# AVL Trees in SASyLF

Sean McCook

smmccook@uwm.edu

7/14/19

Throughout the past semester, and into this summer, I have been working on my bachelor's thesis under the supervision of Dr. John Boyland. When I first approached Dr. Boyland regarding supporting me in this endeavor, I was unsure of the topic I wished to dive into, and I was unsure of what the format of a finished project would look like. After some deliberation about what I might be interested in, we began discussion of his work in SASyLf. SASyLF is a LF-based proof assistant specialized to checking theorems about programming languages and logics. At this point I was intrigued, and the list of potential projects in SASyLF that Dr. Boyland had compiled was taken into consideration. It then was mutually decided that through this thesis, I would gain experience in SASyLF and in the research of the overlap of programming languages and proof systems, and that after becoming acclimated to this new tool, one of the following areas of research would be chosen: Testing the SASyLF Software; using SASyLF to prove theorems motivated by a different class or project; or Implementing an extension in SASyLF. Although all three projects are difficult in their own right, I originally intended on choosing the most difficult one in implementing an extension in SASyLF. I quickly realized that this was perhaps a bit ambitious, and later on it was decided that I would work on the second project, using SASyLF to prove theorems motivated by a different class or project. The theorems would involve AVL trees, which I had learned about in my Algorithm Design and

Analysis class. When I made the decision to work on SASyLF, I had never previously heard of a proof assistant, nor was I very comfortable in writing proofs. In fact, I would have said that writing proofs was my weakest area of computer science or math. This is one of the main reasons why I decided to work with Dr. Boyland on proofs in SASyLF. It was the perfect opportunity to learn and improve in this area.

Learning SASyLF turned out to be a much different process than I originally anticipated. At the very beginning, I was under the impression that reading as many of the associated papers and resources as possible and reviewing as much material about proofs as I could, would be necessary to begin working. These resources included: everything that could be found at SASyLF.org and on github, such as the SASyLF syntax documentation and the paper "SASyLF: An Educational Proof Assistant for Language Theory" by Jonathon Aldrich, Robert Simmons, and Key shin; the Wikipedia about Lambda Calculus, and any other related term deemed important; and even the textbook from a class where I had previously worked on proofs. From these, I learned things such as the structure of a SASyLF file, the syntax of SASyLF, the design philosophy behind SASyLF, the motivation for its creation, and reviewed the basics of writing proofs.

A SASyLF file consists of: an optional package declaration, which I later learned currently serves no purpose; a list of terminals; a description of abstract syntax; a declaration of judgments, and the rules for each; and theorems, or lemmas. Before embarking on this project, I was not at all familiar with formal grammars in respect to programming languages and logic, nor with the concept of terminal versus non-terminal. Through this initial research I learned that SASyLF uses BNF grammar for its abstract syntax, and that a grammar is a set of production

rules that describe all possible strings in a given formal language. I learned that production rules are simple replacements, and terminal and nonterminal symbols are the lexical elements used in specifying the production rules. Terminal symbols are the elementary symbols of the language defined by a formal grammar, and in SASyLF they are used to parse expressions. Nonterminal symbols are replaced with sequences of terminal and nonterminal symbols according to the production rules. Backus-Naur form (BNF) is a notation technique for context-free grammars used by SASyLF. In BNF, production rules are written of the form <symbol> ::= __expression__, where <symbol> is a nonterminal, and the __expression__ consists of one or more sequences of symbols. Multiple sequences of symbols are separated by a pipe '|', indicating a choice, where the sequence chosen is substituted for the symbol on the left. Terminals are symbols that never appear on the left side, while nonterminals are the symbols that do. This is how abstract syntax is defined in SASyLF. In SASyLF a judgment is declared with a name, and a syntax production. An axiom is a statement, or proposition which is regarded as being established, accepted, or self-evidently true. After each judgment declaration, are the rules or axioms that give the definitive rules for generating the judgment's associated facts. Rules are defined with premises above the lines, and conclusions below. Theorems and lemmas are statements which have proofs. They also have inputs, which in SASyLF are preceded by the keyword "forall", and they have outputs, which are preceded by the keyword "exists". The output of a theorem is always a judgment. The inputs consist of either syntax variables, or named judgments, and the output consists of what the theorem proves. The proof itself is a series of statements of the form "name : judge by justification", where name is the identifier by which the statement might later be referred to, judge is the judgment we have proved, and

the justification is the reason for which we believe the statement to be true. The simple design philosophy of SASyLF is that: language and logic syntax, semantics, and meta-theory should be written as closely as possible to the way it is on paper. SASyLF is meant to be a tool for teaching formal language theory that provides immediate feedback. It is specialized for reasoning about languages, programs, and logics, or any system with variable binding.

