Elliot Carter
CS3821
Full Unit Project
Interim Report

December 6th 2020

Abstract

In this document I aim to explain my process and current progress of the development of my C compiler. I will begin by walking the reader through all the necessary theory required to understand the approach I have taken in its development thus far. 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 own grammar and how it has evolved over the course of this project thus far and state my intentions for the project's future.

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. All other theory involved in this project is to be understood over the course of reading this document.

Contents

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.

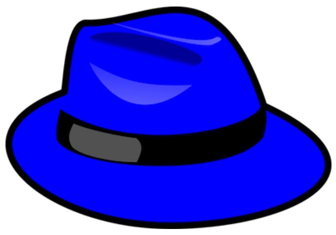
Vocabulary - The set of individual words for a given language

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.

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.

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.

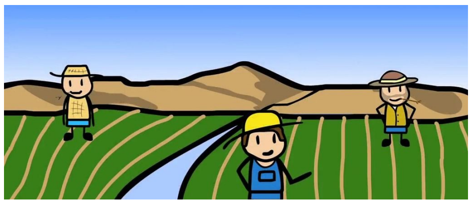
Take the following example

“...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.

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 $A \to \alpha$, where A is a single non terminal symbol and α 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 α in $A \to \alpha$) 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

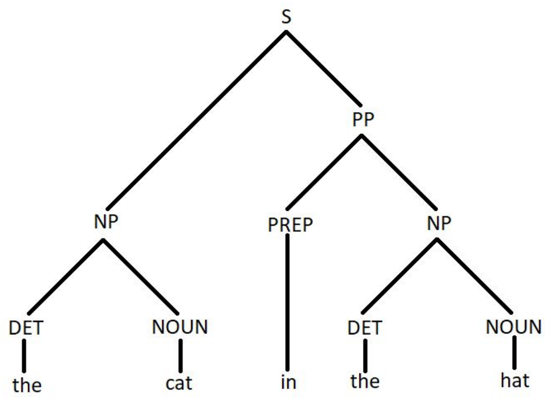
$$PP \rightarrow PREPOSITION\ NP$$
$$NP \rightarrow DETERMINER\ NOUN$$

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]

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
- $\rightarrow$ 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

$T \ = \{0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ +,\ -\}$

$N \ = \{\mathit{list},\ \mathit{digit}\}$

$S \ = \mathit{list}$

$P \ = \{$

$\mathit{list} \rightarrow \mathit{list} + \mathit{digit}$	**1**
$\mathit{list} \rightarrow \mathit{list} - \mathit{digit}$	**2**
$\mathit{list} \rightarrow \mathit{digit}$	**3**
$\mathit{digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$	**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** **(**$\mathit{list} \rightarrow \mathit{digit}$**)**, since 9 is a digit
2. 9 - 5 is a list by production **2** **(**$\mathit{list} \rightarrow \mathit{list} - \mathit{digit}$**)**, since 9 is a list and 5 is a digit
3. 9 - 5 + 3 is a list by production **1** **(**$\mathit{list} \rightarrow \mathit{list} + \mathit{digit}$**)**, since 9 - 5 is a list and 2 is a digit [5]

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

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

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

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 β 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 $\alpha_1 \Rightarrow \alpha_2 \ldots \Rightarrow \alpha_n$ rewrites α_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.

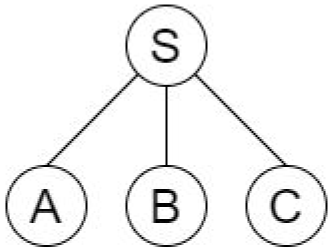
Parsing is one of the key stages of compilation. It is necessary to do so in order to verify that source code has been written correctly as defined by the language's grammar. Additionally, parsing is necessary not just to verify grammatical integrity, but also to infer the semantic actions described 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

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.

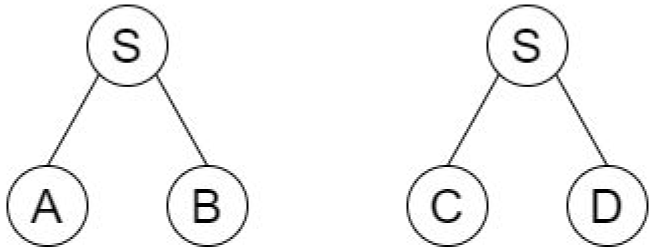
Take the following grammar $S \to ABC$, which can only result in the following parse tree.



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]

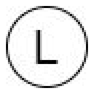
Let's take another example $S \to AB \mid CD$. Now we are dealing with two alternates, which means the grammar can form two possible trees.



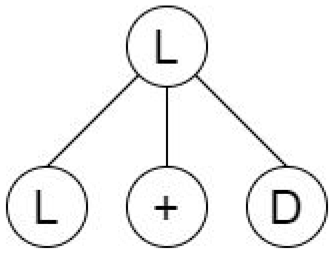
In practice these two trees would likely represent two separate derivations, of two separate input strings, under the same grammar. Note that I explicitly said this is likely however, not necessarily the case, a fact which I will explore in a later section.

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

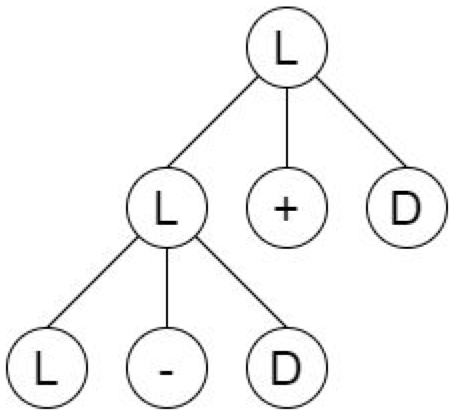
We begin by building a node for the start symbol $List$, which in the tree I will represent as a node containing the letter L.



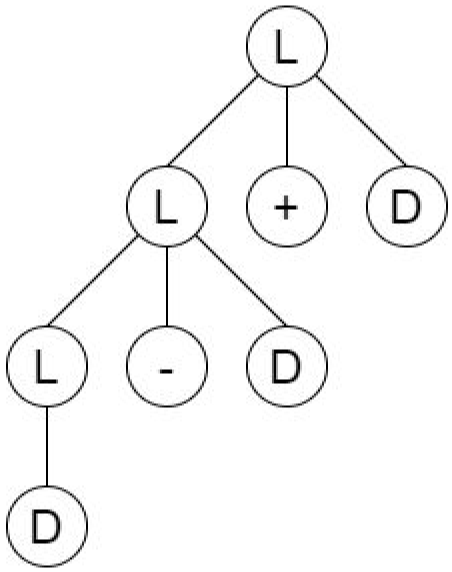
Next we take the rule $List \rightarrow List + Digit$, which for reasons which will become clear in a moment, will allow us to match the rest of the expression. Also note that I am using nodes containing the letter D to represent $Digit$.



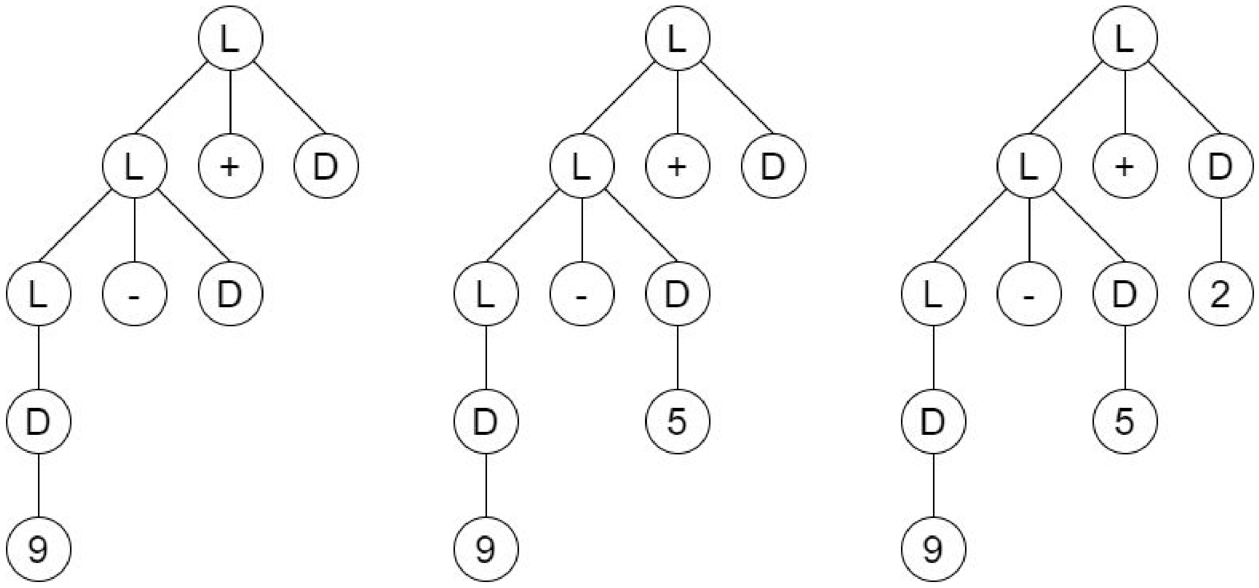
We follow by matching the leftmost leaf node containing L with the rule $List \rightarrow List - Digit$.



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.



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.

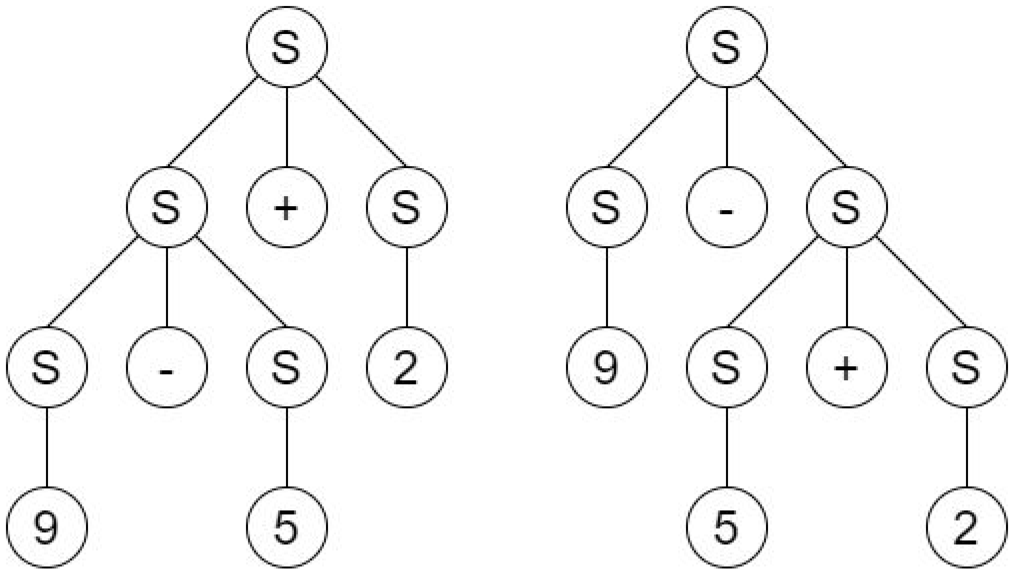
We have a grammar:

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

And a numerical expression:

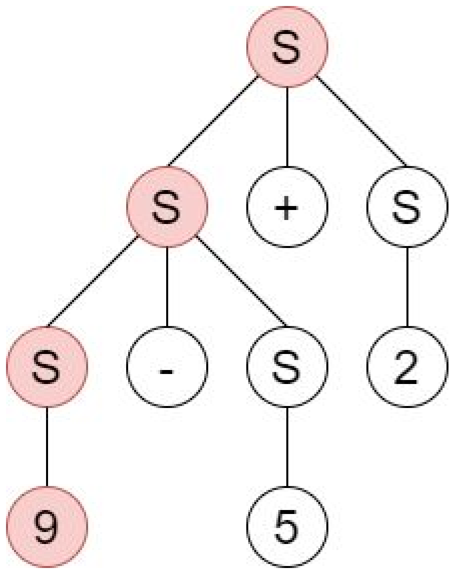
$$9 - 5 + 2$$

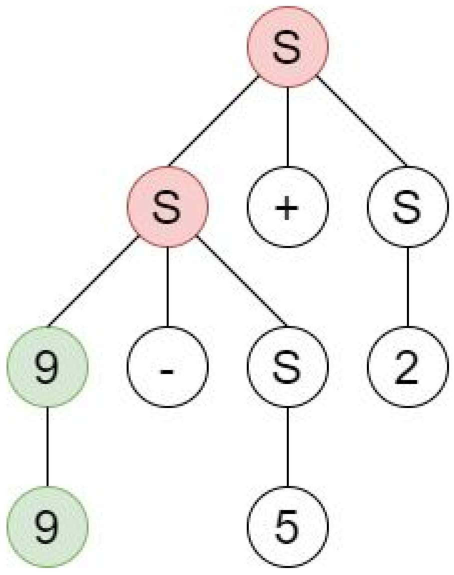
Now observe the following two derivation trees.



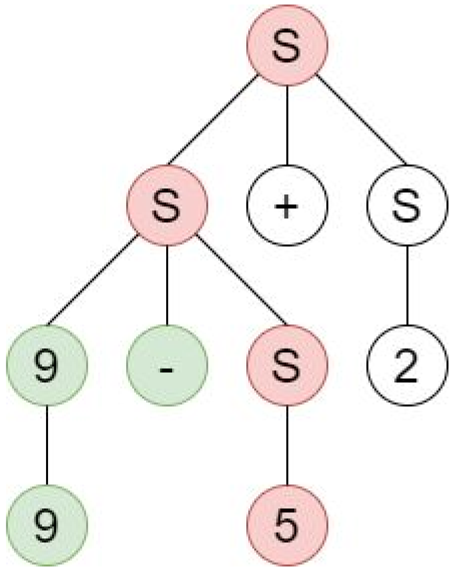
Both of these derivations are perfectly legal according to the grammar which means it is possible to derive the same string in more than one way. This would not be a problem if our only goal was to verify the validity of the string, however when we involve the semantics, we run into a problem.

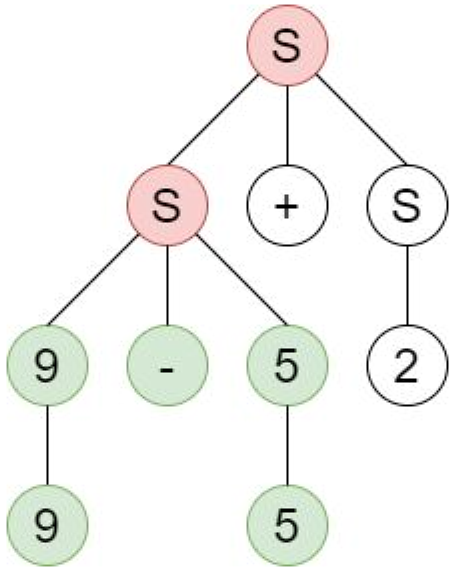
In order to evaluate each expression represented by these trees I will perform a depth first traversal through each where I will only perform numerical operations after all the child nodes of a parent have been visited. Observe the following:



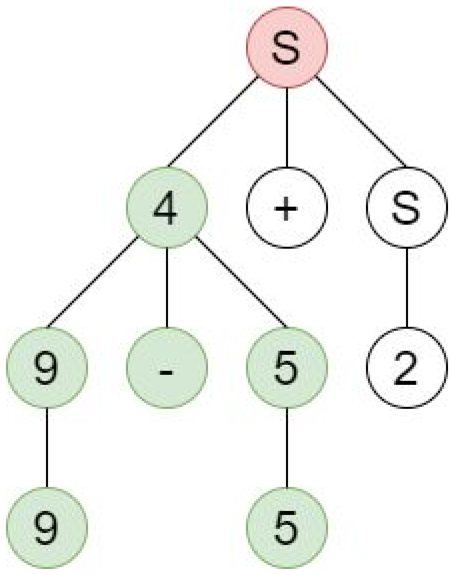


As you might expect, we begin by traversing each leftmost node until we reach a leaf (as depicted by the tree on the left). Since the node containing 9 is its parent's only child, we immediately assign this value to the parent node (as depicted by the tree on the right).

