grammar to allow a parser of this kind using only a single token of lookahead to be compatible with our grammar.

Now that the theory has been covered we can begin to start thinking about the actual implementation. I want to take us back to a comment I made previously on the relationship between parse trees and recursive functions. Let's review the steps of the process we intend to perform.

- We have a grammar containing a set of rules and an input string which.
- We want to determine whether this input string is a sentence in our grammar.
- We begin with our start symbol and proceed to select some alternate based on if our lookahead symbol is an element of the FIRST set, or in some either the FIRST or FOLLOW set of that alternate.
- The alternate is a string of terminal and non terminal symbols.
- If we hit a terminal symbol, we match our lookahead token and proceed to the next symbol in the alternate (if there is one).
- If we hit a non terminal symbol, we are once again exposed to another set (or perhaps the same set) of alternates, and once again we must choose one based on our current lookahead symbol.
- Upon hitting a non terminal symbol in our chosen alternate, there may be symbols to the right of that non terminal which we have yet to visit, but that's okay because we can simply **return** to this point later and continue looking over the rest of the alternate.
- This process continues until either the whole string has been matched, or a mismatch between the lookahead token and a terminal is found, in which case an error occurs and the process is halted.

So what do we have here? We have a function representing our start symbol, which we begin the parse with. Within this function, we have a series of conditional statements which determine which alternate to choose based on our current lookahead token. Nested within these conditions are additional checks for each symbol in the alternate. When a terminal is checked, it either matches our lookahead token, in which case we get our next lookahead token, or there is a mismatch and an error occurs. When a non terminal is checked, we make a function call and this process is repeated. Upon successfully checking each symbol of an alternate, the function returns, and the function which it was called from continues where it left off. Assuming all checks are successful, eventually we return back to the function we started with, which itself will return. If this happens, and we successfully reached the end of the string by this point, then the parse has completed successfully.

Final word count: 46911

While this is the basic idea, this is quite a lot to describe, thus this is best explained using an example.

We have the following grammar:

$S \rightarrow aAx \mid bBy$
$A \rightarrow rs \mid Bv$
$B \rightarrow p \mid q \mid \varepsilon$

- We have a global variable t representing our current lookahead token.
- We have a function gnt() which returns our nest lookahead token.
- We have a function error() which we call in the event that a mismatch is found.
- We represent “End-Of-File” with the $ token.
- We have a set of functions parseS(), parseA() and parseB(). These will hold the logic associated with each production rule.

We shall begin with a main function as the entry point to our parser program.

```
main()  {
      t = gnt();  // assigns our first lookahead token
      parseS();   // begins the parse from the start symbol
      if (t == '$') return SUCCESS; // verifies the whole string was parsed
      else return FAILURE;
}
```

Our first lookahead token has been assigned, and we begin parsing by calling ParseS(), Which represents our start symbol.

```
parseS() {
      if (t ∈ FIRST(aAx)) { // potentially chooses first alternate
          if (t == 'a') t = gnt() else error(); // matches with token ‘a’
          parseA(); // function call representing non terminal A
          if (t == 'x') t = gnt() else error(); // matches with token ‘x’
      } else if (t ∈ FIRST(bBy)) { // potentially chooses second alternate
          if (t == 'b') t = gnt() else error(); // matches with token ‘b’
          parseB(); // function call representing non terminal B
          if (t == 'y') t = gnt() else error(); // matches with token ‘y’
      } else error(); // if no alternate is chosen, an error occurs
}
```

Final word count: 46911

Each alternate of S is checked in turn, and if nothing matches the current lookahead symbol, then we're clearly looking at an invalid string, thus an error is thrown. For each alternate, there is a series of checks which match the symbols string of symbols which form each alternate. A match with a terminal symbol results in t being assigned the next lookahead token, which would be the next character in the input string. If it does not match, then we're clearly looking at an invalid string, thus an error is thrown.

```
parseA() {
        if (t ∈ FIRST(rs)) {
                if (t == 'r') t = gnt() else error();
                if (t == 's') t = gnt() else error();
        } else if (t ∈ FIRST(Bv)) {
                parseB();
                if (t == 'v') t = gnt() else error();
        } else error();
}
```

This function is not too dissimilar from parseS(). The main difference here, is A derives an alternate which contains no non terminals, $A \rightarrow rs$, which means no additional function calls are made if this alternate is chosen. With alternates such as this one it becomes clear how this program can eventually terminate.

```
parseB() {
        if (t ∈ FIRST(p)) {
                if (t == 'p') t = gnt() else error();
        } else if (t ∈ FIRST(q)) {
                if (t == 'q') t = gnt() else error();
        } // B is nullable, thus finding no matching alternate is acceptable
}
```

This time both alternates only contain terminal symbols, so if this function is ever called, assuming the input string is valid this function is guaranteed to create leaf nodes (remember, this is effectively building a tree of branching function calls). One particular difference this function has compared to the other two is that it no longer throws an error if none of the alternates are chosen. Since B is nullable it can simply match against nothing, which is effectively the same as matching against the empty string ε.[17]

Final word count: 46911

EBNF

Up until the previous section we have largely been working in the world of theorey. However now that we've begun to look at how parsers can be written in practice, we can begin exploring tools for representing our grammar in practice also.

EBNF stands for extended backus naur form, and is a tool, similar to BNF which we use to define context free grammars. There are common patterns which occur in production rules, so much so that it is useful to have ways of expressing these patterns in shorthand notation. EBNF provides such shortcuts.

Such shortcuts include:

- Parentheses (...) which create a sub-production (an unnamed production rule)

<u>Example:</u>

$S \rightarrow aB$ can be written as $S \rightarrow a(b)$
$B \rightarrow b$

- Brackets [...] as shorthand for 'zero or one occurrence'
    - May also be written as (...)? (Optional)

<u>Example:</u>

$S \rightarrow aB$ can be written as $S \rightarrow a[B]$
$B \rightarrow b \mid \varepsilon$

- Braces {...} as shorthand for 'zero or more occurrences'
    - May also be written as (...)* (Kleene closure)

<u>Example:</u>

$S \rightarrow aB$ can be written as $S \rightarrow a\{B\}$
$B \rightarrow bB \mid \varepsilon$

Final word count: 46911

- Angle brackets <...> as shorthand for ‘one or more occurences’
  - May also be written as (...)+ (positive closure)

Example:

$$\begin{aligned} S &\rightarrow aB \\ B &\rightarrow bB' \\ B' &\rightarrow bB' \mid \varepsilon \end{aligned}$$ can be written as $$S \rightarrow a < B >$$

These are often useful for representing things like initializer lists, or parameter lists:

$$S \rightarrow int\ ID\ \{,\ ID\}\ ;$$ permits the string **int** x, var, _val, y2;

Note: I’m using ‘ID’ as a token which matches anything which is a valid C identifier.[18]

```
Final word count: 46911
```

rdp

rdp is a system for implementing language processors. It takes as input an LL(1) grammar written in the rdp source language (an EBNF), and outputs a program written in C, which is a recursive descent parser for the language defined by that grammar.[19]

Besides generating a parser for an input grammar, rdp is also capable of verifying that the input grammar is LL(1), and if it is not, will output detailed diagnostics explaining which conflicts the grammar contains and where. It is possible for rdp to accept non-LL(1) grammars if the user so desires it, This is necessary due to a certain ambiguity involved when implementing conditional "if-then-else" statements. When this is the case, the parser will instead use the "longest match" strategy, which I will explain further down.[20]

IBNF

Iterator Backus Naur Form (IBNF) is rdp's source language. It is an EBNF with some additional functionalities, some of which I will cover in this section as they are used in my implementation.

Production rules in IBNF are written in the form:

```
rule_name ::= rule_expression.
```

Where the rule name is a non terminal and the rule expression is a collection of alternates of the grammar rule.

Rule names are written as strings which must follow the same conventions as C identifiers, that is they must begin with either an alphabetic character or an underscore, and thereafter may consist of several more alphanumeric characters or underscores. It is also worth a mention that rdp does not permit duplicate rule names.[21]

Alternates of a rule are to be separated by the pipe character '|' and should consist of defined rule names, strings of terminal symbols wrapped in single quotation marks, or special tokens defined in rdp's specification, which I will cover further down. Additionally, each individual symbol must be separated by white space. Portions of alternates can also be wrapped in any of the brackets covered in the EBNF section. Finally, each rule must end in a full stop character '.' to indicate the end of that rule.

```
translation_unit ::= {initlizalizer_list}.
initializer_list ::= type_specifier ID { ‘,’ type_specificer} ‘;’.
Type_specifier   ::= ‘int’ | ‘char’ | ‘float’ | ‘void’.
```

Final word count: 46911

Special Tokens in rdp

Traditionally when generating a parser for a language, the source input string would initially be read by a Lexical Analyser, which would group strings into tokens which the parser can recognize, so the then tokenized input string can be parsed. rdp has a built in lexical analyzer, meaning the parser can take the source input string directly.

One of the tools IBNF affords us is a set of special predefined tokens which act as regular expressions for commonly used strings.

- The token **ID** matches C-style identifiers
  - Eg. x, var, _val and y23, etc...
- The token **INTEGER** matches with C-style integer literals
  - Eg. 123, 0xB2, etc...
- The token **REAL** matches with C-style real literals
  - Eg. 3.141, 2.414, 4.23e3, 2.67e-4, etc…

There are others also, such as STRING which matches with Pascal style strings and STRING_ESC which matches C-style strings (under some additional constraints), and some others but these aren’t necessary to cover as they do not appear in my implementation.[22]

Comments in rdp

Comment blocks in IBNF are opened using an opening bracket followed by an asterisk, and closed by an asterisk followed by a closing bracket.[23]

```
S ::= ‘a’ A.     (* Start Symbol *)
A ::= ‘b’ | ‘c’. (* Additional Production Rule *)
```

Final word count: 46911

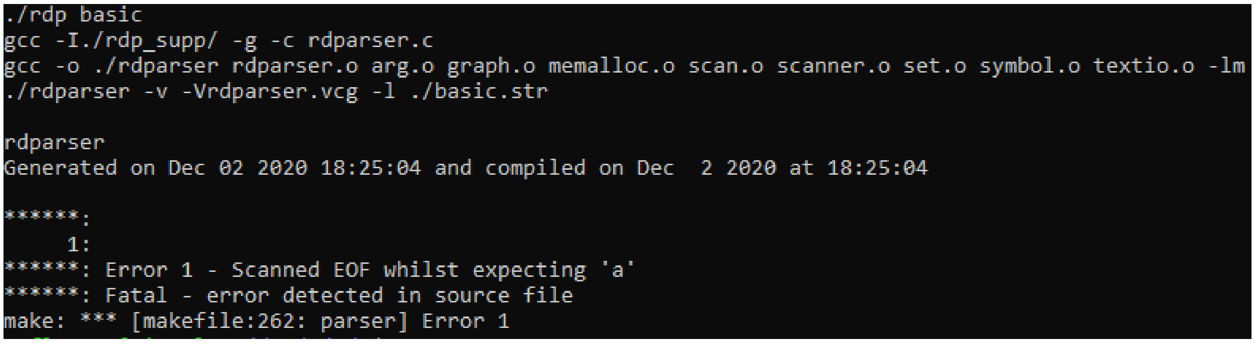
rdp Basic Usage

As a basic tool, rdp can be used to build a parser which can verify that a source input string belongs to the specific grammar used to build the parser. rdp expects two files. The first being a **.bnf** file which contains the LL(1) grammar written in IBNF, which will be used by rdp to build the parser. The Other is a **.str** file which should contain a valid string for the grammar defined in the **.bnf** file, as rdp will use the contents of this file as an input string. As an additional constraint, both files must also have the same name.

As a basic first step to demonstrate this, I will create a parser using a file named “basic.bnf” with the following contents:

```
S ::= ‘a’.
```

Attempting to generate the parser gives me the following diagnostic information:



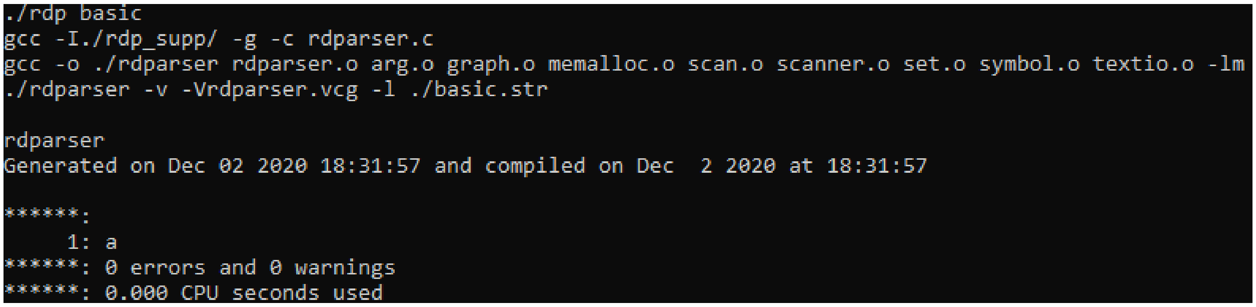
```
./rdp basic
gcc -I./rdp_supp/ -g -c rdparser.c
gcc -o ./rdparser rdparser.o arg.o graph.o memalloc.o scan.o scanner.o set.o symbol.o textio.o -lm
./rdparser -v -Vrdparser.vcg -l ./basic.str

rdparser
Generated on Dec 02 2020 18:25:04 and compiled on Dec  2 2020 at 18:25:04

******:
     1:
******: Error 1 - Scanned EOF whilst expecting 'a'
******: Fatal - error detected in source file
make: *** [makefile:262: parser] Error 1
```

It clearly states: `Error 1 - Scanner EOF whilst expecting ‘a’`

Currently “basic.str” is an empty file, meaning the initial lookahead token is EOF or end-of-file, however the parser was expecting a single ‘a’ character, so the parser successfully generated, but the initial test parse failed. Now I shall add the following contents to “basic.str” to rectify this problem.

`Final word count: 46911`

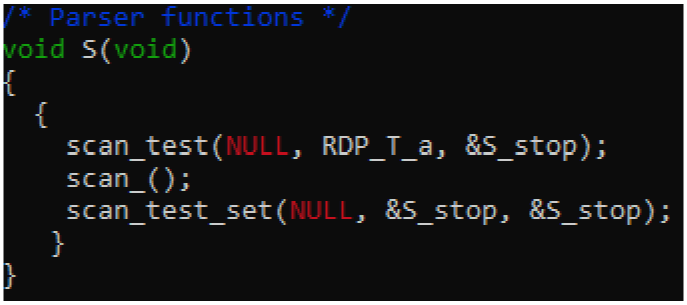
With the input string ‘a’, we get the following output:



```
./rdp basic
gcc -I./rdp_supp/ -g -c rdparser.c
gcc -o ./rdparser rdparser.o arg.o graph.o memalloc.o scan.o scanner.o set.o symbol.o textio.o -lm
./rdparser -v -Vrdparser.vcg -l ./basic.str

rdparser
Generated on Dec 02 2020 18:31:57 and compiled on Dec  2 2020 at 18:31:57

******:
     1: a
******: 0 errors and 0 warnings
******: 0.000 CPU seconds used
```

We see 0 errors, indicating that both the parser generation and the initial test parse were successful. It also shows where the character had been read.

We can also look at the source code that rdp has generated in the “rdparser.c” file.



```
/* Parser functions */
void S(void)
{
  {
    scan_test(NULL, RDP_T_a, &S_stop);
    scan_();
    scan_test_set(NULL, &S_stop, &S_stop);
   }
}
```

This is the parse function for our production rule. Not to go into too much detail, essentially what is happening here is the same process I previously described in an earlier section. The parser is first checking the FIRST set of the alternate, and then it is matching against the terminal symbol ‘a’ itself.

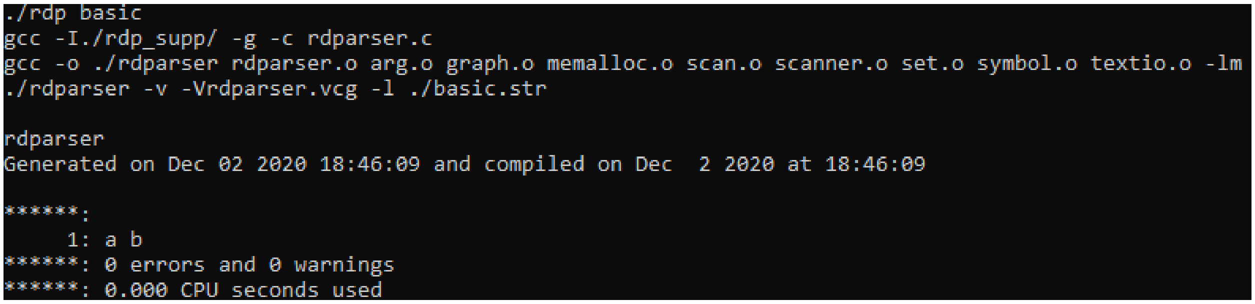
We can make this output slightly more interesting by adding an additional production rule and adding an alternate to S.

```
S ::= ‘a’ B | B.
B ::= ‘b’.
```

We also must change “basic.str”.

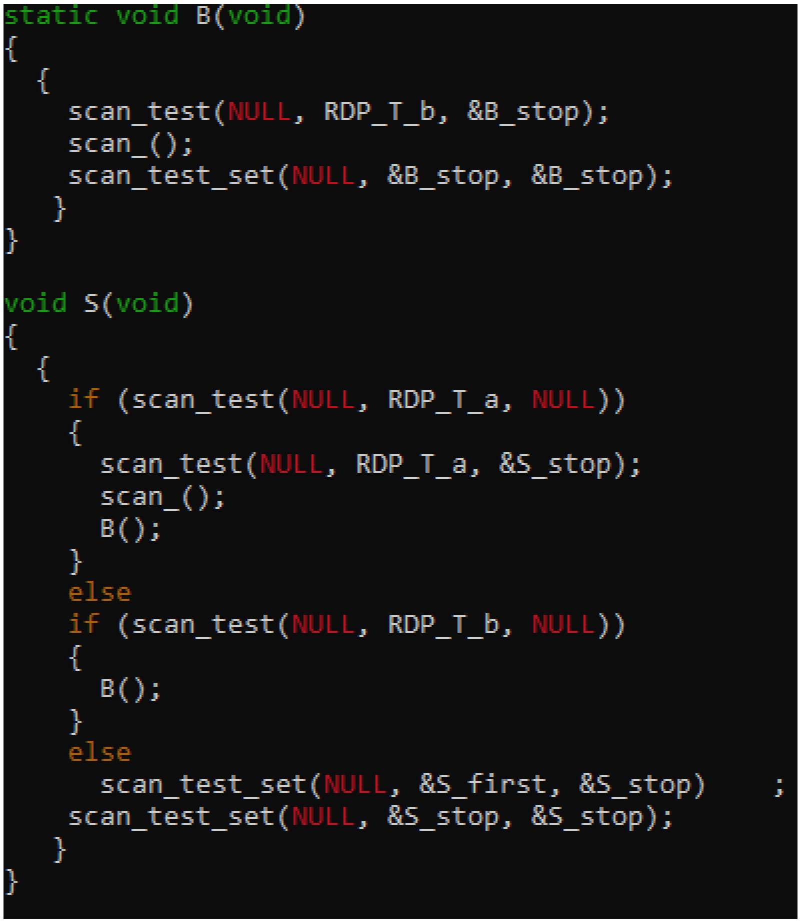
```
a b
```

Final word count: 46911



```
./rdp basic
gcc -I./rdp_supp/ -g -c rdparser.c
gcc -o ./rdparser rdparser.o arg.o graph.o memalloc.o scan.o scanner.o set.o symbol.o textio.o -lm
./rdparser -v -Vrdparser.vcg -l ./basic.str

rdparser
Generated on Dec 02 2020 18:46:09 and compiled on Dec  2 2020 at 18:46:09

******:
     1: a b
******: 0 errors and 0 warnings
******: 0.000 CPU seconds used
```

This is completed with no issues.



```
static void B(void)
{
  {
    scan_test(NULL, RDP_T_b, &B_stop);
    scan_();
    scan_test_set(NULL, &B_stop, &B_stop);
   }
}

void S(void)
{
  {
    if (scan_test(NULL, RDP_T_a, NULL))
    {
      scan_test(NULL, RDP_T_a, &S_stop);
      scan_();
      B();
    }
    else
    if (scan_test(NULL, RDP_T_b, NULL))
    {
      B();
    }
    else
      scan_test_set(NULL, &S_first, &S_stop)    ;
    scan_test_set(NULL, &S_stop, &S_stop);
   }
}
```

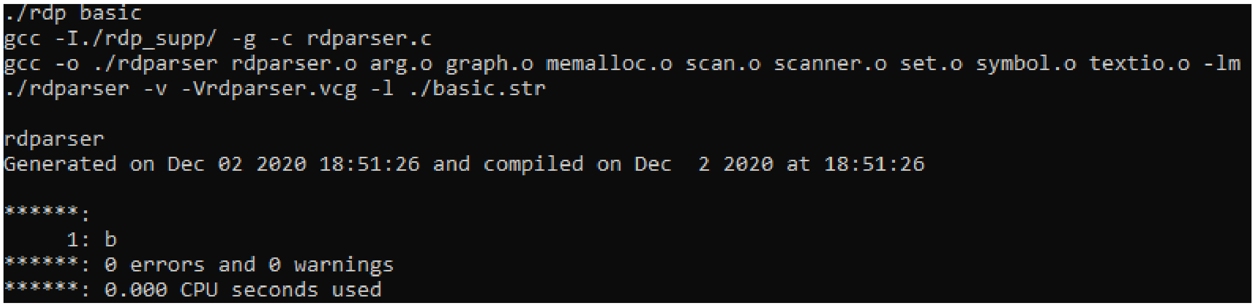
We now have two functions, one for each of our production rules. S also now tests for two different alternates, and makes a call to B() in each of them.

Final word count: 46911

The parse can now also handle more than a single string. We change “basic.str” to the following:

```
b
```

Which is also verified with no issues.



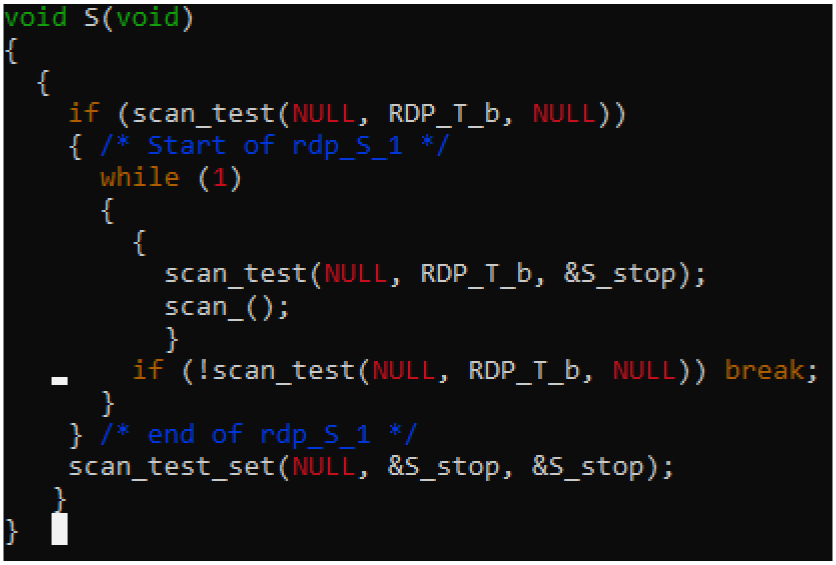
```
./rdp basic
gcc -I./rdp_supp/ -g -c rdparser.c
gcc -o ./rdparser rdparser.o arg.o graph.o memalloc.o scan.o scanner.o set.o symbol.o textio.o -lm
./rdparser -v -Vrdparser.vcg -l ./basic.str

rdparser
Generated on Dec 02 2020 18:51:26 and compiled on Dec  2 2020 at 18:51:26

******:
     1: b
******: 0 errors and 0 warnings
******: 0.000 CPU seconds used
```

As one final demonstration I will show what happens when we use the curly braces, as this the code generation will be useful for understanding things later. I now change the grammar to the follow:

```
S ::= {‘b’}.
```

And “basic.str” to the following

```
b b b b b
```



```
void S(void)
{
  {
    if (scan_test(NULL, RDP_T_b, NULL))
    { /* Start of rdp_S_1 */
      while (1)
      {
        {
          scan_test(NULL, RDP_T_b, &S_stop);
          scan_();
          }
        if (!scan_test(NULL, RDP_T_b, NULL)) break;
      }
    } /* end of rdp_S_1 */
    scan_test_set(NULL, &S_stop, &S_stop);
   }
}
```

As you can see, the Kleene closure is implemented as a while loop. It will be important to know this later.

Final word count: 46911

My Initial Grammar

As my final goal for this project is to create a compiler which can recognise and compile a subset of the C language, my initial grammar would naturally reflect that. My thought process was that I would want enough functionality to make some values change during runtime, but to achieve this without overburdening myself by trying to add too much functionality. I settled on these ideas:

- I need to be able to declare variable and assign values to those variables so I may store data during runtime
- I need to be able to perform arithmetic operations, so data can be changed in ways other than simply assigning values.
- I want to have some form of control flow so the compiler does is not simply a glorified calculator

Due to the fact I also wanted the grammar to be as C-like as possible, I also opted to have a very basic implementation of functions, since I wanted to be able to write a main function. These ideas led me to write the following test string:

```
int a, b, c;

int main() {
      int x, y, z;
      while (x < 5) {
            x = x + y * z;
      }
      return x + 1;
}
```

This includes:

- Declaration list, both global and local to a function
- Function definitions
- While loops as control flow
- Multiplication and addition
- Return statements with appended expressions

The intention of attempting to create a C-like grammar led me to the appendices of “The C Programming Language. 2nd Edition by Brian Kernighan and Dennis Ritchie”, where the grammar for ANSI C is documented. Since the grammar documented there is not LL(1), I tasked myself with converting the relevant parts of the grammar into something I could use with rdp, and this was the result:

Final word count: 46911

```
TRNS_UNIT ::= {EXTN_DECL} .
EXTN_DECL ::= TYPE_SPEC ID DECL.
TYPE_SPEC ::= 'int' .
DECL      ::= FUNC_DEFN | DECL_LIST .
FUNC_DEFN ::= '(' [PARAMS] ')' CMPD_STMT.
DECL_LIST ::= {',' ID} ';' .
PARAMS    ::= TYPE_SPEC ID {',' TYPE_SPEC ID} .
CMPD_STMT ::= '{' {INTL_DECL | STMT} '}' .
INTL_DECL ::= TYPE_SPEC ID DECL_LIST .
STMT      ::= TERM_STMT | ITER_STMT .
TERM_STMT ::= (JUMP_STMT | EXPR) ';' .
ITER_STMT ::= 'while' '(' EXPR ')' (CMPD_STMT | STMT) .
JUMP_STMT ::= 'return' EXPR .
EXPR      ::= PRIMARY (ASSIGN | MATH_EXPR) .
ASSIGN    ::= '=' PRIMARY MATH_EXPR .
MATH_EXPR ::= UNARY_OP PRIMARY MATH_EXPR | BOOL_EXPR  .
BOOL_EXPR ::= {BOOL_OP PRIMARY} .
UNARY_OP  ::= '+' | '*' .
BOOL_OP   ::= '==' | '!=' | '>' | '<' | '>=' | '<=' .
PRIMARY   ::= INTEGER | ID .
```

The grammar above is capable of verifying everything in the test string above as a valid string. Additionally, it also allows the use of boolean expressions. Let’s break this down to understand exactly what is happening here.

```
TRNS_UNIT ::= {EXTN_DECL} .
```

This is the start symbol of the grammar and stands for “translation unit” which essentially refers to a source file in the C language. Immediately I have put a Kleene closure around what it derives and we can look at the next few lines to make it clear why.

```
EXTN_DECL ::= TYPE_SPEC ID DECL.
TYPE_SPEC ::= 'int' .
DECL      ::= FUNC_DEFN | DECL_LIST .
```

First we have “external declaration”, which refers to any declaration which happens in the global scope. EXTN_DECL derives a “type specifier”, followed by an ID and then a “declaration”.

“Type Specifier” refers to the different primitive data types available in C. In the case of this grammar, due to the fact I wanted to keep things simple at first, this grammar was written to only support integers, however, I implemented this way specifically so it would be easier to include more types later if and when I saw fit.

ID simply matches against C-Style identifiers. “Declaration” in this case refers to either a “function definition” or a “declaration list”. So to conclude, the first four rules are intended to allow for the declaration of zero or more function definitions and declaration lists within the translation unit.

```
EXTN_DECL ::= TYPE_SPEC ID DECL.
TYPE_SPEC ::= 'int' .
DECL      ::= FUNC_DEFN | DECL_LIST .
DECL_LIST ::= {',' ID} ';' .
```

Between these four rules, it is possible to declare a list of variables, of type integer, in the global scope, separated by commas and ending with a semicolon as is standard in C.

```
int a, b, c;
```

The use of the Kleene closure here facilitates the implementation of a comma delimited list.

```
EXTN_DECL ::= TYPE_SPEC ID DECL.
TYPE_SPEC ::= 'int' .
DECL      ::= FUNC_DEFN | DECL_LIST .
FUNC_DEFN ::= '(' [PARAMS] ')' CMPD_STMT.
PARAMS    ::= TYPE_SPEC ID {',' TYPE_SPEC ID} .
CMPD_STMT ::= '{' {INTL_DECL | STMT} '}' .
```

This similar set of rules adds function definitions to the grammar. We’re already familiar with the first three rules so i’ll focus on the last three. FUNC_DEFN or “function definition” derives an optional list of comma delimited parameters between two brackets, followed by a “compound statement”, which for now I will simply explain as a list of statements held between two curly braces.

No parameters

```
int func() {
    // content of function
}
```

One parameter

```
int func(int x) {
    // content of function
}
```

Multiple parameters

```
int func(int x, int y) {
    // content of function
}
```

Final word count: 46911

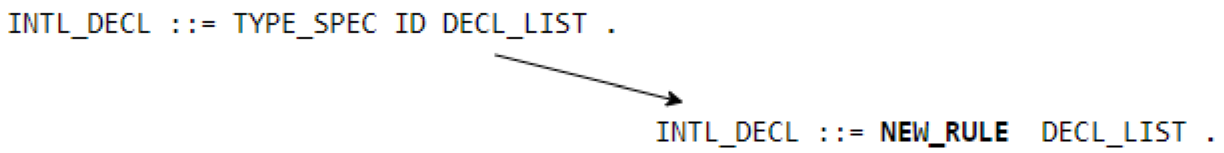
Between these two rules I could have avoided some “code duplication” by adding an additional rule to which derives type specifier followed by ID.

```
EXTN_DECL ::= TYPE_SPEC ID DECL.
PARAMS    ::= TYPE_SPEC ID {',' TYPE_SPEC ID} .
```

By adding such a rule, I could have removed external declaration entirely, making the grammar slightly more concise.

```
TRNS_UNIT ::= {DECL} .
NEW_RULE  ::= TYPE_SPEC ID
TYPE_SPEC ::= 'int' .
DECL      ::= NEW_RULE (FUNC_DEFN | DECL_LIST) .
FUNC_DEFN ::= '(' [PARAMS] ')' CMPD_STMT.
DECL_LIST ::= {',' ID} ';' .
PARAMS    ::= NEW_RULE  {',' NEW_RULE } .
```

This new rule could have also been applied further down to make “internal declaration” more concise also.



```
INTL_DECL ::= TYPE_SPEC ID DECL_LIST .

                                     INTL_DECL ::= NEW_RULE  DECL_LIST .
```

Everything I’ve covered up to this point can only exist within the global scope and beyond this point everything I will cover can only exist within the scope of a function. Now let’s take a closer look at compound statements.

```
CMPD_STMT ::= '{' {INTL_DECL | STMT} '}' .
INTL_DECL ::= TYPE_SPEC ID DECL_LIST .
STMT      ::= TERM_STMT | ITER_STMT .
```

A compound statement can contain zero or more “internal declarations” or “statements”. An internal declaration is simply a declaration list which can be declared within a compound statement. The reason I ended up repeating myself here was due to the fact that I did not want to allow the declaration of functions within functions, which is exactly what using declaration would have permitted.

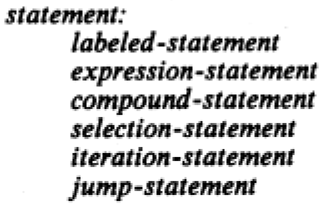
```
DECL      ::= FUNC_DEFN | DECL_LIST .
```

This is something which C allows, however I assumed there would be complications implementing the semantics so I decided not to include this feature.

`Final word count: 46911`

```
STMT      ::= TERM_STMT | ITER_STMT .
TERM_STMT ::= (JUMP_STMT | EXPR) ';' .
ITER_STMT ::= 'while' '(' EXPR ')' (CMPD_STMT | STMT) .
JUMP_STMT ::= 'return' EXPR .
```

My inspiration for implementing statements in this was mostly inspired from the C grammar in the The C Programming Language book.



statement:

labeled-statement

expression-statement

compound-statement

selection-statement

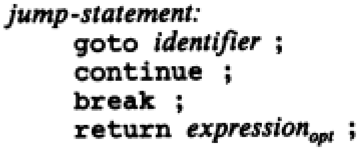
iteration-statement

jump-statement

Figure 1. [24]

I opted to implement three types of statements:

- “Jump statements”, which in this case only refers to **return** statements, but was planned to later encompass **break, continue and goto.**



jump-statement:

goto *identifier* **;**

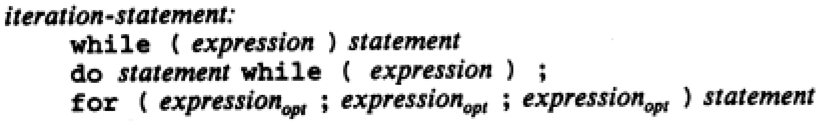
continue ;

break ;

return $expression_{opt}$ **;**

Figure 2. [25]

- “Iteration statements”, which in this case only accounts for **while** loops but was planned to later encompass **do while** and **for** loops.



iteration-statement:

while (*expression* **)** *statement*

do *statement* **while (** *expression* **) ;**

for ($expression_{opt}$ **;** $expression_{opt}$ **;** $expression_{opt}$ **)** *statement*

Figure 3. [26]

- “Expression statements”, which are simply named expressions or “EXPR” to be exact.

TERM_STMT simply exists to group “jump statement” and “expression”, and isn’t actually a real type of statement. The reason these two are grouped this way is because they both end with a semicolon, whereas iteration statements do not, so this was an easy way to make this distinction in the grammar.

```
TERM_STMT ::= (JUMP_STMT | EXPR) ';' .ITER_STMT ::= 'while' '(' EXPR ')'
(CMPD_STMT | STMT) .
JUMP_STMT ::= 'return' EXPR .
```

A useful thing to note about the use of expressions in these two statement types is that in C, any non-zero value is **true** and any **zero** value is false. There isn’t actually an explicit boolean type in C, which means the result of an arithmetic expression can also be treated as the result of a boolean expression. Using expressions in this way means everything below is valid syntax as to be expected from a C compiler.

True	Also True	False	Also False
`while (1 == 1)`	`while (1 + 1)`	`while (1 != 1)`	`while (1 - 1)`

```
ITER_STMT ::= 'while' '(' EXPR ')' (CMPD_STMT | STMT) .
```

Because C allows the logic following while loops to be both inline and nested I decided my grammar should allow this also.

Inline

```
while(1) // these things will loop
```

Nested

```
while (1) {
      // these things will loop
}
```

The exact same subset of the grammar which is permitted to be written inside a function is also permitted to be written inside a while loop. Essentially, any nested scope should permit the same behaviour.

```
EXPR      ::= PRIMARY (ASSIGN | MATH_EXPR) .
ASSIGN    ::= '=' PRIMARY MATH_EXPR .
MATH_EXPR ::= UNARY_OP PRIMARY MATH_EXPR | BOOL_EXPR .
BOOL_EXPR ::= {BOOL_OP PRIMARY} .
UNARY_OP  ::= '+' | '*' .
BOOL_OP   ::= '==' | '!=' | '>' | '<' | '>=' | '<=' .
PRIMARY   ::= INTEGER | ID .
```

Above are all the rules which encompass expressions. This includes boolean, arithmetic and assignment expressions.

`Final word count: 46911`

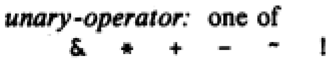
```
EXPR      ::= PRIMARY (ASSIGN | MATH_EXPR) .
ASSIGN    ::= '=' PRIMARY MATH_EXPR .
PRIMARY   ::= INTEGER | ID .
```

This allows the assignment of expressions to both variables and integers literals. Although the latter has no real functionality, this is something which C allows, so at this point in time I decided to allow it also. This is something I would review later.

Legal Syntax	Legal (although useless) Syntax
`x = 3 + 4;`	`0 = 3 + 4;`

```
EXPR      ::= PRIMARY (ASSIGN | MATH_EXPR) .
MATH_EXPR ::= UNARY_OP PRIMARY MATH_EXPR | BOOL_EXPR .
UNARY_OP  ::= '+' | '*' .
PRIMARY   ::= INTEGER | ID .
```

The implementation of “mathematical expressions” is subject to a certain misinterpretation I made.



unary-operator: one of
& * + - ~ !

Figure 4. [27]

Above is the set of unary operators in the C grammar. This includes the reference and dereference operators, bitwise not and logical not, and unary prefixes for indicating whether a value is positive or negative. What I had done is misinterpreted the positive unary prefix and the dereference operator as the binary operators for addition and multiplication. Although this creates no issues when simply verifying the syntax is correct, this would add significant complications when adding semantic actions, a point which I will explore later.

```
MATH_EXPR ::= UNARY_OP PRIMARY MATH_EXPR | BOOL_EXPR .
```

MATH_EXPR derives itself, making it tail recursive, which allows for long arithmetic expressions

```
1 + 2 * 3 + 4 * 5;
```

The reason this grammar only includes multiplication and addition, is because I wanted to avoid some assumed complications with implementing subtraction and division, at least until I had the semantics for these two operators working correctly.

Final word count: 46911

```
MATH_EXPR ::= UNARY_OP PRIMARY MATH_EXPR | BOOL_EXPR .
BOOL_EXPR ::= {BOOL_OP PRIMARY} .
BOOL_OP   ::= '==' | '!=' | '>' | '<' | '>=' | '<=' .
```

Boolean expressions were the last feature I added . The set of alternates under the BOOL_OP production rule have been implemented in a way to match how unary operators have been implemented. Boolean expressions have been implemented in this way to allow for chains of boolean expressions in between arithmetic expressions.

```
1 + 2 + 3 != 4 + 5 + 6;
```

The Problem With This Grammar

There are a handful of minor issues associated with this grammar, which I have already mentioned above, such as the misinterpretation of unary operators as binary operators, and places where the grammar could have been made more concise. Boolean expressions were added as an afterthought, so there are some problems with those also.

The main issue with this grammar, is that it was written without the addition of semantic actions in mind. Due to the way the grammar has been structured, the implementation of C-like semantics would be significantly more difficult or outright impossible. There are quite a few examples of this, but explaining them all would be a length process, and I have yet to properly cover how rdp handles the addition of semantic actions.

One place in particular though is here:

```
MATH_EXPR ::= UNARY_OP PRIMARY MATH_EXPR | BOOL_EXPR  .
UNARY_OP  ::= '+' | '*' .
```

Due to how recursive descent parsers work, with this implementation of arithmetic expressions, It would be difficult to add the semantics detailing the different operator priorities between addition and multiplication. This problem has been solved in my latest grammar however, so I will explain how when going over the details of that grammar.

Final word count: 46911

Semantic Actions in rdp

A grammar defines the set of legal sentences in a language. A parser verifies the syntax of a sentence according to that grammar, however this is not the only job of a parser. A sentence is meaningless without its semantics, quite literally so. A parser not only verifies syntax but may also attach meaning to a sentence.

Recall what a compiler actually does, it takes source code as input and outputs an executable program. A for loop with an iterator which starts a zero and increments by one until it reaches 5 is effectively telling your computer (or whatever might be executing the code) to repeat the same set of actions 5 times. You can do this with natural languages also. You can tell someone to repeat the same set of actions 5 times, although in this case they might not perform the actions (computer always does as it is told), this does not remove the meaning of what had been said though.

rdp allows the attachment of semantic actions to alternates in production rules. The semantic actions are written as C code snippets which are reflected in the code generated by rdp. Essentially what this means is, when parsing a sentence and a certain alternate is chosen, some C code is executed along with it, which will act as the semantic action attached to that rule. As an example, we could have a rule which forms an arithmetic operation, and some C code which performs the arithmetic operation.

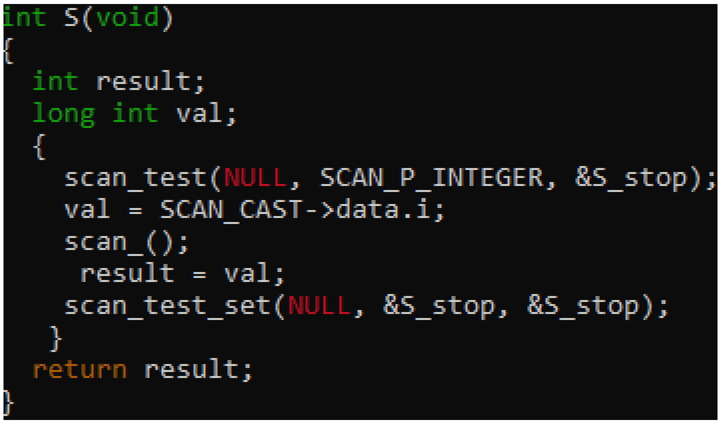
How this is done is much better explained through examples. Examine the following:

basic.bnf:	`S:int ::= INTEGER:val [* result = val; *].`
basic.str:	5

From the example above presents several new features to discuss.

- S:int - The start symbol has been assigned the integer data type.
- INTEGER:val - This token now has a variable name attached to it.
- `[* result = val; *] -` It appears whatever value INTEGER matches with is assigned to some variable “result”.

Let’s discuss this mysterious “result” figure first. When a rule is given a data type as has been done so with S in this example, this is reflected in the code generated by rdp. By default each function which rdp generates holds the return type **‘void’**, however if a type is assigned to a rule, then the associated function will take the assigned type as its return type and the value which the function returns is held in the variable ‘result’.



```
int S(void)
{
  int result;
  long int val;
  {
    scan_test(NULL, SCAN_P_INTEGER, &S_stop);
    val = SCAN_CAST->data.i;
    scan_();
     result = val;
    scan_test_set(NULL, &S_stop, &S_stop);
   }
  return result;
}
```

These variables are known as “attributes”. Attributes are used to pass data between production rules, so the data can be used in the semantic actions of other rule’s alternates. There are two types of attributes in rdp: “synthesized attributes” and “inherited attributes”, the example above uses only the former.[28]

Synthesized attributes are variables created locally within the parse function, so in this case ‘result’ which holds the return value of the parse function, and ‘val’.

