Master Thesis

Design and implementation of a verifier for sequential programs using the Hoare calculus

submitted by: Florian Wege
Student number: 15856
Field of studies: Information and Communication Systems
Merseburg University of Applied Sciences

supervised by: Prof. Dr. phil. Dr. rer. nat. habil. Michael
Merseburg University of Applied Sciences
Prof. Dr. rer. nat. habil. Eckhard Liebscher
Merseburg University of Applied Sciences

Merseburg, October 22, 2017

Table of Contents

List of Tables

List of Figures

Abstract

The *Floyd-Hoare* logic or calculus is a methodology for proving the partial or total correctness of computer programs developed by *C.A.R. Hoare* based on ideas of Robert W. Floyd and has awoken a wave of enthusiasm in the domain of program verification. Though the system has received a great deal of recognition, some fundamental problems in effectively using it remain to be solved, as they were found undecidable.

This paper aims to build a bridge between the often only theoretically-contemplated *Hoare* calculus and the venture to implement such with an example, starting from scratch. It describes the construction of an LL(1) parser, the preparation of corresponding grammars, a transformation of the obtained syntax trees, the Hoare methodology and delves into the issues of searching for loop invariants and the handling of logical and arithmetic expressions, seeking heuristics of the implication problem and resorts to user interaction where automated solutions fall short.

Chapter 1
1 Introduction

1.1 Motivation

Since the introduction of computers, more and more of the human life's activities experienced a shift to digital handling. That transition streamlined a lot of things but also called for a new profession facing up to the proper control of those machines, which would soon become known under the term *software engineering*. As there is a myriad of processes to be described and accounted for, the phenomenon had been bound to fan out. Thus the emergence of specialized engineers for diverse systems and their hybrid forms gradually took place. *Programming* in its common associations has turned into a basic skill and some politicians actually want to integrate it firmly into the curriculums of elementary schools.[1] This discipline has also adapted a certain trait of creativity. With the right idea in mind for an App, arranging the available components in a way of high usability, one can tap a market demand because a range of platforms and services are already in place for unleashing one's creative mind upon. Hardware, too, became more feasible and sophisticated in time but, as the term discloses, software stands for an elevated level of malleability, which initially fosters a trial-and-error-flavored development. Adding more and more layers of abstraction, the exact behavior of a program is often up for speculation. Unless one is working with embedded systems, which require a close-up treatment, being aware of the inner workings, to some extent at least, has been rendered increasingly unnecessary.

> "At least once a semester I hear some kid yell, 'Wow! This is like magic!' and that really motivates them."
>
> —Alfred Thompson, computer science teacher

This aspect is in contradistinction to the foundations of computer science, which seeks to formalize and systematically find solutions to problems. And in fact, as the application of software engineering progressed, the domain of topics enlarged. There is the so-called *Law of leaky Abstractions* as depicted by *Joel Spolsky* in 2002.[2] It states

[1] http://www.npr.org/sections/ed/2016/01/12/462698966/the-president-wants-every-student-to-learn-computer-science-how-would-that-work

[2] https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/

that there is no perfect abstraction of non-trivial procedures. Especially when things do not work as they should, the mask falls and implementation details protrude. Questions of optimization and concerns about security are catching up and thus a particularized insight into the groundwork is about to regain value. As noted above, most everyday business is already handled by software and there are a couple of different applications where utmost accuracy is key:

- monetary transactions: automatic teller machines, to debit the right account, transfering ability to where it is needed
- infrastructure: e.g. ensuring the proper behavior of vehicles, switching and communication systems
- dealing with customer (private) data
- ...

Since it is more economic to do so, most software companies are content with the density of errors below a prior specified threshold. That is, the amount of errors per lines of code is measured and weighed against a target value set within the analysis phase. For applications where an error could cost the life of a human being, the common pick is a limit of 0.5 per 1000 lines (0.05%).[3] Yet one may argue that risks of that kind should be diminished to zero and that an individual life is not up for quantification.

From a historical viewpoint, a broad range of different causes for malfunctioning software could be recorded: A comma in lieu of a dot, a wrong signum leading a numeric expression, the usage of a wrong formula, racing conditions, an insufficient domain of definition, protocol errors, imprecision of floating point operations, overload, buffer overflow/underflow, non-considered constellations and many more. Upcoming are trends like *IoT* (Internet of Things) or autonomous driving that pose new layers of networks and challenge established safety aspects. Another famous concern is cost efficiency: It is known that the later a software mistake is discovered, the more expensive it is to be fixed. After the development phase, the team that initially wrote the program often moved on to newer shores. Or in a case like the Mars rover *Curiosity*, once deployed, replacing the software a posteriori appears rather challenging. Moreover, often enough, the source code is not published, demanded requirements can change later on and glitches may manifest themselves without displaying symptoms right away but which may infect the system and deal a blow to its expandability later on.

Summarizing this section, the sources of programming errors are multifarious. There is a lot of potential hidden in between and it is often vital to know that software is indeed working correctly before shipping or utilizing it. Hence is implied a stricter methodology of scrutinizing software if not one for its systematic development.

[3] https://de.wikipedia.org/wiki/Fehlerquotient#Fehlerdichte_in_der_Informatik

1.2 Analysis vs simulation

To decrease the number of errors, a range of actions is at the software engineer's disposal. Most often that involves peer reviews, i.e. an independent surveyor evaluates one's code. Another idea is to simulate the behavior by carrying out dynamic test cases, directly running the code. Therefore, positive test cases are written that present well-formed inputs or conditions for an algorithm, then that algorithm is executed and the results are checked for integrity. Conversely, negative tests are to confirm bad input or precondition scenarios will yield a proper error handling. The program is not supposed to end in an unregarded segmentation fault (stemming from an invalid access of memory), maybe should instead display a message box and append the information of the exception to a log file.

However, those tests cover but a part of the possible instances the code allows for, therefore fail to vindicate an overall correctness as a famous quote by *E. W. Dijkstra* alleges:

> "Program testing can best show the presence of errors but never their absence."
>
> —Edsger W. Dijkstra

On the other hand, when speaking about verification in the environment of theoretical computer science, the term denotes a genuine proof of the absence of errors within a program according to a specification using formal means. It therefore poses an exact method to evaluate the quality of a software, which, as hinted above, turns out to be crucial at some points and as a consequence rightfully enlarges the discipline of software engineering.

1.3 Basic approaches

Currently, there are two main approaches known to this stipulation: model checking and deductive means. Model checking is the method of deciding whether a program or a model of it suffices a given specification by exploring its state-space. At this, a model serves an abstraction of the reality. The idea is to examine the model in order to draw conclusions about the actual system. It needs to be fitting to the task at hand, should be reduced as far as possible to simplify the issue but still contain all the relevant information. Different models can be mixed to acquire new information but such course of action is quick to decline the collective operability.

The state-space is a set or graph of constellations of the values of variables and the current point of execution. This vector describes the state of the program in its

entirety. By going through the program code, a verifying tool shifts between the states in order to find all possible execution paths. Those are then checked against constraints, e.g. invariants like a combination of variable values that should never occur. If an execution path should be found that violates these conditions, it will be exposed and the programmer can observe the execution path that lead to the error to hopefully fix its root cause. This approach basically traverses all accessible variations before it marks the program as approved. That is why there is an exponential growth in computation time and required memory involved, rendering the algorithm impractical fairly quickly. Model checking also necessitates a closed, finite system (or an algorithm to render it as such). Otherwise the program could keep allocating memory and the number of states would not exhaust, thus the verifier may never come to a conclusion. Dynamic data structures may be examined by dedicated methods like shape analysis[4].

Both the model and the specifications are described in appropriate languages that allow the verifier to work with. Due to the intent of exploring the state space, the modeling language must project a finite state machine. Examples are PROMELA (Process Meta Language), Timed Automata or Petri nets. The specifications or properties checked for are usually decorated by logical expressions like temporal logic as introduced in programs by A. Pnueli [Pnu77], are tacit (a division by zero should never occur) or may be integrated in the modeling language itself, e.g. PROMELA permits to insert assertions as part of the control flow.[5]

Another way of verifying a program and the approach presented to be in this paper is the deductive resolution of theorems. This idea was first introduced by *Alan Turing* on a conference in 1949. Due to typographical mistakes and other circumstances, it went hidden for a bit but later on other researches have retaken the topic. The important aspect back then was the notion that the problem could be modularized and that a program (*Turing* used flowcharts) could be decorated with assertions. Later, in 1969, *C.A.R. Hoare* invented a set of axioms and rules, the so-called *Hoare* triples, that would point out a relation between an elementary program instruction or control flow and its effect on what can be logically assumed from what the semantics of this piece of code are to imply. For example the following listing increments a numerical variable 'x' by 1.

```
1   x=x+1
```

Listing L1: First listing

Now it can be said that prior to this assignment, the variable had been lower by 1 compared to its new state. Or more generally, before the assignment, every occurrence of 'x' had been substituted by the expression of the new value.

[4] https://en.wikipedia.org/wiki/Shape_analysis_(program_analysis)
[5] https://en.wikipedia.org/wiki/Model_checking

```
1  PRE{true}
2    z=x+y;
3    z=z*2;
4    z=z-(x+y)
5  POST{z==x+y}
```

Listing L2: Disadvantages of state-space exploration

The above code snippet shall serve as an easily comprehensible example for when deductive means outclass the procedure of state exploration. Assuming that x and y are only 2 byte integer variables and may take any value in their data type domain, that totals the combination of 32 bits or over 4 billion possibilities to check the post condition for, which is what state exploration would do. On the other hand, a human being should be able to recognize the pattern easily. By substitution, one could argue the assignments should be reduced to a single one (z=x+y) and that straight seems to match the postcondition. So does the above program fulfill the surrounding specification? Maybe, maybe not. It should be considered that, first off, the conditions between the curly braces possess their own language with their own semantics, e.g. the operators may have their own meaning. Secondly, those are not exactly mathematical expressions. As hinted by x and y being 2 byte integer variables, z too might be restricted, thus the add and multiplication instructions may cause a buffer overflow, whereupon the semantics would have been altered by the substitution then. So of course it depends on the underlying system and that system respectively the semantics of the language have to be well-known in advance. Other than that, an axiomatic theorem solver like the human would identify the pattern, apply rules on the structure of the program to see what can be derived from it and then make statements about it.

However, deductive program verification comes with its own set of problems: Those usually revolve around the elementary control flow constructs of imperative programming languages and the implications of those are not necessarily assessable in their entirety. The question whether a predicate logic expression connotes another is known to be undecidable in general. Furthermore, the later precisely examined calculus by itself only establishes relationships and does not exactly hand out an algorithm. From theoretical computer science it can be stated that, in general, it is impossible to say if a program satisfies a set of specifications of any kind. However, it becomes more feasible when narrowed down to classes of programs and regarded languages.

