



The Graduate Institute of Sciences and Engineering

M.Sc. Thesis in Electrical and Computer Engineering

**GENERATING LIBRARIES OF HIGHLY CORRECT AND
SAFE FUNCTIONS BY USING PROOF-BASED FORMAL
METHODS**

by

Ayub Rokhman WAKHID

July 2014
Kayseri, Turkey

GENERATING LIBRARIES OF HIGHLY CORRECT AND SAFE FUNCTIONS BY USING PROOF-BASED FORMAL METHODS

by

Ayub Rokhman WAKHID

A thesis submitted to

the Graduate Institute of Sciences and Engineering

of

Melikşah University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical and Computer Engineering

July 2014
Kayseri, TURKEY

APPROVAL PAGE

This is to certify that I have read the thesis entitled “Generating Libraries of Highly Correct and Safe Functions by Using Proof-Based Formal Methods” by Ayub Rokhman WAKHID and that in my opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science in Electrical and Computer Engineering, the Graduate Institute of Science and Engineering, Melikşah University.

July 9, 2014 _____
Asst. Prof. Aytekin VARGÜN
Supervisor

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

July 9, 2014 _____
Prof. Dr. Murat UZAM
Head of Department

Examining Committee Members

Title and Name		Approved
Asst. Prof. Aytekin VARGÜN	July 9, 2014	_____
Assoc. Prof. Ahmet UYAR	July 9, 2014	_____
Asst. Prof. Mete ÇELİK	July 9, 2014	_____

It is approved that this thesis has been written in compliance with the formatting rules laid down by the Graduate Institute of Science and Engineering.

Prof. Dr. M. Halidun KELEŞTİMUR
Director

July 2014

GENERATING LIBRARIES OF HIGHLY CORRECT AND SAFE FUNCTIONS BY USING PROOF-BASED FORMAL METHODS

Ayub Rokhman WAKHID

M.S. Thesis – Electrical and Computer Engineering
July 2014

Assist. Prof. Aytekin VARGÜN

ABSTRACT

In this thesis, a specification library of highly correct and safe functions which can be used to compose large scale software is constructed. Although it is hard to prove the correctness and safety of large programs, it can be done if the correctness and/or safety of its components are shown beforehand. A similar approach is used in C++ where STL provides many useful small-scale procedures and functions. One of the strongest methods to verify software correctness is to use formal methods. In this research, Athena has been used as a theorem prover to write specifications of programs and prove that they are correct and safe with respect to the given specifications. The axiomatic definitions and the proofs are written in infix notation which makes them more readable and shorter with respect to the prefix notation. In Athena a program called CODEGEN can be used to extract code from the axioms and theorems. But it only works with the definitions in the prefix notation. In order to make it work with the infix definitions, a program that converts axiomatic definitions in infix notation to prefix notation is implemented in Java. The last contribution of this thesis is to define memory by using the infix notation to make it possible to generate code that updates memory locations.

Keywords: Formal Methods, Proof Assistance, Athena, Infix, Prefix, Software Safety and Correctness, Infix to Prefix Conversion.

İSPAT-TABANLI YAZILIM DOĞRULUĞU VE GUVENİLİRLİĞİNİ GÖSTERME TEKNİKLERİNİ KULLANARAK GÜVENİLİR BİR KÜTÜPHANE OLUŞTURMAK

Ayub Rokhman WAKHID

Yüksek Lisans Tezi – Elektrik ve Bilgisayar Mühendisliği
Temmuz 2014

Tez Yöneticisi: Yrd. Doç. Dr. Aytekin VARGÜN

ÖZ

Bu tezde, büyük ölçekli yazılım oluşturabilmek için kullanılacak yüksek düzeyde doğruluğa sahip ve güvenilir fonksiyonların formal tanımlarından oluşan bir kütüphane üretildi. Büyük programların doğruluğunu ve güvenliğini ispatlamak zor olmasına rağmen, eğer kendisinin bileşenlerinin doğruluğu ve/veya güvenliği önceden gösterilebilirse bu işlem yapılabilir. Benzer bir yaklaşım STL'nin pek çok kullanışlı küçük ölçekli prosedürler ve fonksiyonlar sağladığı C++'ta kullanılmaktadır. Yazılım doğruluğunu kontrol etmek için en güçlü yöntemlerden biri formal yöntemleri kullanmaktır. Bu çalışmada, programların özelliklerini yazmak ve onların tanımlanan özellikler açısından doğru ve güvenli olduğunu ispatlamak için Athena isimli bir teorem ispatlayıcı kullanılmıştır. Aksiyomatik tanımlar ve ispatlar prefix notasyonu ile karşılaştırıldığında daha okunabilir ve kısa ifadelerin yazılmasını sağlayan infix notasyonunda yapılmıştır. Athena'da CODEGEN adı verilen bir program, aksiyomlar ve teoremlerden kod üretmek için kullanılabilir. Fakat bu program sadece prefix notasyonundaki tanımlarla çalışır. Onun infix notasyonlarla çalışmasını mümkün kılmak için aksiyomatik tanımları infix notasyonundan prefix notasyonuna dönüştüren bir program Java'da yazılmıştır. Son olarak bu tezde infix notasyonu bellek tanımı yapılmış ve böylelikle bellekte güncelleme yapan programların üretilmesi sağlanmıştır.

Anahtar Kelimeler: Formal Yöntemler, İspat Yardımcısı, Athena, Infix, Prefix Güvenilirlik ve Doğruluk, İnfixten Prefixe Dönüştürme

DEDICATION

Dedicated to my parents for their endless support and patience during the forming phase of this thesis.

ACKNOWLEDGEMENT

I would like to express my gratitude to my supervisor Assist. Prof. Aytekin VARGÜN whose help, advices, encouragement and patience during all of the phases that we have passed together, made the publication of this thesis possible.

I express my thanks to my family for their understanding and motivation. Last but not least, I am thankful to Bilge Kağan Dedetürk for his support and all other colleagues and friends who make me spent a wonderful time in the university while working on this thesis.

TABLE OF CONTENTS

ABSTRACT	iii
ÖZ.....	iv
DEDICATION.....	v
ACKNOWLEDGEMENT	vi
TABLE OF CONTENTS	vii
LIST OF FIGURES	xi
LIST OF TABLES.....	xii
1. INTRODUCTION	1
1.1 STATEMENT OF THE PROBLEM	1
1.2 ORGANIZATION OF THE THESIS.....	3
2. SOFTWARE VERIFICATION	4
2.1 TESTING.....	5
2.2 FORMAL METHODS	6
2.3 PROOF ASSISTANTS.....	7
2.4 ATHENA THEOREM PROVER.....	10
2.4.1 Features of Athena	10
2.5 READABILITY OF ATHENA DEFINITIONS AND PROOFS	14
3. RELATED WORKS	16
4. METHODOLOGY AND CONTRIBUTIONS.....	18
4.1 ATHENA PREFIX AND INFIX LENGTH COMPARISON STUDY ...	19
4.2 EXTENDING LIBRARY IN ATHENA	19
4.2.1 Library hierarchy in Athena	19

4.2.2	New Infix Definitions of Some Functions	21
4.2.2.1	Mathematical Functions	23
4.2.2.2	Excel Formula Related Definitions	25
4.2.2.3	String Related Functions	28
4.2.2.4	Memory Related Definitions	31
4.3	DEVELOPING INFIX TO PREFIX CONVERTER PROGRAM.....	35
5.	COMPARISON OF PROOF AND DEFINITION LENGTHS IN PREFIX AND INFIX NOTATIONS	37
6.	CASE STUDIES	41
6.5	ATHENA PROOF OF FACTORIAL AND FACTORIAL EFFICIENT	41
6.6	DEFINITION OF MINLIST FUNCTION THAT RETURNS THE MINIMUM OF A LIST	46
7.	FORMAL DEFINITION OF THE SUBSTRING METHOD IN JAVA.....	50
7.1	STRING	50
7.2	SUBSTRING METHOD IN JAVA LANGUAGE	50
7.3	CORRECTNESS REQUIREMENTS IN ATHENA FOR JAVA SUBSTRING METHOD.....	51
7.4	SAFETY REQUIREMENTS IN ATHENA FOR JAVA SUBSTRING METHOD	54
7.5	FINALIZING THE CODE FOR CORRECTNESS AND SAFETY REQUIREMENTS	56
7.6	PROVING AN EXAMPLE SAFETY PROPERTY OF STRINGS.....	57
8.	CODE GENERATION	60
8.1	CONVERTING INFIX DEFINITIONS TO PREFIX FORM	61
8.2	CODE EXTRACTION	62
9.	CONCLUSION AND FUTURE WORKS	63
10.	REFERENCES.....	65

11.	APPENDIX A	67
A.1	PREFIX VERSION OF FACTORIAL AND FACTORIAL EFFICIENT FUNCTION.....	67
A.2	AXIOMATIC DEFINITION OF MINIMUM FUNCTION IN INFIX NOTATION WITH SAMPLE PROOFS	70
A.3	AXIOMATIC DEFINITION OF SUMLIST FUNCTION IN INFIX NOTATION WITH SAMPLE PROOFS.....	72
A.4	AXIOMATIC DEFINITION OF REMOVE FUNCTION IN INFIX NOTATION WITH SAMPLE PROOF	74
A.5	AXIOMATIC DEFINITION OF ACCESS FUNCTION IN INFIX NOTATION WITH SAMPLE PROOF	75
A.6	AXIOMATIC DEFINITION OF MEMORY WITH SAMPLE PROOFS IN INFIX NOTATION	76
A.7	AXIOMATIC DEFINITION OF INDEX FUNCTION IN INFIX NOTATION WITH SAMPLE PROOF	79
A.8	AXIOMATIC DEFINITION OF SPLIT FUNCTION IN INFIX NOTATION WITH SAMPLE PROOF	80
A.9	AXIOMATIC DEFINITION OF SUBLIST FUNCTION IN INFIX NOTATION WITH SAMPLE PROOFS.....	82
A.10	AXIOMATIC DEFINITION OF SUBSTRING FUNCTION IN INFIX NOTATION WITH SAMPLE PROOFS.....	84
A.11	AXIOMATIC DEFINITION OF MINLIST FUNCTION IN INFIX NOTATION WITH SAMPLE PROOFS.....	87
A.12	AXIOMATIC DEFINITION OF FACTORIAL FUNCTION IN INFIX NOTATION WITH SAMPLE PROOFS.....	89
12.	APPENDIX B	91
B.1	INFIX TO PREFIX MAIN CLASS	91
B.2	INFIX TO PREFIX LEXICAL CLASS	94
B.3	INFIX TO PREFIX STACK CLASS	101

B.4	INFIX TO PREFIX PARENTH CLASS	104
B.5	INFIX TO PREFIX OPERATOR CLASS	107

LIST OF FIGURES

Figure 4.1 Hierarchy of Athena Library Used in the Thesis.....	20
Figure 4.2 Categorization of Function Definitions	22
Figure 4.3 Use of Previously Proved Theorem in the Definition of New Functions.....	22
Figure 5.1 Length Comparison of Prefix and Infix Notation.....	40
Figure 6.1 Steps to Prove the Equivalence of Factorial and Efficient Factorial Functions	42
Figure 8.1 Flowchart of Code Generation Process	61
Figure 8.2 Flowchart of Infix to Prefix Converter	61
Figure 8.3 Illustration of Code Extraction	62

LIST OF TABLES

Table 2.1 Comparison of Verification Methods	4
Table 2.2 Function and Control Flow Representation	6
Table 2.3 Example of Prefix and Infix Notation.....	7
Table 2.4 Comparison of Proof Assistants Features []	8
Table 2.5 Example Proof of a Propositional Expression	11
Table 2.6: Athena Code and its representation in formal logic	12
Table 2.7 Athena Code and Its Mathematical Representation.....	13
Table 2.8: Example of Implication Proof	14
Table 5.1 Length Comparison of Prefix and Infix Notation	38
Table 7.1 Athena Axioms and Properties and Their Mathematical Representation	57

CHAPTER 1

INTRODUCTION

Wallace and Fuji (1989) explained that in the late 1960s and 1970s verification and validation developed at the same time with the beginning of software usage in military and nuclear-power systems increased. Initially, there were no general standards for validation and verification (V&V). In order to have a specification and for monitoring the performance of V&V efforts the US government and industry began to develop V&V. Today's V&V standards are used in large different kind of communities, also they applicable to many types of software (p.11) [1].

Although verification and validation often used at the same time, they have different meaning. Verification is the process to seek the answer of the question “Are we building the product right?”. It is the problem of checking whether the software meets its specification[2] or not. The validation is the question of “are we building the right product?” which is aimed for ensuring the software to meet consumer expectations [2].

1.1 STATEMENT OF THE PROBLEM

There are different ways of verifying software. The most popular methodology is to use testing which may not provide strong assurance if the software is too big. An alternative way is to use formal methods and theorem proving. Although using formal methods suffers from scalability, it provides higher assurance. The main focus of this thesis is to use formal methods to define small scale functions and construct a library

from them. The functions will be first defined axiomatically in the infix notation in a proof checker called Athena [3]. In the next step we will prove some safety and correctness related properties from these definitions. Since we define algorithms mathematically and logically, this will make it possible to construct a library of highly correct components that can be used for developing larger software.

The examples that are implemented in this thesis are selected from different domains. We mainly concentrated on functions that work with numbers, lists and Strings. We categorized the function definitions into four categories listed below:

- Mathematical functions,
- Excel formula related functions,
- String related functions, and
- Memory related functions.

Mathematical function definitions consist of the implementation of the minimum and factorial functions. We define a function that represents the MIN function in EXCEL. Another one is defined to return the sum of all natural numbers located in a list. We also formalized Java's substring method. In addition we formally defined (and proved properties of) memory by using the infix notation for memory-updating functions. We mainly preferred to use the infix notation since the proofs are more readable and shorter. We show that the infix proofs are shorter than the proofs that are written in the prefix notation in Athena.

Software verification problems can be classified by using the following three categories:

- security problems (i.e., unauthorized access to data or system resources)
- safety problems (i.e., illegal operations or illegal access to memory)
- functional incorrectness (i.e., the delivered code fails to satisfy a required relation between its input and output) [10].

In this research we focus on safety and functional correctness problems.

1.2 ORGANIZATION OF THE THESIS

We start with a basic introduction to major verification techniques in the next chapter. Since the thesis is based on the formal methods, the terminology related to theorem proving and proof checking is explained in the same chapter. Athena theorem prover and its properties with example definition and proofs are presented. Chapter-3 reviews the related work in the area. We discuss our methodology and contributions with many example definitions from different domains in Chapter-4. We compare the lengths of axiomatic definitions and proofs in prefix and infix notation in Chapter-5. We show that infix notation makes the proofs shorter in Athena. We then present various case studies with example proofs in Chapter-6. The next chapter illustrates how to generate code from axioms. In the last chapter, we talk about the conclusions and the future work.

CHAPTER 2

SOFTWARE VERIFICATION

Main objective of software verification is to detect errors in software as early as possible. If the programs that do not meet specification can be identified early, this will result in less cost on error treatment. It also helps in producing safer and more correct software. There are several ways to conduct software verification; some of the methods and their characteristics are listed in table 2.1.

Table 2.1 Comparison of Verification Methods

Name	Explanation	Characteristics
Peer Reviewing	<ul style="list-style-type: none">▪ <i>No code execution</i>▪ <i>Examiners don't have any relations with software production step</i>▪ <i>Code is inspected statically line by line</i>	<ul style="list-style-type: none">▪ <i>Prone to human error since it is done manually</i>▪ <i>Takes too much time</i>▪ <i>Expensive</i>
Testing	<ul style="list-style-type: none">▪ <i>Code is divided into smaller parts</i>▪ <i>Code has to be executed</i>▪ <i>Test the code with test cases (inputs) that aim to cover all paths in the program</i>	<ul style="list-style-type: none">▪ <i>Too many paths</i>▪ <i>Covering all paths is almost impossible in bigger software</i>
Simulation	<ul style="list-style-type: none">▪ <i>Testing the model instead of the software itself</i>	<ul style="list-style-type: none">▪ <i>Model construction is expensive</i>▪ <i>The model should really resemble the behavior of the real software</i>
Formal Verification	<ul style="list-style-type: none">▪ <i>Use formal mathematics for specifications</i>▪ <i>Prove the correctness of the software mathematically</i>	<ul style="list-style-type: none">▪ <i>Greater mathematical certainty for correctness</i>

2.1 TESTING

Since testing is the most widely used verification method, it will be discussed in this section before the formal methods are explained in detailed.

Testing is a way to detect a program's error by trying possible input data. The bigger and more complex a program is, the more possible input data is. However, the depth of testing varies according to testing coverage. There are branch coverage, line coverage, statement coverage, etc. If you want a more thorough test, then more coverage and therefore more input data is needed to be tested.

Here are the steps to test a program:

1. Drawing the control flow representation of the code
2. Calculate possible paths
3. Generate test cases that cover all the paths
4. Check the correctness of every paths

In the table 2.2 it is shown the example of function and its control flow representation.

Table 2.2 Function and Control Flow Representation

Java Code of “Same Function”	Control Flow Representation of The Code
<pre> public static int same(int first, int y1, int y2, int y3) { if (y1 == 1) { // Statement 1 (S1), Branchpoint 1 first++; // Statement 2 2 (S2) } if (y2 == 1) { // Statement 3 (S3), Branchpoint 2 first--; // Statement 4 (S4) 4 } if (y3 == 1) { // Statement 5 (S5), Branchpoint 3 first = first*1; // Statement 6 (S6) } return first; // Statement 7 7 (S7) } </pre>	

2.2 FORMAL METHODS

Most programmers use testing method to verify their programs, however, testing might be inadequate especially when the number of paths to be observed is too many. In such cases, the testers ignore some of the paths. Since the percentage of coverage becomes smaller, this method does not guarantee the absence of errors in the code. Thus, we need more reliable methods. An alternative approach is to use formal methods which are based on mathematical proofs; this type of verification provides stronger assurance of safety and correctness. This methodology requires one to write a specification of the system formally and then prove a correctness or safety property of the system by using a set of axioms and rules of inference. This method has also been assured useful, not difficult, applicable to help any system, and decrease the cost of development in an article written by Anthony Hall [4].

2.3 PROOF ASSISTANTS

In order to write specifications of the systems and proving the properties of programs we need a theorem prover (a proof assistant). Most of the theorem provers are based on functional programming which uses prefix notation. In function calls, operators or function names come before the operands or parameters. This naturally results in proofs that are written in prefix notation.

From the way of it is written, proofs are divided into two categories which are prefix and infix. When function name is written before the parameters or operands then it is prefix. If the function name is placed in the middle between operands or parameters then it is infix. Some examples of prefix and infix notation are shown in the table 2.3.

Table 2.3 Example of Prefix and Infix Notation

Mathematical notation	Prefix	Infix
$0 + 0$	(Plus zero zero)	(zero Plus zero)
$P \times T$	(Times P T)	(P Times T)

Proofs are generally very long. Even a specification of small functions and a proof of a property related to this specification might be large. Therefore, it is important to write shorter proofs. Proofs must be readable and machine checkable. Readable proofs are proofs that do not contain many words of natural language, instead using more mathematical notations and formal language. Using a formal language will also make the proofs shorter since many long words will be represented as one or two symbols. Writing proofs in infix notation may also increase the readability in some cases.

There are many proof checkers and theorem provers. Some popular proof assistants and their features are listed in the next page and shown in table 2.4:

- **ACL2:** it is part of the Boyer-Moore family provers [5]. It is the successor to Nqthm logic and proof system. It is developed by Matt Kaufmann and J Strother Moore. It is developed at computational logic, Inc. and at the University of Texas, Austin. ACL2 is actually stands for “A Computational Logic for Applicative Common Lisp”.

- **Coq:** is an interactive theorem prover developed in France, jointly operated by INRIA, Ecole Polytechnique, Paris-Sud 11 University, Paris Diderot University and CNRS. It is firstly developed by Thierry Coquand and Gerard Huet. Coq can extract certified program into Caml, Haskell or Scheme [6].
- **Isabelle:** Isabelle is the successor of HOL (Higher Order Logic) theorem prover. It is written in Standard ML and is developed at the University of Cambridge (Larry Paulson), Technische Universität München (Tobias Nipkow) and Université Paris-Sud (Makarius Wenzel). Isabelle/HOL can convert executable specifications directly into code in SML, OCaml, Haskell, and Scala [7].

Table 2.4 Comparison of Proof Assistants Features [8]

Proof Assistant	Coq	Isabelle/Isar	ACL2
Small proof kernel ('proofobjects')	+	+	-
Calculations can be proved automatically	+	+	+
Extensible/programmable by the user	+	+	-
Based on higher order logic	+	+	-
Readable proof input files	-	+	+
Large mathematical standard library	+	+	-

In table 2.4, a comparison of these three proof assistants is illustrated. In order to understand the table easier, let's discuss more about the proof assistant features listed on the table:

- **Small proof kernel ('proofobjects')**
Proof rules are based on the proof kernel, so since proof kernel is the source of other proofs, then the kernel should be trustable. Smaller kernel is easier to be checked and more trustable [9].
- **Calculations can be proved automatically**
The system can automatically conduct proofs using existing theorems in the system. Users generally interact with the system when it needs new lemmas to prove the theorem.
- **Extensible/programmable by the user**
The proofs are extensible, which mean that you can add new stuffs to the system like defining and asserting new axioms, declaring new functions, or adding new theorems to the library.

- Based on higher order logic
It support use of function as parameter and can yield function as a result.
- Readable proof input files
Input files are structured and can be understood by human.
- Large mathematical standard library
There are large numbers of already proof mathematical theorems in the library.

We use Athena proof assistant in this research. It is a tool that makes it possible to write proofs that are both human-readable and machine-checkable [10]. Athena has been used in several projects. It has been used in a project titled Termination Checking without Using an Ordering Relation [11] and also has been used to write the system of CCT project [12]. The first project is related to a new alternative to function termination checking method. In this method, instead of using ordering relation, axioms related to termination condition is defined. The axioms are defined automatically by using TCGen which is termination condition generator. TCGen is written in Athena and also generate axioms in Athena proof language. Athena also used in CCT project which is an approach to make transmission of code safer. Instead of sending the code and the proof at the same time to the consumer, CCT makes this transaction safer by only sending the proof and make code extraction possible by the consumer. The implementation of CCT was done by using Athena. However, in those projects Athena were used in prefix notation. In this project we take the advantage of Athena infix notation feature as much as possible.

Athena can be compared to the proof checkers that are listed in table 2.4 as follows:

- Athena uses small proof kernel,
- Calculations can be proved automatically by the use of external theorem provers,
- the system is extensible by the user,
- It is based on higher order logic,
- More readable proof input files,

- It has smaller mathematical standard library compared to the others. This is because Athena is a new proof assistant and so there is still limited contribution to its library.