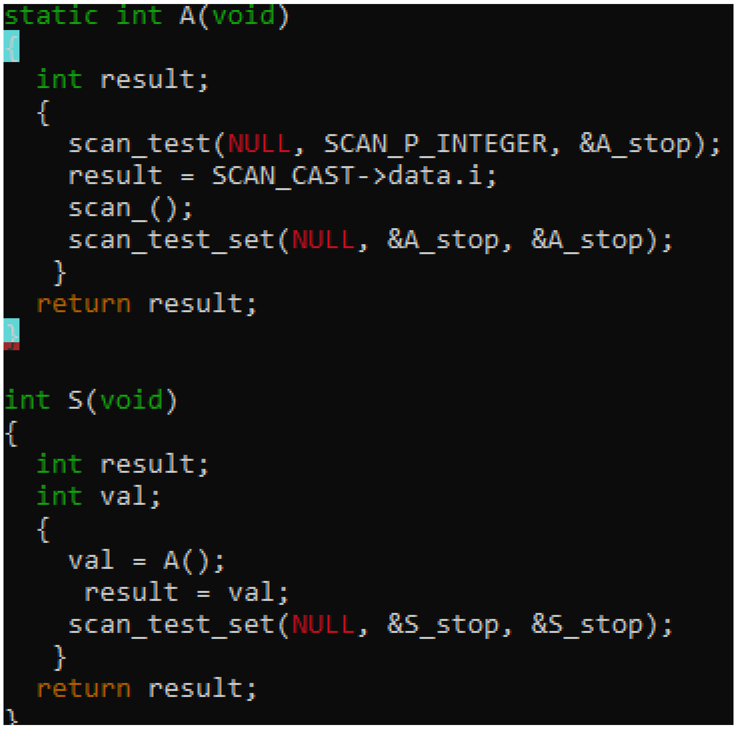
Inherited attributes are parameters of a parse function, so they allow data to be passed into a function, instead of data being returned to a function. I will include a usage example of this type of attribute in a later section.

In this example ‘val’ is to be assigned whatever value the token INTEGER matches with, however if ‘val’ were attached to a non terminal, it would take on the return value of the parse function associated with that non terminal.

basic.bnf:

```
S:int ::= A:val [* result = val; *].
A:int ::= INTEGER:result.
```

basic.str: 5



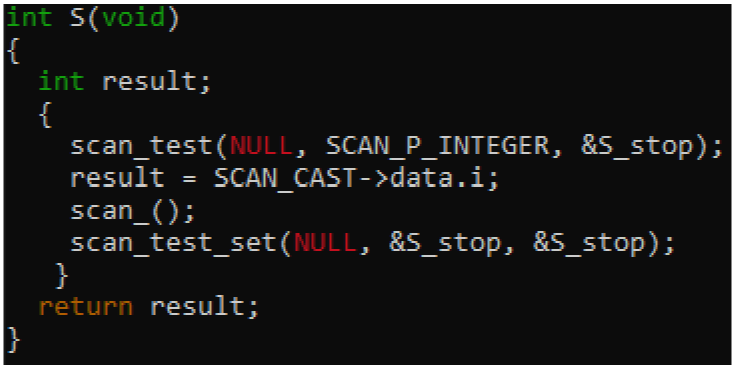
```
static int A(void)
{
  int result;
  {
    scan_test(NULL, SCAN_P_INTEGER, &A_stop);
    result = SCAN_CAST->data.i;
    scan_();
    scan_test_set(NULL, &A_stop, &A_stop);
   }
  return result;
}

int S(void)
{
  int result;
  int val;
  {
    val = A();
     result = val;
    scan_test_set(NULL, &S_stop, &S_stop);
   }
  return result;
}
```

Notice how now in ‘val’ takes the return value of A(). Notice also how in A() the value matched by INTEGER is being assigned directly to result, this is perfectly valid also.

basic.bnf:

```
S:int ::= INTEGER:result.
```

basic.str: 5



```
int S(void)
{
  int result;
  {
    scan_test(NULL, SCAN_P_INTEGER, &S_stop);
    result = SCAN_CAST->data.i;
    scan_();
    scan_test_set(NULL, &S_stop, &S_stop);
   }
  return result;
}
```

Final word count: 46911

Finally, I would like to refer back to the original example, and the C code wrapped between the square brackets and asterisks.

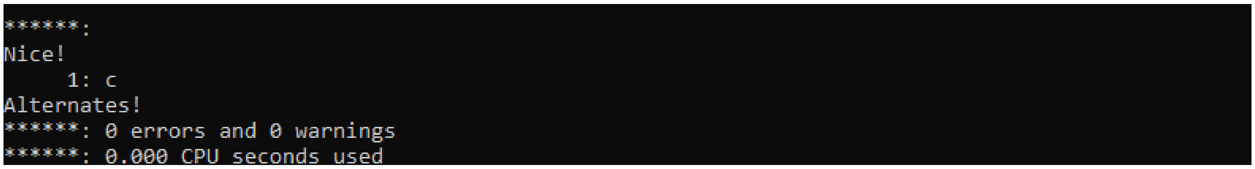
```
[* result = val; *]
```

C-code fragments wrapped in this type of enclosure are the semantic actions which can be attached to alternates of production rules. The code between these brackets is pasted directly into the parse function of the rule it belongs to. It’s exact placement in the function has an exact relation with how it is placed in the production rule.

basic.bnf:

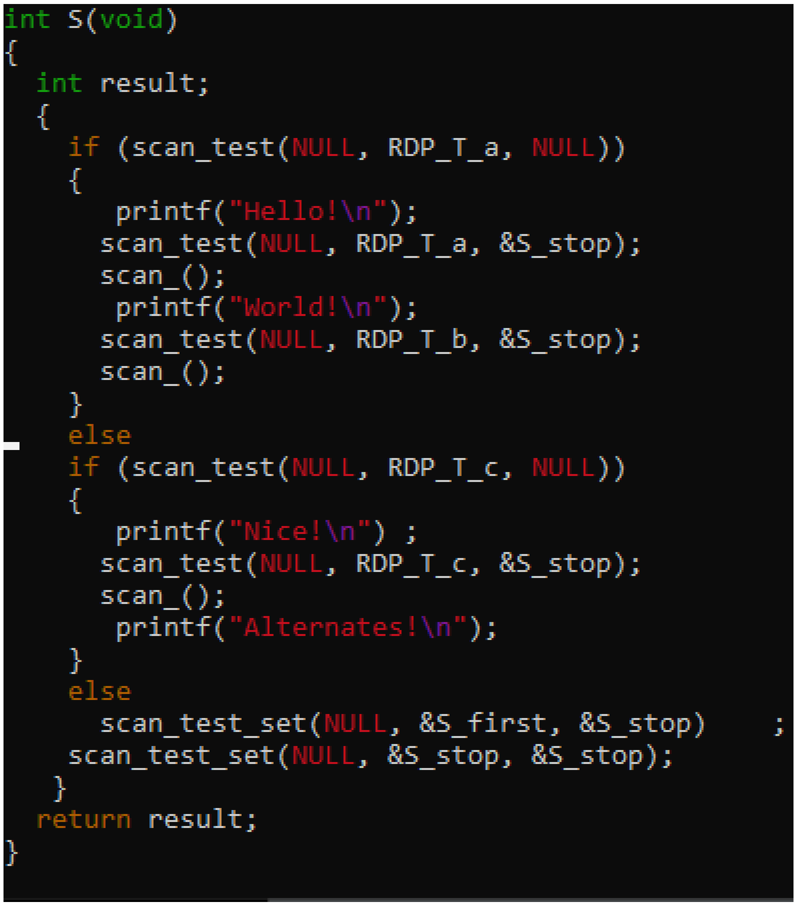
```
S:int ::= [*printf(“Hello!\n”);*] ‘a’ [*printf(“World\n”);*] ‘b’
    | [*printf(“Nice!\n”);*] ‘c’ [*printf(“Alternates\n”);*].
```

basic.str:



```
******:
Nice!
     1: c
Alternates!
******: 0 errors and 0 warnings
******: 0.000 CPU seconds used
```

Final word count: 46911



```
int S(void)
{
  int result;
  {
    if (scan_test(NULL, RDP_T_a, NULL))
    {
       printf("Hello!\n");
      scan_test(NULL, RDP_T_a, &S_stop);
      scan_();
       printf("World!\n");
      scan_test(NULL, RDP_T_b, &S_stop);
      scan_();
    }
    else
    if (scan_test(NULL, RDP_T_c, NULL))
    {
       printf("Nice!\n") ;
      scan_test(NULL, RDP_T_c, &S_stop);
      scan_();
       printf("Alternates!\n");
    }
    else
      scan_test_set(NULL, &S_first, &S_stop)    ;
    scan_test_set(NULL, &S_stop, &S_stop);
   }
  return result;
}
```

Of course these actions aren't useful in any way besides demonstrating my point, so let's look at a slightly more involved example.

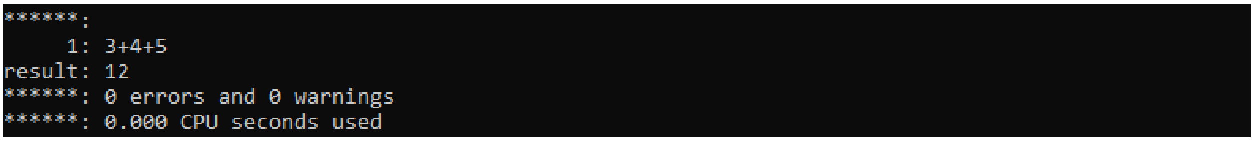
basic.bnf:

```
S:int ::= A:result [* printf("result: %i\n", result);  *].
A:int ::= D:result [ '+' A:val [* result += val; *]].
D:int ::= INTEGER:result.
```

basic.str:

```
3+4+5
```

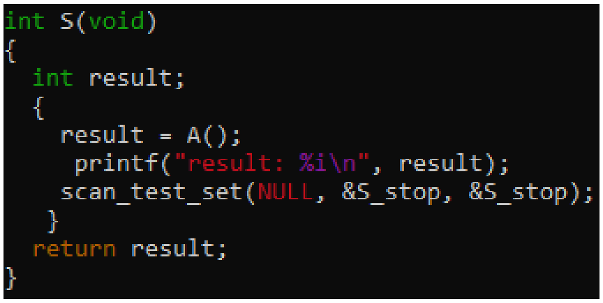
The above grammar will evaluate the sum "3 + 4 + 5" and print the result.



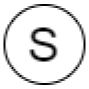
```
******:
     1: 3+4+5
result: 12
******: 0 errors and 0 warnings
******: 0.000 CPU seconds used
```

Now let's try to understand exactly how this is done.

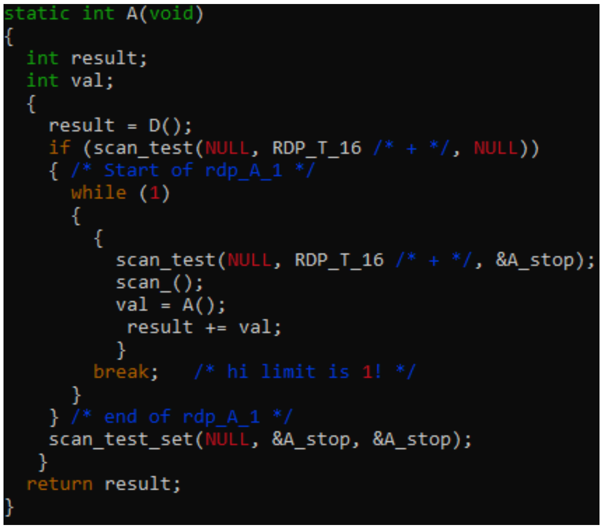
Final word count: 46911



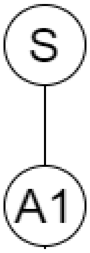
```
int S(void)
{
  int result;
  {
    result = A();
     printf("result: %i\n", result);
    scan_test_set(NULL, &S_stop, &S_stop);
   }
  return result;
}
```



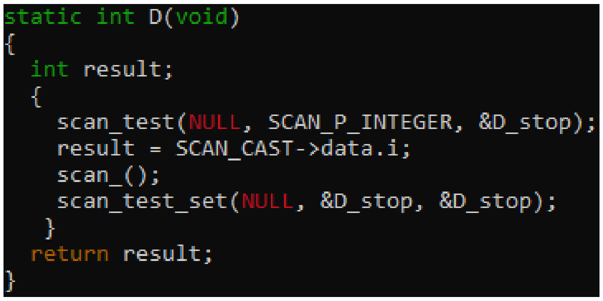
We begin in the parse function for the start symbol. ‘result’ is declared but as of yet has no value, thus we enter the parse function for A in order to obtain this value.



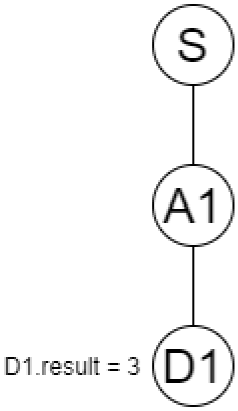
```
static int A(void)
{
  int result;
  int val;
  {
    result = D();
    if (scan_test(NULL, RDP_T_16 /* + */, NULL))
    { /* Start of rdp_A_1 */
      while (1)
      {
        {
          scan_test(NULL, RDP_T_16 /* + */, &A_stop);
          scan_();
          val = A();
           result += val;
          }
        break;   /* hi limit is 1! */
      }
    } /* end of rdp_A_1 */
    scan_test_set(NULL, &A_stop, &A_stop);
   }
  return result;
}
```



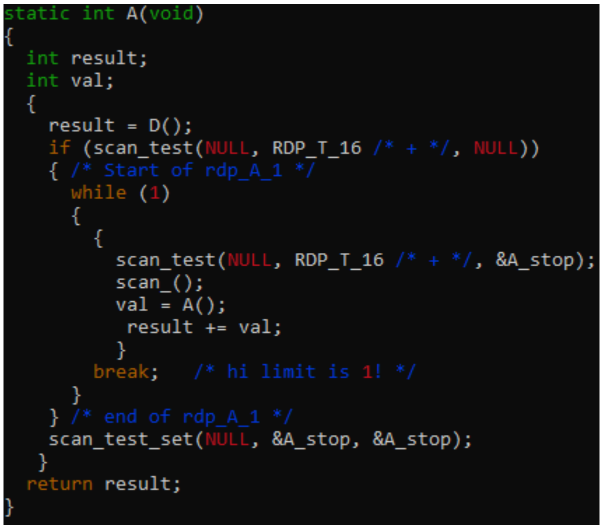
I will refer to this instance of A as A $_1$, where we begin by making a call to the parse function for D in order to obtain a value for the result of A$_1$.



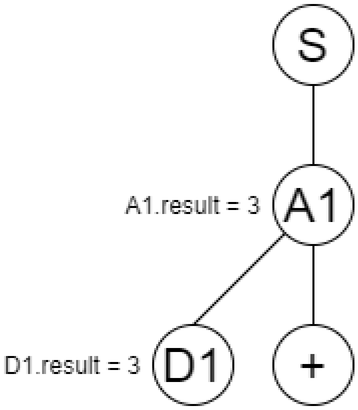
```
static int D(void)
{
  int result;
  {
    scan_test(NULL, SCAN_P_INTEGER, &D_stop);
    result = SCAN_CAST->data.i;
    scan_();
    scan_test_set(NULL, &D_stop, &D_stop);
   }
  return result;
}
```



In this first call to D, we match our first look ahead token ‘3’ with INTEGER and subsequently set the value of 3 as our result which is then returned to A_1.

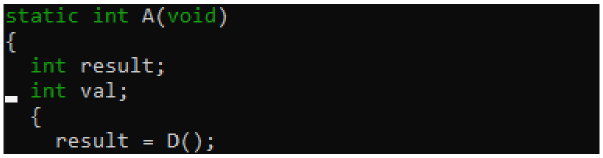


```
static int A(void)
{
  int result;
  int val;
  {
    result = D();
    if (scan_test(NULL, RDP_T_16 /* + */, NULL))
    { /* Start of rdp_A_1 */
      while (1)
      {
        {
          scan_test(NULL, RDP_T_16 /* + */, &A_stop);
          scan_();
          val = A();
           result += val;
          }
        break;   /* hi limit is 1! */
      }
    } /* end of rdp_A_1 */
    scan_test_set(NULL, &A_stop, &A_stop);
   }
  return result;
}
```

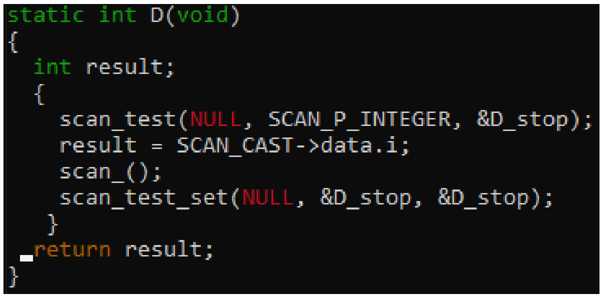


Returning to A_1 begins with 3 being assigned to the result of A_1. The current lookahead token ‘+’ is matched and A_1 makes a recursive call in order to determine a value for A_1.val.

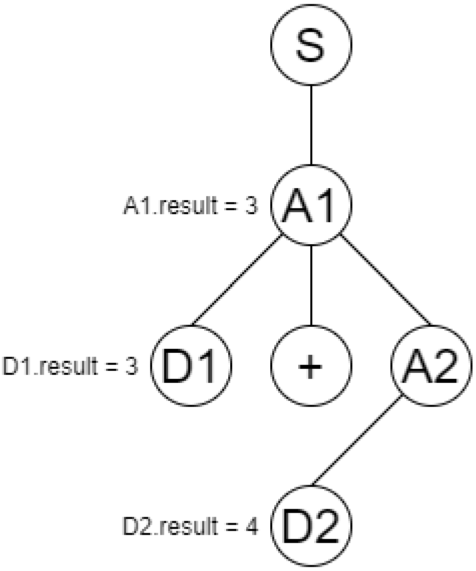
Final word count: 46911



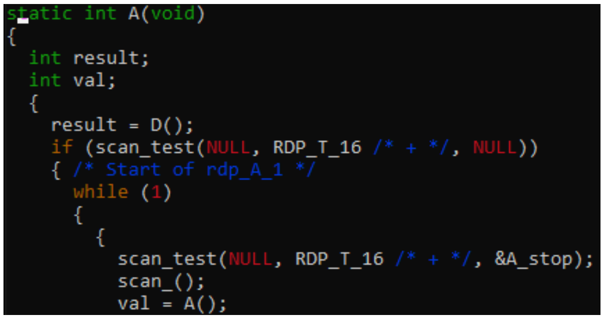
```
static int A(void)
{
  int result;
  int val;
  {
    result = D();
```



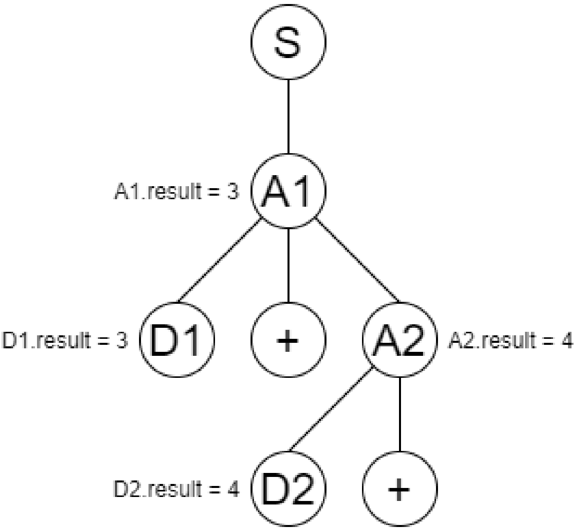
```
static int D(void)
{
  int result;
  {
    scan_test(NULL, SCAN_P_INTEGER, &D_stop);
    result = SCAN_CAST->data.i;
    scan_();
    scan_test_set(NULL, &D_stop, &D_stop);
   }
  return result;
}
```



In A_2 the second call to D() is made, which matches with ‘4’ and returns the result to A_2.

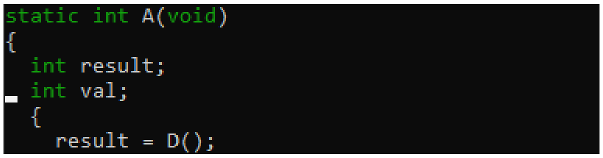


```
static int A(void)
{
  int result;
  int val;
  {
    result = D();
    if (scan_test(NULL, RDP_T_16 /* + */, NULL))
    { /* Start of rdp_A_1 */
      while (1)
      {
        {
          scan_test(NULL, RDP_T_16 /* + */, &A_stop);
          scan_();
          val = A();
```

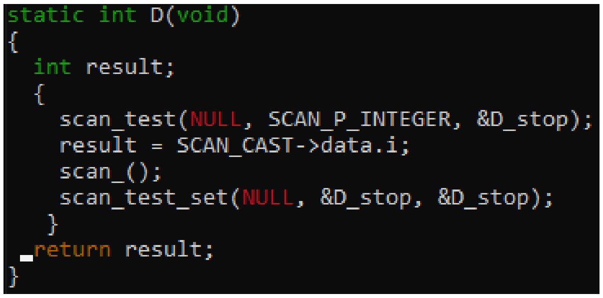


A_2 receives 4 as its result, matches ‘+’ and makes the third call to A()

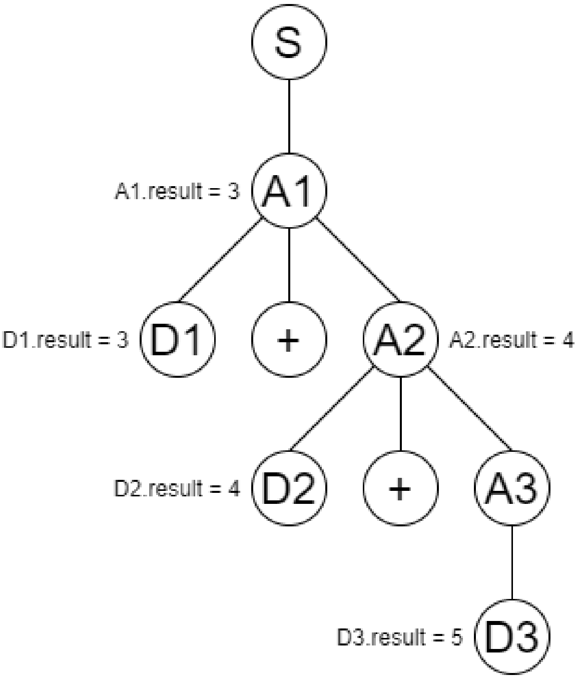
Final word count: 46911



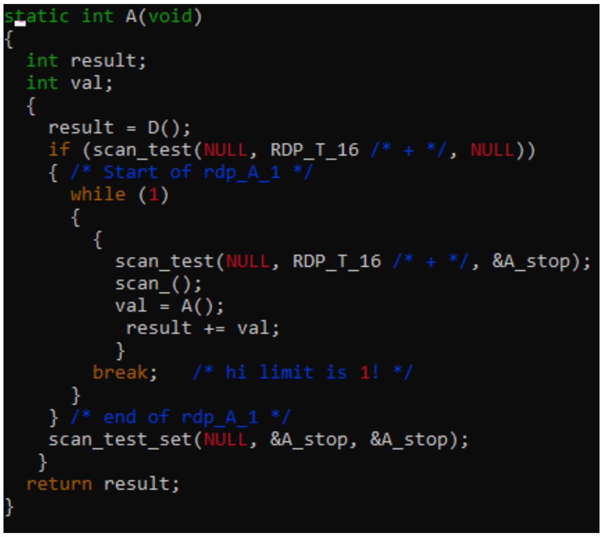
```
static int A(void)
{
  int result;
  int val;
  {
    result = D();
```



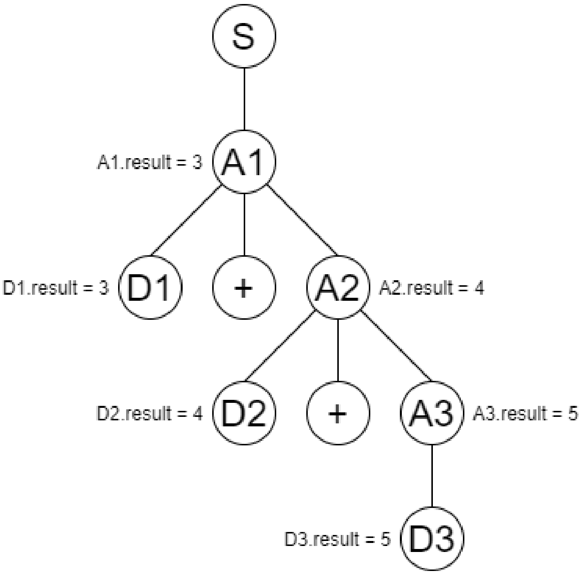
```
static int D(void)
{
  int result;
  {
    scan_test(NULL, SCAN_P_INTEGER, &D_stop);
    result = SCAN_CAST->data.i;
    scan_();
    scan_test_set(NULL, &D_stop, &D_stop);
   }
  return result;
}
```



A$_3$ now makes the third and final call to D(), D$_3$ matches with ‘5’ and returns the result A$_3$.

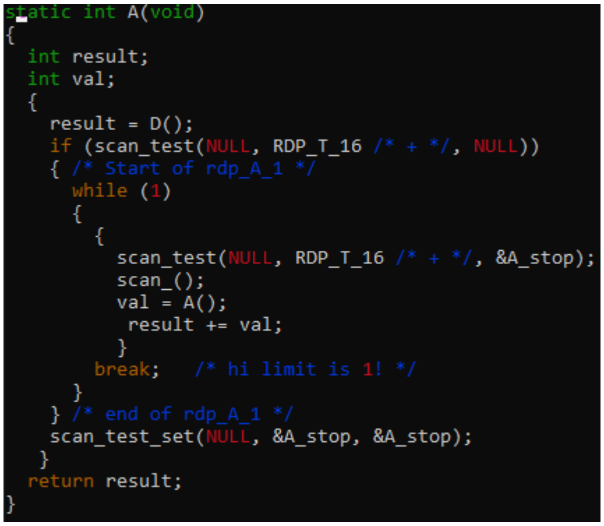


```
static int A(void)
{
  int result;
  int val;
  {
    result = D();
    if (scan_test(NULL, RDP_T_16 /* + */, NULL))
    { /* Start of rdp_A_1 */
      while (1)
      {
        {
          scan_test(NULL, RDP_T_16 /* + */, &A_stop);
          scan_();
          val = A();
           result += val;
          }
        break;   /* hi limit is 1! */
      }
    } /* end of rdp_A_1 */
    scan_test_set(NULL, &A_stop, &A_stop);
   }
  return result;
}
```

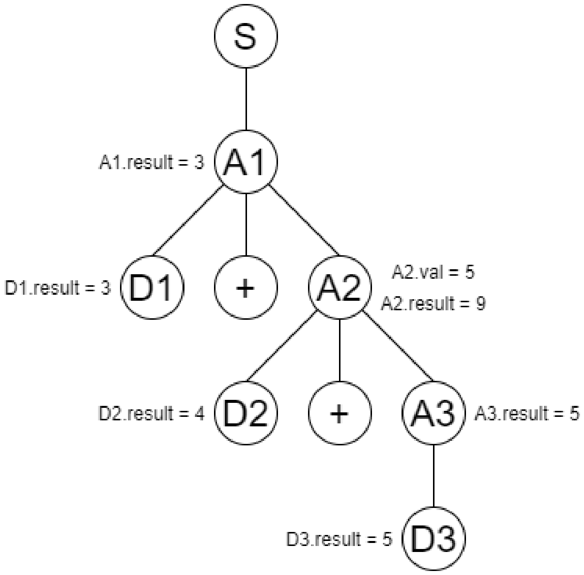


A$_3$ now holds 5 as its result, however unlike its predecessors A$_3$ has no ‘+’ to match against, so instead it returns it’s result of 5 to A$_2$.

Final word count: 46911

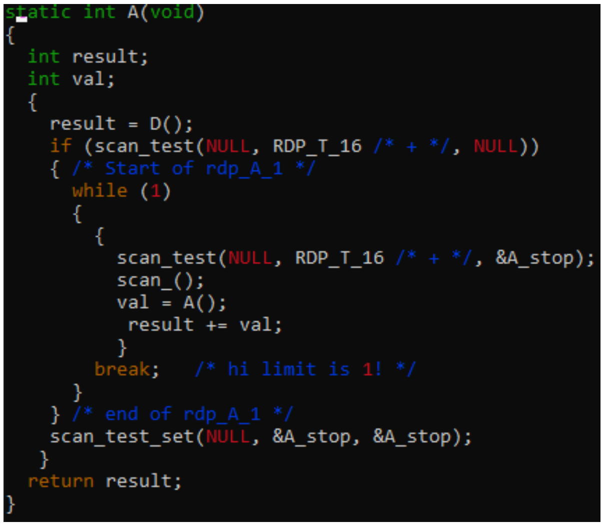


```
static int A(void)
{
  int result;
  int val;
  {
    result = D();
    if (scan_test(NULL, RDP_T_16 /* + */, NULL))
    { /* Start of rdp_A_1 */
      while (1)
      {
        {
          scan_test(NULL, RDP_T_16 /* + */, &A_stop);
          scan_();
          val = A();
           result += val;
          }
        break;   /* hi limit is 1! */
      }
    } /* end of rdp_A_1 */
    scan_test_set(NULL, &A_stop, &A_stop);
   }
  return result;
}
```

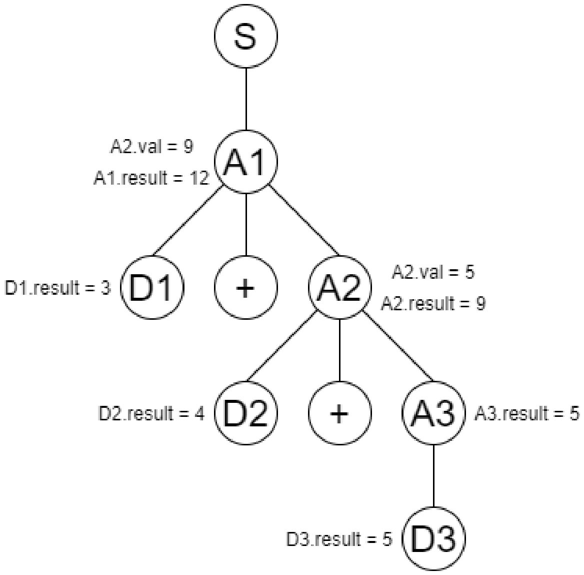


A_2.val now holds the value 5 which is then added to A_2.result, making its result now equal to 9. The while loop hits the unconditional break and A_2 returns the result of 9 to A_1.

Note: while loops created from square brackets in the grammar only ever make one single iteration.

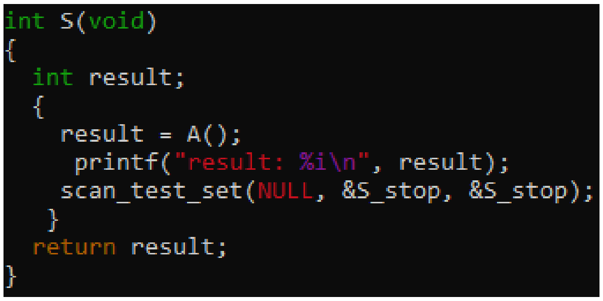


```
static int A(void)
{
  int result;
  int val;
  {
    result = D();
    if (scan_test(NULL, RDP_T_16 /* + */, NULL))
    { /* Start of rdp_A_1 */
      while (1)
      {
        {
          scan_test(NULL, RDP_T_16 /* + */, &A_stop);
          scan_();
          val = A();
           result += val;
          }
        break;   /* hi limit is 1! */
      }
    } /* end of rdp_A_1 */
    scan_test_set(NULL, &A_stop, &A_stop);
   }
  return result;
}
```

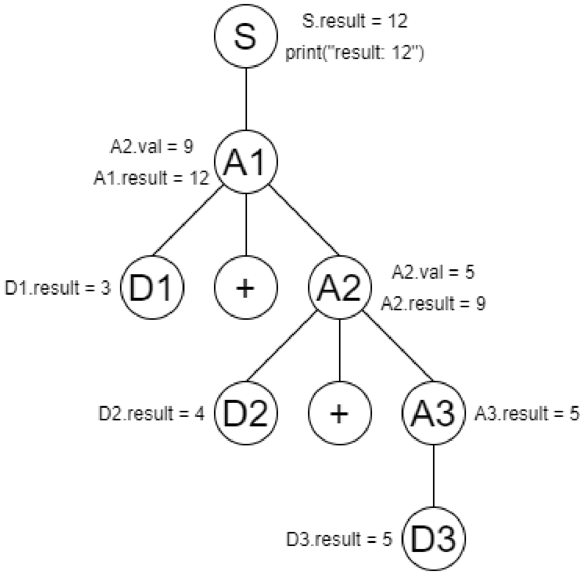


A_1.val now holds the value of 5 which is then added to A_1.result, making its result now equal to 12. A_1 breaks out of it's loop and returns the result of 12 to S.

Final word count: 46911



```
int S(void)
{
  int result;
  {
    result = A();
     printf("result: %i\n", result);
    scan_test_set(NULL, &S_stop, &S_stop);
   }
  return result;
}
```



We finally return to S whose result now holds the value of 12, which it then proceeds to print before returning to the main function where the program is successfully terminated.

Thus the parser was successfully able to perform the addition via the semantic actions attached to each alternate. It is important to understand the order in which the parser performs the derivation because this can be important when implementing semantics. In this case the order the semantics were executed in did not matter since addition is commutative, however it is worth noting that addition is also left associate, however this expression was actually evaluated from right to left (3 + (4 + 5)). We are limited by the fact the grammar can not be left recursive and when implementing other types of behaviour this can end up being a significant roadblock.

Another point i'd like to add is, although the semantics of addition have been implemented here to make this example as clear as possible, the line '`result += val`' could have just as easily been '`result *= val`', which would have resulted in the product of these three values rather than the sum. While this would have looked odd, it is important to understand that semantics are simply what they are defined to be, and should not necessarily be tied down to any particular convention. However in my case I am writing a C compiler, so i'll be sticking as close to the conventions as possible.

`Final word count: 46911`

Symbol Tables

There is one more concept I must cover before I start going over my revised grammar, and that is the concept of symbol tables. Symbol tables are data structures that are used by compilers to hold information about source-program constructs. Entries in the symbol table contain information about identifiers such as the character string it is represented by, it's data type, it's associated value, and some other information currently not relevant to the scope of this document.[29]

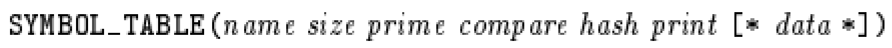
So far in this project specifically, symbol tables are used in the following ways:

- To ensure an identifier has been declared before it is used.
- To prevent re-declarations of identifiers.

I will cover the details of how this is achieved in the coming sections.

Symbol Tables in rdp

Symbol tables are yet another feature rdp provides. They are declared by the user in the **.bnf** file which contains their grammar. Symbol tables are declared using the following signature:



```
SYMBOL_TABLE(name size prime compare hash print [* data *])
```

The parameters refer to the following:

Name - A name which must be a valid C identifier. This will be the identifier for the table.

Size - The symbol table is implemented as a hash table, thus size refers to the number of hash buckets initially allocated to the table.

Prime - An integer prime number to be used in the hash function. This value is expected to be less than size, and coprime with size for best results.

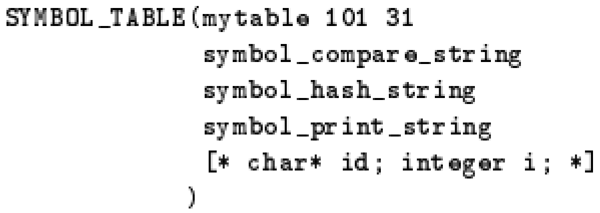
Compare - The name of a compare function, typically `‘symbol_compare_string’`

Hash - The name of a hash function, typically `‘symbol_hash_string’`

Print - the name of a print function, typically `‘symbol_print_string’`

[* data *] - A list of data fields to be associated with each identifier.[30]

In practice the following table definition would be suitable:



```
SYMBOL_TABLE(mytable 101 31
             symbol_compare_string
             symbol_hash_string
             symbol_print_string
             [* char* id; integer i; *]
            )
```

Figure 5. [30]

So this is a symbol table named “mytable” which contains 101 buckets and hases entries using the value 31. It uses the set of functions provided by rdp, and has data fields in which the string used to refer to the identifier, and the variable integer value associated with the identifier are both stored.

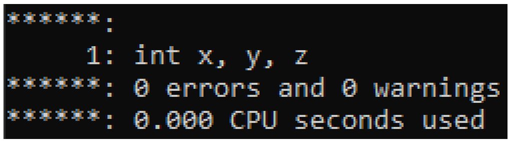
Let’s look at some usage examples:

basic.bnf:

```
SYMBOL_TABLE(mytable 101 31
             symbol_compare_string
             symbol_hash_string
             symbol_print_string
             [* char *id; *]
            )

S ::= [
        'int' ID:name
        [*
          symbol_insert_key(mytable,
                            &name,
                            sizeof(char*),
                            sizeof(mytable_data));
        *]
        {
         ',' ID:name
         [*
           symbol_insert_key(mytable,
                             &name,
                             sizeof(char*),
                             sizeof(mytable_data));
         *]
        }
      ].
```

Final word count: 46911

basic.str: int x, y, z



```
******:
     1: int x, y, z
******: 0 errors and 0 warnings
******: 0.000 CPU seconds used
```

Above is a grammar which permits declaration lists of arbitrary length. Currently the symbol table only holds one data field which refers to the names of the identifiers. In this case those names are x, y and z. We are using the following function to insert each entry into the table.

```
void *symbol_insert_key(const void *table, char *str, size_t size, size_t symbol_size)
```

The parameters refer to the following:

void *table - A reference to the symbol table where the new identifiers are to be inserted. In this case this is “mytable”.

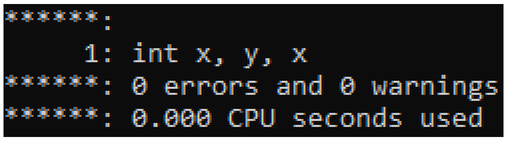
char *str - A reference to the string which is the name of the identifier. In this case these are x, y and z as previously stated.

size_t size - The size in bytes of the names reference. In a typical case, this will be the size of a char*.

size_t symbol_size - The size in bytes of the table entry. In a typical case this will be sizeof(YOUR_TABLES_NAME_data).

Additionally, the table returns a reference to the table entry itself.

These semantics however are currently only inserting entries into the table, however they are not performing any checks preventing duplicate names.



```
******:
     1: int x, y, x
******: 0 errors and 0 warnings
******: 0.000 CPU seconds used
```

x has been redeclared in this case, but no error is thrown. Clearly the semantics must be modified to account for this.

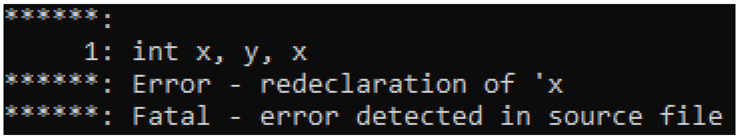
Final word count: 46911

basic.bnf

```
SYMBOL_TABLE(mytable 101 31
             symbol_compare_string
             symbol_hash_string
             symbol_print_string
             [* char *id; *]
            )

S ::= [
       'int' ID:name [* if (symbol_lookup_key(mytable, &name, NULL)) {
                                text_message(TEXT_ERROR, "redeclaration of '%s\n", name);
                        } else {
                                symbol_insert_key(mytable, &name, sizeof(char*), sizeof(mytable_data));
                        }
                     *]

       { ',' ID:name [* if (symbol_lookup_key(mytable, &name, NULL)) {
                                text_message(TEXT_ERROR, "redeclaration of '%s\n", name);
                        } else {
                                symbol_insert_key(mytable, &name, sizeof(char*), sizeof(mytable_data));
                        }
                     *]
       }
      ].
```

Basic.str

```
int x, y ,x
```

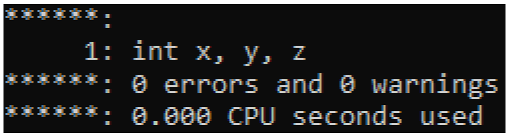


```
******:
     1: int x, y, x
******: Error - redeclaration of 'x
******: Fatal - error detected in source file
```

With these modified semantics, the parser is now able to detect redeclaration, and displays an error message detailing the issue.

Basic.str

```
int x, y ,z
```



```
******:
     1: int x, y, z
******: 0 errors and 0 warnings
******: 0.000 CPU seconds used
```

Upon removal of the redeclaration, the error no longer occurs.

Final word count: 46911

Let’s examine how exactly this has been achieved.

```
if (symbol_lookup_key(mytable, &name, NULL)) {
    text_message(TEXT_ERROR, "redeclaration of '%s\n", name);
} else {
    symbol_insert_key(mytable, &name, sizeof(char*), sizeof(mytable_data));
}
```

We have the following function acting as the condition to our selection statement.

```
void *symbol_lookup_key(const void *table, char *key, void *scope)
```

This function has mostly similar parameters to `symbol_insert_key.` In particular we are concerned with ‘key’ and the return value. This function, based on the value of ‘key’ will either return a reference to an existing table entry or NULL. In other words, if a key had previously been inserted into the table via the ‘str’ parameter of `symbol_insert_key`, then a table entry will exist for that key, indicating that an identifier with that name has already been declared. In the event that this does happen, the parser issues a TEXT_ERROR which explains that an identifier redeclaration has occurred. If the key does not already exist, then the new entry is simply inserted since no conflict was found.

So these semantics stop identifier redeclarations, but what if we wanted to stop the use of undeclared identifiers.

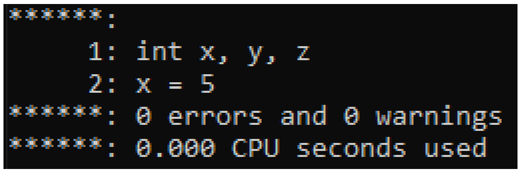
Final word count: 46911

basic.bnf

```
SYMBOL_TABLE(mytable 101 31
             symbol_compare_string
             symbol_hash_string
             symbol_print_string
             [* char *id; *]
)

S ::= { DECLARE | ASSIGN }.

DECLARE ::= 'int' ID:name [* if (symbol_lookup_key(mytable, &name, NULL)) {
                                text_message(TEXT_ERROR, "redeclaration of '%s\n", name);
                             } else {
                                symbol_insert_key(mytable, &name, sizeof(char*), sizeof(mytable_data));
                             }
                          *]

            { ',' ID:name [* if (symbol_lookup_key(mytable, &name, NULL)) {
                                text_message(TEXT_ERROR, "redeclaration of '%s\n", name);
                             } else {
                                symbol_insert_key(mytable, &name, sizeof(char*), sizeof(mytable_data));
                             }
                          *]
            }.

ASSIGN ::= ID:name '=' INTEGER [* if (!symbol_lookup_key(mytable, &name, NULL)) {
                                          text_message(TEXT_ERROR, "'%s' undeclared\n", name);
                                  }
                               *] .
```

Basic.str

```
int x, y ,z
x = 5;
```



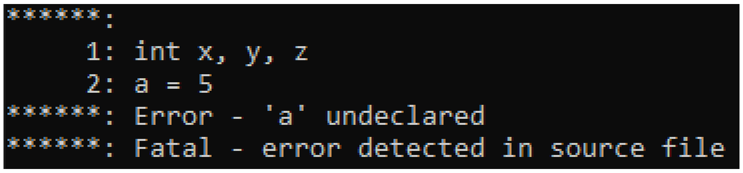
```
******:
     1: int x, y, z
     2: x = 5
******: 0 errors and 0 warnings
******: 0.000 CPU seconds used
```

We have a new rule ‘ASSIGN’ which is intended to allow the assignment of integer values to existing variables. In reality the nothing is actually being assigned due to the semantics not yet being implemented, but I will address that in a moment.

Final word count: 46911

```
if (!symbol_lookup_key(mytable, &name, NULL)) {
    text_message(TEXT_ERROR, "'%s' undeclared\n", name);
}
```

These are currently the semantic actions tied to ASSIGN. In this case if a table entry does not exist anb error message is displayed explaining that the identifier in question has yet to be declared.



```
******:
     1: int x, y, z
     2: a = 5
******: Error - 'a' undeclared
******: Fatal - error detected in source file
```

So now we have a parser which can recognize when identifiers are being redeclared and when undeclared identifiers are used. But what if we wanted to access the values we assign?