Though learning these things was helpful in getting acclimated, similarly to learning a programming language, actually using SASyLF proved to be the most effective tool. Ultimately, the small amount of base knowledge gained through the initial research pales in comparison to the amount and speed at which I learned by actually using SASyLF to write proofs. Therefore, to anyone who might be interested in learning to use SASyLF, I would recommend swimming through the intimidation, and just diving in. After installing SASyLF, to begin the process of actually writing proofs, we started with the example file "sum.slf". This file, written by Jonathon Aldrich, consists of the formalization of integers and addition through its definition of natural numbers in terms of the nonterminal 'n' as either zero, the terminal 'z', or the successor 's', of a natural number 'n'. In BNF this is equivalent to "n ::= z | s n". The file then defines judgments for sum, and less than, and uses them to prove a few simple introduction theorems leading into a more interesting inductive proof that addition commutes. All of which demonstrates the SASyLF language in a tutorial style.

The first theorem is a perfect example of what writing proofs will look like and serves as an introduction into the language. This theorem states that any number plus one equals the successor of that number. It is defined as follows: "theorem n_plus_1_equals_s_n : forall n exists (s (z)) + n = (s n)" where one is represented as the successor of zero, and the parentheses

are optional. Since the form of this statement is declared by the judgment “sum: n + n = n”, the desired output must be obtained using its rules. The two rules are defined as follows:

```
--------- sum-z
z + n = n

n1 + n2 = n3
-------------------- sum-s
(s n1) + n2 = (s n3)
```

The proof for this theorem is very quick and to the point, nearly regurgitating these rules.

```
theorem n_plus_1_equals_s_n : forall n exists (s (z)) + n = (s n).


d1: (z) + n = n       by rule sum-z
d2: (s (z)) + n = (s n)    by rule sum-s on d1

end theorem
```

It is followed by a number of simple proofs intended to demonstrate the auto-solve feature. As this feature is laden with bugs, and since it defeats the purpose of SASyLF, in the sense that it is not conducive to learning, these proofs are perfect to be solved manually as a first introduction to the tool. After having worked through the example proof, I was easily able to solve the auto solve proofs on my own. From there, the next two proofs in the file demonstrate one of our most important tools, induction. Induction is a case analysis on an inductive term that must be an explicit input of the theorem and is similar to recursion. It can only be done one time per theorem, and at the top-level. There is one case for each rule that could be used to produce the indicated judgment. We can do induction on either syntax, or a derivation, as long as it is part of the input of the theorem.  A syntax case is of the form “case

<syntax> is <derivation> end case", where <syntax> refers to a case of the syntax we are doing induction over. There is one case for each possible syntax.

A rule case is of the form "case rule <rule> is <derivation> end case", where <rule> refers to the rules associated with the judgment matching the form of the derivation that we did induction on. Variables defined in the syntax of the case, or judgments defined in the premises of the case rules, are bound and should be used in the derivation. They are often the building blocks of the induction hypothesis. The inductive hypothesis is a justification on inputs that match the theorem, and whose judgment also matches the output of the theorem. It is often necessary to construct an inductive proof, but it can be applied regardless of whether or not induction was used. Another important justification for our derivations is previously defined theorems or lemmas. They can be used just like a rule by giving a syntax or judgment for each of the inputs of the theorem. These are the basic parts of the SASyLF language introduced by the sum.slf file and marks the beginning of my learning process. From this point on we began to work on our own set of theorems, this is indicated on line 405 of the accompanying AVL.slf.

The first theorems we wrote were increasingly more complex. They included proofs that addition is total, that zero is less than the successor of any natural number, and that addition is associative. In the process of writing the addition is associative proof, another tool was introduced. At this point Dr. Boyland showed me how to use where clauses, which are a very helpful tool that makes variable substitution extremely clear. In order to enable the requirement of where clauses in the eclipse plug-in, one can click on the "window" drop down menu in eclipse, then on SASyLF preferences, and then click the box to do so. Where clauses are very easy, and intuitive to use. Whenever a substitution is made where a variable x is

substituted for its equivalent y, when this feature is enabled, there must be a where clause on a following line of the form “where x := y”. If multiple variables have been substituted, such as in a case rule of a case analysis, they may be included in the where clause using the keyword “and”. Where clauses can also be used optionally without enabling their requirement.