1.4 What the paper is about

The objective of this paper shall be to outline the Hoare calculus, to make a design for a verifier that would present how rationales can be applied starting from a raw

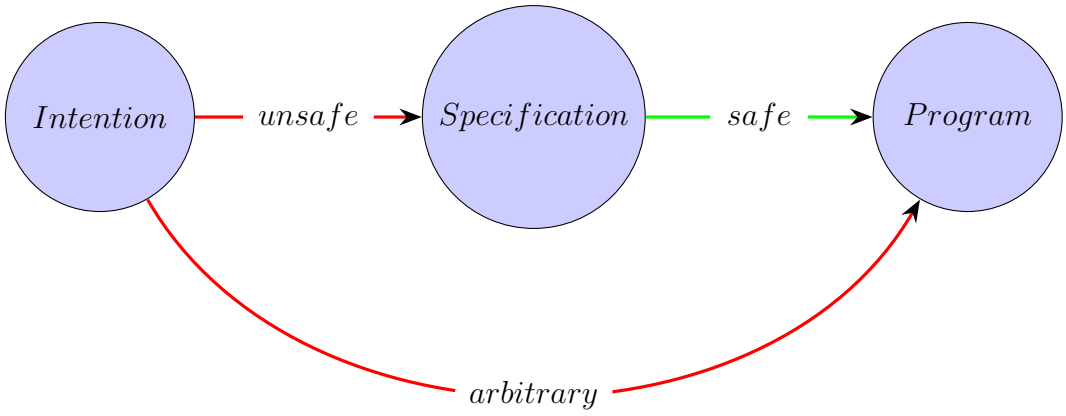

Figure 1.1: From intention to program

string input, how the mathematical formulae that come with it could be transcribed to imperative algorithms and at which points the interaction with a human user is still required. The elaborated design is then to be implemented in the *Java* programming language along with an appropriate graphical user interface to portray the inductive procedure of Hoare-style reasoning.

To conclude the introduction, it should be stated that both model checking and theorem solving rely on a proper specification of the issue. If that is already faulty, which may be the case, since the specification needs a strict formalization as well, any verification will be meaningless and is prone to invoke type II errors (false negatives) because it fails to project the client's true intentions.

figure 1.1 reveals more origins of error. Even when establishing specifications and a model and even obtaining the program by transformation of the model in the target language, there are still risks of human failings in between that may falsify the verification result (and the tooling must be assumed to be working flawlessly). That poses another reason why even a formal verification should only be seen as an additional scheme in the quality assurance environment.

Chapter 2
2 Preliminaries

2.1 Overview

Before plunging into the core topic of this paper, it appears necessary to formalize the target of a prospective verifier. In order to make statements about the validity of a program, whether it holds to certain properties or not, both the program and the properties should be fixated. What could be considered secure knowledge anyway? Such a holistic view only makes sense under the presumption that some basic ideas can already be regarded as irrevocably intrinsic and then means of induction, analogy or similar are used to widen the scope and to declare more statements compatible to the existing knowledge base, verifying or, if they appear as contradictory, objecting them.

And the strategy here is likewise: To know if a program fulfills some conditions, the meaning of the program and the conditions have to be exact. Since this entails lots and lots of programs and attributes in general, it becomes evident that rather than manually and pointlessly contemplating all possible variations, it deems better to ascribe it to some underlying scheme that can be unfolded on demand. Therefore the meaning, also called the semantics, of a program (and later also those of conditions) should be inductively synthesized using an appropriate model kit relating to a set of basic entities. Prior to determining the semantics, these entities also have to be identified as such, which is the part of the syntax analysis and shall be depicted as well.

Therefore the schedule is as follows: The rest of this chapter will give some further classification, tease about the purport and hand over a short preview. It may be skimmed or skipped over if the reader is fond of the contextual knowledge. Chapter chapter **??**: **??** will formally start with the definition of a language, how are they characterized and how they can be processed. Subsequently, the specific language subject to this paper and whose programs are to be verified is firmly presented along with some simple remarks about its significance. Afterwards, in the dedicated chapter **??**: **??** chapter, these semantics are going to be formalized and, moreover, the type of semantics and how to continue with it are explained. The chapter **??**: **??** chapter reasons about using the obtained semantics, talks about the notion of correctness and introduces assertions by extending the language. Finally, a transition to the *Hoare* rule system will be conducted, how to use it for verification of sequential programs and what challenges come with it. Examples will be reviewed and ideas to overcome the

challenges be discussed along with some hints to the implementability of such endeavor. At this point, the theory and the general design will have been covered. The realization of the verifier using the aforementioned theory is carried out via the *Java* programming language and eyed in the hindmost chapter 5: *Implementation* . Before rounding out with a final recapitulation, chapter **??**: **??** touches upon some alternate verification concepts.

Introduced shall be a simplified imperative language that later becomes target of the *Hoare* rule system which is derived from operational semantics. The language contains basic control flow elements like the composition of instructions, the selection routine and condition-controlled loops. More complex programs can be written by combining the aforementioned basic structures.

2.2 How to instruct computers

Computers by definition are devices that understand some sort of digitally represented information and can process it in compliance with a program. That program may be immutably ingrained or be loaded from a mounted memory storage as more dynamic machines permit, which is what bestows a great range of use cases upon them and made them ubiquitous. Yet those machines are set up at some point and consist of a number of rigid hardware components providing their specific functionality and each of that hardware speaks a certain language that needs to be addressed for. To be able to have this orchestra flawlessly work in unison, besides complying with a couple of basic interfaces, there is usually a mediator called the operating system involved and the concept of drivers further helps to identify the spoken language of each component. Hardware and operating system together are then referred to as the platform, serving as a layer to host user-written programs on and more layers can be stacked on top if needed. But the platform is essentially the lowest layer to have the angularity of the hardware components relaxed. At this point, everything channels through its digital interface and is therefore unified in the language referred to as *machine code*.

Still, machine code, as the name indicates, varies between different machines. Infact, in the beginning, software engineers tended to write in *assembly* language, which is a more human-readable representation of machine code yet still platform-dependent. To not have to rewrite the same program logic for different platforms over and over again, more levels of abstractions were piled up and thus high level languages were born. High level languages like *C* aggregate universal coding paradigms like variables or control structures and can combine elementary instructions to larger compounds. This makes it easier for the software engineer to assess the functionality of a program, which in turn in some way is a first step to boost the drafting of correct programs.

For the platform to be able to use such high level programs, of course it appears

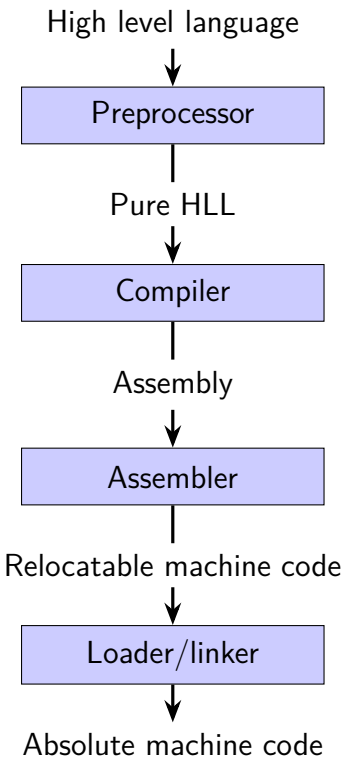

Figure 2.1: High level code to machine code

necessary to reduce them to executable machine code again. The standard procedure is depicted in figure 2.1 . The user-manufactured code may be exposed to a preprocessor, which takes care of some preparatory tasks like the inclusion of other script files or alternative substitutions like those carried out by the *#define* directive in the *C* language. This yields the pure high level language as understandable by a compiler. The compiler does the main part of the translation and outputs assembly code which only needs to be transcribed to machine code and linked together to obtain the final executable version the hardware setup can be operated with.

The language as described in this paper won't be in need of such a preprocessor and the verification is directly applied to the high level language during the compiler phase. Thus any kind of assembly output or lower level persistence is not required here and, in fact, the workings of a compiler can be split up further in detail.

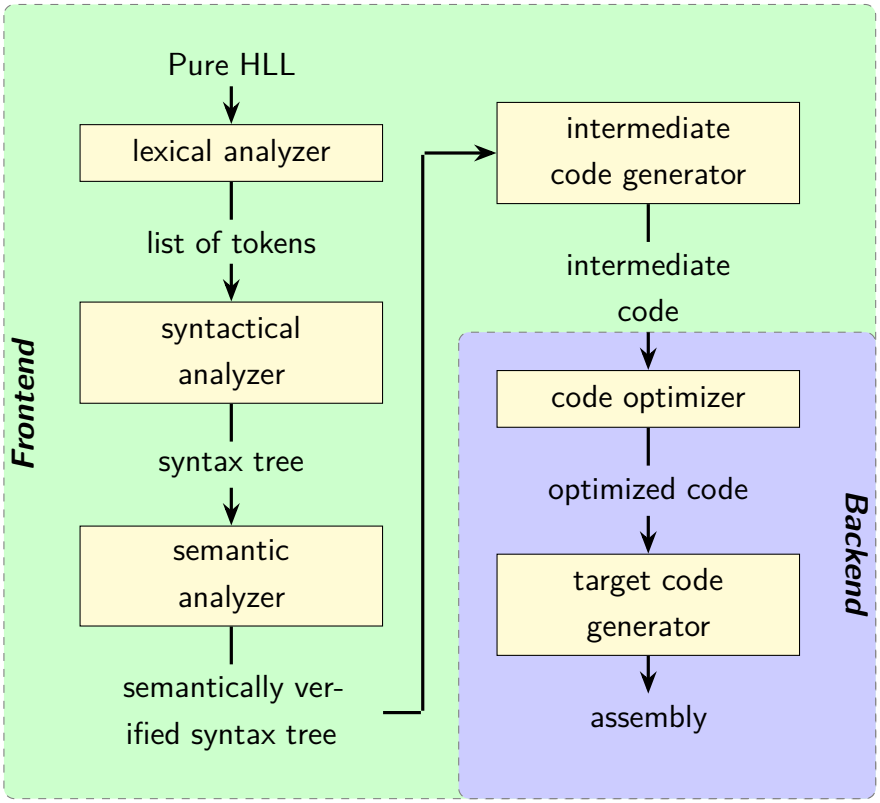

Figure 2.2: Processing chain of a compiler

Figure 2.2 shows the composition of the different components of a compiler. In a first step, a lexical analyzer, usually denoted as *lexer* or *tokenizer*, surveys the raw input sequence of characters and contracts the to be as cohesive identified words (*lexemes*) to a sequence of tokens in the same order. This makes it easier for the syntactical analyzer (*parser*) to progress because it abstracts the tokens as the atomic symbols instead of the initial single characters then. Both lexer and parser work with some grammar based on which the symbols are bestowed semantics but while the task of a lexer is to handle a less complex regular grammar (*Chomsky*'s level 3), which can be realized using regular expressions, parsers must cope with context-free grammars (*Chomsky*'s level 2) and normally it would be desirable for the parser to produce a tree structure incorporating rule precedence. The lexer can also be used to remove unnecessary white space and comments. More about the process can be read in the next chapter.[1]

After the syntactical analysis, there can be a semantic analysis that cross-checks the validity of the constructed syntax tree or sanitizes it, e.g. the typing in a variable assignment statement like:

$$x = y + 1 \tag{2.1}$$

[1] https://stackoverflow.com/questions/2842809/lexers-vs-parsers

Do variable (x) and the expression assigned to it (y+1) actually match in their respective data type? Since the type of the variable may be fixated (declared) far off the assignment instruction, it probably won't reside in the same branch/grammar production rule path. Moreover, the namespace for variables may be shared with other entities like functions, so it might make sense to examine whether the identifier does in fact denote a variable. Thus the semantic analyzer exposes a verified syntax tree.