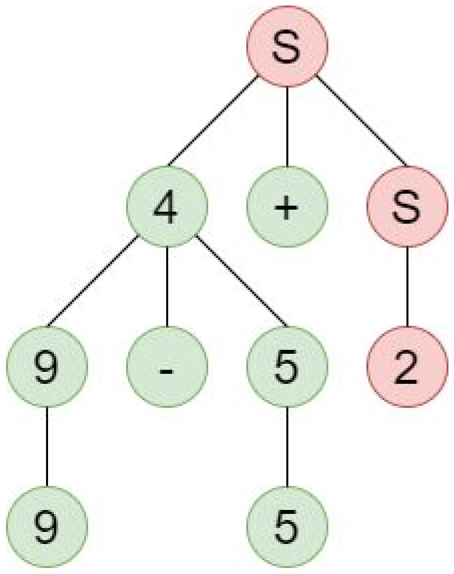


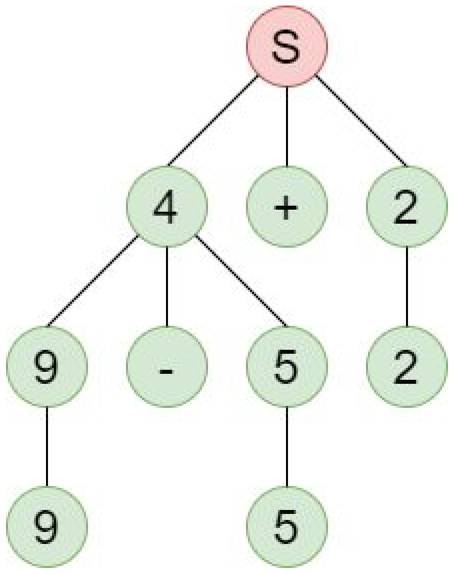


We continue traversing the tree, visiting the node containing the minus and the node containing the 5. Once again the value is immediately assigned to its parent node.

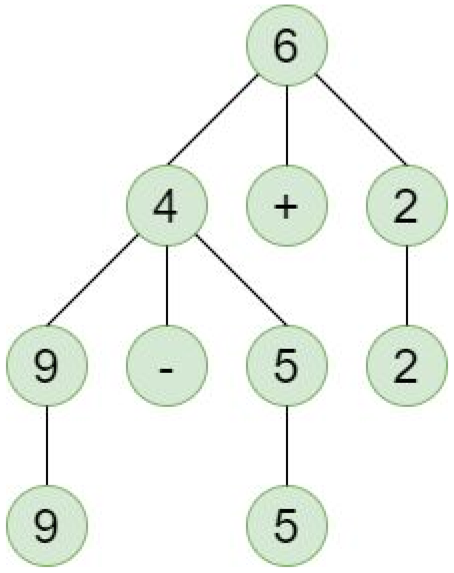


Now that the entire left side has been visited, we evaluate the expression 9 - 5, and assign the resulting value, 4, to the parent node.

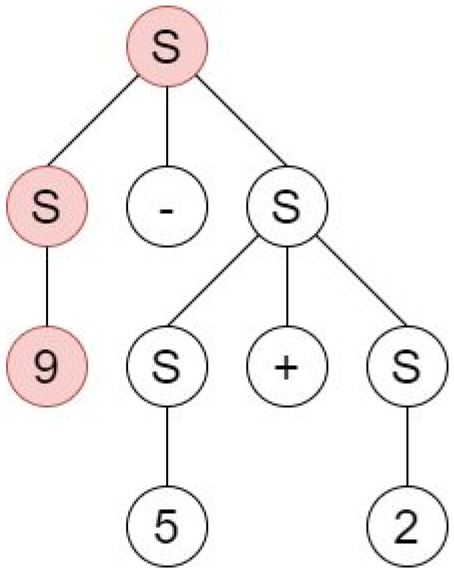


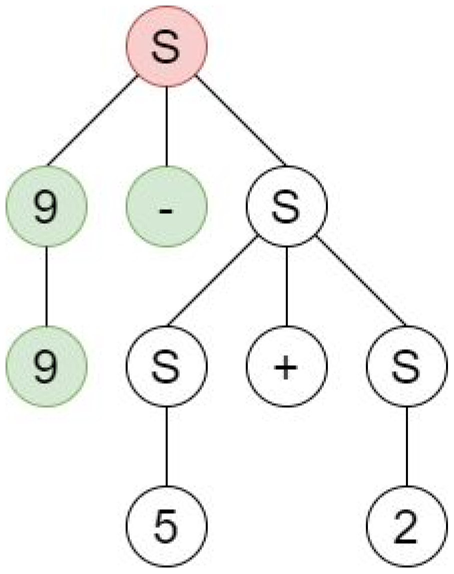


We continue down the right side as we did with the left side, visiting each node and immediately assigning 2 to the parent node.

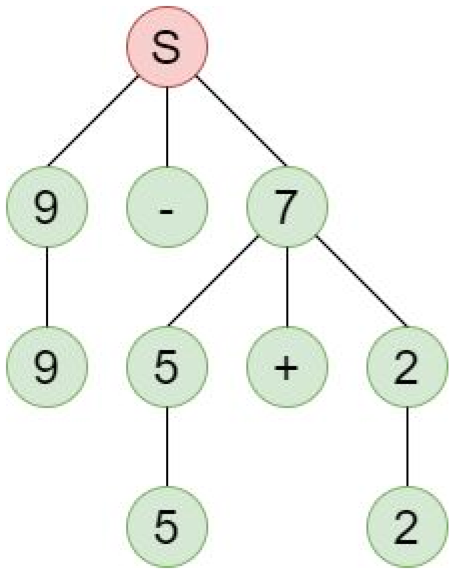


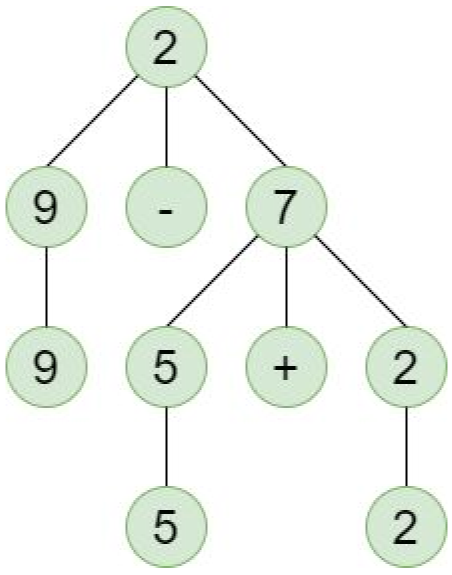
Now we've finally arrived back at the start node after visiting every other node and we can finally perform the final evaluation and assign the expected result of 6 to the start node. Now this might all seem fine, but let's observe what happens when we apply the same process with the other tree.





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





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.

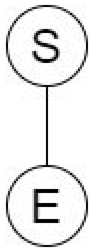
This example serves to demonstrate the fact that if a grammar is not structured correctly to match the intended semantics of the language, this problem will occur regardless of the limitations of the parser, assuming the parser is capable of handling this grammar at all. Note also, that this problem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

Reducing Ambiguity

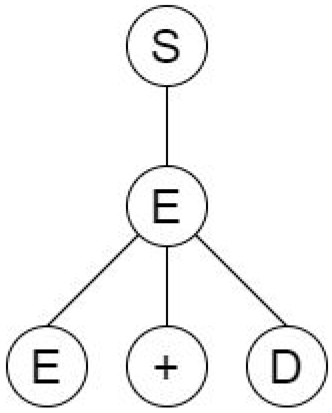
Now examine the following grammar.

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

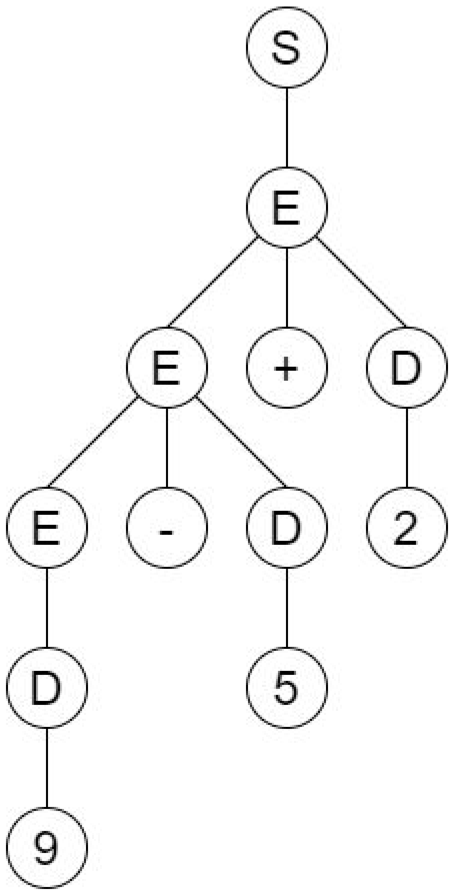
Now let's observe how the tree would be built for the same expression with this grammar.



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



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]

Top Down Parsing

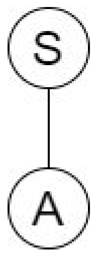
As previously mentioned, a top down parser works by first starting at the start symbol of a grammar and gradually producing the set of terminals matching the input string.

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.

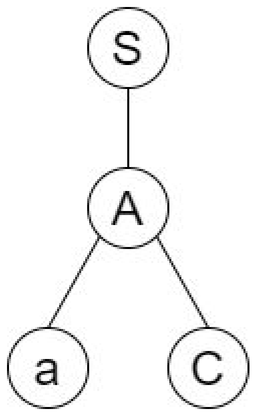
Here is an example of this:

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

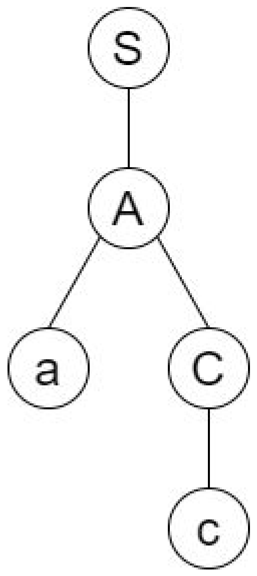
$Input\ String : ac$



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 $A \rightarrow aC$.

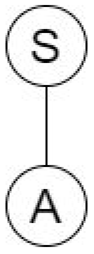


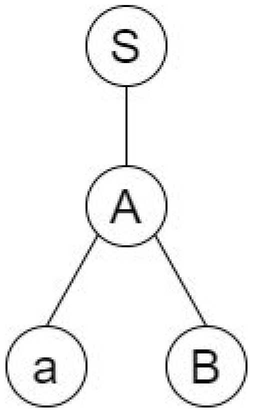
3. At this final step of the derivation we simply take the only remaining optio $C \rightarrow c$, which ends up producing the desired string, ‘ac’.

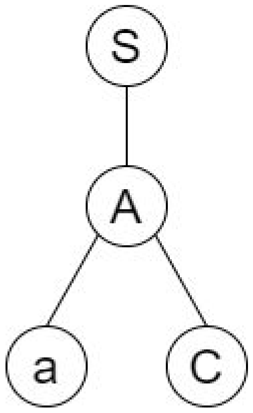
If a parse of an input string which is not permissible by the grammar is attempted, there will come a point in the tree's construction where no alternate of any production rule can be applied to continue the process, meaning it has failed.

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.

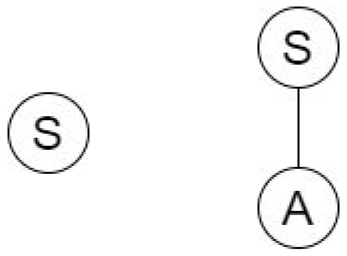
Besides demonstrating the process of a failed parse, this example also highlights another issue which we've encountered previously but somewhat glossed over. The element of choice was presented here, where since $A \rightarrow aB$ would not produce the desired result, $A \rightarrow aC$ was attempted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

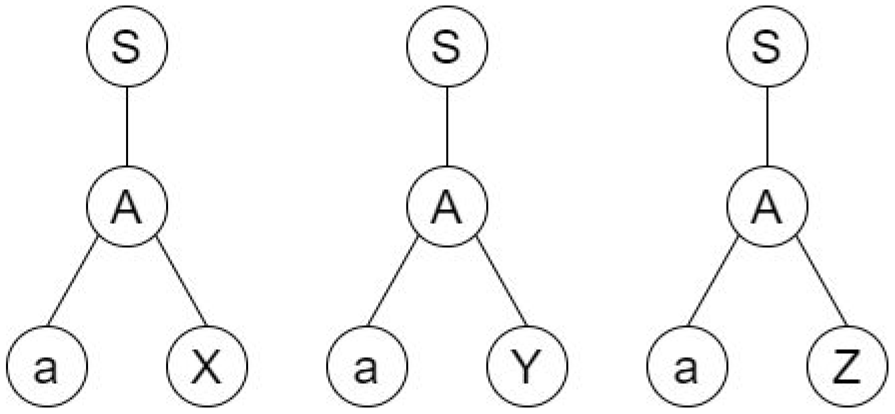
$S \rightarrow A \mid B$
$A \rightarrow$ aX | aY | aZ
$B \rightarrow bX \mid bY \mid bZ$
$X \rightarrow x$
$Y \rightarrow y$
$Z \rightarrow z$

$Input\ String : \ bz$

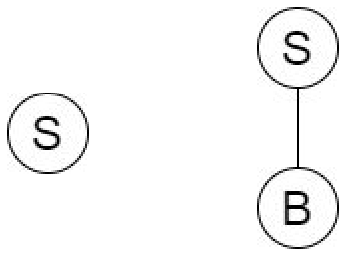
Once again as humans faced with such a simple grammar and a short input string, we can clearly see the set of steps necessary to produce the desired result, but let's observe how a machine might perform the same task.