2.4 ATHENA THEOREM PROVER

Athena has both a computational and a deduction language which can be used as a proof assistant. The computational part of Athena makes it possible to implement Lisp style code. The deduction language part is used for writing specifications and proofs which can be checked automatically by the language itself. We will now explain the features of Athena which are used extensively in the next several chapters.

2.4.1 Features of Athena

Athena has so many features which cannot all be explained in this thesis. Below we mention only several features of Athena that are related to this research. We also talk about some logic-related concepts with the way that is defined in Athena environment.

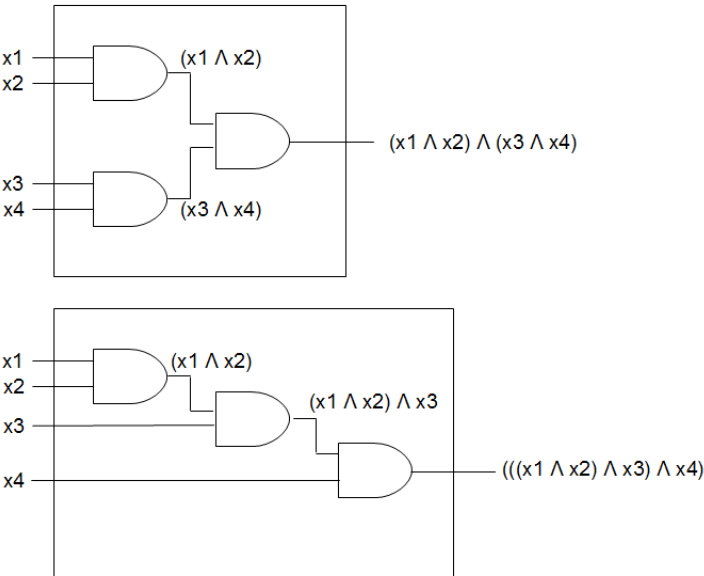
Assumption base

Athena has a concept of assumption base which is a global set of propositions that are true [13]. The assumption base is empty when the system starts. It is filled with premises, theorems, and assumptions while new theorems are proved. While premises and theorems stay there until the system is off or it is manually removed, hypothesis only remains there according to their scopes. We can see the propositions in the assumption base by using `(show-assumption-base)` method.

Propositional Logic

Athena supports several rules of inference that are built in as primitive methods. It provides operators that represent rules of inference including biconditional introduction, biconditional elimination, conjunction introduction, simplification, disjunction introduction, disjunction elimination, modus ponens, constructive dilemma, double negation and more. An application of rule of inference is shown in Table 2.5.

Table 2.5 Example Proof of a Propositional Expression

Logic representation of a simple circuit	Athena Proof
	<pre> declare x1,x2,x3,x4 : Boolean assert Ax1 := ((x1 & x2) & (x3 & x4)) (!both (!both (!left-and Ax1) (!left-and (!right-and Ax1))) (!right-and (!right-and Ax1)))</pre>

Proof by Contradiction Technique

Sometimes a proposition P can be proved in an indirect way. Instead of proving it directly, we assume that the negation of P is correct and try to find a contradiction from it. We then conclude that the proposition is right. This proof technique is called proof by contradiction. In Athena, proof by contradiction can be done in two ways. The first one is to use `suppose-absurd` and the second way is to use the `by-contradiction` method. The general format of `suppose-absurd` method is:

```
(suppose-absurd P D)
```

where P is a hypothesis and D is the body of the proof. The scope of hypothesis P is only valid inside the body D . Table 2.6 has an example where `suppose-absurd` is used.

Table 2.6: Athena Code and its representation in formal logic

Proof in Athena	Formal logic
<pre> ((if (and (if p q) (not q)) (not p)) BY (assume (and (if p q) (not q)) (suppose-absurd p (!absurd (!mp (!left-and (and (if p q) (not q))) p) (not q)))))) </pre>	$(((p \Rightarrow q) \wedge \sim q) \Rightarrow \sim p)$ <p>Assumption base</p> $(p \Rightarrow q) \wedge \sim q$ <p>p</p> <p>mp of $(p \Rightarrow q)$ p which is q</p> $\sim q$

We can also use by-contradiction method to do this by the form

(by-contradiction P D) as shown in the following example.

<pre> (assume P (!by-contradiction D assume ~D (!absurd conclude D ~D))) </pre>
--

Proof by Rewriting

Proof by rewriting consists of several methods which are reduce, extend, setup and combine. Rewriting is the process where a term in a sentence is replaced by another one. In fact, after rewriting an expression we get an equivalent of it which can be used in a proof. The rewritten expression can be either simpler or a more complex one. When you are going from the more complex to the simpler, it is called reduction. On the contrary, if it is done from simpler to the more complex one it is called extending. In Athena infix notation, reducing and extending are represented with “-->” and “<--“ symbols. An example proof that uses rewriting frequently in Athena is illustrated in Table 2.7.

Table 2.7 Athena Code and Its Mathematical Representation

Proof in Athena	Mathematical Representation
<pre>assert left-zero := (forall ?n . (zero + ?n = ?n))</pre>	$n + 0 = n$
<pre>assert left-nonzero := (forall ?n ?m . (S ?m) + ?n = (S (?m + ?n)))</pre>	$(m + 1) + n = (m + n) + 1$ $((0+1)+1) + (0+1) = (0+1) + ((0+1)+1)$
<pre>conclude (((S (S zero)) + (S zero)) = ((S zero) + (S (S zero)))) (!chain [(S (S zero)) + (S zero) --> (S ((S zero) + (S zero))) [N.Plus.left-nonzero] --> (S (S (zero + (S zero)))) [N.Plus.left-nonzero] --> (S (S (S zero))) [N.Plus.left-zero] <-- (S (zero + (S (S zero)))) [N.Plus.left-zero] <-- ((S zero) + (S (S zero))) [N.Plus.left-nonzero]]))</pre>	$((0+1)+1) + (0+1)$ $((0+1) + (0+1))+1$ $((0 + (0+1))+1)+1$ $((0+1)+1)+1$ $(0+((0+1)+1))+1$ $(0+1)+((0+1)+1)$

Proof by Induction

Proof by induction is done by proving two cases. The first case is the base case and the second one is the induction step as in mathematics. This can also be seen as an application of rewriting where we replace the induction variable with zero and (succ n) for those cases if we are proving an expression for natural numbers. We then prove each expression separately.

For example, to prove that $0+n=n$, we apply rewriting both for $n=0$ and $n=(\text{succ } n)$ in the following proof.

```
define left-zero := (forall ?n . (zero + ?n = ?n))

by-induction left-zero {
  zero => conclude (zero + zero = zero)
          (!chain [(zero + zero) --> zero [right-zero]])
  | (S n) => conclude (zero + (S n) = (S n))
          let {induction-hypothesis := (zero + n = n)}
          (!chain [(zero + (S n))
                   --> (S (zero + n)) [right-nonzero]
                   --> (S n) [induction-hypothesis]])
}
```

Proving Implication

In order to prove an expressions that have the form:

if P then Q

which is an implication, we need to assume that P is correct and then we prove and conclude Q. Its format is shown in Table 2.8.

Table 2.8: Example of Implication Proof

Proof in Athena	Formal logic
(!(conclude (if P Q)) (assume P (!(conclude Q) ...)))	Get conclusion that (p => q)

2.5 READABILITY OF ATHENA DEFINITIONS AND PROOFS

A useful proof should be both machine checkable and human readable. Human readable proofs make it easier to understand an existing proof and edit it. Proof maintenance would take less time and would be easier to develop new proofs. Human readable proofs should be similar to natural languages that are used by humans in their daily lives. Also it should be reflecting the structure of mathematical expressions and proofs and be very similar to the hand-written proofs. Athena strives to achieve these goals by giving some features which are explained below:

- **Supports both prefix and infix**

Athena supports both prefix and infix writing style. You can implement a function both in prefix or infix notation. As an example you can write the plus function in both ways as follows:

(Plus a b) or (a Plus b)

In this case, the second is more readable since we use infix notation for addition in our daily lives assuming also that we replace Plus with the + symbol.

- **Symbols can be used instead of the name of functions**

When we look at the example in (a Plus b) case, we still did not use the actual symbols that are used in mathematics. In Athena, you can change the function name Plus with + sign in order to make it more readable. The function will be written as (a + b) which makes it more readable.

- **Usage of infix and mathematical notation will result in shorter proofs in Athena**

In Athena, if we combine the infix and mathematical notation, we can get shorter proofs in some cases. For example, the name plus contains 4 characters which can be decreased to 1 character if we use + sign instead. Obviously the + sign is more descriptive than any other symbols or words for the addition operator.

CHAPTER 3

RELATED WORKS

Verification of safety and correctness properties of software has been practiced extensively in many projects. CCT (Code-Carrying Theory) [10] and PCC (Proof-Carrying Code) [14] are just two of them. PCC (Proof-Carrying Code) [15] is a software mechanism that makes a code consumer possible to check a code that is sent from an untrusted agent (code producer) to be checked for its safety before the code is applied to the consumer's system. In this system, the consumer determines a safety condition that must be proved by the code producer before the execution of the code sent. The condition is written in a way that it is related to the safety of the code delivered. The code producer constructs the proof (which is hard) and sends it along with the code. The consumer checks the proof (which is an easy process compared to building the proof) and then run the binary. The proofs in PCC are semi-automatic and proof readability is not an issue.

CCT (Code-Carrying Theory) [10] is also an approach of secure code delivery. In PCC the producer sends both the code and the safety proofs. But in CCT only the definitions, proofs and axioms are sent. The consumer then obtains the code by an automatic extraction of the code from the axioms and theorems. The extraction is done by a tool called CODEGEN that is implemented using Athena. Although the work in CCT is very similar to our work, the definitions and the proofs are written in prefix notation. We believe that the new proofs are more understandable.

Our main focus is to construct libraries of related functions which are defined by using mathematical and logical axioms. We preferred to use infix notation whenever possible. Note that there are many theorem provers or proof checkers that support this notation. Proof assistants including Coq, Isabelle, and ACL2 have already been providing this feature. Although they adopt this notation, the syntax of Athena looks better which makes it easier to read definitions and proofs. Most Coq and Isabelle proofs can be obtained automatically or by using tactics. Therefore they are short which

prevents these proofs to be understood in a shorter period of time. However, we also note that it is better to use these theorem provers instead in some applications that do not care about the structure of the proof.

Our work also contains the infix and prefix definition of Memory and Java's substring function. Although Coq and Isabelle have larger theorem libraries, definitions of Substring and memory are not included in their libraries.

CHAPTER 4

METHODOLOGY AND CONTRIBUTIONS

This chapter presents the steps of our methodology and contributions. We used the Athena proof checker as the implementation environment. In order to build a library of functions, we selected some examples first. These functions are then defined axiomatically in Athena. In some cases, we set and proved some properties based on these definitions which could be used in the proofs of more complex properties. Some of the properties define safety and correctness of functions. We especially selected various functions that can work with data which are from different domains such as natural numbers, lists and mixture of them. We categorized the functions as Mathematical, Excel-related and String-related. We formalized the Java's substring function with this way and proved some useful properties.

There are many functions that use memory for only reading purposes. However, in real life applications memory is updated frequently. In order for this to be possible, memory must be defined formally in Athena. We used infix notation to simulate memory access, assignment and swap operations as part of our research.

Another contribution is that Athena does not have any mechanisms to convert infix proofs into their equivalent in prefix notation. In some cases, it is better to use definitions in prefix notation. Also the current code extractor program in Athena can only extract code from the definitions in prefix notation. Therefore, it is necessary to translate infix definitions into prefix definitions. We have implemented an infix to prefix converter in Java. This program will produce the prefix version of any definitions whether they are axioms, theorems or properties which do not have any proofs.

By using formal method that is applied using Athena proof assistant, safety theorem is constructed. The idea is that the program will counter to an illegal instruction if any unsafe instruction is encountered during the execution of the code [10]. In order to get a safe function, we will need a safety requirements or also called safety policy (e.g., no out-of-bounds array-indexing) [10]. Also to counter functional incorrectness, we need functional correctness requirements (e.g., an algorithm that result correct output) [10].

4.1 ATHENA PREFIX AND INFIX LENGTH COMPARISON STUDY

We compare Athena old library which used prefix notation with library update which is written in infix notation. In this comparison study, we try to prove that infix notation is shorter than the legacy. Even the usage of symbols makes the codes simpler and closer to its mathematical representation. The complete study will be discussed in Chapter 5.

4.2 EXTENDING LIBRARY IN ATHENA

4.2.1 Library hierarchy in Athena

Athena has its own library collection. In this library there are proofs of some mathematical properties from the very basic to the more complex one. More complex properties often defined by using previously proved properties. For example to define any multiplication-related properties of natural numbers, we need to use the properties that are define the addition operation for natural numbers and this already defined in Athena. Other examples are shown in the Figure 4.1:

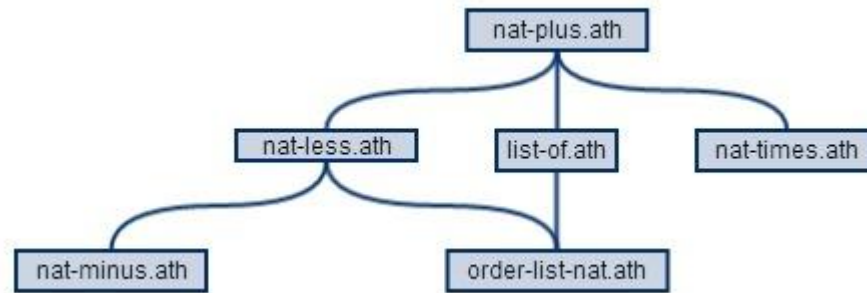


Figure 4.1 Hierarchy of Athena Library Used in the Thesis

As it is shown in Figure 4.1, the definition of plus operator and some related properties and proofs are used in the definition of less-than operator. The definition of plus operator is included in the file named `nat-plus`. Similarly, the definition of less than operator is written in `nat-less.ath` file. All these operators are defined for only natural numbers where we use `nat` as an abbreviation. We use the same naming convention for the other definitions shown in the figure. For example, one can interpret that `nat-minus.ath` has the definitions and related proofs of the minus operator for natural numbers and is based on the definitions and theorems of less-than operator.

Below is a little more explanation about each library file.

nat-plus.ath : Defines Natural numbers as a type and has axioms, theorems and proofs for some properties including associativity and commutativity of plus operator.

nat-less.ath : Defines the less than operator and has axioms, theorems and proofs for some properties including transitivity, being asymmetric and trichotomy.

list-of.ath : Defines List as a type and has axioms, theorems and proofs for some properties related to list join operator and other functions. For example there are proofs for the associativity of join, a property that reverses a list, another property about the length of lists, a property that has replace function that replaces an element in a list.

nat-times.ath : Defines the multiplication operator (times) and has axioms, theorems and proofs for some properties including commutativity and associativity of times, and distributivity of times over plus operators.

nat-minus.ath : Defines the minus operator and has axioms, theorems and proofs for some properties including plus-cancel, second-equal, second-greater, second-greater-or-equal.

order-list-nat.ath : Defines List.<=L function which checks whether a natural number is less than or equal to the first element of a list of natural numbers.

The advantage of proving complex properties by using existing theorems (which are proven already) is to achieve more correctness and safety. It is like building a wall from powerful bricks, you will get powerful wall in return, because every brick's durability has been tested. Here the idea is to get the safer and more correct function by using proven theorems.

In this research, we have extended Athena library by writing additional axiomatic definitions of some new functions. We have proved many properties of them. Using the existing definitions and proofs in the Athena library that is shown in Figure 4.1, we have defined new functions such as `Min`, `MinList`, `Split`, `Access`, `Substring` etc., as show in Figure 4.2.

4.2.2 New Infix Definitions of Some Functions

We have defined and proved new functions by using the infix notation in Athena. We classified the functions into 4 categories which are

- Mathematical functions,
- Excel formula related functions,
- String related functions, and
- Memory related functions.

In each category there are definitions of functions which are related to the category itself. For example, in mathematical function category, there are formal definitions of minimum and the factorial function. Also there are many other useful definitions in each category. The full content of each category is shown in Figure 4.2.

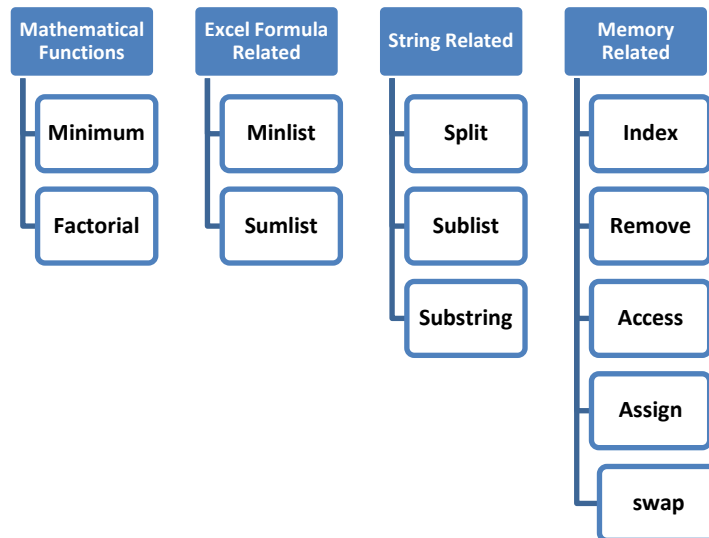


Figure 4.2 Categorization of Function Definitions

While writing the formal definition of functions in Figure 4.2, we used previously written axioms and proven theorems from Athena's libraries. Figure 4.3 shows the hierarchy that explains about which file is used in the development of other files.

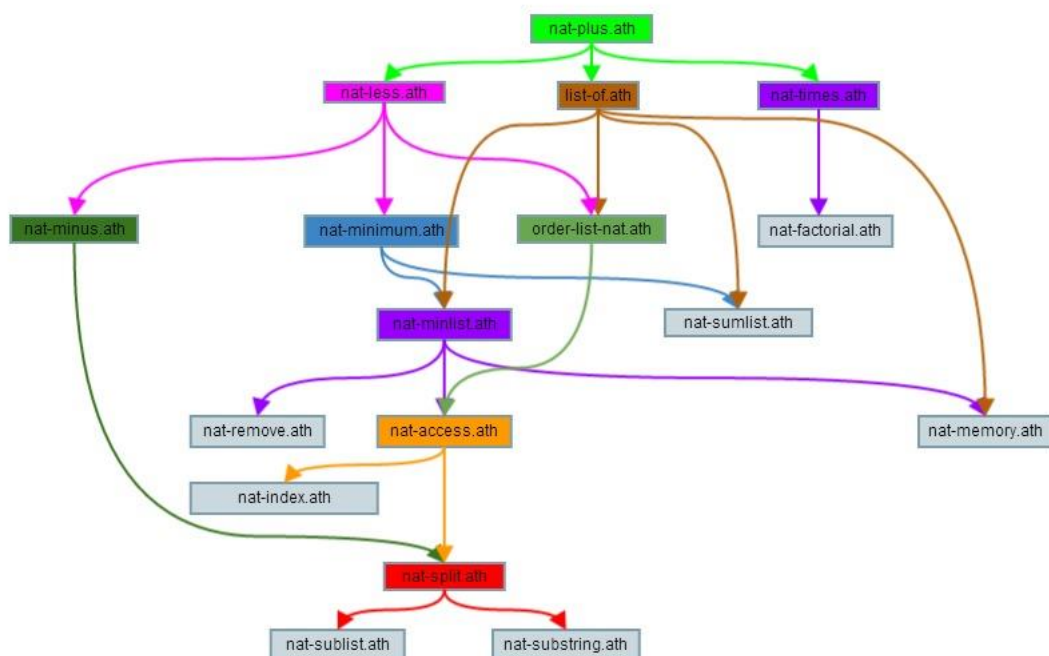


Figure 4.3 Use of Previously Proved Theorem in the Definition of New Functions

We will talk about these function definitions briefly here. More detailed explanations for some representative examples will be given in chapter-6 and chapter-7.

4.2.2.1 Mathematical Functions

Min: The axiomatic definition of minimum represents a function that takes two natural numbers and returns the minimum of them. It also has example proofs about some properties of the minimum function. We first show its definitions below.

Definition of `Min`:

```
assert less2 := (forall ?x ?y . ?x < ?y ==> (Min ?x ?y) = ?x)
assert not-less2 :=
  (forall ?x ?y . ~ (?x < ?y) ==> (Min ?x ?y) = ?y)
```

There are two axioms that define the function `Min`. The first axiom states that if the first natural number is less than the second natural number, then the `Min` function returns the the first number since it is the smaller one. The second axiom states that if the first number is not less than the second one then the minimum is the second number. Because either they are equal or the second number is smaller.

Here are some properties that we have defined and proved for the `Min` function:

```
define commutative := (forall ?x ?y . (Min ?x ?y) = (Min ?y ?x))
define associative :=
  (forall ?x ?y ?z . (Min (Min ?x ?y) ?z) = (Min ?x (Min ?y ?z)))
define idempotent := (forall ?x . (Min ?x ?x) = ?x)
```

For example, the first two properties which are named as commutative and associative state that the `Min` function is both commutative and associative as in the addition operator. More details and example proofs are listed in Appendix-A.2.

Factorial: We show below the axiomatic definitions of the regular factorial function and its tail-recursive version which is more efficient. We also proved the equivalence of both but the proof will be presented later in chapter-6.

Here is the definition of regular `Factorial` function:

```
define Factorial-zero-axiom := ((Factorial zero) = (S zero))

define Factorial-succ-axiom :=
  (forall ?x .
    (Factorial (S ?x)) = (S ?x) * (Factorial ?x))
```

The first axiom states that the factorial of zero is equal to one. The second axiom states that the factorial of $(x+1)$ is equal to $(x+1)*x!$.

We below define a more efficient version which is tail-recursive:

```
define FactorialEfficientHelper-zero-axiom :=
  (forall ?res . (FactorialEfficientHelper zero ?res) = ?res)

define FactorialEfficientHelper-succ-axiom :=
  (forall ?n ?res .
    (FactorialEfficientHelper (S ?n) ?res) =
    (FactorialEfficientHelper ?n ((S ?n) * ?res)))

define FactorialEfficient-axiom :=
  (forall ?n .
    (FactorialEfficient ?n) =
    (FactorialEfficientHelper ?n (S zero)))
```

The `FactorialEfficientHelper` takes two parameters. The first parameter is the number that undergoes factorial operation; the second one is an accumulator that helps factorial function to be performed in a more effective manner. The first axiom states that `FactorialEfficientHelper` of zero with the second parameter `res` is equal to `res`. The second axiom states that `FactorialEfficientHelper` of $(n+1)$ with the second parameter `res` is equal to `FactorialEfficientHelper` of n with second parameter $(n+1)*res$ which is going to be returned as the result of the factorial operation. And finally the third axiom states that `FactorialEfficient` of n can be computed by calling the function represented as `FactorialEfficientHelper` with the parameters n and one. The value of n will

be decreased by one with each call and the factorial will be computed by multiplying each value of it with the second parameter.