The Intermediate Code Generator translates the syntax tree into another intermediate representation, which serves as an interface between the high level language and the platform. The last two steps (*Code Optimizer* and *Target Code Generator*) can then be replaced per setup, so the other components are left untouched.

The program verifier as described in this paper is inserted right after the syntactical analyzer. It would ideally be done after a semantic analysis but the contemplated language and the samples are simple enough that such won't be required. Nowadays, the described analyses are often processed incrementally in background threads parallel to the programmer writing code and, in this way, assisted deductive program verification can be blended in on-the-fly but suchlike tool and verification in general are rather subject to specialized languages at this point in time. Examples of those languages are *Spec#*, an extension to *C#* or *JML (Java Modeling Language)*, which seeks to introduce verification specifications in the *Java* programming language by wilily wrapping the additional semantics required in comment syntax.

Chapter 3
3 Introduction of language

3.1 On languages and grammars

3.1.1 Definitions

The considered programs are made up from a sequence of symbols of a given *alphabet*. A sequence of symbols is said to be a *word*.

Definition 1. *An alphabet is a set of symbols. Ex: {a, b, c}*

Definition 2. *A word is a string or finite sequence of symbols (associated to a specific alphabet). Ex: {aaba, bbb, acb} using alphabet {a, b, c}*

To later derive a meaning from it, some code must be established in advance and not just any word shall be accepted by the compiler but a well-defined set of words as denoted by a formal *language*.

Definition 3. *A language is a set of words. Since those sets are usually infinite, a language is commonly described by a predicate.*

Ex: A language may be designated by the notation of a regular expression like for instance *a*b*, the asterisk being a quantifier for the preceding symbol, indicating "an arbitrary number of", so the given example would encompass the words *b*, *ab*, *aab*, *aaab* and so on but regular expressions can only describe regular languages (Chomsky's level 3) while there are other ways to address greater sets of languages. Since languages as defined above alone still lack a proper structure to bind semantics to, the concept of grammars is to be introduced. Those consist of a set of production rules to span a language incrementally, chunk and organize their words into a tree structure, called the tree of the syntax. Grammars always relate to a (specific type of) language and are formalized as following:

Definition 4. *A grammar is a tuple of a set of variables (also called non-terminals), terminal symbols, production rules and a starting symbol (or set thereof).*

$$G = (V, T, P, S)$$

- *V - set of non-terminal symbols or variables*
- *T - set of terminal symbols*
- *P - production rules*
- *S - starting symbol(s)*

The first example of a grammar in figure 3.1 displays the standard *Backus-Naur Form*[1] notation that will also be used throughout this paper. In a mathematical set notation it corresponds to:

$$
\begin{aligned}
G = (\\
&V = \{\langle \text{exp} \rangle, \langle \text{boolExp} \rangle\},\\
&T = \{\text{`id'}, \text{`+'}, \text{`*'}, \text{`\&'}, \text{`|'}, \text{`<'}, \text{`>'}, \text{`<='}, \text{`>='}, \text{`='}, \text{`<>'}, \text{`true'}, \text{`false'}\},\\
&P = \{\langle \textit{exp} \rangle ::= \langle \text{exp} \rangle \text{ `+' } \langle \text{exp} \rangle,\\
&\langle \textit{exp} \rangle ::= \langle \text{exp} \rangle \text{ `*' } \langle \text{exp} \rangle,\\
&\langle \textit{exp} \rangle ::= \text{`id'},\\
&\langle \textit{boolExp} \rangle ::= \langle \text{boolExp} \rangle \text{ `\&' } \langle \text{boolExp} \rangle,\\
&\langle \textit{boolExp} \rangle ::= \langle \text{boolExp} \rangle \text{ `|' } \langle \text{boolExp} \rangle,\\
&\langle \textit{boolExp} \rangle ::= \langle \text{exp} \rangle \text{ `<' } \langle \text{exp} \rangle,\\
&\langle \textit{boolExp} \rangle ::= \langle \text{exp} \rangle \text{ `>' } \langle \text{exp} \rangle,\\
&\langle \textit{boolExp} \rangle ::= \langle \text{exp} \rangle \text{ `<=' } \langle \text{exp} \rangle,\\
&\langle \textit{boolExp} \rangle ::= \langle \text{exp} \rangle \text{ `>=' } \langle \text{exp} \rangle,\\
&\langle \textit{boolExp} \rangle ::= \langle \text{exp} \rangle \text{ `=' } \langle \text{exp} \rangle,\\
&\langle \textit{boolExp} \rangle ::= \langle \text{exp} \rangle \text{ `<>' } \langle \text{exp} \rangle,\\
&\langle \textit{boolExp} \rangle ::= \text{`true'},\\
&\langle \textit{boolExp} \rangle ::= \text{`false'}\},\\
&S = \{\langle \text{exp} \rangle\}\\
)
\end{aligned}
$$

This assumes that $\langle \text{exp} \rangle$ is indeed the starting symbol, which is not quite clarified in the first notation and will instead be separately stated when needed. Variables are those that appear on the left-hand side of the production rules and which form the non-terminal nodes of the syntax tree. Terminals are atomic symbols, which means they cannot be split up any further and become leaves of the syntax tree. They can only be found on the right-hand side of the rules. Furthermore, production rules tell how the input words shall be broken down into variables and terminals. The starting symbol (or set thereof) determines what rule(s) to regard on the highest level. The pipe or vertical line symbol | indicates an alternative. Thus the variable $\langle \text{exp} \rangle$ can either be derived to *exp* + *exp*, to *exp* * *exp* or to *id* here.

[1] http://matt.might.net/articles/grammars-bnf-ebnf/

```
⟨exp⟩      ::= ⟨exp⟩ '+' ⟨exp⟩
             | ⟨exp⟩ '*' ⟨exp⟩
             | id

⟨boolExp⟩  ::= ⟨boolExp⟩ '&&' ⟨boolExp⟩
             | ⟨boolExp⟩ '||' ⟨boolExp⟩
             | ⟨exp⟩ '<' ⟨exp⟩
             | ⟨exp⟩ '>' ⟨exp⟩
             | ⟨exp⟩ '<=' ⟨exp⟩
             | ⟨exp⟩ '>=' ⟨exp⟩
             | ⟨exp⟩ '=' ⟨exp⟩
             | ⟨exp⟩ '<>' ⟨exp⟩
             | 'true'
             | 'false'
```

Figure 3.1: first grammar

Now there are different classes of grammars. Depending on it, the required strategy of a parser will look different. The later introduced language shall suffice the context-free LL(1) type. This is why the following steps explain the constraints and construction of a LL(1) grammar. To obtain such one, ambiguity, left recursion and non-determinism shall be erased. Some examples are to be inspected and transformed accordingly before stepping further and applying the learned techniques on the actual used language.

3.1.2 Ambuigity, Associativity, Precedence

Most of the conductions and remarks in this section refer to the compiler design lectures by *Ravindrababu Ravula*[Rav17].

In a first step, the grammar of figure 3.2 shall depict the concatenation of id terminals by the + operator, namely the infix notation of addition. When exposing an input *id+id+id* to that grammar, the two different syntax trees shown in figure 3.3 may be obtained. That means the derivation process is not definite. Even with the arithmetic addition of two numbers being associative and commutative (Abelian), ambiguity in grammars is not really desired. There shall exist no more than one possible syntax tree for the same input and grammar. Telling if a grammar is ambiguous in

```
⟨exp⟩      ::= ⟨exp⟩ '+' ⟨exp⟩
             | 'id'
```

Figure 3.2: Two-sided recursive grammar

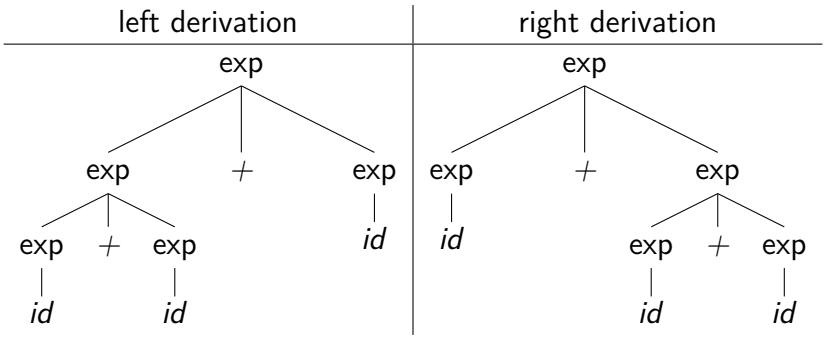

Figure 3.3: syntax trees of the example grammar for *id*+*id*+*id*

general is not decidable.[2] However, the ambiguity at hand is evidently induced because it is unclear whether the addition operator binds to the lefthand or righthand side. This can be taken care of by rewriting the production rules to not include the same non-terminal on both ends. The grammar of figure 3.4 has the non-terminal situated at the ending, its syntax trees grow right-sided (rightmost derivation). The grammar of figure 3.5 has the non-terminal situated at the beginning, its syntax trees grow left-sided (leftmost derivation). With the addition being commutative, it does not matter semantics-wise but in order to be capable of elucidating another point about leftmost derivation, it shall be persevered with.

$$\begin{aligned} \langle exp \rangle \quad & ::= \text{`id'} \ \text{`+'} \ \langle exp \rangle \\ & \mid \ \text{`id'} \end{aligned}$$

Figure 3.4: right-most derivation

[2] https://en.wikipedia.org/wiki/Ambiguous_grammar#Recognizing_ambiguous_grammars

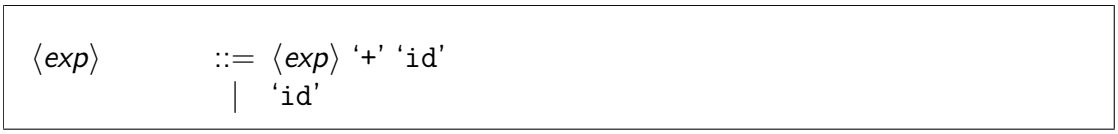

Figure 3.5: left-most derivation

In the following step, the grammar is extended by multiplication according to figure 3.6 . Again an example input *id+id*id* yields two different syntax trees as visible in figure 3.7 . Moreover, they entail different semantics. When evaluating the semantics of an expression like applying the mathematical operations of addition and multiplication, it is not reasonable to re-synthesize the string containing the mathematical expression as that would again call for the necessity of analyzing the structure but rather the syntax tree should directly be worked on progressively. The children of a node would recursively be conflated before advancing to their parent. So in figure 3.7 , the left tree would prioritize and execute the addition first and the resulting sum would become a factor in the multiplication, which would amount to a different value than what the mathematical expression *id+id*id* denotes. The multiplication must be carried out before the summation directive. This raises the question on how to enforce precedence of operators within a grammar. Operations of higher precedence have to be on a deeper tree level. The solution is to split the grammar into more stages of variables.