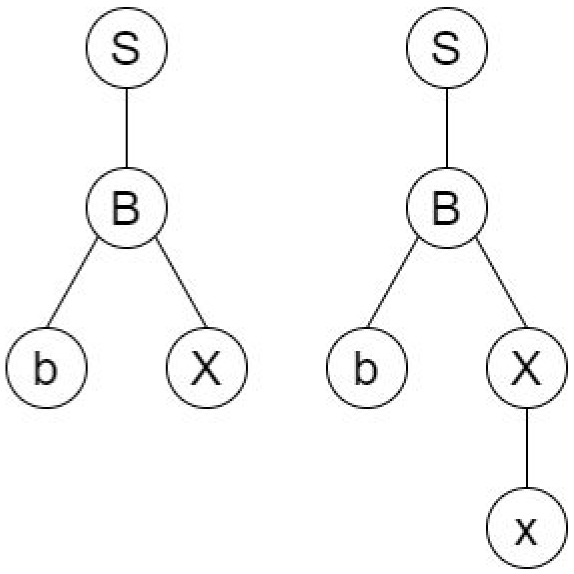
As usual we begin with the start symbol, and thereafter, are immediately presented with a choice of either $S \rightarrow A$ or $S \rightarrow B$. Since we are a methodical machine, we will simply work our way through these alternates from left to right until we either produce the desired string, or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



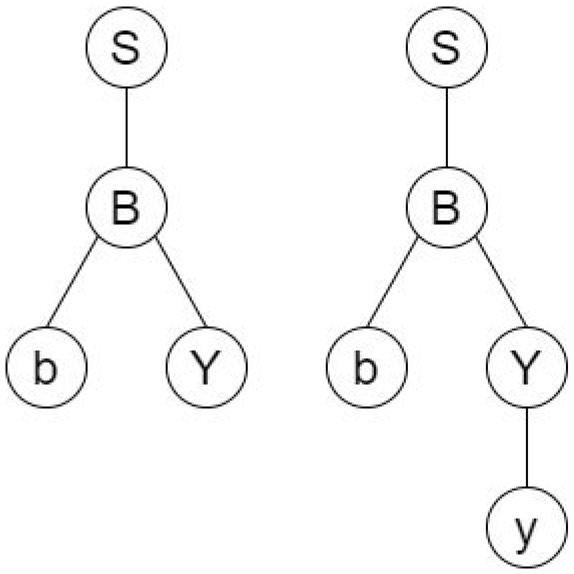
Continuing with our task, we try all possible alternatives of A in turn before concluding that it will not be possible to match the character 'b' given our previous step. Thus, we must backtrack.



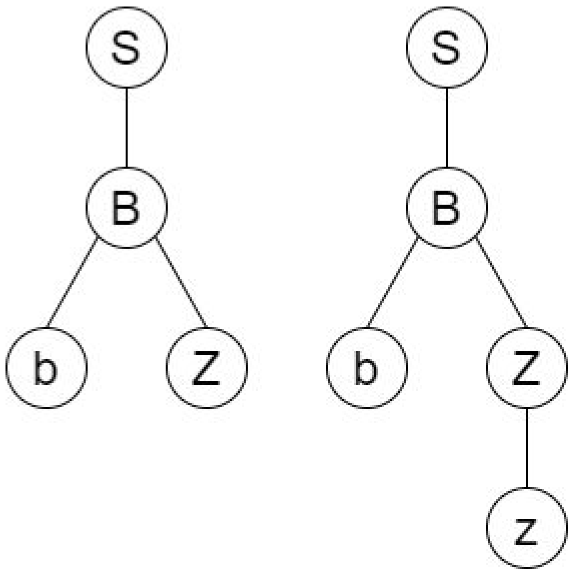
We have now backtracked into a previous state and are now ready to attempt the possible choices presented via the second alternate of S, $S \rightarrow B$.



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.



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.

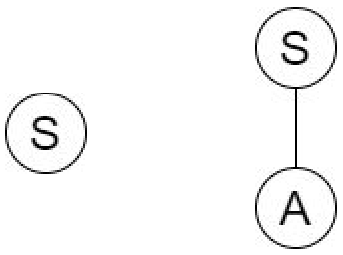
$S \rightarrow A \mid B$
$A \rightarrow$ aX | aY | aZ
$B \rightarrow bX \mid bY \mid bZ$
$X \rightarrow x$
$Y \rightarrow y$
$Z \rightarrow z$

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

$$A \rightarrow \text{aX | aY | aZ}$$

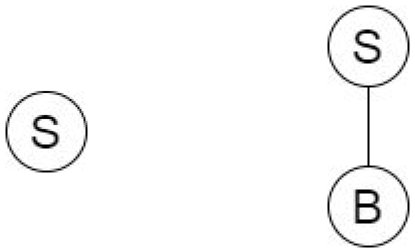
We can observe that for A the first terminal symbol which appears in each of its alternates is the character ‘a’. This is an important fact because it means we have a method of grouping these alternates in a way which can determine whether or not we need to check any of them at all.

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

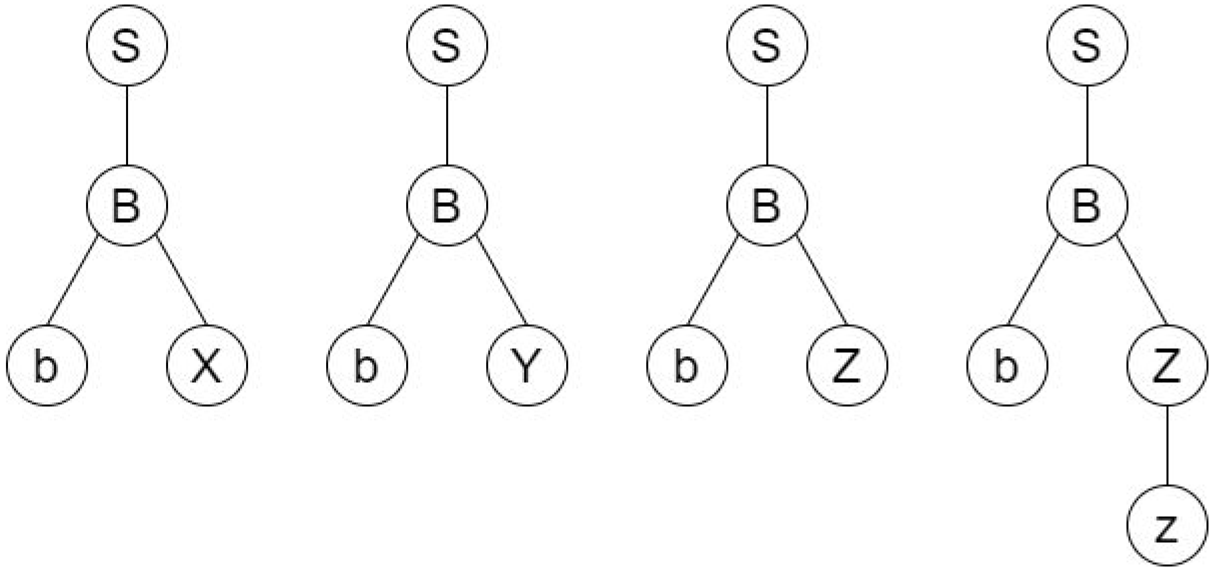


We begin the process in the exact same way as before, checking the alternates of S in turn, only this time we know not to check the individual alternates of A because we know ahead of time that the only character we could possibly match against first if is ‘a’ and the first character we intend to match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

$B \rightarrow bX \mid bY \mid bZ$



Having immediately concluded that $S \rightarrow A$ has no chance of producing the desired result, we immediately turn to the second alternate of S, $S \rightarrow B$. 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 $S \rightarrow A$, we also avoid explicitly checking 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 α 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 = $\varnothing$
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)/{$\varepsilon$})
            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.

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

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

$If\ A \Rightarrow^* \varepsilon,\ First(A\delta) = (First(A) \setminus \{\varepsilon\}) \cup First(\delta)$
$Else\ \ First(A\delta) = First(A)$

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

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, $A \to Ad$ recall:

$If\ A \Rightarrow^* \varepsilon,\ First(A\delta) = (First(A) \setminus \{\varepsilon\}) \cup First(\delta)$
$Else\ \ First(A\delta) = First(A)$

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, $A \to d$ recall $First(\alpha\delta) = \{\alpha\}$. 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 $S \to aBA \mid BB \mid Bd$, we get the following:

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

For the first alternate, $S \to aBA$, again we make use of $First(\alpha\delta) = \{\alpha\}$, 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, $S \to BB$, recall once again:

$If\ A \Rightarrow^* \varepsilon,\ First(A\delta) = (First(A) \setminus \{\varepsilon\}) \cup First(\delta)$
$Else\ \ First(A\delta) = First(A)$

Because B is nullable, we do the following:

$First(BB) = (First(B) \setminus \{\varepsilon\}) \cup First(B)$

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

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

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

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

Once again $(First(B) \setminus \{\varepsilon\}) = \varnothing$, 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:

$$\textit{If for any grammar } G, \textit{ which contains a rule } S \rightarrow \alpha \mid \beta\text{ , } \textit{where } First(\alpha) \cap First(\beta) \neq \varnothing$$
$$\textit{Then } G \textit{ is not left factored}$$ [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:

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

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

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

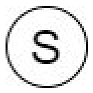
We can clearly see tokens ‘a’, ‘b’ and ‘c’ follow D in these rules, therefore

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

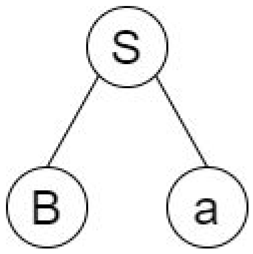
The reason the we sometimes needs to know which token follows a certain non terminal, is because when that non terminal is nullable, we need to know if we can match the current token of lookahead with not only the the set of tokens in the FIRST set of that non terminal, but also the set of tokens which 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.

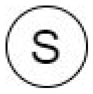


We begin with our start token S and our first and only lookahead token ‘a’. We know $First(Ba)$ to be {a, b}, which contains our lookahead token, so we can expand using this alternate.

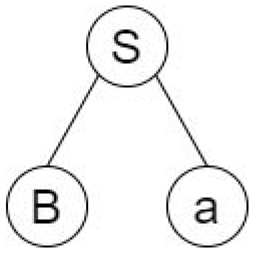


We have expanded our tree, and proceed to check our leftmost node, however, $First(b)$ is {b}, neither of which match our lookahead token ‘a’.

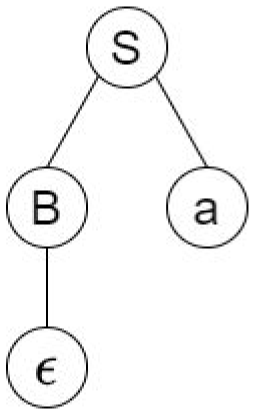
Because the FOLLOW set of B isn’t taken into consideration, whilst parsing B we have no way of verifying that “a” is meant to be a legal matching symbol. Now we shall do the same only this time, we consider both the FIRST and FOLLOW sets of B, since it is nullable.



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.



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:

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

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

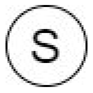
However, it turns out that if a grammar is left factored, then the following properly is enough to guarantee that it is also follow determined:

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

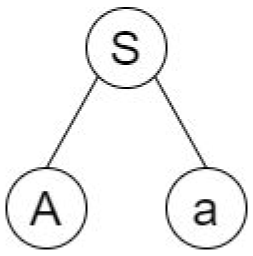
We can examine the following grammar in order to understand how this affects the parser:

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

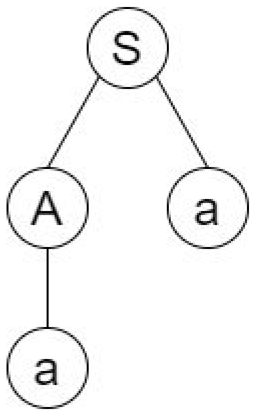
The two sentences which exist in this grammar are “a” and “aa”. Let's observe what happens if we attempt to parse “aa”.



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.



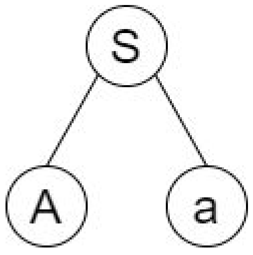
After expanding the tree, we move to our leftmost node. We know A is nullable and $First(a) \cup Follow(A)$ to be {a}. Our lookahead token is still ‘a’, so we match it with the rule $A \rightarrow a$ and move onto our next lookahead token, which is also ‘a’.



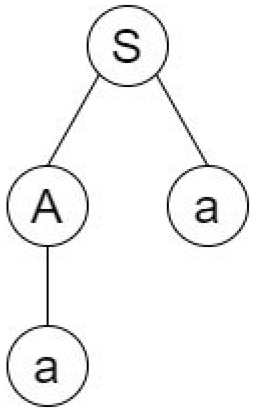
After moving back to the root node, we explore the rightmost node, and find that our second token of lookahead matches ‘a’, thus we have successfully produced our input string.

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



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.

“End-Of-File”.

As demonstrated above, the problem revolves around the fact that there was no way of knowing whether to take the ‘a’ or ε when 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.

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* α

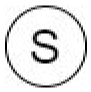
A production shows immediate left recursion if $A \rightarrow A\alpha$ [16]

Consider the grammar:

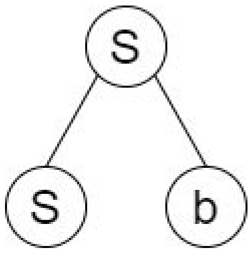
$S \rightarrow Sb \mid a$

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.

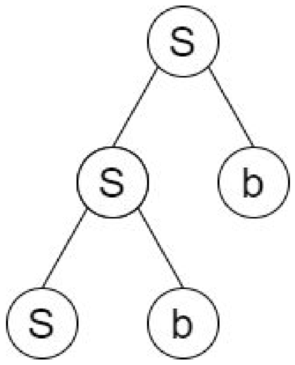
$Input\ String:\ ab$



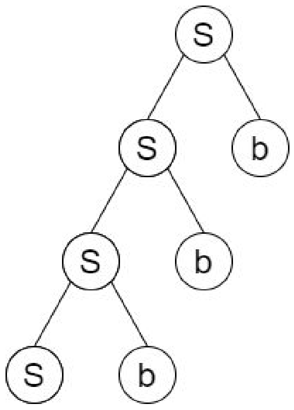
As with any derivation, we begin with the start symbol. Our lookahead token is currently ‘a’. $S \rightarrow Sb$ is the leftmost alternate, so this is the alternate which we interact with first. $First(Sb) = \{a\}$ so we proceed with this alternate.



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...

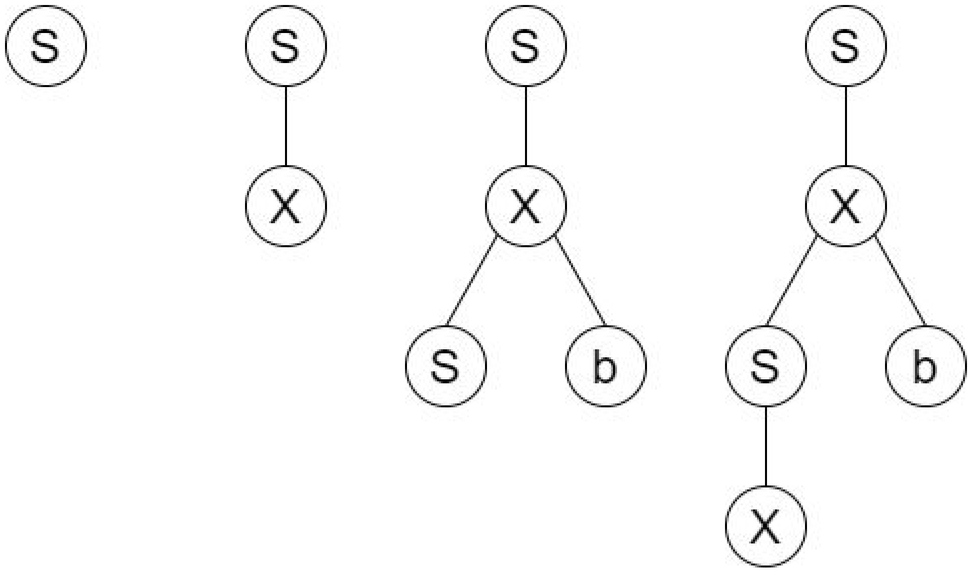


...and thus it never terminates.