Final word count: 46911

basic.bnf

```
SYMBOL_TABLE(mytable 101 31
             symbol_compare_string
             symbol_hash_string
             symbol_print_string
             [* char *id; integer i; *]
            )

S ::= { DECLARE | ASSIGN | PRINT }.

DECLARE ::= 'int' ID:name [* if (symbol_lookup_key(mytable, &name, NULL)) {
                                text_message(TEXT_ERROR, "redeclaration of '%s\n", name);
                             } else {
                                symbol_insert_key(mytable, &name, sizeof(char*), sizeof(mytable_data));
                             }
                          *]

            { ',' ID:name [* if (symbol_lookup_key(mytable, &name, NULL)) {
                                text_message(TEXT_ERROR, "redeclaration of '%s\n", name);
                             } else {
                                symbol_insert_key(mytable, &name, sizeof(char*), sizeof(mytable_data));
                             }
                          *]
            }.

ASSIGN ::= ID:name '=' PRIMITIVE:val [* if (!symbol_lookup_key(mytable, &name, NULL)) {
                                           text_message(TEXT_ERROR, "'%s' undeclared\n", name);
                                        } else {
                                           mytable_cast(symbol_lookup_key(mytable, &name, NULL))->i = val;
                                        }
                                     *].

PRINT ::= 'print' PRIMITIVE:val [* printf("%i\n", val);  *].

PRIMITIVE:int ::= INTEGER:result |
                  ID:name [* if (!symbol_lookup_key(mytable, &name, NULL)) {
                                     text_message(TEXT_ERROR, "'%s' undeclared\n", name);
                                 } else {
                                     result = mytable_cast(symbol_lookup_key(mytable, &name, NULL))->i;
                                 }
                          *].
```

Basic.str

```
int x, y, z
x = 5
y = x
print y
```

Final word count: 46911

The grammar has started to grow quite significantly, but we can review the individual changes that have been made in order to understand this more easily.

```
SYMBOL_TABLE(mytable 101 31
             symbol_compare_string
             symbol_hash_string
             symbol_print_string
             [* char *id; integer i; *]
            )
```

To begin with, the symbol table now has fields for storing both the name of the identifier and an integer value associated with it.

```
S ::= { DECLARE | ASSIGN | PRINT }.
```

It is now possible to either declare a list of variables, assign a value to an existing variable or print a primitive value.

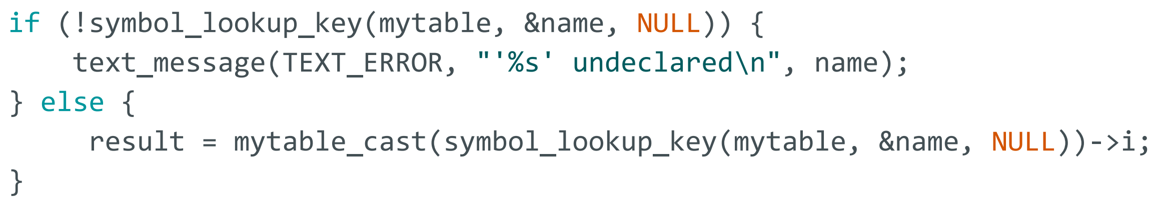
The semantics surrounding DECLARE remain unchanged, however ASSIGN has undergone some changes.

```
ASSIGN ::= ID:name '=' PRIMITIVE:val
```

Ignoring the semantics for the moment, we see that we are now assigning PRIMITIVE rather than INTEGER. Additionally, ‘PRIMITIVE’ has a synthesized attribute ‘val’ associated with it, where previously INTEGER did not.

```
PRIMITIVE:int ::= INTEGER:result | ID:name
```

A primitive can either be an integer literal or an identifier, meaning both can be assigned. Let’s take a look at the semantics attached to primitive.



```
if (!symbol_lookup_key(mytable, &name, NULL)) {
    text_message(TEXT_ERROR, "'%s' undeclared\n", name);
} else {
     result = mytable_cast(symbol_lookup_key(mytable, &name, NULL))->i;
}
```

These are the semantics specifically associated with the second alternate of primitive, ‘`ID:name`’. Whenever this alternate is used, the same check as with assignment is performed to verify that the identifier was previously declared. If this is the case, we are able to access the value associated with the identifier via the reference to the table entry returned by the symbol lookup function. Because the lookup function returns a **void***, the reference must be type cast before the integer field can be accessed. The value is subsequently stored in ‘result’

```
ASSIGN ::= ID:name '=' PRIMITIVE:val
```

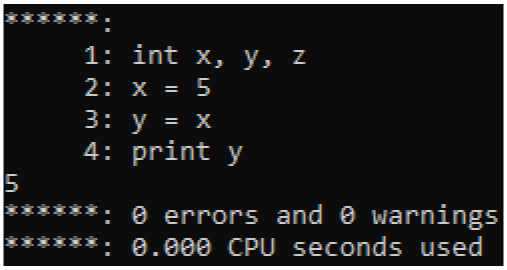
Now we understand the semantics attached to primitive, we know the value to be assigned here is either that of an integer literal or some integer stored in an existing variable.

```
if (!symbol_lookup_key(mytable, &name, NULL)) {
    text_message(TEXT_ERROR, "'%s' undeclared\n", name);
} else {
    mytable_cast(symbol_lookup_key(mytable, &name, NULL))->i = val;
}
```

These are the semantics associated with assignment. We are running the same check as before and additionally, if the identifier is found to exist, the val is then stored in the integer field of the table entry. This means these values are now stored and accessible at any time by the parser.

```
PRINT ::= 'print' PRIMITIVE:val [* printf("%i\n", val);  *].
```

The print rule is fairly simple, and simply prints integers using the C ‘`printf`’ function.



```
******:
     1: int x, y, z
     2: x = 5
     3: y = x
     4: print y
5
******: 0 errors and 0 warnings
******: 0.000 CPU seconds used
```

Final word count: 46911

My Improved Grammar

```
SYMBOL_TABLE(global 101 31
           symbol_compare_string
           symbol_hash_string
           symbol_print_string
           [* char* id; integer i; *]
          )

TRANSLATION_UNIT ::= { DECLARATION | STATEMENT } .

DECLARATION:int  ::= TYPE_SPECIFIER ID:name [ '=' EXPRESSION:result ] ';'
                  [* if (symbol_lookup_key(global, &name, NULL))
                       text_message(TEXT_ERROR, "redeclaration of '%s'\n", name);
                     } else {
                       global_cast(symbol_insert_key(global, &name, sizeof(char*), sizeof(global_data)))->i = result;
                     }
                    *].

TYPE_SPECIFIER   ::= 'int'.

STATEMENT:int    ::= ID:name
                    [* if (!symbol_lookup_key(global, &name, NULL))
                        text_message(TEXT_ERROR, "'%s' undeclared\n", name);
                      else
                         result = global_cast(symbol_lookup_key(global, &name, NULL))->i;
                  *]
                    [ '=' EXPRESSION:result
                       [* global_cast(symbol_lookup_key(global, &name, NULL))->i = result; *]
                    ] ';'.

EXPRESSION:int   ::= BOOL:result.

BOOL:int         ::= SUM:result
                    { '==' SUM:val [* result = result == val; *] |
                      '!=' SUM:val [* result = result != val; *] |
                      '>=’ SUM:val [* result = result >= val; *] |
                      '<=' SUM:val [* result = result <= val; *] |
                      '>'  SUM:val [* result = result >  val; *] |
                      '<'  SUM:val [* result = result <  val; *]
                    }.

SUM:int          ::= PROD:result
                  { '+' PROD:val [* result += val; *] |
                    '-' PROD:val [* result -= val; *]
                   }.

PROD:int         ::= PRIMITIVE:result
                    { '*' PRIMITIVE:val [* result *= val; *] }.

PRIMITIVE:int    ::= INTEGER:result |
                     ID:name [* if (!symbol_lookup_key(global, &name, NULL))
                                     text_message(TEXT_ERROR, "'%s' undeclared\n", name);
                                 else
                                     result = global_cast(symbol_lookup_key(global, &name, NULL))->i;
                              *].
```

Above is my current and improved grammar. Since there are aspects of this grammar which I have already covered in great detail in previous sections, I will not be going over those aspects again, at least not in any detail. There are however issues which this grammar resolves which I will be covering in this section, as well as breaking down exactly what this grammar permits.

Final word count: 46911

The functionality of this grammar includes:

- Declaration of integer variables.
- Value assignment to existing variables via expressions.
- Prohibition of variable redeclaration and the use of undeclared variables.
- Evaluation of boolean expressions through both equalities and inequalities.
- Evaluation of arithmetic expressions. Expressions may include a mix of:
    - Addition
    - Subtraction
    - Multiplication
- Expressions are evaluated correctly according to conventional arithmetic operator semantics and operator priority.

```
SYMBOL_TABLE(global 101 31
        symbol_compare_string
        symbol_hash_string
        symbol_print_string
        [* char* id; integer i; *]
       )
```

To begin with, this grammar uses the exact same hash table as in the previous section which allows for the storage of identifier names and integer values associated with them. This table has been named ‘global’ to refer to the fact it exists in the global scope.

```
TRANSLATION_UNIT ::= { DECLARATION | STATEMENT }.
DECLARATION:int  ::= TYPE_SPECIFIER ID:name [ '=' EXPRESSION:result ].
STATEMENT:int    ::= ID:name [ '=' EXPRESSION:result ] ';.
```

The translation unit now allows for an arbitrary number of declarations and statements. This grammar does not support declaration lists, however it does support optional value assignment to variables upon declaration. A statement is similar to a declaration, only that a statement simply allows optional value assignment to an existing variable.

The semantics attached to declaration ensure no variables are ever redeclared, and additionally assign the result of an expression to the newly declared identifiers integer field within the symbol table. Statements are the same, except they check they verify the identifier’s existence rather than its non-existence.

```
EXPRESSION:int   ::= BOOL:result.

BOOL:int         ::= SUM:result
                     { '==' SUM:val [* result = result == val; *] |
                       '!=' SUM:val [* result = result != val; *] |
                       '>=’ SUM:val [* result = result >= val; *] |
                       '<=' SUM:val [* result = result <= val; *] |
                       '>'  SUM:val [* result = result >  val; *] |
                       '<'  SUM:val [* result = result <  val; *]
                     }.

SUM:int          ::= PROD:result
                     { '+' PROD:val [* result += val; *] |
                       '-' PROD:val [* result -= val; *]
                     }.

PROD:int         ::= PRIMITIVE:result
                     { '*' PRIMITIVE:val [* result *= val; *] }.

PRIMITIVE:int    ::= INTEGER:result | ID:name
```

Primitive’s are as they were in the previous section with the exact same semantics.

Expressions have been altered significantly and now evaluate correctly according to the conventional arithmetic operator semantics and operator priority. As before. Expressions derive both boolean and arithmetic expressions, so to conform with the standard C semantics of evaluating **non zero** and **zero** values as either **true** or **false**.

```
SUM:int          ::= PROD:result
                     { '+' PROD:val [* result += val; *] |
                       '-' PROD:val [* result -= val; *]
                     }.
```

I want to ignore boolean expressions and multiplication for a moment and talk about how the semantics of SUM successfully perform left associative addition and subtraction.

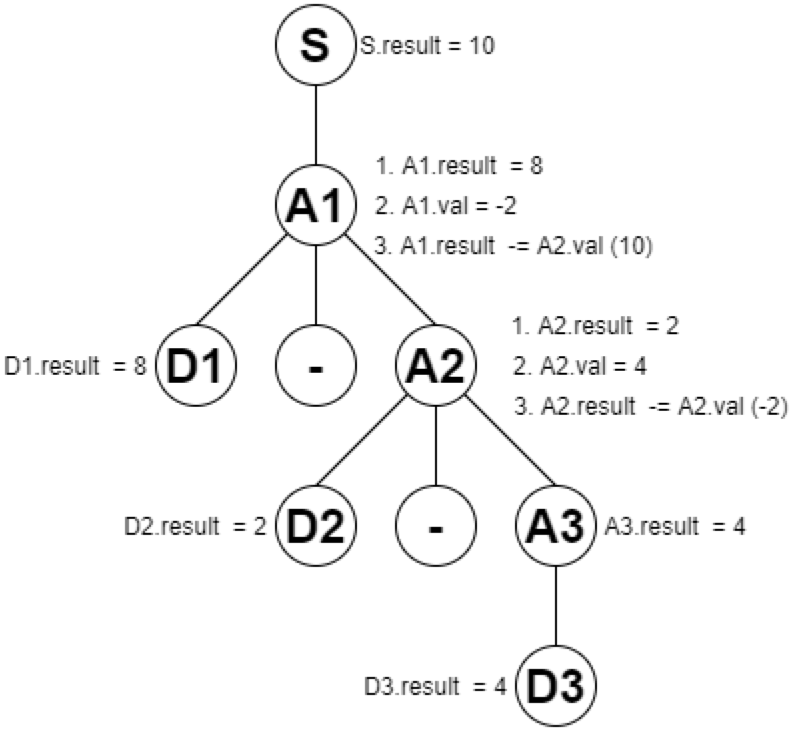
basic.bnf:

```
S:int ::= A:result [* printf("result: %i\n", result);  *].
A:int ::= D:result [ '-' A:val [* result -= val; *]].
D:int ::= INTEGER:result.
```

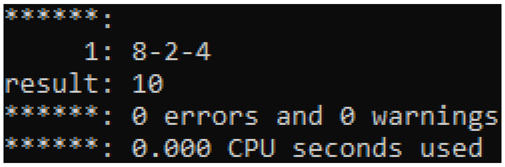
basic.str:

```
8-2-4
```

Let’s assume for a moment we were working with the following grammar. The expected result of the above expressions should be 2, however when evaluated by the semantics above, the result is 10. Here is a parse tree detailing why.



The expression is evaluated as 8 - (2 - 4), which is right associative, and incorrect according to convention.

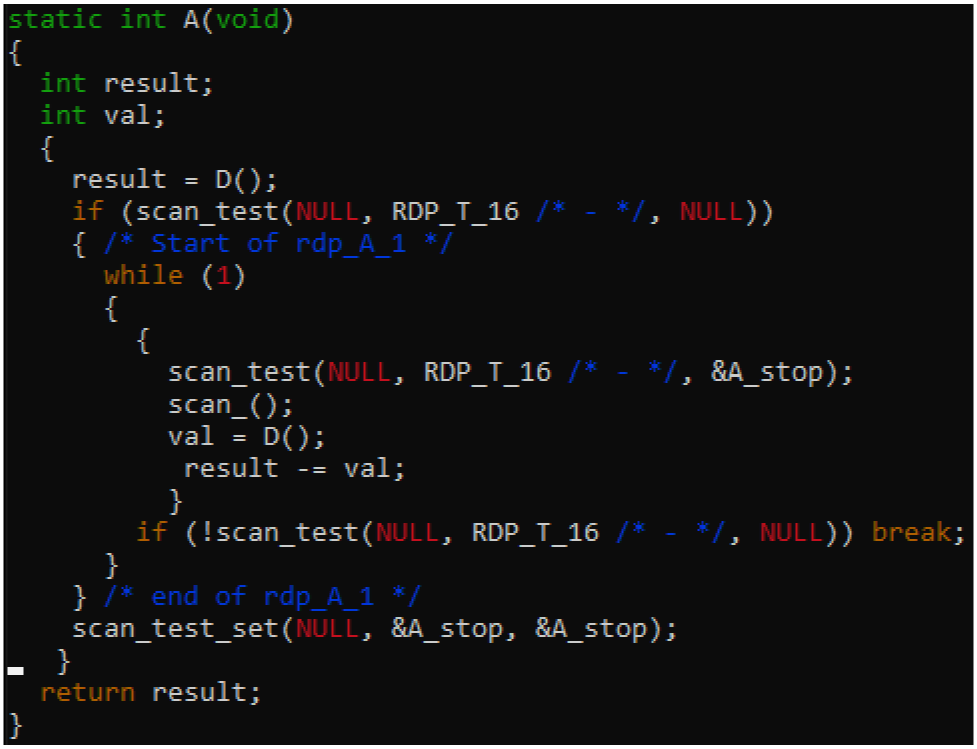


```
******:
     1: 8-2-4
result: 10
******: 0 errors and 0 warnings
******: 0.000 CPU seconds used
```

Final word count: 46911

```
basic.bnf:   S:int ::= A:result [* printf("result: %i\n", result);  *].
             A:int ::= D:result { '-' D:val [* result -= val; *]}.
             D:int ::= INTEGER:result.

basic.str:   8-2-4
```

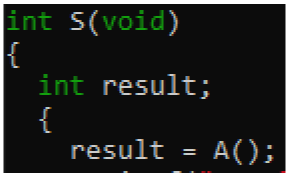
This grammar however implements the semantics as they are implemented in my current grammar. If you are to recall back to the section on basic rdp usage, I commented on the importance of how the Kleene closure has been implemented in rdp using a while loop.



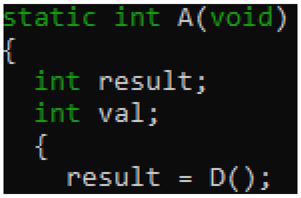
```
static int A(void)
{
  int result;
  int val;
  {
    result = D();
    if (scan_test(NULL, RDP_T_16 /* - */, NULL))
    { /* Start of rdp_A_1 */
      while (1)
      {
        {
          scan_test(NULL, RDP_T_16 /* - */, &A_stop);
          scan_();
          val = D();
           result -= val;
          }
        if (!scan_test(NULL, RDP_T_16 /* - */, NULL)) break;
      }
    } /* end of rdp_A_1 */
    scan_test_set(NULL, &A_stop, &A_stop);
   }
  return result;
}
```

With this implementation, the result is initially set to 8 as before, however now result is updated through iterations of the while loop, thus 2 is first subtracted from 8 before 4 is subtracted from the result of 8 - 2, making the evaluation left associative. Let’s examine this process closer.

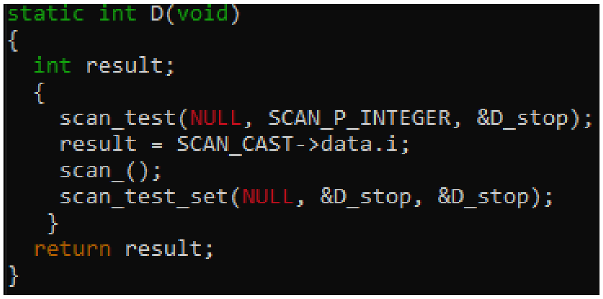
Final word count: 46911



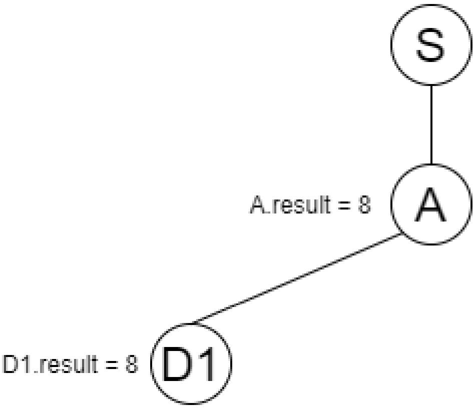
```
int S(void)
{
  int result;
  {
    result = A();
```



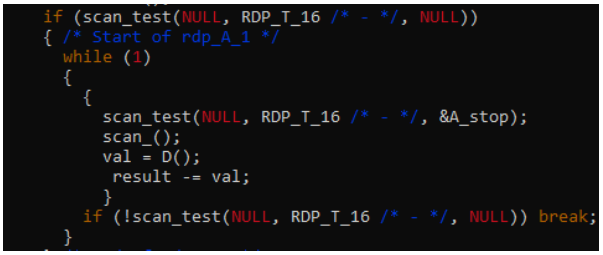
```
static int A(void)
{
  int result;
  int val;
  {
    result = D();
```



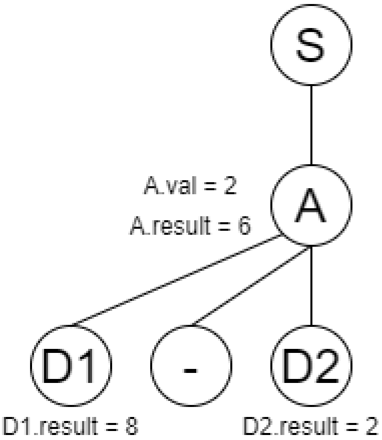
```
static int D(void)
{
  int result;
  {
    scan_test(NULL, SCAN_P_INTEGER, &D_stop);
    result = SCAN_CAST->data.i;
    scan_();
    scan_test_set(NULL, &D_stop, &D_stop);
   }
  return result;
}
```



We begin with S calling A which makes its first call to D in order to acquire its initial result of 8.

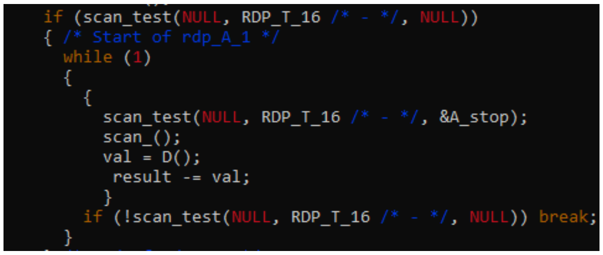


```
    if (scan_test(NULL, RDP_T_16 /* - */, NULL))
    { /* Start of rdp_A_1 */
      while (1)
      {
        {
          scan_test(NULL, RDP_T_16 /* - */, &A_stop);
          scan_();
          val = D();
           result -= val;
          }
        if (!scan_test(NULL, RDP_T_16 /* - */, NULL)) break;
      }
```

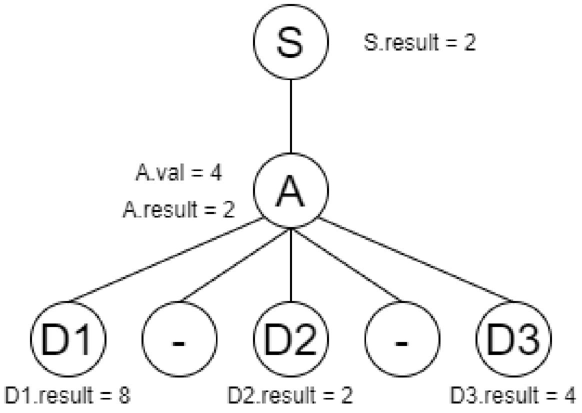


A then confirms the current lookahead token to be '-' and enters it's loop. D is called which returns the value 2 to A.val. A.val is subtracted from A.result, making A.result equal 2. A Test is performed to confirm that the current lookahead symbol isn't '-'. The symbol is however '-' so the loop goes around once more.

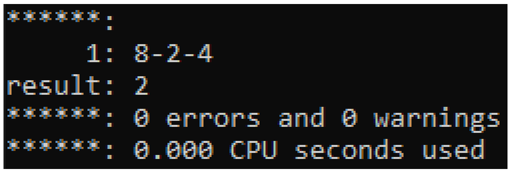
Final word count: 46911



```
  if (scan_test(NULL, RDP_T_16 /* - */, NULL))
  { /* Start of rdp_A_1 */
    while (1)
    {
      {
        scan_test(NULL, RDP_T_16 /* - */, &A_stop);
        scan_();
        val = D();
         result -= val;
        }
      if (!scan_test(NULL, RDP_T_16 /* - */, NULL)) break;
    }
```



The second '-' is matched and D is called which returns the value 4 to A.val. Once again A.val is subtracted from A.result, making A.result equal to 2. This time the lookahead token is no longer '-' so the loop is broken and the result of 2 is returned to S which prints the result.



```
******:
     1: 8-2-4
result: 2
******: 0 errors and 0 warnings
******: 0.000 CPU seconds used
```

Thus correctly evaluating the expression and demonstrating how right associative arithmetic can be implemented in rdp.

basic.bnf:

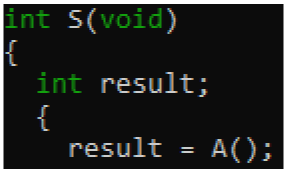
```
S:int ::= A:result [* printf("result: %i\n", result);  *].
A:int ::= B:result { '+' B:val [* result += val; *]}.
B:int ::= D:result { '*' D:val [* result *= val; *]}.
D:int ::= INTEGER:result.
```

basic.str:

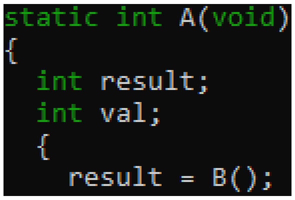
```
6+3*2
```

I will now demonstrate how my parser achieves correct order of operations. Once again I have isolated the functionality into a separate grammar so as to simplify the demonstration.

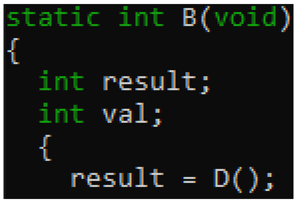
The idea behind how this works is that the parse functions which evaluate the subexpressions containing the higher priority operators are further down in the call order. This means that when the parse functions are returning, the subexpressions containing the higher priority operators are evaluated first because the functions containing that logic were called last. Let's examine this



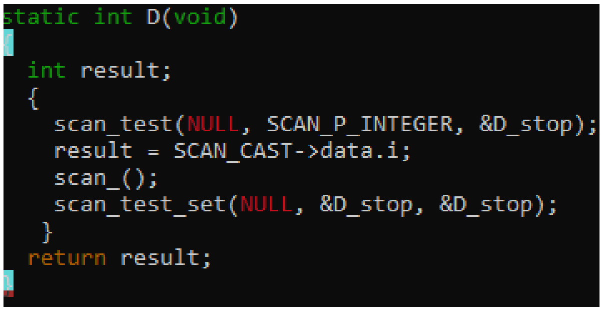
```
int S(void)
{
  int result;
  {
    result = A();
```



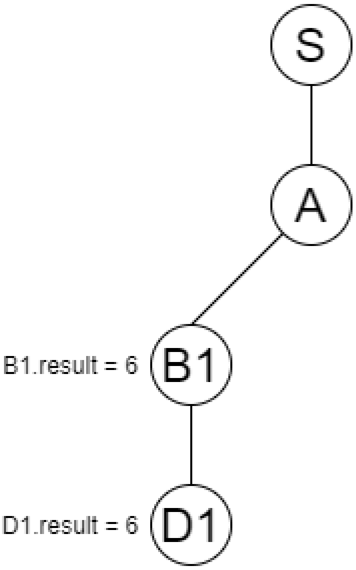
```
static int A(void)
{
  int result;
  int val;
  {
    result = B();
```



```
static int B(void)
{
  int result;
  int val;
  {
    result = D();
```

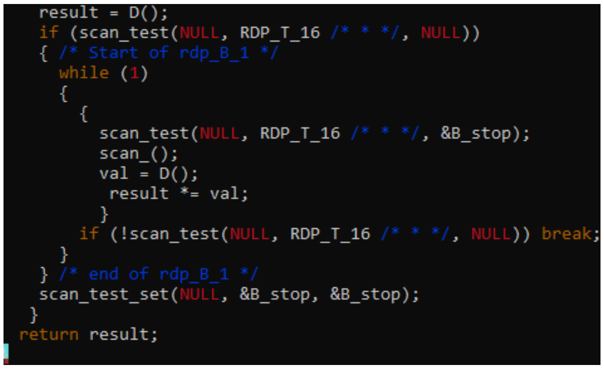


```
static int D(void)
{
  int result;
  {
    scan_test(NULL, SCAN_P_INTEGER, &D_stop);
    result = SCAN_CAST->data.i;
    scan_();
    scan_test_set(NULL, &D_stop, &D_stop);
  }
  return result;
}
```

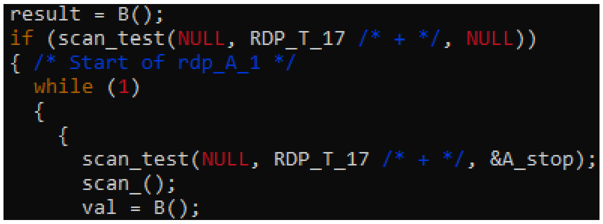


S calls A which calls B which calls D. D matches ‘6’ and returns the value to B.result.

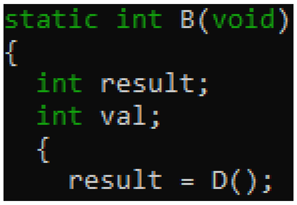
Final word count: 46911



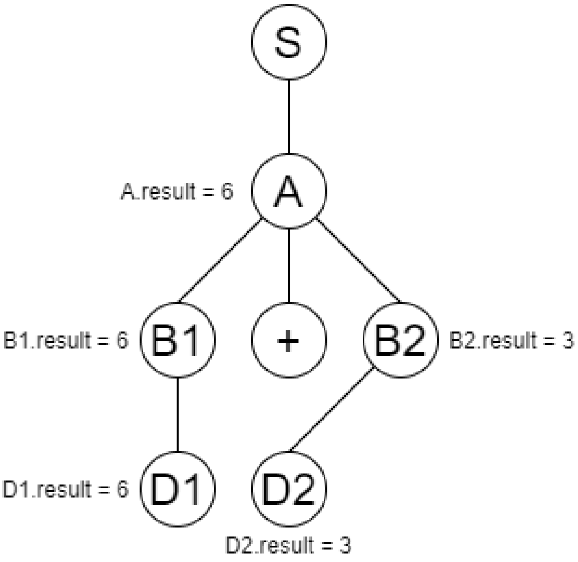
```
   result = D();
   if (scan_test(NULL, RDP_T_16 /* * */, NULL))
   { /* Start of rdp_B_1 */
     while (1)
     {
       {
         scan_test(NULL, RDP_T_16 /* * */, &B_stop);
         scan_();
         val = D();
          result *= val;
         }
       if (!scan_test(NULL, RDP_T_16 /* * */, NULL)) break;
     }
   } /* end of rdp_B_1 */
   scan_test_set(NULL, &B_stop, &B_stop);
  }
 return result;
```



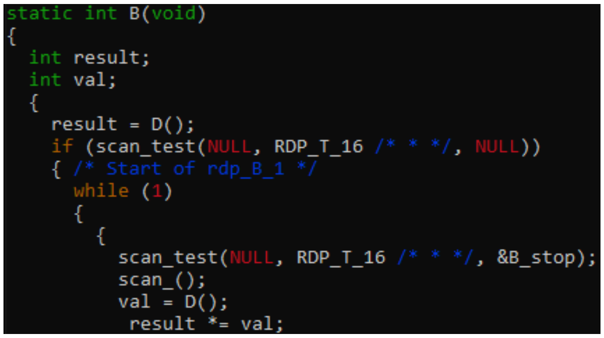
```
result = B();
if (scan_test(NULL, RDP_T_17 /* + */, NULL))
{ /* Start of rdp_A_1 */
  while (1)
  {
    {
      scan_test(NULL, RDP_T_17 /* + */, &A_stop);
      scan_();
      val = B();
```



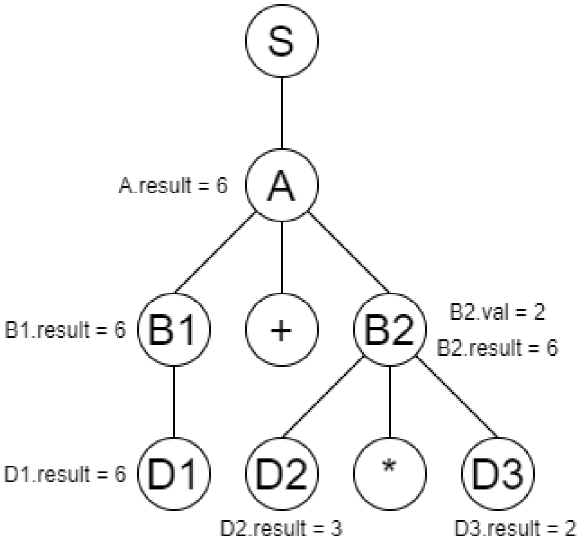
```
static int B(void)
{
  int result;
  int val;
  {
    result = D();
```



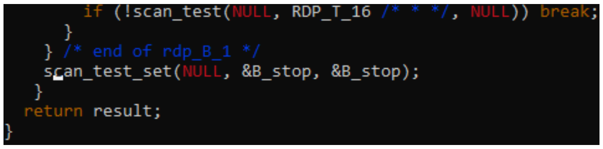
Because the current lookahead token isn’t ‘*’, B immediately returns 6 to A. The current lookahead token is ‘+’ which A matches and enters its loop, resulting in A calling B once more. B calls D which matches ‘3’ and returns the value to B.result.



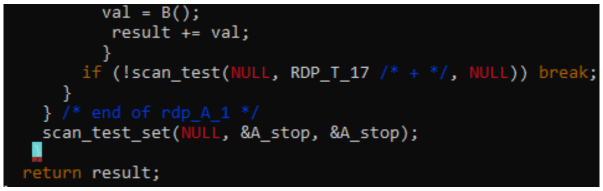
```
static int B(void)
{
  int result;
  int val;
  {
    result = D();
    if (scan_test(NULL, RDP_T_16 /* * */, NULL))
    { /* Start of rdp_B_1 */
      while (1)
      {
        {
          scan_test(NULL, RDP_T_16 /* * */, &B_stop);
          scan_();
          val = D();
           result *= val;
```



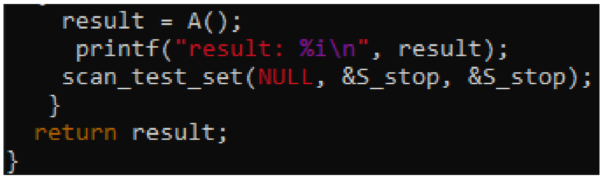
This time the lookahead token is ‘*’ so B enters its loop and calls D. D matches with ‘2’ and returns the result to B.val. B.result is multiplied by B.val making B.result equal to 6.



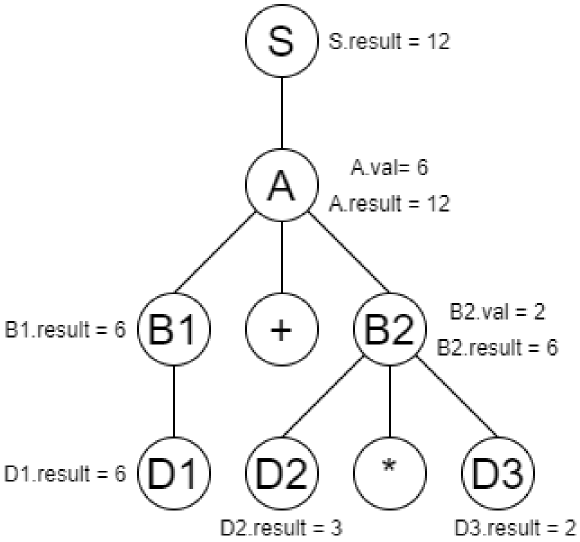
```
        if (!scan_test(NULL, RDP_T_16 /* * */, NULL)) break;
      }
    } /* end of rdp_B_1 */
    scan_test_set(NULL, &B_stop, &B_stop);
  }
  return result;
}
```



```
          val = B();
           result += val;
          }
        if (!scan_test(NULL, RDP_T_17 /* + */, NULL)) break;
      }
    } /* end of rdp_A_1 */
    scan_test_set(NULL, &A_stop, &A_stop);
  }
  return result;
```

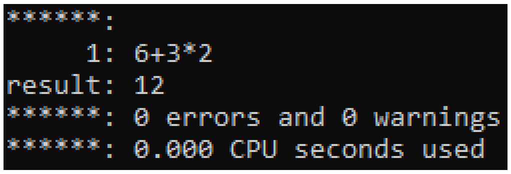


```
    result = A();
     printf("result: %i\n", result);
    scan_test_set(NULL, &S_stop, &S_stop);
  }
  return result;
}
```



B ends its loop due to the lookahead token not being ‘*’. B returns 6 to A.val which is then added to A.result making A.result equal to 12. A ends its loop due to ‘+’ not being the lookahead token. A then returns 12 to S.result which prints the value.

Final word count: 46911



```
******:
     1: 6+3*2
result: 12
******: 0 errors and 0 warnings
******: 0.000 CPU seconds used
```

A couple of observations worth noting here, are that subtraction and addition have equal priority, as do multiplication and division, so these can effectively be implemented in the same rule. Additionally, bracketed expressions are higher priority than multiplication and division, so they can be added as an alternate of D.

```
S:int ::= A:result [* printf("result: %i\n", result);  *].
A:int ::= B:result { '+' B:val [* result += val; *] |
                     '-' B:val [* result -= val; *]
                   }.
B:int ::= D:result { '*' D:val [* result *= val; *] |
                     '/' D:val [* result /= val; *]
                   }.
D:int ::= INTEGER:result | ‘(‘ A:result ‘)’.
```

Final word count: 46911

Further Improvements

A potential improvement to this grammar would be to add some sort of flow control such as conditional statements. It is possible to implement conditional statements using inherited attributes.

```
SYMBOL_TABLE(tbl 101 31
             symbol_compare_string
             symbol_hash_string
             symbol_print_string
             [* char* id; integer i; *]
            )

S    ::=  { DECL | STMT(1) }.

DECL ::= 'int' ID:name
[* if (symbol_lookup_key(tbl, &name, NULL))
                 text_message(TEXT_ERROR, "redeclaration of %s\n", name);
             else
                 symbol_insert_key(tbl, &name, sizeof(char*), sizeof(tbl_data));
          *]
          ['=' EXPR:val [* tbl_cast(symbol_lookup_key(tbl, &name, NULL))->i = val; *]].

STMT(flag:int) ::= ID:name '=' EXPR:val
                   [* if (flag) {
                          if (!symbol_lookup_key(tbl, &name, NULL))
                              text_message(TEXT_ERROR, "%s undeclared\n", name);
                          else
                              tbl_cast(symbol_lookup_key(tbl, &name, NULL))->i = val;
                      }
                   *] |
                   'if' BOOL:cond [* cond = cond && flag; *] 'then' STMT(cond) |
                   'print' EXPR:val
                   [* if (flag) {
                          printf("%i\n", val);
                      }
                   *].

BOOL:int ::= EXPR:result { '==' EXPR:val [* result = result == val; *] |
                           '!=' EXPR:val [* result = result != val; *] } .

EXPR:int ::= PRIMITIVE:result { '+' PRIMITIVE:val [* result += val;  *] |
                                '-' PRIMITIVE:val [* result -= val;  *] }.

PRIMITIVE:int ::= INTEGER:result |
              ID:name
              [* if (!symbol_lookup_key(tbl, &name, NULL))
                     text_message(TEXT_ERROR, "%s undeclared\n", name);
                 else
                     result = tbl_cast(symbol_lookup_key(tbl, &name, NULL))->i;
              *].
```

The above grammar has similar functionality to my current grammar. It includes variable declaration and value assigned, and includes checks to ensure only declared variables are used in statements and no variables are ever redeclared. This grammar includes both boolean and arithmetic expressions however only equality checks and addition and subtraction have been

Final word count: 46911

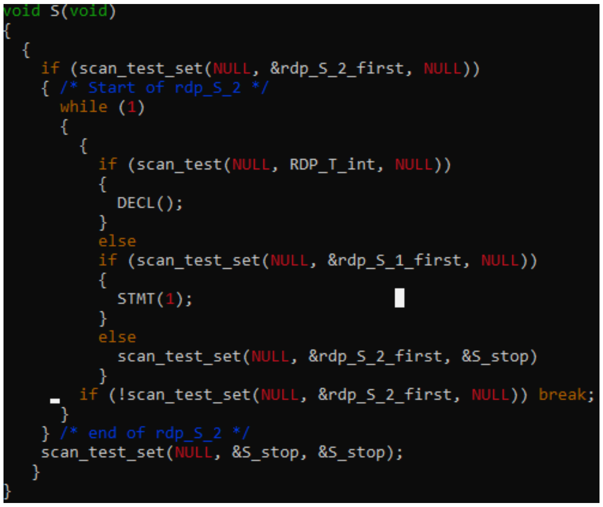
included since I only wrote this grammar to provide a demonstration on how conditional statements can be implemented.

```
STMT(flag:int) ::= ID:name '=' EXPR:val |
                   'if' BOOL:cond 'then' STMT(cond) |
                   'print' EXPR:val.
```

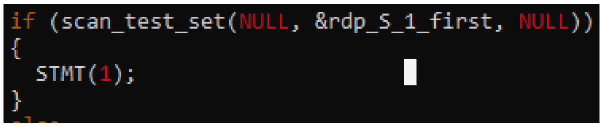
Statements can either be assignment statements, if statements or print statements. Both value assignment and printing can be subject to the condition of an if statement. This means that the semantics of these two statements will only be executed if the condition of the preceding statement is true.

```
S ::=  { DECL | STMT(1) }.
STMT(flag:int) ::= 'if' BOOL:cond 'then' STMT(cond).
```

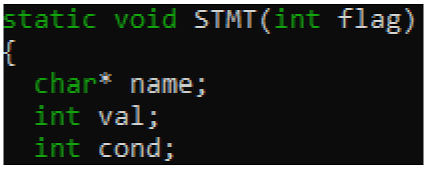
The value 1 is always passed into STMT whenever it is evoked from S.



```
void S(void)
{
  {
    if (scan_test_set(NULL, &rdp_S_2_first, NULL))
    { /* Start of rdp_S_2 */
      while (1)
      {
        {
          if (scan_test(NULL, RDP_T_int, NULL))
          {
            DECL();
          }
          else
          if (scan_test_set(NULL, &rdp_S_1_first, NULL))
          {
            STMT(1);
          }
          else
            scan_test_set(NULL, &rdp_S_2_first, &S_stop)
          }
        if (!scan_test_set(NULL, &rdp_S_2_first, NULL)) break;
      }
    } /* end of rdp_S_2 */
    scan_test_set(NULL, &S_stop, &S_stop);
  }
}
```



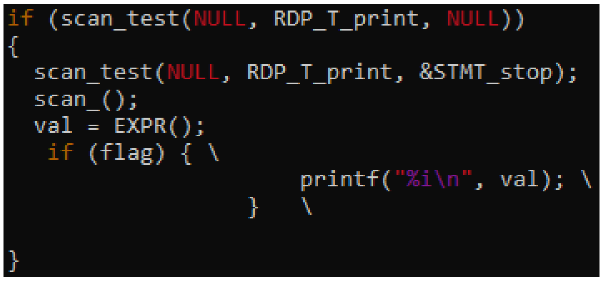
```
if (scan_test_set(NULL, &rdp_S_1_first, NULL))
{
  STMT(1);
}
```



```
static void STMT(int flag)
{
  char* name;
  int val;
  int cond;
```

This is what is meant by inherited attributes, they are values which can be passed into a rule's parse function.

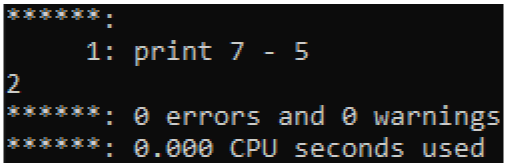
Final word count: 46911



```
if (scan_test(NULL, RDP_T_print, NULL))
{
  scan_test(NULL, RDP_T_print, &STMT_stop);
  scan_();
  val = EXPR();
   if (flag) { \
                   printf("%i\n", val); \
               }   \

}
```

```
STMT(flag:int) ::= 'print' EXPR:val
                 [* if (flag) {
                        printf("%i\n", val);
                    }
                 *].
```

Above are the semantics of the print statement. Before the print function is executed a conditional check is performed using ‘flag’. Since 1 is flag’s default value, this check will typically result in being true, thus the print is executed.