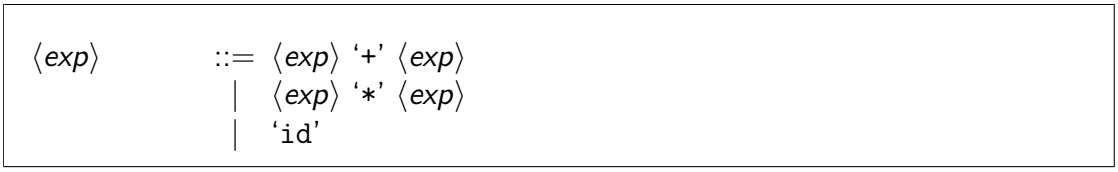

Figure 3.6: exp grammar with multiplication

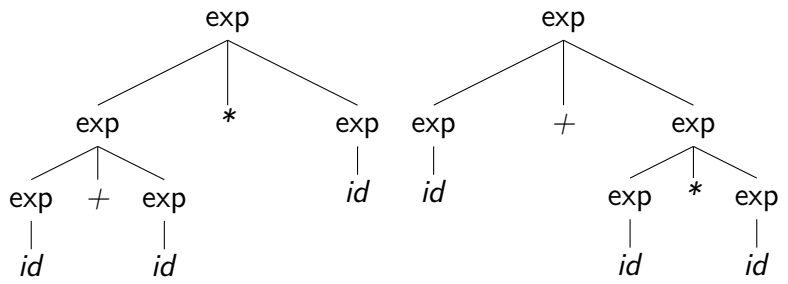

Figure 3.7: syntax trees of the multiplication-extended exp grammar

Now, in the grammar of figure 3.8 , the '+' and the '∗' operators are disconnected, they reside on different levels. Only the ⟨prod⟩ variable is able to contain multiplication and it cannot go back to ⟨exp⟩. The purpose of ⟨exp⟩ is to realize summation but it can derive to occurrences of ⟨prod⟩. So it realizes a one-way street and the sum derivation happens at the upper levels. The intrinsic ambiguity was eliminated as well. The raison d'être of the second production rule of ⟨exp⟩ is for the case that there is no summation involved, it should go straight to ⟨prod⟩ then. The second rule of ⟨prod⟩ acts as a terminator, otherwise the tree would keep on growing and it accounts for the case that maybe there is no multiplication within the expression.

In summarization, to fix associativity, the recursivity of the rules has to be adjusted (⟨*exp*⟩ ::= ⟨exp⟩ '+' `id` or ⟨*exp*⟩ ::= `id` '+' ⟨exp⟩ but no ⟨*exp*⟩ ::= ⟨exp⟩ '+' ⟨exp⟩). To fix precedence, a hierarchy of non-terminals has to be established. This information will be adduced when designing the language whose words shall be verified and that is up for implementation.

⟨*exp*⟩ ::= ⟨*exp*⟩ '+' ⟨*prod*⟩
| ⟨*prod*⟩

⟨*prod*⟩ ::= ⟨*prod*⟩ '∗' `id`
| `id`

Figure 3.8: grammar with operator precedence

3.1.3 The problem with left recursion

When writing a grammar, it should be ensured that a real parser will have to be able to work with it. In a grammar as depicted in figure 3.9 with the starting symbol being ⟨exp⟩, a top-down parser has to make a decision whether to pick the rule ⟨*exp*⟩ ::= ⟨exp⟩ '+' 'id' or ⟨*exp*⟩ ::= 'id'. The basis for a specific decision is the input string. When entering the ⟨exp⟩ variable, the type of parser that will be illustrated here would investigate the first part of the first production rule ⟨*exp*⟩ ::= ⟨exp⟩ '+' 'id', which again is ⟨exp⟩. Without having made any progress, it finds itself in the same situation, the parser will try to derive ⟨exp⟩ and see ⟨*exp*⟩ ::= ⟨exp⟩ '+' 'id' as the path to pursue. It turns out to be an never-ending loop. This is why left recursion should be avoided in all production rules. To retain the generated language yet still get rid of left recursion, there is a simple conversion prescript demonstrated in figure 3.10 . Another non-terminal is inserted that may right-recursively spawn new instances of α (everything that follows the initial non-terminal of the production rule causing the left-recursion) or end with ϵ which is the empty word. ϵ does not take any token from the input. Now, α and β may be substituted by any sequence of variables/terminals. To reform the above example grammar in figure 3.8 and match the conversion pattern, α corresponds to '+' ⟨mult⟩ and β corresponds to ⟨mult⟩. The transformation is illustrated in figure 3.11 . On top of that, the pattern can be extended for multiple α and β branches like in figure 3.12 .

⟨*exp*⟩ ::= ⟨*exp*⟩ '+' 'id'
| 'id'

Figure 3.9: grammar with left recursion

left recursion	right recursion
⟨*A*⟩ ::= ⟨*A*⟩ α | β	⟨*A*⟩ ::= β ⟨*A'*⟩ ⟨*A'*⟩ ::= α ⟨*A'*⟩ | ϵ

Figure 3.10: elimination of left recursion

left recursion	right recursion
$\langle exp \rangle \quad ::= \langle exp \rangle \text{ test}$ $\quad \mid \beta$	$\langle exp \rangle \quad ::= \text{<mult>} \langle exp' \rangle$ $\langle exp' \rangle \quad ::= + \text{<mult>} \langle exp' \rangle$ $\quad \mid \epsilon$

Figure 3.11: exp with right recursion

left recursion	right recursion
$\langle A \rangle \quad ::= \langle A \rangle \alpha_1$ $\quad \mid \langle A \rangle \alpha_2$ $\quad \mid \langle A \rangle \alpha_3 \ldots$ $\quad \mid \beta_1$ $\quad \mid \beta_2$ $\quad \mid \beta_3 \ldots$	$\langle A \rangle \quad ::= \beta_1 \langle A' \rangle$ $\quad \mid \beta_2 \langle A' \rangle$ $\quad \mid \beta_3 \langle A' \rangle \ldots \langle A' \rangle$ $\quad ::= \alpha_1 \langle A' \rangle$ $\quad \mid \alpha_2 \langle A' \rangle$ $\quad \mid \alpha_3 \langle A' \rangle \ldots$ $\quad \mid \epsilon$

Figure 3.12: elimination of left recursion with multiple instances of α and β

Lastly, in order to produce a LL(1) grammar, non-determinism must be preempted. That means that the parser must be able to deduce which rule to use looking only at the next token of the input sequence, never tracing back. Common prefixes in the production rules of the same variable have to be factored out to realize that feature. That process is also called left factoring and is visualized in figure 3.13 .

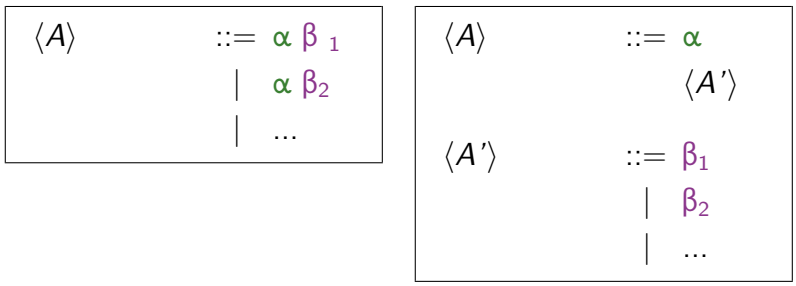

Figure 3.13: exp with right recursion

3.1.4 Construction of an LL(1) parser table

LL(1) stands for processing the input from left to right, right-most derivation and having a lookahead of 1 token, so no back tracing required. Using a parsing table, a LL(1) parser can directly make the rule picking decision by knowing the current non-terminal and the next terminal obtained from the input sequence: $NT \times T \mapsto P$. Before moving on to the actual language going to be used, the process of constructing a LL(1) parser table shall be outlined, which requires the concept of *First* and *Follow*. *First* and *Follow* both describe sets of terminals. *First* can furthermore contain the empty word ϵ and *Follow* the terminating symbol \$ (\$ is just an extra symbol appended to the input sequence in order to mark the ending). Continuing to construct the parser table, each variable is going to be assigned a pair of *First* and *Follow* sets. Considering the grammar in figure 3.14 , table T1 must be filled in.

Table T1: empty First-Follow table

Non-terminal	First	Follow
S		
A		
B		
C		

$$
\begin{array}{lll}
\langle S \rangle & ::= & \langle A \rangle\ \langle C \rangle\ \langle B \rangle \\
 & | & \langle C \rangle\ \text{`b'}\ \langle B \rangle \\
 & | & \langle B \rangle\ \text{`a'} \\
\langle A \rangle & ::= & \text{`d'}\ \text{`a'} \\
 & | & \langle B \rangle\ \langle C \rangle \\
\langle B \rangle & ::= & \text{`g'} \\
 & | & \epsilon \\
\langle C \rangle & ::= & \text{`h'} \\
 & | & \epsilon
\end{array}
$$

Figure 3.14: grammar to clarify the concept of First and Follow

First: *First* of a variable ⟨X⟩ is obtained by taking a look at each of its production rules Q_i, processing its sequence of symbols P_i, collecting *First* of P_i. To get *First* of P_i, look at its first symbol. If it is a terminal, add it to the set and leave P_i. If it is a variable ⟨Sub⟩, get *First* of ⟨Sub⟩. The next step depends on if *First* of ⟨Sub⟩ contains the empty word ϵ, for then ⟨Sub⟩ might be erased in P_i while parsing. If ϵ is not included or if the variable being investigated is the last symbol of P_i, add all of *First* of ⟨Sub⟩ to the set and leave P_i. If ϵ was contained and ⟨Sub⟩ was not the last symbol of P_i, everything except ϵ is added and P_i advances to the next symbol.