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.

$$\begin{aligned} S &\rightarrow X \\ X &\rightarrow Sb \mid a \end{aligned}$$

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:

$$\begin{aligned} S &\rightarrow aS' \\ S' &\rightarrow bS' \mid \varepsilon \end{aligned}$$

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.

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
}
```

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]

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>

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

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

<u>Example:</u>

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

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

<u>Example:</u>

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

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

<u>Example:</u>

$$\begin{aligned} S &\rightarrow aB \\ B &\rightarrow bB' \\ B' &\rightarrow bB' \mid \varepsilon \end{aligned}$$

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]

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’.
```

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 *)
```

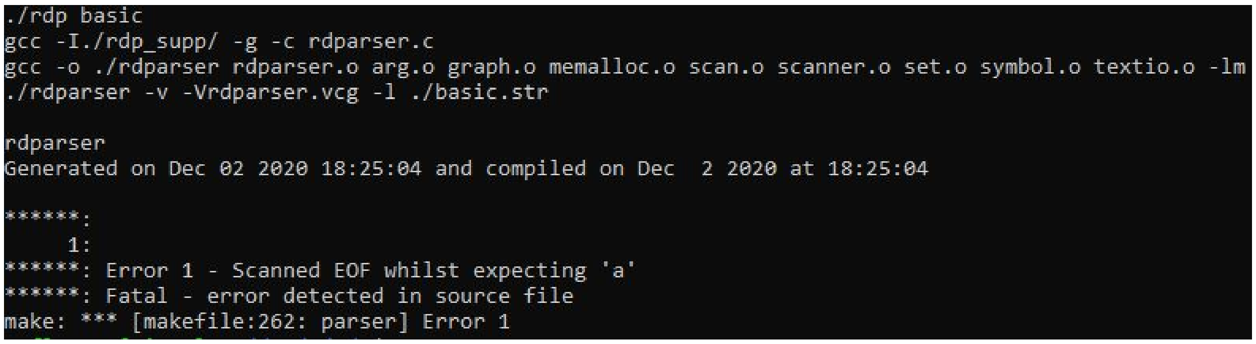
rdp Basic Usage

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:

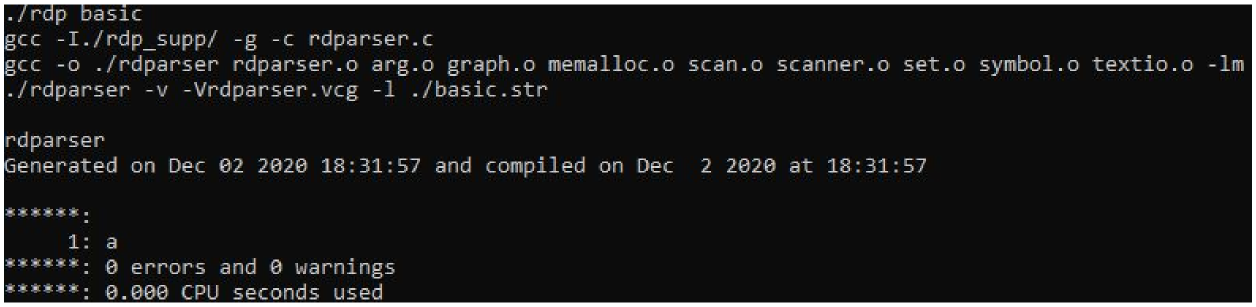


```
./rdp basic
gcc -I./rdp_supp/ -g -c rdparser.c
gcc -o ./rdparser rdparser.o arg.o graph.o memalloc.o scan.o scanner.o set.o symbol.o textio.o -lm
./rdparser -v -Vrdparser.vcg -l ./basic.str

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.

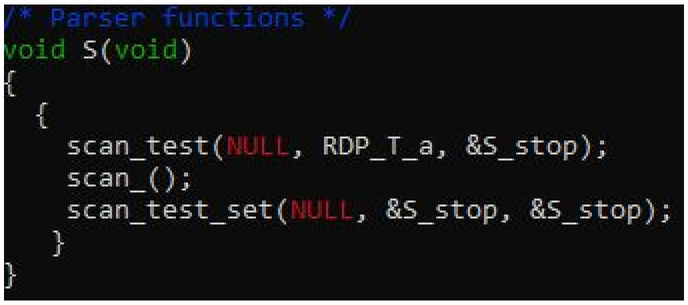
With the input string ‘a’, we get the following output:



```
./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);
   }
}
```

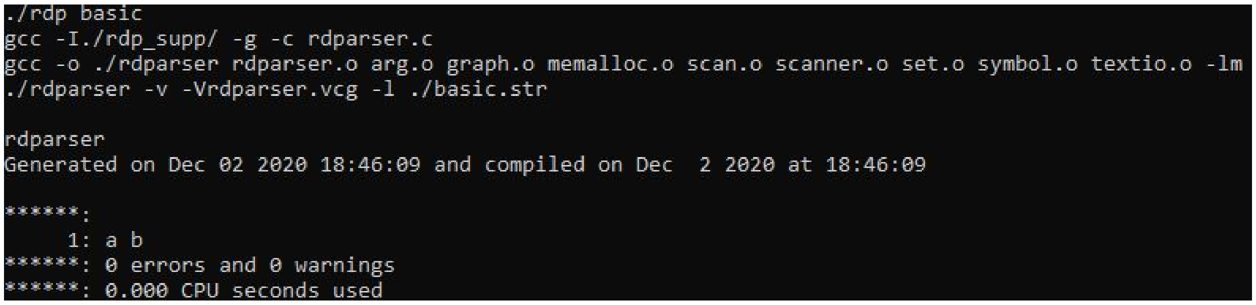
This is the parse function for our production rule. Not to go into too much detail, essentially what is happening here is the same process I previously described in an earlier section. The parser is first checking the FIRST set of the alternate, and then it is matching against the terminal symbol ‘a’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

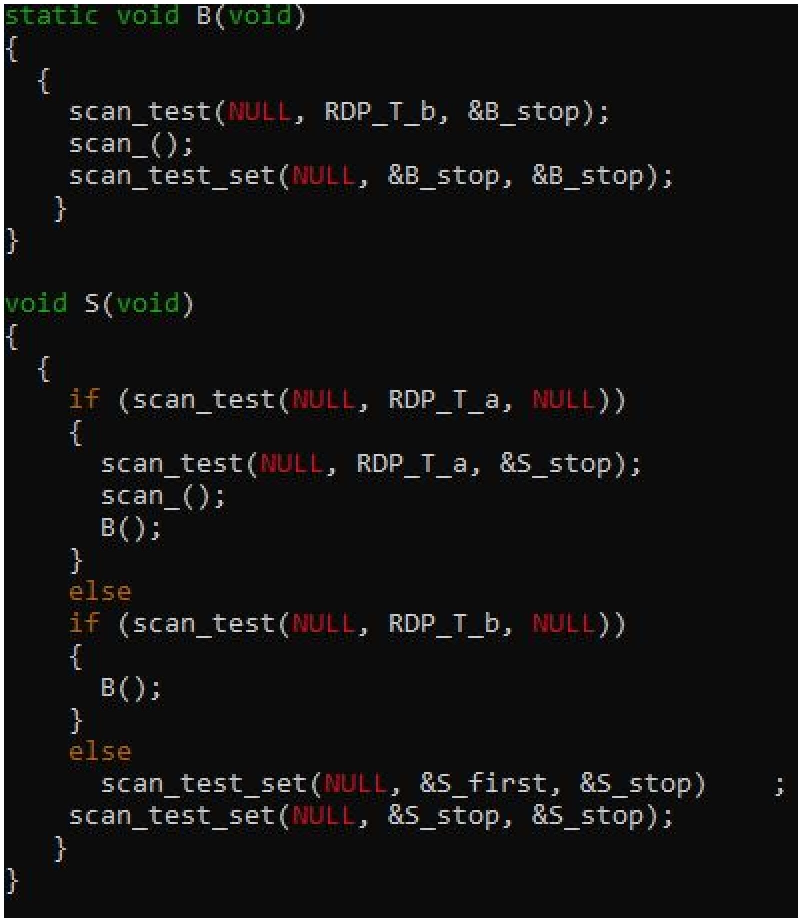
We also must change “basic.str”.

```
a b
```



```
./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.



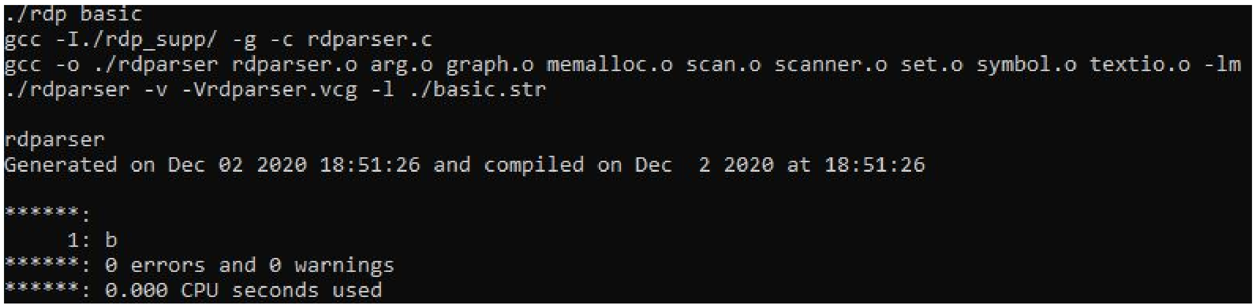
```
static void B(void)
{
  {
    scan_test(NULL, RDP_T_b, &B_stop);
    scan_();
    scan_test_set(NULL, &B_stop, &B_stop);
   }
}

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.

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.



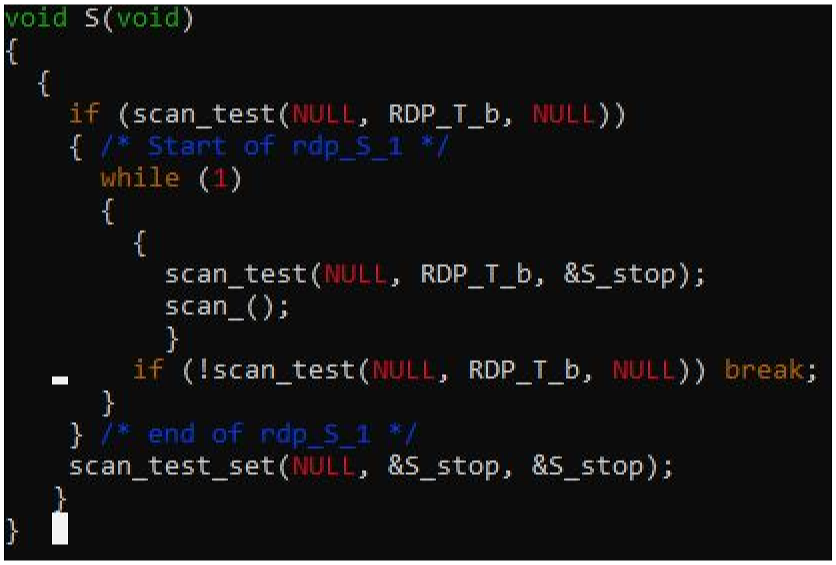
```
./rdp basic
gcc -I./rdp_supp/ -g -c rdparser.c
gcc -o ./rdparser rdparser.o arg.o graph.o memalloc.o scan.o scanner.o set.o symbol.o textio.o -lm
./rdparser -v -Vrdparser.vcg -l ./basic.str

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.

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:

```
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
}
```

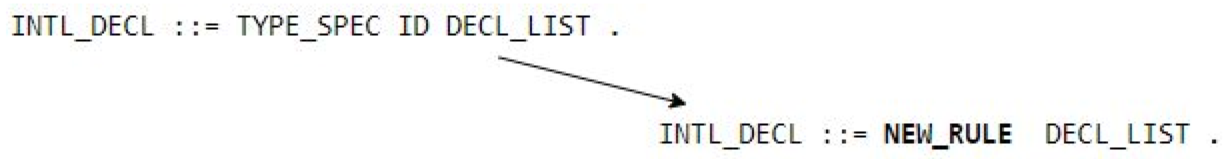
Between these two rules I could have avoided some “code duplication” by adding an additional rule to which derives type specifier followed by ID.

```
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 .
```

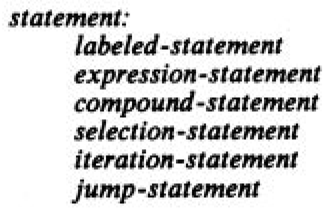
A compound statement can contain zero or more “internal declarations” or “statements”. An internal declaration is simply a declaration list which can be declared within a compound statement. The reason I ended up repeating myself here was due to the fact that I did not want to allow the declaration of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

```
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.

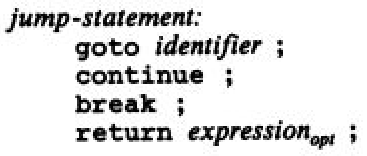


statement:
labeled-statement
expression-statement
compound-statement
selection-statement
iteration-statement
jump-statement

Figure 1. [24]

I opted to implement three types of statements:

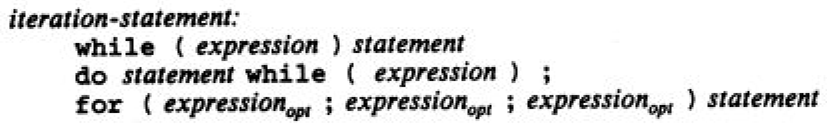
- “Jump statements”, which in this case only refers to **return** statements, but was planned to later encompass **break, continue and goto.**



```
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.

```
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.

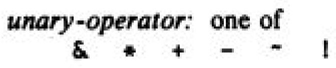

unary-operator: one of
& * + - ~ !