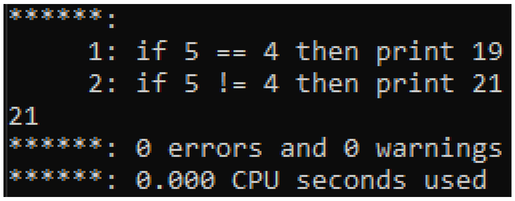
```
******:
     1: print 7 - 5
2
******: 0 errors and 0 warnings
******: 0.000 CPU seconds used
```

```
.str

print 7 -  5
```

The flag can however be modified by the grammar’s if statement.

```
STMT(flag:int) ::= 'if' BOOL:cond [* cond = cond && flag; *] 'then' STMT(cond)
```

‘cond’ is the result of a boolean expression, meaning it will either be 1 or 0 based on whether the boolean expression was true or false. ‘cond’ is then passed into the statement followed by the if statement, meaning a statement following an if statement is subject to the result of the boolean expression.



```
******:
     1: if 5 == 4 then print 19
     2: if 5 != 4 then print 21
21
******: 0 errors and 0 warnings
******: 0.000 CPU seconds used
```

```
.str

if 5 == 4 then print 19
if 5 != 4 then print 21
```

As can be seen from the example above, 19 is not printed because the condition preceding the first print statement is false, however 21 is printed because the condition preceding the second print statement is true.

Final word count: 46911

Intermediate Representation

Although this improved grammar has some useful features, it simply is not suited for the intended task of creating a C compiler capable of compiling a subset of the C language. What a compiler typically does is convert a high level language such as C into some target machine architecture (MIPS, 6502, x86 etc...). In order to achieve this, the approach taken to implement the language's semantics will have to be changed entirely.

Moving forward, rather than having the parser execute the program itself, it will instead emit an intermediate representation (IR) of the C code, which itself can later be translated into the intended machine code.

Before I go into the implementation details, I'd like to touch on what intermediate representation is, and how it is advantageous to take this approach, rather than simply having the compiler emit the machine code itself.

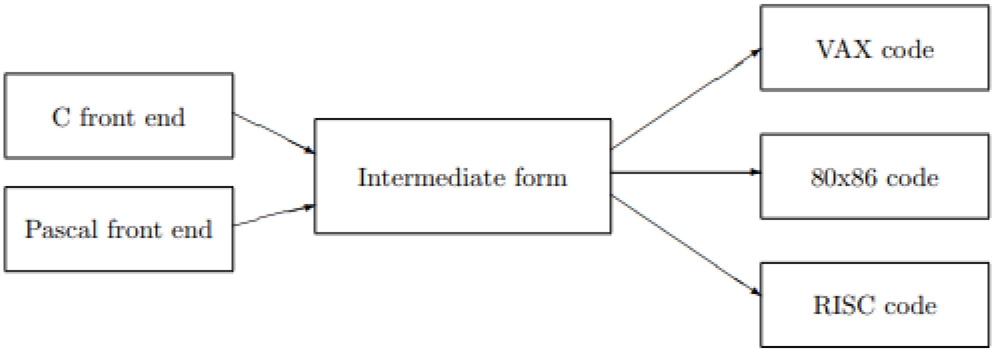


Figure 6. [31]

An intermediate representation acts as a sort of bridge between the high level source code and the low level machine code. In practice, the form an intermediate representation, or specifically intermediate code takes, is closer to that of assembly languages then high level programming languages, however the advantage is that the intermediate representation is general enough that it can be translated to, from many different high level languages and be translated from, to many different low level languages. This methodology promotes the portability of high level languages greatly[32].

Looking at the figure above, let's say for example that c++ was added as a front end to the intermediate form. It then would be capable of targeting all three of the machine architectures which the intermediate form can currently be translated to, significantly reducing the work which would have been required to target all three, had the intermediate representation not already done so.

Additionally, translation to an intermediate form significantly aids code optimization. Code optimization is not one of the focuses of this project, so I will avoid going into too much depth on this topic, however in brief, intermediate forms, like assembly languages, are far more concise than their high level counterpart, whilst retaining the same semantic meaning. This makes the intermediate form a far better target for analysis and rearrangement then the high level form. Intermediate representations are also not concerned with registers and memory allocation, making them better targets for these types of optimizations over their low level counterparts.

Three Address Code

Although there are other forms of intermediate representation, for example Abstract Syntax Trees, I'll primarily be focusing on Three Address Code, as it is what I have used in my implementation.

Three Address Code (often abbreviated to TAC or 3AC) is a type of intermediate code which can easily be converted to machine code, due to it having a close relationship to your typical assembly languages. Each TAC instruction consists of at most three operands, and is typically a combination of assignment and a binary operator[33].

A general representation would be the following:

```
a = b op c
```

Where a, b and c would represent operands such as variable names from the source language, constants (integer literals etc..), or compiler generated temporary variables, which will be demonstrated shortly[34].

Since there is at most one operator on the right side of an instruction, the source languages expression **`x + y - 5`**, would be translated into the following sequence of TAC instructions:[35]

$$\mathbf{t_1 = y - 5}$$
$$\mathbf{t_2 = x + t_1}$$

t_1 and t_2 are examples of the previously mentioned compiler generated temporary variables, x and y are simply named variables copied directly from the source code, and 5 is a constant, also taken directly from the source code.

Besides binary arithmetic, TAC must also be capable of expressing various other programming constructs. Some examples would be:[36]

- Copy instructions of the form x = y. These might be the result of a variable initialization, or any value assignment which does not involve any further operations.
  - `int x = 5;`
  - `y = 3;`

- Unconditional jumps of the form `goto L`, where L is a label indicating some location in the program. These may appear in the body of an `if-else if-else` chain, or at the end of a loop.
- Conditional jumps of the form `if x op y goto L`, where op would refer to some relation operator eg. ==, >=, < etc. These may appear at the start of an if statement or loop.

example of flow control representation in TAC

C code

```
While (x < 5) {
    x = y + z + 1;
}
```

TAC equivalent

```
L1
    if x >= 5 goto L2
    $t_1$ = z + 1
    x = y + $t_1$
    goto L1
L2
```

Note also that similarly to temporary variables, the labels L1 and L2 would be generated by the compiler.

Representations of Three Address Code

The description of TAC instructions specifies the components of each type of instruction, but it does not specify the representation of these instructions in practice[37].

There are three representations of TAC, which are:

1. Quadruples
2. Triples
3. Indirect Triplets

Quadruples consist of four fields; result, op, arg1 and arg2. Op denotes the operator used in the instruction. Arg1 and arg2 represent the operands, arg1 being the left operands and arg2 being the right. Result is used to store the result of an expression. Note also that depending on the instruction, either of or both arg2 and result may or may not be used[38].

Three Address Code

```
L1
    z = y
    $t_1$ = z + 1
    x = y - $t_1$
    goto L1
L2
```

Quadruples

op	arg1	arg2	result
=	y		z
+	z	1	t_1
-	y	t_1	x
goto	L1		

Although the use of quadruples results in more temporary variables, they seemed like the most intuitive solution and efficiency is not my primary concern, so for my implementation I went with quadruples as my method for representing three address code.

Final word count: 46911

Generating Intermediate Code Using RDP

Before I talk about my grammar, I’d like to take a moment to explain how writing a grammar which emits intermediate code can be done using RDP. Below is a basic example grammar which emits TAC for binary expressions and copy instructions.

emit.bnf

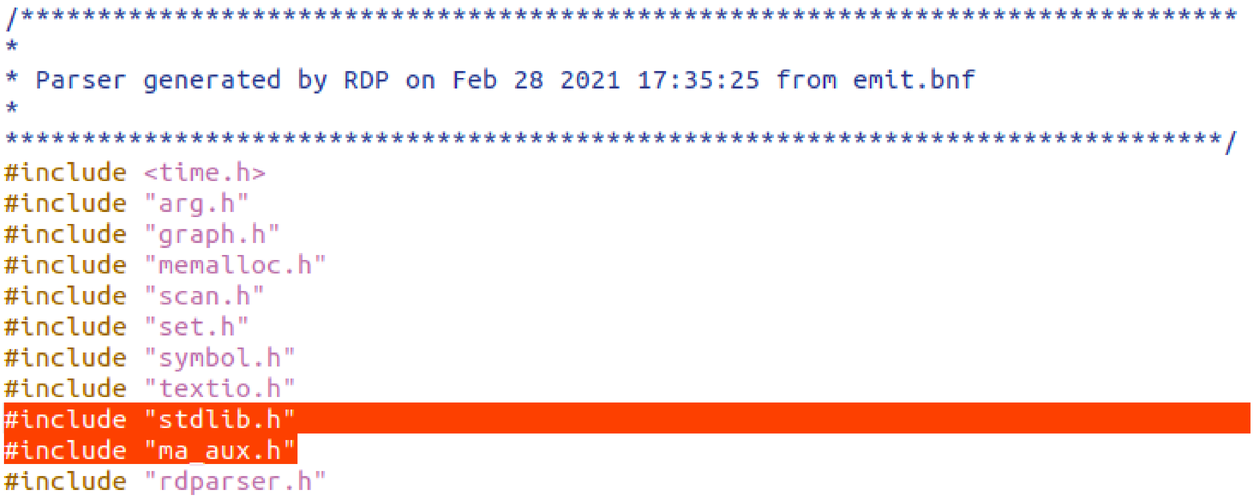
```
USES("stdlib.h")
USES("ma_aux.h")

(* Start Symbol *)
S ::= [* fp = fopen("emit.tac", "w"); *]
         { A }
         [* fclose(fp); *].

(* Assignment *)
A ::= ID:name '=' E:val
      [*  fprintf(fp, "\t%s = %s\n", name, val); *].

(* Addition Expressions *)
E:char* ::= P:val1 [* result = val1; *]
                   {
                       '+' P:val2
                       [*
                           result = new_temp();
                           fprintf(fp, "\t%s = %s + %s\n", result, val1, val2);
                           val1 = result;
                       *]
                   }.

(* Primitives *)
P:char* ::= ID:result
          | INTEGER:val
            [*
                 result = (char*) mem_malloc(12);
                 sprintf(result, "%li", val);
            *].
```

I’ve not bothered to use a symbol table to check whether or not the variables have been declared as the only goal here is to demonstrate how to generate intermediate code.

The first thing to address is the “USES” keyword used right at the start of the grammar.

```
USES("stdlib.h")
USES("ma_aux.h")
```

USES behaves in an RDP .bnf file as “#include” does in a C source file. In fact, each header file specified with this directive is added to the list of includes of the parser generated for this grammar by rdp.



```
/*******************************************************************************
*
* Parser generated by RDP on Feb 28 2021 17:35:25 from emit.bnf
*
*******************************************************************************/
#include <time.h>
#include "arg.h"
#include "graph.h"
#include "memalloc.h"
#include "scan.h"
#include "set.h"
#include "symbol.h"
#include "textio.h"
#include "stdlib.h"
#include "ma_aux.h"
#include "rdparser.h"
```

This means any functions provided by these headers can be used in the semantics of the grammar. For example, in this case the C standard file io functions fopen, fclose, fprintf and sprintf which are all used in the grammar above are provided by “stdlib.h”.

The other include “ma_aux.h” is part of the rdp support library and it provides us with two things. The first, is a file handle “fp”, which provides a file to write the intermediate code to.

```
(* Start Symbol *)
S ::= [* fp = fopen("emit.tac", "w"); *]
         { A }
      [* fclose(fp); *].
```

This file is opened for writing at the start of the parsing process, and subsequently closed at the end.

The second is a pair of functions “new_temp” and “new_label”. Each time new_temp is used, it generates a new temporary variable in the form “_tn_” where n is an integer starting from one and incrementing with each use of the function. This provides the functionality to generate an arbitrary number of temporary variables for use in the TAC. new_label behaves the same, except it produces labels in the form “_Ln_”.

ma_aux.h

```
#include <stdio.h>

FILE * fp;

char * new_temp(void);
char * new_lab(void);
```

Final word count: 46911

ma_aux.c (original)

```
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "textio.h"
#include "memalloc.h"
#include "ma_aux.h"

static long unsigned temp_count = 1;
static long unsigned lab_count = 1;

char * new_temp(void)
{
  char * ret =(char *) mem_malloc(30);

  sprintf(ret, "_t%lu_", temp_count++);

  return ret;
}

char * new_lab(void)
{
  char * ret =(char *) mem_malloc(30);

  sprintf(ret, "_L%lu_", lab_count++);

  return ret;
}
```

ma_aux.c (modified)

```
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "textio.h"
#include "memalloc.h"
#include "ma_aux.h"

static long unsigned temp_count = 1;
static long unsigned lab_count = 1;

char * new_temp(void)
{
  char * ret =(char *) mem_malloc(30);

  sprintf(ret, "$t%lu", temp_count++);

  return ret;
}

char * new_lab(void)
{
  char * ret =(char *) mem_malloc(30);

  sprintf(ret, "#L%lu", lab_count++);

  return ret;
}
```

For the current version of my compiler, I chose to make some small modifications to ma_aux.c. Instead of temporary variables and label being generated in the form “_tn_” and “_Ln_”, they are instead of the form “$tn” and “#Ln”. I’ve two reasons why I made these modifications.

The first is due to the fact that _tn_ and_Ln_ are both valid identifiers in C, so in order to avoid any of the potential problems associated with this issue later on, I simply changed the formats to no longer be valid C identifiers.

The second revolves around the fact _tn_and _Ln_ both begin with underscores. This creates an element of ambiguity when reading labels and temporary variables, so instead I replaced the preceding underscores with ‘$’ and ‘#’ for temporary variables and labels respectively, effectively removing the ambiguity. I also removed the trailing underscore, since it seemed unnecessary.

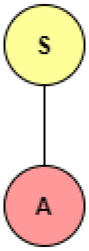
Final word count: 46911

Moving back to the point, what this grammar does is takes binary addition expressions, and writes the appropriate Three Address Code for them into a file. The file containing the TAC, can then be read by a separate program in order to generate the appropriate machine code.

Recall that, at least in the case of binary expressions, TAC consists of a combination of variable names, constants and temporary variables, alongside the operators in order to form the expressions. Let's look at a few examples of how these expressions are generated.

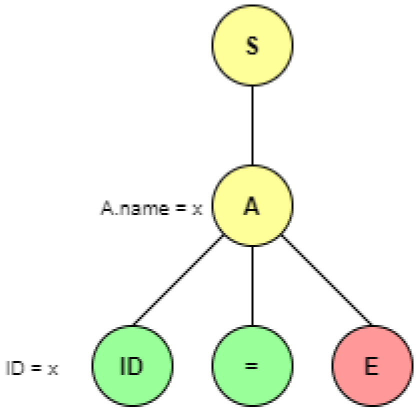
Let's first look at an example of how a copy instruction would be generated with the following string:

x = 5



```
S ::= [* fp = fopen("emit.tac", "w"); *]
         { A }
      [* fclose(fp); *].
```

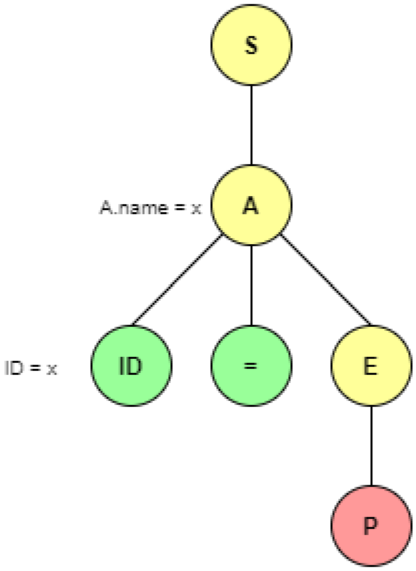
From the start node, we begin by opening the file which the TAC will be written to, and continue to attempt to match the non-terminal **A**. The node **S** has been marked as yellow here, as it has only partially completed its function, having not closed the file yet. The node **A** has been marked red, as any actions associated with this node have yet to take place.



```
A ::= ID:name '=' E:val
[* fprintf(fp, "\t%s = %s\n", name, val); *].
```

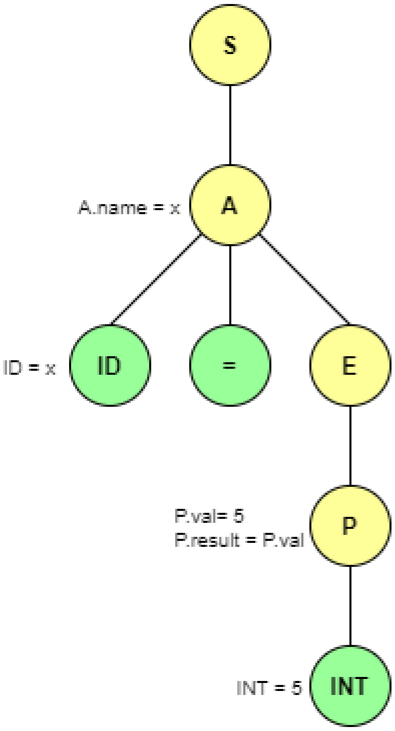
'x' and '=' are matched, and 'name' is assigned the value "x". The nodes for **ID** and = are, since they will not be revisited. The non-terminal **E** is then attempted to be matched.

Final word count: 46911



```
E:char* ::=
P:val1 [* result = val1; *]
{
  '+' P:val2
  [*
    result = new_temp();
    fprintf(fp, "\t%s = %s + %s\n", result, val1, val2);
    val1 = result;
  *]
}.
```

Upon entering the node **E,** the non-terminal **P** is attempted to be matched.



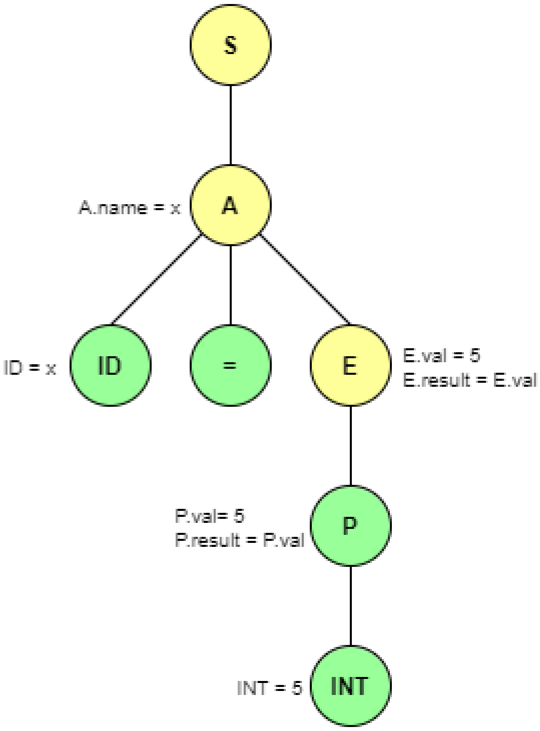
```
P:char* ::=
  ID:result
| INTEGER:val
  [*
    result = (char*) mem_malloc(12);
    sprintf(result, "%li", val);
  *].
```

Since ‘5’ is an integer, it is matched with the second alternate with **P** and assigned to val. Since in this case val is an **int** and result is a **char*,** val must be written to result as a string. This is easily achieved using sprintf, however before this can be done, an amount of memory sufficient for storing a 32 bit integer as a string must be allocated to result.

Final word count: 46911

+/-	**2**	**1**	**4**	**7**	**4**	**8**	**3**	**6**	**4**	**7**	**\0**

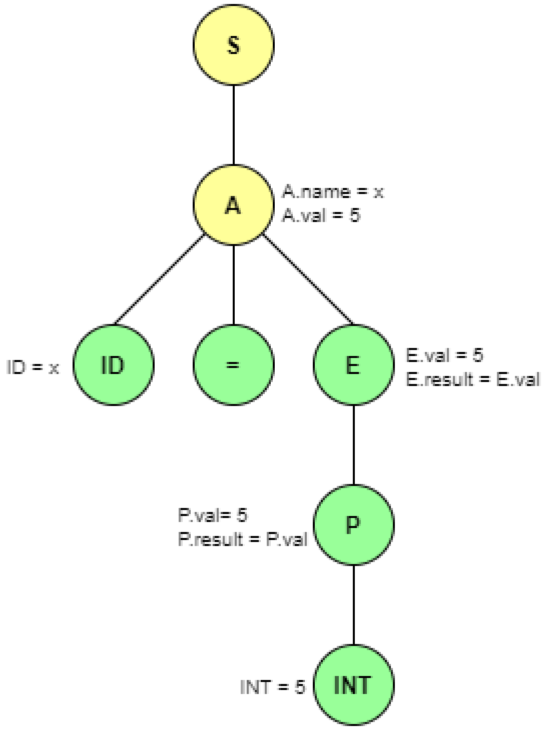
Above is a diagram displaying why 12 bytes is both necessary and sufficient in this case, since a byte is needed to store the sign, the largest (and smallest) numbers will contain no more than 10 digits, and a byte is needed for the NULL terminator.



```
E:char* ::=
P:val1 [* result = val1; *]
{
  '+' P:val2
  [*
    result = new_temp();
    fprintf(fp, "\t%s = %s + %s\n", result, val1,
val2);
    val1 = result;
  *]
}.
```

The value ‘5’ has now propagated all the way up to **E** which takes the ‘5’ as its result. Since there is no ‘+’ symbol to match, no further actions are taken here.

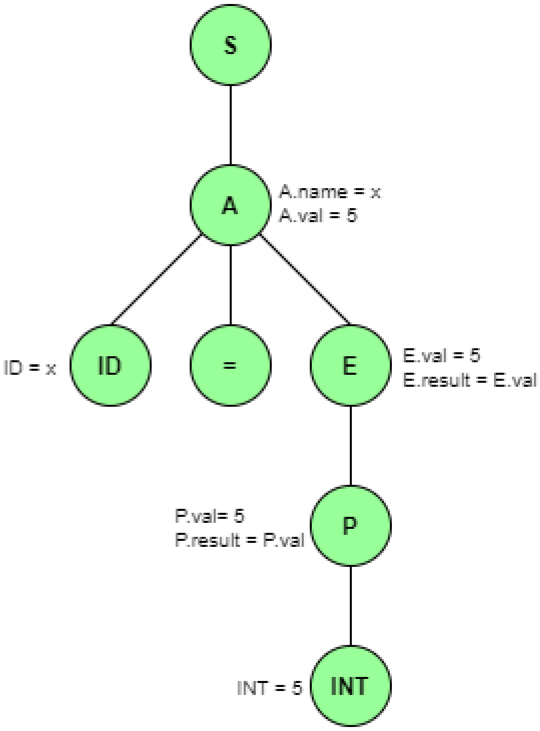
```
Final word count: 46911
```



```
A ::= ID:name '=' E:val
[* fprintf(fp, "\t%s = %s\n", name, val); *].
```

Now after returning to **A**, all the necessary information for the copy instruction has been gathered and is thus written to the file.

```
Final word count: 46911
```



```
S ::= [* fp = fopen("emit.tac", "w"); *]
        { A }
        [* fclose(fp); *].
```

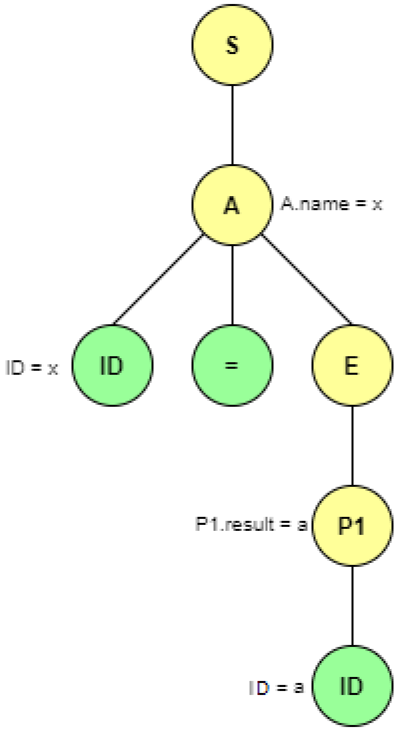
Upon returning to **S**, there are no more expressions to be read, thus the process is not repeated, the file is subsequently closed and the parse is completed successfully. The file “emit.tac” now has the following contents:

```
x = 5
```

Interestingly enough the output in this case matches the input. This is simply a quirk of how this particular grammar generates copy instructions, my current grammar behaves slightly differently in this case which is something I will demonstrate and explain why further down the line. For now the basic process of how the grammar generates TAC has been demonstrated, so now I’ll move on to how the TAC for an addition expression would be generated.

Let’s take the following expression:

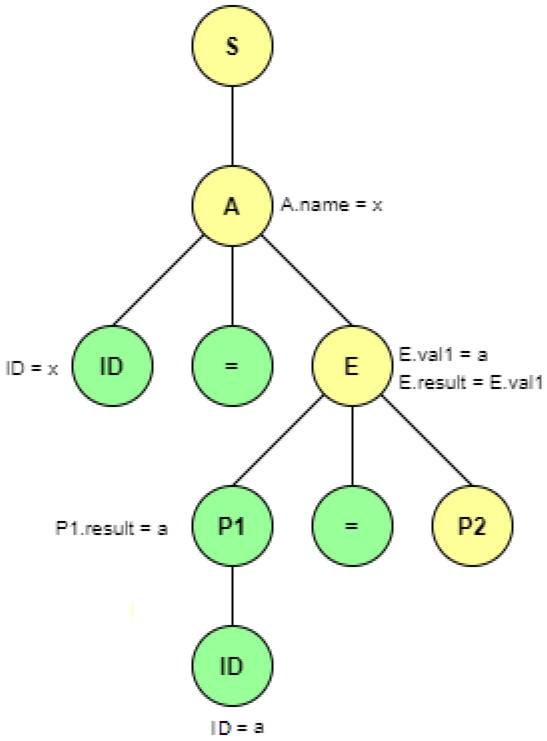
x = a + b + 4



```
P:char* ::=
  ID:result
| INTEGER:val
  [*
    result = (char*) mem_malloc(12);
    sprintf(result, "%li", val);
  *].
```

Since the process of matching the ‘x = a’ is largely the same as before, we’ll skip straight to the node **P1.** Here instead of matching an integer, we match an identifier, thus the first alternate is used and ‘a’ is returned as the result of **P1.**

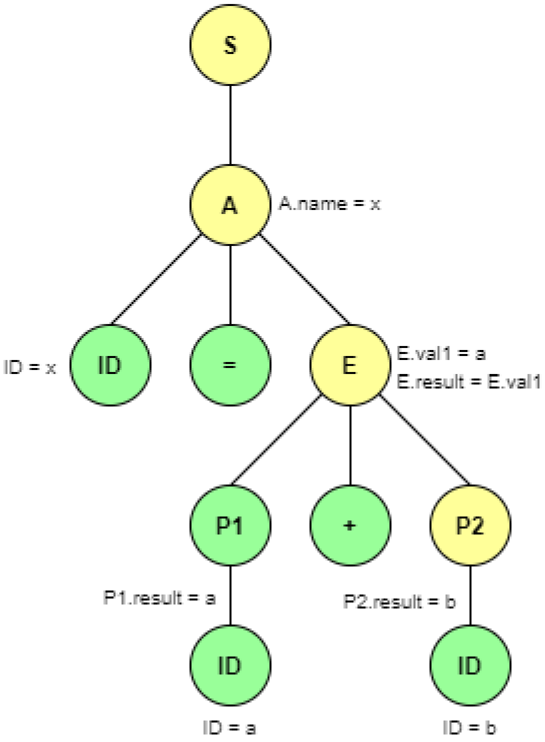
Final word count: 46911



```
E:char* ::=
P:val1 [* result = val1; *]
{
  '+' P:val2
  [*
    result = new_temp();
    fprintf(fp, "\t%s = %s + %s\n", result, val1,
val2);
    val1 = result;
  *]
}.
```

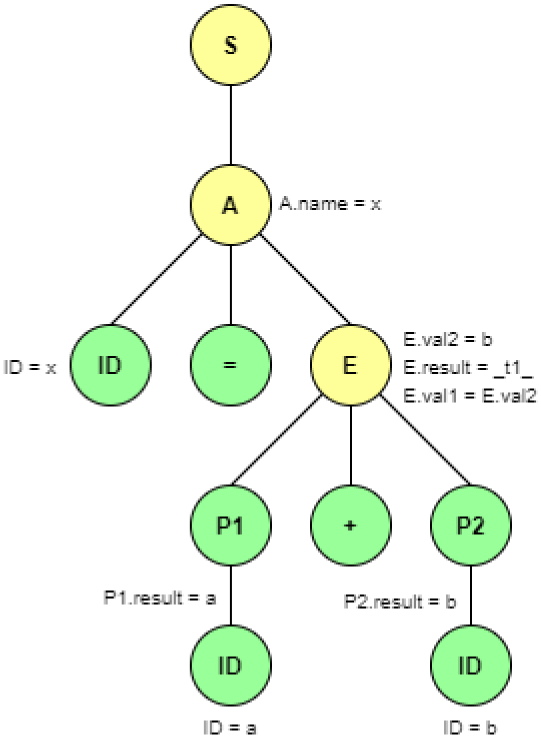
Upon returning to **E**, ‘a’ is assigned to the result of **E**. the first ‘+’ symbol is matched. A match for **P2** is then attempted.

Final word count: 46911



```
P:char* ::=
  ID:result
| INTEGER:val
  [*
    result = (char*) mem_malloc(12);
    sprintf(result, "%li", val);
  *].
```

Similar to before, the symbol ‘b’ is matched and set as the result of **P2.**



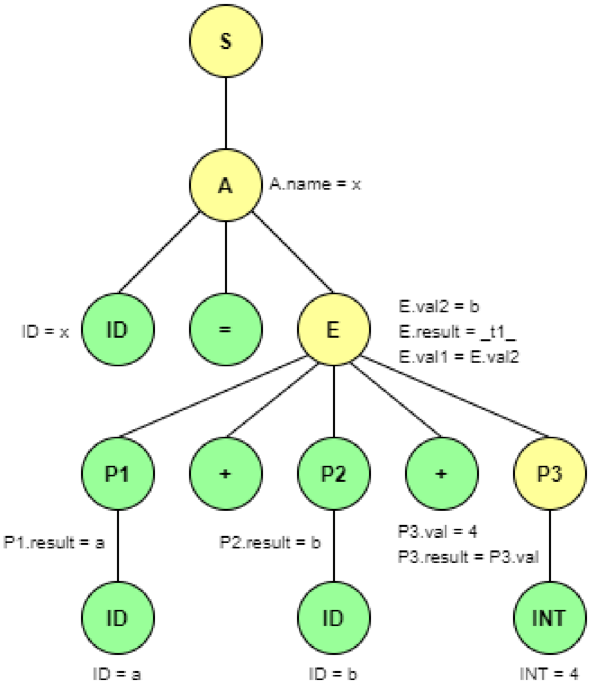
```
E:char* ::=
P:val1 [* result = val1; *]
{
  '+' P:val2
  [*
    result = new_temp();
    fprintf(fp, "\t%s = %s + %s\n", result, val1,
val2);
    val1 = result;
  *]
}.
```

Final word count: 46911

Now something slightly more interesting happens. Recall that in a TAC expression, there can be at most one operator on the right side of an expression. Currently, we're only aware of a single operator, however we don't know ahead of time how many operators there might be (at least not by this grammar's implementation). Additionally, even if this were not the case, the result of this expression will at some point be assigned to 'x' via a copy instruction. For this to be possible, the result must be held in a single location, hence we generate a temporary variable to act as that storage location. All the necessary information necessary to generate this first portion of the expression has been gathered, thus, the following is written to the file:

```
_t1_ = a + b
```

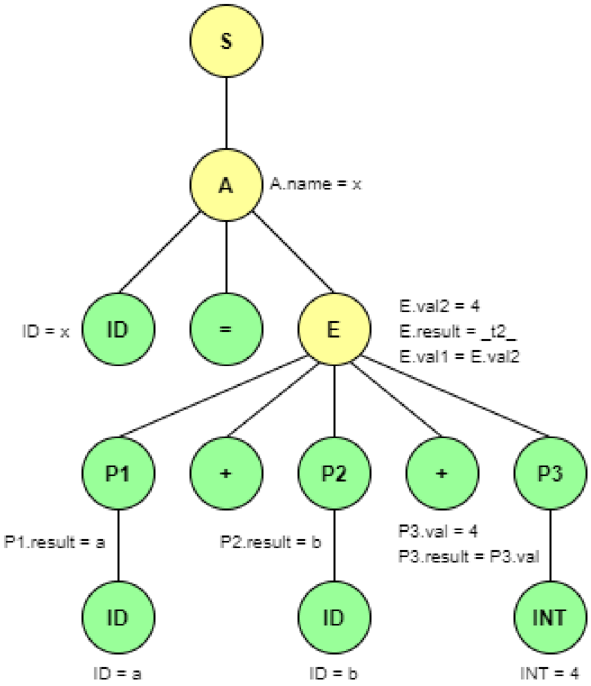
To account for the case of an additional operator being matched, 'result' which holds the temporary variable representing the sum of 'a' and 'b' is stored in val1, so it can be used as the left operand of the next portion of the expression, if necessary.



```
E:char* ::=
P:val1 [* result = val1; *]
{
  '+' P:val2
  [*
    result = new_temp();
    fprintf(fp, "\t%s = %s + %s\n", result, val1,
val2);
    val1 = result;
  *]
}.

P:char* ::=
  ID:result
| INTEGER:val
  [*
    result = (char*) mem_malloc(12);
    sprintf(result, "%li", val);
  *].
```

The second '+' symbol is matched, resulting in a third **P** node. This time the symbol matched by **P** is an integer '4'. This is set as the result of **P3** which is returned to **E**.

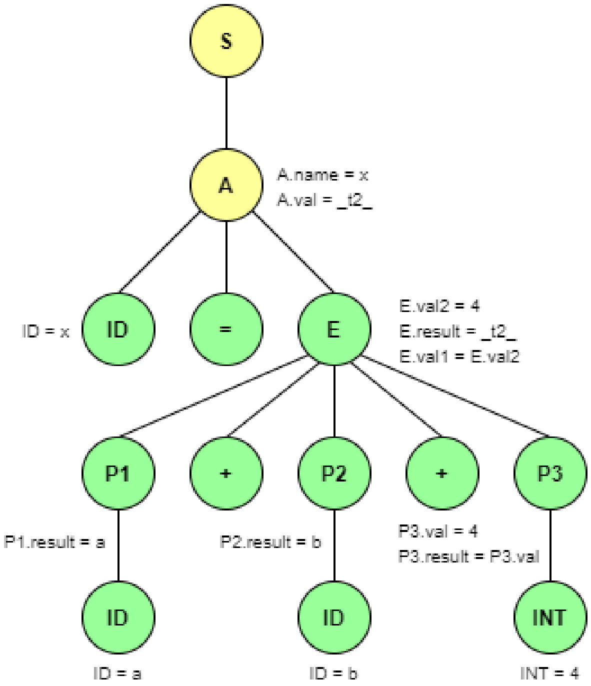
Final word count: 46911



```
E:char* ::=
P:val1 [* result = val1; *]
{
  '+' P:val2
  [*
    result = new_temp();
    fprintf(fp, "\t%s = %s + %s\n", result, val1,
val2);
    val1 = result;
  *]
}.
```

Once again a temporary variable is generated, and the following is written to the file:

```
_t2_ = _t1_ + 4
```

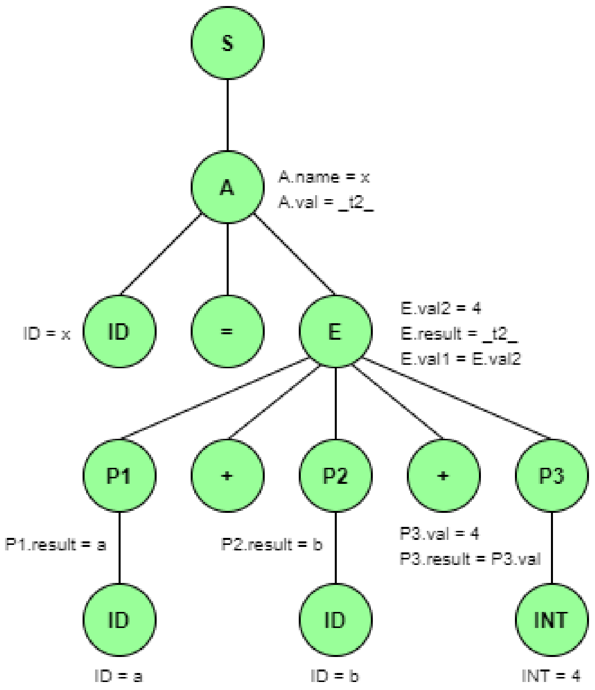


```
A ::= ID:name '=' E:val
[* fprintf(fp, "\t%s = %s\n", name, val); *].
```

Final word count: 46911

No more ‘+’ symbols are matched, thus the result of **E** which is now ‘_t2_’, is returned to **A** where the following copy instruction is written to the file.

```
x = _t2_
```



```
S ::= [* fp = fopen("emit.tac", "w"); *]
         { A }
      [* fclose(fp); *].
```

There are no further symbols to be matched, thus the file is closed and the parse ends successfully producing the following output in “emit.tac”:

```
_t1_ = a + b
_t2_ = _t1_ + 4
x = _t2_
```

Through these demonstrations, we can begin to see how the generation of intermediate code using RDP can be performed. Information is propagated through the parser until all information necessary to produce the next instruction is gathered, which is then subsequently written to file using the supporting functions. As my grammar has grown in complexity and size, although there are several more moving parts, most if not all rests on this principle.

It is also worth noting that in this example, no values have been assigned to any of the variables aside from ‘x’, and even in the case of ‘x’, there is no method of retrieving the value which has been assigned to it from the generated intermediate code alone. For this we’d once again have to employ the use of the symbol table.

Final word count: 46911

My Initial IR Generating Grammar

```
SYMBOL_TABLE(global 101 31
       symbol_compare_string
       symbol_hash_string
       symbol_print_string
       [* char* id; *]
       )

USES("stdlib.h")
USES("ma_aux.h")
USES("tbl_aux.h")

S ::= [* fp = fopen("eac.tac", "w");
         ft = fopen("eac.tbl", "w"); *]
         { D ';' }
         [* fclose(fp);
            fclose(ft); *].

D ::= 'int' ID:name (* declaration *)
         [* if (symbol_lookup_key(global, &name, NULL)) {
                    text_message(TEXT_ERROR, "redeclaration of '%s'\n", name);
              } else {
                    symbol_insert_key(global, &name, sizeof(char*),
sizeof(global_data));
              } *]
         [ '=' E:val
              [*  fprintf(fp, "\t%s = %s\n", name, val);
              *]
         ]
         [* fprintf(ft, "%s\n", name); // print new symbol table entry to file *]
       | ID:name (* assignment *)
         [* if (!symbol_lookup_key(global, &name, NULL)) {
                  text_message(TEXT_ERROR, "'%s' undeclared\n", name);
              }
         *]
         [ '=' E:val
              [* fprintf(fp, "\t%s = %s\n", name, val);
              *]
         ]
       | C.
```

Final word count: 46911

```
(* provisional if statement implementation *)

C ::= 'if' '(' E:val ')'
         [* char* end = new_lab();
              char* lab = new_lab();
              fprintf(fp, "\tif %s == 0 goto %s\n", val, lab);
         *]
         E
         [* fprintf(fp, "\tgoto %s\n", end);
              fprintf(fp, "%s\n", lab);
         *]
         { 'else' 'if' '(' E:val ')'
             [* lab = new_lab();
                fprintf(fp, "\tif %s == 0 goto %s\n", val, lab);
             *]
             E
             [* fprintf(fp, "\tgoto %s\n", end);
                fprintf(fp, "%s\n", lab);
          *]
         }
         [ 'otherwise' E ]
         [* fprintf(fp, "%s\n", end); *].

E:char* ::= P:val1
                [* result = new_temp();
                      fprintf(fp, "\t%s = %s\n", result, val1);
                      val1 = result;
                *]
                { '+' P:val2 (* addition *)
                  [* result = new_temp();
                   fprintf(fp, "\t%s = %s + %s\n", result, val1, val2);
                   val1 = result;
                  *]
                }.

P:char* ::= INTEGER:val (* integer literals *)
                 [* result = (char*) mem_malloc(12);
                    sprintf(result, "%li", val); *]
               | ID:name (* variables *)
                 [* if (!symbol_lookup_key(global, &name, NULL)) {
                            text_message(TEXT_ERROR, "'%s' undeclared\n", name);
                    } else {
                           result = name;
                    }
                 *].
```

Final word count: 46911

Above is the initial grammar I wrote which emits three address code. Its functionality includes the following:

- Declaration of integer variables.
- Value assignment to integer variables via arithmetic expressions.
    - Arithmetic expressions currently handle addition.
- Detects variable redeclaration and attempted use of undeclared variables.
- Contains provisionary logic for "if / else if / else" statements.

Functionally this grammar is very similar to the example grammar described in the previous section, so i'll mainly be focusing on the parts where they deviate. The first thing to notice is the use of a symbol table, and the header file “tbl_aux.h”. The header has a simple purpose, which is to provide access to the file“ft”.

```
ft = fopen("eac.tbl", "w");
```

Simply put, this file contains certain information held by the symbol table during the parsing process. Specifically in this case, the file contains the identifier names. The reason for this is the initial code generator I wrote requires these variable names in order to produce appropriate code. I will go into more details on this when I talk about the code generator, however here is an example of the contents this file might hold.

eac.str	eac.tbl
`int x;`	`x`
`int y = 3`	`y`
`int z = y;`	`z`

So it simply contains the names of the variables, and ignores any other information associated with them. Note that this is not the case for the final grammar, as will be explained further down the line.

The symbol table is used to detect redeclaration of existing variables, and the use of undeclared variables, the approach to which I have previously covered in the section “Symbol Tables in rdp”.

Non terminals S, E and P behave the same as their identically named counterparts from the grammar detailed in the previous section, aside from some additional logic which checks for variable declarations.

The non terminal ‘D’ contains particular deviations from the example grammar, however largely serves the same purpose as ‘A’ from the example.

```
D ::= 'int' ID:name (* declaration *)
         [* if (symbol_lookup_key(global, &name, NULL)) {
                      text_message(TEXT_ERROR, "redeclaration of '%s'\n", name);
               } else {
                      symbol_insert_key(global, &name, sizeof(char*), sizeof(global_data));
               } *]
         [ '=' E:val
              [*  fprintf(fp, "\t%s = %s\n", name, val);
              *]
         ]
         [* fprintf(ft, "%s\n", name); // print new symbol table entry to file *]
       | ID:name (* assignment *)
         [* if (!symbol_lookup_key(global, &name, NULL)) {
                    text_message(TEXT_ERROR, "'%s' undeclared\n", name);
               }
         *]
         [ '=' E:val
              [* fprintf(fp, "\t%s = %s\n", name, val);
              *]
         ]
       | C.
```

D handles variable declarations. Semantics aside, this is identical to how declaration and assignments were parsed in previous, non IR emitting versions of the grammar, so for details on that refer to the section “My Improved Grammar”. D emits three address code in the same manner as the A from the previous example, however the key difference here is that the variables are actually being declared here, this results in two key differences.

The first is that any declared variables have their variable names written to the .tbl file as described before.

```
[* fprintf(ft, "%s\n", name); // print new symbol table entry to file *]
```

The second is that D does not require value assignment, as A did. Because of this fact, declaring variables does not always generate TAC, as there is no reason to do so if no value is assigned.

```
int x;
```

The above example would result in the variable name ‘x’ being written eac.tbl, however no TAC is generated. This is because in this version of the grammar, all variables are global and they are all statically allocated meaning there is no runtime consequence of variable declaration. However, all variables are zero initialized and have their values assigned at runtime, therefore if a value assignment is performed, some TAC is produced to represent that.

This isn’t a great approach due to its inefficiency. If global variables are statically allocated they can have their values calculated during the parsing process, and assigned values when allocated rather than during runtime. This places a slight burden on the compiler however it results in a more efficient binary, which I believe is a fair trade off. Later versions of the compiler do exactly this.