Follow: *Follow* of a variable ⟨X⟩ is obtained by taking a look at each of its occurrences in every production rule Q_i of the grammar, processing its remaining symbols after the occurrence of ⟨X⟩ P_i, collecting *Follow* of P_i. Additionally, start variables automatically get the terminator symbol \$ bestowed. If P_i contains symbols, get *First* of P_i. If *First* of P_i contains the empty word ϵ, add everything of *First* of P_i to the set except for ϵ and also get *Follow* of the variable Q_i resides on. If ϵ is not included, simply add everything of *First* of P_i. If P_i was empty to begin with, add *Follow* of the variable its original production rule Q_i resides on. Footnote: The *Follow* set never contains ϵ.

Since production rules may reference themselves or create a cycle dependency, both algorithms should mark which non-terminals were already visited. Using these algorithms for *First* and *Follow* - *Java* listings are provided in chapter 5: *Implementation* - the table can be filled in like in table T2 . Transcribing the First-Follow table, the promised predictive LL(1) parser table is about to erect. Therefore, each of the variables qualifies a row once again and the columns are made up of all terminals of the grammar, including the termination symbol ''.

Table T2: filled First-Follow table

Non-terminal	First	Follow
S	$\{a, b, d, g, h, e\}$	$\{\$\}$
A	$\{d, g, h, e\}$	$\{g, h, \$\}$
B	$\{g, e\}$	$\{a, g, h, \$\}$
C	$\{h, e\}$	$\{b, g, h, \$\}$

The procedure appears straightforward: For every variable ⟨X⟩ and every rule *P*, check the first symbol of *P*. If that symbol is a terminal, put *P* in the cell denoted by ⟨X⟩ and the actual terminal ($X \times t \mapsto \langle X \rangle ::=$ t). If it is a non-terminal ⟨Y⟩, put *P* in every cell denoted by ⟨X⟩ and any of the terminals in the *First* set of ⟨Y⟩ ($\forall y \in First(Y)$: $X \times y \mapsto \langle X \rangle ::=$ Y). If (as a last option) it should be the empty word ϵ, the rule will be to derive ⟨X⟩ to ϵ for all of the terminals of the Follow set of ⟨X⟩ ($\forall x \in Follow(X)$: $X \times x \mapsto \langle X \rangle ::= \epsilon$).

Table T3: predictive parser table

NT \ T	a	b	d	g	h	$
S						
A			‘d’ ‘a’	⟨B⟩ ⟨C⟩		
B	ϵ			ϵ	ϵ	ϵ
C		ϵ		ϵ	ϵ	ϵ

If any of the cells should be object to multiple entries using the method just described, the grammar has not been LL(1) as seeing only the next token is not enough to make a decision in that case. All cells that remain empty are instances of failure. If the parser comes across such a combination, it has to throw an exception and the input must not be accepted. It should be noted that the above described algorithm for developing a grammar does not ensure LL(1) behavior. Left factorization lifts the ambiguity evoked by a common prefix in multiple production rules of the same variable but is purely syntactic. Variables, whose decomposition may yield the same prefix of terminals, are not of further interest, e.g. the rules $\langle A \rangle ::=$ ‘a’ ‘b’ and $\langle A \rangle ::=$ ⟨A’⟩ ‘b’ cannot be unified/left factored even if ‘a’ $\subseteq$ ⟨A’⟩. Additionally, there may be overlapping with the *Follow* sets. For a grammar to be LL(1), the *First* sets of all production rules per variable must be disjoint and the intersection of the *First* and *Follow* sets of a variable must be empty in case the variables has an ϵ production rule. One type of parser that can handle LL(1) grammars is those that descend recursively. It is presented in chapter 5: *Implementation* . More can be read in.

3.2 Core language

3.2.1 Commands

Having previewed the example, now the core language shall be constructed. The *Hoare* calculus provides rules for the typical elementary imperative flow control elements and statements: variable assignment, selection, head-controlled loops and of course sequential composition. It matches that of *while* programs. Some more commonly known structures can be extrapolated. Foot-controlled loops are semantically equivalent to head-controlled loops with the loop body duplicated once in front of the loop for example:

```
1 do {
2   BODY;
3 } while (CONDITION)
4
5 //same as
6 BODY;
7 while (CONDITION) {
8   BODY;
9 }
```

Listing L1: Conversion foot-loop to while-loop (Java)

Count-controlled loops can be converted to head-controlled loops as well:

The ternary conditional assignment can be expanded to a selection:

```
1 x = pred ? y : z;
2
3 //same as
4 if (pred) x = y; else x = z;
```

Listing L2: Conversion conditional assignment to selection (Java)

```
1  for (INIT; CONDITION; INCREMENT) {
2    BODY;
3  }
4
5  //same as
6  INIT;
7  while (CONDITION) {
8    BODY;
9    INCREMENT;
10 }
```

Listing L3: Conversion count-controlled loop to while-loop (Java)

Although converting the old-school goto jumps to loops may not be trivial and without the use of additional support variables, it is possible to do so. This just to state that the concepts presented here are transferable to more elaborated structures as they exist in real programming languages.

⟨*assign*⟩ ::= ⟨*var*⟩ '=' ⟨*exp*⟩

⟨*alt*⟩ ::= 'IF' ⟨*bool_exp*⟩ 'THEN' ⟨*prog*⟩ 'ELSE' ⟨*prog*⟩ 'FI'

⟨*loop*⟩ ::= 'WHILE' ⟨*bool_exp*⟩ 'DO' ⟨*prog*⟩ 'OD'

⟨*skip*⟩ ::= 'SKIP'

Figure 3.15: commands

⟨*prog*⟩ ::= ⟨*prog*⟩ ';' ⟨*prog*⟩
| ⟨*skip*⟩
| ⟨*assign*⟩
| ⟨*alt*⟩
| ⟨*loop*⟩

Figure 3.16: ⟨prog⟩

The commands for variable assignment, if selection (alternative) and while loop are depicted in figure 3.15 . The variable assignment ⟨assign⟩ assigns ⟨exp⟩ to ⟨var⟩ (no variable subscription/arrays here). The if selection ⟨alt⟩ checks the truth value of ⟨bool_exp⟩. Should it evaluate to true, the ⟨prog⟩ of the THEN-branch will be executed, otherwise it will execute the ⟨prog⟩ of the ELSE-branch. The ⟨loop⟩ command checks the truth value of ⟨bool_exp⟩. Should it evaluate to false, the execution point will jump right after the loop. In case it is true, the inner ⟨prog⟩ will be called and afterwards the whole loop mechanism be repeated. Additionally, there is ⟨skip⟩, which does nothing of meaning but can be placed to suffice the syntax (like maybe if the ELSE-branch of the selection is superfluous). After having defined the single commands, they can be tied together via ⟨prog⟩ as shown in figure 3.16 , which is also the starting symbol, and put in a sequential composition using the ';' separator. The first rule of ⟨prog⟩ displays left-recursion which must be get rid of. This is easily solved in figure 3.17 . The addition of ⟨alt_ else⟩ and refactoring of ⟨alt⟩ renders the else branch of selections optional for convenience.

⟨*prog*⟩ ::= ⟨*cmd*⟩ ⟨*prog'*⟩

⟨*prog'*⟩ ::= ';' ⟨*cmd*⟩ ⟨*prog'*⟩
| ϵ

⟨*cmd*⟩ ::= ⟨*skip*⟩
| ⟨*assign*⟩
| ⟨*alt*⟩
| ⟨*while*⟩

⟨*skip*⟩ ::= 'SKIP'

⟨*assign*⟩ ::= 'id' ':=' ⟨*exp*⟩

⟨*alt*⟩ ::= 'IF' ⟨*bool_exp*⟩ 'THEN' ⟨*prog*⟩ ⟨*alt_else*⟩ 'FI'

⟨*alt_else*⟩ ::= 'ELSE' ⟨*prog*⟩
| ϵ

⟨*while*⟩ ::= 'WHILE' ⟨*bool_exp*⟩ 'DO' ⟨*prog*⟩ 'OD'

Figure 3.17: ⟨prog⟩

3.2.2 Numeric expressions

The ⟨exp⟩ and ⟨bool_exp⟩ non terminals remain open for definition. ⟨exp⟩ describes an expression. In normal programming languages, this could be of any data type. Here, it is restrained to numeric expressions. Floating-point operations in numerical systems are commonly imprecise respectively on the bit level, which would make a semantic observation more difficult. Some techniques to cope with fractions will be covered later.

⟨*exp*⟩ ::= ⟨*exp*⟩ '+' ⟨*exp*⟩
| ⟨*exp*⟩ '−' ⟨*exp*⟩
| ⟨*exp*⟩ '∗' ⟨*exp*⟩
| ⟨*exp*⟩ '/' ⟨*exp*⟩
| ⟨*exp*⟩ '^' ⟨*exp*⟩
| ⟨*exp*⟩ '!'
| 'id'
| 'num'

Figure 3.18: core ⟨exp⟩ grammar

The indication of this initial grammar is as it appears most intuitive. For example, a numeric expression might be a multiplication of two other ⟨exp⟩ or using any binary arithmetic operation for that matter. '+' stands for addition, '−' for subtraction, '∗' for multiplication, '/' for (integer) division, '^' for potentization and '!' for factorial (which is unary). The terminal 'id' matches a variable name (*[a-zA-Z][a-zA-Z0-9]**), 'num' is any integer literal (*[1-9][0-9]** | *0*). The grammar rules for ⟨exp⟩ contain left recursion and the precedence must be fixated to influence the setup of the syntax tree as described in section 3.1: *On languages and grammars* . First off, a total order of the operators must be established. It seems obvious to do it according to the mathematical notation rules:

$$+ \doteq - \lessdot * \doteq / \lessdot \hat{} \lessdot !$$

Table T4: operator table of ⟨exp⟩

	+	-	*	/	^	!
+	⋗	⋗	⋖	⋖	⋖	⋖
-	⋗	⋗	⋖	⋖	⋖	⋖
*	⋗	⋗	⋗	⋗	⋖	⋖
/	⋗	⋗	⋗	⋗	⋖	⋖
^	⋗	⋗	⋗	⋗	⋖	⋖
!	⋗	⋗	⋗	⋗	⋗	⋗

So addition and subtraction come with the lowest precedence, followed by multiplication and division, then follows exponentiation and finally the factorial operator stands at the top. The hierarchy is displayed in table T5 , too. Moreover, this table denotes the associativity of the binary operators: addition through division are left-associative but exponentiation is not. The expression 2^2^3 shall denote $2^{2^3} = 2^8 = 256$ as opposed to $(2^2)^3 = 4^3 = 64$. While semantics-wise, associativity does not matter for addition and multiplication, it is fixed to left bias on default to reduce ambiguity. Using the methods proposed in section 3.1: *On languages and grammars* , the grammar fragment for ⟨exp⟩ can be arranged. The first version in figure 3.19 shows the precedence levels. Associativity is applied in figure 3.20 , in figure 3.21 left recursion is abolished and finally the grammar has been exposed to left factoring in figure 3.22 . The last version encloses the auxiliary rule ⟨*exp_elem*⟩ ::= '(' ⟨exp⟩ ')', whose parentheses' encapsulation permits the prioritization of any operation over others.