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.

```
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.

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.

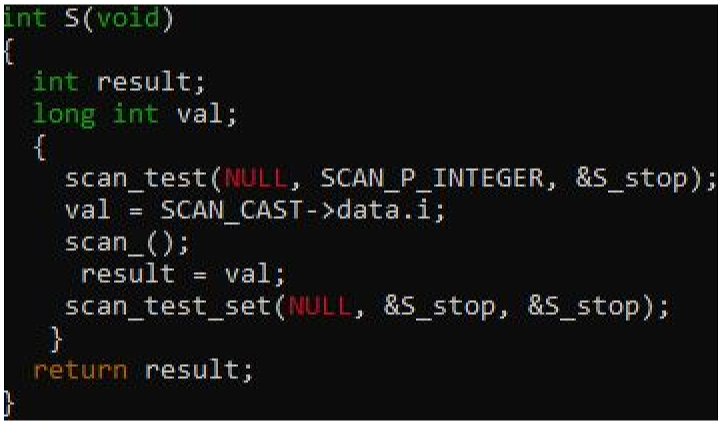
How this is done is much better explained through examples. Examine the following:

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’.

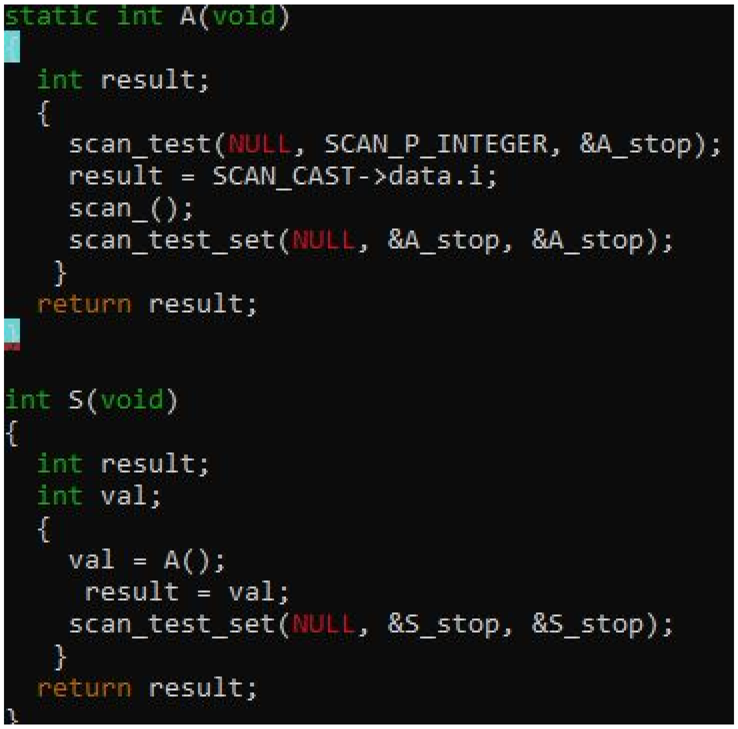
Inherited attributes are parameters of a parse function, so they allow data to be passed into a function, instead of data being returned to a function. I will include a usage example of this type of attribute in a later section.

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



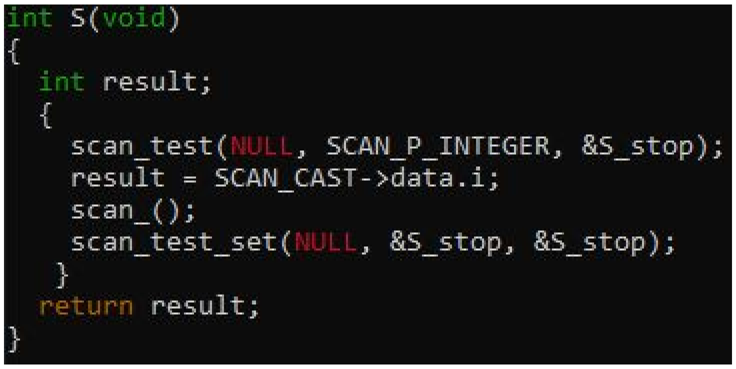
```
static int A(void)
{
  int result;
  {
    scan_test(NULL, SCAN_P_INTEGER, &A_stop);
    result = SCAN_CAST->data.i;
    scan_();
    scan_test_set(NULL, &A_stop, &A_stop);
   }
  return result;
}

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;
}
```

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

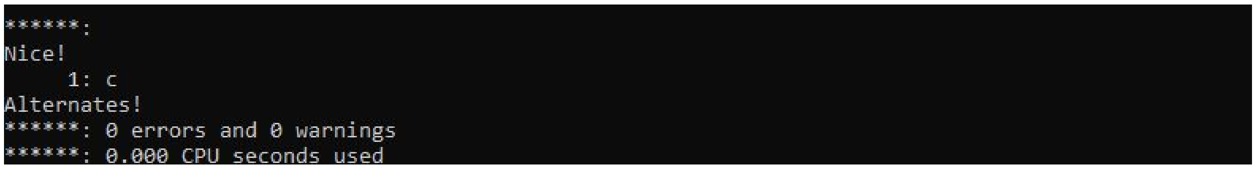
```
[* result = val; *]
```

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.

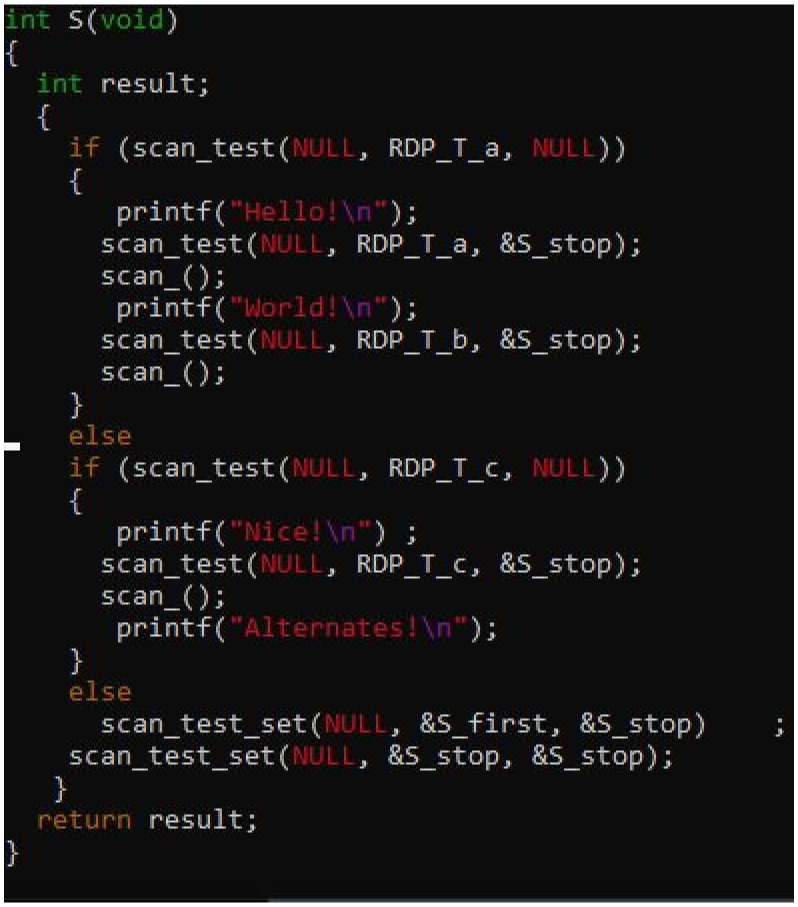
basic.bnf:

```
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
```



```
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.

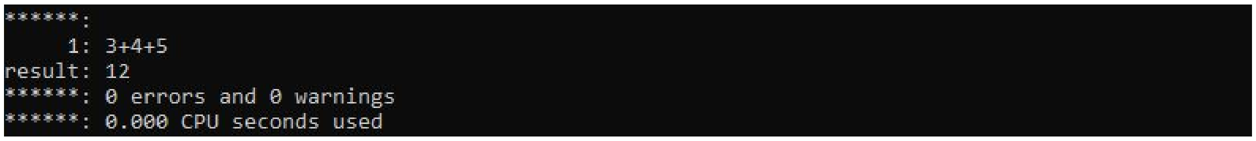
basic.bnf:

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

basic.str:

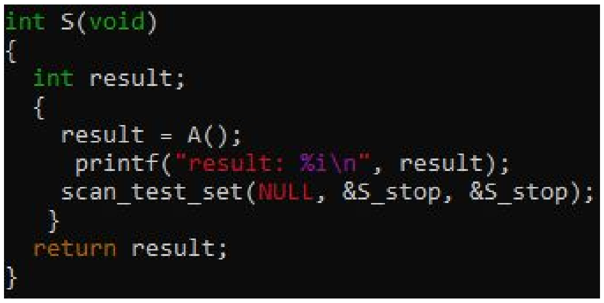
```
3+4+5
```

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

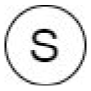


```
******:
     1: 3+4+5
result: 12
******: 0 errors and 0 warnings
******: 0.000 CPU seconds used
```

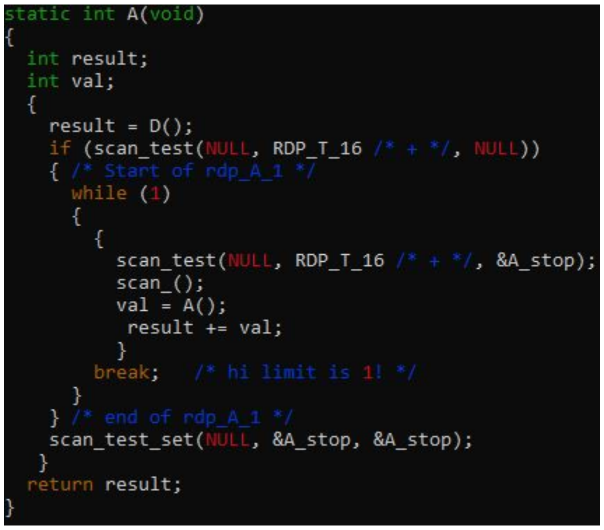
Now let's try to understand exactly how this is done.



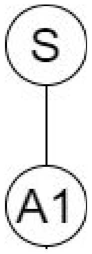
```
int S(void)
{
  int result;
  {
    result = A();
     printf("result: %i\n", result);
    scan_test_set(NULL, &S_stop, &S_stop);
   }
  return result;
}
```



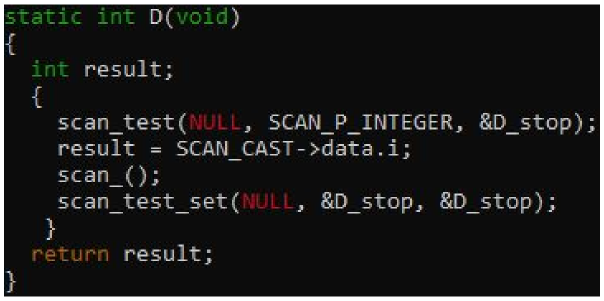
We begin in the parse function for the start symbol. ‘result’ is declared but as of yet has no value, thus we enter the parse function for A in order to obtain this value.



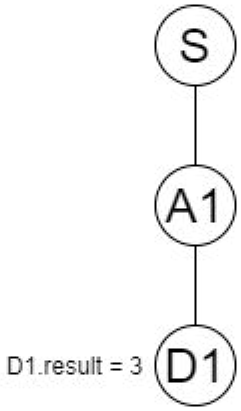
```
static int A(void)
{
  int result;
  int val;
  {
    result = D();
    if (scan_test(NULL, RDP_T_16 /* + */, NULL))
    { /* Start of rdp_A_1 */
      while (1)
      {
        {
          scan_test(NULL, RDP_T_16 /* + */, &A_stop);
          scan_();
          val = A();
           result += val;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



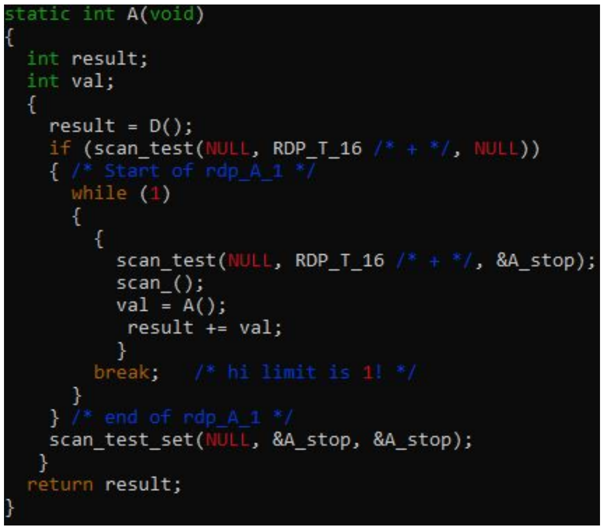
I will refer to this instance of A as A_1, where we begin by making a call to the parse function for D in order to obtain a value for the result of A_1.



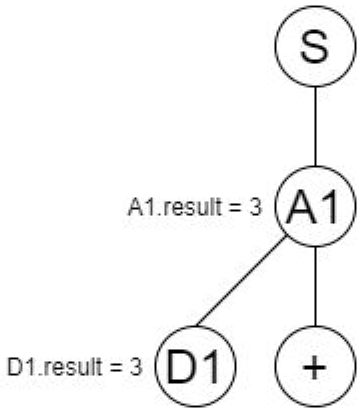
```
static int D(void)
{
  int result;
  {
    scan_test(NULL, SCAN_P_INTEGER, &D_stop);
    result = SCAN_CAST->data.i;
    scan_();
    scan_test_set(NULL, &D_stop, &D_stop);
   }
  return result;
}
```



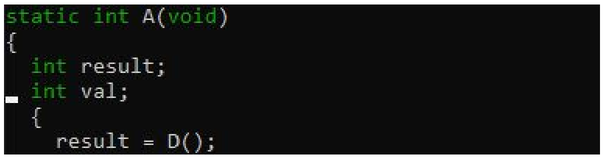
In this first call to D, we match our first look ahead token ‘3’ with INTEGER and subsequently set the value of 3 as our result which is then returned to A_1.



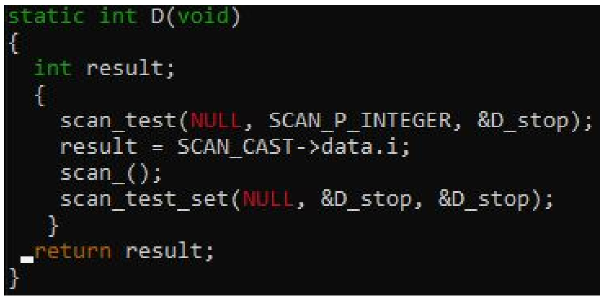
```
static int A(void)
{
  int result;
  int val;
  {
    result = D();
    if (scan_test(NULL, RDP_T_16 /* + */, NULL))
    { /* Start of rdp_A_1 */
      while (1)
      {
        {
          scan_test(NULL, RDP_T_16 /* + */, &A_stop);
          scan_();
          val = A();
           result += val;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



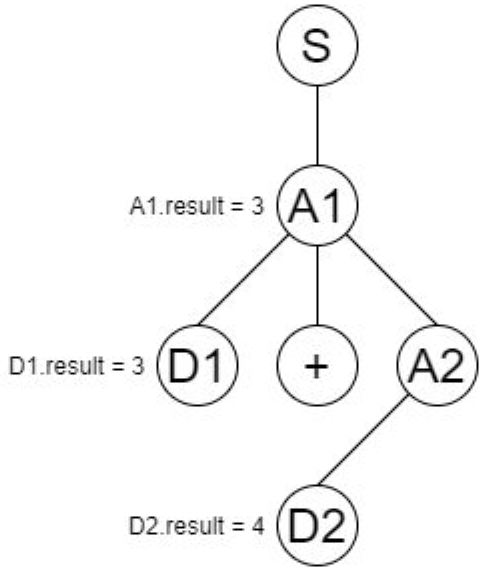
Returning to A_1 begins with 3 being assigned to the result of A_1. The current lookahead token ‘+’ is matched and A_1 makes a recursive call in order to determine a value for A_1.val.