The following property is defined and proved (the proof is shown in chapter-6 and Appendix-A.1). The proof shows that the factorial efficient can be used instead of the factorial.

```

define P1 :=
  (forall ?n ?a .
    (FactorialEfficientHelper ?n ( ?a * (S zero))) =
    ( ?a * (FactorialEfficientHelper ?n (S zero))))

define Equality :=
  (forall ?n .(Factorial ?n) = (FactorialEfficient ?n))

```

This is actually the correctness or equivalence condition to be proved for the stated problem.

4.2.2.2 Excel Formula Related Definitions

The two function definitions that we are going to discuss in this section are related to Excel formulas. Remember that in Microsoft Excel we can enter a formula that finds the minimum [16] number among a list of numbers that are located in a row or column. Similarly, it is also possible to compute the sum of numbers in a row or column. We present axiomatic definitions for `MinList` and `sum-list` to define these excel functions. Our only restriction is that we work with only natural numbers. Please also note that our definition is more general in the sense that they do not have to be for numbers in cells as in Excel. Our definitions can be used for any data structures that represent a list of numbers in a list fashion.

MinList: Axiomatic definition that represents a function which takes a list of natural numbers as an input and returns the smallest number in the list. We first show the two axioms that we decided to define the `MinList` function. We then show some properties that we already proved. More detailed explanation and proofs will be covered in the Chapter-6.

Here is the formal definition of `MinList`:

```

assert MinList-One-axiom :=
  (forall ?x:N .
    (MinList (?x :: nil)) = ?x)

assert MinList-Cons-axiom :=
  (forall ?x ?L .
    (MinList (?x :: ?L)) = (Min ?x (MinList ?L)))

```

After defining the `MinList` symbol axiomatically, we specified some properties. These properties which are shown below are proved in Athena. Some proofs about the properties of the `MinList` function is explained in chapter 6.

```

assert MinList-ConsMin-property :=
  (forall ?x ?y .
    (MinList (?x :: (?y :: nil))) = (Min ?x ?y))

define MinListMin-Relation-property :=
  (forall ?L ?x ?y .
    (MinList (?x :: (?y :: ?L))) =
    (MinList ((Min ?x ?y) :: ?L)))

define MinList-Min-property :=
  (forall ?L ?x:N ?y:N .
    (N.Min ?x (MinList (?y :: ?L))) =
    (MinList (?x:N :: (?y :: ?L))))

define MinList-Join-property :=
  (forall ?L1 ?x:N ?y:N ?L2 .
    (Min (MinList (?x :: ?L1)) (MinList (?y :: ?L2))) =
    (MinList ((?x :: ?L1) List.Join (?y :: ?L2))))

define Length2 :=
  (forall ?L2:(List N) .
    ( ~((List.Length ?L2) = zero)
      ==>
      (exists ?y:N ?Lx:(List N) .
        (?L2 = (?y :: ?Lx))))

define MinList-Join-property-if :=
  (forall ?L1 ?L2 .
    ( ~((List.Length ?L1) = zero)
      & ~((List.Length ?L2) = zero)
    ==> ((Min (MinList ?L1) (MinList ?L2)) =
          (MinList (List.Join ?L1 ?L2))))

```

sum-list: Axiomatic definition that represents a function that takes a list of natural numbers and returns the sum of them. `Sum-list` was defined in the prefix notation

here[11] in the past. In this study, we defined it in infix notation. This example is also used for showing the proof process of a correctness property. In order to do this, we defined the `sum-list` function in two different ways. The first version represents a regular recursive function which may consume the stack frames with repeated calls. The second version is tail-recursive and thus more efficient. One can use the first definition as the specification of how to compute the sum of numbers in a list. In the next step we will just need to prove the equivalence of this specification and the definition of the tail-recursive version. The definitions are shown below.

Definition of `sum-list@` (This is not efficient and can serve as the specification) :

```
define sum-list@-empty := ((sum-list@ nil) = zero)

define sum-list@-nonempty :=
  (forall ?x:N ?L:(List N) .
    (sum-list@ (?x :: ?L)) = (?x + (sum-list@ ?L)))
```

Definition of `sum-list` (This is the more efficient version):

```
define sum-list-empty := ((sum-list nil)
                          = zero)
define sum-list-nonempty :=
  (forall ?L:(List N) ?x:N .
    (sum-list (?x :: ?L))
    = (sum-list-compute ?L ?x))
```

Definition of `sum-list-compute` which is the extension of `sum-list`:

```
define sum-list-compute-empty :=
  (forall ?x .
    (sum-list-compute nil ?x)
    = ?x);;

define sum-list-compute-nonempty :=
  (forall ?L ?x ?y .
    (sum-list-compute (?y :: ?L) ?x)
    = (sum-list-compute ?L (?x + ?y)))
```

Below is the definition of `sum-list-compute-relation`. This intermediate property needs to be proved first. The proof is presented in Appendix-A.3.

```
define sum-list-compute-relation :=
  (forall ?L ?x .
    (sum-list@ (?x:N :: ?L:(List N)))
    = (sum-list-compute ?L ?x))
```


The definition of `sum-list-correctness` whose proof is listed in Appendix-A.3 is shown below. This is the actual correctness condition. If the proof is obtained, one can use the more efficient version instead of the non-tail-recursive function.

```
define sum-list-correctness :=
  (forall ?L:(List N) .
    (sum-list ?L) =
      (sum-list@ ?L))
```

4.2.2.3 String Related Functions

The library below is generally used with String domain in real life applications. In this thesis we limit the scope into natural numbers for uniformity.

Split: Consist of 2 axiomatic function definitions, which are `SplitLeft` and `SplitRight`. They take two parameters which are a list and an index. `SplitLeft` resembles a function that returns a list which consists of elements from the smallest index of the list (index zero) until the given index passed as a parameter. `SplitLeft` function actually also resembles the specification of Java's `Substring` method that takes only one parameter. `SplitRight` resembles a function that returns a list with elements that are located in the index positions starting from the input index (passed as a parameter) to the end of the list.

The Definition of `SplitLeft` Function:

```
define SplitLeft-zero-axiom :=
  (forall ?L:(List N) .
    (SplitLeft ?L zero) = nil)

define SplitLeft-succ-axiom :=
  (forall ?L:(List N) ?n:N .
    (SplitLeft ?L (S ?n)) =
      ((SplitLeft ?L ?n)
        List.Join ((Access ?L (S ?n)) :: nil)))

define SplitLeft-nil-axiom :=
  (forall ?n:N .
    (SplitLeft nil ?n) = nil)

define SplitLeft-cons-axiom :=
  (forall ?x:N ?L:(List N) ?n:N .
    (SplitLeft (?x :: ?L) ?n) =
      (?x :: (SplitLeft ?L (?n N.- N.one))))
```

The definition of SplitRight Function:

```

define SplitRight-zero-axiom :=
  (forall ?L:(List N) . (SplitRight ?L zero) = ?L)

define SplitRight-succ?-axiom :=
  (forall ?L:(List N) ?n:N .
    (SplitRight ?L ?n) =
      ((Access ?L ?n) :: (SplitRight ?L (S ?n))))

define SplitRight-nil-axiom :=
  (forall ?n:N .
    (SplitRight nil ?n) = nil)

define SplitRight-cons-axiom :=
  (forall ?x:N ?L:(List N) ?n:N .
    (SplitRight (?x :: ?L) ?n) =
      (SplitRight ?L (?n N.- N.one)))

```

We show a property that uses both SplitLeft and SplitRight with the join operator that is already defined for lists, the complete proof is shown in Appendix-A.8.

```

define SplitCombination-property :=
  (forall ?L:(List N) ?n:N .
    ((SplitLeft ?L ?n) List.Join (SplitRight ?L ?n)) = ?L)

```

- **SubList:** Axiomatic definitions that represent a function that takes 3 parameters, which are a list, an index, and a length. It will return the sublist starting from the given index to the $(\text{index} + \text{length} - 1)$ position. The length parameter indicates the number of elements to be taken from the list. The definitions are listed below. The complete proofs are in the Appendix-A.9.

Definitions of SubList:

```

define SubList-zero-axiom :=
  (forall ?L ?l .
    (SubList ?L zero ?l) = (SplitLeft ?L ?l))

define SubList-succ-axiom :=
  (forall ?L ?n ?l .
    (SubList ?L (S ?n) ?l) =
      (SplitRight (SplitLeft ?L ((S ?n) + ?l)) (S ?n)))

define SubList-nil-axiom :=
  (forall ?n ?l .
    (SubList nil ?n ?l) = nil)

```

```

define SubList-cons-axiom :=
  (forall ?x ?L ?n ?l .
    (SubList (?x :: ?L) ?n ?l) =
      (SplitRight (SplitLeft (?x::?L) (?n + ?l)) ?n))

```

After defining the SubList function we proved the following properties whose proofs are presented in Appendix-A.9.

```

define SubList-n-property :=
  (forall ?n ?L ?l .
    (SubList ?L ?n ?l) =
      (SplitRight (SplitLeft ?L (?n + ?l)) ?n))

define SubList-L-property :=
  (forall ?L ?n ?l .
    (SubList ?L ?n ?l) =
      (SplitRight (SplitLeft ?L (?n + ?l)) ?n))

```

- **Substring:** Axiomatic definition that represents a function that takes 3 parameters which are a list, a beginIndex, and an endIndex. This symbol models the specification of the Substring method [17] that takes two parameters in Java String library. The function returns the sublist from beginIndex to endIndex-1. Below is the list of definitions inside the substring file. Substring will be explained in more detail in chapter-7.

Substring definitions:

```

define Substring-zeron-axiom :=
  (forall ?L ?f .
    ?f N.<= (List.Length ?L)
    ==> (Substring ?L zero ?f) = (SplitLeft ?L ?f))

define Substring-succn-axiom-if :=
  (forall ?L ?n ?f .
    (S ?n) N.<= ?f & ?f N.<= (List.Length ?L)
    ==> (Substring ?L (S ?n) ?f) =
      (SplitRight (SplitLeft ?L ?f) (S ?n)))

define Substring-n-Length-axiom :=
  (forall ?L ?n .
    ?n N.<= (List.Length ?L)
    ==> (Substring ?L ?n (List.Length ?L)) =
      (SplitRight ?L ?n))

define Substring-equalinput-Length-axiom :=
  (forall ?L ?n .
    ?n = ?f ==> (Substring ?L ?n ?f) = nil)

```

We defined and proved the following Substring properties. The complete proof is illustrated in Appendix-A.10.

```

define Substring-property1 :=
  (forall ?n:N ?L ?f:N .
    ?n N.<= ?f & ?f N.<= (List.Length ?L)
    ==>(Substring ?L ?n ?f) =
      (SplitRight (SplitLeft ?L ?f) ?n))

define full-list-property :=
  (forall ?L ?n .
    ?n N.<= (List.Length ?L)
    ==> ((Substring ?L zero ?n) List.Join
      (Substring ?L ?n (List.Length ?L))) = ?L)

```

4.2.2.4 Memory Related Definitions

A memory definition is very important to construct memory-updating functions. In many specifications, we need to assign, access, remove, swap, and get the index of an element inside memory. This requirement can be seen in the Split file library that we have implemented to define Java's String method. In the file, we can see that in order to define `Split`, we need to access memory. Memory definition is also applicable to many other definitions. Previously, memory had been defined in CCT[11] project by using the prefix notation. In this thesis, we rewrite them by using the combination of infix and prefix notation. We also added more definitions and proofs in infix notation.

Index: Axiomatic definition that represents a function that returns an index location of an element in a list. The function takes 2 parameters which are an element that is going to be searched and a list where the element might occur.

Below is the definition of `ElementLocation` function:

```

define ElementLocationHelper-Next-axiom :=
  (forall ?z:N ?x:N ?L:(List N) ?n:N .
    (?z in (?x :: ?L)) & (?z != ?x)
    ==> (ElementLocationHelper ?z (?x :: ?L) ?n) =
      (ElementLocationHelper ?z ?L (S ?n)))

define ElementLocationHelper-True-axiom :=
  (forall ?z ?L ?n .
    (?z in (?z :: ?L))
    ==> (ElementLocationHelper ?z (?z :: ?L) ?n) = ?n)

define ElementLocation-axiom :=

```

```
(forall ?z ?L . (?z in ?L)
  ==> (ElementLocation ?z ?L) =
      (ElementLocationHelper ?z ?L zero))
```

Below is the definition of `ElementLocation` property. The proof of it is located in the Appendix-A.7.

```
define Property1 :=
  (forall ?z:N ?L:(List N) .
    (?z in ?L)
      ==> (Access ?L (ElementLocation ?z ?L)) = ?z)
```

Remove: Axiomatic definition that represents a function that takes a list of natural numbers and an index. It removes the element located in the given index and returns the updated list. The definitions located inside this file are listed as follows.

Definition of `RemoveWithIndex` function:

```
define RemoveWithIndex-zero-axiom :=
  (forall ?x:N ?L:(List N) .
    (RemoveWithIndex (?x :: ?L) zero) = ?L)

define RemoveWithIndex-succ-axiom :=
  (forall ?x:N ?L:(List N) ?n .
    (RemoveWithIndex (?x :: ?L) (S ?n)) =
      (RemoveWithIndex ?L ?n))

define RemoveWithIndex-nil-axiom :=
  (forall ?x:N ?L:(List N) ?p .
    (RemoveWithIndex nil ?p) = nil)
```

The definition of `RemoveWithIndex` property related to `MinList`. The proof can be seen in the Appendix-A.4.

```
define Property1 :=
  (forall ?x:N ?L:(List N) .
    (MinList (RemoveWithIndex (?x :: ?L) zero)) =
      (MinList ?L))
```

Memory: We defined memory and some operators to process it. The major operations on memory can be done by using `access`, `assign`, and `swap` functions. In the previous work in CCT, a memory model was defined in prefix notation. In this thesis, since our work is limited to natural numbers, we define `Memory` as a list of natural numbers. Also this new version has definitions in both infix and the prefix version.

The `access` function takes a list of natural numbers and an index value as parameters. It returns the content of the memory location accessed by the index value. The `assign` function takes 3 parameters which are a list, an index, and a value which will be assigned to the memory location with the given index. It returns a new Memory which is basically another list with the content of related index is updated with the given value. The `swap` function also takes 3 parameters which are a list and two indices. The list represents a memory and the two indices are used for swapping the content of the memory locations identified by these two indices. The returned list is a new memory where the contents of memory locations identified by `index1` and `index2` are exchanged. We now will give the definition of these operators:

Definitions of the `access` function in infix and prefix notation:

```
define Access-first-axiom :=
  (forall ?x:N ?L:(List N) .
    (Access (?x :: ?L) zero) = ?x)

define Access-Cons-axiom :=
  (forall ?x:N ?L:(List N) ?n:N .
    (Access (?x :: ?L) (S ?n)) = (Access ?L ?n))
```

As an example, we show a very important property about the equivalence of two memory definitions below. This property is proved and the proof is presented in Appendix-A.6.

```
define Memory-equality :=
  (forall ?M1 ?M2 .
    (?M1 = ?M2) <==>
    (forall ?a .
      (access ?M1 ?a) =
      (access ?M2 ?a)))
```

The assign function can be defined by using the following two axioms:

```

define assign-axiom1 :=
  (forall ?M ?a ?x .
    (access (assign ?M ?a ?x) ?a) = ?x)

define assign-axiom2 :=
  (forall ?M ?a ?b ?x .
    ( ?a /= ?b)
    ==>
    (access (assign ?M ?a ?x) ?b) =
    (access ?M ?b))

```

Definition of swap function:

```

define swap-axiom1 :=
  (forall ?M ?a ?b .
    (access (swap ?M ?a ?b) ?a) =
    (access ?M ?b));;

define swap-axiom2 :=
  (forall ?M ?a ?b .
    (access (swap ?M ?a ?b) ?b) =
    (access ?M ?a));;

define swap-axiom3 :=
  (forall ?M ?a ?b ?c .
    ((?c /= ?a) & (?c /= ?b))
    ==> (access (swap ?M ?a ?b) ?c) =
    (access ?M ?c));;

```

Here is a property which uses the assign operator. The proof of this one is included in Appendix-A.6.

```

define Double-assign :=
  (forall ?b ?M ?a ?x ?y .
    (access (assign (assign ?M ?a ?x) ?a ?y) ?b) =
    (access (assign ?M ?a ?y) ?b));;

```

```

define Direct-double-assign :=
  (forall ?M ?a ?x ?y .
    (assign (assign ?M ?a ?x) ?a ?y) =
      (assign ?M ?a ?y));;

define Self-assign :=
  (forall ?M ?a ?b .
    (access (assign ?M ?a (access ?M ?a)) ?b) =
      (access ?M ?b));;

```

The proofs of the following two swap properties are included in Appendix-A.6.

```

define Direct-double-swap :=
  (forall ?M ?a ?b .
    (swap (swap ?M ?a ?b) ?b ?a)
      = ?M)

define Direct-same-swap :=
  (forall ?M ?a ?b .
    (swap (swap ?M ?a ?b) ?a ?b) = ?M)

```

Below is the definition of a property related to `access` and `MinList` whose proof is listed in Appendix-A.5.

```

define Property1 :=
  (forall ?x:N ?L:(List N) .
    (List.ordered (?x :: ?L))
      ==> (Access (?x :: ?L) zero) =
          (MinList (?x :: ?L)))

```

Athena is using a declarative proof language. So the proofs that are mentioned before are also a contribution to declarative formal proof literature. A declarative proof language is easier to be understood both by machines and humans than procedural proof languages.

4.3 DEVELOPING INFIX TO PREFIX CONVERTER PROGRAM

Athena's newer version has additional capabilities. It is now possible to write and check proofs in infix notation. The previous version of Athena only accepts prefix notation. However, a code generator tool that can generate code in a programming language (OZ language) by using a set of axioms that are written in the prefix notation in Athena was already developed. In order to generate code from our proofs that are

written in new Athena, we need to convert definitions in infix notation to their prefix form. We have implemented a Java program that converts an infix definition into its prefix equivalent.

CHAPTER 5

COMPARISON OF PROOF AND DEFINITION LENGTHS IN PREFIX AND INFIX NOTATIONS

As mentioned before Athena supports both prefix and infix notation. Infix notation in Athena is not just changing the place of operators, but there is more of it. While developing infix notation, Athena also overload several operator names into symbol, which makes codes shorter, simpler, and more readable. Not only operator names, even they change several commands name into symbols. Hence, makes them also more similar to their mathematical representation.

We compare four library files of Athena which are `nat-half.ath`, `nat-plus.ath`, `nat-times.ath`, and `nat-less.ath`. We use `note++` as a tool to get number of characters, words, lines, and document length of the file. In order to get better comparison, we also use the help of <http://www.charcounter.com/> to count characters ignoring spaces. Comparing characters without counting the spaces eliminates the space usage differences in both files compared. The result is shown in table 5.1 below.

Table 5.1 Length Comparison of Prefix and Infix Notation

Library	Notation	(prefix)	(infix)	difference
Half.ath vs. Nat-half.ath	Characters without spaces	6149	4111	2038
	Characters	8147	6184	1963
	Words	1301	955	346
	Lines	297	171	126
	Document Length	8739	6354	2385
Naturals.ath vs. Nat-plus.ath	Characters without spaces	5317	3525	1792
	Characters	7047	4935	2112
	Words	1175	761	414
	Lines	305	180	125
	Document Length	7655	5114	2541
Naturals-times vs. Nat-times.ath	Characters without spaces	4997	3363	1634
	Characters	6672	4815	1857
	Words	1047	665	382
	Lines	263	156	107
	Document Length	6934	4970	1964
Naturals-ordering.ath vs. nat-less.ath	Characters without spaces	21866	13154	8712
	Characters	28944	19271	9673
	Words	4375	2684	1691
	Lines	1181	617	564
	Document Length	31304	19887	11417

Characters without spaces: the most obvious way to see length difference between prefix and infix. It only counts characters and ignores other characters and blanks.

Characters: counts all characters including spaces without counting blanks.

Words: count words that are separated by spaces and blanks

Lines: count number of lines in the code. In order to increase the readability of an expression in prefix notation, we need to structure by using more lines of Athena expressions. But by using infix notation, one can write the same things by using fewer lines of code. In other words, using fewer lines still make the code more readable. It changes the culture of Athena writing style. Formerly, in order to make the code more readable, users had to use many lines as shown below:

```
(define nonzero-half-even
  (forall ?n
    (if (and (not (= ?n zero))
            (Even ?n))
        (not (= (half ?n) zero))))))
```

After infix notation has been introduced, it can be written as:

```
define half-nonzero-if-nonzero-Even :=
  (forall ?n .
    ?n /= zero & (Even ?n)
    ==> (half ?n) /= zero)
```

Document Length: the total length of the document in character unit.