$$
\begin{array}{lll}
\langle exp \rangle & ::= & \langle sum \rangle \\
\langle sum \rangle & ::= & \langle sum \rangle \text{ `+' } \langle sum \rangle \\
 & | & \langle sum \rangle \text{ `-' } \langle sum \rangle \\
 & | & \langle prod \rangle \\
\langle prod \rangle & ::= & \langle prod \rangle \text{ `*' } \langle prod \rangle \\
 & | & \langle prod \rangle \text{ `/' } \langle prod \rangle \\
 & | & \langle pow \rangle \\
\langle pow \rangle & ::= & \langle pow \rangle \text{ `\^{}' } \langle pow \rangle \\
 & | & \langle fact \rangle \\
\langle fact \rangle & ::= & \langle exp\_elem \rangle \text{ `!'} \\
 & | & \langle exp\_elem \rangle \\
\langle exp\_elem \rangle & ::= & \text{`id'} \\
 & | & \text{`num'}
\end{array}
$$

Figure 3.19: grammar of ⟨exp⟩ (precedence)

$$
\begin{array}{lll}
\langle exp \rangle & ::= & \langle sum \rangle \\
\langle sum \rangle & ::= & \langle sum \rangle \text{ `+' } \langle prod \rangle \\
 & | & \langle sum \rangle \text{ `-' } \langle prod \rangle \\
 & | & \langle prod \rangle \\
\langle prod \rangle & ::= & \langle prod \rangle \text{ `*' } \langle pow \rangle \\
 & | & \langle prod \rangle \text{ `/' } \langle pow \rangle \\
 & | & \langle pow \rangle \\
\langle pow \rangle & ::= & \langle fact \rangle \text{ `\^{}' } \langle pow \rangle \\
 & | & \langle fact \rangle \\
\langle fact \rangle & ::= & \langle exp\_elem \rangle \text{ `!'} \\
 & | & \langle exp\_elem \rangle \\
\langle exp\_elem \rangle & ::= & \text{`id'} \\
 & | & \text{`num'}
\end{array}
$$

Figure 3.20: core grammar of ⟨exp⟩ (associativity)

$$
\begin{array}{lll}
\langle \textit{exp} \rangle & ::= & \langle \textit{sum} \rangle \\
\langle \textit{sum} \rangle & ::= & \text{`+'}\ \langle \textit{prod} \rangle\ \langle \textit{sum'} \rangle \\
 & | & \text{`-'}\ \langle \textit{prod} \rangle\ \langle \textit{sum'} \rangle \\
 & | & \epsilon \\
\langle \textit{prod} \rangle & ::= & \langle \textit{pow} \rangle\ \langle \textit{prod'} \rangle \\
\langle \textit{prod'} \rangle & ::= & \text{`*'}\ \langle \textit{pow} \rangle\ \langle \textit{prod'} \rangle \\
 & | & \text{`/'}\ \langle \textit{pow} \rangle\ \langle \textit{prod'} \rangle \\
 & | & \epsilon \\
\langle \textit{pow} \rangle & ::= & \langle \textit{fact} \rangle\ \text{`\^{}'}\ \langle \textit{pow} \rangle \\
 & | & \langle \textit{fact} \rangle \\
\langle \textit{fact} \rangle & ::= & \langle \textit{exp\_elem} \rangle\ \text{`!'} \\
 & | & \langle \textit{exp\_elem} \rangle \\
\langle \textit{exp\_elem} \rangle & ::= & \text{`id'} \\
 & | & \text{`num'}
\end{array}
$$

Figure 3.21: core grammar of ⟨exp⟩ (right recursive)

$$
\begin{array}{lll}
\langle exp \rangle & ::= & \langle sum \rangle \\
\langle sum \rangle & ::= & \langle prod \rangle\ \langle sum' \rangle \\
\langle sum' \rangle & ::= & \text{`+'}\ \langle prod \rangle\ \langle sum' \rangle \\
 & | & \text{`-'}\ \langle prod \rangle\ \langle sum' \rangle \\
 & | & \epsilon \\
\langle prod \rangle & ::= & \langle pow \rangle\ \langle prod' \rangle \\
\langle prod' \rangle & ::= & \text{`*'}\ \langle pow \rangle\ \langle prod' \rangle \\
 & | & \text{`/'}\ \langle pow \rangle\ \langle prod' \rangle \\
 & | & \epsilon \\
\langle pow \rangle & ::= & \langle fact \rangle\ \langle pow' \rangle \\
\langle pow' \rangle & ::= & \text{`\^{}'}\ \langle pow \rangle \\
 & | & \epsilon \\
\langle fact \rangle & ::= & \langle exp\_elem \rangle\ \langle fact' \rangle \\
\langle fact' \rangle & ::= & \text{`!'} \\
 & | & \epsilon \\
\langle exp\_elem \rangle & ::= & \text{`id'} \\
 & | & \text{`num'} \\
 & | & \text{`('}\ \langle exp \rangle\ \text{`)'}
\end{array}
$$

Figure 3.22: grammar of ⟨exp⟩ (final)

3.2.3 Boolean expressions

The same procedure must be done for boolean expressions. ⟨bool_exp⟩s yield a boolean value. Here, it may either be a conjunction (‘&’), a disjunction (‘|’), a negation (‘~’) or an elementary entity: The comparison of two ⟨exp⟩ via the comparison operators (less (‘<’), greater (‘>’), less or equal(‘<=’), greater or equal(‘>=’), equal (‘=’) or unequal (‘<>’)) or the boolean literals ‘true’ or ‘false’.

$$\text{`|'} \lessdot \text{`\&'} \lessdot \text{`\~{}'} \lessdot \text{`<'} \doteq \text{`>'} \doteq \text{`<='} \doteq \text{`>='} \doteq \text{`='} \doteq \text{`<>'}$$

Table T5: operator table of ⟨exp⟩

	||	&&	¬	compOp
||	⩾	⩽	⩽	⩽
&&	⩾	⩾	⩽	⩽
¬	⩾	⩾	⩾	⩾
compOp	⩽	⩽	⩽	⩾

⟨*bool_exp*⟩ ::= ⟨*bool_exp*⟩ '&' ⟨*bool_exp*⟩
| ⟨*bool_exp*⟩ '|' ⟨*bool_exp*⟩
| '~' ⟨*bool_exp*⟩
| ⟨*exp*⟩ '<' ⟨*exp*⟩
| ⟨*exp*⟩ '>' ⟨*exp*⟩
| ⟨*exp*⟩ '<=' ⟨*exp*⟩
| ⟨*exp*⟩ '>=' ⟨*exp*⟩
| ⟨*exp*⟩ '=' ⟨*exp*⟩
| ⟨*exp*⟩ '<>' ⟨*exp*⟩
| '`true`'
| '`false`'

Figure 3.23: core grammar of ⟨bool_exp⟩ (raw)

⟨*bool_exp*⟩ ::= ⟨*bool_or*⟩

⟨*bool_or*⟩ ::= ⟨*bool_and*⟩ '|' ⟨*bool_and*⟩

⟨*bool_and*⟩ ::= ⟨*bool_neg*⟩ '&' ⟨*bool_{}neg*⟩

⟨*bool_neg*⟩ ::= '~' ⟨*bool_elem*⟩
| ⟨*bool_elem*⟩

⟨*bool_elem*⟩ ::= ⟨*exp*⟩ '<' ⟨*exp*⟩
| ⟨*exp*⟩ '>' ⟨*exp*⟩
| ⟨*exp*⟩ '<=' ⟨*exp*⟩
| ⟨*exp*⟩ '>=' ⟨*exp*⟩
| ⟨*exp*⟩ '=' ⟨*exp*⟩
| ⟨*exp*⟩ '<>' ⟨*exp*⟩
| '`true`'
| '`false`'

Figure 3.24: grammar of ⟨bool_exp⟩ (precedence)

$$
\begin{aligned}
\langle \textit{bool\_exp} \rangle &::= \langle \textit{bool\_or} \rangle \\
\langle \textit{bool\_or} \rangle &::= \langle \textit{bool\_and} \rangle \ \langle \textit{bool\_or'} \rangle \\
\langle \textit{bool\_or'} \rangle &::= \text{`|'} \ \langle \textit{bool\_and} \rangle \ \langle \textit{bool\_or'} \rangle \\
&\quad | \quad \text{`}\varepsilon\text{'} \\
\langle \textit{bool\_and} \rangle &::= \langle \textit{bool\_neg} \rangle \ \langle \textit{bool\_and'} \rangle \\
\langle \textit{bool\_and'} \rangle &::= \text{`\&'} \ \langle \textit{bool\_neg} \rangle \ \langle \textit{bool\_and'} \rangle \\
&\quad | \quad \epsilon \\
\langle \textit{bool\_neg} \rangle &::= \langle \textit{bool\_elem} \rangle \\
&\quad | \quad \text{` '} \ \langle \textit{bool\_elem} \rangle \\
\langle \textit{bool\_elem} \rangle &::= \langle \textit{exp} \rangle \ \text{`<'} \ \langle \textit{exp} \rangle \\
&\quad | \quad \langle \textit{exp} \rangle \ \text{`<'} \ \langle \textit{exp} \rangle \\
&\quad | \quad \langle \textit{exp} \rangle \ \text{`>'} \ \langle \textit{exp} \rangle \\
&\quad | \quad \langle \textit{exp} \rangle \ \text{`<='} \ \langle \textit{exp} \rangle \\
&\quad | \quad \langle \textit{exp} \rangle \ \text{`>='} \ \langle \textit{exp} \rangle \\
&\quad | \quad \langle \textit{exp} \rangle \ \text{`='} \ \langle \textit{exp} \rangle \\
&\quad | \quad \langle \textit{exp} \rangle \ \text{`<>'} \ \langle \textit{exp} \rangle \\
&\quad | \quad \text{`}\texttt{true}\text{'} \\
&\quad | \quad \text{`}\texttt{false}\text{'} \\
&\quad | \quad \text{`}\texttt{[}\text{'} \ \langle \textit{bool\_exp} \rangle \ \text{`}\texttt{]}\text{'}
\end{aligned}
$$

Figure 3.25: grammar of ⟨bool_exp⟩ (final)

Complying with the steps in accordance to the previous section, one arrives at figure 3.25 . An analogous parentheses mechanism (⟨*bool_ elem*⟩ ::= '[' ⟨bool_ exp⟩ ']') is inserted but it uses brackets instead of parentheses. The reason is that ⟨bool_ elem⟩ possesses rules starting with ⟨exp⟩. ⟨exp⟩ can already start with '(', using '(' again would break the LL(1) trait (*First* sets of ⟨exp⟩ and ⟨bool_exp⟩ would not be disjunct) and are therefore not desirable.