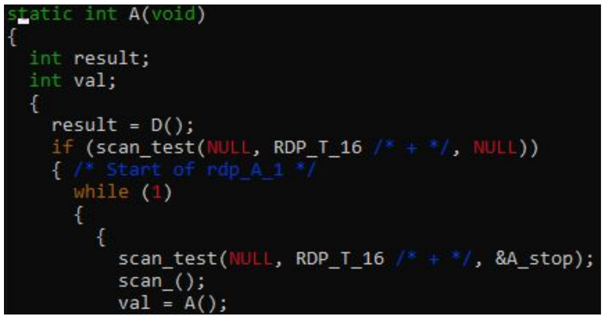
```
static int A(void)
{
  int result;
  int val;
  {
    result = D();
```



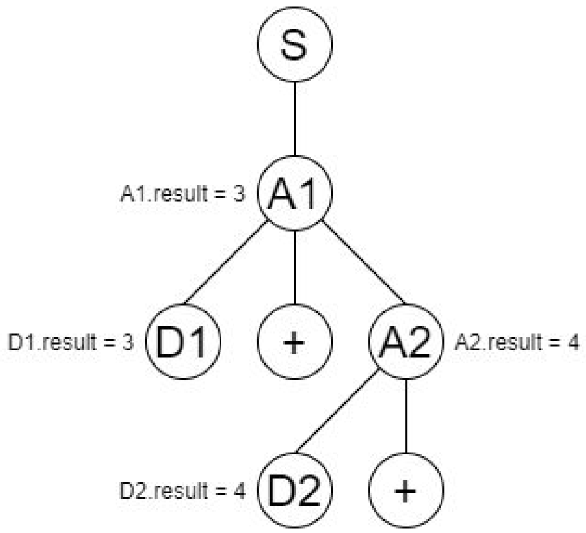
```
static int D(void)
{
  int result;
  {
    scan_test(NULL, SCAN_P_INTEGER, &D_stop);
    result = SCAN_CAST->data.i;
    scan_();
    scan_test_set(NULL, &D_stop, &D_stop);
   }
  return result;
}
```



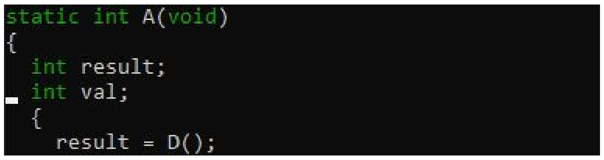
In A_2 the second call to D() is made, which matches with ‘4’ and returns the result to A_2.



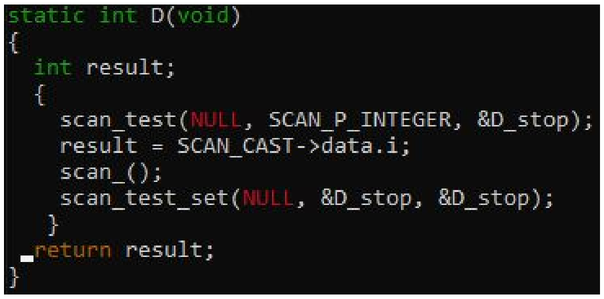
```
static int A(void)
{
  int result;
  int val;
  {
    result = D();
    if (scan_test(NULL, RDP_T_16 /* + */, NULL))
    { /* Start of rdp_A_1 */
      while (1)
      {
        {
          scan_test(NULL, RDP_T_16 /* + */, &A_stop);
          scan_();
          val = A();
```



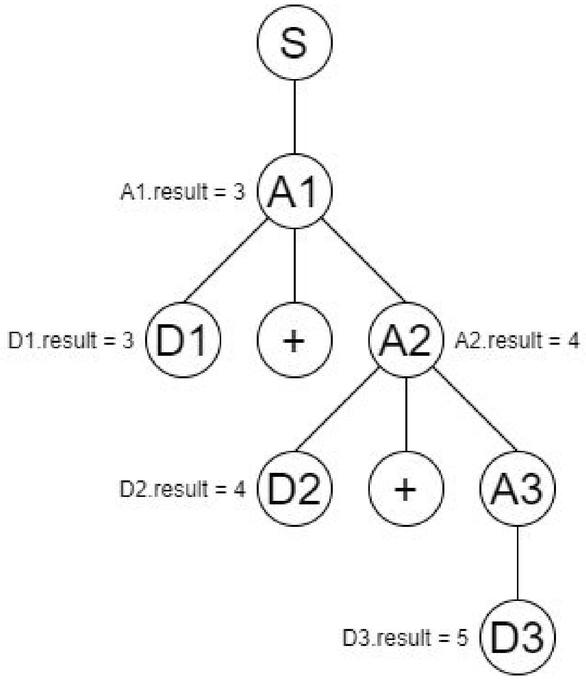
A_2 receives 4 as its result, matches ‘+’ and makes the third call to A()



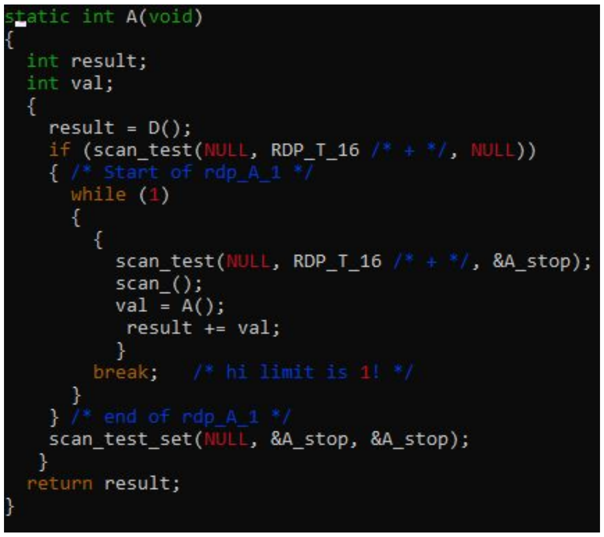
```
static int A(void)
{
  int result;
  int val;
  {
    result = D();
```



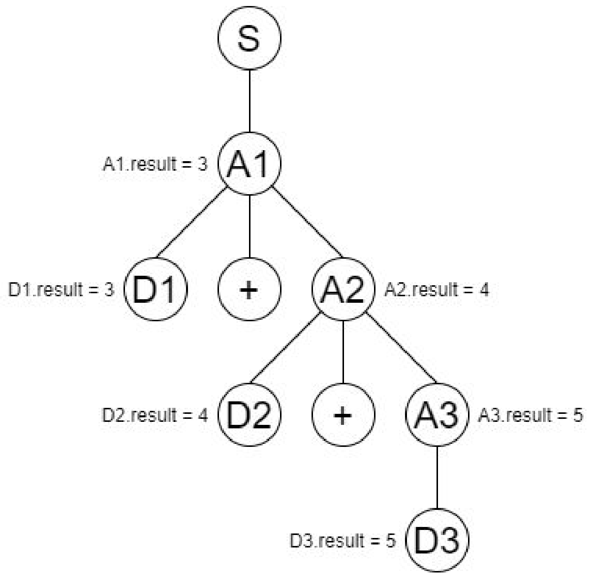
```
static int D(void)
{
  int result;
  {
    scan_test(NULL, SCAN_P_INTEGER, &D_stop);
    result = SCAN_CAST->data.i;
    scan_();
    scan_test_set(NULL, &D_stop, &D_stop);
   }
  return result;
}
```



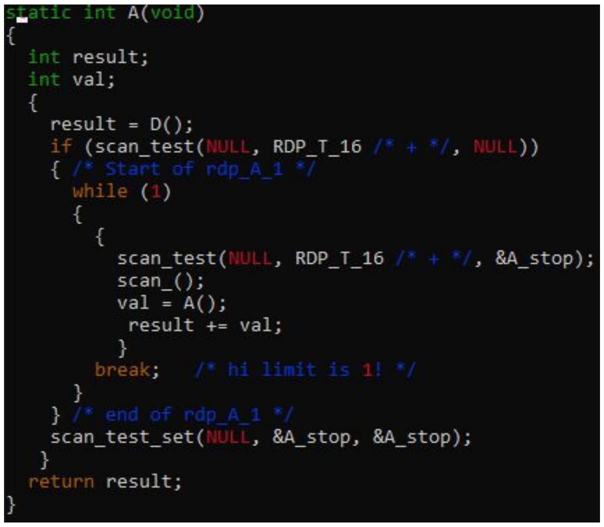
A_3 now makes the third and final call to D(), D_3 matches with ‘5’ and returns the result A_3.



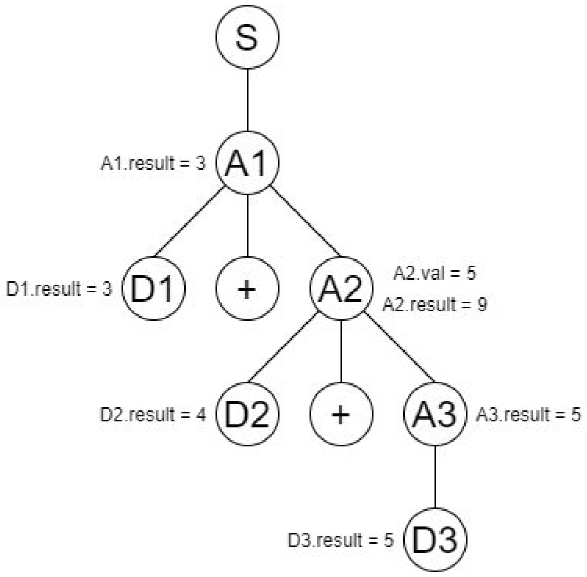
```
static int A(void)
{
  int result;
  int val;
  {
    result = D();
    if (scan_test(NULL, RDP_T_16 /* + */, NULL))
    { /* Start of rdp_A_1 */
      while (1)
      {
        {
          scan_test(NULL, RDP_T_16 /* + */, &A_stop);
          scan_();
          val = A();
           result += val;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

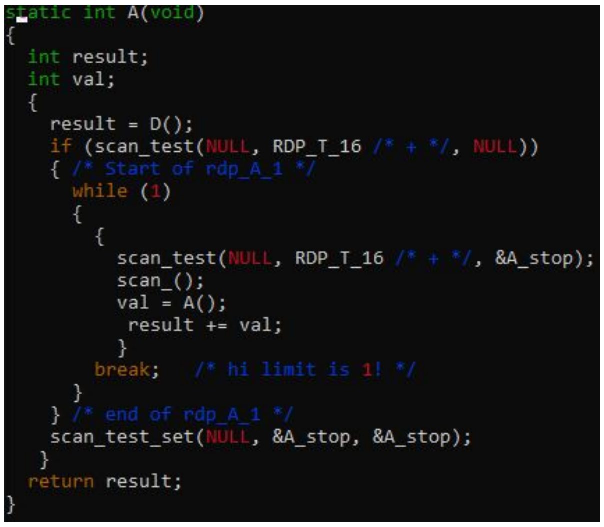


```
static int A(void)
{
  int result;
  int val;
  {
    result = D();
    if (scan_test(NULL, RDP_T_16 /* + */, NULL))
    { /* Start of rdp_A_1 */
      while (1)
      {
        {
          scan_test(NULL, RDP_T_16 /* + */, &A_stop);
          scan_();
          val = A();
           result += val;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

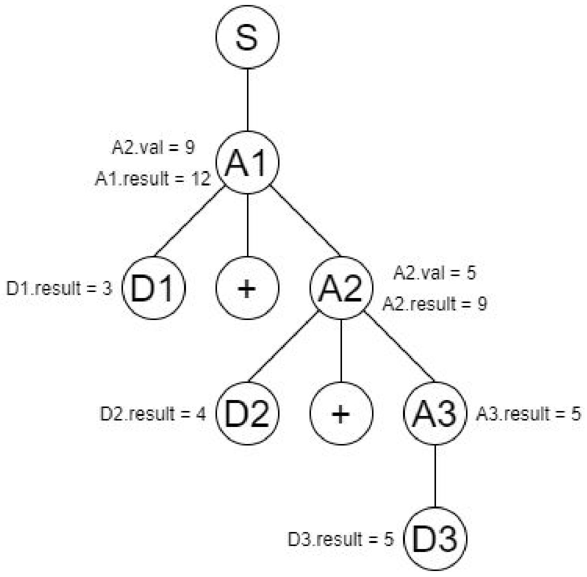


A_2.val now holds the value 5 which is then added to A_2.result, making its result now equal to 9. The while loop hits the unconditional break and A_2 returns the result of 9 to A_1.

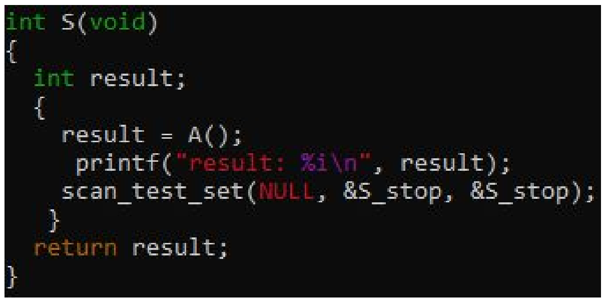
Note: while loops created from square brackets in the grammar only ever make one single iteration.



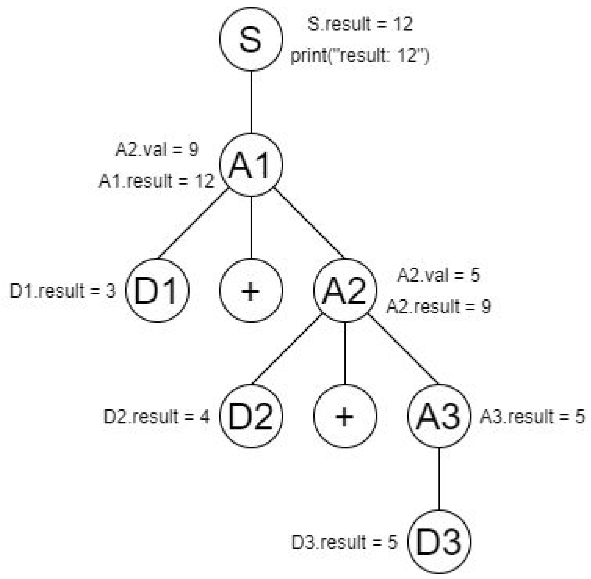
```
static int A(void)
{
  int result;
  int val;
  {
    result = D();
    if (scan_test(NULL, RDP_T_16 /* + */, NULL))
    { /* Start of rdp_A_1 */
      while (1)
      {
        {
          scan_test(NULL, RDP_T_16 /* + */, &A_stop);
          scan_();
          val = A();
           result += val;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



```
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.

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.

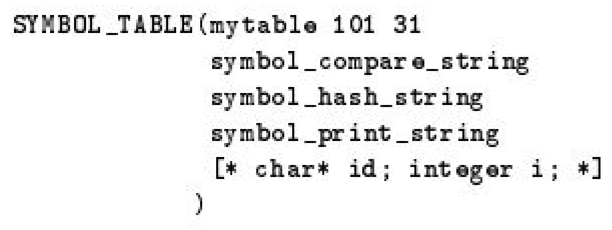
Compare - The name of a compare function, typically `‘symbol_compare_string’`

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 4. [30]

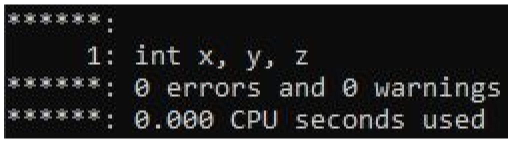
So this is a symbol table named “mytable” which contains 101 buckets and hases entries using the value 31. It uses the set of functions provided by rdp, and has data fields in which the string used to refer to the identifier, and the variable integer value associated with the identifier are both stored.