After having written these slightly more complex theorems, in order to be able to solve more powerful things, we first defined a new judgment formalizing equality. This judgment consists of a single rule with a conclusion of the form “n=n”. Which we then used to write a theorem that proves the equality of the sums of two equal numbers. From there, to continue on the path of providing ourselves with a more powerful toolset of theorems, we began to work on proving contradictions and proving things with contradictions. The first being a theorem that proves a contradiction exists in having a natural number that is less than zero, remembering that natural numbers are the set of positive integers including zero. The statement of this theorem, and it’s accompanying proof is declared as follows:

```
theorem nothing-lt-z: forall d1: n < z exists contradiction

proof by induction on d1:

    case rule

        dst1: n < n3
        dst2: n3 < z
        --------------- less-transitive
        dsc: n < z

    is

        proof by induction hypothesis on dst2
    end case

end induction
```

```
end theorem
```

At first, I found the introduction of contradictions to be a bit confusing, but more so because I was unsure of the syntax rather than the concept. Through practice, I learned that the syntax is actually rather easy. Contradictions are very powerful and can be used in a couple different ways. You can write a theorem or lemma that proves that a contradiction exists, and you can prove things by contradiction. Proof by contradiction is a form of proof that establishes the truth or validity of a proposition by showing that assuming the proposition to be false leads to a contradiction. As seen in the theorem statement of the previous proof, when trying to prove a contradiction exists in SASyLF, the output of the theorem's statement is "exists contradiction." Contradictions can also be used as the justification for a derivation in the form, "by contradiction on *name*." But only if a case analysis on *name* needs no cases. This is often incredibly useful since when a contradiction can be shown, it serves as the final derivation of that particular case or proof.

We went on to use contradictions in both of these ways, increasing the complexity of each theorem as we progressed, and often stopping to write new theorems that were necessary to complete the ones I was working on. Through this process, I wrote a few very useful contradiction theorems that I used frequently along the way. These theorems consisted of proofs such as the anti-symmetric proof, which at the time I really struggled to write, that shows a contradiction where a number is both greater than and less than another. And the less than contradict proof, that shows a contradiction where a number is less than itself. Both turned out to be incredibly useful in my later work on AVL trees.

At this point I was feeling pretty comfortable with SASyLF and writing proofs. I had just encountered the most difficulty of the entire process so far, where I had gotten really stuck writing the anti-symmetric and add-cancel proofs. Working through this showed me that I was capable of writing these proofs even though they were difficult. Looking back now these proofs seem easy, but it is a learning process. SASyLF has a learning curve, and some of these proofs can be very tough. I was almost ready to begin writing something much more interesting. Before this, there was one last tool left for Dr. Boyland to show me. Up until this point, every time I did a case analysis on a judgment, I manually wrote all of the case rules. Early on, I struggled with choosing proper variable names in writing the case rules, as I did not fully understand the purpose. Adding where clauses helped but writing the rules themselves was integral to the learning process. It lent to a much better understanding of what a case rule consisted of, and what I would want the variables to be. This last tool, or shortcut, is a way to automatically fill in, or write the case rule. To do so, first a case analysis or proof by induction must occur. After, but before typing out the cases, if the error marker on the line where the case analysis occurred is double clicked, a window will pop up with a prompt to automatically insert the cases for the analysis. Although this tool saves a lot of time and is incredibly useful in expediting the process of writing proofs in SASyLF, it often provides variable names that lead to confusion. Eventually, towards the end of this work, I realized that in order to most effectively use this shortcut and remove confusion, it was necessary to rename the variables that are automatically chosen, to those that make sense to me. After learning of this shortcut and completing some more difficult proofs, now that I was feeling more confident in using SASyLF, I was ready to start writing my own proofs motivated by another class.

AVL trees are self-balancing binary search trees. A tree is defined to be an AVL tree, only if the balance factor for every node in the tree has an absolute value of no larger than 1. In a binary tree, the balance factor of a node N is defined to be the height difference of its two child trees. If during a modifying operation such as insert or delete, a height difference of more than 1 arises between the child subtrees of any node, in order to maintain the AVL tree height-balance property, the subtree of this node has to be rebalanced. The given tools for rebalancing are left rotations, and right rotations. They can be performed in a single rotation or a double rotation sequence, which is either a left-right rotation, or a right-left rotation. To have an unbalanced tree, we need a tree with at least a height of two. In order to begin working on AVL trees in SASyLF, I needed to define some new syntax. To do so, I first declared some new terminal symbols: null, h, max, and AVL. And then formalized a syntax for binary trees of integers as "t ::= null | t n t". After doing so, I began to write the judgments that would be necessary to formalize AVL trees. These were judgments that defined: the max of two natural numbers; the height of a binary tree; the closeness of two numbers; the AVL property of a binary tree; tree equality; and tree insertion.