This concludes the language definition subject to the forthcoming verifying methods. It will be extended by the proof specification decorations later on. The whole language up to now along with the here omitted parser table can be viewed in appendix D: *Grammmar for while programs* . Line breaks and white space is tolerated for readability and used as delimiters of keywords because e.g. *IFa* should be regarded as an identifier 'IFa' rather than the keyword 'IF' plus the identifier 'a'.

3.3 Semantic tree

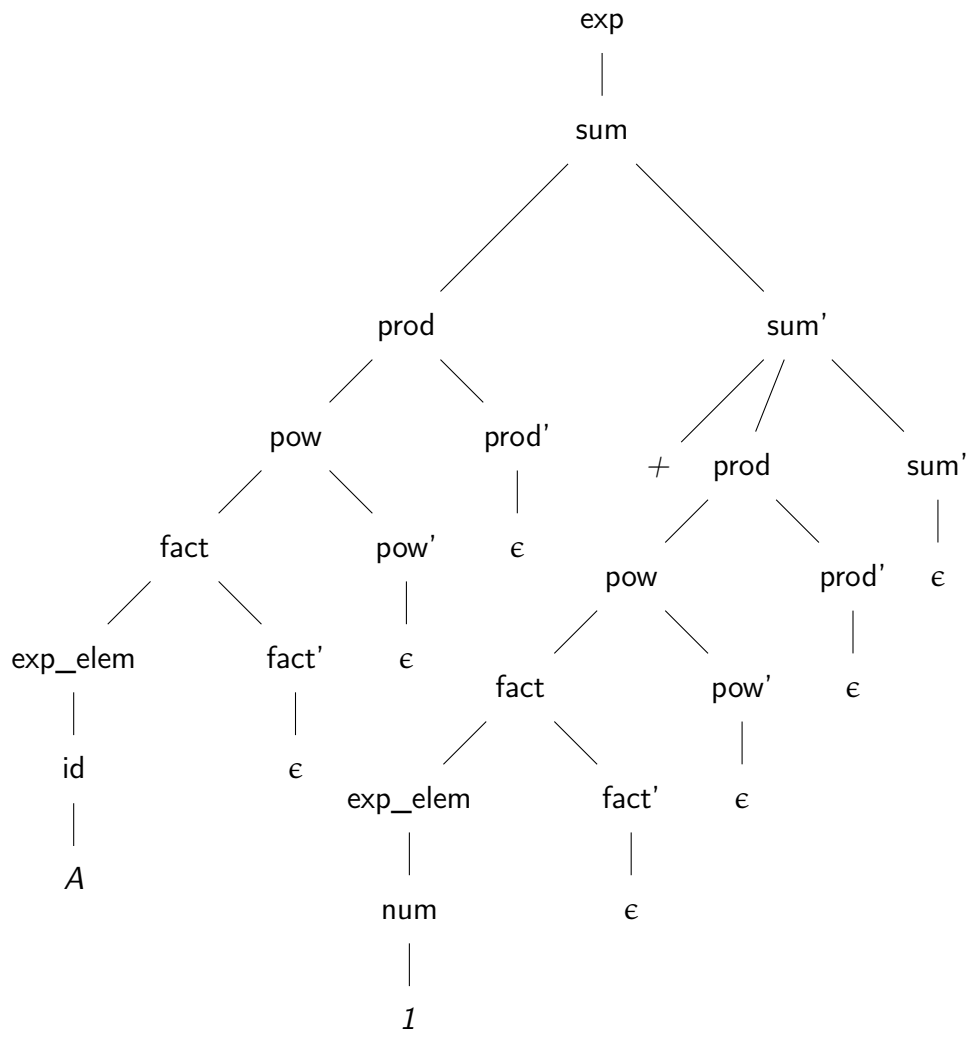

Figure 3.26: extensive syntax tree of *A+1*

The syntax tree constructed so far was meant to match the workings of a LL(1) parser. It creates a lot of unnecessary levels and the right-recursion spawns extended cascades and ϵ-branches. Ex: The syntax tree of a simple expression *A+1* produces the syntax tree in figure 3.26 . As can easily be observed, this representation is immensely oversized, contains redundancy and traversing it would be convoluted. Each additional summand appends another operator and another $\langle$prod$\rangle$-sub tree. It turns out to be impractical for further processing, which is why it should be converted into a more semantically meaningful tree. The deflated result of such is shown in figure 3.27 . The expression is reduced to just a sum of an id and a numeric value, as a one would intuitively identify it. The operator is also left out because it can be implied by the fact that the treated node is a sum (a minus sign could be realized by inserting another intermediate *negation* node, not defined yet). More on the technical aspects of

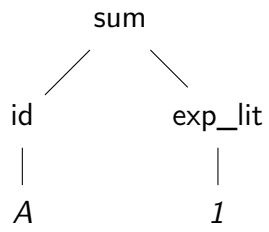

Figure 3.27: semantic tree of *A+1*

typification of nodes and the transition from one tree to another are to be unveiled in the *Implementation* chapter. The considerations up to this point are to showcase the elements of the language to be verified and to give credit to the required preparatory steps. The author will henceforth be using the term *semantic tree* to refer to the new type when trying to emphasize a distinction between it and the original *syntax tree*. The usual term found in literature is *abstract syntax tree* while *semantic tree* is reserved for a quite similar entity in logic. The author reckons the meaning can be generalized to encompass the described topic and is fitting.

Chapter 4
4 How to prove

4.1 Of operational semantics

Speaking about verification of programs, it is of utmost importance to know what a program does exactly, namely its semantics. These are inductively defined and should be formally specified. The model checking part in the introduction mentioned that the current point of execution (the instruction position) and the values of the involved variables identify a snapshot of the program in its entirety. Theoretically, in a deterministic environment where the next instruction to handle is fixed, the behavior can be foretold as the machine would compute.

The idea now is to use the semantics defined at the end of the previous chapter in order to make statements about the execution of a given program. This is similar to actually running the program, traversing the entered lines/statements of code in succession and keeping memory of the current variable values. The mapping of all variables used inside the program as well as additional helper variables to specific values is called a state. As according to the introduced language, only the assignment instruction has the ability to alter the state.

Definition 5. *Be S a program. $var(S)$ denotes the set of all variables occurring in S.*

Ex: In listing L1 $var(S_{var}) = \{x, y, z, q\}$

```
1 x := x + 1;
2 y := z;
3 IF z = q THEN
4   y := y + 1
5 FI
```

Listing L1: S_{var} example

Definition 6. *Be S a program. $change(S)$ denotes the set of all variables modified in S (each one that has at least one assignment, irrespective to the instruction being actually executed).*

Ex: In listing L1 $change(S_{var}) = \{x, y\}$

Definition 7. *A state is a function, mapping a value to every variable in $var(S)$ as well as to auxiliary variables.*

Definition 8. *Be S a program and σ a state. The pair $< S, \sigma >$ is called a configuration.*

Through computations, a program can shift between configurations. E denotes the end of the program.

Definition 9. *Be S a program and σ a state. The pair $< S, \sigma >$ is called a configuration.*

Definition 10. *Be S a program and σ a state. The pair $< S, \sigma >$ is called a configuration.*

Ex: $< x := 1, x = 0 > - >< E, x = 1 >$

Definition 11. *M[[s]] mapping reachable states from initial states, in det. programs in PW exactly one*

The semantics of the input words are inductively defined, which means to make sense of a construct, its components are recursively examined. To obtain the semantics of an expression for example, the involved variables need to be resolved for their current value.

proof outline -> assertional decoration

assertions, extension of language partial correctness, total correctness

The following axioms and rules of the *transition system* are transcribed from page 59 of [APdBO10]:

$$< skip, s > \quad < E, s > \tag{4.1}$$

$$< u = t, s > \quad < E, s[u = s(t)] > \tag{4.2}$$

$$\frac{< S_1, \sigma > \quad < S_2, \tau >}{< S_1; S, \sigma > \quad < S_2; S, \tau >} \tag{4.3}$$

$$< IF\ COND\ THEN\ S_1\ ELSE\ S_2\ FI, \sigma > \rightarrow\ < S_1, \sigma >\ where\ \sigma \models COND \tag{4.4}$$

$$< IF\ COND\ THEN\ S_1\ ELSE\ S_2\ FI, \sigma > \rightarrow\ < S_2, \sigma >\ where\ \sigma \not\models COND \tag{4.5}$$

$$< WHILE\ COND\ DO\ S\ OD, \sigma > \rightarrow\ \ < S; WHILE\ CONDITION\ DO\ S\ OD, \sigma >\ where\ \sigma \models C \tag{4.6}$$

$$< WHILE\ COND\ DO\ S\ OD, \sigma > \rightarrow\ \ < E, \sigma >\ where\ \sigma \not\models COND \tag{4.7}$$

(4.1) states that *skip* does not cause a state change and just transitions to *E*. In (4.2), the state is altered by replacing the value of the variable u by t. (4.3) is the propagation characteristic of the sequential composition: the first part is evaluated and passed to the second part. (4.4) and (4.5) handle alternatives and (4.6) together with (4.7) loops. It can be noticed that the assignment is the only statement to evoke a state change and that only the loop ((4.6)) has the ability to have the "remaining" program to be processed of the target configuration larger than that of the source configuration. Thus it is the only one with potential for divergence: to make the program never end.

4.2 Transition to Hoare calculus

As hinted in the introduction, correctness is a somewhat vague term. For all it may concern, a program could be declared correct for halting, containing specific statements, fulfilling liveliness parameters (the program keeps visiting "good" configurations) or safety conditions (it never comes across "bad" configurations) for instance.

The now presented *Hoare* calculus defines relations of input/output behavior. That is, if a program starts in a state satisfying a precondition p and executes a program S, it is guaranteed to have taken on a state fulfilling a postcondition q afterwards.

$$\{p\}\ S\ \{q\} \tag{4.8}$$

The notation is as shown in (4.8) and named *Hoare triple*. The conditions, also called assertions, are put in curly braces although *C.A.R. Hoare* originally did it the other way around ($p\ \{S\}\ q$). The difference between assertion and state is that an assertion is a predicate for allowed states, it designates a set of states. Using assertions, the programs of the developed language can be decorated to express a supposed to have behavior:

```
1 PRE {x<y}
2   x=x+y
3 POST {x<2*y}
```

Listing L2: Decoration example

In that sense, of course, this is an arbitrary proposition that has to be verified. Do all states where x is smaller than y initially lead to a state where x is smaller than $2 * y$ after having added y to x? It may be comprehensible in this case but less so for more complex problems. Fortunately, there is the *Hoare* calculus to help systematizing them.