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));
         *]
        }
      ].
```

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”.

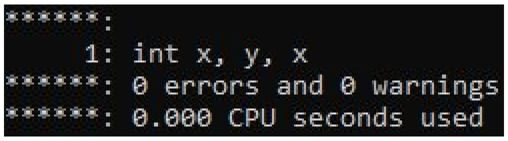
char *str - A reference to the string which is the name of the identifier. In this case these are x, y and z as previously stated.

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
```

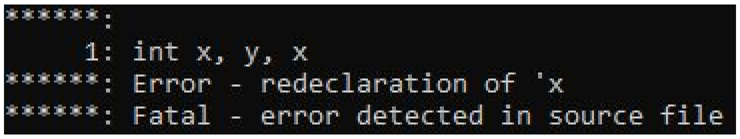
x has been redeclared in this case, but no error is thrown. Clearly the semantics must be modified to account for this.

<u>basic.bnf</u>

```
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));
                        }
                     *]
       }
      ].
```

<u>Basic.str</u>

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

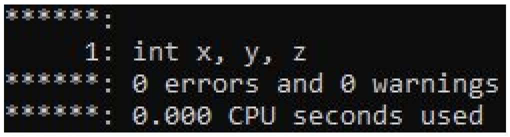


```
******:
     1: int x, y, x
******: Error - redeclaration of 'x
******: Fatal - error detected in source file
```

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

<u>Basic.str</u>

```
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.

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.

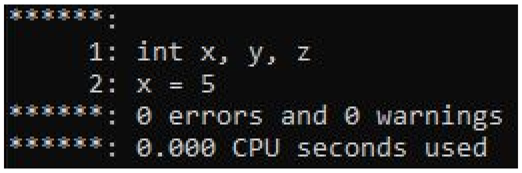
So these semantics stop identifier redeclarations, but what if we wanted to stop the use of undeclared identifiers.

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;
```

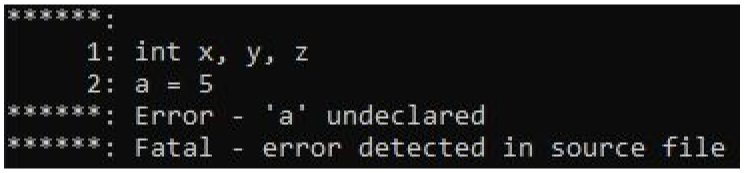


```
******:
     1: int x, y, z
     2: x = 5
******: 0 errors and 0 warnings
******: 0.000 CPU seconds used
```

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.

```
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?

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
```

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
```

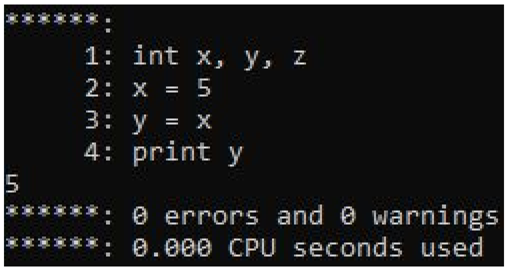
Now we understand the semantics attached to primitive, we know the value to be assigned here is either that of an integer literal or some integer stored in an existing variable.

```
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
```

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.

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**.

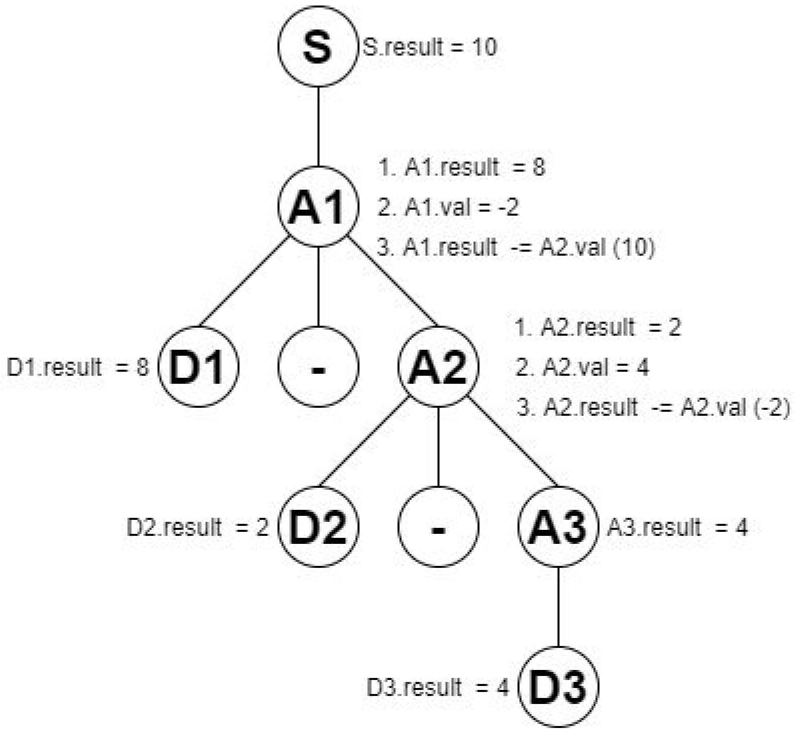
```
SUM:int          ::= PROD:result
                     { '+' PROD:val [* result += val; *] |
                       '-' PROD:val [* result -= val; *]
                     }.
```

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.

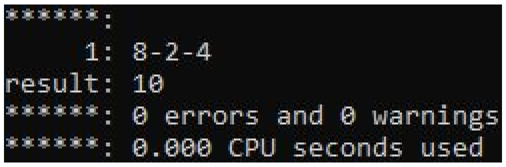
basic.bnf:
```
S:int ::= A:result [* printf("result: %i\n", result);  *].
A:int ::= D:result [ '-' A:val [* result -= val; *]].
D:int ::= INTEGER:result.
```

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
```

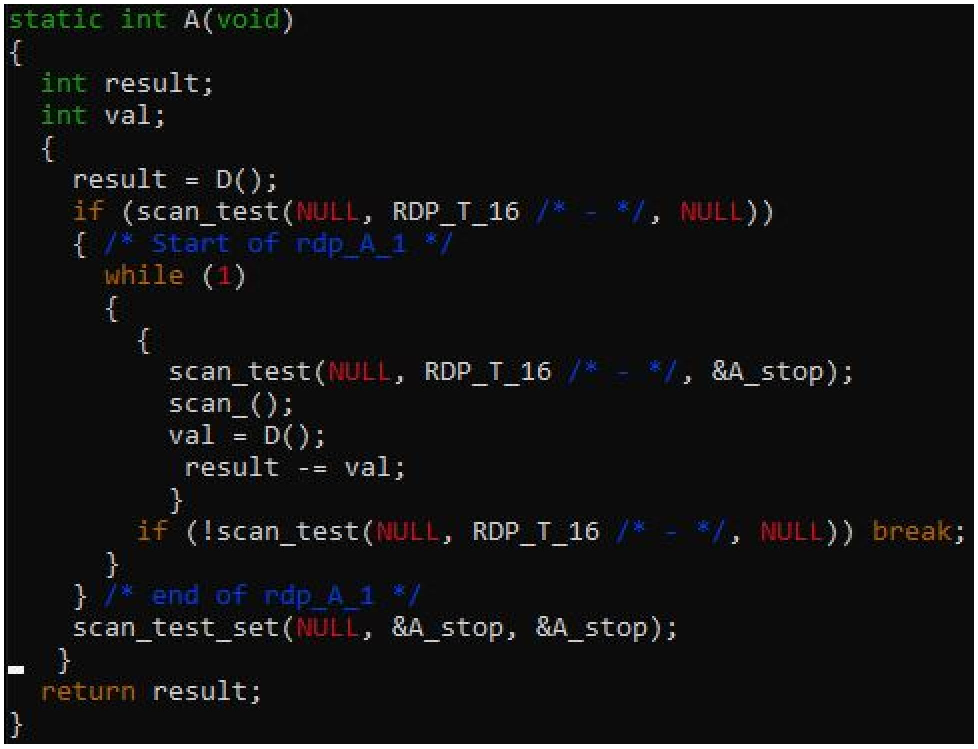
basic.bnf:

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

basic.str:

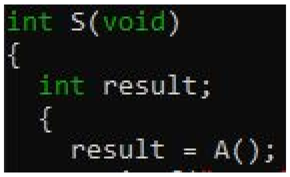
```
8-2-4
```

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.

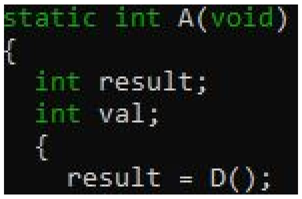


```
static int A(void)
{
  int result;
  int val;
  {
    result = D();
    if (scan_test(NULL, RDP_T_16 /* - */, NULL))
    { /* Start of rdp_A_1 */
      while (1)
      {
        {
          scan_test(NULL, RDP_T_16 /* - */, &A_stop);
          scan_();
          val = D();
           result -= val;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

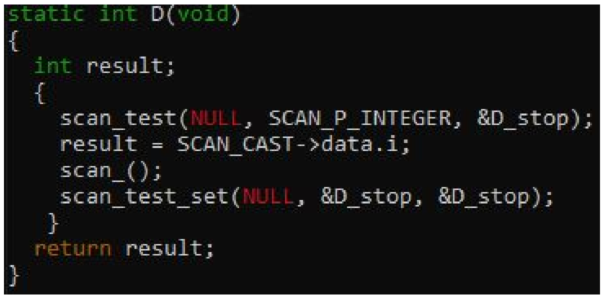
With this implementation, the result is initially set to 8 as before, however now result is updated through iterations of the while loop, thus 2 is first subtracted from 8 before 4 is subtracted from the result of 8 - 2, making the evaluation left associative. Let’s examine this process closer.



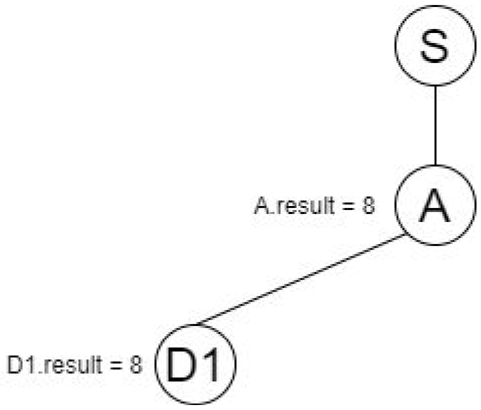
```
int S(void)
{
  int result;
  {
    result = A();
```



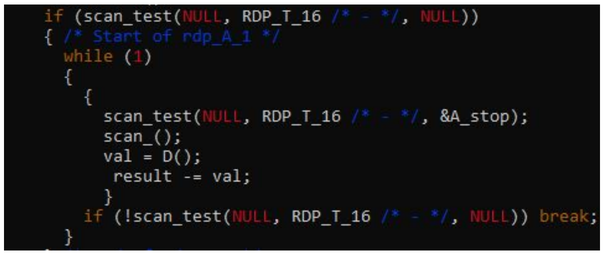
```
static int A(void)
{
  int result;
  int val;
  {
    result = D();
```



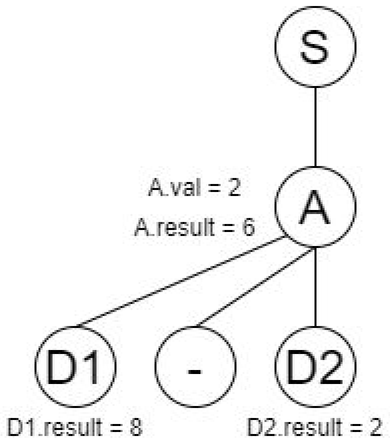
```
static int D(void)
{
  int result;
  {
    scan_test(NULL, SCAN_P_INTEGER, &D_stop);
    result = SCAN_CAST->data.i;
    scan_();
    scan_test_set(NULL, &D_stop, &D_stop);
  }
  return result;
}
```



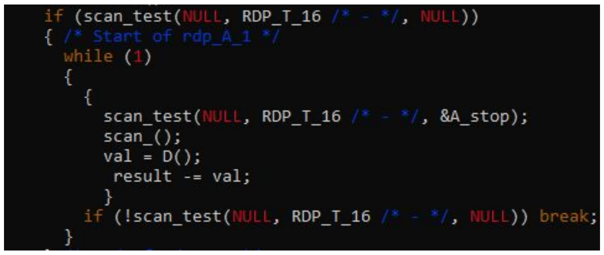
We begin with S calling A which makes its first call to D in order to acquire its initial result of 8.



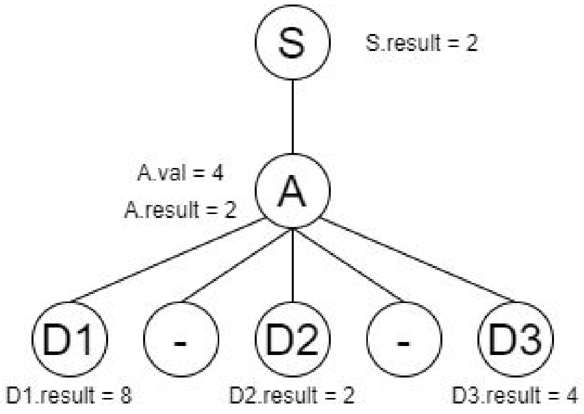
```
    if (scan_test(NULL, RDP_T_16 /* - */, NULL))
    { /* Start of rdp_A_1 */
      while (1)
      {
        {
          scan_test(NULL, RDP_T_16 /* - */, &A_stop);
          scan_();
          val = D();
           result -= val;
          }
        if (!scan_test(NULL, RDP_T_16 /* - */, NULL))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