The last thing I want to highlight in this grammar is the non terminal C which stands for “conditional” and handles if, else if, else statements. Because of the limitations of LL(1) grammars, in order to avoid ‘else if’ and ‘else’ having a first first conflict, in this grammar I replaced ‘else’ with ‘otherwise’. You can actually force rdp to ignore these errors and in this particular case it will still work due to rdp using the longest match strategy, however I did want to use the forced generation feature during development, in case some errors went unnoticed. In the final version this has been changed since the grammar is meant to emulate C.

Whilst on the subject of emulating C, I’d like to draw your attention to the implementation of the semantics.

```
C ::= 'if' '(' E:val ')'
         [* char* end = new_lab();
               char* lab = new_lab();
               fprintf(fp, "\tif %s == 0 goto %s\n", val, lab);
         *]
```

In C, boolean statements are evaluated as ‘True’ or ‘False’ but rather as zero or non zero. If the expression used as the condition results in a non zero value then the statement is true and if only false if the result is exactly zero (so negative results are also true). Because of this, any arithmetic expression can be used as a condition, rather than only boolean expressions, hence ‘E’ being the conditional variable here.

The three address code this produces also matches this convention:

```
fprintf(fp, "\tif %s == 0 goto %s\n", val, lab);
```

The condition always checks if the result is equal to zero and if this is the case, the program will execute a goto statement which skips the logic which would have been executed if the result were non zero. By implementing conditional statements in this way, this accurately emulates C like behaviour.

Final word count: 46911

Below is an example program containing an if statement, and the TAC it would generate.

```
int x;
if (1) {
    x = 5;
}
```

```
    _t1_ = 1
    if _t1_ == 0 goto L2
    _t2_ = 5
    x = _t2_
    goto _L1_
_L2_
_L1_
```

We can see from the TAC on the right that the conditional jump is first evaluated and then directly beneath it is the assignment of the value 5 to the variable ‘x’. Placed directly after the assignment logic is a label which the program will jump too if the condition is found not to be true. In this case the condition is always true, however in the event that it is not, the assignment logic will simply be skipped, as one might expect with any if statement.

You’ll have also noticed the label ‘L1’ has been placed directly after ‘L2’. The reason for this is that a label is always placed implicitly to account for the possibility of an ‘else if’ and ‘otherwise’ statements. Take the following example:

```
if (0) 21
else if (1) 23
else if (2) 25
else if (3) 27
otherwise 29;
```

```
       _t1_ = 0
       if _t1_ == 0 goto _L2_
       _t2_ = 21
       goto _L1_
_L2_
       _t3_ = 1
       if _t3_ == 0 goto _L3_
       _t4_ = 23
       goto _L1_
_L3_
       _t5_ = 2
       if _t5_ == 0 goto _L4_
       _t6_ = 25
       goto _L1_
_L4_
       _t7_ = 3
       if _t7_ == 0 goto _L5_
       _t8_ = 27
       goto _L1_
_L5_
       _t9_ = 29
_L1_
```

As can be seen from the right, at the end of the if statement and each of the else if statements, the program jumps unconditionally to the bottom of the code block. If these jumps were not there, the program would incorrectly execute additional conditional checks, and would always always execute the instructions following the ‘otherwise’. Using a flag, I could detect whether or not it is necessary to include the jump to L1, however the consequence of it being there is quite small, and I prefer to avoid using flags where possible, as I find them to be quite a messy approach to solving problems. Since the redundant label is quite a small issue, i’ve yet to come up with a more elegant solution to solve this issue, so this is something which still exists in the current version of the grammar.

Clearly this grammar is rather simplistic and actually includes less functionality than my earlier grammars which did not produce TAC, however this was something I created as a starting point so I was able to begin working on a code generator, since I wanted to get a grasp of how to tackle all the stages of compilation before adding more complicated features to the language.

Code Generation

Code generation is the process of producing the code for a target architecture. In our case, this is done by taking the intermediate code, and converting it into assembly code. Additional to the IR, certain information from the symbol table is also necessary to generate the appropriate code.

A code generator has three primary tasks: instruction selection, register allocation and instruction ordering. Naturally the manner in which these tasks must be performed changes across different architectures, due to differing instruction sets and numbers of registers[39].

Instruction selection simply refers to the process of identifying a set assembly code instructions which correctly execute the semantics of the intermediate code. With an appropriately high level IR, it is possible to generate machine level code by simply matching each line of intermediate code with a template. Take the following example:[40]

Three Address Code	Mips Assembly

`x = 5`	`la $t0, x`
	`li $t1, 5`
	`sw $t1, ($t0)`

To the left we have a single line of TAC indicating that the value 5 is to be assigned to the variable ‘x’. To the right we have three lines of mips assembly which match the semantics of the TAC to the left. First the address of the variable ‘x’ is loaded into a register, then the value 5 is loaded into a different register, and finally the value 5 is stored at the memory location represented by ‘x’. These three lines of mips can be used as the template for assigning values to variables, where only the value and the variable name need change each time.

This isn’t a particularly efficient approach to code generation. If the TAC better reflected the details of the machine’s architecture, this could be used to generate more efficient code[41].

Register Allocation is the process of deciding which values are to be held in which registers. In order for a CPU to process data it must first load said data from the main memory into a register. It is entirely possible to simply load values from main memory every time a variable is used, however this is very inefficient due to the latency incurred by the data transmission from main memory to the CPU.[42]

It is instead possible to hold values which are often reused in registers for extended periods of time, thus removing the need to retrieve data from main memory. Let's take the following example:

```
while (x) {
    x = x - 1;
}
```

```
L1:
    la $a1, x          # load address of x
    lw $t1, ($a1)      # load value of x
    beqz $t1, L2       # end loop if x is 0
    la $a2, x          # load address of x
    lw $t2, ($a2)      # load value of x
    li $t3, 1          # load value 1
    sub $t4, $t2, $t3  # subtract 1 from x
    la $a0, x          # load address of x
    sw $t4, ($a0)      # store new value of x
    j L1               # loop once more
L2:
```

To the left is a basic while loop in C, and to the right is the equivalent in mips assembly, however each time the variable 'x' is accessed, it is loaded from main memory. Without a register allocation strategy which attempts to hold on to variables currently in use, this is how one might expect such a loop to be generated as assembly code. Now let's try a different approach.

```
    la $a0, x
    lw $t0, ($a0)
L1:
    beqz $t0, L2
    li $t1, 1
    sub $t0, $t0, $t1
    j L1
L2:
    sw $t0, ($a0)
```

Here we have a much shorter program which achieves the exact same outcome. Rather than retrieving 'x' from main memory and storing its updated value in main memory multiple times during every iteration of the loop, it is instead loaded once before entering the loop, and updated once afterwards exiting the loop. This demonstrates how using the simple concept of keeping variables assigned to registers can avoid many potentially unnecessary memory accesses, and significantly improve performance.

The problem of register allocation however in practice is rather complex as in a typical case there are several more variables in a program than available registers. For this reason, it is never as simple as assigning certain variables to certain registers, but rather identifying periods where different groups of variables need to be accessed and identifying a set of registers to accommodate them. There are several algorithms which achieve this, the predominant approach being Graph Colouring Allocation by Chaitain et al, however I did not use any such algorithms in my implementation, and would also be quite lengthy to explain, as well as falling outside of the scope of this project[43].

For my personal approach, I have chosen to implement my own, fairly simple register allocation strategy which takes advantage of the large number of general purpose registers of my chosen architecture. I have also chosen to adopt a template based code generation strategy. My reasoning for these choices is that the focus of this project is not optimisation, and for my purposes a basic strategy which works with my chosen architecture is perfectly acceptable.

Choosing An Assembly Language

When choosing a specific architecture to target there were three choices which I had in mind from the start. The were the following:

- x86
- 6502 Assembly
- Mips

Out of these three, I’ve only had experience with 6502 and mips, however I wanted to consider x86 as a potential candidate, since it is the architecture my machine runs on, and thus I may have been able to test the program my compiler produced without the use of an emulator. My lack of experience with the language however, paired with the fact that x86 is a CISC architecture made it a poor choice.

6502 I have had brief experience working with, more so in the context of attempting to emulate it rather than writing programs in it, but I had some brief experience writing programs in it also. The main problem with 6502 is that it is an old architecture and this fact alone introduces complications.

6502 is an 8-bit architecture. Among other things this means all registers other than the program counter (including the accumulator) are 8-bit, meaning any value larger than 255 would have to be stored across multiple registers. There are only three general purpose registers, these being registers X, Y and the accumulator, and in order to access each of them specific instruction have to be used, for example LDA loads a value to the accumulator and LDX and LDY do the same for the X and Y registers. There are also no instructions for multiplication or division, so this functionality would also have to be implemented manually, and when the add instruction results in a carry, this has to be handled manually.[44][45]

Essentially, if I were to target 6502 I’d have ran into several complications due to the dated nature of the architecture. It would have complicated the development of the code generator unnecessarily, likely making it take significantly longer to build.

Mips I worked with briefly during the first year of my degree. I had also experimented with the language slightly during this period to test some of the features which were not covered in the course content, so I was aware of some of its capabilities such as being able to perform multiplication, and having instructions geared towards the implementation of functions.

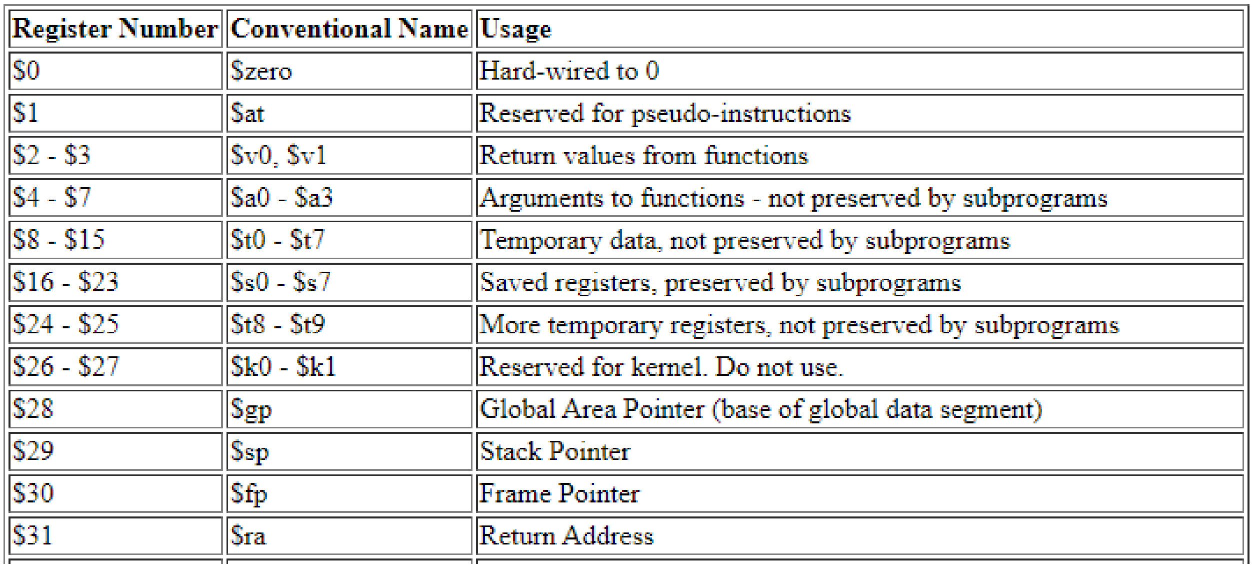
Mips is a 32 / 64-bit architecture with an abundance of general purpose registers, so would be unlikely that I'd run into issues with overflow like if I were working with 6502, and the large number of registers meant I'd have a much easier time implementing my register allocation strategy. Mips is also a RISC architecture, so it would likely be easier to work with than x86. I was also aware of the fact that although I did not have hardware which could run my compiled programs (the only system I personally know of which ran on mips was the Nintendo 64), it is in fact easily emulated with a program called SPIM, which I also had experience using.

After being recommended by my supervisor, Mips was a clear choice as it would likely be the architecture I would have to spend the least time familiarising myself, as well as likely being the architecture I'd run into the least problems with when building the back end.

Mips

Since Mips will be used a lot in the coming sections I'd like to give a short outline of the features I've utilized. Mips has 32 and 64-bit implementations however for my purposes I will be focusing on the 32-bit implementation[46].

Mips has 32 32-bit general purpose registers. It also has 32 32-bit registers for use with floating point operations, however I have not implemented floating point types so I will go into no further details on those. It also has a 64 bit accumulator, however this can not be accessed directly as with the general purpose registers[47].



Register Number	Conventional Name	Usage
$0	$zero	Hard-wired to 0
$1	$at	Reserved for pseudo-instructions
$2 - $3	$v0, $v1	Return values from functions
$4 - $7	$a0 - $a3	Arguments to functions - not preserved by subprograms
$8 - $15	$t0 - $t7	Temporary data, not preserved by subprograms
$16 - $23	$s0 - $s7	Saved registers, preserved by subprograms
$24 - $25	$t8 - $t9	More temporary registers, not preserved by subprograms
$26 - $27	$k0 - $k1	Reserved for kernel. Do not use.
$28	$gp	Global Area Pointer (base of global data segment)
$29	$sp	Stack Pointer
$30	$fp	Frame Pointer
$31	$ra	Return Address

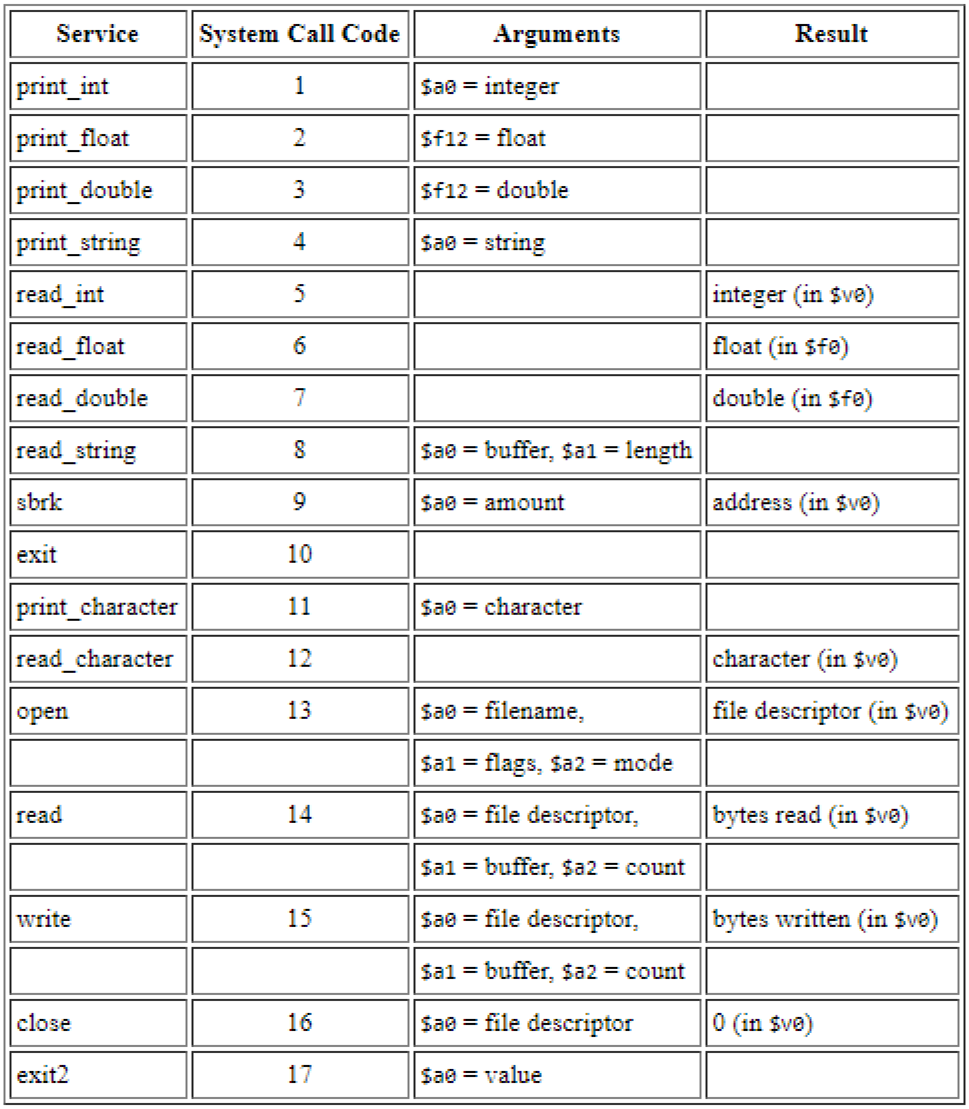
Figure 7 [48]

`Final word count: 46911`

As can be seen from the table, the registers each have their associated usage convention, some of which I have followed and others I did not due to their impracticality when put in the context of my code generator's implementation.

$v0, $v1

As stated above, conventionally these are used for carrying the return values of functions. $v0 has a special purpose where in order to perform syscalls, the system call code must first be loaded into this register. System calls are platform dependent however since I'd planned on running all my programs in SPIM, which itself provides a small set of operating system like services through the syscall instruction, I felt the usage of $v0 in this manner is something I should account for.



Service	System Call Code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		
print_character	11	$a0 = character	
read_character	12		character (in $v0)
open	13	$a0 = filename,	file descriptor (in $v0)
		$a1 = flags, $a2 = mode	
read	14	$a0 = file descriptor,	bytes read (in $v0)
		$a1 = buffer, $a2 = count	
write	15	$a0 = file descriptor,	bytes written (in $v0)
		$a1 = buffer, $a2 = count	
close	16	$a0 = file descriptor	0 (in $v0)
exit2	17	$a0 = value	

Figure 8 [49]

Ultimately I would only ever make use of syscalls 1 and 10. 10 because I wanted to explicitly have each program terminate after reaching the end of the main function (although I doubt this was really necessary), and 1 because I wanted to have a print statement for both debug purposes and for use in demonstrations.

Since I wanted to avoid any potential complications, I opted to avoid using $v0 for anything other than holding system call codes. In retrospect this wasn't really necessary. If I were to only use it to hold the return values of functions, at least with the pair of syscalls I implemented there wasn't any chance of data loss through the contents of $v0 overwritten at inopportune moments. This was a preemptive precaution to avoid any potential problems further down the line in case I ended up implementing any other system calls.

$v1 on the other hand I ended up using solely for holding return values of functions. Again this was a precaution, since I wanted to avoid the possibility of data loss through overwriting paired with the fact I had so many general purpose registers to work with, it seemed entirely plausible and sensible to assign a single purpose to this register, thus avoiding any possibility of complications.

$a0 - $a3

I completely ignored the usage convention for these registers, as it would have needlessly complicated the implementation for a slight performance boost. Since the arguments for functions would be stored in the program's stack anyway, I didn't see the necessity to have them explicitly stored in specific registers. Additionally, since functions in C can have up to 127 arguments, I found it hard to reason about sticking to a convention where four of those arguments would be initially held in registers[50].

Since $a0 also has specific uses related to system calls, and I had yet to run into any issues with my register allocation strategy, I initially avoided using these registers other than $a0, which I only used to hold values which were to be printed. I later however ran into some issues with register allocation, where registers were having their values overwritten before they were meant to be used. The easiest fix for this problem was to simply increase the number of registers I was using for general purposes (storing addresses, performing arithmetic etc). In order to accomplish this, I ended up designating $a0 - $a3 to specifically storing memory addresses ($a0 was also still used for printing, since in that case there was still no chance of any data being accidentally overwritten).

$t0 - $t7 / $t8 - $t9

These registers have no special purposes, and as far as I understand have no particular conventional usage. Initially, these were the only registers I was using (in fact in the prototype I was only using registers $t0, $t1, $t2 and $v0). I used $t0 to $t3 to store values, and $t4 to $t7 to store memory addresses. This strategy appeared to be working fine until I had issues with data loss shortly after implementing functions. In response to this problem I slightly modified my strategy to simply utilize more registers. Now I use $t0 to $t9 for values and $a0 to $a3 for memory addresses.

$ra

This register is automatically assigned the location of the next program instruction after a function call (if the `jal` instruction is used). In other words it holds the return address of a function. Correctly maintaining the value stored in this register is vital when implementing functions. It is also necessary when implementing recursion that the value held by this register be stored on the stack before the recursive call is made, since making another function call will overwrite the contents of this registers, however the return address of the calling function is still vital information, as without it, it is impossible to navigate back to the initial function call.

$fp, $sp

Mips has dedicated registers for storing the location of the top of the stack and the location on the stack of the top of the current stack frame. In contrast, 6502 has a stack pointer but no frame pointer, so in order to maintain stack frames, it is necessary to allocate some space in memory as a pseudo-register to perform the role of the frame pointer. I used both of these registers in my implementation of functions, which I will discuss in a later section.

Instructions

Mips has several instructions, most of which were not necessary in order to implement such a small subset of C, so I'll briefly go over the ones I used and how I utilized certain instructions where the usage may not seem too obvious[51].

Data Transfer

`li   (load immediate) - Loads a register with a constant value. Used alongside la to assign integer literal to variables`

`la   (load address) - Loads a register with a memory address location. Necessary for any assignment operation.`

`lw   (load word) - Loads a register with a value held in memory. Necessary whenever any value held by a variable is used, for example in an expression.`

`sw   (store word) - Stores the contents of a register in memory. Necessary for any assignment operation.`

`move (move) - Moves the contents of one register to another register. I mainly used this to manipulate the frame pointer.`

Program Traversal

j   (jump) - Jumps to a specified location in the program. Necessary for escaping loops and else if chains

jal (jump and link) - Jumps to a specified location in the program and stores the location of the next instruction in $ra. This is effectively a function call.

beqz (branch if equal to zero) - Jumps to a specified location in the program under the condition that the value held in a specified register is equal to zero. Necessary for creating conditional statements such as loops and if statements.

jr   (jump return) - Jumps to the location held in $ra. This is effectively a function return.

Arithmetic

addi (add immediate) - Sets the contents of a specified destination register to the sum of the contents of a specified source register and constant value. I specifically used this for stack pointer manipulation, and never actually used it to evaluate expressions.

add (add) - Sets the contents of a specified destination register to the sum of the contents of two specified source registers.

sub (subtract) - Sets the contents of a specified destination register to the result of the subtraction of the contents of source register A from course register B.

mul (multiply) - Sets the contents of a specified destination register to the product of the contents of two specified source registers.

div (divide) - Performs division using the contents of two specified source registers. The result is held in the accumulator, and must be accessed using instructions ‘mflo’ and ‘mfhi’ The high 4 bytes contain the result of the division, whilst the low 4 bytes contain the remainder. Because of both the result of the division and the remainder is stored, this was used to perform both division and modulus operations in expressions.

Final word count: 46911

mflo (move from low) - Moves the contents of the four low bytes of the accumulator to a specified destination register. Necessary to retrieve the result of a modulo operation.

mfhi (move from high) - Moves the contents of the four high bytes of the accumulator to a specified destination register. Necessary to retrieve the result of a division operation.

Inequalities

seq (source equal to) - Sets the contents of a specified destination register to 1 if the contents of two specified source registers are equal, and sets it to 0 otherwise.

sge (source greater than or equal to) - Sets the contents of a specified destination register to 1 if the contents of a specified register A is greater than or equal to the contents of a specified register B, and sets it to 0 otherwise.

sgt (source greater than) - Sets the contents of a specified destination register to 1 if the contents of a specified register A is greater than the contents of a specified register B, and sets it to 0 otherwise.

sle (source less than or equal to) - Sets the contents of a specified destination register to 1 if the contents of a specified register A is less than or equal to the contents of a specified register B, and sets it to 0 otherwise.

slt (source less than) - Sets the contents of a specified destination register to 1 if the contents of a specified register A is less than the contents of a specified register B, and sets it to 0 otherwise.

sne (source not equal to) - Sets the contents of a specified destination register to 1 if the contents of two specified source registers are not equal, and sets it to 0 otherwise.

Final word count: 46911

A Prototype Code Generator

In order to get an idea of how I wanted to approach solving the problem of code generation, I decided to write a prototype code generator to work with the IR generated by the grammar presented in the section “My Initial IR generating grammar”.

```
#define INTEGERS "0123456789"

FILE *tac_src;
FILE *tbl_src;
FILE *asm_src;

char src_buf[32];
char *pch;
char *opr1, opr2;

bool is_temp(char* tmp) {
    return tmp[0] == '_' && tmp[1] == 't';
}

char* is_int(char* tmp) {
    return strchr(INTEGERS, *tmp);
}

int main() {

    asm_src = fopen("eac.a", "w");

    /* read from symbol table and generate .data segment of mips code */
    tbl_src = fopen("eac.tbl", "r");
    fprintf(asm_src, "\t.data\n");
    while (fgets(src_buf, sizeof src_buf, tbl_src)) {
        fprintf(asm_src, "%c:\t.word 0x0\n", *src_buf);
    }
    fclose(tbl_src);

    /* read from the intermediate code file and generate the mips program */
    tac_src = fopen("eac.tac", "r");
    fprintf(asm_src, "\t.text\nmain:\n");
    while (fgets(src_buf, sizeof src_buf, tac_src)) {
        src_buf[strlen(src_buf) - 1] = '\0';
        pch = strtok(src_buf, " \t");
        while (pch) {
            // is either a temp or an identifier
            if (is_temp(pch)) {
                pch = strtok(NULL, " \t");
                if (*pch == '=') {
                    pch = strtok(NULL, " \t");
                    if (is_int(pch)) {
                        // TEMP = INTEGER_LITERAL
                        fprintf(asm_src, "\tli $t0, %s\n", pch);
```

Final word count: 46911

```
                    } else if (is_temp(pch)) {
                        pch = strtok(NULL, " \t");
                        if (*pch == '+') {
                            pch = strtok(NULL, " \t");
                                if (is_int(pch)) {
                                    // TEMP = TEMP + INTEGER_LITERAL
                                    fprintf(asm_src, "\taddi $t0, $t0, %s\n", pch);
                                } else {
                                    // TEMP = TEMP + IDENTIFER
                                    fprintf(asm_src, "\tla $t2, %s\n", pch);
                                    fprintf(asm_src, "\tlw $t1, ($t2)\n");
                                    fprintf(asm_src, "\tadd $t0, $t0, $t1\n");
                                }
                            }
                        } else {
                            // TEMP = IDENTIFER
                            fprintf(asm_src, "\tla $t1, %s\n", pch);
                            fprintf(asm_src, "\tlw $t0, ($t1)\n");
                        }
                    }
                } else {
                    // IDENTIFER = TEMP
                    fprintf(asm_src, "\tla $t1, %s\n", pch);
                    fprintf(asm_src, "\tsw $t0, ($t1)\n");
                    pch = strtok(NULL, " \t");
                    pch = strtok(NULL, " \t");
                }
                pch = strtok(NULL, " \t");
            }
        }
        fclose(tac_src);

        /* generate code to terminate program */
        fprintf(asm_src, "\tli $v0, 10\n");
        fprintf(asm_src, "\tsyscall\n");
        fclose(asm_src);

        return 0;
}
```

The entire program is effectively an ad hoc parser for the IR the aforementioned parser emits, one thing to mention however is that although the grammar permits the use of conditional if statements, this original implementation of the code generator does not generate code for them.

Simply put the code generator works by reading the contents of the two files generated by the parser to create an output file containing Mips assembly which matches the semantics of the original C program.

`Final word count: 46911`

With that being said, this code generator is of generating code for the following:

Variable Declaration

As I mentioned previously, my original IR generating grammar outputs to two files, one which contains the TAC, and another which contains variable names.

```
/* read from symbol table and generate .data segment of mips code */
tbl_src = fopen("eac.tbl", "r");
fprintf(asm_src, "\t.data\n");
while (fgets(src_buf, sizeof src_buf, tbl_src)) {
    fprintf(asm_src, "%c:\t.word 0x0\n", *src_buf);
}
fclose(tbl_src);
```

This particular code segment is executed at the start of the code generation process, and essentially generates a head for the Mips program. Within this header, each variable declaration is made by simply reading the variable name from the .tbl file and statically allocates each variable before the main program generation begins. Each variable is zero initialized here also.

C code	Generated Output

int x;
int y;
int z;
	.data
	x: .word 0x0
	y: .word 0x0
	z: .word 0x0

This strategy for variable declaration works perfectly fine during this stage of development where all variables were global, however this quickly became obsolete once scope was introduced, since different variables in a program can be declared with the same name, as long as they are in different scopes. For the purposes of simply getting some code generating however, this was a quick and easy way to get something which worked.

Assignment and Addition

Moving into the main loop, this code generator works by simply treating any sequence of characters between white space as tokens. This was a strategy which I continued with in later versions of the code generator, as although it is a very simple strategy, due to how I'd designed the IR, I never had to employ anything more complex.

In this version of the IR there were only two things which could have a value assigned to them, a temporary variable, and an identifier. Identifiers can be fairly complex strings of characters, and this would require regular expression in order to detect, however because temporary variables are easily identified, and only two things could be assigned to, identifying what was being assigned to was as simple as testing whether or not something is a temporary variable, and if it isn't, simply assume it is an identifier.

```
bool is_temp(char* tmp) {
    return tmp[0] == '_' && tmp[1] == 't';
}

...

pch = strtok(src_buf, " \t");
while (pch) {
    // is either a temp or an identifier
    if (is_temp(pch)) {
        /* logic for assignment to temporary variables */
    } else {
        /* logic for assignment to identifiers */
    }
    pch = strtok(NULL, " \t");
}
```

After this step, it was simply a matter of identifying what could be assigned to each of these. For identifiers this was simple, as only temporary variables are ever assigned to identifiers.

```
        } else {
            // IDENTIFER = TEMP
            fprintf(asm_src, "\tla $t1, %s\n", pch);
            fprintf(asm_src, "\tsw $t0, ($t1)\n");
            pch = strtok(NULL, " \t");
            pch = strtok(NULL, " \t");
        }
```

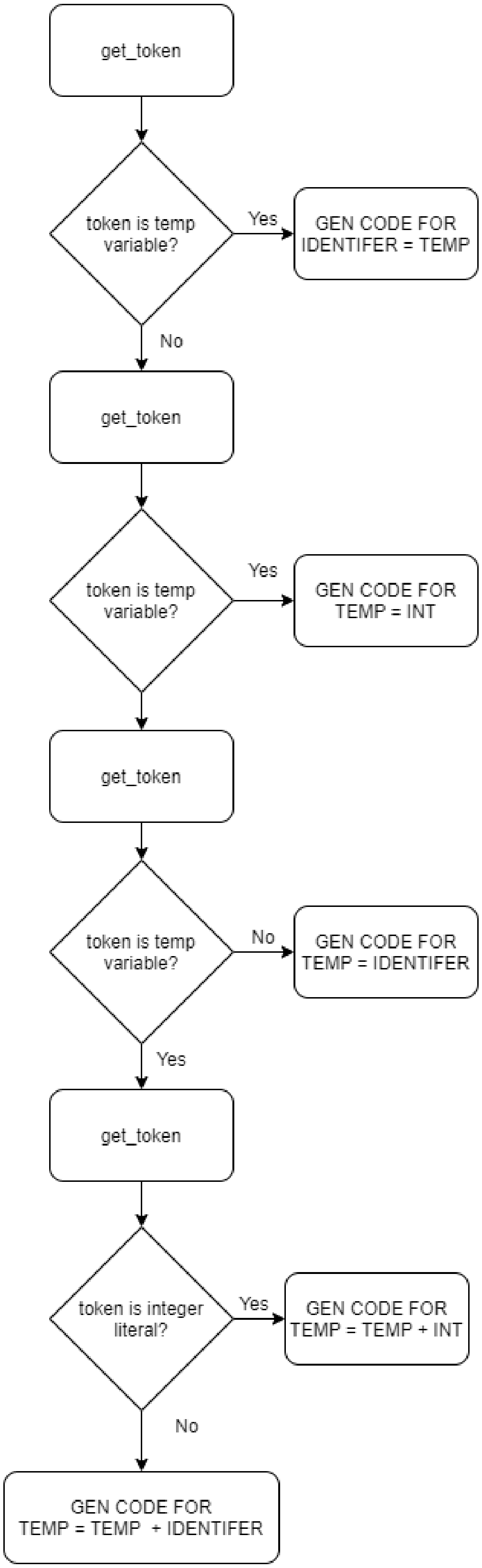
Assignments to temporary variables on the other hand could come in the following forms:

```
TEMP = INTEGER_LITERAL
TEMP = TEMP + INTEGER_LITERAL
TEMP = TEMP + IDENTIFIER
TEMP = IDENTIFIER
```

Each of these possibilities is checked in sequence until one of the possibilities is identified. If none are found, the next like of TAC is simply read, however this shouldn't ever happen since the parser guarantees the IR is always formatted correctly.

Below I've provided a flowchart of this whole process. When compared with the code, the flowchart below does not include the checks for the equals and plus symbols, as whether or not these checks are performed is inconsequential in the current build of the program, and were placed there to account for the fact that I may have added other arithmetic operators later.

Final word count: 46911


get_token
token is temp variable?
Yes
GEN CODE FOR IDENTIFER = TEMP
No
get_token
token is temp variable?
Yes
GEN CODE FOR TEMP = INT
get_token
token is temp variable?
No
GEN CODE FOR TEMP = IDENTIFER
Yes
get_token
token is integer literal?
Yes
GEN CODE FOR TEMP = TEMP + INT
No
GEN CODE FOR TEMP = TEMP + IDENTIFER

Final word count: 46911

Once each line of the .tac file has been read, the code generator also outputs code to explicitly terminate the generated program.

```
/* generate code to terminate program */
fprintf(asm_src, "\tli $v0, 10\n");
fprintf(asm_src, "\tsyscall\n");
```

This approach served as a good first step to solving some of the problems associated with writing a code generator, however this program had limited usefulness moving forward, due to the fact that the code is in no way modular. Adding new features to the languages involved further nesting of an already heavily nested if statement.

By the final version of this code generator, I had implemented subtraction, if statements and print statements, however due to its lack of modularity the code was difficult to work with, so it was not long before I scrapped this code generator in place of a far more sophisticated one, which I will talk about in a later section.

Additions To The Grammar

Before reworking the whole grammar to accommodate the addition of functions, I added several features. As I showed before, the first version of my IR generating grammar could only handle declaration, assignment, addition and if statements. By the final version of this particular grammar, I had added the following:

- While loops
- Break statements
- Subtraction
- Print statements
- Compound statements

I also refactored the grammar to improve readability and make it easier to work with.

```
SYMBOL_TABLE(global 101 31
                             symbol_compare_string
                             symbol_hash_string
                             symbol_print_string
                             [* char* id; *]
                      )

USES("stdlib.h")
USES("ma_aux.h")
USES("tbl_aux.h")

TRNS_UNIT ::= [* fp = fopen("eac.tac", "w");
                              ft = fopen("eac.tbl", "w");
                              loop_stack = create_label_stack();
                        *]
```

Final word count: 46911

```
                        { DECL | STMT }
                        [* fclose(fp);
                              fclose(ft);
                              free(loop_stack);
                        *].

DECL ::= 'int' ID:name (* variable declaration *)
            [* if (symbol_lookup_key(global, &name, NULL)) {
                          text_message(TEXT_ERROR, "redeclaration of '%s'\n", name);
                      } else {
                             symbol_insert_key(global, &name, sizeof(char*),
sizeof(global_data));
                      }
               *]
               [ '=' EXPR:val [* fprintf(fp, "\t%s = %s\n", name, val); *] ]
               [* fprintf(ft, "%s\n", name); // print new symbol table entry to file *]
               ';'.

STMT ::= ( ASSIGN | PRINT | BREAK ) ';' | COND | LOOP.

ASSIGN ::= ID:name
                 [* if (!symbol_lookup_key(global, &name, NULL)) {
                        text_message(TEXT_ERROR, "'%s' undeclared\n", name);
                     }
                 *]
                 [ '=' EXPR:val [* fprintf(fp, "\t%s = %s\n", name, val); *] ].

PRINT ::= 'print' '(' EXPR:val ')'
                [* fprintf(fp, "\tprint %s\n", val);
                *].

COND ::= 'if' '(' EXPR:val ')'
            [* char* end = new_lab();
                  char* lab = new_lab();
                  fprintf(fp, "\tif %s == 0 goto %s\n", val, lab);
            *]
            '{' COMP_STMT '}'
            [* fprintf(fp, "\tgoto %s\n", end);
                  fprintf(fp, "%s\n", lab);
            *]
            { 'elif' '(' EXPR:val ')'
                 [* lab = new_lab();
                    fprintf(fp, "\tif %s == 0 goto %s\n", val, lab);
                 *]
                 '{' COMP_STMT '}'
                 [* fprintf(fp, "\tgoto %s\n", end);
                    fprintf(fp, "%s\n", lab);
              *]
            }
            [ 'else' '{' COMP_STMT '}' ]
            [* fprintf(fp, "%s\n", end); *].

LOOP ::= 'while' '(' EXPR:val ')'
```

Final word count: 46911

```
                [* char* begin = new_lab();
                      char* end = new_lab();
                      push(loop_stack, end);
                      fprintf(fp, "%s\n", begin);
                      fprintf(fp, "\tif %s == 0 goto %s\n", val, end);
                *]
                '{' COMP_STMT '}'
                [* fprintf(fp, "\tgoto %s\n", begin);
                      fprintf(fp, "%s\n", end);
                      char* label = pop(loop_stack);
                *].

COMP_STMT ::= { STMT }.

BREAK ::= 'break' [* char* label = pop(loop_stack);
                                      if (!label) {
                                            text_message(TEXT_ERROR, "use of 'break' must be
within a loop\n");
                                      } else {
                                            fprintf(fp, "\tgoto %s\n", label);
                                      }
                                *].

EXPR:char* ::= PRIMITIVE:val1
                  [* result = new_temp();
                        fprintf(fp, "\t%s = %s\n", result, val1);
                        val1 = result;
                  *]
                  { '+' PRIMITIVE:val2 (* addition *)
                    [* result = new_temp();
                      fprintf(fp, "\t%s = %s + %s\n", result, val1, val2);
                      val1 = result;
                    *]

                  | '-' PRIMITIVE:val2 (* subtraction *)
                    [* result = new_temp();
                      fprintf(fp, "\t%s = %s - %s\n", result, val1, val2);
                      val1 = result;
                    *]
                  }.

PRIMITIVE:char* ::= INTEGER:val (* integer literals *)
                           [* result = (char*) mem_malloc(12);
                              sprintf(result, "%li", val);
                           *]
                         | ID:name (* variables *)
                           [* if (!symbol_lookup_key(global, &name, NULL)) {
                                  text_message(TEXT_ERROR, "'%s' undeclared\n", name);
                              } else {
                                  result = name;
                              }
                           *].
```

Readability has been improved by splitting assignment and declaration into their own rules. This

Final word count: 46911

also promotes modularity in the grammar since an alternate can’t be placed into another rule as easily as a non terminal can be.

```
PRINT ::= 'print' '(' EXPR:val ')'
                [* fprintf(fp, "\tprint %s\n", val);
                *].
```

Printing was added specifically as a debug feature. Since at this point I was actually generating code which I could run in spim, it was useful to provide a way to see some output. Initially when testing the mips programs, I would have to manually write the instructions to make the syscalls which printed for example, the results of expressions in order to verify they had been implemented correctly. Being able to use a print statement in the C code, and have this produce some output in the program became a far more efficient way to verify programs were working correctly, since I wouldn’t have to manually edit the asm files each time.

Because of this ‘print’ became an addition to my intermediate form. This somewhat breaks conventions given that printing isn’t system agnostic, which is why I’ve been particular about marking this down as a debug feature.

```
EXPR:char* ::= PRIMITIVE:val1
                  [* result = new_temp();
                        fprintf(fp, "\t%s = %s\n", result, val1);
                        val1 = result;
                  *]
                  { '+' PRIMITIVE:val2 (* addition *)
                    [* result = new_temp();
                      fprintf(fp, "\t%s = %s + %s\n", result, val1, val2);
                      val1 = result;
                    *]

                  | '-' PRIMITIVE:val2 (* subtraction *)
                    [* result = new_temp();
                      fprintf(fp, "\t%s = %s - %s\n", result, val1, val2);
                      val1 = result;
                    *]
                  }.
```

Subtraction has been implemented similar to addition, and they are part of the same rule in order to maintain their equal operator priority.

Final word count: 46911

```
COND ::= 'if' '(' EXPR:val ')'
            [* char* end = new_lab();
                  char* lab = new_lab();
                  fprintf(fp, "\tif %s == 0 goto %s\n", val, lab);
            *]
            '{' COMP_STMT '}'
            [* fprintf(fp, "\tgoto %s\n", end);
                  fprintf(fp, "%s\n", lab);
            *]
            { 'elif' '(' EXPR:val ')'
                 [* lab = new_lab();
                    fprintf(fp, "\tif %s == 0 goto %s\n", val, lab);
                 *]
                 '{' COMP_STMT '}'
                 [* fprintf(fp, "\tgoto %s\n", end);
                    fprintf(fp, "%s\n", lab);
              *]
            }
            [ 'else' '{' COMP_STMT '}' ]
            [* fprintf(fp, "%s\n", end); *].
```

In this version of the grammar, I've changed 'otherwise' to else and 'else if' to 'elif'. There's no much justification for this besides thinking it was a better way to circumvent the issue caused by the conflict (python uses elif, so it's more recognisable). I did however want to make it possible to have both inline if statements as well as those followed compound statements. Unfortunately I was unable to get this to work, due to FIRST FOLLOW conflicts. In the end I simply settled for only having compound statements follow, since they provide the most functionality. This is an unfortunate concession I had to make since it meant the language was slightly less C-like, however I personally was unable to find a way around this particular limitation of LL1 grammars.

```
LOOP ::= 'while' '(' EXPR:val ')'
               [* char* begin = new_lab();
                     char* end = new_lab();
                     push(loop_stack, end);
                     fprintf(fp, "%s\n", begin);
                     fprintf(fp, "\tif %s == 0 goto %s\n", val, end);
               *]
               '{' COMP_STMT '}'
               [* fprintf(fp, "\tgoto %s\n", begin);
                     fprintf(fp, "%s\n", end);
                     char* label = pop(loop_stack);
               *].
```

Loops behave similarly to if statements in that they produce a condition jump in the IR. Loops however require a label at the start of the code block and an unconditional jump at the end in order to loop back on themselves.

Final word count: 46911

```
while (x) {
    x = x - 1;
}
```

```
L1:
    la $a1, x           # load address of x
    lw $t1, ($a1)       # load value of x
    beqz $t1, L2        # end loop if x is 0
    la $a2, x           # load address of x
    lw $t2, ($a2)       # load value of x
    li $t3, 1           # load value 1
    sub $t4, $t2, $t3 # subtract 1 from x
    la $a0, x           # load address of x
    sw $t4, ($a0)       # store new value of x
    j L1                # loop once more
L2:
```

In order to implement ‘break’ I implemented a stack which keeps track of whether or not the parser is inside a loop. If there is nothing on the stack, the parser is not inside a loop, otherwise it is. If there are 3 items on the stack it means the parser is in a triple nested loop, however knowing the nesting level isn’t important, what’s important is having something to pop off the stack whenever a loop is exited.

The exact implementation of the stack isn’t particularly important, however in the current implementation, only the end label of the loop is held on the stack, however it is necessary to also hold the start label in order to implement ‘continue’.

```
TRNS_UNIT ::= [* fp = fopen("eac.tac", "w");
                 ft = fopen("eac.tbl", "w");
                 loop_stack = create_label_stack();
              *]
              { DECL | STMT }
              [* fclose(fp);
                 fclose(ft);
                 free(loop_stack);
              *].
```

An empty stack specifically for keeping track of loops is created at the start of the parse, and subsequently freed at the end.

```
LOOP ::= 'while' '(' EXPR:val ')'
               [* char* begin = new_lab();
                      char* end = new_lab();
                      push(loop_stack, end);
                      fprintf(fp, "%s\n", begin);
                      fprintf(fp, "\tif %s == 0 goto %s\n", val, end);
               *]
```

Each time a loop is entered, the end label of the loop is pushed to the stack. The end label is necessary because when break is called, the program needs to know where to jump to.