The max judgment is of the form "max n n = n" and consists of three rules, each with one premise. The first rule requires that the first number is less than the second, the second rule requires that the second number is less than the first, and the last rule requires that both numbers be equal. The number on the right-hand side of the equal sign is the max. As you would assume, in the case that the two numbers are not equal, the max is the larger of the two. This judgment is necessary to define the height of a binary tree, because the height of a binary tree is the successor of the larger height of its two subtrees. The max judgement is necessary to

determine the larger height of the two subtrees. Naturally, the height judgment comes next, with a syntax production of the form “h t = n”. It consists of two rules. One for when the tree is null, and one for when the tree is a normal node. In the case that the tree is null, the rule has no premise and the height of the tree is zero. In the case that the tree is a node with two subtrees, the rule requires three premises - the height of the first subtree, the height of the second subtree, and the max of both. Since the tree is one node taller than its subtrees, its height is the successor of the max of their heights. The next judgment is “close”, which has a production syntax of “n1 ~ n2”. The tilde represents that two natural numbers have a difference of no more than one and are therefore “close”.

The key part of what defines an AVL tree, is that the subtrees of any node in the tree must have heights that differ by no more than one or are “close”. Meaning that this “close” judgment is instrumental in defining the properties of an AVL tree. The close judgment is made up by three rules, each of which have no premise. A number is close to its successor, a number is close to itself, and the successor of a number is close to the number. Typically, an AVL tree is strictly defined as an ordered binary search tree. It is important to point out that in our definition of AVL trees, for simplicities sake, we are ignoring the ordering property. We made the decision to do so to ensure that the scope of this project was not too great for one semester. Although this is the case, the ordering property is later enforced through our insertion judgments.

Now that we have all the judgments necessary to define an AVL tree, the AVL judgment has a production syntax of “AVL t” and is made up of two rules. The first, which has no premise and a conclusion that a null tree is an AVL tree. The second, defines what it means to be an AVL

tree for non-null trees. It has five premises: that the left subtree be an AVL tree; the right subtree be an AVL tree; the height of the left subtree; the height of the right subtree; and that the height of both subtrees be close to each other. Next, I defined tree equality, which is rather straight forward. It has a production syntax of the form “t = t”, and one rule with no premise and a matching conclusion. The last judgment we added, which was necessary to begin doing anything very interesting with our AVL trees, is insertion. Inserting into an AVL tree is synonymous with maintaining the height-balance property of an AVL tree, an AVL tree is self-balancing. That made this by far the most difficult judgment to write. It actually needed to be rewritten twice since it requires an intimate understanding of how to rebalance AVL trees. Initially Dr. Boyland and I had hypothesized that inserting into an AVL tree would never require a double rotation to rebalance. We believed this to be true after a few initial tests of inserting into an AVL tree from scratch, but later, during our initial attempt at proving that AVL properties are preserved through insertion, we realized this may not be the case. This prompted a thorough exploration and testing of all potential insertion cases and their accompanying heights, which later confirmed our suspicions.

This testing and exploration consisted of analyzing the possible heights of each of the subtrees after insertion, where the height of the subtree on the opposite side of the insertion is always k. Through this testing, not only did we realize that the double rotation would be necessary, but we also encountered a height case that should be impossible when inserting into an AVL tree. This outlier case occurs after insertion when the subtrees of the tree which the new node was inserted to, are both of the height k + 1. For example, given the AVL tree t1 n1 t2, the insertion t1 n1 t2 + n2, and n1 < n2. Which means that we will be inserting on the right,

and the height of t1 is therefore k. And given that the height of t1 is k; since t1 n1 t2 is an AVL tree, the height of t2 must be either k-1, k, or k+1. Meaning that since the heights of t1 and t2 are close, and since the height of t2 can be at most k+1, the subtrees of t2 are of at most height k. Therefore, after the insertion of only one node, it is impossible for both the subtrees of t2 to be of height k+1. Through an extensive amount of testing on paper, I was never able to definitively prove this, but I was almost positive it was the case. Under this assumption, I was able to rewrite the initial insertion rules to include double rotation rules for insertion on each side. Though I later discovered a couple pesky typos in these rules, they were now more or less the final insertion rules. The insertion judgment consists of the syntax production "t + n = t" and has eight rules. The first two have no premise, and cover insertion into a null tree and the insertion of a duplicate node. To make things less complicated, we made the decision to not allow duplicate nodes in our trees. This leaves the other three insertion cases, each of which have two rules, one for insertion on each side. The three cases are: insertion with no rotation, insertion with a single rotation, and insertion with a double rotation. Each insertion rule is written carefully so that self-balancing is maintained.