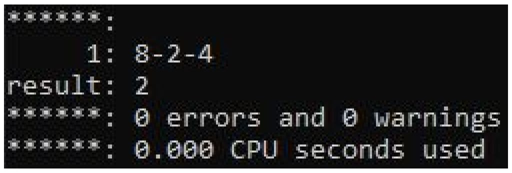
A then confirms the current lookahead token to be ‘-’ and enters it’s loop. D is called which returns the value 2 to A.val. A.val is subtracted from A.result, making A.result equal 2. A Test is performed to confirm that the current lookahead symbol isn’t ‘-’. The symbol is however ‘-’ so the loop goes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



```
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:

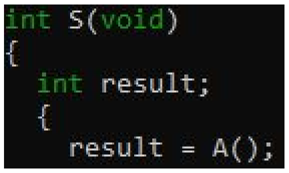
```
S:int ::= A:result [* printf("result: %i\n", result);  *].
A:int ::= B:result { '+' B:val [* result += val; *]}.
B:int ::= D:result { '*' D:val [* result *= val; *]}.
D:int ::= INTEGER:result.
```

basic.str:

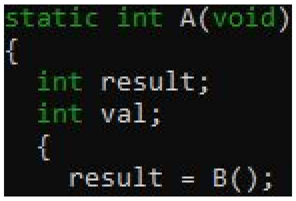
```
6+3*2
```

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.

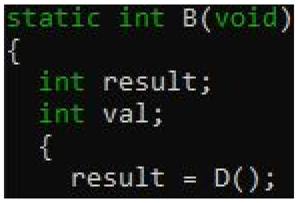
The idea behind how this works is that the parse functions which evaluate the subexpressions containing the higher priority operators are further down in the call order. This means that when the parse functions are returning, the subexpressions containing the higher priority operators are evaluated first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



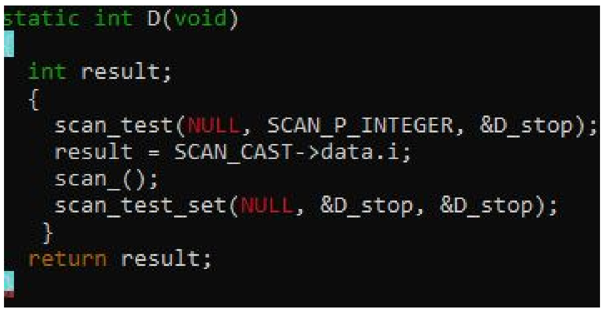
```
int S(void)
{
  int result;
  {
    result = A();
```



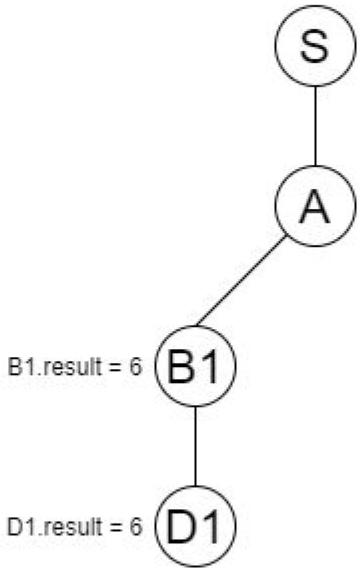
```
static int A(void)
{
  int result;
  int val;
  {
    result = B();
```



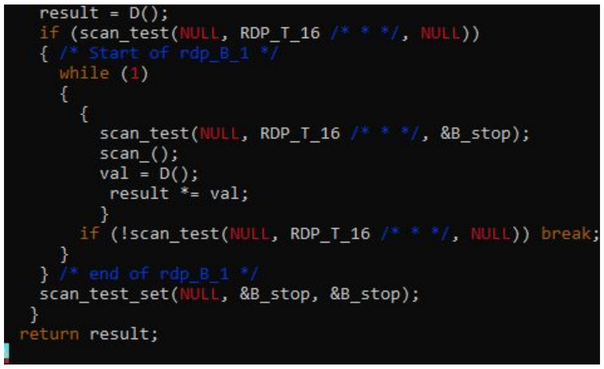
```
static int B(void)
{
  int result;
  int val;
  {
    result = D();
```



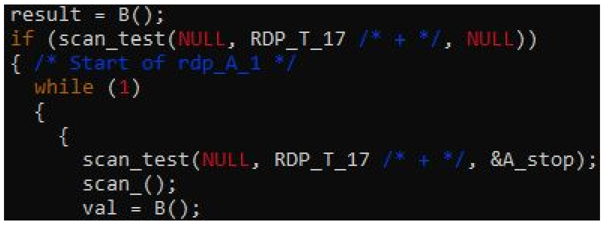
```
static int D(void)

  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;
```



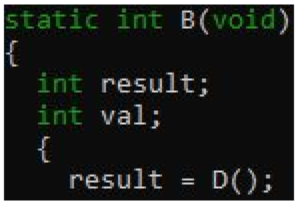
S calls A which calls B which calls D. D matches ‘6’ and returns the value to B.result.



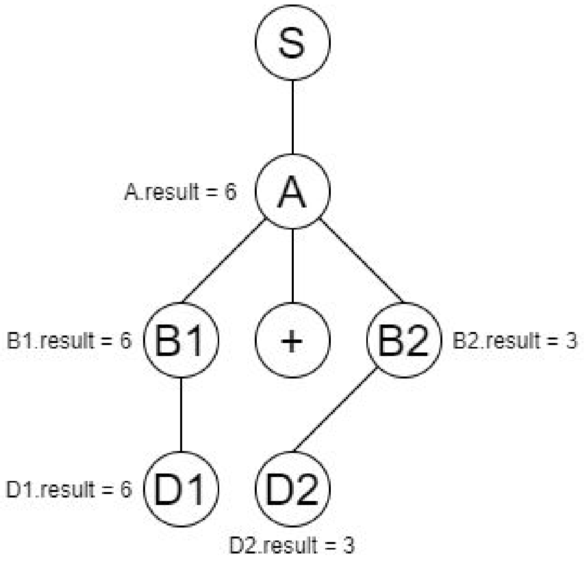
```
    result = D();
    if (scan_test(NULL, RDP_T_16 /* * */, NULL))
    { /* Start of rdp_B_1 */
      while (1)
      {
        {
          scan_test(NULL, RDP_T_16 /* * */, &B_stop);
          scan_();
          val = D();
           result *= val;
          }
        if (!scan_test(NULL, RDP_T_16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



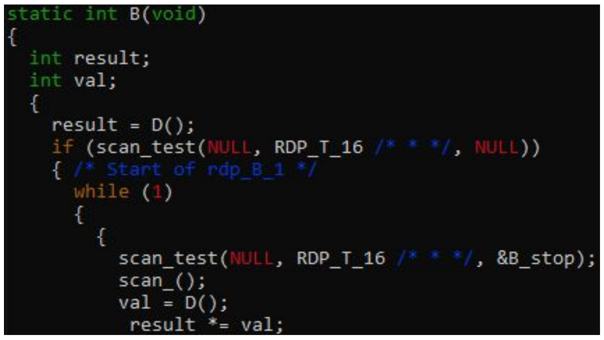
```
result = B();
if (scan_test(NULL, RDP_T_17 /* + */, NULL))
{ /* Start of rdp_A_1 */
  while (1)
  {
    {
      scan_test(NULL, RDP_T_17 /* + */, &A_stop);
      scan_();
      val = B();
```



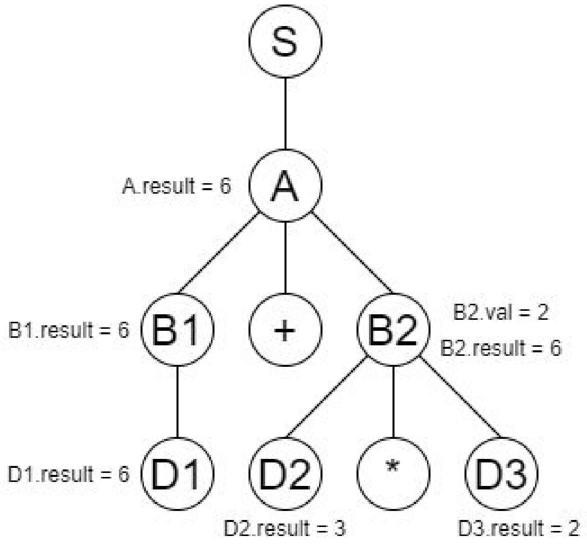
```
static int B(void)
{
  int result;
  int val;
  {
    result = D();
```



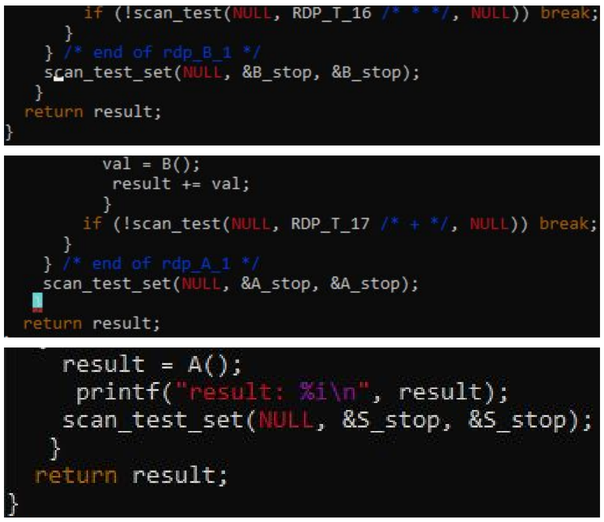
Because the current lookahead token isn't '*', B immediately returns 6 to A. The current lookahead token is '+' which A matches and enters its loop, resulting in A calling B once more. B calls D which matches '3' and returns the value to B.result.



```
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;
```



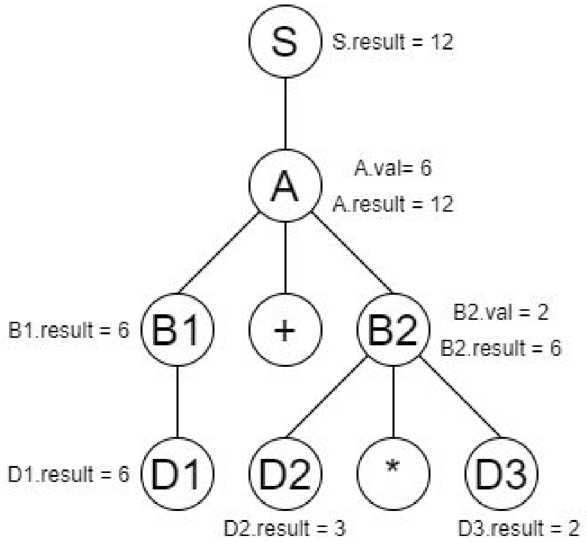
This time the lookahead token is '*' so B enters its loop and calls D. D matches with '2' and returns the result to B.val. B.result is multiplied by B.val making B.result equal to 6.



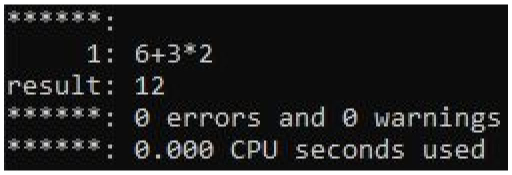
```
        if (!scan_test(NULL, RDP_T_16 /* * */, NULL)) break;
      }
    } /* end of rdp_B_1 */
    scan_test_set(NULL, &B_stop, &B_stop);
  }
  return result;
}
```

```
          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.



```
******:
     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 ‘)’.
```

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;
              *].
```

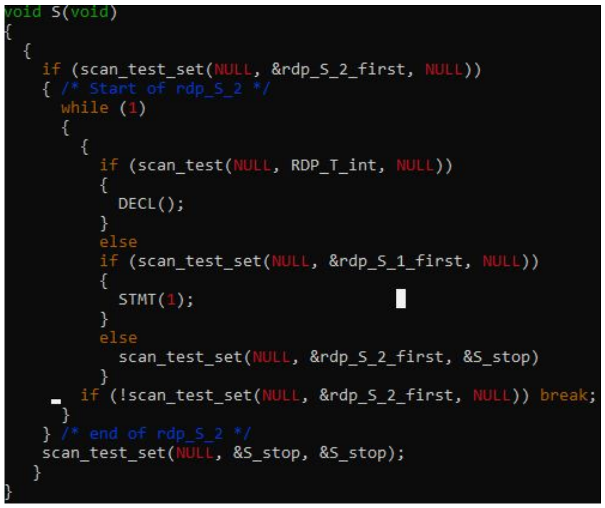
The above grammar has similar functionality to my current grammar. It includes variable declaration and value assigned, and includes checks to ensure only declared variables are used in statements and no variables are ever redeclared. This grammar includes both boolean and arithmetic expressions however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

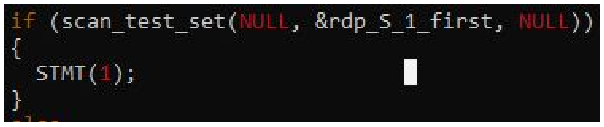
Statements can either be assignment statements, if statements or print statements. Both value assignment and printing can be subject to the condition of an if statement. This means that the semantics of these two statements will only be executed if the condition of the preceding statement is true.

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

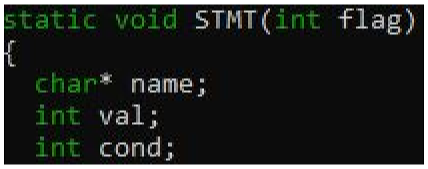
The value 1 is always passed into STMT whenever it is evoked from S.



```
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);
  }
}
```

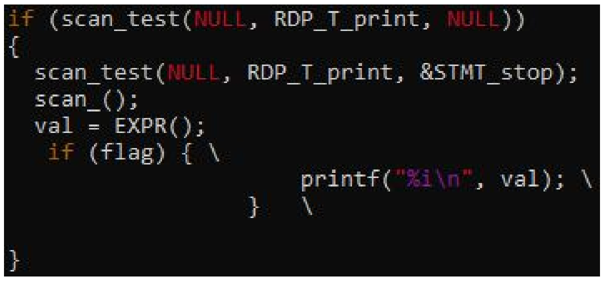


```
if (scan_test_set(NULL, &rdp_S_1_first, NULL))
{
  STMT(1);
}
```



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

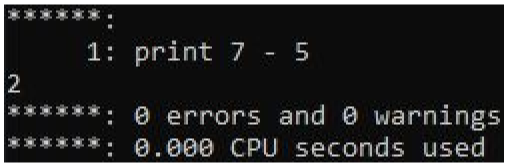
This is what is meant by inherited attributes, they are values which can be passed into a rule’s parse function.



```
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
```

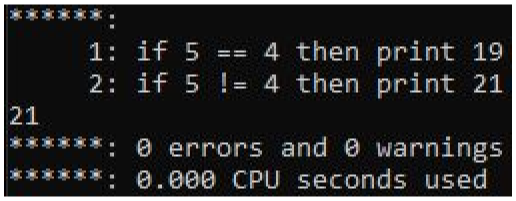
.str

```
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.

The Plan Moving Forward

My current grammar has some nice features and as demonstrated could be fairly easily extended with bracketed expressions and conditional statements. It is however not fit for my intended purpose due to the manor in which the semantic actions are implemented. If I continue in this fashion I will be creating a program which parses some subset of the C language which is subsequently compiled by an existing C compiler. I would effectively be placing a wrapper around C which reduces its functionality, clearly this is not ideal.

What a compiler typically does is convert a high level language such as C into some target machine architecture (MIPS, 6502, x86 etc...). Moving forward this will be my goal.

Currently I am considering MIPS as my target architecture. Originally I had intended to use x86 as this could be executed directly by my machine's processor, however upon receiving advice it seems it'd be unwise to attempt to work with such a complex instruction set, having had no previous experience with it. This leaves MIPS and 6502, both I have some experience with. Out of the two, MIPS is both more modern and easier to work with, so the current plan is to write a C compiler which compiles C code down to MIPS assembly, which I will then be able to run on an emulator such as SPIM.

Before the conversion to MIPS however there will be an intermediate step where the C source code will be converted to an intermediate language, three-address-code. Conversion to three-address-code will facilitate the following:

- Conversion to three-address-code will simplify the process of producing assembly code as the relationship between three-address-code and assembly is closer than that of assembly and high level languages.
- Conversion to three-address-code opens up the possibility of code optimization before the final conversion to assembly.
- Conversion to an intermediate language would significantly simplify things if I choose to have my compiler target multiple machine architectures.

rdp provides support for intermediate code generation through the “ma_aux” auxiliary file included in the software package. The details of how this is performed, I will likely explain in a future document where I can cover three-address-code and intermediate code generation in more depth.

Project Goals

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.[31]

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

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. ISO/IEC 9899:TC3:2007, Information technology - Programming languages - C