Final word count: 46911

```
LOOP ::= 'while' '(' EXPR:val ')'
               [* char* begin = new_lab();
                     char* end = new_lab();
                     push(loop_stack, end);
                     fprintf(fp, "%s\n", begin);
                     fprintf(fp, "\tif %s == 0 goto %s\n", val, end);
               *]
               '{' COMP_STMT '}'
               [* fprintf(fp, "\tgoto %s\n", begin);
                     fprintf(fp, "%s\n", end);
                     char* label = pop(loop_stack);
               *].
```

When the loop is exited the label is popped off the stack.

```
BREAK ::= 'break' [* char* label = pop(loop_stack);
                     if (!label) {
                         text_message(TEXT_ERROR, "use of 'break' must be within a loop\n");
                     } else {
                         fprintf(fp, "\tgoto %s\n", label);
                     }
                  *].
```

When break is called, the parser attempts to pop from the stack. If the stack is empty, it detects that break was called outside of a loop and returns an error. If the stack is nonempty, an unconditional jump is made to the end label popped off the stack, which will always be the end label of the last loop the program entered.

This is actually a bug which I later fixed, where break should be peeking the stack rather than popping it. It should only pop when the parser leaves the loop.

Continue is implemented in a similar way, except it jumps to the start of the loop, rather than to the end, hence why both the start and end label must be pushed to the stack in order to implement both.

Final word count: 46911

Improving the Code Generator

As I previously mentioned, the functionality of the prototype code generator was not easily extended due to how the code was structured. In order to continue making extensions to the language, I decided on doing a total rewrite of the code generator.

The prototype consisted of a single source file where file io, parsing and code generation was all performed. If I wanted to make the new program modular I knew I had to avoid writing a monolithic main function again.

In order to do this I would first need to identify the individual concerns of the program and split them into their respective translation units.

I considered moving file io out of the main function but in the end decided against it since the structural benefit would be minimal, and the logic was identical to that of the prototype's. Parsing logic and code generation logic however would each have their own translation units.

With parsing no longer being intertwined with code generation, I would need some way to transfer the information collected by the parser, to the code generator. This would come in the form of quads. Parsing each line of TAC would result in a quad being formed which would then be handed over to the code generator for the target architecture. Since interpreting TAC and forming quad are two separate concerns, these would each have their own translation unit.

I wanted to also open up the possibility to target multiple architectures, so I decided I'd have a specific translation unit to generate mips code.

With this set of ideas in mind, I ended up with the following set of translation units:

cgen.c - Contains the main function where file io is performed. From here, calls are made within the main loop to other translation units which contain the parsing and code generation logic.

quad.c - Contains most of the logic for parsing the IR. Its main purpose is to identify the operation, arguments and result associated with each like of TAC, and place them into a quad so they can be handed over to the code generator for the target architecture.

tac.c - Contains information associated with three address code instructions, such as a set of possible keywords, and a set of possible operations. It also functions for determining what individual tokens are, such as temporary variables, and integers. This is mainly used by quad.c to assist in the parsing process, by resolving the identity of individual tokens and determining the operation each line of TAC is meant to perform.

Final word count: 46911

target.h - Contains the type definition for targets. A target acts as a system agnostic template for machine architectures such as mips. It contains a set of generic operations such as add and jump, which can be pointed towards a function containing the implementation for the operation for a specific architecture. This is how targeting multiple architectures is facilitated.

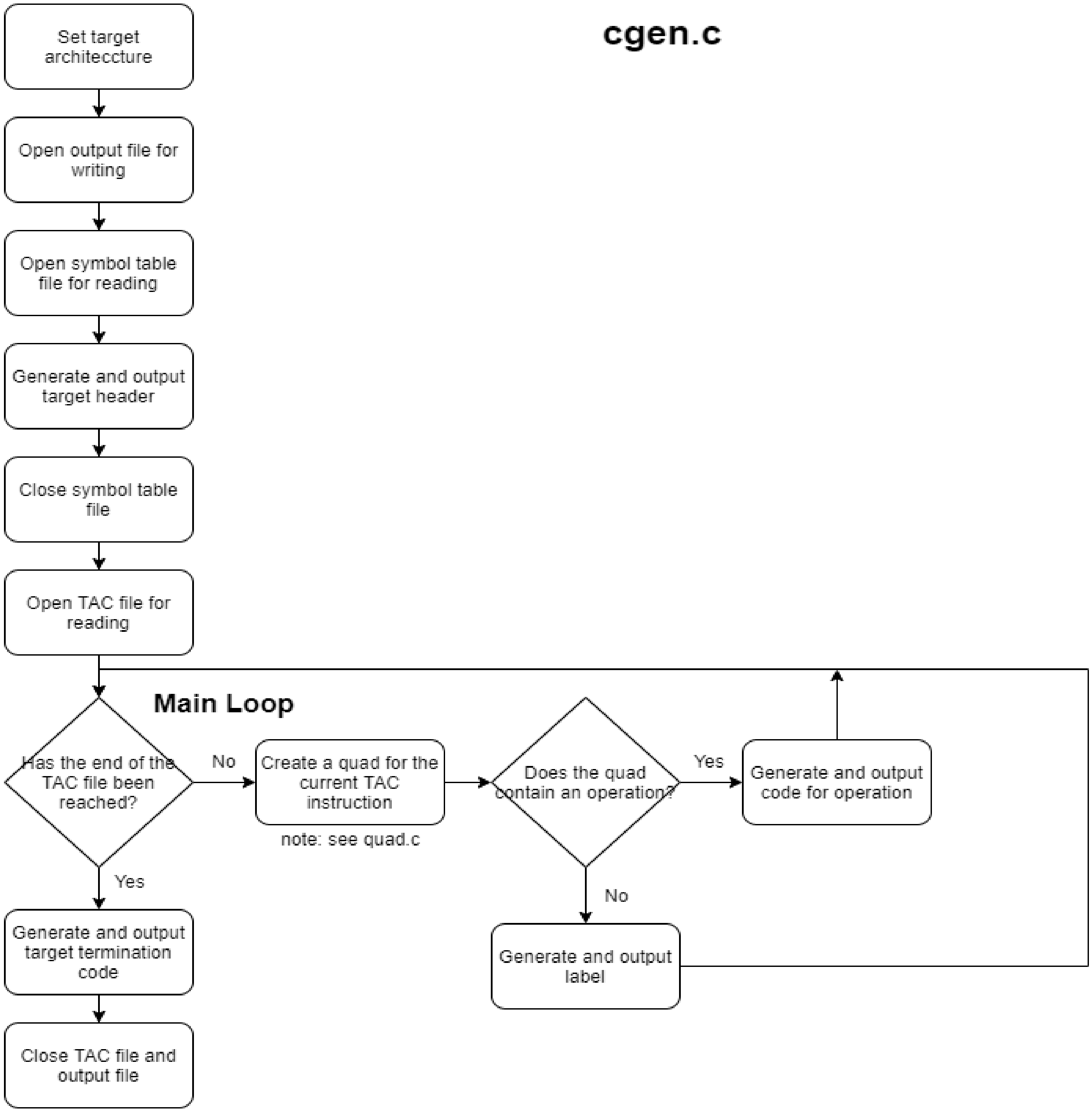
mips.c - Provides a set of functions which contain code generation templates for the mips architecture. It also contains the logic for the register allocation strategy for mips.

These descriptions only provide a brief overview of the code generators process. The following section will provide a more detailed explanation.

Final word count: 46911

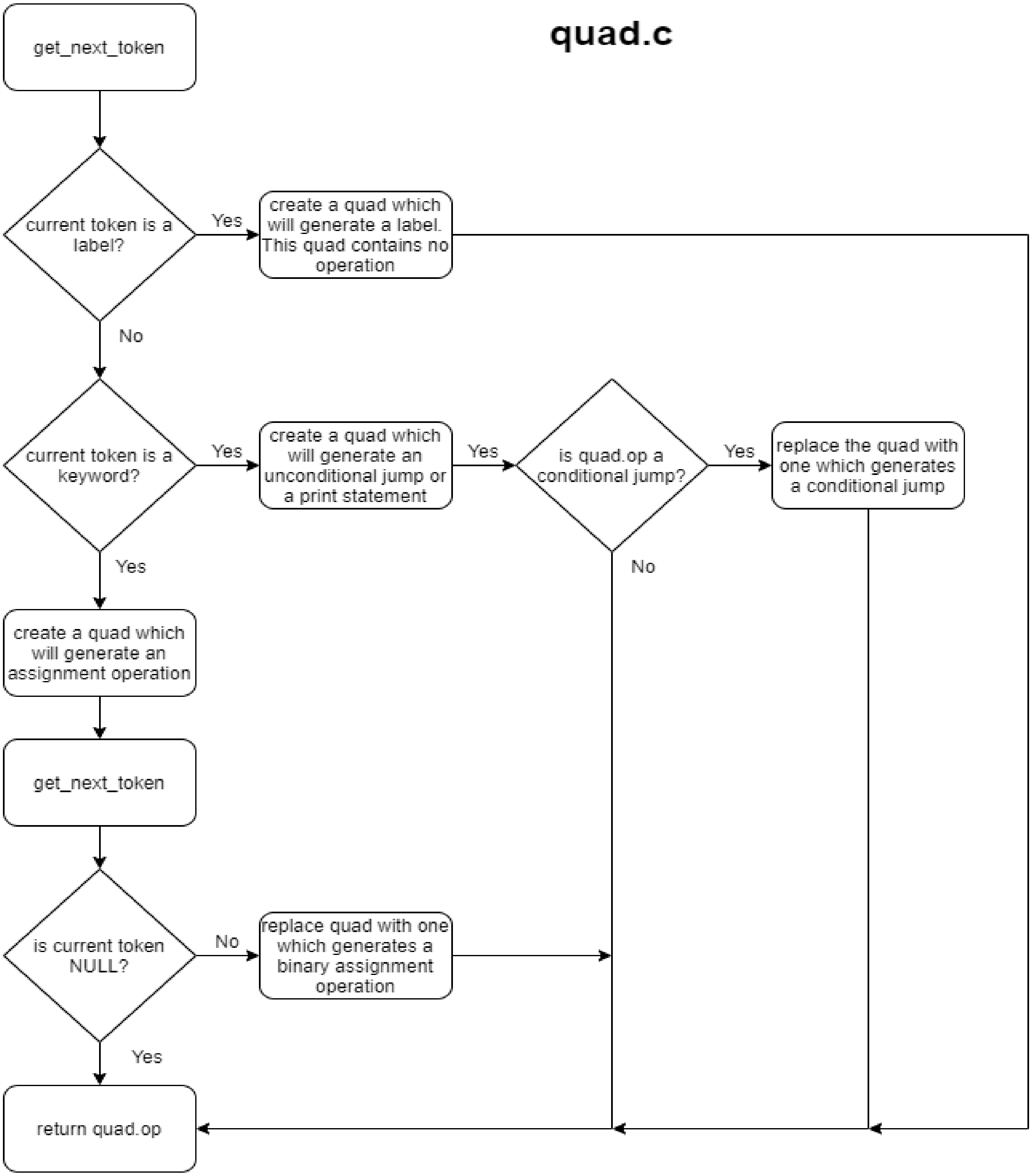
The Process of Generating Code

The main loop is still run from within the cgen source file. It has no responsibilities beyond setting the target architecture, opening and closing resource and output files, and providing a central location from which other translation units can communicate.


cgen.c
Set target architeccture
Open output file for writing
Open symbol table file for reading
Generate and output target header
Close symbol table file
Open TAC file for reading
Main Loop
Has the end of the TAC file been reached?
No
Create a quad for the current TAC instruction
note: see quad.c
Does the quad contain an operation?
Yes
Generate and output code for operation
No
Generate and output label
Yes
Generate and output target termination code
Close TAC file and output file

“Create a quad for the current TAC operation” makes a call to quad.c in order to retrieve an operation, and a set of arguments with which to generate code from. If there is no operation present in the quad, then a label is generated. All code generation is handled by mips.c, the process of which I will describe shortly.

Before any code can be generated, a line of TAC must be parsed. There is a finite number of TAC instruction formats, so the process of creating a quad is effectively determining which format the current instruction is in, and collecting the attributes of the instruction in the quad. These attributes being the operation, the result and up to two arguments. Any number of these attributes can be present or missing depending on the format of the instruction.


quad.c
get_next_token
current token is a label?
Yes
create a quad which will generate a label. This quad contains no operation
No
current token is a keyword?
Yes
create a quad which will generate an unconditional jump or a print statement
Yes
is quad.op a conditional jump?
Yes
replace the quad with one which generates a conditional jump
No
Yes
create a quad which will generate an assignment operation
get_next_token
is current token NULL?
No
replace quad with one which generates a binary assignment operation
Yes
return quad.op

Final word count: 46911

After the quad is formed, the operation is returned to the main function, so it can be used to locate the function which will generate the necessary code.

Type Definition of quad

```
typedef struct _quad {
      char *op;
      char *arg1;
      char *arg2;
      char *result;
} quad;
```

Possible Keywords

```
if
goto
print
```

Possible Operations

Operation	Operation Type	TAC Format	Quad Layout			
			op	arg1	arg2	result
=	Assignment	result = arg1	=	TEMP / VAR / INT	**NULL**	TEMP / VAR
+	Binary Assignment	result = arg1 op arg2	+	TEMP	TEMP	TEMP
-	Binary Assignment	result = arg1 op arg2	-	TEMP	TEMP	TEMP
==	Binary Assignment	result = arg1 op arg2	==	TEMP	TEMP	TEMP
>=	Binary Assignment	result = arg1 op arg2	>=	TEMP	TEMP	TEMP
>	Binary Assignment	result = arg1 op arg2	>	TEMP	TEMP	TEMP
<=	Binary Assignment	result = arg1 op arg2	<=	TEMP	TEMP	TEMP
<	Binary Assignment	result = arg1 op arg2	<	TEMP	TEMP	TEMP
!=	Binary Assignment	result = arg1 op arg2	!=	TEMP	TEMP	TEMP
if	Conditional Jump	op arg1 == 0 goto result	if	TEMP	**NULL**	LABEL
goto	Unconditional Jump	op arg1	goto	LABEL	**NULL**	**NULL**
print	Print	op arg1	print	TEMP	**NULL**	**NULL**

One thing to take note of is that results can only ever be temporary variables. This is because results are often directly translated to registers (this is also the case for arguments), and the register allocation strategy uses the number associated with temporary variables to determine which register will be used.

Final word count: 46911

A change was made to the grammar in order to facilitate this where all assignment operations would generate a temporary variable which would be used to determine which register would be used.

```
ASSIGN ::= ID:name
           [* if (!symbol_lookup_key(global, &name, NULL)) {
                  text_message(TEXT_ERROR, "'%s' undeclared\n", name);
              }
           *]
            [ '=' EXPR:val [* fprintf(fp, "\t%s = %s\n", name, val); *] ].

EXPR:char* ::= SUM:result.

SUM:char* ::= UNARY:val1 [* result = val1; *]
                      { '+' UNARY:val2 (* addition *)
                        [* result = new_temp();
                           fprintf(fp, "\t%s = %s + %s\n", result, val1, val2);
                           val1 = result;
                        *]

                      | '-' UNARY:val2 (* subtraction *)
                        [* result = new_temp();
                           fprintf(fp, "\t%s = %s - %s\n", result, val1, val2);
                           val1 = result;
                        *]
                      }.

UNARY:char* ::= PRIMITIVE:val1
                [* result = new_temp();
                   fprintf(fp, "\t%s = %s\n", result, val1);
                   val1 = result;
                *].
```

Above is an excerpt from the grammar from the same time as when the code generator was rewritten, except the section which parses boolean expressions has been removed to make it more concise.

The use of expressions is required to assign a value to a variable. Expressions always pass through UNARY, which always generates and outputs a temporary variable.

```
UNARY:char* ::= PRIMITIVE:val1
                [* result = new_temp();
                   fprintf(fp, "\t%s = %s\n", result, val1);
                   val1 = result;
                *].
```

This means before any further processing ever happens, every primitive is always assigned to a temporary variable, and that temporary variable will be used to determine which register that primitive will be held in.

This change results in larger TAC files, however it significantly simplifies the register allocation logic, which given the time restraint of the project felt like a reasonable tradeoff, the alternative being spending more time thinking of a more sophisticated register allocation method in order to facilitate the TAC files which were being output prior to this change.

```
while(fgets(src_buf, sizeof src_buf, tac_src)) {
        src_buf[strlen(src_buf) - 1] = '\0';
        quad_fill(&qd, src_buf);
        if (qd.op) {
                int op = get_op(qd.op);
                ((operation_t)((target_t**)&target)[op])(&qd, dst);
        } else {
                target.label(&qd, dst);
        }
}
```

Above is the main loop which resides in cgen.c. “quad_fill” is the function from quad.c which is used to build the quad for the next section of code to be generated. The operation stored in the quad can be used directly to determine which code generating function the target should use. The following array and function are taken from tac.c.

```
const char *OP[] = {
        "=",     // 1
        "+",     // 2
        "-",     // 3
        "==",     // 4
        ">=",    // 5
        ">",     // 6
        "<=",    // 7
        "<",      // 8
        "!=",     // 9
        "if",    // 10
        "goto",  // 11
        "print", // 12
        NULL
};

int get_op(char *op) {
        for (int i = 0; OP[i]; i += 1) {
                if (!strcmp(op, OP[i])) {
                        return i + 1;
                }
        }
        return 0;
}
```

Above is an array containing the set of possible operations, and the “get_op” function which when given one of the possible symbols, returns a numerical value representing that symbol. This numerical value for each operation can be seen from the comments next to each symbol. Using what is effectively pointer arithmetic, the numerical value can be used to determine which code generating function will be used.

Final word count: 46911

```
typedef void (*header_t)(FILE*, FILE*);
typedef void (*operation_t)(quad*, FILE*);
typedef void (*terminate_t)(FILE*);

typedef struct _target_t {
       header_t header;
       operation_t assign;
       operation_t add;
       operation_t sub;
       operation_t equal;
       operation_t greater_than_or_equal;
       operation_t greater_than;
       operation_t less_than_or_equal;
       operation_t less_than;
       operation_t not_equal;
       operation_t cond_jump;
       operation_t jump;
       operation_t print;
       operation_t label;
       terminate_t terminate;
} target_t;
```

This is the type definition of target, taken from target.h. As I described before, a target is effectively a template for different machine architectures. It provides a set of functions which we would expect architecture to support.

At the top of this file are the definitions for three different function types. The header function is meant to provide a way for different architectures to generate a header in each program. The header function takes handles to the output file and the file containing information from the symbol table. The header would typically contain variable declarations and runtime functions (something I implemented later on to reduce output file sizes).

The termination function simply provides a way for different architectures to generate code which will terminate the program. This function takes the output file as its only argument, since an implicit program termination system call at the bottom of each program doesn’t require any additional resources.

The operation function takes a quad and the output file and is used as a generic function for any operation any architecture should be capable of performing, such as addition and jumping.

It is important that ‘target’ be kept entirely system agnostic so it can always be used as a template for any architecture within reason. I fear the inclusion of printing may have breach this fact, however as stated previously it has been implemented as a debug feature to make program verification a more efficient process.

`Final word count: 46911`

```
void mips_header(FILE *dst, FILE *src);

void mips_assign(quad *qd, FILE *dst);

void mips_add(quad *qd, FILE *dst);

void mips_sub(quad *qd, FILE *dst);

void mips_equal(quad *qd, FILE *dst);

void mips_greater_than_or_equal(quad *qd, FILE *dst);

void mips_greater_than(quad *qd, FILE *dst);

void mips_less_than(quad *qd, FILE *dst);

void mips_less_than_or_equal(quad *qd, FILE *dst);

void mips_not_equal(quad *qd, FILE *dst);

void mips_cond_jump(quad *qd, FILE *dst);

void mips_jump(quad *qd, FILE *dst);

void mips_print(quad *qd, FILE *dst);

void mips_label(quad *qd, FILE *dst);

void mips_terminate(FILE *dst);

static const target_t mips = {
        mips_header,
        mips_assign,
        mips_add,
        mips_sub,
        mips_equal,
        mips_greater_than_or_equal,
        mips_greater_than,
        mips_less_than_or_equal,
        mips_less_than,
        mips_not_equal,
        mips_cond_jump,
        mips_jump,
        mips_print,
        mips_label,
        mips_terminate
};
```

Above are the function signatures for the mips architecture, and the mips target. The mips target simply contains a set of functions which match the template set by the target type. The order is important because of how the numerical value determined by the operation is used to find the correct code generation function.

Final word count: 46911

```
int op = get_op(qd.op);
((operation_t)((target_t**)&target)[op])(&qd, dst);
```

So taking you back to these two lines here from the main function. The operation stored in the quad which represents the current TAC instruction is used to produce an ordinal value which is then used as an index to locate the correct code generation function from the target architecture's set of code generating functions.

Once the function is located, the process then simply boils down to allocating registers and writing the generated assembly code to the output file.

```
void mips_add(quad *qd, FILE *dst) {
	int dst_reg   = get_val_reg(qd->result);
	int src_reg0 = get_val_reg(qd->arg1);
	int src_reg1 = get_val_reg(qd->arg2);
	fprintf(dst, "\tadd $t%i, $t%i, $t%i\n", dst_reg, src_reg0, src_reg1);
}
```

Above is the function which generates the code for the mips add instruction. It allocates registers for the instruction’s three arguments and writes to the output file.

```
int get_val_reg(char *temp) {
	return (temp[strlen(temp) - 2] - '0') % 4;
}
int get_adr_reg(char *temp) {
	return 4 + get_val_reg(temp);
}
```

These are the two register allocation functions for mips. I originally allocated 8 registers to handle all general operations (printing uses $v0 and $a0). These registers are $t0 to $t7. The first four are used to hold values, and the last four are used to hold memory addresses. I decided to make this distinction to avoid the potential for data loss from a register being assigned a value which was holding an address which was still in use, and vice versa.

The register in question is chosen simply by taking the numerical value associated with the temporary variable and modding it against the total number of available registers (which in this case is 4). The same is done for both value registers and address registers, except address registers use an offset so they only use $t3 to $t7.

Now instead of sifting through a forest of if statements, adding new features to the compiler was a much simpler more efficient process, as I no longer had to search for an exact location in the existing logic, and instead could just write a new code generating function in mips.c, add the function to the mips target and target’s type definition, and add the new operation in tac.h. Occasionally there was also a need to update the parsing logic in quad.c, however the program never grew so large that this became confusing.

Final word count: 46911

Implementing Functions

With a modular code generator which was easy to extend the functionality of, I could now continue to focus on adding new features to the language. The point the project was at now approximately at the point I had originally seen it reaching, I wanted it to be possible to use the language to write a program which could generate prime numbers, and the only features missing to achieve this were division and modulo, which would simply be extensions to the already existing logic for expressions.

During a meeting with my supervisor however, it was suggested to me that I should attempt to implement either functions or code optimisations. Neither seemed like particularly trivial tasks, however functions to me were the clear choice due to the fact they would visibly extend the functionality of the language as a whole. The idea of having a language which looked and behaved like C on the surface was my primary goal.

I had a basic understanding of how functions worked. I knew they were subroutines and I knew the program would jump back and forth from them, and I knew what a stack frame was and that I knew i'd have to find a way to automatically generate all these things in order to get functions working.

Large steps of the process could be broken down into the following tasks:

- Figure out what goes into a stack frame
- Figure out how to write and execute subroutines in mips
- Figure out how to manipulate the stack in mips
- Figure out how to implement scope using rdp
- Figure out how to model function calls in TAC

There was also writing the code generation logic to consider, although assuming everything above had been worked out correctly this wouldn't be too trying of a task.

<u>Scope</u>

I assumed implementing scope would likely be one of the more problematic areas, so I decided a good first step would be to write a grammar which implemented the semantics of scope. This would require:

- Having a global scope where variables which were declared there could be recognised from all other scopes
- Creating scope chains with variables which were accessible from within their respective scope chains, but were no longer accessible once they had fallen out of scope
- Shadowing so variables declared lower in a scope chain with identical names of variables declared higher in the scope chain take precedence.

After reading parts of the rdp support library manual to identify which functions I would need to use to leverage rdp's inbuilt scope support, I came up with the following implementation:

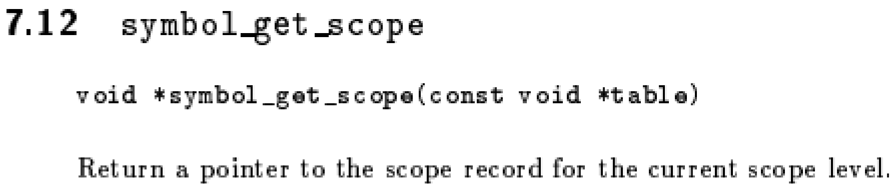
```
SYMBOL_TABLE(table 101 31
             symbol_compare_string
             symbol_hash_string
             symbol_print_string
             [* char* id; int i*]
            )

USES("ma_aux.h")
USES("stack.h")

A ::= [* stk_push(&scope_stack, symbol_get_scope(table)); *] S.

S ::= DECL
    | [* void *scp = symbol_get_scope(table); *]
      SCOPE
      [* symbol_set_scope(table, scp); *]
    | CALL.

DECL ::= 'var' ID:name '=' INTEGER:val
         [* void *scp = symbol_get_scope(table);
            if (symbol_lookup_key(table, &name, scp)) {
                printf("'%s' has already been declared in this scope\n", name);
            } else {
                table_cast(symbol_insert_key(table, &name, sizeof(char*), sizeof(table_data)))->i = val;
            }
        *] [ S ].

CALL ::= ID:name
        [* void *scp = symbol_get_scope(table);
           node_t *node = stk_peek(&scope_stack);
           void *key;
           while (node) {
               if ((key = symbol_lookup_key(table, &name, node->data))) {
                   break;
               }
               node = node->prev;
           }
           if (!node) {
               printf("'%s' has not been declared in any existing scope\n", name);
           } else {
               printf("%d\n", table_cast(key)->i);
           }
        *] [ S ].

SCOPE ::= '{' [* void *scp = symbol_new_scope(table, new_lab());
                 stk_push(&scope_stack, scp);
              *]
              [ S ]
          '}' [* symbol_unlink_scope(scp);
                 free(stk_pop(&scope_stack));
              *] [ S ].
```

This is a very simple grammar where you can either declare a variable, "use" a previously declared variable (for no particular purpose other than simplifying displaying its value), or open up a new scope. Variable declarations and usages are both possible within scopes, and it is also possible to create scopes withins scopes.

```
A ::= [* stk_push(&scope_stack, symbol_get_scope(table)); *] S.
```

I initially had some issues traversing the scope chain. This was due to the fact that the version of rdp I was using was an older version which had certain bugs in the scope implementation. I recognised this by noticing discrepancies in the source file I was looking at between the version on my laptop where I did most of my project work, and the version on my pc which turned out to be a newer version of rdp without these issues. I ended up writing my own scope chain however in order to circumvent any further issues.

I begin the scope chain with the following function:



7.12 symbol_get_scope

```
void *symbol_get_scope(const void *table)
```

Return a pointer to the scope record for the current scope level.

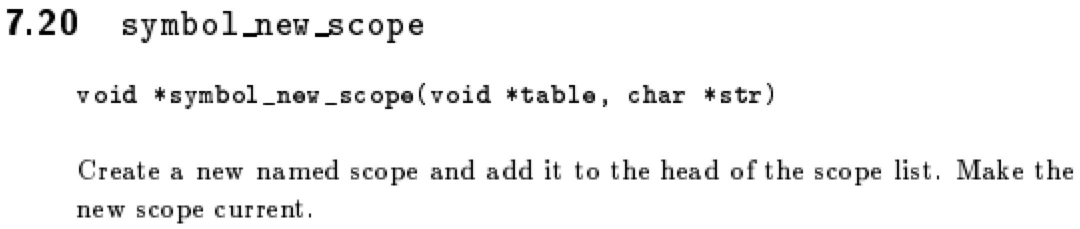
Figure 9 [52]

What this line effectively does is it adds the global scope to my scope chain.

```
S ::= DECL
    | [* void *scp = symbol_get_scope(table); *]
      SCOPE
      [* symbol_set_scope(table, scp); *]
    | CALL.
```

Focusing on opening scopes for a moment, before a new scope is opened, the current scope is stored so it can be immediately retrieved after the new scope closes.

Upon opening a new scope, the following rdp function is used:



7.20 symbol_new_scope

```
void *symbol_new_scope(void *table, char *str)
```

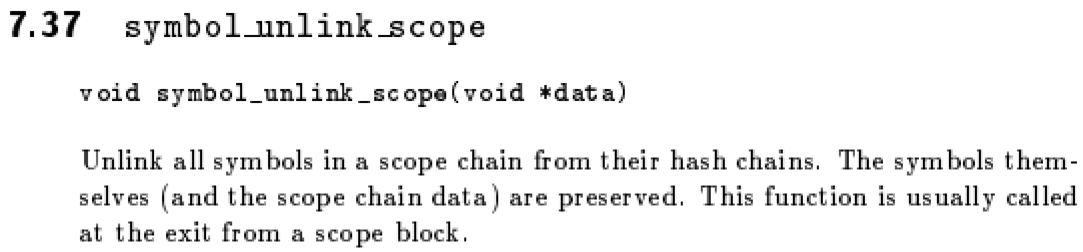
Create a new named scope and add it to the head of the scope list. Make the new scope current.

Figure 10 [53]

This function will return a new scope which should automatically be linked to the current active scope chain within rdp (although this is the exact feature which caused me problems, and why I used my own scope chain in the end).

```
SCOPE ::= '{' [* void *scp = symbol_new_scope(table, new_lab());
                 stk_push(&scope_stack, scp);
              *]
              [ S ]
          '}' [* symbol_unlink_scope(scp);
                 free(stk_pop(&scope_stack));
              *] [ S ].
```

Whenever a new scope is opened, I link the scope rdp generates for me and link it to my own scope chain (I implemented the scope chain using stack, hence the function call “stk_push”). When a scope is closed, the following function is used to remove it from the rdp scope chain:



7.37 symbol_unlink_scope

```
void symbol_unlink_scope(void *data)
```

Unlink all symbols in a scope chain from their hash chains. The symbols themselves (and the scope chain data) are preserved. This function is usually called at the exit from a scope block.

Figure 11 [54]

I also pop the scope from my stack, removing it from my own scope chain.

```
DECL ::= 'var' ID:name '=' INTEGER:val
         [* void *scp = symbol_get_scope(table);
            if (symbol_lookup_key(table, &name, scp)) {
                printf("'%s' has already been declared in this scope\n", name);
            } else {
                table_cast(symbol_insert_key(table, &name, sizeof(char*), sizeof(table_data)))->i = val;
            }
        *] [ S ].
```

When a variable is declared, only the current scope needs to be checked to ensure the variable isn’t being redeclared in the same scope. The same functions are used to retrieve and submit symbols to the symbol table as before, however the lookup function is now searching the current scope, rather than searching the whole table, which is what occurs when the last argument is set to NULL.

If a redeclaration is found in the same scope, this is reported, however an error is not thrown since this was simply a proof of concept grammar.

Final word count: 46911

```
CALL ::= ID:name
        [* void *scp = symbol_get_scope(table);
           node_t *node = stk_peek(&scope_stack);
           void *key;
           while (node) {
               if ((key = symbol_lookup_key(table, &name, node->data))) {
                   break;
               }
               node = node->prev;
           }
           if (!node) {
               printf("'%s' has not been declared in any existing scope\n", name);
           } else {
               printf("%d\n", table_cast(key)->i);
           }
        *] [ S ].
```

When retrieving variables from the table, all scopes current active scopes must be taken into account, however the scope chain must be traversed sequentially as the most recent scopes take precedence.

As I previously mentioned, I had trouble traversing the scope chain which was internal to rdp, so I instead used my own external scope chain which achieved the desired results. The loop attempts to find a matching symbol in the current scope, and if it fails to do so will keep traversing higher up the scope chain until it reaches the global scope. If it finds no matching symbol in the global scope, then the variable has not been declared. This idea of traversing the scope chain sequentially is what gives more recent scopes precedence and what also allows shadowing to be possible.

Stack Frames

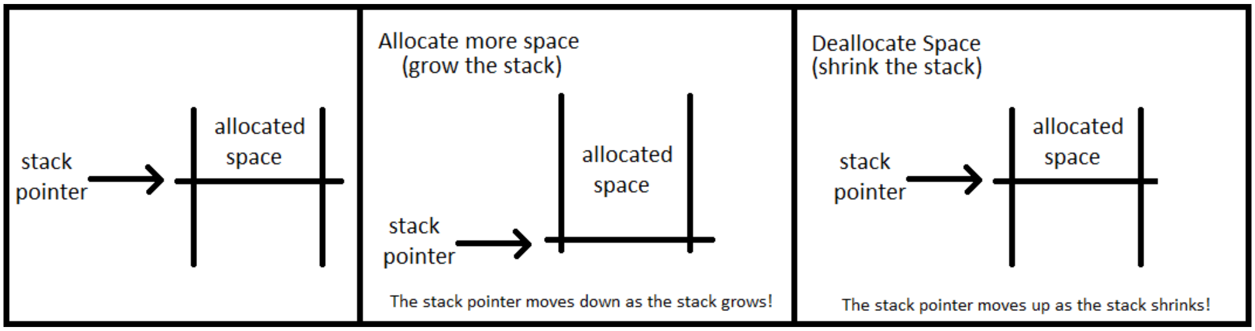
Having gotten a working model for implementing scope, the next order of business was to work out the structure of a stack frame and what went into one.

Before we can go into any details on the development process, it is necessary to understand what the program stack is, what a stack frame is and how the stack pointer and frame pointer are used to implement these two concepts in a typical machine architecture.

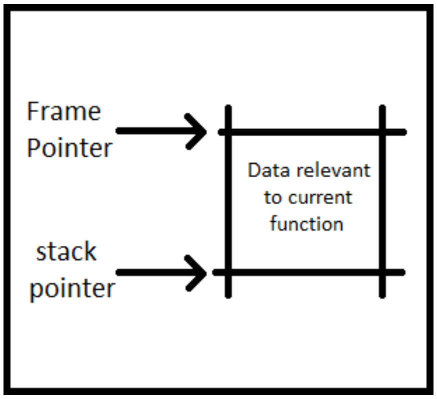
The program stack is an area in memory which a program uses for temporary data storage. The stack grows and shrinks as needed relative to the amount of temporary data storage needed at any given time while the program is running[55].

When a function call is made by a program, a certain amount of space is allocated to the stack. This space is referred to as a “stack frame”. The stack frame will hold any data which is relevant to the function, and when the function returns, the stack frame is deallocated from the stack[56].

The stack pointer is a register which always points to the top of the stack. When the stack grows, the stack pointer is moved further down, and when the stack shrinks it is moved further up. This sounds a bit strange, but the stack grows downwards, and shrinks upwards, so it is essentially an upside down stack.


stack pointer
allocated space
Allocate more space (grow the stack)
allocated space
stack pointer
The stack pointer moves down as the stack grows!
Deallocate Space (shrink the stack)
stack pointer
allocated space
The stack pointer moves up as the stack shrinks!

The frame pointer is a register which always points to the top of the current stack frame.



Both the stack pointer and the frame pointer are vital for retrieving data from the stack as they provide locations which are always relative to the locations of data stored on the stack. I will better illustrate this point further down.

After sifting through various resources I found one which gave me the following list:[57]

- The return address
- Arguments passed to the function
- Variables local to the function
- Saved copies of any registers modified by the subprogram that need to be restored

It is necessary to store the return address of the function on the stack frame because without it it would be impossible to return back to where the function call was made from.

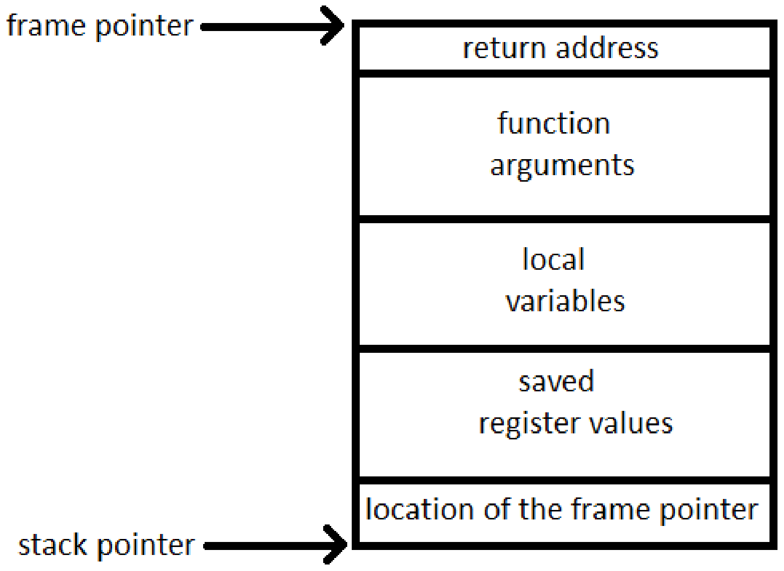
The reason why function arguments and local variables are pushed to the program stack will become clearer further into this section, however in brief it is because the subroutine needs a direct method of accessing these variables.

The values stored within the registers must be stored because it must be possible for the program to pick up from where it left off before the function call was made. There is a fair likelihood that during the course of the subroutine many of the registers will have their values overwritten, however upon returning from the function, many of these values may still have been relevant to the task being performed by the calling function.

There is one thing this resource failed to mention, which is that the location of the frame pointer must also be stored within the current stack frame. This is necessary because when a stack frame is deallocated, it is then necessary to move the frame pointer back to its previous location, which was the top of the stack frame of the calling function.

To give an example, at the start of a program the stack frame for the main function is created, and the frame pointer points to the top of the main functions stack frame. When a function call is made from the main function, a new stack frame is created and the frame pointer is moved down the stack to point at the top of the new stack frame. When that function returns, its stack frame is removed from the stack, and the frame pointer must then be moved back to the top of the main function's stack frame, the location of which is retrieved from the main function's stack frame, and had this not been the case, there would be no way to relocate the frame pointer back to its original position.

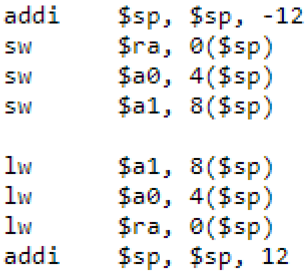
With all the necessary components understood I came up with the following model of a complete stack frame:



The specific order of the information in the stack frame holds significant importance with regards to my implementation, the reasons for which I will explain further down, however in brief this is because the data held directly after and directly before the the frame and stack pointers are more easily retrievable.

Stack Manipulation and Subroutines in Mips.

Having determined the structure of my stack frames, I next needed to work out how stack manipulation is performed using mips, and how I could implement subroutines.



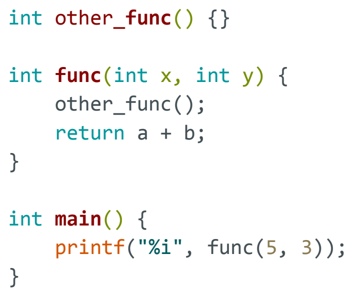
```
addi    $sp, $sp, -12
sw      $ra, 0($sp)
sw      $a0, 4($sp)
sw      $a1, 8($sp)

lw      $a1, 8($sp)
lw      $a0, 4($sp)
lw      $ra, 0($sp)
addi    $sp, $sp, 12
```

Figure 12 [58]

Allocating space on the stack can be performed by using the “add immediate” instruction. Subtracting from $sp, which is the stack pointer register results in the stack pointer being moved down, effectively growing the stack and allocating new space. Adding to $sp has the opposite result of moving the stack pointer upwards, effectively shirking the stack and deallocating space.

The “store word” instruction can be used to store information on the stack. By providing an offset, data can be stored on the stack relative to the position of the stack pointer. Data can be retrieved in the same way by using “load word”.

The frame pointer ($fp), can be manipulated in the same way, so information can be stored and retrieved relative to both the stack pointer and the frame pointer. The following example will provide an idea of how this can be utilized.



```c
int other_func() {}

int func(int x, int y) {
    other_func();
    return a + b;
}

int main() {
    printf("%i", func(5, 3));
}
```

Above is a simple C program containing a main function, a function which returns the sum of two integers, and an empty function.

Final word count: 46911

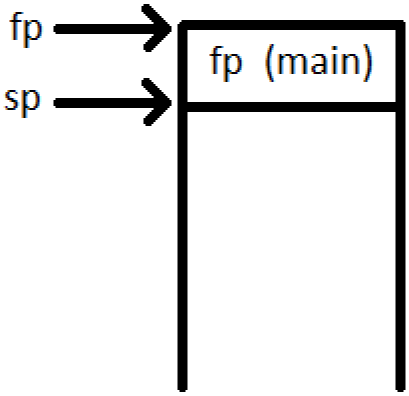
```
        .text

other_func:
        jr $ra                   # return from other_func

func:
        sw   $ra, 0($fp)         # place the return address of the function at the top of the stack frame
        jal  other_func          # call other_func
        lw   $t0, -4($fp)        # retrieve the first argument
        lw   $t1, -8($fp)        # retrieve the second argument
        add  $v1, $t0, $t1       # sum up the two arguments
        lw   $ra, 0($fp)         # retrieve the return address
        jr $ra                   # return from func

main:
        move $fp, $sp            # move the frame pointer to the same location as the stack pointer
        addi $sp, $sp, -4        # allocate space on the stack to store the fp's current location
        sw   $fp, 4($sp)         # push the fp's location (the top of the main function) to the stack
        move $fp, $sp            # move the frame pointer down to the top of the new stack frames
        addi $sp, $sp, -12       # allocate space on the stack for func’s stack frame
        li   $a0, 5              # ...
        li   $a1, 3              # ...
        sw   $a0, -4($fp)        # place the first argument into the stack frame
        sw   $a1, -8($fp)        # place the second argument into the stack frame
        jal  func                # call func
        addi $sp, $sp, 12        # pop the stack frame
        move $a0, $v1            # retrieve the return value of func
        lw   $fp, 4($sp)         # restore the frame pointer's original location
        li   $v0, 1              # print the result of func
        syscall                  # ...
        li   $v0, 10             # terminate the program
        syscall                  # ...
```

This is a program written in mips which attempts to mimic the C program above. This example attempts to demonstrate some of the ideas we’ve discussed so far regarding the use of stack frames.

```
main:
        move $fp, $sp            # move the frame pointer to the same location as the stack pointer
        addi $sp, $sp, -4        # allocate space on the stack to store the fp's current location
        sw   $fp, 4($sp)         # push the fp's location (the top of the main function) to the stack
```

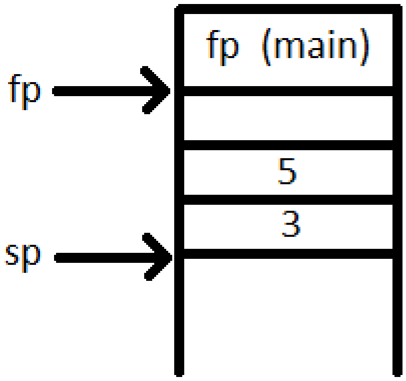
The program starts off by simply setting the position of the frame pointer to the position of the stack pointer, effectively creating the first stack frame of the program. Space is then allocated on the stack to store the current location of the frame pointer, in preparation for when the current stack frame needs to be restored after the upcoming function call. The frame pointer’s location is pushed to the stack using an offset from the stack pointer.