Now having defined all of the judgments and syntax necessary, we set out to prove a couple of things using our definition of AVL trees. The first being that the AVL property is preserved through insertion, and the second that, insertion into an AVL tree is total. Meaning that given any AVL tree and any natural number, a tree will be obtained from inserting the natural number into the AVL tree. Rather than combining the two theorems, we opted to keep them separate for the increased power of being able to use them separately. In order to prove such complex things with our newly written syntax, I realized I would need more theorems

using our new judgments. The first of which would be proofs that max and height are total. Having previously only written proofs using established syntax and judgments, it was very exciting to begin using my own, and I really started to enjoy writing proofs in SASyLF. With these under my belt, I moved onto a much more powerful and profound theorem, height-increase. This height-increase theorem sought to prove that for every insertion into a tree, the height of the new tree must either be the height of the old tree, or the successor of the height of the old tree. This turned out to be my most difficult proof yet, spanning almost a thousand lines, and containing a little bit of everything that I had worked with previously. In order to solve this proof, multiple new theorems were written, and the theorem itself was even rewritten to include a new input. The theorem needed to be rewritten because after working most of the way through it, I came to a point where the final expected derivation was that of a contradiction that we did not yet have the information to prove. The solution to this was adding an additional input to the theorem, being that the tree we are considering is an AVL tree. The additional information provided by the associated rules of an AVL tree was necessary in showing the contradiction. Specifically, the close relation between the two subtrees of the AVL tree. The new supporting theorems that were written to help solve the height-increase theorem included extremely useful proofs that max, and height are unique. Meaning that the max of two identical numbers is always equal, and that the height of two identical trees are always equal. These two theorems produce equalities that are more often than not crucial in the effort to solve a proof.

An extremely useful tool in SASyLF for making variable equalities clear is inversion. In place of a derivation, one may use the syntax “use inversion of (rule) rule_name on name” if

the case analysis of name has only one case, and that rule has no premises. This derivation doesn't prove anything, but it performs unification of variables presumed by the case. Although inversion had been introduced early on, after working on AVL trees, I quickly realized how important it is to use inversion on equality judgments. If not done immediately, it often leads to confusion that can be detrimental to progress. This is the case because the equality judgment shows that there are two different names being used for an identical value, but unless inversion is used, these two names are not unified.

After having written new theorems on my own, using my own syntax and judgments, and using them to solve the behemoth that was the height-increase proof, I felt more than confident to tackle one of our final proofs. Being that the AVL insertion total proof was the harder of the two, I opted to approach it first. What I was not expecting, is that almost immediately we would encounter a number of SASyLF bugs within this proof that rendered it incompletable for the time being. The first bug being one that did not allow me to add necessary where clauses, and the second bug being one that did not allow me to perform a case analysis on syntax bound from the induction hypothesis. While this roadblock was in effect, I shifted my focus to the other final proof, that the AVL property of a tree is preserved through insertion. This was a very fun proof to write, repeatedly requiring the use of every proof we had written thus far using our own judgments and syntax. Having gained a lot of experience using these theorems in writing the height-increase proof, I easily breezed through the majority of the insertion cases. Where I encountered a bit of trouble was the two double rotation cases. In order to solve these final case rules, I needed to write a new theorem. This theorem was a useful one that proves that the close relation is symmetric. Using this close

symmetric theorem that I wrote, I was able to finally solve one of our final proofs and prove that the AVL properties are preserved through insertion, which in turn was very rewarding.

After the bugs were fixed that were preventing me from completing the final proof that insertion into an AVL tree is total, I immediately resumed work on it. This proof was very fun to write too. It felt like a puzzle, and the entire time spent working on it, it seemed as if I was just on the cusp of solving it. This theorem also required the use of all of the newer theorems we had written, and most interestingly required the use of nearly all of the contradiction theorems we had written months previously. The hardest thing about this theorem though, was that in order to complete it, I needed to prove that the outlier k+1, k+1 height case discussed earlier, was impossible. Doing so was extremely difficult, but equally rewarding. It took nearly as much time as, if not more than, solving the height increase proof. Which up until that point had been the biggest challenge yet.