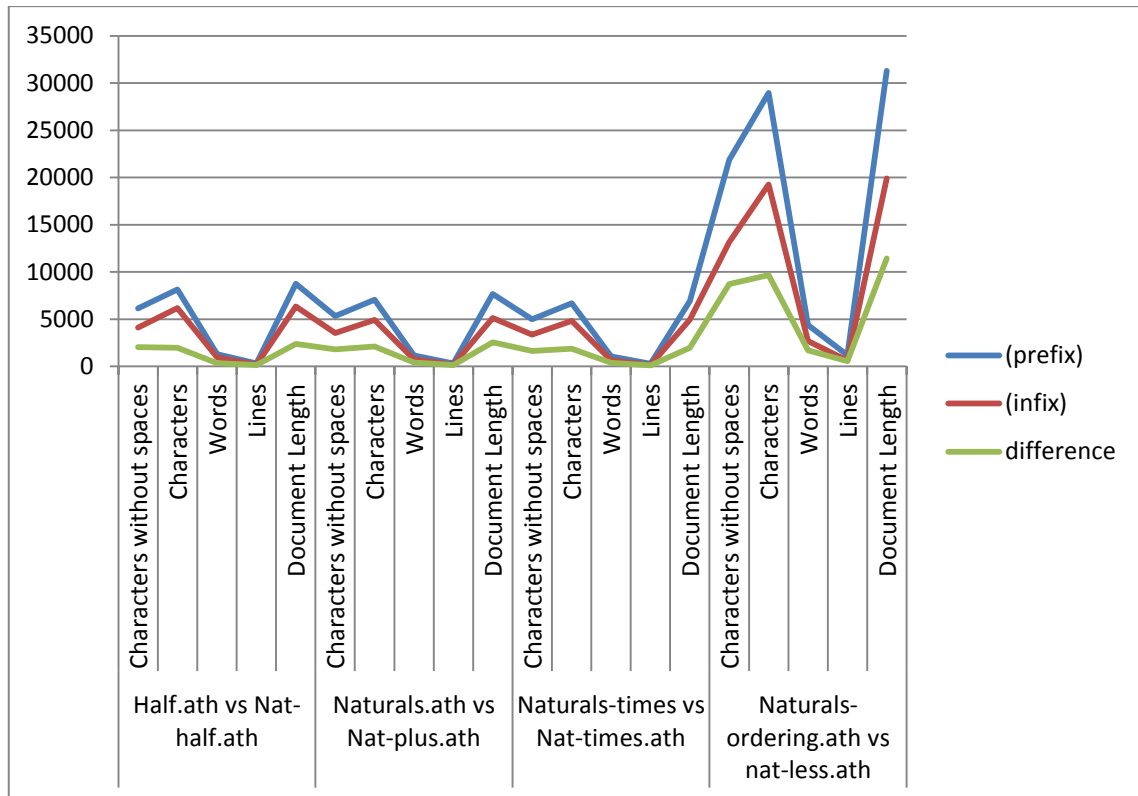


Figure 5.1 Length Comparison of Prefix and Infix Notation

Figure 5.1 shows that the length differences between prefix and infix codes is proportional with the length of the entire code. It can be clearly seen in nat-times comparison and nat-less comparison. The entire code of nat-times prefix is 4997 characters (withous spaces) and infix is 3363 characters and the difference is 1634 characters. While in nat-less the entire code for prefix is 21866 characters and infix is 13154 characters the difference is 8712 characters. So as the code size gets bigger, the number of character difference will be bigger too.

CHAPTER 6

CASE STUDIES

6.5 ATHENA PROOF OF FACTORIAL AND FACTORIAL EFFICIENT

In computer programming, we can implement functions doing the same work in different ways. However, the way we implement can affect the performance of the function. Some implementations are safer and some are faster than the others. This notion also applies in factorial function implementations. Definition of factorial mathematically is,

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ (n-1)! \times n & \text{if } n > 0. \end{cases}$$

In java programming language, we can implement a function based on these definitions intuitively as

```
public class Factorial {  
  
    // Version-1: This version is not efficient  
    public static int Factorial(int n){  
        if (n==0)  
            return 1;  
        else  
            return n*Factorial(n-1);  
    }  
}
```

However, this code is not efficient since it uses too many stack frames when it is called with a big number. If n is too big, repeated calls lead to stack overflow.

The solution to this issue is to implement the same function in a tail-recursive way. In that case we will need to define two functions but the code will be more efficient since it is easy to translate this code into an iterative version. In the next page, we show the implementation of the same function in tail-recursive form in Java.

```

public static int FactorialEfficient(int n){
    return FactorialEfficient_Helper(n,1);
}
public static int FactorialEfficient_Helper(int n, int res){
    if (n==0)
        return res;
    else
        return FactorialEfficient_Helper(n-1, n*res);
}

```

Here, in each recursion, only one memory frame is needed. So `FactorialEfficient` is both faster and safer because it does not cause any stack overflow problems.

However, we still need to be sure that the efficient factorial function is giving the same result as the inefficient factorial for all inputs. In order to be sure about that, we need to prove the equivalence of the “factorial” and the “efficient factorial” functions. We prove this equivalence by using Athena proof assistant. First we are going to create a specification for `Factorial` function and then define the `FactorialEfficient` axiomatically. In the next step, we define a correctness condition that specifies the equality of these two functions. After that we are going to prove that the correctness condition. We show the steps of proving equality of these functions in figure 5.1.

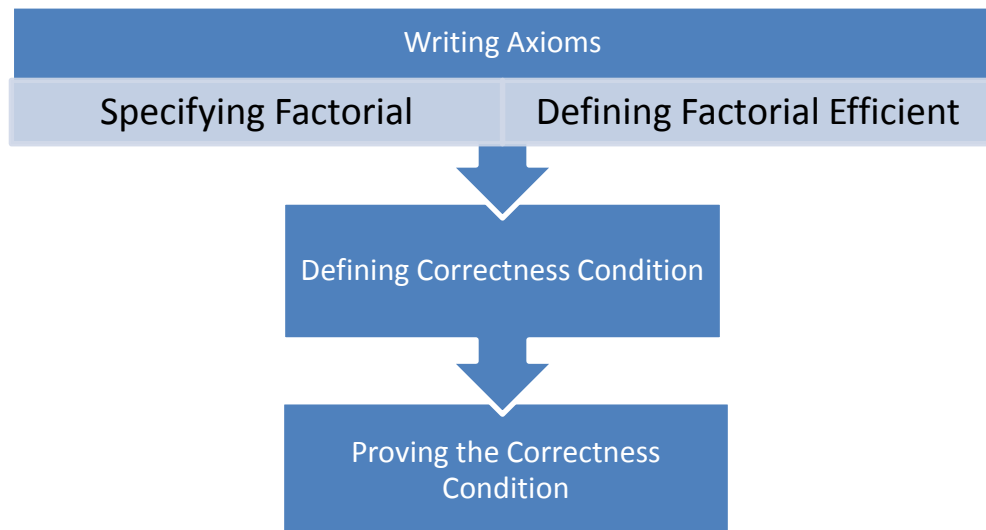


Figure 6.1 Steps to Prove the Equivalence of `Factorial` and `Efficient Factorial` Functions

In Athena we first declare the `Factorial` symbol. In the next step we define this function by using two axioms as follows,

```
declare Factorial: [N] -> N

assert Factorial-zero-axiom := ((Factorial zero) = (S zero))

assert Factorial-succ-axiom :=
  (forall ?x . (Factorial (S ?x)) = (S ?x) * (Factorial ?x))
```

Then we assert them so that Athena can save it into its assumption base. With this way, we make sure that these axioms can be used in the proof of new properties.

```
(assert Factorial-zero-axiom Factorial-succ-axiom)
```

We do the same things for the `FactorialEfficient`. But in this case we have to define two functions one of which is a helper function as shown below:

```
declare FactorialEfficientHelper : [N N] -> N

assert FactorialEfficientHelper-zero-axiom :=
  (forall ?res .
    (FactorialEfficientHelper zero ?res) = ?res)

assert FactorialEfficientHelper-succ-axiom :=
  (forall ?n ?res .
    (FactorialEfficientHelper (S ?n) ?res) =
    (FactorialEfficientHelper ?n ((S ?n) * ?res)))

declare FactorialEfficient : [N] -> N

assert FactorialEfficient-axiom :=
  (forall ?n .
    (FactorialEfficient ?n) =
    (FactorialEfficientHelper ?n (S zero)))
```

Again we assert these definitions to put them into the assumption base.

```
(assert
  FactorialEfficientHelper-zero-axiom
  FactorialEfficientHelper-succ-axiom)
(assert FactorialEfficient-axiom)
```


Now all the definitions are ready to be used in the proofs. The hard part is to prove the correctness condition which needs to be defined in Athena properly as a property. Here is the definition:

```
define Equality
  (forall ?n .(Factorial ?n) = (FactorialEfficient ?n))
```

Unless this correctness condition is proved we cannot use the `FactorialEfficient` instead of `Factorial`. Note also that this condition must be proved. Hence it cannot be asserted as it is done with axioms.

While proving this equation, we use the top down approach. We attempt to prove the main goal but sometimes we may need to break it down to smaller properties. We first prove these smaller ones to make the proof process easier. We call these smaller theorems as lemmas. In this case we also need to prove a lemma before the main theorem. Here is the proof of factorial equality.

```
by-induction Equality{
  zero =>
    (!chain [(Factorial zero)
             -->(S zero)
             [Factorial-zero-axiom]
             <--(FactorialEfficientHelper zero (S zero))
             [FactorialEfficientHelper-zero-axiom]
             <--(FactorialEfficient zero)
             [FactorialEfficient-axiom]])
  | (S n) =>
    let {induction-hypothesis :=
         (Factorial n) = (FactorialEfficient n)}
        (!chain
         [(Factorial (S n))
          -->((S n)*(Factorial n))
          [Factorial-succ-axiom]
          -->((S n)*(FactorialEfficient n))
          [induction-hypothesis]
          -->((S n)*(FactorialEfficientHelper n (S zero)))
          [FactorialEfficient-axiom]
          <--(FactorialEfficientHelper n ((S n)*(S zero)))
          [...]]
          <--(FactorialEfficientHelper (S n) (S zero))
          [FactorialEfficientHelper-succ-axiom]
          <--(FactorialEfficient (S n))
          [FactorialEfficient-axiom]])
}
```

You can see that in the proof there is a part with dots. We cannot finish the equivalence proof without getting a new theorem there. In that part we should fill a theorem that `proof (FactorialEfficientHelper (S n) (S zero))` is equal to `(FactorialEfficientHelper n ((S n)*(S zero)))`. So in order to fill the theorem, now we define a lemma called P1 and prove it as follows:

```

define P1 :=
  (forall ?n ?a .
    (FactorialEfficientHelper ?n ( ?a * (S zero)))
    = ( ?a * (FactorialEfficientHelper ?n (S zero))))

by-induction P1{
  zero =>
  pick-any a
  (!chain [(FactorialEfficientHelper zero (a*(S zero)))
    --> (a*(S zero))
    [FactorialEfficientHelper-zero-axiom]
    <-- (a*(FactorialEfficientHelper zero (S zero)))
    [FactorialEfficientHelper-zero-axiom]])
| (S zero) =>
  let {induction-hypothesis :=
    (forall ?a .
      (FactorialEfficientHelper n (?a*(S zero)))=
      (?a*(FactorialEfficientHelper n (S zero)))}
  pick-any a
  (!chain
  [(FactorialEfficientHelper (S n) (a*(S zero)))
  -->(FactorialEfficientHelper n ((S n)*(a*(S zero))))
  [FactorialEfficientHelper-succ-axiom]
  <--(FactorialEfficientHelper n (((S n)*a)*(S zero)))
  [Times-Associativity]
  -->(((S n)*a)*(FactorialEfficientHelper n (S zero)))
  [induction-hypothesis]
  -->((a*(S n))*(FactorialEfficientHelper n (S zero)))
  [Times-Commutativity]
  <--(a*((S n)*(FactorialEfficientHelper n (S zero))))
  [Times-Associativity]
  <--(a*(FactorialEfficientHelper n ((S n)*(S zero))))
  [induction-hypothesis]
  <--(a*(FactorialEfficientHelper (S n) (S zero))))])
  [FactorialEfficientHelper-succ-axiom]])
}

```

After P1 is proved, we use it in the proof the Equality property. We simply write the name of the property in the part identified by [...] where we replace the dots with the name P1.

6.6 DEFINITION OF MINLIST FUNCTION THAT RETURNS THE MINIMUM OF A LIST

In the previous case study, we have seen a definition of a function that accepts only natural numbers. We now present a function that takes a List as a parameter. The `MinList` function that we define here returns the smallest number of a list which is passed as a parameter. We show below the 3 axioms that define the `MinList` function:

```
assert MinList-One-axiom :=
  (forall ?x:N .
    (MinList (?x :: nil)) = ?x)

assert MinList-Cons-axiom :=
  (forall ?x ?L .
    (MinList (?x :: ?L)) = (Min ?x (MinList ?L)))

assert MinList-ConsMin-property :=
  (forall ?x ?y .
    (MinList (?x :: (?y :: nil))) = (Min ?x ?y))
```

The `MinList` function can only work with non-empty lists. The first axiom states that if the list has only one element, it is the smallest number. Here the `::` operator stands for the `cons` operator which is used for appending an element to the front of a list. The second axiom is defined for lists that have more than one element. It states that the minimum number is either the first element of the list or among the other elements in the sublist.

After defining the axioms, we proved some useful properties. Then we try to prove some of `MinList` Properties. Those properties are as follows:

```
define MinListMin-Relation-property :=
  (forall ?L ?x ?y .
    (MinList (?x :: (?y :: ?L))) =
    (MinList ((Min ?x ?y) :: ?L)))
```

The `MinList` of `x cons y cons L` will return the minimum number among the numbers `x,y` and the numbers in list `L`. We can simply find the minimum of `x` and `y` first

and then assume that this is going to be appended to a list. The `MinList` will select the minimum of this new list. That is what this property has stated.

```
define MinList-Min-property :=
  (forall ?L ?x:N ?y:N .
    (N.Min ?x (MinList (?y :: ?L))) =
     (MinList (?x:N :: (?y :: ?L))))
```

The minimum of `x` and `MinList` of `y` consed by any `List` is equal to `MinList` of `x cons y cons any List`.

```
define MinList-Join-property :=
  (forall ?L1 ?x:N ?y:N ?L2 .
    (Min (MinList (?x :: ?L1)) (MinList (?y :: ?L2))) =
     (MinList ((?x :: ?L1) List.Join (?y :: ?L2))))
```

The minimum between `MinList x cons L1` and `MinList y cons L2` is equal to the `MinList` of the lists consed together.

```
define MinList-Join-property-if :=
  (forall ?L1 ?L2 .
    ( ~((List.Length ?L1) = zero)
      & ~((List.Length ?L2) = zero))
    ==> ((Min (MinList ?L1) (MinList ?L2)) =
          (MinList (List.Join ?L1 ?L2))))
```

This is the other way of making sure that the list is not empty. In the previous properties we used `cons`. We now have also used an `if (==>)` in the above property.

If `Length` of `L1` is not zero and `Length` of `L2` is not zero then `Minimum` of the result from `(MinList L1)` and result from `MinList L2` is equal to the result of `MinList` from the combination of two lists.

We only discuss some sample proofs for several properties here. Other proofs can be seen in the `Appendix-A.11`.

First we set a condition to identify the relationship between a list' length and list itself as follows.

```
define Length2 :=
  (forall ?L2:(List N) .
    ( ~((List.Length ?L2) = zero))
    ==> (exists ?y:N ?Lx:(List N) .
          (?L2 = (?y :: ?Lx))))
```

```
assert Length2
```

Here `length2` definition says that if `L2` is not zero then there must exist `y cons Lx`.

Then we apply by `induction` method to this property. In the factorial example, we applied the method by using natural numbers, here, we apply proof by induction on lists. Hence, we need to identify base case and inductive step for list data type. The base case for list is `nil` and the inductive step is `(x cons L)`.

```
by-induction MinList-Join-property-if {
  nil =>
pick-any L2:(List N)
  assume (~((List.Length nil:(List N)) = zero) &
    ~((List.Length L2) = zero))
  conclude ((Min (MinList nil) (MinList L2))
    = (MinList (List.Join nil L2)))
  (!dn
    (suppose-absurd (~((Min (MinList nil) (MinList L2))
      = (MinList (List.Join nil L2)))))
  (!absurd
    conclude ((List.Length nil:(List N)) = zero)
      (!chain
        [(List.Length nil:(List N))
          --> zero [List.Length.empty]
        ])
    (!left-and (~((List.Length nil:(List N)) = zero))
      & (~((List.Length L2) = zero))))))
  | (x :: L1) =>
pick-any L2:(List N)
  assume if-state := ( ~((List.Length (x:: L1)) = zero) &
    ~((List.Length L2) = zero))
  let {C1 := conclude (exists ?y ?Lx .
    (L2 = (?y :: ?Lx)))
    (!mp (!uspec Length2 L2)
      (!right-and if-state))
  }
  pick-witnesses s Ls for C1
  (!chain
    [(Min (MinList (x :: L1)) (MinList L2))
     --> (Min (MinList (x :: L1)) (MinList (s :: Ls)))
      [(L2 = (s :: Ls))]
     --> (MinList ((x :: L1) List.Join (s :: Ls)))
      [MinList-Join-property]
     <-- (MinList (List.Join (x :: L1) L2))
      [(L2 = (s :: Ls))]
    ])
}
```

By using this function as an example, we also want to show that in Athena, previously proved theorems can be effectively used to define other theorems. We call previously proven nat-minimum and list-of file inside `MinList`.

For example, the following proof segment is using `List.Length.empty` theorem which is part of the library.

```
conclude ((List.Length nil:(List N)) = zero)
  (!chain
    [(List.Length nil:(List N))
     --> zero [List.Length.empty]
    ])
```

The proof uses `List.Length.empty` theorem to prove that Length of a nil list is zero. This command only works if `list-of.ath` file is loaded to the current proof file.

The same also prevail for the proof below:

```
by-induction MinListMin-Relation-property{
  nil =>
    pick-any x:N y:N
    (!chain [(MinList (x :: (y :: nil)))
             -->(N.Min x y)
             [MinList-ConsMin-property]
             <--(MinList ((N.Min x y) :: nil))
             [MinList-One-axiom]])
  | (z :: p) =>
    pick-any x:N y:N
    (!chain [(MinList (x :: (y :: (z :: p))))
             -->(Min x (MinList (y :: (z :: p))))
             [MinList-Cons-axiom]
             -->(Min x (N.Min y (MinList (z :: p))))
             [MinList-Cons-axiom]
             <--(Min (Min x y) (MinList (z :: p)))
             [Min.associative]
             <--(MinList ((Min x y) :: (z :: p)))
             [MinList-Cons-axiom]])
}
```

The proof uses `Min.associative` theorem. It is previously proven inside the `nat-nimum.ath` file. Even `nat-nimum` itself use previously proven theorem from `nat-less.ath` file.

CHAPTER 7

FORMAL DEFINITION OF THE SUBSTRING METHOD IN JAVA

7.1 STRING

String [18] is a data type which generally represents a sequence of characters. Java library has a string class to represent characters in the in the programing language. This class has many methods that can help users to deal with string instances they write with this programing language. Users can apply methods to a created string like compare between strings, replace characters in a string, get specific index of a character, get specific character interval of a string, get character length of a string, etc. Here is an example of the string usage:

```
String str = "dealwithstringinjava";  
String replace = test.replace('t', 'p');  
System.out.println(replace);
```

7.2 SUBSTRING METHOD IN JAVA LANGUAGE

Substring method [16] is a method that is located under String Class in Java library. By using this method, user can take a specified substring from the given string. Substring takes two parameters which are beginIndex and endIndex. As it is stated in java documentation, the method can be implemented as follows:

public [String](#) substring(int beginIndex, int endIndex)

Returns a new string that is a substring of this string. The substring begins at the specified beginIndex and extends to the character at index endIndex - 1. Thus the length of the substring is endIndex-beginIndex.

Examples:

"hamburger".substring(4, 8) returns "urge"

"smiles".substring(1, 5) returns "mile"

Parameters:

beginIndex - the beginning index, inclusive.

endIndex - the ending index, exclusive.

Returns:

the specified substring.

Throws:

[IndexOutOfBoundsException](#) - if the beginIndex is negative, or endIndex is larger than the length of this String object, or beginIndex is larger than endIndex.

Here are some more examples:

```
String test = "abcdefg";
System.out.println(test);
System.out.println(test.substring(5, 7));
System.out.println(test.substring(5, 6));
System.out.println(test.substring(5, 5));
System.out.println(test.substring(9, 11));
System.out.println(test.length());
System.out.println(test.substring(5, 0));
```

7.3 CORRECTNESS REQUIREMENTS IN ATHENA FOR JAVA SUBSTRING METHOD

In order to create a correctness condition for Substring method, first we need to define the method in Athena. To create the definition, we formerly need a proved

Athena method that can represent the definition of substring method. Here we will use `SplitLeft` and `SplitRight` method to define substring method.

First let's take a brief review about `SplitLeft` and `SplitRight` method in Athena.

```
declare SplitLeft : (N) [(List N) N]-> (List N)
```

By looking into the declaration, it is clear that `SplitLeft` is a function that takes List of Natural numbers and also one more natural number that represents an index. The implementation format is as follow :

```
(SplitLeft ?List ?index)
```

Here, index is the position to which element of the list will be taken. Index is excluded which means the result will show list elements up to `index-1`. So let's say we have a List called alphabet which consists of ('a 'b 'c 'd' 'e 'f 'g) characters, the function `(SplitLeft alphabet 3)` returns list ('a 'b 'c).

Next function we need to know is `SplitRight` function. `SplitRight` function is really similar to `SplitLeft` function. Below is the declaration and implementation format of `SplitLeft` function.

```
declare SplitRight : (N) [(List N) N]-> (List N)
(SplitRight ?List ?index)
```

While `SplitLeft` bring you the element of the list from the beginning to `index-1` position, `SplitRight` get you list elements from index to the end of the list.

Moreover, `SplitRight` function is actually represent one of Java substring method which uses only one parameter.

public String substring(int beginIndex)

Returns a new string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string.

Examples:

"unhappy".substring(2) returns "happy"

"Harbison".substring(3) returns "bison"

"emptiness".substring(9) returns "" (an empty string)

Parameters:

beginIndex - the beginning index, inclusive.

Returns:

the specified substring.

Throws:

IndexOutOfBoundsException - if beginIndex is negative or larger than the length of this String object.

So we can also say that `SplitRight` is the definition of `substring (int beginIndex)` method.