Final word count: 46911



```
        move $fp, $sp           # move the frame pointer down to the top of the new stack frames
        addi $sp, $sp, -12      # allocate space on the stack to store the arguments to the function
        li   $a0, 5             # ...
        li   $a1, 3             # ...
        sw   $a0, -4($fp)       # place the first argument into the stack frame
        sw   $a1, -8($fp)       # place the second argument into the stack frame
        jal  func               # call func
```

The frame pointer is then moved down in order to start creating func's stack frame. Enough space is allocated to store the return address, and the two arguments to the function. The arguments of the function are pushed to the stack using offsets to the frame pointer. The function is then called.



```
func:
        sw   $ra, 0($fp)        # place the return address of the function at the top of the stack frame
        jal  other_func         # call other_func
        lw   $t0, -4($fp)       # retrieve the first argument
        lw   $t1, -8($fp)       # retrieve the second argument
        add  $v1, $t0, $t1      # sum up the two arguments
        lw   $ra, 0($fp)        # retrieve the return address
        jr $ra                  # return from func
```

Above our main function is the subroutine “func”. Despite being before main in the programs order, main is executed first because spim always expects a main function which it will always execute first.

Final word count: 46911

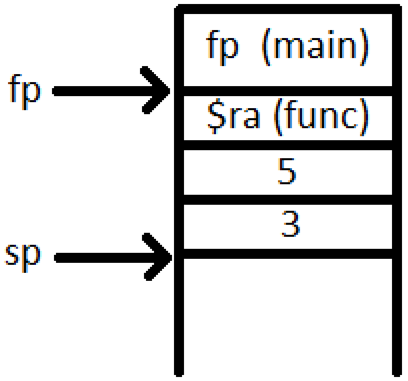
The return address of the function is stored on the stack and a call to “other_func” is made. Because ‘jal’ has been called, $ra now has its value overwritten.

```
other_func:
        jr $ra                  # return from other_func
```

“Other_func” simply returns. I’ve not bothered building a stack frame for this function because I’ve only included it to demonstrate why it is necessary to store return addresses on the stack.

```
        lw   $t0, -4($fp)       # retrieve the first argument
        lw   $t1, -8($fp)       # retrieve the second argument
        add  $v1, $t0, $t1      # sum up the two arguments
        lw   $ra, 0($fp)        # retrieve the return address
        jr $ra                  # return from func
```

After returning to “func”, the arguments which we had pushed to the stack from the main function are retrieved using the same offsets which were used to store them. This demonstrates how the frame pointer can be used as a reference point to locate data needed by the function. If the function had local variables, we’d simply have specific offsets for where they were located on the stack, and data could be stored there and accessed from there in the exact same way as has been demonstrated above.



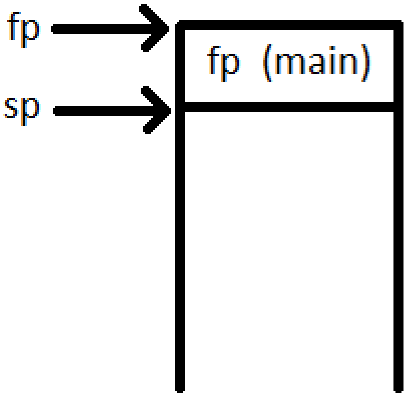
Currently $ra holds the return address of “other_func”, however before making this function call, the return address of “func” was stored on the stack, so even though a function call was made from “func” it is still possible to return to main, simply by retrieving the return address from the stack.

```
        addi $sp, $sp, 12       # pop the stack frame
        move $a0, $v1           # retrieve the return value of func
        lw   $fp, 4($sp)        # restore the frame pointer's original location
        li   $v0, 1             # print the result of func
        syscall                 # ...
        li   $v0, 10            # terminate the program
        syscall                 # …
```

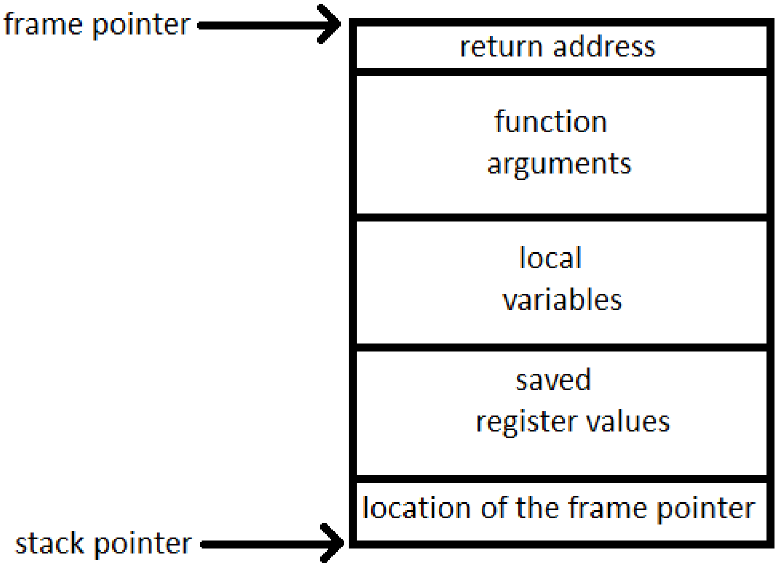
Upon returning from main the space allocated on the stack for “func” is deallocated, effectively popping the stack frame from the stack, as it is no longer needed.

Final word count: 46911

The stack pointer has been returned to its original location before the stack frame for “func” was made, placing it just below where the original frame pointer was stored. This is the same offset which the frame pointer was stored with, meaning the same offset can be used to retrieve it now the stack pointer has returned to this location. This will always be the case, as the amount allocated for a stack frame when a function call is made will always be equal to the amount deallocated.



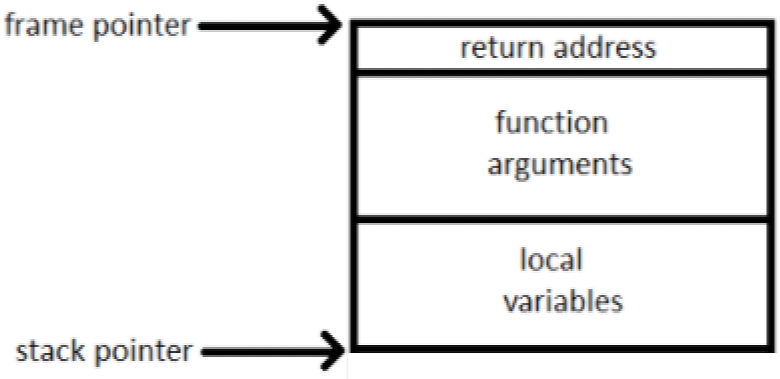
The rest of the program is simply the return value of “func” being printed and the program terminating. But this should have at least provided a much clearer idea of how stack manipulation is performed using mips, and how it is possible to retrieve specific data from the stack using offsets from the frame pointer.



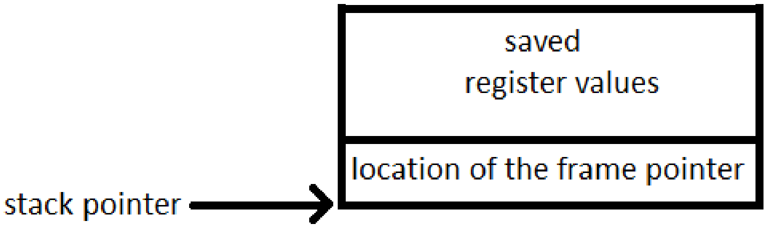
This takes us back to the model of the stack frame, although certain information was omitted in the example program for simplicity. Specifically, the reason why the return address is stored at the top, is because the number of arguments and local variables can vary, however if the return address is always stored in this location, regardless of how many local variable and parameters

Final word count: 46911

there the function has, the return address will always be retrievable with the same offset from every function. The frame pointer is always stored at the bottom of the stack frame for essentially the same reason, however there is a slight nuance to this.



If no function call is made from within a function, the stack frame can simply take the form demonstrated above. The return address, arguments and local variables of a function are always necessary, however the registers and frame pointer need only be restored if the program were to return from another function. This means the registers and frame pointer can be appended to the stack frame if and when another function call is made.

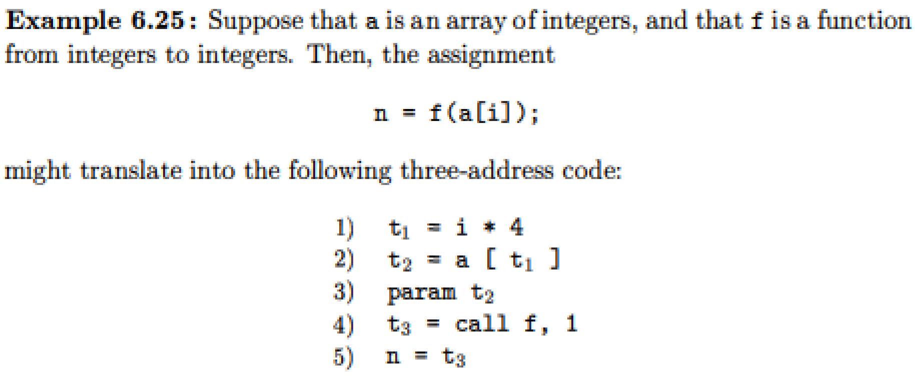


Because this portion of the stack frame is appended, its distance from the frame pointer can vary due to the variable number of parameters and local variables. This isn’t a problem however, because the stack pointer will always sit at the bottom of the current stack frame after returning from a function. This paired with the fact that my register allocation strategy always only ever uses a fixed number of registers for general use means the appended data items can always be located using offsets from the stack pointer.

Final word count: 46911

Representing Function Calls in Three Address Code

Now that I had a solid idea of the code I'd have to generate, I needed to figure out what the code temples were, and what the associated three address code would be.



Example 6.25: Suppose that a is an array of integers, and that f is a function from integers to integers. Then, the assignment

```
n = f(a[i]);
```

might translate into the following three-address code:

1) t_1 = i * 4
2) t_2 = a [t_1]
3) param t_2
4) t_3 = call f, 1
5) n = t_3

Figure 13 [59]

I found the example above in the dragon book. `t`$_3$ `= call f` seemed like a good place to start. It made sense that a function call would assign a somewhere, since they can return values, and the fact it assigns a value to a temporary variable works well with my current model.

Having an instruction to push parameters onto the stack also seemed very practical, however I decided to take my implementation in a slightly different direction, since I felt it would be a good idea to avoid having unnecessary keywords where possible.

Reducing the problem down to its simplest form, I needed to be able to do three things:

- Call functions
- Manipulate the stack
- Return from functions

This led to having the following three instructions:

`push`

Push's main task would be managing the stack, however due to the nature of managing the stack being quite a broad task, paired with the restrictions posed by the parsing order on how I could implement the semantics, push ended up having several responsibilities, and thus could take several different arguments.

`push $fp` - this effectively means "create new stack frame". This meant it would append the registers and the frame pointer to the stack frame, however it would not move the frame pointer itself, as this would be handled by "call".

`push $sp` - This instruction specifically creates the stack frame for the main function. The main function has two special cases compared to other functions:

- Returning from the main function terminates the program so it doesn't have a return address (This might not be entirely accurate in reality, but within the context of my compiler's implementation this was certainly true)
- The main function needs to initialize the frame pointer (move $fp, $sp)

I wanted to avoid having special cases littered across the language semantics and the code generator, and found this to be the best solution for that.

`push $ra` - This specifically pushes the return address onto the stack. This action can not be performed by "call", because the return address isn't known until after the function call is made. This means the return address has to be pushed onto the stack from within the subroutine. In other words this code is generated by the function declaration, not the function call, and since the main function behaved differently, this code couldn't simply be generated implicitly beneath function labels, at least not without writing a special case, which was something I wanted to avoid.

`push INT` - This allows the stack to be grown and shrunk by an arbitrary size. I originally added this because my plan was to have the parser count the number of arguments and local variables in a function and allocate all the space needed with a single push call, however due to restrictions imposed by the grammar, stack allocation ended up being split into 3 stages:

- Allocating space for the return address
- Allocating space for the arguments
- Allocating space for the local variables

In the end this instruction was relegated to allocating space specifically for the return address, which means whenever it is used, it is always
"push 4" (the return address is always 4 bytes long). Every "push 4" is also preceded by "push $fp" so the instruction is actually redundant, and space allocation for the return address could simply be appended to "push $fp".

`push $tn INT` - This pushes arguments of functions into the stack. Because each argument is an expression, the result of which is assigned to a temporary variable, this instruction takes a temporary variable as an argument in order to retrieve the result of the expression. The integer is the argument's offset from the frame pointer. This was added as part of the plan to bulk allocate space for functions, however in the current implementation, parameters are actually pushed onto the stack sequentially as the stack grows, so they're actually pushed on relative to the stack pointer rather than the frame pointer.

`Final word count: 46911`

call

Call handles moving the frame pointer into its new position, and space allocation for local variables. It also handles calling the function and restoring the stack frame of the calling function. Call is always followed by a label, which should be the name of a function, and can optionally be assigned to a variable.

return

Return retrieves the return address from the stack, pops the stack frame and returns to the calling function. Return has an optional argument, which allows it to return a value.

The Semantics of Functions

Because the inclusivity of functions changed the grammar so drastically, I took this as an opportunity to rewrite the grammar once more. The grammar and semantics would need to change so drastically that it would be easier to do a rewrite rather than modify the current implementation.

The size of the grammar had grown significantly by this point, and many parts hold little relevance to the implementation of functions so rather than show the entire thing in this section, I'll try to mention only the relevant parts.

```
SYMBOL_TABLE(table 101 31
             symbol_compare_string
             symbol_hash_string
             symbol_print_string
             [* char* id; int type; int offset; int params; int size *]
                         )

TRNS_UNIT ::= [* fp = fopen("func.tac", "w");
                                  ft = fopen("func.tbl", "w");
                                  link_scope(symbol_get_scope(table));
                                  global_scope = *this_scope();
                           *]
                            { DECL:name ( VAR_DECL(name 0) | FUNC_DECL(name) ) }
                           [* fclose(fp);
                                  fclose(ft);
                                  if (!main_scope) {
                                         text_message(TEXT_ERROR, "no main function has been declared\n");
                                  }
                           *].
```

As with all of my grammars, the start symbol is a translation unit. Several features I've the previous grammars are still present, such as the file to write the IR to and the file to write the symbol table to. I'd changed the implementation of the scope chain, but it achieves the same goal as with the scope chain example grammar, “link_scope” simply adds whatever scope you pass it to the end of the scope chain. I also make a point of storing the global scope explicitly for reasons which will become apparent as we explore this grammar.

```
SYMBOL_TABLE(table 101 31
             symbol_compare_string
             symbol_hash_string
             symbol_print_string
             [* char* id; int type; int offset; int params; int size *]
                         )
```

The symbol table now also holds significantly more attributes per symbol. Before the only relevant information was the identifier, however several new attributes needed to be added to account for functions.

`Type` - Because the language now supported integers and functions which returned integers, there needed to be a way to distinguish between the two. For this reason I had to add what is at best a rudimentary type system.

The reason there isn't much in the way of a type system is because the design of the type system revolved around simply having a way to distinguish between integers and functions, rather than being able to distinguish between a larger set of types. I also didn't expect to be adding another large feature to the compiler after getting functions working, so I saw no need to account for this extension to the compiler.

I included a simple header file, type.h to facilitate this:

```
#ifndef TYPE_H
#define TYPE_H

enum type {
        INT = 1,
        FUNC
};

#endif /* TYPE_H */
```

It only contains an enum, however this was useful when I had to refer to types within the language semantics, I could write the type names rather than the values '1' and '2'.

`Offset` - Since local variables and function arguments were stack allocated, representing these was not as simple as having just an identifier for them, it is also necessary to store their offset from the frame pointer. In order for the parser to locate these local variables during the parsing phase, the identifier is still used, since during this stage the variables are still distinguishable by scope, however there IR and the generated program have no concept of scope, thus the offset information is required, as these variables can no longer be referred to by name.

`Params -` Simply stores the number of parameters for any given function. This is necessary for the paraser because it needs to know if the correct number of arguments are given when a function is called.

`Size` - Simply stores the size of the function, so the space for the stack frame can be correctly allocated. In the end this was only used to allocate the space for the local variables specifically.

```
TRNS_UNIT ::= [* fp = fopen("func.tac", "w");
                                   ft = fopen("func.tbl", "w");
                                   link_scope(symbol_get_scope(table));
                                   global_scope = *this_scope();
                           *]
                            { DECL:name ( VAR_DECL(name 0) | FUNC_DECL(name) ) }
                           [* fclose(fp);
                                   fclose(ft);
                                   if (!main_scope) {
                                           text_message(TEXT_ERROR, "no main function has been declared\n");
                                   }
                           *].

DECL:char* ::= 'int' ID:result.
```

Declaring Variables

From the global scope it is possible to declare variables and functions. Both VAR_DECL and FUNC_DECL are preceded by DECL which simply prepends the type ‘int’ and identifier to each of them. This is fine since integers are still the only type.

```
VAR_DECL(name:char* offset:int) ::=
[*  void *key;
        void *scp = *this_scope();
        if (symbol_lookup_key(table, &name, scp)) {
                text_message(TEXT_ERROR, "redeclaration of '%s'\n", name);
        } else {
                key = symbol_insert_key(table, &name, sizeof(char*), sizeof(table_data));
                table_cast(key)->type   = INT;
                table_cast(key)->offset = offset;
                if (scp == global_scope) {
                        fprintf(ft, "%s %d %d\n", name, INT, 0);
                }
        }
*]
[ '=' EXPR:val
[*
  if (offset) {
          fprintf(fp, "\t$fp(%d) = %s\n", offset, val);
  } else {
          fprintf(fp, "\t%s = %s\n", name, val);
  }
*]
] ';'.
```

Variable declarations are mostly the same as in the scope example grammar, however there are a few extra things to take care of.

VAR_DECL handles variable declarations in both local scopes and the global scope. Global variables are still statically allocated and thus, can still be referred to by name by the compiled program.

```
if (scp == global_scope) {
    fprintf(ft, "%s %d %d\n", name, INT, 0);
}
```

Since they are referred to by name in the generated code, the names of global variables still need to be written to the symbol table file. Their type is also written to the file so the code generator knows to statically allocate a variable, rather than print a label for a subroutine, which is what is done for functions. The 0 which is written to the file is unused, and is simply there to make the format of the symbol table file more uniform so that the logic which reads the file could be made slightly simpler.

Variable declarations now also take an additional inherited attribute ‘offset’.

```
VAR_DECL(name:char* offset:int)
```

Offset is used to keep track of a local variable’s offset from the frame pointer.

```
{ DECL:name ( VAR_DECL(name 0) | FUNC_DECL(name) ) }
```

This is simply set to 0 for global variables and is effectively ignored. For local variables however, this value is necessary to represent them in the IR.

```
FUNC_DECL(name:char*) ::=
[* void *key = 0;
   int size = 4;
   int locals = 0;
   if (symbol_lookup_key(table, &name, *this_scope())) {
       text_message(TEXT_ERROR, "redeclaration of '%s'\n", name);
   } else {
       key = symbol_insert_key(table, &name, sizeof(char*), sizeof(table_data));
       table_cast(key)->type = FUNC;
       fprintf(fp, "#%s\n", name);
       if (strcmp(name, "main")) {
               fprintf(fp, "\tpush $ra\n");
           } else {
               fprintf(fp, "\tpush $sp\n");
           }
   }
*]
'(' [* void *scp = symbol_new_scope(table, NULL);
           link_scope(scp);
           if (!strcmp(name, "main")) {
               main_scope = scp;
           }
        *]
[ PARAMS(scp):params
  [* if (key) {
             table_cast(key)->params = params;
             size += (4 * params);
             if (!strcmp(name, "main")) {
                fprintf(fp, "\tpush %i\n", 4 * params);
             }
         }
  *]
]
')'
SCOPE(scp size):declarations
[*      locals = declarations;
        size += (4 * declarations) + 4;
        table_cast(key)->size = size;
        if (scp != main_scope) {
                fprintf(fp, "\treturn\n");
    } else {
        fprintf(fp, "\tterminate\n");
    }
    fprintf(ft, "%s %d %d\n", name, FUNC, locals);
*].
```

Above is the grammar for function declarations. I want to come back to this further down to talk about declaring functions specifically, but for now I want to focus on declaring local variables, but I'm just providing the entire production rule so you can see the whole picture when I show snippets from it.

```
FUNC_DECL(name:char*) ::=
[* void *key = 0;
   int size = 4;
```

When a function is declared, its 'size' is set to 4. This is to account for the fact that the return actress of a function is always present in the stack frame, so there must always be at least 4 bytes to store it.

```
[ PARAMS(scp):params
  [* if (key) {
             table_cast(key)->params = params;
             size += (4 * params);
```

This value is then increased based on the number of parameters the function has. This is necessary because local variables are stored after the functions arguments in the stack frame.

```
SCOPE(scp size):declarations
```

The local scope for the function is then opened up, and the current size is passed as an inherited attribute of SCOPE.

```
SCOPE(scp:void* offset:int):int ::=
[* result = 0; *]
'{' { DECL:name VAR_DECL(name offset)
      [* offset += 4;
```

Variable declarations made from within a scope are local to some function, and therefore will be stored in that function's stack frame.

Final word count: 46911

Each time a variable declaration is made from within a scope, the offset is incremented to account for the fact the next variable is stored further down on the stack.

```
VAR_DECL(name:char* offset:int) ::=
[*  void *key;
        void *scp = *this_scope();
        if (symbol_lookup_key(table, &name, scp)) {
                text_message(TEXT_ERROR, "redeclaration of '%s'\n", name);
        } else {
                key = symbol_insert_key(table, &name, sizeof(char*), sizeof(table_data));
                table_cast(key)->type   = INT;
                table_cast(key)->offset = offset;
                if (scp == global_scope) {
                        fprintf(ft, "%s %d %d\n", name, INT, 0);
                }
        }
*]
[ '=' EXPR:val
[*
  if (offset) {
          fprintf(fp, "\t$fp(%d) = %s\n", offset, val);
  } else {
          fprintf(fp, "\t%s = %s\n", name, val);
  }
*]
] ';'.
```

When a local variable is declared it is represented in the IR, in the form:

```
$fp(X) = TEMP
```

Where X is variable offset from the frame pointer, and TEMP is some temporary variable which will later be used to allocate a register. This code generator can then use this information later to correctly place local variables on the stack when necessary. This is how the distinction is made between global variables and local variables in the code generation stage.

Global	Local
`ID = TEMP`	`$fp(X) = TEMP`

```
SCOPE(new_scope offset):declarations
```

Scopes can also be opened from within scopes, for if statements and while loops. As long as these scopes are part of the same function declaration, the offset keeps getting passed into these scopes to account for any variable declarations within those scope also.

Final word count: 46911

Declaring Functions

```
FUNC_DECL(name:char*) ::=
[* void *key = 0;
   int size = 4;
   int locals = 0;
   if (symbol_lookup_key(table, &name, *this_scope())) {
       text_message(TEXT_ERROR, "redeclaration of '%s'\n", name);
   } else {
       key = symbol_insert_key(table, &name, sizeof(char*), sizeof(table_data));
       table_cast(key)->type = FUNC;
       fprintf(fp, "#%s\n", name);
       if (strcmp(name, "main")) {
               fprintf(fp, "\tpush $ra\n");
           } else {
               fprintf(fp, "\tpush $sp\n");
           }
   }
*]
```

When a function is declared it is given a name and checked to make sure it isn’t a redeclaration, as would be done with a variable declaration.

```
int size = 4;
int locals = 0;
```

The stack frame’s size is initially set to 4 to account for the return address, and the number of local variable declarations is initially set to 0.

```
key = symbol_insert_key(table, &name, sizeof(char*), sizeof(table_data));
table_cast(key)->type = FUNC;
fprintf(fp, "#%s\n", name);
```

After the declared function is successfully submitted to the symbol table with its type ‘FUNC’, the start of the subroutine is represented in the IR by its identifier as a label.

Function declarations can only be done from within the global scope, so there is no special mechanism in place to account for local declarations.

```
if (strcmp(name, "main")) {
    fprintf(fp, "\tpush $ra\n");
} else {
    fprintf(fp, "\tpush $sp\n");
}
```

A special case is however to distinguish between the main function and other functions. You can refer to the previous section for details on what ‘`push $ra`’ and ‘`push $sp`’ do, and how this relates to the differences between the main function and other functions.

Final word count: 46911

```
'(' [* void *scp = symbol_new_scope(table, NULL);
           link_scope(scp);
           if (!strcmp(name, "main")) {
               main_scope = scp;
           }
        *]
[ PARAMS(scp):params
  [* if (key) {
             table_cast(key)->params = params;
             size += (4 * params);
             if (!strcmp(name, "main")) {
                fprintf(fp, "\tpush %i\n", 4 * params);
             }
         }
  *]
]
')'
```

Following the verification of the identifiers and the output of the start of the subroutine, a new scope is created and added to the scope chain, to account for the fact that declarations made from within a function's definition are local to that function’s scope. This includes the parameters, so the scope is passed to PARAMS as an inherited attribute.

```
PARAMS(scp:void*):int ::=
  DECL:name
  [* void* key = symbol_insert_key(table, &name, sizeof(char*), sizeof(table_data));
         int offset;
         result = 1;
         table_cast(key)->type = INT;
         offset = table_cast(key)->offset = result * 4;
  *]
  { ',' DECL:name
        [* if (symbol_lookup_key(table, &name, scp)) {
               text_message(TEXT_ERROR, "redeclaration of '%s'\n", name);
           } else {
                   key = symbol_insert_key(table, &name, sizeof(char*), sizeof(table_data));
                   result += 1;
                   table_cast(key)->type = INT;
                   offset  = table_cast(key)->offset = result * 4;
           }
    *]
  }.
```

Parameters are declared as a comma separated list. Each new parameter which is declared is immediately added to the symbol table, to ensure the following parameters aren’t redeclarations of previous parameter names. Each parameter is stored in the symbol table with its offset from the frame pointer, and the number of parameters is counted and returned to FUNC_DECL. Knowing the number of parameters is important because the number of parameters a function has affects the offset of the local variables from the frame pointer, and also because when the function is being called, there needs to be a way to verify the correct number of arguments are being used.

Final word count: 46911

```
SCOPE(scp size):declarations
[*      locals = declarations;
        size += (4 * declarations) + 4;
        table_cast(key)->size = size;
        if (scp != main_scope) {
                fprintf(fp, "\treturn\n");
    } else {
        fprintf(fp, "\tterminate\n");
    }
    fprintf(ft, "%s %d %d\n", name, FUNC, locals);
*].
```

Opening (and subsequently closing) the scope is the final part of a function declaration.

```
SCOPE(scp:void* offset:int):int ::=
[* result = 0; *]
'{' { DECL:name VAR_DECL(name offset)
      [* offset += 4;
         result += 1;
      *]
 |    ( ID:name ( FUNC_CALL(name 0) | ASSIGN(name) ) | RETURN | PRINT | BREAK | CONTINUE ) ';'
 |    COND(offset):declarations
      [* result += declarations; *]
 |    WHILE(offset):declarations
      [* result += declarations; *]
 |    [* void *new_scope = symbol_new_scope(table, NULL);
         link_scope(new_scope);
      *]
      SCOPE(new_scope offset):declarations
      [* result += declarations; *]
    }
'}'
 [* symbol_unlink_scope(table);
    unlink_scope();
    symbol_set_scope(table, *this_scope());
 *].
```

Within a scope most things are possible besides declaring functions. The result of a scope is used to keep track of the number of local variable declarations made. Most of the available features have already been presented in previous versions of the grammar, so I don’t want to go into any further detail on those, and assignment works by traversing the scope chain to find the most recent declaration with a certain name, the same as it did in the scope example grammar.

```
int offset = table_cast(key)->offset;
if (offset) {
   fprintf(fp, "\t$fp(%d) = %s\n", offset, val);
} else {
    fprintf(fp, "\t%s = %s\n", name, val);
}
```

The offset associated with variables however is used to determine whether or not whatever variable being assigned to is a local variable or a global variable.

Final word count: 46911

```
 |    COND(offset):declarations
      [* result += declarations; *]
 |    WHILE(offset):declarations
      [* result += declarations; *]
 |    [* void *new_scope = symbol_new_scope(table, NULL);
         link_scope(new_scope);
      *]
      SCOPE(new_scope offset):declarations
      [* result += declarations; *]
```

Also, anything which opens up a new scope has the current offset handed over to it, and returns the number of declarations made within its own scope, so the outer scope can be updated with the information.

```
RETURN ::= 'return' [ EXPR:val
                      [* if (*this_scope() != main_scope) {
                             fprintf(fp, "\treturn %s\n", val);
                         } else {
                             fprintf(fp, "\tterminate\n");
                         }
                      *]
                     ].
```

Calling return either writes the ‘return’ instructions to the IR. Return can be optionally followed by an expression, which will be the return value of the function. If return is called from the main function, the ‘terminate’ instruction is written to the IR instead.

```
 [* symbol_unlink_scope(table);
    unlink_scope();
    symbol_set_scope(table, *this_scope());
 *].
```

For any scope, when it is closed, it is unlinked both from the rdp internal scope chain, and my own. I then manually set the rdp internal scope chain to match my own. This was a work around I decided on after experiencing issues using only the internal rdp scope chain.

```
SCOPE(scp size):declarations
[*      locals = declarations;
        size += (4 * declarations) + 4;
        table_cast(key)->size = size;
        if (scp != main_scope) {
                fprintf(fp, "\treturn\n");
    } else {
        fprintf(fp, "\tterminate\n");
    }
    fprintf(ft, "%s %d %d\n", name, FUNC, locals);
*].
```

When the function’s scope is closed, the final size of the stack frame is calculated using the number of locally declared variables, and subsequently stored in the symbol table. An implicit ‘return’ is added at the end of each function, or an implicit ‘terminate’ in the case of the main function.

Final word count: 46911

```
fprintf(ft, "%s %d %d\n", name, FUNC, locals);
```

The name, type and number of local variables are written to the symbol table file, as this is all information which is necessary during the code generation stage.

Function Calls

```
FUNC_CALL(name:char* assign:int):char* ::=
[*  void **chain = this_scope();
    if (*chain == global_scope) {
        text_message(TEXT_ERROR, "initializer element '%s' is not constant\n", name);
    }
    void *key;
    args = 0;
    while (chain) {
        if ((key = symbol_lookup_key(table, &name, *chain))) {
            if (table_cast(key)->type != FUNC) {
                text_message(TEXT_ERROR, "'%s' is not a function or function pointer\n", name);
            }
            break;
        }
        chain = prev_scope(chain);
    }
    if (!key) {
        text_message(TEXT_ERROR, "'%s' undeclared\n", name);
    } else {
        result = mem_malloc(7 + strlen(name));
        sprintf(result, "call %s", name);
        fprintf(fp, "\tpush $fp\n");
        fprintf(fp, "\tpush 4\n");
   }
*]
'(' [ ARGS:args ]
    [* if (key) {
           if (table_cast(key)->params != args) {
               text_message(TEXT_ERROR, "parameter mismatch in function '%s'\n", name);
           }
      }
   *]
 ')'[* if (!assign) fprintf(fp, "\tcall %s\n", name); *].

ARGS:int ::= EXPR:val
             [* result = 1;
                fprintf(fp, "\tpush %s %d\n", val, result * 4);
             *]
             { ',' EXPR:val
               [* result += 1;
                  fprintf(fp, "\tpush %s %d\n", val, result * 4);
               *]
             }.
```

I have shown the full production rule here once more to provide a full picture to refer to from the snippets I use to describe the process of the semantics.

Final word count: 46911

```
FUNC_CALL(name:char* assign:int):char* ::=
[*  void **chain = this_scope();
    if (*chain == global_scope) {
        text_message(TEXT_ERROR, "initializer element '%s' is not constant\n", name);
    }
```

When a function is called a check is first made to ensure the function call is not being made within the global scope, as this should not be possible.

```
while (chain) {
    if ((key = symbol_lookup_key(table, &name, *chain))) {
        if (table_cast(key)->type != FUNC) {
            text_message(TEXT_ERROR, "'%s' is not a function or function pointer\n", name);
        }
        break;
    }
    chain = prev_scope(chain);
}
```

A check is then made to verify that what is being called is in fact a function and not simply an integer. The ‘type’ field from the symbol table is used to verify this fact.

```
if (!key) {
    text_message(TEXT_ERROR, "'%s' undeclared\n", name);
} else {
    result = mem_malloc(7 + strlen(name));
    sprintf(result, "call %s", name);
    fprintf(fp, "\tpush $fp\n");
    fprintf(fp, "\tpush 4\n");
}
```

If no matching identifier is found anywhere on the scope chain, the function is deemed undeclared and an error is returned to reflect that as normal. If the function call is successful however, the ‘call’ instruction followed by the function’s name is stored in ‘result’. This is to account for when a function all is used as part of an expression

Function calls can either be made simply as a call to a subroutine with no expectation to return a value to anything.

```
SCOPE(scp:void* offset:int):int ::=
[* result = 0; *]
'{' { DECL:name VAR_DECL(name offset)
      [* offset += 4;
         result += 1;
      *]
 |    ( ID:name ( FUNC_CALL(name 0) | ASSIGN(name) ) | RETURN | PRINT | BREAK | CONTINUE ) ';'
```

Final word count: 46911

Or a function call can be used as an operand of an expression, in which case the ‘call’ will be assigned to a temporary variable

```
PRIMITIVE:char*   ::= INTEGER:val (* integer literals *)
                     [* result = (char*) mem_malloc(12);
                        sprintf(result, "%li", val);
                     *]
                   | [* int func = 0; *]
                     ID:name [ FUNC_CALL(name 1):call
                               [* func = 1;
                                  result = call;
                               *]
                             ]
```

Because of the restrictions of LL1 grammars, I had to implement function calls used in expressions as an extension of variables as primitive types (since they both start with an identifier, this would cause a FIRST FIRST conflict if I attempted to implement function calls as an alternate of PRIMITIVE).

```
FUNC_CALL(name:char* assign:int):char*
```

The attribute ‘assign’ is a flag, which is set to 1 if the function call is being used in an expression, and 0 if not.

```
'(' [ ARGS:args ]
    [* if (key) {
           if (table_cast(key)->params != args) {
               text_message(TEXT_ERROR, "parameter mismatch in function '%s'\n", name);
           }
      }
   *]
 ')'[* if (!assign) fprintf(fp, "\tcall %s\n", name); *].
```

This flag is checked at the end of FUNC_CALL. If it is set to 0, the program knows to write a call instruction to the TAC file, as the call will not have already been made as part of an expression.

```
[* int func = 0; *]
ID:name [ FUNC_CALL(name 1):call
          [* func = 1;
             result = call;
          *]
        ]
```

If a function call is made as part of an expression, the ‘call’ instruction returned by FUNC_CALL is stored as the result of PRIMITIVE.

Final word count: 46911

```
UNARY:char* ::= PRIMITIVE:val1
                [* result = new_temp();
                   fprintf(fp, "\t%s = %s\n", result, val1);
                  val1 = result;
                 *]
              | '(' EXPR:val1 [* result = val1; *] ')'.
```

This then propagates up to UNARY, where the function call is written to the TAC file as an assignment to a temporary variable, the same as with a variable. This is what allows the return value of a function to be assigned somewhere.

```
'(' [ ARGS:args ]
    [* if (key) {
           if (table_cast(key)->params != args) {
               text_message(TEXT_ERROR, "parameter mismatch in function '%s'\n", name);
           }
      }
   *]
 ')'[* if (!assign) fprintf(fp, "\tcall %s\n", name); *].

ARGS:int ::= EXPR:val
             [* result = 1;
                fprintf(fp, "\tpush %s %d\n", val, result * 4);
             *]
             { ',' EXPR:val
               [* result += 1;
                  fprintf(fp, "\tpush %s %d\n", val, result * 4);
               *]
             }.
```

Arguments are written as a comma separated list. The number of arguments which are made is stored in the result of ARGS. This has two responsibilities. The first is to keep track of each argument's offset from the frame pointer as the result is incremented with each additional argument. The second is once all arguments have been made, the result is handed back to FUNC_CALL which then performs a check to ensure the number of arguments used in the function call matches the number of parameters in the function's declaration. If this is not the fact, a parameter mismatch error is displayed.

Final word count: 46911

Code Generation for Functions

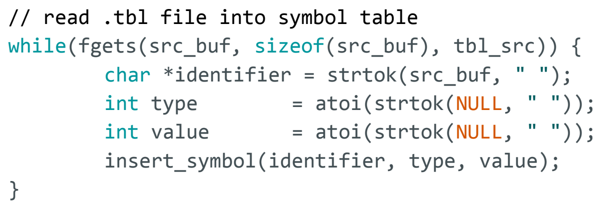
Several additions had to be made to the code generator in order to implement code generation for functions.

```
typedef struct _symbol_node {
        symbol_t symbol;
        struct _symbol_node *next;
} symbol_node;

symbol_node **symbol_table;
int size, capacity;
```

Because the symbol table file was now holding more information than just the names of global variables, I added a simple symbol table implementation to the code generator, to avoid additional file reads when the data had to be referenced.

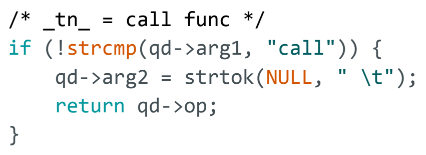
```
typedef struct _symbol {
        char *identifier;
        int type;
        int value;
} symbol_t;

void insert_symbol(char *token, int type, int value);

symbol_t *get_symbol(char *token);
```

The symbol table has been implemented as a hash map, and insert_symbol and get_symbol are functions for storing and retrieving data in and from the table.



```
// read .tbl file into symbol table
while(fgets(src_buf, sizeof(src_buf), tbl_src)) {
        char *identifier = strtok(src_buf, " ");
        int type         = atoi(strtok(NULL, " "));
        int value        = atoi(strtok(NULL, " "));
        insert_symbol(identifier, type, value);
}
```

The table is populated with the contents of the symbol table file before the TAC file is read and used to generate the code. The ‘type’ refers to whether the symbol is a variable or a function, and the value in the case of functions refers to the size of the stack frame, so the code generator knows how much space to allocate when the function is called.



```
/* _tn_ = call func */
if (!strcmp(qd->arg1, "call")) {
    qd->arg2 = strtok(NULL, " \t");
    return qd->op;
}
```

The “quad_fill” function now has a case which detects when a function call is the target of an assignment, and modifies the quad accordingly so the ‘assign’ code generation function can make the necessary adjustments.

Final word count: 46911

```
switch (keyword) {
    case 1: /* conditional jump */
         for (int i = 0; i < 4; i++) {
             pch = strtok(NULL, " \t");
         }
         qd->result = pch;
         break;
    case 6: /* push */
         qd->arg2 = strtok(NULL, " \t");
        break;
    default:
        return NULL;
}
```

Additionally, “quad_fill” now has a case for detecting when ‘push; is used. The list of keywords and operations in tac.c have also had the new operations and keywords added (call, push, terminate, return) in order for the program to be able to parse the TAC file properly.

```
void mips_header(FILE *dst, FILE *src) {
        char buf[32];
        fprintf(dst, "\t.data\n");
        fprintf(dst, "newline:\t .asciiz \"\\n\"\n");
        while (fgets(buf, sizeof buf, src)) {
                char *token = strtok(buf, " ");
                int type    = atoi(strtok(NULL, " "));
                if (type == INT) {
                        fprintf(dst, "%s:\t.word 0x0\n", token);
                }
        }
        fprintf(dst, "\t.text\n");

        // sub routine for storing registers and fp
        fprintf(dst, "save_frame:\n");
        fprintf(dst, "\taddi $sp, $sp, -44\n");
        for (int i = 0; i < 10; i += 1) {
                fprintf(dst, "\tsw $t%i, %i($sp)\n", i, 44 - (i * 4));
        }
        fprintf(dst, "\tsw $fp, 4($sp)\n");
        fprintf(dst, "\tjr $ra\n");

        // sub routine for restoring registers and fp
        fprintf(dst, "load_frame:\n");
        for (int i = 0; i < 10; i += 1) {
                fprintf(dst, "\tlw $t%i, %i($sp)\n", i, 44 - (i * 4));
        }
        fprintf(dst, "\tlw $fp, 4($sp)\n");
        fprintf(dst, "\taddi $sp, $sp, 44\n");
        fprintf(dst, "\tjr $ra\n");
}
```

The mips header now includes two runtimes functions “save_frame” and “load_frame”. “save_frame” is a function which pushes the contents of the registers and the frame pointer’s current position to the stack. “load_frame” does the opposite by restoring the saved values to their respective registers, and restoring the frame pointer’s previous position.

Final word count: 46911

These functions are both used every time a function call is made (save_frame when a function is called, and load_frame when one returns). Originally this code was being generated where every function call was made, however the code is always the same regardless of the nature of the function, so I included these runtime functions in order to significantly reduce program sizes.

Assignment of values to local variables and assignment of return values from functions is handled by the same assign function which handles all non binary assignment operations.

```
void mips_assign(quad *qd, FILE *dst) {
        int val_reg, adr_reg, offset;
        if (is_temp(qd->result)) {
                if (is_int(qd->arg1)) { /* $tn = INTEGER */
                        val_reg = get_val_reg(qd->result);
                        fprintf(dst, "\tli $t%i, %s\n", val_reg, qd->arg1); // load immediates
                } else if (!strcmp(qd->arg1, "call")) { /* $tn = call func */
                        val_reg = get_val_reg(qd->result);
                        qd->arg1 = qd->arg2;
                        mips_call(qd, dst);
                        fprintf(dst, "\tmove $t%i, $v1\n", val_reg); // assign return value
                } else if (is_fp(qd->arg1)){ /* $tn = $fp(n) */
                        val_reg = get_val_reg(qd->result);
                        offset  = get_fp_offset(qd->arg1);
                        fprintf(dst, "\tlw $t%i, -%i($fp)\n", val_reg, offset);
                } else { /* _tn_ = x */
                        val_reg = get_val_reg(qd->result);
                        adr_reg = get_adr_reg(qd->result);
                        fprintf(dst, "\tla $a%i, %s\n", adr_reg, qd->arg1); // load address
                        fprintf(dst, "\tlw $t%i, ($a%i)\n", val_reg, adr_reg); // load word
                }
        } else if (is_fp(qd->result)) { /* $fp(n) = $tn */
                val_reg = get_val_reg(qd->arg1);
                offset = get_fp_offset(qd->result);
                fprintf(dst, "\tsw $t%i, -%i($fp)\n", val_reg, offset);
        } else { /* x = _tn_ */
                val_reg = get_val_reg(qd->arg1);
                adr_reg = get_adr_reg(qd->arg1);
                fprintf(dst, "\tla $a%i, %s\n", adr_reg, qd->result); // load address
                fprintf(dst, "\tsw $t%i, ($a%i)\n", val_reg, adr_reg); // load word
        }
}
```

This is by far the largest code generating function as it has to handle the most different cases. A combination of the “load address”, “load word” and “store word” instructions paired with the use of the register allocation functions is sufficient for most cases. The main difference in most cases is the target location, where local variables are pushed to the stack using the offsets provided by the TAC instructions. Return values of functions are always stored in register $v1 before returning from the function, and if they are assigned somewhere, the value is subsequently retrieved from $v1 into an allocated general use register using the move instruction.

Final word count: 46911

```
void mips_return(quad *qd, FILE *dst) {
        // load return address from stack
        fprintf(dst, "\tlw $ra, 0($fp)\n");
        // either return the desired value or return 0
        // if no value was given
        if (qd->arg1) {
                int val_reg = get_val_reg(qd->arg1);
                fprintf(dst, "\tmove $v1, $t%i\n", val_reg);
        } else {
                fprintf(dst, "\tli $v1, 0\n");
        }
        // pop the stack frame and return
        fprintf(dst, "\tmove $sp, $fp\n");
        fprintf(dst, "\tjr $ra\n");
}
```

The return function simply retrieves the return address from the stack, and either assigns the given return value to $v1, or assigns 0 by default. This is in case no return value is given, but the function's return value is still assigned somewhere. The stack frame is then popped and the return instruction is used.