To do this, it was necessary to split the case up into two supporting theorems. One in which the case arose after a left insertion, and one in which the case arose after a right insertion. Both theorems having inputs that outlined this case, and outputs of a contradiction.

```
theorem no-equal-height-after-left-insert:
    forall d1: AVL t1 n0 t2
    forall d2: h t2 = n2
    forall d3: t1 + n = t1'
    forall d4: AVL t1'
    forall d5: t1' = t11 n11 t12
    forall d6: h t11 = s n2
    forall d7: h t12 = s n2
    exists contradiction.

theorem no-equal-height-after-right-insert:
    forall d1: AVL t1 n0 t2
    forall d2: h t1 = n1
```

```
    forall d3: t2 + n = t2'
    forall d4: AVL t2'
    forall d5: t2' = t21 n2 t22
    forall d6: h t21 = s n1
    forall d7: h t22 = s n1
    exists contradiction.
```

After consulting with Dr. Boyland on how best to approach solving these supporting theorems, he recommended coming up with a common thread in each and factoring it out into a new theorem, so that the hardest part need not be done twice. In terms of a common thread, he suggested something very similar, if not identical, to that which I ended up using:

```
theorem common-case:
    forall d1: AVL t
    forall d2: t + n' = t1' n11' t2'
    forall d3: h t = s n
    forall d4: h t1' = s n
    forall d5: h t2' = s n
    exists contradiction.
```

After dedicating some time to it, I was intuitively able to use the common case theorem to prove the two earlier supporting theorems. But now I was stuck with a new problem, which was really the same problem as before. How do I solve the common case theorem, and prove that this is impossible? It took me much longer than I would like to admit, but through many, many hours, and different approaches I was finally able to put this case to rest, and prove that when the height of the subtree of a node in an AVL tree is k, after inserting into the opposite subtree of that node, it is impossible for the two subtrees of the subtree of which was inserted into, to have heights of both k+1. This is the case because in order for the tree with which was inserted into to be an AVL tree, since the height of the subtree that was not inserted into is of height k,

following the height balance property, the opposite subtree must either be of height k-1, k, or k+1. Meaning that the child subtrees of this opposite subtree, must have heights of k at the most. Making it impossible for both of them to be of height k+1 after the insertion of only one node. Proving so simply required a case analysis of the second input, which was of the form declared by our insertion judgment. Out of all of the proofs that I have worked on throughout this project, because of the circumstances, this is the one that I am most proud of.

In retrospect, as expected, I learned a lot about the source material of my bachelor's thesis. I am now an expert on insertion in AVL trees, and know more than most about SASyLF, but I also learned a lot about what it takes to do independent research, and about my ability to learn and improve in areas of weakness. I grew to really enjoy using SASyLF to write proofs and would definitely recommend the tool to others. The learning process was enjoyable, and although at times I felt very overwhelmed that I was not easily able to solve a proof, now I feel very confident in my ability to solve proofs using SASyLF and am proud of myself for embarking on this endeavor in a previous area of weakness. I would no longer call writing proofs my weakest area of math, nor computer science, and that in itself is rewarding. As SASyLF had previously only been used to prove things about natural numbers, type theory, and logic cut-elimination stuff, it was fascinating using it to prove things in a new area such as data structures. Given the chance to do so again, I would, and I would definitely recommend SASyLF to those interested in better learning how to write proofs.

## References


"Terminal and Nonterminal Symbols." *Wikipedia*, Wikimedia Foundation, 23 Nov. 2018, en.wikipedia.org/wiki/Terminal_and_nonterminal_symbols.

Aldrich, Jonathan. *SASyLF: An Educational Proof Assistant for Language Theory*, www.cs.cmu.edu/~aldrich/sasylf/.

Boyland, John. "Boyland/Sasylf." *GitHub*, github.com/boyland/sasylf/wiki.

Aldrich, J., & Simmons, R., & Shin, K. *SASyLF: An Educational Proof Assistant for Language Theory*

"Backus–Naur Form." *Wikipedia*, Wikimedia Foundation, 1 June 2019, en.wikipedia.org/wiki/Backus%E2%80%93Naur_form.

"AVL Tree." *Wikipedia*, Wikimedia Foundation, 30 May 2019, en.wikipedia.org/wiki/AVL_tree.

"Lambda Calculus." *Wikipedia*, Wikimedia Foundation, 27 May 2019, en.wikipedia.org/wiki/Lambda_calculus.