After understanding `Split` functions, now we can continue defining `substring` by using them. The definition is divided into 3 steps which are declaration, axioms, and properties.

a. Declaration

Since the function of `substring` is to get elements in a list that is in specific interval, so we need 3 parameters here, which are a `List`, a `beginIndex` and an `endIndex`. `beginIndex` and `endIndex` cannot be negative, so we use natural number as their domain. Also, since in this thesis we only cover natural numbers element, we only permit the element of `List` to be members of natural numbers. Those requirements will be implemented in `Athena` declaration as follows:

```
declare Substring : (N) [(List N) N N] -> (List N)
```

`Substring` : name of the function

(N) : function will only operates in natural number domain

[(List N) N N] : data type for each parameter

(List N) : data type of the result

b. Axioms

The next step is defining axioms. While defining axiom, we write simple application of the function and its result. Here are the axioms that we use to define `substring` function:

```
(Substring ?L zero ?f) = (SplitLeft ?L ?f)

(Substring ?L (S ?n) ?f) =
  (SplitRight (SplitLeft ?L ?f) (S ?n))

(Substring ?L ?n (List.Length ?L)) = (SplitRight ?L ?n)

(Substring ?L ?n ?f) = nil
```

c. Properties

The last step in the correctness requirements is to define property for the function.

```
(Substring ?L ?n ?f) = (SplitRight (SplitLeft ?L ?f) ?n)

((Substring ?L zero ?n) List.Join
  (Substring ?L ?n (List.Length ?L))) = ?L
```

7.4 SAFETY REQUIREMENTS IN ATHENA FOR JAVA SUBSTRING METHOD

In this part, we present an example that guarantees the safety of `substring` method by preventing out of bounds array access. If we take a look again on how the `substring` method is used, we understand that we need some limitations to its `beginIndex` and `endIndex`. Without any limitations to the parameters, this method will generate error that is caused by index out of bonds. Here are the limitations that we need:

1. The beginIndex must be positive
2. The endIndex must be smaller than the length of the string
3. The beginIndex must to be smaller than the endIndex

So we apply these limitations into definitions that are written before in correctness requirements.

a. Declaration

We do not need to apply limitation to declaration so the declaration is still like this:

```
declare Substring : (N) [(List N) N N] -> (List N)
```

b. Axioms

We will observe through all the axioms again and see whether they need limitations.

```
(Substring ?L zero ?f) = (SplitLeft ?L ?f)
```

This axiom clearly needs limitation in parameters L and f. We have to put rule for these parameters. By looking into limitation points we have discussed above we can decide that we need the second rule here, endIndex must be smaller than the length of the string. So we add `?f N.<= (List.Length ?L)` to ensure the safety of the function.

```
(Substring ?L (S ?n) ?f) =  
(SplitRight (SplitLeft ?L ?f) (S ?n)))
```

We add `(S ?n) N.<= ?f & ?f N.<= (List.Length ?L)` so that `(S ?n) beginIndex` is smaller or equal to `?f endIndex`. Also we add `?f N.<= (List.Length ?L)` so that `endIndex` is smaller or equal to the list length.

```
(Substring ?L ?n (List.Length ?L)) =  
(SplitRight ?L ?n)
```

`?n(beginIndex)` should be less than or equal to list length but since `beginIndex` must be less than or equal to `endIndex`, and `endIndex` must be less than or equal to list length, then automatically `beginIndex` will be less than list length. So we don't need to add specific rule to this axiom.

```
(Substring ?L ?n ?f) = nil)
```

The result of substring can be nil if `beginIndex` is equal to `endIndex` so here we need to add if `?n = ?f` rule to avoid array out of bound error in programming languages.

c. Properties

Since the property consists of `beginIndex` and `endIndex` parameters, we also need to give limitation to them.

We add the limitation `?n N.<= ?f & ?f N.<= (List.Length ?L)` to `(Substring ?L ?n ?f) = (SplitRight (SplitLeft ?L ?f) ?n)` and `?n N.<= (List.Length ?L)` to `((Substring ?L zero ?n) List.Join (Substring ?L ?n (List.Length ?L))) = ?L)`

7.5 FINALIZING THE CODE FOR CORRECTNESS AND SAFETY REQUIREMENTS

We have the correctness and safety requirement definitions now. But the quantifiers are not added to the definitions presented in this section for the sake of simplicity. We will now complete the definitions by adding the required Athena quantifiers and other syntactic objects. We show the mathematical representations and their Athena equivalents in the table shown in the next page.

Below is a table that shows the final code and its resemblance in mathematical notations.

Table 7.1 Athena Axioms and Properties and Their Mathematical Representation

Axioms and Properties	Mathematical Notations
<pre>define Substring-zero-axiom := (forall ?L ?f . ?f N.<= (List.Length ?L) ==> (Substring ?L zero ?f) = (SplitLeft ?L ?f))</pre>	$\forall L, f \in \mathbb{N}. f \leq \text{length } L \Rightarrow$ $(\text{Substring2 } L \text{ zero } f) =$ $(\text{SplitLeft } L f)$
<pre>define Substring-succn-axiom-if := (forall ?L ?n ?f . (S ?n) N.<= ?f & ?f N.<= (List.Length ?L) ==> (Substring ?L (S ?n) ?f) = (SplitRight (SplitLeft ?L ?f) (S ?n)))</pre>	$\forall L, n, f \in \mathbb{N}. (S n) \leq f \ \& \ f$ $\leq \text{length } L$ $\Rightarrow (\text{Substring2 } L (S n) f)$ $= (\text{SplitRight } (\text{SplitLeft } L f) (S n))$
<pre>define Substring-n-Length-axiom := (forall ?L ?n . ?n N.<= (List.Length ?L) ==> (Substring ?L ?n (List.Length ?L)) = (SplitRight ?L ?n))</pre>	$\forall L, n \in \mathbb{N}. n \leq \text{length } L$ $\Rightarrow (\text{Substring2 } L n (\text{length } L))$ $= (\text{SplitRight } L n)$
<pre>define Substring-equalinput-Length-axiom := (forall ?L ?n . ?n = ?f ==> (Substring ?L ?n ?f) = nil)</pre>	$\forall L, n \in \mathbb{N}. n = f$ $\Rightarrow (\text{Substring2 } L n f)$ $= \text{nil}$
<pre>define Substring-property1 := (forall ?n:N ?L ?f:N . ?n N.<= ?f & ?f N.<= (List.Length ?L) ==> (Substring ?L ?n ?f) = (SplitRight (SplitLeft ?L ?f) ?n))</pre>	$\forall n, L, f \in \mathbb{N}. n \leq f \ \& \ f$ $\leq \text{length } L \Rightarrow (\text{Substring2 } L n f)$ $= (\text{SplitRight } (\text{SplitLeft } L f) n)$
<pre>define full-list-property := (forall ?L ?n . ?n N.<= (List.Length ?L) ==> ((Substring ?L zero ?n) List.Join (Substring ?L ?n (List.Length ?L))) = ?L)</pre>	$\forall L, f \in \mathbb{N}. n \leq \text{length } L$ $\Rightarrow ((\text{Substring2 } L \text{ zero } n)$ $\cap (\text{Substring2 } L n (\text{length } L)) = L$

7.6 PROVING AN EXAMPLE SAFETY PROPERTY OF STRINGS

Now, we need to prove the properties in order to be sure that they all really work and there is no inconsistency. Here, we use the combination of by-induction and by-contradiction while proving the properties.

For the first property, we apply induction on variable n . We prove for 2 cases of n , when n is zero and when n is $(S \ n)$. In both cases, we apply contradiction so that it

can deduce the implication rule in the property. Here is the full proof of Substring-property1:

```

define Substring-property1
:= (forall ?n:N ?L ?f:N . ?n N.<= ?f & ?f N.<= (List.Length ?L)
    ==>(Substring ?L ?n ?f) = (SplitRight (SplitLeft ?L ?f) ?n))

conclude Substring-property1
by-induction Substring-property1 {
zero =>
pick-any L:(List N) f:N
assume test := (zero N.<= f & f N.<= (List.Length L))
(!by-contradiction ((Substring L zero f) = (SplitRight (SplitLeft L f) zero))
assume (~(Substring L zero f) = (SplitRight (SplitLeft L f) zero))
(!absurd
conclude ((Substring L zero f) = (SplitRight (SplitLeft L f) zero))
let {A := (!mp (!uspec* Substring-zero-axiom [L f])
(!right-and test) )}
(!chain
[(Substring L zero f)
--> (SplitLeft L f) [A]
<-- (SplitRight (SplitLeft L f) zero) [SplitRight-zero-axiom]]
)

(~(Substring L zero f) = (SplitRight (SplitLeft L f) zero))
))
| (S n) =>
pick-any L:(List N) f:N
assume test := ((S n) N.<= f & f N.<= (List.Length L))
(!by-contradiction ((Substring L (S n) f) = (SplitRight (SplitLeft L f) (S n)))
assume (~(Substring L (S n) f) = (SplitRight (SplitLeft L f) (S n)))
(!absurd
conclude ((Substring L (S n) f) = (SplitRight (SplitLeft L f) (S n)))
let {B := (!mp (!uspec* Substring-succn-axiom-if [n L f])
test)}}
(!chain
[(Substring L (S n) f)
--> (SplitRight (SplitLeft L f) (S n)) [B]]
)

(~(Substring2 L (S n) f) = (SplitRight (SplitLeft L f) (S n)))
))
}

```

Also, for the second property, we apply by-induction and by-contradiction, but this time we apply induction on L. It means we prove the case when L is nil and (x :: L).

```

define full-list-property
:= (forall ?L ?n . ?n N.<= (List.Length ?L)
    ==> ((Substring ?L zero ?n)
List.Join (Substring ?L ?n (List.Length ?L))) = ?L)

conclude full-list-property
by-induction full-list-property {
nil =>
pick-any n:N
assume test := (n N.<= (List.Length nil:(List N)))
(!by-contradiction (((Substring nil zero n)
List.Join (Substring nil n (List.Length nil:(List N)))) = nil)

```

```

assume (~((Substring nil zero n)
         List.Join (Substring nil n (List.Length nil:(List N)))) = nil))
(!absurd
 conclude ((Substring nil zero n)
           List.Join (Substring nil n (List.Length nil:(List N)))) = nil)
let {A := (!mp (!uspec* Substring-zeron-axiom [nil n])
          test )}
let {B := (!mp (!uspec* Substring-n-Length-axiom [nil n])
          test )}
(!chain
  [((Substring nil zero n)
    List.Join (Substring nil n (List.Length nil:(List N))))
   -->((SplitLeft nil n)
    List.Join (Substring nil n (List.Length nil:(List N))))
    [A]
   --> ((SplitLeft nil n) List.Join (SplitRight nil n))
    [B]
   --> (nil List.Join (SplitRight nil n))[SplitLeft-nil-axiom]
   --> (nil List.Join nil)[SplitRight-nil-axiom]
   <-- nil:(List N) [List.Join.left-empty]]
)

(~((Substring nil zero n)
   List.Join (Substring nil n (List.Length nil:(List N)))) = nil))
)
| (x:N :: L) =>
pick-any n:N
assume test := (n N.<= (List.Length (x :: L)))
(!by-contradiction ((Substring (x :: L) zero n)
  List.Join (Substring (x :: L) n (List.Length (x :: L)))) = (x :: L))
assume (~((Substring (x :: L) zero n)
         List.Join (Substring (x :: L) n (List.Length (x :: L)))) = (x :: L)))
(!absurd
 conclude ((Substring (x :: L) zero n)
           List.Join (Substring (x :: L) n (List.Length (x :: L)))) = (x :: L))
let {A := (!mp (!uspec* Substring-zeron-axiom [(x :: L) n])
          test )}
let {B := (!mp (!uspec* Substring-n-Length-axiom [(x :: L) n])
          test )}
(!chain
  [((Substring (x :: L) zero n)
    List.Join (Substring (x :: L) n (List.Length (x :: L))))
   --> ((SplitLeft (x :: L) n)
    List.Join (Substring (x :: L) n (List.Length (x :: L))))
    [A]
   --> ((SplitLeft (x :: L) n) List.Join (SplitRight (x :: L) n))
    [B]
   --> (x :: L) [SplitCombination-property]
])
)

(~((Substring (x :: L) zero n)
   List.Join (Substring (x :: L) n (List.Length (x :: L)))) = (x :: L)))
)
}

```


CHAPTER 8

CODE GENERATION

One of the aims of this research is to obtain safe and/or more correct programming language functions. In the previous chapter we have discussed how to define and prove safety and/or correctness features. It is possible to generate code by using existing axioms and theorems. Usually this done through a code generator program. The previous version of Athena which uses prefix notation already has a program called CODEGEN[11] which converts Athena axioms to code in the Oz programming language. Since Athena now makes it possible to write expressions in infix notation we now also need to have a code generator that translates axioms written in the infix notation to some code in a programming language. However, to the date, there is still no tool that can generate code from axioms that are written in the newer version of Athena with infix notation. Finally we wrote a Java program which serves as a bridge between infix notation of Athena and CODEGEN. This tool converts the axiom and theorems that are written in the infix notation to their prefix equivalent. It can be seen as an infix to prefix notation converter. After obtaining the prefix version, the CODEGEN program can be used as is to generate the code. We believe that this is the best solution since it is also possible to use the converter for other purposes too. The figure 8.1 below explains the code generation steps.

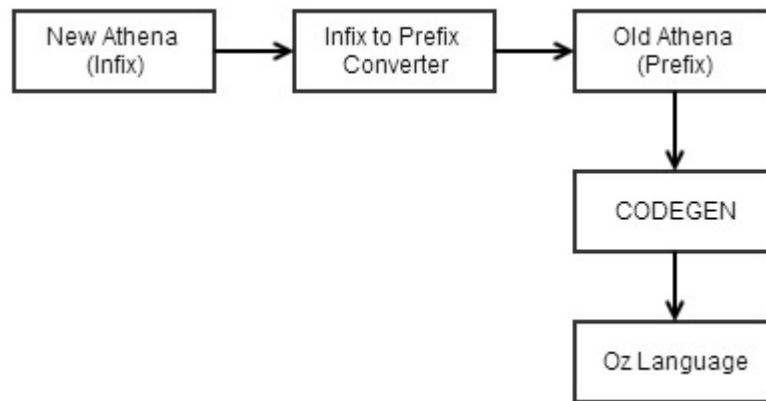


Figure 8.1 Flowchart of Code Generation Process

8.1 CONVERTING INFIX DEFINITIONS TO PREFIX FORM

We used Java[18] to develop infix to prefix converter. The flowchart that describes the steps of the program is as follows:

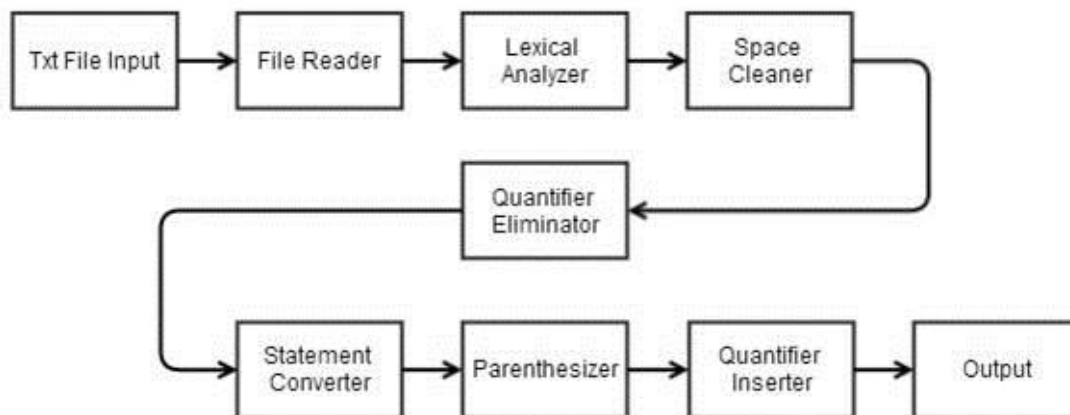


Figure 8.2 Flowchart of Infix to Prefix Converter

Txt File Input: Infix axiom or definition of a proof is inserted into text document as an input.

File Reader: reads the input from txt file.

Lexical Analyzer: Lexical analyzer tokenizes the input that comes character by character, and put the result into an ArrayList.

Space Cleaner: Delete spaces that exist in the lexical analyzer result and put them into a new ArrayList.

Quantifier Eliminator: Eliminates quantifier (which in this case is: forall) so that only particular statement that is ready to be converted remains.

Statement Converter: Convert the statement from infix to prefix.

Parenthesizer: Put the result from Statement converter into the correct parentheses.

Quantifier Inserter: Bring back quantifier that was eliminated before conversion.

Output: Show the prefix notation result for the related infix input.

While designing this program, we assume the axioms and a definition that is going to be converted has already been accepted and running correctly in Athena software. Based on this assumption, we do not include system that checks the correctness of the input. When a user enters some expression as an input, the program will still work and try to produce an output without checking the correctness of its input.

8.2 CODE EXTRACTION

Code generation is done by CODEGEN tool that take axioms as its input and returns code in Oz language as an output. Below is an example of input (Athena) that is converted into language called Oz.

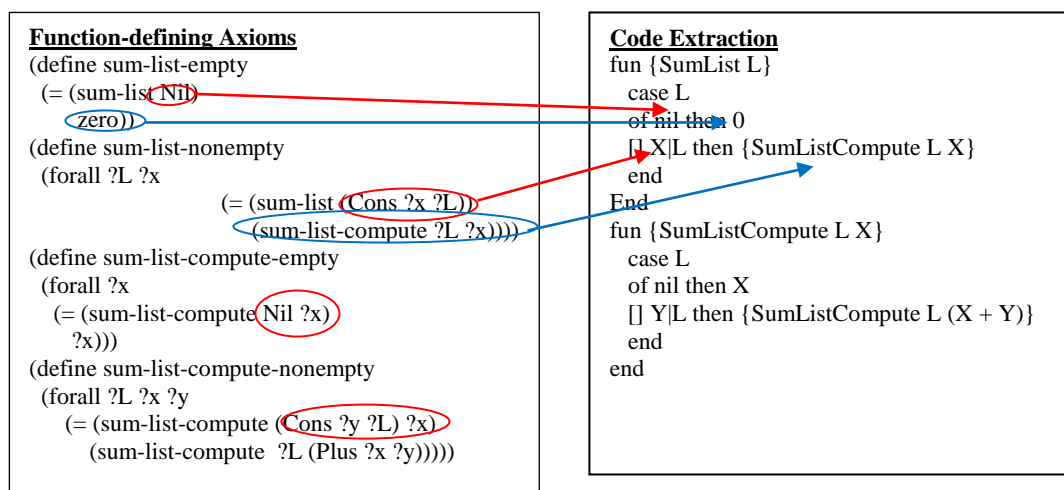


Figure 8.3 Illustration of Code Extraction

CHAPTER 9

CONCLUSION AND FUTURE WORKS

In this thesis, we formed libraries that contain specifications of small-scale functions which can be used to compose more complex programs. We selected functions to represent different problem domains and then defined them axiomatically in the Athena theorem prover and proof checker. Many theorems are proved. We showed that some of these theorems are used in the proof of more complex properties. This is useful even in the case of using automatic theorem provers which can use existing theorems to find the proof others. Our definitions are done both in infix and prefix notation. In many cases it is easier to understand definitions and proofs in infix form. But since Athena allows us to use mixture of these notations, we used them both when needed.

We presented proofs of correctness and safety properties of functions that are defined axiomatically. We wrote formal definitions of some mathematical functions (some of which represent several EXCEL functions). Java's substring method is formally defined and used in the proof of related properties. This and related definitions can be used to prove the correctness and safety conditions of some programs which use these structures after some extensions. An infix definition of Memory is also provided to make it possible for one to write specifications of functions that read and update memory.

Our contributions include a Java program that converts definitions written in infix (and mixture of infix and prefix) form to their equivalent in prefix notation. Since the code generator in Athena extracts code from the definitions only in prefix form, this step was necessary. In addition, we also showed that the size of the definitions and proofs in infix form is shorter than the prefix versions in Athena.

In a broader time there are many things that can be done to improve this project. More complex software and properties can be proved by using the definitions in the libraries. We can also develop more formal definitions of EXCEL functions and seek for a method to show the correctness of a given formula using an automatic theorem prover. This project can be improved so that code in a high-level language like Java can be generated automatically from a given set of specifications. When this step is accomplished, user can build a library of imperative language from already proved functions. This library then is going to be a safe and correct library of imperative language. If this step also has been accomplished, user can build a plugin that can prove the safety or correctness of high-level code as in imperative languages.

A wider approach is to create a medium file extension that can be opened in many proof assistant software. This way users can choose proof assistant that is convenient to them but still others will be able to use his/her work using different proof assistant. If this project could be done, people will be able to create a huge library of proofs that does not depend on the particular proof assistant.

REFERENCES

- [1] Wallace, D. R., & Fujii, R. U. (1989). Software verification and validation: an overview. *IEEE Software*, 6(3), 10-17
- [2] Sommerville, I. (2011). *Software engineering* (9th ed.). Boston: Pearson
- [3] Athena. (n.d.). Overview. Retrieved July 9, 2014, from <http://www.proofcentral.org/athena/index.html>
- [4] Hall, A. (1990). Seven Myths of Formal Methods. *IEEE Software*, 7(5), 11.
- [5] ACL2 Version 6.4. (n.d.). ACL2 Version 6.4. Retrieved May 20, 2014, from <http://www.cs.utexas.edu/users/moore/acl2/>
- [6] The Coq Proof Assistant. (n.d.). Welcome ! | The Coq Proof Assistant. Retrieved May 21, 2014, from <http://coq.inria.fr/>
- [7] Isabelle. (n.d.). Isabelle. Retrieved May 21, 2014, from <http://isabelle.in.tum.de/>
- [8] Scott, D. S. (Ed.). (2006). *The seventeen provers of the world*. Heidelberg: Springer.
- [9] Geuvers, H. (2009). Proof assistants: History, ideas and future. *Sadhana*, 34(1), 3-25.
- [10] Musser, D., & Vargun, A. (2003). Proving theorems with Athena
- [11] Vargun, A. (2007, November). Termination checking without using an ordering relation. In *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications* (pp. 130-135). ACTA Press.
- [12] Vargun, A. (2006). Code-carrying theory (Doctoral dissertation, Rensselaer Polytechnic Institute).
- [13] Arkoudas, K. (1999). An Athena tutorial. Unpublished manuscript.
- [14] Necula, G. C., & Lee, P. (1996). Proof-carrying code. Pittsburgh, Pa.: School of Computer Science, Carnegie Mellon University
- [15] Necula, G. C., & Lee, P. (1996). Proof-carrying code. Pittsburgh, Pa.: School of Computer Science, Carnegie Mellon University
- [16] MIN. (n.d.). - *Excel*. Retrieved July 2, 2014, from <http://office.microsoft.com/en-us/excel-help/min-HP005209176.aspx?CTT=5&origin=HP003056100>

- [17] Class String. (n.d.). . Retrieved May 22, 2014, from
[http://docs.oracle.com/javase/7/docs/api/java/lang/String.html#substring\(int, int\)](http://docs.oracle.com/javase/7/docs/api/java/lang/String.html#substring(int, int))
- [18] Gosling, J. (Ed.). (2000). The Java language specification. Addison-Wesley Professional.