Push and call I have already explained in quite some detail in the previous section, and I don't feel there are not any particular implementation details worth mentioning here.

Final word count: 46911

Final Features

Having gotten functions working, I had surpassed my expectations for how this project would turn out, however there was still plenty of time to add new features, and some of the features which I needed to reach my original goal of being able to compile a prime number generation program were still missing. Arrays were also not currently part of the language and would be a significant language feature which wouldn’t be particularly difficult to implement.

Extensions to Expressions

```
SUM:char* ::=
PROD:val1 [* result = val1; *]
{ '+' PROD:val2
  [* result = new_temp();
         fprintf(fp, "\t%s = %s + %s\n", result, val1, val2);
         val1 = result;
  *]
| '-' PROD:val2 (* subtraction *)
  [* result = new_temp();
     fprintf(fp, "\t%s = %s - %s\n", result, val1, val2);
     val1 = result;
  *]
}.

PROD:char* ::=
UNARY:val1 [* result = val1; *]
{ '*' UNARY:val2
 [* result = new_temp();
    fprintf(fp, "\t%s = %s * %s\n", result, val1, val2);
    val1 = result;
 *]
|
 '/' UNARY:val2
 [* result = new_temp();
        fprintf(fp, "\t%s = %s / %s\n", result, val1, val2);
        val1 = result;
 *]
|
 '%' UNARY:val2
 [* result = new_temp();
    fprintf(fp, "\t%s = %s %% %s\n", result, val1, val2);
    val1 = result;
 *]
}.
```

This honestly could have been added a lot earlier, but I avoided doing it because I assumed there would be complications generating the code, which in the end was not the case. Extending the capabilities of expression with multiplication, division and modulus was simple because the semantics were identical to those of addition and subtraction, except the operators were different. Mul, div and mod are part of the same production rule because they have the same operator priority. PROD comes in the expression chain after SUM because the set of operators in PROD have a higher priority then those in SUM, and because this is a recursive descent parser, whatever is in PROD is evaluated first.

Final word count: 46911

```
EXPR:char*  ::= BOOL:result.

BOOL:char* ::=
SUM:val1 [* result = val1; *]
{ '==' SUM:val2
  [* result = new_temp();
     fprintf(fp, "\t%s = %s == %s\n", result, val1, val2);
     val1 = result;
  *]
| '>=' SUM:val2
  [* result = new_temp();
     fprintf(fp, "\t%s = %s >= %s\n", result, val1, val2);
     val1 = result;
  *]
| '>' SUM:val2
  [* result = new_temp();
         fprintf(fp, "\t%s = %s > %s\n", result, val1, val2);
         val1 = result;
  *]
| '<=' SUM:val2
  [* result = new_temp();
         fprintf(fp, "\t%s = %s <= %s\n", result, val1, val2);
         val1 = result;
  *]
| '<' SUM:val2
  [* result = new_temp();
         fprintf(fp, "\t%s = %s < %s\n", result, val1, val2);
         val1 = result;
  *]
| '!=' SUM:val2
  [* result = new_temp();
         fprintf(fp, "\t%s = %s != %s\n", result, val1, val2);
         val1 = result;
  *]
}.
```

Boolean operators are evaluated last so if a boolean operator is ever used in an expression, the result of the expression will be 1 or 0 (unless brackets are used).

Because mips has instructions for multiplication, division, modulus, and all the boolean operations above, writing the code generation functions was trivial, as they were effectively the same as the existing functions for addition and subtractions, only with different instructions.

```
UNARY:char* ::=
PRIMITIVE:val1
[* result = new_temp();
   fprintf(fp, "\t%s = %s\n", result, val1);
   val1 = result;
*]
| '(' EXPR:val1 [* result = val1; *] ')'.
```

Bracketed expressions could also be added very simply, and required no change to the code generator, as they generated nothing unique, but rather just generated more binary assignment instructions.

Final word count: 46911

Reimplementing Global Variables

Originally global variables had their values assigned at runtime, however after implementing functions, the code which assigned global variables their values had now become unreachable, since this code wasn’t inside the main function or any subroutine. Because global variables were statically allocated, they could actually have their value assigned before the program began execution. This was a point I raised previously, and was a potential optimisation so since this feature needed fixing anyway, I could kill two birds with one stone.

The current problem was that global variables were just regular variable declarations which could be performed outside of any scope. I saw two potential solutions to rectify the issue. Since I was the global scope anyway, I could have the parser detect the current scope and if it was in the global scope, run different logic specifically for handling global variables. The other option was to write a separate set of production rules for handling global variables.

I felt writing a seperate set of production rules was the cleaner solution in this case, since having more production rules would make the grammar more flexible, and having additional checks in the existing declaration semantics would reduce their efficiency.

Since the code which declared the global variables was unreachable, there was no longer a necessity for global variables to produce any code. Instead their values would be calculated by the parser, and written to the symbol table file so the code generator could retrieve them later and assign them directly in the targets header, instead of zero initializing all the global variables as I had been doing up until this point.

```
TRNS_UNIT ::=
[* fp = fopen("func.tac", "w");
   ft = fopen("func.tbl", "w");
   link_scope(symbol_get_scope(table));
   global_scope = *this_scope();
*]
{ DECL:name ( GLB_VAR(name) | GLB_ARR(name) | FUNC_DECL(name) ) }
[* fclose(fp);
   fclose(ft);
   if (!main_scope) {
       text_message(TEXT_ERROR, "no main function has been declared\n");
   }
*].
```

This became the new translation unit rule. All declarations within the global scope were now specifically global declarations (and function declarations, which are also only possible in the global scope).

```
GLB_VAR(name:char*) ::=
[*  void *key;
        if (symbol_lookup_key(table, &name, global_scope)) {
                text_message(TEXT_ERROR, "redeclaration of '%s'\n", name);
        }
        val = 0;
*]
```

Final word count: 46911

```
[ '=' GLB_EXPR(0):val ] ';'
[* fprintf(ft, "%s %d %d\n", name, INT, val);
   key = symbol_insert_key(table, &name, sizeof(char*), sizeof(table_data));
   table_cast(key)->id     = name;
   table_cast(key)->type   = INT;
   table_cast(key)->offset = 0;
   table_cast(key)->value  = val;
*].
```

Initially, declaring global variables was still largely the same as before. They were still checked for redeclaration in the same way, and still inserted into the symbol table in the way, however now they would have their values calculated by the parser and stored in the symbol table, as well as being written to the symbol table file. The reason why the value needed to be stored is because they would need to be retrievable in the case that another variable would have its value assigned using an existing global variable, like the following example:

```
int x = 5;
int y = x + 2;
int main() {
    int z = x + y;
}
```

Because the value of variables needed to be calculated by the parser, the global expression would need to be capable of performing arithmetic during the parser's runtime, rather than emitting code which would translate to the relevant calculations later. This was a feature I'd implemented in older versions of the grammar however, so this was a straightforward task.

```
GLB_EXPR(array:int):int  ::= GLB_BOOL(array):result.

GLB_BOOL(array:int):int  ::=
        GLB_SUM(array):result
        { '==' GLB_SUM(array):val [* result = result == val; *]
    | '!=' GLB_SUM(array):val [* result = result != val; *]
    }.

GLB_SUM(array:int):int   ::=
        GLB_PROD(array):result
        { '+' GLB_PROD(array):val [* result += val; *]
    | '-' GLB_PROD(array):val [* result -= val; *]
    }.

GLB_PROD(array:int):int  ::=
        GLB_PRIM(array):result
        { '*' GLB_PRIM(array):val [* result *= val; *]
        | '/' GLB_PRIM(array):val [* result /= val; *]
    | '%' GLB_PRIM(array):val [* result %= val; *]
    }.
```

The expression semantics above would effectively act as a calculator for the parser, allowing it to evaluate expressions during runtime.

```
GLB_PRIM(array:int):int ::=
        INTEGER:result
```

Final word count: 46911

```
|   ID:name [*  void *key;
                if (!array) {
                    if (!(key = symbol_lookup_key(table, &name, global_scope))) {
                        text_message(TEXT_ERROR, "'%s' undeclared\n", name);
                    } else {
                        result = table_cast(key)->value;
                    }
                } else {
                    text_message(TEXT_ERROR, "ecc does not support variable length arrays\n", name);
                }
            *]
|   '(' GLB_EXPR(array):result ')'.
```

The implementation of primitive types would either return values to expression from integer literals, or from retrieving the values of existing global variables from the symbol table.

The code generators symbol table already stored values associated with symbols, since this feature had already been added so functions could be added to the table with their stack frame sizes. The exact same attribute of the symbols could be used with global variables, but instead to store their assigned value. It was also already possible to make the distinction between symbols which were function names and symbols which were variables, because the symbol's data type was also being stored for each symbol. This meant the code could be generated with only a small modification to the existing program.

Arrays

Out of the set of final features, the addition of arrays was by far the largest. The first order of business was extending the types enums to account for arrays.

```
#ifndef TYPE_H
#define TYPE_H

enum type {
        INT = 1,
        FUNC,
        ARRAY
};

#endif /* TYPE_H */
```

There needed to be a separate method of declaration of arrays within the global scope and within a local scope. There would be additional complications to declaring arrays in the global scope global arrays would be statically allocated. This means that the individual elements of the array would have to be stored initially so they could be retrieved during the code generation stage. I'll talk more on this in a moment though, since local array declaration is a lot simpler, whilst using a lot of the same ideas.

Final word count: 46911

In order to make array declarations C like, I had to include the following:

```
int arr[5];
int arr[5] = {3, 5, 7};
```

It needed to be possible to declare arrays by setting the number of elements in the array. In this case, an initializer list to assign the elements also needed to be an option.

```
int arr[] = {3, 5, 7};
int arr[]; // invalid
int arr[] = {};
```

It also needed to be possible to have the size of the array determined by the number of elements in the initializer list, however given neither a size nor an initializer list needed to be invalid. In the event that both no size was given and an empty initializer list was given, it would be valid and a warning would be issued explaining that a single element had been assigned implicitly.

```
int arr[3] = {1, 2, 3, 4, 5}; // issues a warning
```

It was also reasonable to allow the number of elements in an initializer list to to exceed the size of the array, however this should also issue a warning as the array should not be resized to accommodate the excess elements.

```
int arr[-1] // invalid
```

I also needed to disallow the declarations of arrays with a negative number of elements.

```
Arr[2] = 7;
x = 4 + arr[2];
```

It needed to be possible to assign values to individual elements of arrays and to be able to use the individual elements in expressions.

Since C allows pointer arithmetic it should also be possible to use the array itself in expressions, however due to the fact I had not added support for pointers, this would not be a feature I would include.

Additionally, due to how complex the problem ended up being, I could not be allowing arrays to be arguments of functions. I had ideas on how I would do this but there wasn't any elegant way to implement them due to problems posed by the extra level of indirection when retrieving values from arrays, compared to when retrieving values from variables.

Final word count: 46911

```
ARR_DECL(name:char* offset:int):int ::=
[*  void *key;
        void *scp = *this_scope();
        if (symbol_lookup_key(table, &name, scp)) {
                text_message(TEXT_ERROR, "redeclaration of '%s'\n", name);
        }
        int sized = 0;
        int count = 0;
*]
'['
    [
          GLB_EXPR(1):size
          [* sized = 1;
                 if (size < 0) {
                 text_message(TEXT_ERROR, "size of array '%s' is negative\n", name);
             }
          *]
        ]
']'
```

The first order of business was making it possible for arrays to be sized. To keep track of whether or not an array had been sized, a flag `‘sized’`, is initially set to 0, and changed to 1 upon the array being sized.

```
int arr[5 + 3];
```

The size of an array can be the result of an expression, and the result of the expression needs to be known during compile time (in case the result is negative for example, then a compile time error must be issued). Thankfully, because of my implementation of global expressions, I already had logic which effectively provided the compiler with an internal calculator.

```
[
        '=' '{' [ EXPR:val
                          [* fprintf(fp, "\t$fp(%d) = %s\n", offset + (4 * count), val);
                                 count += 1;
                          *]
                          { ',' EXPR:val
                            [* fprintf(fp, "\t$fp(%d) = %s\n", offset + (4 * count), val);
                               count += 1;
                            *]
                          }
                        ]
                '}'
        [* if (sized && (count > size)) {
               text_message(TEXT_WARNING, "excess elements in array initializer\n");
           }
        *]
]
```

The next step was adding the initializer list. Each element in the list immediately generates the TAC which translates to the element being pushed onto the stack, the instruction for which is the same as any other stack allocated variable. The number of elements in the list is kept track of, and if a size for the array has been set and the number of elements in the initializer list exceeds that size, then a warning is issued. If the array has not been sized however, then this does not happen, as the array will be sized according to the number of elements in the initializer list.

```
[*  key = symbol_insert_key(table, &name, sizeof(char*), sizeof(table_data));
    if (!(count + sized)) {
            text_message(TEXT_WARNING, "array '%s' assumed to have one element\n", name);
            table_cast(key)->size = 4;
            result = 1;
    } else if (sized) {
        table_cast(key)->size = 4 * size;
        result = size;
    } else {
        table_cast(key)->size = 4 * count;
        result = count;
    }
    table_cast(key)->id = name;
    table_cast(key)->type = ARRAY;
    table_cast(key)->offset = offset;
*] ';'.
```

When placing the array into the symbol table, a set of checks are made to determine how the array had been sized, so the correct size for the array can be stored in the table.

If both the count and the ‘sized’ flag are 0, the array is assumed to have a single element, and the size of the array is set to 4 bytes (the size of a single integer).

If the array had been sized, this takes precedence over the number of elements in an initializer list, if one had been used. Therefore the size of the array is 4 times the size given.

If no size has been set and an initializer list has been used then the size of the array is 4 times the number of elements.

It is necessary that the size of the array is known because it is used to calculate the amount of space needed to store the local variables in a function’s stack frame.

Final word count: 46911

```
GLB_ARR(name:char*) ::=
[* void *key;
        if (symbol_lookup_key(table, &name, global_scope)) {
                text_message(TEXT_ERROR, "redeclaration of '%s'\n", name);
        }
        int sized = 0;
        int count = 0;
*]
'['
    [
          GLB_EXPR(1):size
          [* sized = 1;
                 if (size < 0) {
                 text_message(TEXT_ERROR, "size of array '%s' is negative\n", name);
             }
          *]
        ]
']'
```

Moving onto global arrays, size allocation is handled in the exact same way as before.

```
[
        '=' '{' [ GLB_EXPR(0):val
                    [*  int *temp = malloc(sizeof(int));
                        *temp = val;
                        enqueue(&array_queue, temp);
                        count += 1;
                    *] { ',' GLB_EXPR(0):val
                         [*  temp = malloc(sizeof(int));
                             *temp = val;
                             enqueue(&array_queue, temp);
                             count += 1;
                         *]
                       }
                    ]
                 '}'
        [* if (sized && (count > size)) {
               text_message(TEXT_WARNING, "excess elements in array initializer\n");
           }
        *]
]
```

The initializer list had to be implemented differently however, due to the fact that global arrays can not have their elements assigned during run time. I needed a method of storing array elements temporarily so I wrote a queue specifically for this purpose. The elements themselves need to be calculated during compile time so global expressions are used to do that. The number of elements is still counted and the same warning is still issued if the count exceeds the allocated size.

Final word count: 46911

```
[*  key = symbol_insert_key(table, &name, sizeof(char*), sizeof(table_data));
    if (!(count + sized)) {
            text_message(TEXT_WARNING, "array '%s' assumed to have one element\n", name);
            table_cast(key)->size = 4;
    } else if (sized) {
        table_cast(key)->size = 4 * size;
    } else {
        table_cast(key)->size = 4 * count;
    }
    table_cast(key)->id = name;
    table_cast(key)->type = ARRAY;
    table_cast(key)->offset = 0;
    fprintf(ft, "%s %i %i", name, ARRAY, table_cast(key)->size);
    while (count) {
        int value = **((int**)dequeue(&array_queue));
        fprintf(ft, " %i", value);
        count--;
    }
    fprintf(ft, "\n");
*]
';'.
```

After the same set of checks as before are run to determine the size of the array, the identifier and data type (ARRAY) is written to the symbol table file, along with all the elements of the array. This is done by removing is element from the queue in sequence until it becomes empty. This will later be used by the code generator to create the array in the program.

```
ASSIGN(name:char*) ::=
[* int array = 0; *]
[ '[' EXPR:index ']' [* array = 1; *] ]
```

A flag is set to indicate when a value is being assigned to an element of an array. A distinction has to be made from assigning a value to a variable because ASSIGN contains the semantics for both.

```
if (offset) {
    fprintf(fp, "\t$fp(%d)[%s] = %s\n", offset, index, val); // global arrays
} else {
    fprintf(fp, "\t%s[%s] = %s\n", name, index, val); // local arrays
}
```

Assignment of array elements differs depending on if the array is local or global, however compared to their regular variable counterparts, they're not too dissimilar:

`var = $t1`	`arr[2] = $t1`
`$fp(4) = $t1`	`$fp(4)[2] = $t1`

On the left we have TAC representing assignments to local and global variables, and on the right we have the same but for array elements. In the case of the local array, the stack offset '4' points to the start of the array, and then the index '2' will be used to increase the offset in order to find the particular element.

Final word count: 46911

For global arrays the index is used in the same way, except the array is statically allocated.

```
PRIMITIVE:char* ::=
  INTEGER:val (* integer literals *)
  [* result = (char*) mem_malloc(12);
         sprintf(result, "%li", val);
  *]
| [* int func = 0, array = 0; *]
  ID:name
  [ FUNC_CALL(name 1):call
        [* func = 1;
           result = call;
        *]
  | '[' EXPR:index
    ']'
    [* array = 1; *]
  ]
```

If an array element is used in an expression, a flag is set indicating the primitive value is that of an array element, the same as when a value is being assigned to an array.

```
if (offset) {
    result = (char*) mem_malloc(30);
    sprintf(result, "$fp(%d)[%s]", offset, index);
} else {
    result = (char*) mem_malloc(strlen(name) + strlen(index) + 2);
    sprintf(result, "%s[%s]", name, index);
}
```

The same approach is used here as with assignment to represent array elements in the intermediate form.

```
'['
    [
          GLB_EXPR(1):size
          [* sized = 1;
                 if (size < 0) {
                 text_message(TEXT_ERROR, "size of array '%s' is negative\n", name);
             }
          *]
        ]
']'
```

```
GLB_PRIM(array:int):int
```

One last thing to note is the use of the ‘array’ flag when using global expressions to determine array sizes. This is to perform a check to ensure that only integer literals are used to assign sizes to arrays.

```
Int arr[x + y]; // invalid
Int arr[func()]; //invalid
```

Final word count: 46911

```
GLB_PRIM(array:int):int ::=
        INTEGER:result
|   ID:name [*  void *key;
                if (!array) {
                    if (!(key = symbol_lookup_key(table, &name, global_scope))) {
                        text_message(TEXT_ERROR, "'%s' undeclared\n", name);
                    } else {
                        result = table_cast(key)->value;
                    }
                } else {
                    text_message(TEXT_ERROR, "ecc does not support variable length arrays\n", name);
                }
            *]
|   '(' GLB_EXPR(array):result ')'.
```

This is done to effectively disable variables length arrays (VLA), and if this is attempted an error is returned. A variable length array is an array whose length is determined at run time. Adding support for VLAs would be a very complicated task which far exceeds the scope of this project. Additionally, VLAs were relegated to being a conditional feature in C11[60].

```
int func() {
    int x = 5;
    int arr[x];
}
```

In the example above we have a variable ‘x’ and an array ‘arr’ which are both local to the function ‘func’. For my implementation of functions, the space allocated for local variables is calculated at compile time, however ‘x’ does not have its value assigned until run time. This means the size of ‘arr’ can’t be known until the program is running, however the size of arr is required to determine the size of func’s stack frame.

Because global variables have their values assigned during runtime, it was actually possible to permit global arrays to have variable lengths, however I decided against having this as it would seem inconsistent, so I’ve intentionally disabled the option for both local and global arrays.

Final word count: 46911

Testing

For Testing I wrote a set of test programs to verify all the features are working correctly.

The test programs and what they refiy are as follows:

fib - Calculates the first 10 Fibonacci numbers and prints them to the console. I've written this program to demonstrate that the compiler is capable of handling recursive function calls, and the use of functions in expressions. This program also demonstrates the use of local variable declarations, and how variables can be separated by scope (the variable name 'x' is used both for a global variable and a parameter of the fib function).

prime - This program "attempts" to calculate every prime number up to 2^32 - 1 (I've only actually run it up to about 100,000). This program was mainly written to test whether or not the stack was being manipulated correctly and that programs run for a long period of time would not simply crash due to some unforeseen error (such as the stack blowing due to poor deallocation logic). It also demonstrates nested loops, use of various operators and local variables.

sum - Calculates the of sum integers from one to one hundred. This demonstrates the use of both local and global variables and how programs can use duplicate identifiers, provided they are separated by scope. Additionally, global variables now have their initial values assigned at compile time, An issue which initially occurred after adding functions.

bubs - an example program demonstrating array usage. This program demonstrates:

- An array declared as a local variable using an initializer list to assign elements to the array
- Array elements used in expressions (Boolean expressions in this case)
- Using array elements as both the source and destination of an assignment operation.

It does not demonstrate the following, however it is possible to:

- Declare arrays globally and use them in much the same fashion as in this program.
- Declare arrays in the following ways:
  - int arr[5]; (explicit number of elements but no initializer list)
  - int arr[5] = {1, 2, 3}; (explicit number of elements alongside an initializer list)

Final word count: 46911

Reflections

Based on my original goals for the project I could confidently call the project a success. My original goal was to have a compiler which could compile a program which calculates prime numbers, a goal which I managed to achieve, and have included this exact program among the set of examples I've made for demonstrations and testing.

I surpassed my own expectations by implementing functions. I did not expect to be able to do so since I originally believed the task would be too complex. A lot of work went into including them as a feature and I'm pleased with how they turned out despite certain features not being available, such as being able to pass arrays as arguments to functions or not allowing arrays to be the return type of functions.

Certain minor features such as switch case statements and for loops have not been included. Including them wouldn't have necessarily taken a significant amount of work, however since while loops and if statements which I have included serve the same purpose, I never saw switch case and for loops as large priorities.

The layout of the semantics in the grammar became a significant burden as the project progressed. As the bnf file grew larger it became increasingly more difficult to add new features simply due to the fact it took time to find what I was looking for. If I were to do a project similar to this in the future, I'd likely make a point of writing the semantic actions into auxiliary files, particularly the logic which handled symbol table insertions and symbol retrieval.

The type system I implemented was rudimentary at best. In a more ideal situation I'd have likely rewritten the grammar another time before adding arrays. I had to use flags to separate the semantics for different types, which further contributed to the code getting messy. Had I planned better for the inclusion of types, the code would have ended up being a lot cleaner overall. This would likely be the next step I would take if I were to continue working on the project.

I originally wanted to have the concept of pointers present in my implementation before adding arrays. I think if I already had the mechanisms in place for handling indirection, I would have likely ended up with a better design for arrays and would possibly found a more elegant approach to passing arrays as arguments to functions. Adding arrays was effectively me forcing myself to implement a feature without providing the proper foundation. I am not a fan of this approach to development, but working towards deadlines often means cutting corners, a concept I'll likely have to get used.

In my opinion, the single largest improvement I could realistically make to the compiler would be the inclusion of a more sophisticated register allocation strategy. This would significantly reduce the amount of data having to be retrieved directly from main memory, since registers could hold onto values for longer instead of data constantly being swapped in and out. This would also

Final word count: 46911

allow me to optimize my run time functions for saving and restoring values held by registers. Rather than having to save every register, I could store only those holding relevant values.

My ability to write grammars has significantly improved over the course of this project to the point where it feels very natural. Throughout the course of writing this report I have written several grammars purely for demonstrating certain ideas. These grammars took very little time and effort for me to write, compared to when I wrote my first grammar which was a very slow process. I feel this is a useful skill which I'd have otherwise not gained, which could very likely be applied to other projects (writing a DSL for example).

Professional Issues

It is not possible to create software without compilers. This is true at least in the conventional sense, so why is it then that historically the majority of compilers have been proprietary software? Indeed, There are only a handful of production quality open source compilers out there, most of which compile the same two languages (C and C++)[61].

A good compiler is difficult to write. What even constitutes a good compiler? Good code Optimization? Target's many architectures? Parses quickly and efficiently? None of these problems are easily solvable both due to their complexity and due to the large amount of work required to implement them. Therefore it is reasonable that as individuals we'd expect some monetary reward for the fruits of our labor. However with a tool as useful as a compiler, the reward does not necessarily have to come in the form of one that is monetary.

Historically compilers have been prohibitively expensive, often costing in the range of thousands of dollars (and remember that this dates back between 20 to 40 years ago, so it is even more expensive than it sounds).

Here are some prices of compiler's with varying price ranges:[62]

- Tiny Pascal for the TRS-80 ($15)
- FORTRAN for the TRS-80 ($99)
- Lattice C for IBM/MS-DOS (roughly $300, as I recall)
- Mix C for IBM/MS-DOS (roughly $75, I think)
- Intel Parallel Studio XE with Fortran & C++ compilers for Windows (roughly $1200)
- Visual Studio Professional with MSDN Subscription (about $1200 for one year) for Windows
- As late as 2004, Sun's Forte C compilers were around $3,000
- A standalone, "perpetual" Microsoft Visual Studio license still cost $499 only 4 years ago.

Final word count: 46911

Originally, between the 50's and 70's, compiler's would be included as part of a computer's purchase, since a compiler was necessary in order to render the machines back then useful in any way. However hardware was prohibitively expensive during this period anyway, so the addition of a quote un quote free compiler wasn't in any way shape or form a great deal for the consumer[63].

The C language first appeared in the early 70's[64], and it wasn't until the late 80's that gcc, an entirely free and open source compiler, became available[65]. Before then the portable C compiler which was under the BSD license was shipped with BSD Unix[66]. This was under the BSD license which is a permissive free software license[67], however the compiler itself was not free to obtain due to the fact that it was bundled with other software.

> "Absolutely. I can remember in high school paying $200 for a C compiler. In the 1980s for the Atari ST, that was a steal for a good compiler. In college I bought Turbo C (and a bunch of the C++ follow ons) to write code in DOS and later Windows and OS/2. Windows compiler options were mostly commercial products until about 10 years ago. High quality ones still do cost money"[68]
>
> Quote from "Shawn Masters"

I believe a large part of why C exists in so many places today is because the general public had access to a tool like gcc as early as the late 80's and early 90's. Having access to tools which can be used to build better tools means we will have better tools in the future. It is a fulfilling prophecy that progress begets progress, and this is a strong example of that.

Today, in the modern world we are fortunate enough to have access to many free compilers and interpreters which impose no restrictions on the products we produce using these tools, and I have personally been witness to innovation in many areas due to this fact. Not only this but a large portion of my education can be attributed to the fact that these tools have always been freely available to me. Whenever it was that I decided to teach myself C++, I was simply able to google for an IDE and begin writing programs and teaching myself the invaluable skills of programming.

I developed the implementation for my project entirely on a free linux installation where I was able to obtain a c compiler for free using simple terminal commands within seconds. The ease of access has allowed me to freely work and develop my tool without let or hindrance, and without ever having a financial burden placed upon me beyond having to purchase the machine which I was developing it on.

Because the option of a fully open source c compiler exists, there is nothing stopping me from looking directly at the source code and learning how to implement features in my own

Final word count: 46911

tools by studying an existing implementation. For the purposes of my project this wasn't necessary as taking this route would have likely far over complicated the development process, however it was always an option which was open to me.

Rdp, the tool which I have been using to generate my grammar, I have access to for free and am able to inspect and modify the source code as I see fit. This was particularly useful to me during the development of the project because I was able to make modifications to the auxiliary files. For example, I changed the implementation of the functions which generated temporary variables and labels to better fit my needs. When I was experiencing problems with scope, I was able to look directly into the source code, identify the problem and apply my own fix. I was also able to gain a much better understanding of how the internal mechanics of the software worked because I had this level of access to the tool I was using also.

In the realms of software the benefits of freedom of use and modification have allowed me and many others to succeed, and I hope we can continue to move in this direction going into the future.

Final word count: 46911

Appendix A - Submission Directory Structure

Note: all folder names are underlined. Everything else is a file.

- final_Project
  - documents
    - zfac127.final.pdf - The final report document for this project
    - ecc_user_manual.pdf - The user manual for the ecc compiler
  - ecc
    - bin
      - cgen - The code generation program used by ecc
      - rdparser - The parser used by ecc
    - src
      - cgen
        - Cgen.c - The main source file for the code generator
        - Mips.c - Contains the code generators implementation of the mips architecture
        - Mips.h - The header file for mips.c
        - Quad.c - Contains the logic which converts TAC instructions into quads
        - Quad.h - The header file for quad.c
        - sym_tbl.c - An implementation of a symbol table
        - sym_tbl.h - The header file for sym_tbl.c
        - tac.c - Contains the set of keywords and operations which can appear in a TAC file. Also contains functions used by quad.c to parse TAC files.
        - Tac.h - The header file for tac.c
        - Target.h - Contains the type definition of target, which is a generic template for different machine architectures.
      - parser
        - Func.bnf - The grammar for the C subset language which ecc compiles
        - Ma_aux.c - An auxiliary file containing functions for generating temporary variables and labels for use in three address code generation
        - Ma_aux.h - The header file for ma_aux.c
        - Queue.c - An implementation of a queue, used for temporary storage of array elements submitted via an initializer list during the parse phase
        - Queue.h - the header file for queue.c
        - Rdparser.c - The source file for the parser, generated by rdp and func.bnf
        - Rdparser.h - the header file for rdparser.c
        - Scope_chain.c - An implementation of a scope chain, which is effectively a linked list. This is used in place of rdp's internal scope chain.
        - Scope_chain.h - the header file for scope_chain.c
        - Stack.c - An implementation of a stack, used for keeping track of whether the parser is inside a loop or not.
        - Stack.h - the header file for stack.c
        - Tbl_aux.h - Contains the file handle used to output symbol table information
        - Type.h - Contains an enum which manages the different data types available in the C subset language which ecc compiles
      - README.txt - describes the contents of the cgen and parser folders
    - Test
      - Examples
        - Bubsort
          - Bubs.a - An executable mips program which sorts of array of integers
          - Bubs.c - The C source file for the bubs program
          - Bubs.tac - The TAC file for the bubs program
          - Bubs.tbl - The symbol table file for the bubsort program
          - Bubs_test_output.png - A screenshot of bub's output
          - README.txt - A description of bub's purpose
        - Fib
          - Fib.a - An executable mips program which displays the first 10 fibonacci numbers
          - Fib.c -The C source file for the fib program

Final word count: 46911

        - Fib.tac - The TAC file for the fib program
        - Fib.tbl - The symbol table file for the fib program
        - Fib_test_output.png - A screenshot of fib’s output
        - README.txt - A description of fib’s purpose
      - Prime
        - Prime.a - An executable mips program which calculates attempts to calculate every prime number from 1 to 2^{31}-1
        - Prime.c -The C source file for the prime program
        - Prime.tac - The TAC file for the prime program
        - Prime.tbl - The symbol table file for the prime program
        - Prime_test_output.png - A screenshot of prime.a running
        - README.txt - A description of prime’s purpose
      - Sum
        - Sum.a -An executable mips program which calculates the sum of integers from 1 to 20
        - Sum.c - The C source file for the sum program
        - Sum.tac - The TAC file for the sum program
        - Sum.tbl - The symbol table file for the sum program
        - Sum_test_output.png - A screenshot of sum’s output
        - README.txt - A description of sum’s purpose
    - cgen - The code generation program use dby ecc
    - ecc.sh - A shell script to make use of the compiler more practical
    - rdparser - The parser used by ecc
- various grammars
  - Conditional grammar
    - cond - executable binary / parser with working conditional statements
    - cond.bnf - grammar used by rdp to generate cond source files
    - cond.c - rdp generated source file, contains implementation for cond
    - cond.h - rdp generated header file, used by cond.c
    - cond.o - object code generated by gcc and cond source files
    - cond.str - input string parsable by cond
    - cond_output.png - output from cond binary / cond.str
    - cond_test_output.png - output from cond.bnf / cond.str from rdp
    - README.txt - explains cond’s inclusion in hand in
  - emit
    - emit.bnf - An example grammar which emits three address code
    - emit.str - input string for parsable by the grammar in emit.bnf
    - emit.tac - A file containing three address code which emit outputs
    - README - describes the purpose of emit.bnf
  - initial grammar
    - cll1 - executable binary / parser without semantic actions
    - cll1.bnf - grammar used by rdp to generate cll1 source files
    - cll1.c - rdp generated source file, contains implementation for cll1
    - cll1.h - rdp generated header file, used by cll1.c
    - cll1.o - object code generated by gcc and cll1 source files
    - cll1.str - input string parsable by cll1
    - cll1_test_output.png - output from cll1.bnf / cll1.str from rdp
    - README.txt - describes the functionality of expr
  - scope
    - README - describes the purpose of scope.bnf
    - scope.bnf - An example grammar which implements scoping rules
    - scope.str - input string parsable by the grammar in scope.bnf
  - semantic grammar
    - expr - executable binary / parser with working expression evaluation
    - expr.bnf - grammar used by rdp to generate expr source files
    - expr.c - rdp generated source file, contains implementation for expr
    - expr.h - rdp generated header file, used by expr.c
    - expr.o - object code generated by gcc and expr source files
    - expr.str - input string parsable by expr
    - expr_output.png - output from expr binary / expr.str
    - expr_test_output.png - output from expr.bnf / expr.str from rdp

Final word count: 46911

- README.txt - describes the functionality of expr
- README.md - contains the changelog for the project
- README.txt - contains a description of the file directory of the project folder

Final word count: 46911

References

1. Elizabeth Scott. (2020). Compilers and Code Generation. Description of vocabulary, grammar and semantics. pp. 3 - 4
2. Elizabeth Scott. (2020). Compilers and Code Generation. Formal definition of context free grammars. pp. 58.
3. Steven Bird, Ewan Klein, and Edward Loper. Natural Language Processing with Python (2019). Chapter 8. Analyzing Sentence Structure.
4. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. (2007). Compilers Principles, Techniques, & Tools Second Edition. Formal definitions of terminals, non terminals and production rules. pp. 197
5. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. (2007). Compilers Principles, Techniques, & Tools Second Edition. Example grammar and derivation. pp. 43 - 44
6. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. (2007). Compilers Principles, Techniques, & Tools Second Edition. Derivations. pp. 199
7. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. (2007). Compilers Principles, Techniques, & Tools Second Edition. Formal definition of derivation step and derivations. pp. 200
8. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. (2007). Compilers Principles, Techniques, & Tools Second Edition. Formal definition of parse trees. pp. 45
9. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. (2007). Compilers Principles, Techniques, & Tools Second Edition. Ambiguity, operator associativity, reducing ambiguity. pp. 47 - 49
10. Elizabeth Scott. (2020). Compilers and Code Generation.Formal definition of first sets. pp. 58
11. Elizabeth Scott. (2020). Compilers and Code Generation.Calculating first sets by hand. pp. 59
12. LL Parser. Wikipedia, Wikimedia Foundation, 27 October 2020, https://en.wikipedia.org/wiki/LL_parser. Definition of LL parsers.
13. Parsing - Lookahead. Wikipedia, Wikimedia Foundation, 27 October 2020, https://en.wikipedia.org/wiki/Parsing#Lookahead. LL(k) parser and lookahead tokens.
14. Elizabeth Scott. (2020). Compilers and Code Generation. Formal definitions of LL(1) Grammar, left factored grammars and follow determinism. pp. 63
15. Elizabeth Scott. (2020). Compilers and Code Generation. Adapted definition of LL(1) grammars and FIRST /FIRST conflicts. Formal definition of follow sets and example grammar. pp. 62
16. Elizabeth Scott. (2020). Compilers and Code Generation. Adapted definition of LL(1) grammars and FIRST /FIRST conflicts. Formal definition of left recursion. pp. 60
17. Elizabeth Scott. (2020). Compilers and Code Generation. Recursive descent parsing. pp. 63 - 65
18. Elizabeth Scott. (2020). Compilers and Code Generation. Definition & explanation of EBNF. pp. 65
19. Elizabeth Scott. (2020). Compilers and Code Generation. Description of rdp. pp. 68

Final word count: 46911

20. Elizabeth Scott. (2020). Compilers and Code Generation.Languages accepted by rdp. pp. 68 - 69
21. Adrian Johnstone, Elizabeth Scott. (1997). rdp - a recursive descent compiler compiler User manual for version 1.5. Description of IBNF. pp. 7
22. Adrian Johnstone, Elizabeth Scott. (1997). rdp - a recursive descent compiler compiler User manual for version 1.5. Scanner Elements. pp. 16
23. Adrian Johnstone, Elizabeth Scott. (1997). rdp - a recursive descent compiler compiler User manual for version 1.5. Layout and comments. pp. 9
24. Brian W. Kernighan, Dennis M. Ritchie. (1978). The C Programming Language Second Edition. Appendix A, Grammar, Statements. pp. 236
25. Brian W. Kernighan, Dennis M. Ritchie. (1978). The C Programming Language Second Edition. Appendix A, Grammar, Jump Statements. pp. 237
26. Brian W. Kernighan, Dennis M. Ritchie. (1978). The C Programming Language Second Edition. Appendix A, Grammar, Iteration Statements. pp. 237
27. Brian W. Kernighan, Dennis M. Ritchie. (1978). The C Programming Language Second Edition. Appendix A, Grammar, Unary Operators. pp. 237
28. Adrian Johnstone, Elizabeth Scott. (1997). rdp - A tutorial guide to rdp for new users. Description of attributes. pp. 37
29. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. (2007). Compilers Principles, Techniques, & Tools Second Edition. Definition of symbol tables. pp. 85
30. Adrian Johnstone, Elizabeth Scott. (1997). rdp symbol table function signature, parameters and example. pp. 31
31. Elizabeth Scott. (2020). Compilers and Code Generation.Languages accepted by rdp. pp. 12
32. Elizabeth Scott. (2020). Compilers and Code Generation.Languages accepted by rdp. pp. 11 - Portability
33. Three-address code Wikipedia, Wikimedia Foundation, 25 March 2021, https://en.wikipedia.org/wiki/Three-address_code. Definition of LL parsers - Definition of three-address code
34. Three address code in Compiler GeeksforGeeks, 25 March 2021, https://www.geeksforgeeks.org/three-address-code-compiler/ - General representation of three address code
35. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. (2007). Compilers Principles, Techniques, & Tools Second Edition. Three-Address Code. pp 363
36. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. (2007). Compilers Principles, Techniques, & Tools Second Edition. Addresses and Instructions. pp. 364 - 365
37. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. (2007). Compilers Principles, Techniques, & Tools Second Edition. Quadruples. .pp. 366
38. Three address code in Compiler GeeksforGeeks, 25 March 2021, https://www.geeksforgeeks.org/three-address-code-compiler/ - The Structure of quadruples.

Final word count: 46911

39. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. (2007). Compilers Principles, Techniques, & Tools Second Edition. The Primary Tasks of Code Generation. pp. 505
40. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. (2007). Compilers Principles, Techniques, & Tools Second Edition. Instruction Selection. pp. 508
41. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. (2007). Compilers Principles, Techniques, & Tools Second Edition. Generating More Efficient Code. pp. 509
42. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. (2007). Compilers Principles, Techniques, & Tools Second Edition. Register Allocation. pp. 510
43. Register Allocation Wikipedia, Wikimedia Foundation, 25 March 2021, https://en.wikipedia.org/wiki/Register_allocation. Graph Colouring Allocation
44. The Registers, Obelisk, Andrew John Jacobs, 25 March 2021, http://www.obelisk.me.uk/6502/registers.html
45. The Instruction Set, Obelisk, Andrew John Jacobs, 25 March 2021, http://www.obelisk.me.uk/6502/instructions.html
46. Mips Architecture Wikipedia, Wikimedia Foundation, 25 March 2021, https://en.wikipedia.org/wiki/MIPS_architecture. MIPS32/64
47. MIPS32 Instruction Set Reference, MIPS, 25 March 2021. https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00565-2B-MIPS32-QRC-01.01.pdf
48. The MIPS Register Files, University of Wisconsin-Milwaukee, 25 March 2021, http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch05s03.html
49. System Calls, Department of Computing - Imperial College London, 25 March 2021, https://www.doc.ic.ac.uk/lab/secondyear/spim/node8.html
50. ISO/IEC 9899:TC3:2007, Information technology - Programming languages - C pp. 21
51. MIPS32 Instruction Set Reference, MIPS, 25 March 2021. https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00565-2B-MIPS32-QRC-01.01.pdf
52. Adrian Johnstone, Elizabeth Scott. (1997). rdp_supp - support routines for the rdp compiler compiler, User manual for version 1.5. Function signature for symbol_get_scope. pp. 33
53. Adrian Johnstone, Elizabeth Scott. (1997). rdp_supp - support routines for the rdp compiler compiler, User manual for version 1.5. Function signature for symbol_new_scope. pp. 34
54. Adrian Johnstone, Elizabeth Scott. (1997). rdp_supp - support routines for the rdp compiler compiler, User manual for version 1.5. Function signature for symbol_unlink_scope. pp. 37
55. Stack vs Heap. What's the difference and why should I care? Nickolas Teixeira Lanza, 25 March 2021, https://nickolasteixeira.medium.com/stack-vs-heap-whats-the-difference-and-why-should-i-care-5abc78da1a88
56. Understanding how function call works, Zeyuan Hu, 25 March 2021, https://zhu45.org/posts/2017/Jul/30/understanding-how-function-c

Final word count: 46911

57. The Stack Frame, University of Wisconsin-Milwaukee, 25 March 2021, http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch05s03.html
58. Allocating Space on the Stack for Local Variables, University of Wisconsin-Milwaukee, 25 March 2021, http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch05s03.html
59. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. (2007). Compilers Principles, Techniques, & Tools Second Edition.Intermediate Code for Procedures. pp. 422
60. Variable-length Array Wikipedia, Wikimedia Foundation, 25 March 2021, https://en.wikipedia.org/wiki/Variable-length_array. Definition of VLA
61. List of Compilers Wikipedia, Wikimedia Foundation, 25 March 2021, https://en.wikipedia.org/wiki/List_of_compilers.
62. Did code compilers used to cost money? Quora, Ken Gregg, Toby Thain, 25 March 2021, https://www.quora.com/Did-code-compilers-used-to-cost-money
63. Did code compilers used to cost money? Quora, Jesse Pollard, 25 March 2021, https://www.quora.com/Did-code-compilers-used-to-cost-money
64. C (Programming Language), Wikimedia Foundation, 25 March 2021, https://en.wikipedia.org/wiki/C_(programming_language). When did C first appear?
65. GNU Compiler Collection, Wikimedia Foundation, 25 March 2021, https://en.wikipedia.org/wiki/GNU_Compiler_Collection. Initial release of GCC
66. Portable C Compiler, Wikimedia Foundation, 25 March 2021, https://en.wikipedia.org/wiki/Portable_C_Compiler. PCC
67. BSD Licenses, Wikimedia Foundation, 25 March 2021, https://en.wikipedia.org/wiki/BSD_licenses. Description of BSD licensing
68. Did code compilers used to cost money? Quora,Shawn Masters, 25 March 2021, https://www.quora.com/Did-code-compilers-used-to-cost-money

Final word count: 46911