APPENDIX A

LIBRARY OF SPECIFICATIONS IN ATHENA

A.1 PREFIX VERSION OF FACTORIAL AND FACTORIAL EFFICIENT FUNCTION

Specify Factorial Function
<pre>(declare Factorial (-> (Nat) Nat)) (define Factorial-zero-axiom (= (Factorial zero) (succ zero))) (define Factorial-succ-axiom (forall ?x (= (Factorial (succ ?x)) (Times (succ ?x) (Factorial ?x))))) (assert Factorial-zero-axiom) (assert Factorial-succ-axiom)</pre>
Define Efficient Factorial Function
<pre>(declare FactorialEfficientHelper (-> (Nat Nat) Nat)) (define FactorialEfficientHelper-zero-axiom (forall ?res (= (FactorialEfficientHelper zero ?res) ?res))) (define FactorialEfficientHelper-succ-axiom (forall ?n ?res (= (FactorialEfficientHelper (succ ?n) ?res) (FactorialEfficientHelper ?n (Times (succ ?n) ?res))))) (declare FactorialEfficient (-> (Nat) Nat)) (define FactorialEfficient-axiom (forall ?n (= (FactorialEfficient ?n) (FactorialEfficientHelper ?n (succ zero))))) (assert FactorialEfficientHelper-zero-axiom) (assert FactorialEfficientHelper-succ-axiom) (assert FactorialEfficient-axiom)</pre>

Define the correctness condition

```
(define Equality
  (forall ?n
    (= (Factorial ?n)
       (FactorialEfficient ?n))))
```

Proof

```
(by-induction
 Equality
 (zero
  (conclude (= (Factorial zero)
               (FactorialEfficient zero))
            (dseq
             (!setup left (Factorial zero))
             (!setup right (FactorialEfficient zero))
             (!reduce left (succ zero) Factorial-zero-axiom)
             (!reduce right (FactorialEfficientHelper zero (succ zero))
                           FactorialEfficient-axiom)
             (!reduce right (succ zero)
                           FactorialEfficientHelper-zero-axiom)
             (!combine left right))))
 ((succ n)
  (conclude (= (Factorial (succ n))
               (FactorialEfficient (succ n)))
            (dlet ((induction-hypothesis
                    (= (Factorial n)
                       (FactorialEfficient n))))
              (dseq
               (!setup left (Factorial (succ n)))
               (!setup right (FactorialEfficient (succ n)))
               (!reduce left (Times (succ n) (Factorial n))
                           Factorial-succ-axiom)
               (!reduce left (Times (succ n) (FactorialEfficient n))
                           induction-hypothesis)
               (!reduce left (Times (succ n) (FactorialEfficientHelper n (succ zero)))
                           FactorialEfficient-axiom)
               (!reduce right (FactorialEfficientHelper (succ n) (succ zero))
                           FactorialEfficient-axiom)
               (!reduce right (FactorialEfficientHelper n (Times (succ n) (succ zero)))
                           FactorialEfficientHelper-succ-axiom))
```

We cannot continue the proof by using only the existing axioms.

We needed to prove a sub-property to be used in the proof of the correctness proof.

We take the term in the last left and the last right side from the proof and constructed the sub-property

Then we used this property to complete the proof.

```
(define P1
  (forall ?n ?a
    (= (FactorialEfficientHelper ?n (Times ?a (succ zero)))
       (Times ?a (FactorialEfficientHelper ?n (succ zero))))))
```

```
(by-induction
 P1
 (zero
  (conclude (forall ?a
              (= (FactorialEfficientHelper zero (Times ?a (succ zero)))
                 (Times ?a (FactorialEfficientHelper zero (succ zero))))
            (pick-any a
              (dseq
               (!setup left (FactorialEfficientHelper zero (Times a (succ zero))))
               (!setup right (Times a (FactorialEfficientHelper zero (succ zero))))
               (!reduce left (Times a (succ zero))
                           FactorialEfficientHelper-zero-axiom)
               (!reduce right (Times a (succ zero))
                           FactorialEfficientHelper-zero-axiom)
               (!combine left right))))
 ((succ n)
```

```

(conclude (forall ?a (= (FactorialEfficientHelper (succ n) (Times ?a (succ zero)))
                      (Times ?a (FactorialEfficientHelper (succ n) (succ zero)))))
(dlet ((induction-hypothesis
       (forall ?a (= (FactorialEfficientHelper n (Times ?a (succ zero)))
                    (Times ?a (FactorialEfficientHelper n (succ zero))))))
(pick-any a
 (dseq
  (!setup left (FactorialEfficientHelper (succ n) (Times a (succ zero))))
  (!setup right (Times a (FactorialEfficientHelper (succ n) (succ zero))))
  (!reduce left (FactorialEfficientHelper n (Times (succ n) (Times a (succ zero))))
            FactorialEfficientHelper-succ-axiom)
  (!expand left (FactorialEfficientHelper n (Times (Times (succ n) a) (succ zero)))
              Times-Associativity)
  (!reduce left (Times (Times (succ n) a) (FactorialEfficientHelper n (succ zero)))
              induction-hypothesis)
  (!reduce right (Times a (FactorialEfficientHelper n (Times (succ n) (succ zero))))
                FactorialEfficientHelper-succ-axiom)
  (!reduce right (Times a (Times (succ n) (FactorialEfficientHelper n (succ zero))))
                induction-hypothesis)
  (!expand right (Times (Times a (succ n)) (FactorialEfficientHelper n (succ zero)))
                Times-Associativity)
  (!reduce right (Times (Times (succ n) a) (FactorialEfficientHelper n (succ zero)))
                Times-Commutativity)
  (!combine left right))))))

```

The intermediate property P1 is proved first so that we use P1 to complete the correctness proof of the factorial

```

(!reduce right (Times (succ n) (FactorialEfficientHelper n (succ zero)))
 P1)
(!combine left right))))))

```

A.2 AXIOMATIC DEFINITION OF MINIMUM FUNCTION IN INFIX NOTATION WITH SAMPLE PROOFS

```

# The min function returns the minimum of two natural numbers.
(load-file "C:/athena/lib/main/nat-less.ath")

extend-module N {
declare Min: [N N] -> N

module Min {
assert less2 := (forall ?x ?y . ?x < ?y ==> (Min ?x ?y) = ?x)
assert not-less2 :=
  (forall ?x ?y . ~ (?x < ?y) ==> (Min ?x ?y) = ?y)
define commutative := (forall ?x ?y . (Min ?x ?y) = (Min ?y ?x))
define associative :=
  (forall ?x ?y ?z . (Min (Min ?x ?y) ?z) = (Min ?x (Min ?y ?z)))

conclude commutative
pick-any x:N y:N
conclude ((Min x y) = (Min y x))
  (!two-cases
  assume (x < y)
  let { _ :=
    (!chain-last [(x < y) ==> (~ (y < x)) [Less.asymmetric]])}
  (!chain [(Min x y)
    --> x [less2]
    <-- (Min y x) [not-less2]])
  assume (~ (x < y))
  (!two-cases
  assume (y = x)
  (!chain [(Min x y)
    --> (Min y y) [(y = x)]
    <-- (Min y x) [(y = x)]]))
  assume (y /= x)
  let { _ :=
    (!chain-last
    [(y /= x)
    ==> (x /= y) [sym]
    ==> (~ (x < y) & (x /= y)) [augment]
    ==> (y < x) [Less.trichotomy]])}
  (!chain [(Min x y)
    --> y [not-less2]
    <-- (Min y x) [less2]]))

#---
conclude associative
pick-any x:N y:N z:N
  (!two-cases
  assume (x < y)
  (!two-cases
  assume (x < z)
  let {e1 := (!chain [(Min (Min x y) z)
    --> (Min x z) [less2]
    --> x [less2]]);
    e2 := conclude ((Min x (Min y z)) = x)
    (!two-cases
    assume (y < z)
    (!chain [(Min x (Min y z))
    --> (Min x y) [less2]
    --> x [less2]])
    assume (~ (y < z))
    (!chain [(Min x (Min y z))
    --> (Min x z) [not-less2]
    --> x [less2]]))}
  (!combine-equations e1 e2)
  assume (~ (x < z))
  let {e3 := (!chain [(Min (Min x y) z)
    --> (Min x z) [less2]

```

```

--> z [not-less2]]);
_ := (!chain-last
  [ (~ (x < z))
    ==> (x < y & ~ (x < z)) [augment]
    ==> (z < y) [Less.transitive2]]);
e4 := conclude ((Min x (Min y z)) = z)
  (!chain [(Min x (Min y z))
    --> (Min x (Min z y)) [commutative]
    --> (Min x z) [less2]
    --> z [not-less2]]})
  (!combine-equations e3 e4))
assume (~ (x < y))
  (!two-cases
    assume (z < y)
      let {e5 := (!chain [(Min (Min x y) z)
        --> (Min y z) [not-less2]
        --> (Min z y) [commutative]
        --> z [less2]]);
        _ := (!chain-last
          [(z < y)
            ==> (~ (x < y) & (z < y)) [augment]
            ==> (z < x) [Less.transitive1]]);
          e6 := conclude ((Min x (Min y z)) = z)
            (!chain
              [(Min x (Min y z))
                --> (Min x (Min z y)) [commutative]
                --> (Min x z) [less2]
                --> (Min z x) [commutative]
                --> z [less2]]})
            (!combine-equations e5 e6))
          assume (~ (z < y))
            (!two-cases
              assume (x < z)
                (!combine-equations
                  (!chain [(Min (Min x y) z)
                    --> (Min y z) [not-less2]
                    --> (Min z y) [commutative]
                    --> y [not-less2]]))
                  (!chain [(Min x (Min y z))
                    --> (Min x (Min z y)) [commutative]
                    --> (Min x y) [not-less2]
                    --> y [not-less2]]))
                assume (~ (x < z))
                  (!combine-equations
                    (!chain [(Min (Min x y) z)
                      --> (Min y z) [not-less2]
                      --> (Min z y) [commutative]
                      --> y [not-less2]]))
                    (!chain [(Min x (Min y z))
                      --> (Min x (Min z y)) [commutative]
                      --> (Min x y) [not-less2]
                      --> y [not-less2]]))))))
  )
define idempotent := (forall ?x . (Min ?x ?x) = ?x)

conclude idempotent
  pick-any x:N
    (!chain-last [true ==> (~ (x < x)) [Less.irreflexive]
      ==> ((Min x x) = x) [not-less2]])
} # Min
} # N

```

A.3 AXIOMATIC DEFINITION OF SUMLIST FUNCTION IN INFIX NOTATION WITH SAMPLE PROOFS

```

# The sumlist function returns the sum of natural numbers in a list.

load "list-of-temp.ath";
load-file "nat-minimum.ath";

declare sum-list@ : (N) [(List N)] -> N;;

define sum-list@-empty := ((sum-list@ nil) = zero);;

define sum-list@-nonempty :=
  (forall ?x:N ?L:(List N) . (sum-list@ (?x :: ?L))
   = (?x + (sum-list@ ?L)));;

(assert sum-list@-empty sum-list@-nonempty)

declare sum-list : (T)
  [(List T)]
  -> T;;

declare sum-list-compute : (T) [(List T) T] -> T;;

define sum-list-compute-empty :=
  (forall ?x .
   (sum-list-compute nil ?x)
   = ?x);;

define sum-list-compute-nonempty :=
  (forall ?L ?x ?y .
   (sum-list-compute (?y :: ?L) ?x)
   = (sum-list-compute ?L (?x + ?y)));;

(assert sum-list-compute-empty sum-list-compute-nonempty)

define sum-list-empty := ((sum-list nil)
  = zero);;

define sum-list-nonempty :=
  (forall ?L:(List N) ?x:N .
   (sum-list (?x :: ?L))
   = (sum-list-compute ?L ?x));;

(define sum-list-rules
  [sum-list-empty sum-list-nonempty])

assert sum-list-rules;;

define sum-list-compute-relation :=
  (forall ?L ?x .
   (sum-list@ (?x:N :: ?L:(List N)))
   = (sum-list-compute ?L ?x));;

by-induction
sum-list-compute-relation {
nil =>
  conclude (forall ?x:N .
    (sum-list@ (?x:N :: nil:(List N)))
    = (sum-list-compute nil ?x))
  pick-any x:N
  (!chain
  [(sum-list@ (x :: nil:(List N)))
  --> (x + (sum-list@ nil)) [sum-list@-nonempty]
  --> (x + zero) [sum-list@-empty]
  --> x [N.right-zero]
  <-- (sum-list-compute nil x) [sum-list-compute-empty]
  ])
| (y :: L) =>

```

```

conclude (forall ?x:N .
  (sum-list@ (?x :: (y :: L)))
  = (sum-list-compute (y :: L) ?x))
let {induction-hypothesis := (forall ?x:N .
  (sum-list@ (?x :: L))
  = (sum-list-compute L ?x))}

pick-any x:N
(!chain
 [(sum-list@ (x :: (y :: L)))
 --> (x + (sum-list@ (y :: L))) [sum-list@-nonempty]
 --> (x + (y + (sum-list@ L))) [sum-list@-nonempty]
 <-- ((x + y) + (sum-list@ L)) [N.associative]
 --> (sum-list@ ((x + y) :: L)) [sum-list@-nonempty]
 <-- (sum-list-compute L (x + y)) [induction-hypothesis]
 <-- (sum-list-compute (y :: L) x) [sum-list-compute-nonempty]
 ])
];;

define sum-list-correctness :=
  (forall ?L:(List N) .
    (sum-list ?L) =
      (sum-list@ ?L));;

by-induction sum-list-correctness{
nil =>
  (!chain
  [(sum-list nil:(List N))
  --> zero [sum-list-empty]
  <--(sum-list@ nil) [sum-list@-empty]])
| (x:N :: L:(List N)) =>
  (!chain
  [(sum-list (x :: L))
  --> (sum-list-compute L x) [sum-list-nonempty]
  <-- (sum-list@ (x :: L)) [sum-list-compute-relation]]));;

```

A.4 AXIOMATIC DEFINITION OF REMOVE FUNCTION IN INFIX NOTATION WITH SAMPLE PROOF

```

# The remove function returns List after specific element is deleted.

load "nat-minlist.ath"

declare RemoveWithIndex : (N) [(List N) N]-> (List N);;

define RemoveWithIndex-zero-axiom
:= (forall ?x:N ?L:(List N) .
    (RemoveWithIndex (?x :: ?L) zero) = ?L)

define RemoveWithIndex-succ-axiom
:= (forall ?x:N ?L:(List N) ?n .
    (RemoveWithIndex (?x :: ?L) (S ?n)) = (RemoveWithIndex ?L ?n))

define RemoveWithIndex-nil-axiom
:= (forall ?x:N ?L:(List N) ?p .
    (RemoveWithIndex nil ?p) = nil)

assert RemoveWithIndex-zero-axiom, RemoveWithIndex-succ-axiom,
RemoveWithIndex-nil-axiom;;

define Property1
:= (forall ?x:N ?L:(List N) .
    (MinList (RemoveWithIndex (?x :: ?L) zero)) = (MinList ?L))

conclude Property1
pick-any x L
(!chain
  [(MinList (RemoveWithIndex (x :: L) zero))
   --> (MinList L) [RemoveWithIndex-zero-axiom]
  ]
)

```

A.5 AXIOMATIC DEFINITION OF ACCESS FUNCTION IN INFIX NOTATION WITH SAMPLE PROOF

```

# The access function returns content specific index in a list.

load "nat-minlist.ath"
load "ordered-list-nat2.ath"

extend-module MinList {

define MinList-first
:= (forall ?x:N ?L:(List N) . (?x:N N.Less= (MinList ?L:(List N) ))
   => (MinList (?x :: ?L)) = ?x )

define OrderedMinList
:= (forall ?x ?L . (List.ordered (?x :: ?L))
   => (?x N.Less= (MinList ?L)))

assert MinList-first
assert OrderedMinList

} # MinList

declare Access : (N) [(List N) N]-> N

module Access {
open MinList
define Access-first-axiom
:= (forall ?x:N ?L:(List N) .
   (Access (?x :: ?L) zero) = ?x)

assert Access-first-axiom

define Access-Cons-axiom
:= (forall ?x:N ?L:(List N) ?n:N .
   (Access (?x :: ?L) (S ?n)) = (Access ?L ?n))

assert Access-Cons-axiom

define Property1
:= (forall ?x:N ?L:(List N) . (List.ordered (?x :: ?L))
   => (Access (?x :: ?L) zero) = (MinList (?x :: ?L)))

conclude Property1
pick-any x:N L:(List N)
assume test := (List.ordered (x :: L))
(!by-contradiction ((Access (x :: L) zero) = (MinList (x :: L))))
assume (~((Access (x :: L) zero) = (MinList (x :: L))))
(!absurd
  conclude ((Access (x :: L) zero) = (MinList (x :: L)))
  let {A := (!mp (!uspec* OrderedMinList [x L])
             test )}
  let {B := (!mp (!uspec* MinList-first [x L]) A)}
  (!chain [(Access (x :: L) zero)
           --> x [Access-first-axiom]
           <-- (MinList (x :: L)) [B]
           ])
  (~((Access (x :: L) zero) = (MinList (x :: L))))
))

} # Access

```


A.6 AXIOMATIC DEFINITION OF MEMORY WITH SAMPLE PROOFS IN INFIX NOTATION

The memory is represented as a List of natural numbers. We have defined a function called `access` to receive the content of a memory location. In addition we also defined a function called `assign` to a natural number to a location in memory. The `swap` function is used for swapping the contents of two memory locations.

```

load "nat-minlist.ath";;
load "list-of-temp.ath";;

declare access : (N) [(List N) N] -> N;;
declare assign : (N) [(List N) N N] -> (List N);;
declare swap : (N) [(List N) N N] -> (List N);;

define assign-axiom1 :=
  (forall ?M ?a ?x .
    (access (assign ?M ?a ?x) ?a)
    = ?x);;

define assign-axiom2 :=
  (forall ?M ?a ?b ?x .
    (~ (= ?a ?b))
    ==> (access (assign ?M ?a ?x) ?b)
    = (access ?M ?b));;

define swap-axiom1 :=
  (forall ?M ?a ?b . (access (swap ?M ?a ?b) ?a)
    = (access ?M ?b));;

define swap-axiom2 :=
  (forall ?M ?a ?b .
    (access (swap ?M ?a ?b) ?b)
    = (access ?M ?a));;

define swap-axiom3 :=
  (forall ?M ?a ?b ?c .
    ((~ (?c = ?a)) & (~ (?c = ?b)))
    ==> (access (swap ?M ?a ?b) ?c)
    = (access ?M ?c));;

assert assign-axiom1, assign-axiom2, swap-axiom1, swap-axiom2, swap-axiom3;;

define Memory-equality :=
  (forall ?M1 ?M2 .
    (?M1 = ?M2) <==>
    (forall ?a .
      (access ?M1 ?a)
      = (access ?M2 ?a)))
assert Memory-equality;;

define Double-assign :=
  (forall ?b ?M ?a ?x ?y .
    (access (assign (assign ?M ?a ?x) ?a ?y) ?b)
    = (access (assign ?M ?a ?y) ?b));;

conclude Double-assign
pick-any b M a x y
(dseq
  (!two-cases
    assume (a = b)
    (!chain
      [(access (assign (assign M a x) a y) b)
        --> (access (assign (assign M a x) a y) a) [(a = b)]
        --> y [assign-axiom1]
        <-- (access (assign M a y) a) [assign-axiom1]
        <-- (access (assign M a y) b) [(= a b)]]
      )
    assume (~ (a = b))
    (!chain

```

```

        [(access (assign (assign M a x) a y) b)
         --> (access (assign M a x) b) [assign-axiom2]
         --> (access M b) [assign-axiom2]
         <-- (access (assign M a y) b) [assign-axiom2]]
      )
    );;

#-----

define Direct-double-assign :=
  (forall ?M ?a ?x ?y .
    (assign (assign ?M ?a ?x) ?a ?y)
    = (assign ?M ?a ?y));;

conclude Direct-double-assign
pick-any M i x y
(dseq
  conclude
    (forall ?a .
      (access (assign (assign M i x) i y) ?a)
      = (access (assign M i y) ?a))
    pick-any a
      (!spec Double-assign
        [a M i x y])
      (!spec-right Memory-equality
        [(assign (assign M i x) i y)
         (assign M i y)]));;

#-----

define Self-assign :=
  (forall ?M ?a ?b .
    (access (assign ?M ?a (access ?M ?a)) ?b)
    = (access ?M ?b));;

conclude Self-assign
pick-any M a b
(dseq
  (!two-cases
    assume (a = b)
    (!chain
      [(access (assign M a (access M a)) b)
       --> (access (assign M a (access M a)) a) [(a = b)]
       --> (access M a) [assign-axiom1]
       <-- (access M b) [(= a b)]
      ]
    )
    assume (~ (a = b))
    (!chain
      [(access (assign M a (access M a)) b)
       --> (access M b) [assign-axiom2]
      ]
    )
  )
)
);;

#-----

define Direct-self-assign :=
  (forall ?M ?a .
    (assign ?M ?a (access ?M ?a))
    = ?M)

conclude Direct-self-assign
pick-any M i
(dseq
  conclude (forall ?b .
    (access (assign M i (access M i)) ?b)
    = (access M ?b))
  (pick-any b
    (!spec Self-assign [M i b]))
)
)

```

```

      (!spec-right Memory-equality
        [(assign M i (access M i)) M]))
#-----
define Double-swap :=
  (forall ?c ?M ?a ?b .
    (access (swap (swap ?M ?a ?b) ?b ?a) ?c)
    = (access ?M ?c));;
assert Double-swap;;

define Direct-double-swap :=
  (forall ?M ?a ?b .
    (swap (swap ?M ?a ?b) ?b ?a)
    = ?M)

conclude Direct-double-swap
pick-any M a b
(dseq
  conclude (forall ?c .
    (access (swap (swap M a b) b a) ?c)
    = (access M ?c))
  (pick-any c
    (!spec Double-swap
      [c M a b]))
  (!spec-right Memory-equality
    [(swap (swap M a b) b a) M]))

#-----
define Same-swap :=
  (forall ?c ?M ?a ?b .
    (access (swap (swap ?M ?a ?b) ?a ?b) ?c)
    = (access ?M ?c));;
assert Same-swap;;

define Direct-same-swap :=
  (forall ?M ?a ?b .
    (swap (swap ?M ?a ?b) ?a ?b)
    = ?M)

conclude Direct-same-swap
pick-any M a b
(dseq
  conclude (forall ?c .
    (access (swap (swap M a b) a b) ?c)
    = (access M ?c))
  (pick-any c
    (!uspec* Same-swap
      [c M a b]))
  (!spec-right Memory-equality
    [(swap (swap M a b) a b) M]));;

```

A.7 AXIOMATIC DEFINITION OF INDEX FUNCTION IN INFIX NOTATION WITH SAMPLE PROOF

```

#return the first appearance position of element that is found

load "access.ath"

declare ElementLocationHelper: (N) [N (List N) N] -> N;;

declare ElementLocation: (N) [N (List N)] -> N;;

define ElementLocationHelper-Next-axiom
:= (forall ?z:N ?x:N ?L:(List N) ?n:N . (?z in (?x :: ?L)) & (?z /= ?x)
    ==> (ElementLocationHelper ?z (?x :: ?L) ?n)
    = (ElementLocationHelper ?z ?L (S ?n)))

define ElementLocationHelper-True-axiom
:= (forall ?z ?L ?n . (?z in (?z :: ?L))
    ==> (ElementLocationHelper ?z (?z :: ?L) ?n)
    = ?n)

define ElementLocation-axiom
:= (forall ?z ?L . (?z in ?L)
    ==> (ElementLocation ?z ?L)
    = (ElementLocationHelper ?z ?L zero))

assert ElementLocationHelper-Next-axiom, ElementLocationHelper-True-axiom,
ElementLocation-axiom;;

define Property1
:= (forall ?z:N ?L:(List N) . (?z in ?L)
    ==> (Access ?L (ElementLocation ?z ?L)) = ?z)

conclude Property1
pick-any z:N L:(List N)
assume test := (?z in ?L)
(!by-contradiction (Access ?L (ElementLocation ?z ?L)) = ?z)
assume (~(Access ?L (ElementLocation ?z ?L)) = ?z)
(!absurd
  conclude ((Access ?L (ElementLocation ?z ?L)) = ?z)

  (!chain [(Access ?L (ElementLocation ?z ?L))
    <-- (Access (x :: L) (S (ElementLocation ?z ?L)))
    [Access-Cons-axiom]
    <-- z [B]
  ])
  (~(Access ?L (ElementLocation ?z ?L)) = ?z)
))

```

A.8 AXIOMATIC DEFINITION OF SPLIT FUNCTION IN INFIX NOTATION WITH SAMPLE PROOF

```

#SplitLeft List PositionToWhichElementIsTaken(Excluded)
#example: (SplitLeft List 3) will bring element with index 0,1, and 2

load "nat-access.ath"
load "nat-minus.ath"

#SplitLeft List PositionToWhichElementIsTaken(Excluded)
#example: (SplitLeft List 3) will bring element with index 0,1, and 2

declare SplitLeft : (N) [(List N) N]-> (List N)

define SplitLeft-zero-axiom
:= (forall ?L:(List N) .
  (SplitLeft ?L zero) = nil)

define SplitLeft-succ-axiom
:= (forall ?L:(List N) ?n:N .
  (SplitLeft ?L (S ?n)) =
  ((SplitLeft ?L ?n) List.Join ((Access ?L (S ?n)) :: nil)))

define SplitLeft-nil-axiom
:= (forall ?n:N .
  (SplitLeft nil ?n) = nil)

define SplitLeft-cons-axiom
:= (forall ?x:N ?L:(List N) ?n:N .
  (SplitLeft (?x :: ?L) ?n) = (?x :: (SplitLeft ?L (?n N.- N.one))))

assert SplitLeft-zero-axiom, SplitLeft-succ-axiom, SplitLeft-nil-axiom,
SplitLeft-cons-axiom

#-----

#SplitRight List PositionFromWhichElementIsTaken(Included)
#example: (SplitRight List 3) will bring element from index 3 to the end

declare SplitRight : (N) [(List N) N]-> (List N)

define SplitRight-zero-axiom
:= (forall ?L:(List N) .
  (SplitRight ?L zero) = ?L)

define SplitRight-succ?-axiom
:= (forall ?L:(List N) ?n:N .
  (SplitRight ?L ?n) = ((Access ?L ?n) :: (SplitRight ?L (S ?n))))

define SplitRight-nil-axiom
:= (forall ?n:N .
  (SplitRight nil ?n) = nil)

define SplitRight-cons-axiom
:= (forall ?x:N ?L:(List N) ?n:N .
  (SplitRight (?x :: ?L) ?n) = (SplitRight ?L (?n N.- N.one)))

assert SplitRight-zero-axiom, SplitRight-succ?-axiom, SplitRight-nil-axiom,
SplitRight-cons-axiom

#-----

define SplitCombination-property
:= (forall ?L:(List N) ?n:N .
  ((SplitLeft ?L ?n) List.Join (SplitRight ?L ?n)) = ?L)

by-induction SplitCombination-property {
  nil =>

```

```

pick-any n:N
(!chain
  [((SplitLeft nil n) List.Join (SplitRight nil n))
   --> (nil List.Join (SplitRight nil n)) [SplitLeft-nil-axiom]
   --> (nil List.Join nil) [SplitRight-nil-axiom]
   --> nil [List.Join.left-empty]]
)
| (x :: L) =>
let { ind-hyp := (forall ?n:N .
  ((SplitLeft L ?n) List.Join (SplitRight L ?n)) = L ) }
pick-any n:N
(!chain
  [((SplitLeft (x :: L) n) List.Join (SplitRight (x :: L) n))
   --> ((x :: (SplitLeft L (n N.- N.one))) List.Join (SplitRight (x :: L) n))
   [SplitLeft-cons-axiom]
  --> (x :: ((SplitLeft L (n N.- N.one)) List.Join (SplitRight (x :: L) n)))
   [List.Join.left-nonempty]
  --> (x :: ((SplitLeft L (n N.- N.one))
   List.Join (SplitRight L (n N.- N.one))))
   [SplitRight-cons-axiom]
  --> (x :: L)
   [ind-hyp]
])
}
;;

```

A.9 AXIOMATIC DEFINITION OF SUBLIST FUNCTION IN INFIX NOTATION WITH SAMPLE PROOFS

```

#(SubList List PositionFromWhichElementIsTaken(Included) LengthOfElementTaken)
#example: (SubList List 3 4) Will bring element from index 3 up to 4 further
#which are index 3, 4, 5, and 6

load "nat-split.ath"

#(SubList List PositionFromWhichElementIsTaken(Included) LengthOfElementTaken)
#example: (SubList List 3 4) Will bring element from index 3 up to 4 further
#which are index 3, 4, 5, and 6

declare SubList : (N) [(List N) N N] -> (List N)

define SubList-zero-axiom
:= (forall ?L ?l .
  (SubList ?L zero ?l) = (SplitLeft ?L ?l))

define SubList-succ-axiom
:= (forall ?L ?n ?l .
  (SubList ?L (S ?n) ?l) = (SplitRight (SplitLeft ?L ((S ?n) + ?l)) (S ?n)))

define SubList-nil-axiom
:= (forall ?n ?l .
  (SubList nil ?n ?l) = nil)

define SubList-cons-axiom
:= (forall ?x ?L ?n ?l .
  (SubList (?x :: ?L) ?n ?l) = (SplitRight (SplitLeft (?x::?L) (?n + ?l)) ?n))

assert SubList-zero-axiom, SubList-succ-axiom, SubList-nil-axiom ,
SubList-cons-axiom;;

#-----

define SubList-n-property
:= (forall ?n ?L ?l .
  (SubList ?L ?n ?l) = (SplitRight (SplitLeft ?L (?n + ?l)) ?n))

by-induction SubList-n-property {
  zero =>
  pick-any L:(List N) l:N
  (!chain
    [(SubList L zero l)
     --> (SplitLeft L l) [SubList-zero-axiom]
     <-- (SplitRight (SplitLeft L l) zero) [SplitRight-zero-axiom]
     <-- (SplitRight (SplitLeft L (zero + 1)) zero) [N.Plus.left-zero]]
  )
  | (S n) =>
  pick-any L:(List N) l:N
  (!chain
    [(SubList L (S n) l)
     --> (SplitRight (SplitLeft L ((S n) + 1)) (S n)) [SubList-succ-axiom]
    ])
  }

#-----

define SubList-L-property
:= (forall ?L ?n ?l .
  (SubList ?L ?n ?l) = (SplitRight (SplitLeft ?L (?n + ?l)) ?n))

by-induction SubList-L-property {
  nil =>
  pick-any n:N l:N
  (!chain
    [(SubList nil n l)

```

```
    --> nil [SubList-nil-axiom]
    <-- (SplitRight nil n) [SplitRight-nil-axiom]
    <-- (SplitRight (SplitLeft nil (n + 1)) n) [SplitLeft-nil-axiom]]
)
| (x :: L) =>
pick-any n:N l:N
(!chain
 [(SubList (x :: L) n l)
  --> (SplitRight (SplitLeft (x :: L) (n + 1)) n) [SubList-cons-axiom]
])
}
```


A.10 AXIOMATIC DEFINITION OF SUBSTRING FUNCTION IN INFIX NOTATION WITH SAMPLE PROOFS

```

load "nat-split.ath"

#(Substring List PositionFromElementTaken(Included) LastPosition(exclude))
#example: (Substring List 3 5) Will bring element from index 3 and 4
#which are index 3 and 4

declare Substring : (N) [(List N) N N] -> (List N)

define Substring-zeron-axiom
:= (forall ?L ?f . ?f N.<= (List.Length ?L)
  ==> (Substring ?L zero ?f) = (SplitLeft ?L ?f))

define Substring-succn-axiom-if
:= (forall ?L ?n ?f .
  (S ?n) N.<= ?f & ?f N.<= (List.Length ?L)
  ==> (Substring ?L (S ?n) ?f) = (SplitRight (SplitLeft ?L ?f) (S ?n)))

define Substring-n-Length-axiom
:= (forall ?L ?n . ?n N.<= (List.Length ?L)
  ==> (Substring ?L ?n (List.Length ?L)) = (SplitRight ?L ?n))

define Substring-equalinput-Length-axiom
:= (forall ?L ?n . ?n = ?f
  ==> (Substring ?L ?n ?f) = nil)

assert Substring-n-Length-axiom, Substring-zeron-axiom,
  Substring-equalinput-Length-axiom, Substring-succn-axiom-if;;
#-----

define Substring-property1
:= (forall ?n:N ?L ?f:N . ?n N.<= ?f & ?f N.<= (List.Length ?L)
  ==>(Substring ?L ?n ?f) = (SplitRight (SplitLeft ?L ?f) ?n))

conclude Substring-property1
by-induction Substring-property1 {
zero =>
pick-any L:(List N) f:N
assume test := (zero N.<= f & f N.<= (List.Length L))
(!by-contradiction ((Substring L zero f) = (SplitRight (SplitLeft L f) zero))
assume ((Substring L zero f) /= (SplitRight (SplitLeft L f) zero))
(!absurd
  conclude ((Substring L zero f) = (SplitRight (SplitLeft L f) zero))
  let {A := (!mp (!uspec* Substring-zeron-axiom [L f])
    (!right-and test) )}
  (!chain
    [(Substring L zero f)
     --> (SplitLeft L f) [A]
     <-- (SplitRight (SplitLeft L f) zero) [SplitRight-zero-axiom]]
  )
  ((Substring L zero f) /= (SplitRight (SplitLeft L f) zero))
))
| (S n) =>
pick-any L:(List N) f:N
assume test := ((S n) N.<= f & f N.<= (List.Length L))
(!by-contradiction ((Substring L (S n) f) = (SplitRight (SplitLeft L f) (S n)))
assume ((Substring L (S n) f) /= (SplitRight (SplitLeft L f) (S n)))
(!absurd
conclude ((Substring L (S n) f) = (SplitRight (SplitLeft L f) (S n)))
  let {B := (!mp (!uspec* Substring-succn-axiom-if [L n f])
    test)}
  (!chain

```

```

    [(Substring L (S n) f)
     --> (SplitRight (SplitLeft L f) (S n))    [B]]
  )

  ((Substring L (S n) f) =/= (SplitRight (SplitLeft L f) (S n)))
))
}

define full-list-property
:= (forall ?L ?n . ?n N.<= (List.Length ?L)
   ==> ((Substring ?L zero ?n)
        List.Join (Substring ?L ?n (List.Length ?L))) = ?L)

conclude full-list-property
by-induction full-list-property {
nil =>
pick-any n:N
assume test := (n N.<= (List.Length nil:(List N)))
(!by-contradiction ((Substring nil zero n)
  List.Join (Substring nil n (List.Length nil:(List N)))) = nil)
assume ((Substring nil zero n)
  List.Join (Substring nil n (List.Length nil:(List N)))) =/= nil)
(!absurd
conclude ((Substring nil zero n)
  List.Join (Substring nil n (List.Length nil:(List N)))) = nil)
let {A := (!mp (!uspec* Substring-zeron-axiom [nil n])
  test )}
let {B := (!mp (!uspec* Substring-n-Length-axiom [nil n])
  test )}
(!chain
  [((Substring nil zero n)
    List.Join (Substring nil n (List.Length nil:(List N))))
   -->((SplitLeft nil n)
    List.Join (Substring nil n (List.Length nil:(List N))))
   [A]
   --> ((SplitLeft nil n) List.Join (SplitRight nil n))
   [B]
   --> (nil List.Join (SplitRight nil n))[SplitLeft-nil-axiom]
   --> (nil List.Join nil)[SplitRight-nil-axiom]
   <-- nil:(List N) [List.Join.left-empty]]
)

  ((Substring nil zero n)
   List.Join (Substring nil n (List.Length nil:(List N)))) =/= nil)
)
| (x:N :: L) =>
pick-any n:N
assume test := (n N.<= (List.Length (x :: L)))
(!by-contradiction ((Substring (x :: L) zero n)
  List.Join (Substring (x :: L) n (List.Length (x :: L)))) = (x :: L))
assume ((Substring (x :: L) zero n)
  List.Join (Substring (x :: L) n (List.Length (x :: L)))) =/= (x :: L))
(!absurd
conclude ((Substring (x :: L) zero n)
  List.Join (Substring (x :: L) n (List.Length (x :: L)))) = (x :: L))
let {A := (!mp (!uspec* Substring-zeron-axiom [(x :: L) n])
  test )}
let {B := (!mp (!uspec* Substring-n-Length-axiom [(x :: L) n])
  test )}
(!chain
  [((Substring (x :: L) zero n)
    List.Join (Substring (x :: L) n (List.Length (x :: L))))
   --> ((SplitLeft (x :: L) n)
    List.Join (Substring (x :: L) n (List.Length (x :: L))))
   [A]
   --> ((SplitLeft (x :: L) n) List.Join (SplitRight (x :: L) n))
   [B]
   --> (x :: L)    [SplitCombination-property]
)
])

```

```
((Substring (x :: L) zero n)
  List.Join (Substring (x :: L) n (List.Length (x :: L)))) /= (x :: L)
))
}
```

A.11 AXIOMATIC DEFINITION OF MINLIST FUNCTION IN INFIX NOTATION WITH SAMPLE PROOFS

```

load "nat-minimum.ath"
load "list-of-temp.ath"

declare MinList: (N) [(List N)] -> N

module MinList {
open N
assert MinList-One-axiom
      := (forall ?x:N . (MinList (?x :: nil))
      = ?x)

assert MinList-Cons-axiom
      := (forall ?x ?L . (MinList (?x :: ?L))
      = (Min ?x (MinList ?L)))

assert MinList-ConsMin-property
      := (forall ?x ?y . (MinList (?x :: (?y :: nil)))
      = (Min ?x ?y))

define MinListMin-Relation-property
      := (forall ?L ?x ?y . (MinList (?x :: (?y :: ?L)))
      = (MinList ((Min ?x ?y) :: ?L)))

by-induction MinListMin-Relation-property{
nil =>
  pick-any x:N y:N
  (!chain [(MinList (x :: (y :: nil)))
    -->(N.Min x y) [MinList-ConsMin-property]
    <--(MinList ((N.Min x y) :: nil)) [MinList-One-axiom]])
| (z :: p) =>
  pick-any x:N y:N
  (!chain [(MinList (x :: (y :: (z :: p))))
    -->(Min x (MinList (y :: (z :: p)))) [MinList-Cons-axiom]
    -->(Min x (N.Min y (MinList (z :: p)))) [MinList-Cons-axiom]
    <--(Min (Min x y) (MinList (z :: p))) [Min.associative]
    <--(MinList ((Min x y) :: (z :: p))) [MinList-Cons-axiom]])
}

define MinList-Min-property
      := (forall ?L ?x:N ?y:N . (N.Min ?x (MinList (?y :: ?L)))
      = (MinList (?x:N :: (?y :: ?L))))

by-induction MinList-Min-property {
nil =>
  pick-any x:N y:N
  (!chain [(Min x (MinList (y :: nil)))
    -->(Min x y) [MinList-One-axiom]
    <--(MinList (x :: (y :: nil))) [MinList-ConsMin-property]]
| (z :: p) =>
  pick-any x:N y:N
  (!chain [(Min x (MinList (y :: (z :: p))))
    <-- (MinList (x :: (y :: (z :: p)))) [MinList-Cons-axiom]
  ]})
}}

define MinList-Join-property
      := (forall ?L1 ?x:N ?y:N ?L2 .
      (Min (MinList (?x :: ?L1)) (MinList (?y :: ?L2)))
      = (MinList ((?x :: ?L1) List.Join (?y :: ?L2))))

```

```

by-induction MinList-Join-property {
  nil =>
  pick-any x:N y:N L2
  (!chain
    [(Min (MinList (x :: nil)) (MinList (y :: L2)))
     --> (Min x (MinList (y :: L2))) [MinList-One-axiom]
     --> (MinList (x :: (y :: L2))) [MinList-Min-property]
     <-- (MinList (x :: (nil List.Join (y :: L2)))] [List.Join.left-empty]
     <-- (MinList ((x :: nil) List.Join (y :: L2))] [List.Join.left-nonempty]]
  )
  | (z :: L1) =>

let { ind-hyp := (forall ?x:N ?y:N ?L2 .
                 (Min (MinList (?x :: L1)) (MinList (?y :: ?L2)))
                  = (MinList ((?x :: L1) List.Join (?y :: ?L2))))}
  pick-any x y L2
  (!chain
    [(Min (MinList (x :: (z :: L1))) (MinList (y :: L2)))
     --> (Min (Min x (MinList (z :: L1))) (MinList (y :: L2)))
     [MinList-Cons-axiom]
     --> (Min x (Min (MinList (z :: L1)) (MinList (y :: L2))))
     [Min.associative]
     --> (Min x (MinList ((z :: L1) List.Join (y :: L2))))
     [ind-hyp]
     <-- (MinList ((x :: ((z :: L1) List.Join (y :: L2))))
     [MinList-Cons-axiom]
     <-- (MinList ((x :: (z :: L1)) List.Join (y :: L2)))
     [List.Join.left-nonempty]
  ])
}

}#MinList

```

A.12 AXIOMATIC DEFINITION OF FACTORIAL FUNCTION IN INFIX NOTATION WITH SAMPLE PROOFS

```

load "C:/athena/lib/main/nat-times.ath"

#-----
declare Factorial: [N] -> N

assert Factorial-zero-axiom := ((Factorial zero) = (S zero))

assert Factorial-succ-axiom :=
  (forall ?x . (Factorial (S ?x)) = (S ?x) * (Factorial ?x))
#-----

declare FactorialEfficientHelper : [N N] -> N

assert FactorialEfficientHelper-zero-axiom :=
  (forall ?res . (FactorialEfficientHelper zero ?res) = ?res)

assert FactorialEfficientHelper-succ-axiom :=
  (forall ?n ?res . (FactorialEfficientHelper (S ?n) ?res)
    = (FactorialEfficientHelper ?n ((S ?n) * ?res)))

declare FactorialEfficient : [N] -> N

assert FactorialEfficient-axiom :=
  (forall ?n . (FactorialEfficient ?n)
    = (FactorialEfficientHelper ?n (S zero)))

#-----
define P1 :=
  (forall ?n ?a .
    (FactorialEfficientHelper ?n (?a * (S zero)))
    = (?a * (FactorialEfficientHelper ?n (S zero))))
#-----

by-induction P1 {
  zero =>

  pick-any a
  (!chain [(FactorialEfficientHelper zero (a * (S zero)))
    --> (a * (S zero))
    [FactorialEfficientHelper-zero-axiom]
    <-- (a * (FactorialEfficientHelper zero (S zero)))
    [FactorialEfficientHelper-zero-axiom]])

| (S n) =>
  let {induction-hypothesis :=
    (forall ?a .
      (FactorialEfficientHelper n (?a * (S zero)))
      = (?a * (FactorialEfficientHelper n (S zero))))}
  pick-any a
  (!chain
    [(FactorialEfficientHelper (S n) (a * (S zero)))
    -->(FactorialEfficientHelper n ((S n) * (a * (S zero))))
    [FactorialEfficientHelper-succ-axiom]
    <--(FactorialEfficientHelper n ((S n) * a) * (S zero))
    [N.Times.associative]
    -->(((S n) * a) * (FactorialEfficientHelper n (S zero)))
    [induction-hypothesis]
    -->((a * (S n)) * (FactorialEfficientHelper n (S zero)))
    [N.Times.commutative]
    <--(a * ((S n) * (FactorialEfficientHelper n (S zero))))
    [N.Times.associative]
    <--(a * (FactorialEfficientHelper n ((S n) * (S zero))))
    [induction-hypothesis]
  ])
}

```


APPENDIX B

INFIX TO PREFIX CONVERTER CODES

B.1 INFIX TO PREFIX MAIN CLASS

```
package newinfthopre;

import java.io.DataInputStream;
import java.util.ArrayList;

/**
 *
 * @author TOSHÄ°BA
 */
public class NewInfToPre {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        lexical lexClass = new lexical();
        String programInIsmi = "C:\\Users\\TOSHIBA\\Desktop\\ORNEK.txt";

        lexClass.program = lexClass.programiOku(programInIsmi);
        lexClass.getCharInput();
        while (lexClass.nextToken != lexClass.EOF) {
            lexClass.lex();
        }
        System.out.println("Array with space " + lexClass.arrayListString);

        ArrayList<String> newArray = new ArrayList<String>();
        for (int i = 0; i < lexClass.arrayListString.size(); i++) {
            if (!lexClass.arrayListString.get(i).equals(" ")) {
                newArray.add(lexClass.arrayListString.get(i));
            }
        }
        System.out.println("new array " + newArray);
        boolean notEqualSign = false;
        ArrayList<String> newArrayStatement = new ArrayList<String>();
        for (int j = 0; j < newArray.size(); j++) {
            //no quantifier
            if (newArray.get(j).equals(":=")
                && !newArray.get(j + 2).equals("forall")
                && !newArray.get(j + 2).equals("exists")) {
                for (int k = j + 1; k < newArray.size() - 1; k++) {
                    newArrayStatement.add(newArray.get(k));
                }
            }
            break;
            //one quantifier
        } else if (newArray.get(j).equals(".")
            && !newArray.get(j + 3).equals("exists")
            && !newArray.get(j + 2).equals("exists")) {
```



```

        for (int k = j + 1; k < newArray.size() - 1; k++) {
            newArrayStatement.add(newArray.get(k));
        }
        //quantifier and
    } else if (newArray.get(j).equals(".")
        && (newArray.get(j + 2).equals("exists")
            || newArray.get(j + 3).equals("exists"))) {
        j++;
        while (!newArray.get(j).equals(".")) {
            j++;
        }
        for (int k = j + 1; k < newArray.size() - 2; k++) {
            newArrayStatement.add(newArray.get(k));
        }
        //forall not exists or exists not exists
    } else if (newArray.get(j).equals(".")
        && newArray.get(j + 1).equals("~")
        && newArray.get(j + 3).equals("exists")) {
        j++;
        System.out.println(newArray.get(j));
        while (!newArray.get(j).equals(".")) {
            j++;
        }
        for (int k = j + 1; k < newArray.size() - 2; k++) {
            newArrayStatement.add(newArray.get(k));
        }
    }
}

System.out.println("newArray Statement" + newArrayStatement);

//read statement and turn into prefix
ArrayList<String> s;
int n;
Evaluation inf;
DataInputStream inp = new DataInputStream(System.in);
s = newArrayStatement;
inf = new Evaluation(s);
ArrayList<String> prefixNotation = inf.inToPre();

System.out.println("Prefix " + prefixNotation);

//
//          give parenthesis
ArrayList<String> str = prefixNotation;

parenth baru = new parenth(str);
baru.i = 0;
baru.nextChar = str.get(baru.i++);
if (baru.nextChar.equals("(")) {
    baru.solParantez();
} else {
    baru.operator();
}
System.out.println("with parenth " + baru.output);

// convert to prefix full
boolean forallAndExistChecker = false;
boolean forallOrExistChecker = false;
boolean forallAndNotExistChecker = false;
String converted = "(";
for (int i = 0; i < newArray.size(); i) {
    //replace ~ to not
    if (newArray.get(i).equals("~")) {
        newArray.set(i, "not");
        //no quantifier
    } else if (newArray.get(i).equals(":=")
        && !newArray.get(i + 2).equals("forall"))

```

```

        && !newArray.get(i + 2).equals("exists")) {
    forallOrExistChecker = false;
    System.out.println("no");
    break;
    //quantifier, no exists
} else if (newArray.get(i).equals("."))
    && (!newArray.get(i + 2).equals("exists")
    && !newArray.get(i + 3).equals("exists"))) {
    forallOrExistChecker = true;
    System.out.println("single");
    break;
    //quantifier and exists
} else if (newArray.get(i).equals(".")
    && newArray.get(i + 2).equals("exists")) {
    System.out.println("double");
    forallAndExistChecker = true;
    //quantifier and not exists
} else if (newArray.get(i).equals(".")
    && newArray.get(i + 1).equals("~")
    && newArray.get(i + 3).equals("exists")) {
    System.out.println("double not");
    forallAndNotExistChecker = true;
}
    converted = converted + " " + newArray.get(i);
    i++;

}
if (forallOrExistChecker && forallAndExistChecker == false
    && forallAndNotExistChecker == false) {
    System.out.println("single quantifier");
    converted = converted + baru.output + " ) )";
} else if (!forallOrExistChecker) {
    System.out.println("no quantifier");
    converted = converted + baru.output + " )";
} else if (forallOrExistChecker && forallAndExistChecker) {
    System.out.println("double quantifier");
    converted = converted + baru.output + " ) )";
} else if (forallOrExistChecker && forallAndNotExistChecker) {
    System.out.println("double not quantifier");
    converted = converted + baru.output + " ) ) )";
}

System.out.println("full converted " + converted);
}

```

B.2 INFIX TO PREFIX LEXICAL CLASS

```

package newinfytopre;

import java.io.File;
import java.io.FileReader;
import java.util.ArrayList;
import java.util.Scanner;

public class lexical {

    public final int LETTER = 0;
    public final int DIGIT = 88;
    public final int UNKNOWN = 99;
    public final int SYMBOL = 77;
    public final int MINUS = 66;

    public final int INT_LIT = 10;
    public final int IDENT = 11;
    public final int NOT = 12;
    public final int IFANDONLYIF = 13;
    public final int ASSIGN_OP = 20;
    public final int ADD_OP = 21;
    public final int SUB_OP = 22;
    public final int MULT_OP = 23;
    public final int LEFT_PAREN = 25;
    public final int RIGHT_PAREN = 26;
    public final int SEM_COL = 27;
    public final int BIG_THAN = 28;
    public final int SMALL_THAN = 29;
    public final int TAB = 30;
    public final int SPACE = 31;

    public final int ASSERT = 32;
    public final int FORALL = 33;
    public final int DEFINE = 34;
    public final int SUCC = 35;
    public final int IMPLICATIONSIGN = 51;
    public final int NOTEQUAL = 52;
    public final int EQUAL = 53;
    public final int CONS = 54;
    public final int TITIKSAMADENGAN = 55;
    public final int TITIKDUA = 56;
    public final int BIGTHANEQUAL = 57;
    public final int LESSTHANEQUAL = 58;
    public final int QUESTIONMARK = 59;
    public final int POINT = 60;
    public final int AND = 61;
    public final int DOUBLEAND = 62;
    public final int EOF = 1000;

    String program;
    int charClass;
    int lexLen;
    char[] lexeme = new char[100];
    char nextChar;
    int token;
    public int nextToken;
    int lexNumber = 0;
    int satirNum = 1;
    int satirNumPrint = 0;
    int i = 0;
    int count = 0;

    String expOutput = "";
    String tempString = "";
    ArrayList<String> arrayListString = new ArrayList<String>();
    ArrayList<String> arrayListStatementString = new ArrayList<String>();

    /**
     * *****detect alphabet*****
     */
}

```

```

    public boolean isAlpha(char character) {
        if (((int) character >= (int) 'A' && (int) character <= (int) 'Z') ||
            ((int) character >= (int) 'a' && (int) character <= (int) 'z')) {
            return true;
        } else {
            return false;
        }
    }

    public boolean isMinus(char character) {
        if ((int) character == (int) '-') {
            return true;
        } else {
            return false;
        }
    }

    public boolean isDigit(char character) {
        if ((int) character >= (int) '0' && (int) character <= (int) '9') {
            return true;
        } else {
            return false;
        }
    }

    public boolean isSymbol(char character) {
        if ((int) character == (int) '@'// && (int) character <= (int) '/')
            // || ((int) character >= (int) ':' && (int) character <= (int)
'@')
            // || ((int) character >= (int) '[' && (int) character <= (int)
'`')
            // || ((int) character >= (int) '{' && (int) character <= (int)
'~')
            ) {
            return true;
        } else {
            return false;
        }
    }

    /**
     * *****detect space*****
     */
    public boolean isNull(char character) {
        if ((int) character == 0) {
            return true;
        } else {
            return false;
        }
    }

    public boolean isTab(char character) {
        if (character == '\t') {
            return true;
        } else {
            return false;
        }
    }

    public int lookup(char ch) {
        switch (ch) {
            case '.':
                addCharLexeme();
                nextToken = POINT;
                break;
            case ' ':
                addCharLexeme();
                nextToken = SPACE;
                break;
        }
    }

```

```

case '?':
    addCharLexeme();
    getCharInput();
    while (isAlpha(nextChar)) {
        addCharLexeme();
        getCharInput();
    }
    undoGetChar();
    nextToken = IDENT;
    break;
case '(':
    addCharLexeme();
    nextToken = LEFT_PAREN;
    break;
case ')':
    addCharLexeme();
    nextToken = RIGHT_PAREN;
    break;
case '+':
    addCharLexeme();
    nextToken = ADD_OP;
    break;
case '*':
    addCharLexeme();
    nextToken = MULT_OP;
    break;
case '~':
    addCharLexeme();
    nextToken = NOT;
    break;
case '=':
    addCharLexeme();
    getCharInput();
    if (nextChar == '=') {
        addCharLexeme();
        getCharInput();
        if (nextChar == '>') {
            addCharLexeme();
            getCharInput();
        }
    }

    nextToken = IMPLICATIONSIGN;
    break;
}
if (nextChar == '/') {
    addCharLexeme();
    getCharInput();
    if (nextChar == '=') {
        addCharLexeme();
        nextToken = NOTEQUAL;
        break;
    }
} else {

    nextToken = EQUAL;
}
undoGetChar();
break;
case ':':
    addCharLexeme();
    getCharInput();
    if (nextChar == '=') {
        addCharLexeme();
        getCharInput();
        nextToken = TITIKSAMADENGAN;
    } else if (nextChar == ':') {
        addCharLexeme();
        getCharInput();
        nextToken = CONS;
    } else {

```

```

        nextToken = TITIKDUA;
    }
    undoGetChar();
    break;
case ';':
    addCharLexeme();
    nextToken = SEM_COL;
    break;
case '>':
    addCharLexeme();
    getCharInput();
    if (nextChar == '=') {
        addCharLexeme();
        nextToken = BIGTHANEQUAL;
    } else {
        nextToken = BIG_THAN;
    }
    undoGetChar();
    break;
case '<':
    addCharLexeme();
    getCharInput();
    if (nextChar == '=') {
        addCharLexeme();
        getCharInput();
        nextToken = LESSTHANEQUAL;
        if (nextChar == '=') {
            addCharLexeme();
            getCharInput();
            if (nextChar == '>') {
                addCharLexeme();
                getCharInput();
                nextToken = IFANDONLYIF;
            }
        }
    }
    } else {
        nextToken = SMALL_THAN;
    }
    undoGetChar();
    break;
case '&':

    addCharLexeme();
    getCharInput();
    if (nextChar == '&') {
        addCharLexeme();
        getCharInput();
        nextToken = DOUBLEAND;
    } else {

        nextToken = AND;
    }
    undoGetChar();
    break;
case '\t':
    addCharLexeme();
    nextToken = TAB;
    break;
default:
    addCharLexeme();
    nextToken = EOF;
    break;
}
return nextToken;
}

public char peekChar() {
    nextChar = program.charAt(i);
    return nextChar;
}

```

```

public void getCharInput() {
    nextChar = program.charAt(i);
    if (nextChar != '#') {
        if (isAlpha(nextChar)) {
            charClass = LETTER;
        } else if (isDigit(nextChar)) {
            charClass = DIGIT;
        } else if (isSymbol(nextChar)) {
            charClass = SYMBOL;
        } else if (isMinus(nextChar)) {
            charClass = MINUS;
        } else {
            charClass = UNKNOWN;
        }
    } else {
        charClass = EOF;
    }
    i++;
}

public void undoGetChar() {
    i = i - 1;
}

public void addCharLexeme() {
    if (lexLen <= 98) {
        lexeme[lexLen++] = nextChar;
        lexeme[lexLen] = 0;
    } else {
        System.out.println("Error - lexeme is too long \n");
        count = count + 1;
        System.out.print("Line Number: " + satirNum + ", ");
        System.out.println("Lex Number: " + lexNumber);
    }
}

public String getStringRepresentation(ArrayList<Character> list) {
    StringBuilder builder = new StringBuilder(list.size());
    for (Character ch : list) {
        builder.append(ch);
    }
    return builder.toString();
}

public void getNonTab() {
    while (isTab(nextChar)) {
        getCharInput();
    }
}

public void getNonLine() {
    while (nextChar == '\n') {
        getCharInput();
        satirNum = satirNum + 1;
    }
}

public int lex() {
    lexNumber = lexNumber + 1;
    lexLen = 0;
    while (isTab(nextChar) == true || nextChar == '\n') {
        getNonLine();
        getNonTab();
    }
    switch (charClass) {
        /*
         * Parse identifiers
         */
        case LETTER:
            addCharLexeme();
            getCharInput();

```

```

        while (charClass == LETTER || charClass == DIGIT || charClass ==
SYMBOL || charClass == MINUS) {

            addCharLexeme();
            getCharInput();
        }
        nextToken = IDENT;
        break;
    /*
    * Parse integer literals
    */
    case DIGIT:
        addCharLexeme();
        getCharInput();
        while (charClass == DIGIT) {
            addCharLexeme();
            getCharInput();
        }
        nextToken = INT_LIT;
        break;
    case MINUS:
        addCharLexeme();
        getCharInput();
        nextToken = SUB_OP;
        break;
    /*
    * Parentheses and operators
    */
    case UNKNOWN:
        lookup(nextChar);
        getCharInput();
        break;
    /*
    * EOF
    */

    case EOF:
        nextToken = EOF;

        lexeme[0] = 'E';
        lexeme[1] = 'O';
        lexeme[2] = 'F';
        lexeme[3] = 0;
        break;
} /*
* End of switch
*/

System.out.print("Next token is: " + nextToken + ", Next lexeme is: ");
char temp[] = new char[lexeme.length];
for (int j = 0; j < lexeme.length; j++) {
    if (lexeme[j] != 0) {
        temp[j] = lexeme[j];
    } else {
        break;
    }
}

lexeme = temp;

ArrayList<Character> arrayListChar = new ArrayList<Character>();
for (int j = 0; j < lexeme.length; j++) {
    if (lexeme[j] != '0' && isNull(lexeme[j]) == false) {
        arrayListChar.add(temp[j]);
    } else {
        break;
    }
}

tempString = getStringRepresentation(arrayListChar);

```



```
System.out.print(lexeme);
System.out.println("tsfdfsfsf");
String concat = new String(tempString + " ");
System.out.println(concat);

    if (nextToken != 1000) {
        arrayListString.add(tempString);
    }
    return nextToken;
} /*
 * End of function lex
 */

public String programiOku(String programinIsmi) {
    String tumSatirlar = "";
    String satir = "";
    try {

        File programinDosyasi = new File(programinIsmi);
        FileReader dosyal = new FileReader(programinIsmi);
        Scanner dosya = new Scanner(dosyal);

        while (dosya.hasNextLine()) {

            satir = dosya.nextLine();
            tumSatirlar = tumSatirlar + satir + "\n";
        }
        dosya.close();
    } catch (Exception e) {
        System.out.println("Dosya Bulunamadi!" + e);
    }
    // Programin sonunu gosterme icin # karakteri kullaniyoruz.
    tumSatirlar = tumSatirlar + " " + "#";
    return tumSatirlar;
}

}
```

B.3 INFIX TO PREFIX STACK CLASS

```

package newinftopre;

import java.io.DataInputStream;
import java.io.IOException;
import java.util.ArrayList;

public class Stack {

    private String[] a;
    private int top, m;

    public Stack(int max) {
        m = max;
        a = new String[m];
        top = -1;
    }

    public void push(String key) {
        a[++top] = key;
    }

    public String pop() {
        return (a[top--]);
    }

    public String peek() {
        return (a[top]);
    }

    public boolean isEmpty() {
        return (top == -1);
    }
}

class Evaluation {

    private Stack s;
    private ArrayList<String> input;
    private ArrayList<String> output = new ArrayList<String>();
    private int i;

    public Evaluation(ArrayList<String> str) {
        input = str;
        s = new Stack(str.size());
    }

    public ArrayList<String> inToPre() {

        int count;
        i = input.size() - 1;
        while (i >= 0) {
            String ch = input.get(i);
            if (ch.equals("==>") || ch.equals("<==")) {
                gotOperator(ch, 0, "");
            } else if (ch.equals("&") || ch.equals("|")) {
                gotOperator(ch, 1, "");
            } else if (ch.equals(">") || ch.equals("<") || ch.equals("=") ||
ch.equals(">=") || ch.equals("<=") || ch.equals("=/=")) {
                gotOperator(ch, 2, "");
            } else if (ch.equals("+") || ch.equals("-")) {
                gotOperator(ch, 3, "");
            } else if (ch.equals("*") || ch.equals("/")) {
                gotOperator(ch, 4, "");
            } else if (ch.equals("::")) {
                gotOperator(ch, 5, "");
            } else if (ch.equals("")) {
                s.push(ch);
                output.add(0, ch);
            }
        }
    }
}

```

```

        } else if (ch.equals("(")) {
            gotParenthesis("(");
        } else {
            output.add(0, ch);
        }
        i--;
    }

    while (!s.isEmpty()) {
        output.add(0, s.pop());
    }
    return output;
}

private void gotParenthesis(String x) {
    int c = i;
    while (!s.isEmpty()) {
        String ch = s.pop();
        if (ch.equals(x)) {
            output.add(0, "(");
            break;
        } else {
            if (x.equals("(")) {
                output.add(0, ch);
            } else {
                output.add(ch);
            }
        }
    }
}

private void gotOperator(String opThis, int prec1, String x) {
    int c2 = i;
    while (!s.isEmpty()) {
        String opTop = s.pop();
        if (opTop.equals(x)) {
            s.push(opTop);
            break;
        } else {
            int prec2;
            if (opTop.equals("::")) {
                prec2 = 5;
            } else if (opTop.equals("*")) {
                prec2 = 4;
            } else if (opTop.equals("+") || opTop.equals("-")) {
                prec2 = 3;
            } else if (opTop.equals("<") || opTop.equals(">") ||
opTop.equals("<=") || opTop.equals(">=") || opTop.equals("=") ||
opTop.equals("/=")) {
                prec2 = 2;
            } else if (opTop.equals("&") || opTop.equals("|")) {
                prec2 = 1;
            } else {
                prec2 = 0;
            }
            if (prec2 < prec1 && x.equals("(")) {
                s.push(opTop);
                break;
            } else if (prec2 <= prec1 && x.equals("(")) {
                s.push(opTop);
                break;
            } else {
                if (x.equals("(")) {
                    output.add(0, opTop);
                } else {
                    output.add(opTop);
                }
            }
        }
    }
}

```

```
        s.push(opThis);  
    }  
}
```

B.4 INFIX TO PREFIX PARENTH CLASS

```

package newinfytopre;

import java.util.ArrayList;

class parenth {

    public String nextChar;
    public String output = "";
    private ArrayList<String> s;
    public int i;
    Operator test = new Operator();

    public parenth(ArrayList<String> str) {
        s = str;
    }

    public static boolean isAlpha(String character) {
        return ((int) character.toCharArray()[0] >= (int) 'A' && (int)
character.toCharArray()[0] <= (int) 'Z') || ((int) character.toCharArray()[0] >=
(int) 'a' && (int) character.toCharArray()[0] <= (int) 'z');
    }

    public void operator() {
        boolean unary = false;
        output = output + "(";
        if (nextChar.equals("+")) {
            output += test.Plus;
            nextChar = s.get(i++);
            variable();
            if (!unary) {
                if (i <= s.size()) {

                    variable();

                }
            }
            output += ")";
        } else if (nextChar.equals("-")) {
//            output += test.Substract;
        } else if (nextChar.equals("*")) {
            output += test.Times;
            nextChar = s.get(i++);
            variable();
            if (!unary) {
                if (i <= s.size()) {

                    variable();

                }
            }
            output += ")";
        } else if (nextChar.equals("/")) {
            output += test.Devide;
            nextChar = s.get(i++);
            variable();
            if (!unary) {
                if (i <= s.size()) {

                    variable();

                }
            }
            output += ")";
        } else if (nextChar.equals("::")) {
            output += test.Cons;
            nextChar = s.get(i++);
            variable();
            if (!unary) {
                if (i <= s.size()) {

                    variable();

                }
            }
        }
    }
}

```

```

    }
    output += " ";
} else if (nextChar.equals("==>")) {
    output += test.condition;
    nextChar = s.get(i++);
    variable();
    if (!unary) {
        if (i <= s.size()) {

            variable();

        }
    }
    output += " ";
} else if (nextChar.equals("<==>")) {
    output += test.doubleCondition;
    nextChar = s.get(i++);
    variable();
    if (!unary) {
        if (i <= s.size()) {

            variable();

        }
    }
    output += " ";
} else if (nextChar.equals("~")) {
    output += test.not;
    unary = true;
    nextChar = s.get(i++);
    variable();
    if (!unary) {
        if (i <= s.size()) {

            variable();

        }
    }
    output += " ";
} else if (nextChar.equals("/=")) {
    output += test.notEqual;
    nextChar = s.get(i++);
    variable();
    if (!unary) {
        if (i <= s.size()) {

            variable();

        }
    }
    output += " ";
    output += " ";
} else if (nextChar.equals("&")) {
    output += test.and;
    nextChar = s.get(i++);
    variable();
    if (!unary) {
        if (i <= s.size()) {

            variable();

        }
    }
    output += " ";
} else if (nextChar.equals("|")) {
    output += test.or;
    nextChar = s.get(i++);
    variable();
    if (!unary) {
        if (i <= s.size()) {

            variable();

        }
    }
    output += " ";
} else {
    output += nextChar;

```

```

        nextChar = s.get(i++);
        variable();
        if (!unary) {
            if (i <= s.size()) {

                variable();

            }
        }
        output += " ";
    }
}

public void variable() {
    output += " ";

    if      (nextChar.equals("=")      ||      nextChar.equals("~")      ||
nextChar.equals("/=")
            || nextChar.equals("+") || nextChar.equals("-")
            || nextChar.equals("*") || nextChar.equals("/")
            || nextChar.equals("<")  || nextChar.equals(">")  ||
nextChar.equals("<=") || nextChar.equals(">=")
            || nextChar.equals("::")
            || nextChar.equals("==>") || nextChar.equals("<==>")
            || nextChar.equals("&") || nextChar.equals("|")) {
        operator();
    } else if (nextChar.charAt(0) == '?' || isAlpha(nextChar)) {

        output += nextChar;
        //to avoid outofbound array we put if before incrementing
s.get(i++)
        if (i <= s.size() - 1) {
            nextChar = s.get(i++);
        }
    } else if (nextChar.equals("(")) {
        solParantez();
        if (i <= s.size() - 1) {
            nextChar = s.get(i++);
        }
    }
}

public void solParantez() {
    output += nextChar;
    nextChar = s.get(i++);
    while (!nextChar.equals("")) {
        if (nextChar.equals("(")) {
            solParantez();
        } else {
            if (nextChar.equals("+")) {
                output += test.Plus;
            } else if (nextChar.equals("-")) {
                output += test.Substract;
            } else if (nextChar.equals("*")) {
                output += test.Times;
            } else if (nextChar.equals("/")) {
                output += test.Devide;
            } else if (nextChar.equals("::")) {
                output += test.Cons;
            } else if (nextChar.equals("==>")) {
                output += test.condition;
            } else if (nextChar.equals("<==>")) {
                output += test.doubleCondition;
            } else if (nextChar.equals("/=")) {
                output += test.notEqual;
            } else if (nextChar.equals("~")) {
                output += test.not;
            } else if (nextChar.equals("&")) {
                output += test.and;
            } else if (nextChar.equals("|")) {
                output += test.or;
            } else {
                output += " " + nextChar;
            }
        }
    }
}

```

```

        }
    }
    nextChar = s.get(i);
    i++;
}
output += nextChar;
}
}

```

B.5 INFIX TO PREFIX OPERATOR CLASS

```

package newinfopre;

public class Operator {

    String Plus;
    String Subtract;
    String Times;
    String Devide;
    String Cons;
    String condition;
    String doubleCondition;
    String not;
    String notEqual;
    String and;
    String or;

    public Operator() {
        Plus = "Plus";
        Subtract = "Minus";
        Times = "Times";
        Devide = "/";
        Cons = "Cons";
        condition = "if";
        doubleCondition = "iff";
        not = "not";
        notEqual = "notEqual";
        and = "and";
        or = "or";
    }
}

```