HOARE CALCULUS (PW):

SKIP AXIOM:

$$\{p\}\ \mathbf{SKIP}\ \{p\} \tag{4.9}$$

ASSIGNMENT AXIOM:

$$\{p[u := t]\}\ u := t\ \{p\} \tag{4.10}$$

COMPOSITION RULE:

$$\frac{\{p\}\ S_1\ \{r\}, \{r\}\ S_2\ \{q\}}{\{p\}\ S_1; S_2\ \{q\}} \tag{4.11}$$

CONDITIONAL RULE:

$$\frac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \mathbf{IF}\ B\ \mathbf{THEN}\ S_1\ \mathbf{ELSE}\ S_2\ \mathbf{FI}\ \{q\}} \tag{4.12}$$

LOOP RULE:

$$\frac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \mathbf{WHILE}\ B\ \mathbf{DO}\ S\ \mathbf{OD}\ \{p \wedge \neg B\}} \tag{4.13}$$

CONSEQUENCE RULE:

$$\frac{p \to p_1, \{p_1\}\ S\ \{q_1\}, q1 \to q}{\{p\}\ S\ \{q\}} \tag{4.14}$$

as listed in [APdBO10] 65f.

More precisely, there are two versions. The above one is for partial correctness only. The case of divergence, a program never halting, is not covered here. The proof system consisting of axioms and rules establishes a relationship between pre- and postconditions of a set of basic programming constructs. These are the groundwork for further verification ventures. The horizontal bar is a fancier notation for an implication: If all of the conditions above the bar hold true, the statements below it are ensured to be correct as well.

Again, the $SKIP$ axiom in (4.9) confirms that 'SKIP' is unable to do anything to the state. The $ASSIGNMENT$ axiom says that the precondition matches the postcondition only that all occurrences of the variable are replaced by the new value within the precondition. This is a bit problematic since substitution is not a reversible (bijective) procedure, ergo a left to right evaluation becomes difficult. The composition displays its transitive properties once more in (4.11). To obtain the postcondition of a composition, the postcondition of its first part is fed as precondition to the second part or vice versa. The $CONDITIONAL$ rule (4.12) merges the triples of the individual branches with the respectively associated condition for entering that branch. Perhaps most intriguing is the rule responsible for loops in (4.13). The precondition becomes a conjunction with the negation of the loop condition but that is only the case (this transformation is only allowed to be conducted) if the conjunction of the precondition and the loop condition applied to the loop body ends in the precondition. p is called a loop invariant in that regard because it does not change within the iteration. The binding material to plug the holes like enhancing the versatility of the $LOOP$ rule is found in the $CONSEQUENCE$ rule (4.14). A precondition can be strengthened by finding another one that implies the former precondition. On the other hand, a postcondition can always be relaxed by a weaker assertion. Essentially, the strongest predicate would be $\{false\}$, enveloping no state. A program annotated with $\{false\}$ at the end should never be found correct unless it diverges. The weakest assertion is $\{true\}$, encompassing all states, which is like the cartesian product of all possible variable values. Both strengthening and weakening must be exerted with care, for overdoing it would trigger type I "false positive" errors. A program could be mistakenly identified as incorrect because e.g. the postcondition was needlessly rendered too weak and does no longer imply the final statement. [Kle09] p. 228 additionally states:

$$\frac{\{p\}\ S_1\ \{q_1\} \wedge \{p\}\ S_2\ \{q_2\}}{\{p\}\ \textbf{IF}\ B\ \textbf{THEN}\ S_1\ \textbf{ELSE}\ S_2\ \textbf{FI}\ \{q_1 \vee q_2\}} \tag{4.15}$$

and

$$\frac{var(B) \notin change(S_1) \wedge var(B) \notin change(S_2)}{\{p\}\ \textbf{IF}\ B\ \textbf{THEN}\ S_1\ \textbf{ELSE}\ S_2\ \textbf{FI}\ \{(B \rightarrow q_1) \wedge (\neg B \rightarrow q_2)\}} \tag{4.16}$$

The *Hoare* calculus by itself does not specify an algorithm yet. There is some flexibility in there. The analysis of a program may commence from the beginning onwards or push up from the ending, or even a combination thereof. However, there are a couple of reasons why the second option seems preferable. The assignment axiom does present an explicit instruction to receive the precondition from the postcondition. Going from left to right requires more thinking and case distinction.

Assuming that the precondition p does not include the modified variable x, it would

be intuitive that the strongest derived postcondition just adds the clause of x possessing the newly assigned value:

$$\{p\}\ x := 2\ \{p \wedge x = 2\} \tag{4.17}$$

If p does not contain x and the assigned value is a function of x, as there is no presumption about x, p will stay the same:

$$\{p\}x = x + 2\{p\} \tag{4.18}$$

or one could insert some auxiliary anchor variable:

$$\{p \wedge X = x\}x = x + 2\{p \wedge X = x - 2\} \tag{4.19}$$

If p contains x and the value expression does not, the clauses in p that contain x would have to be discarded and and x possessing the newly assigned value added:

$$\{p\}\ x := 2\ \{p[x :=] \wedge x := 2\} \tag{4.20}$$

More sustainable seems to be the *wlp* (weakest liberal precondition) algorithm by *E. W. Dijkstra*:

$$wlp(\mathbf{SKIP}, q) \leftrightarrow q \tag{4.21}$$

$$wlp(u := t, q) \leftrightarrow q[u := t] \tag{4.22}$$

$$wlp(S_1; S_2, q) \leftrightarrow wlp(S_1, wlp(S_2, q)) \tag{4.23}$$

$$\begin{gathered} wlp(\mathbf{IF}\ B\ \mathbf{THEN}\ S_1\ \mathbf{ELSE}\ S_2\ \mathbf{FI}, q) \\ \leftrightarrow \\ (B \wedge wlp(S_1, q)) \vee (\neg B \wedge wlp(S_2, q)) \end{gathered} \tag{4.24}$$

$$\begin{gathered} wlp(\mathbf{WHILE}\ B\ \mathbf{DO}\ S_1\ \mathbf{OD}, q) \wedge B \\ \rightarrow \\ wlp(S_1, wlp(\mathbf{WHILE}\ B\ \mathbf{DO}\ S_1\ \mathbf{OD}, q)) \end{gathered} \tag{4.25}$$

$$wlp(\mathbf{WHILE}\ B\ \mathbf{DO}\ S_1\ \mathbf{OD}, q) \wedge \neg B \rightarrow q \tag{4.26}$$

$$\models \{p\}\ S\ \{q\} \iff p \rightarrow wlp(S, q) \tag{4.27}$$

as listed in [APdBO10] 87f.

The wlp is a function supposed to return the weakest liberal precondition of a program and a postcondition. (4.23) indicates that the program is processed in reverse. In a composition, the wlp of S_1 depends on the wlp of S_2, so S_2 must be evaluated first. The already addressed skip and variable assignment handling is the same as before. The alternative is transformed into a logical disjunction of both branches. It is unknown which branch will be executed at runtime, so this can be imagined like the least common multiple. The problem of finding a loop invariant remains but there are statements concerning it.

So the rough progressed algorithm is as follows: Apply wlp to the overall program (the root $\langle$prog$\rangle$) and the given postcondition. Depending on the current program part, act accordingly:

Skip: Return the postcondition.

Assignment: To get the precondition, replace all occurrences of the assigned variable in the postcondition by the new value. This means the syntax/semantic tree of the assertion must be recursively examined for the variable.

Composition: Find wlp of the rear program part using the given postcondition, then find wlp of the anterior part by supplying the previously received precondition to determine the precondition of the composition. Of course this can be nested. This corresponds to a syntax tree from the language definition being descended rightwards, then working the way back up in accordance to a stack (recursive calls).

Alternative: Find wlp of the program in the **THEN**-branch (p_then) and the **ELSE**-branch (p_else) in arbitrary order, then return $B \wedge p_then \vee \neg B \wedge p_else$ with B being the condition of the alternative. This possesses the potential of easily inflating the precondition.

Loop: Try to find a loop invariant p automatically (some ideas are discussed in section 4.3: *sec:Challenges*) otherwise ask the user to present one. If it is indeed a loop invariant, it can simply be returned but it is unlikely to be sure of it (both automatic retrieval and user query), so doing the next inspection steps appears reasonable. Check if $p \wedge \neg B$ implies the available postcondition (if it is an equivalent or stronger version of it). It cannot be a fitting invariant if that is not the case, retry another one. Find wlp of the loop body with p as the postcondition, then check the returned predicate whether it is implied by $p \wedge B$. It cannot be a fitting invariant if that is not the case, retry another one. With both checks being successful, p can be picked as precondition of the loop. However, proving that one predicate implies another is an undecidable problem

of its own (see section 4.3: *sec:Challenges*), thus it cannot be fully automatized, either, and must be user-supported.

Ex:

4.3 sec:Challenges

4.3.1 Finding invariants

A main challenge of using the Hoare calculus consists of finding fitting loop invariants. Those invariants are supposed to imply the postcondition.

p ->

4.3.2 Resolving implications

Chapter 5
5 Implementation

5.1 Java, Surface

The verifier tool is implemented using the *Java* programming language. Since the user input codes may vary, should be able to be changed on-the-fly for experimentation purposes and more user input is required, a graphical user interface seems plausible. Moreover, verifying assistance should ideally be integrated into an *IDE*, so that the context would be a better fit. Yet for simplicity, independence of implementation and concentration on the core algorithms, the idea of a standalone application surpasses the one of a plug-in for an existing IDE. *JavaFX* is chosen as a framework for the *GUI*. It enforces the *MVC* pattern, thereby separates the graphical representation from the programming logic in an uniform way. Since the framework lacks a widget for so-called *rich text*[1], which allows for individual styling of characters inside a text area, the *RichTextFX* library by *TomasMikula* is added in order to be able to properly illustrate both inputs and outputs. That package also contains a dedicated widget for code areas. Features like line numbers, highlighting of keywords or breakpoints can be realized. figure 5.1 shows the main window of the GUI.

The program provides basic text editor functionality: opening, saving and creating new files. Multiple files can be opened at once, each one being represented by a tab. The displayed tab can be split into multiple views (figure 5.2). Besides showing the input code (1), this can reveal the lexer-generated tokens (2) and the syntax tree (3) produced by the parser for introspection purposes. The branches that only contain the empty word are hidden on default since the tree can get pretty convoluted even without them. More output is visible in the console () and the syntax chart, which is an alternative representation of the syntax tree. After the code has been parsed, the interactive proving functionality becomes activatable (*H* button).

[1] https://techterms.com/definition/richtext

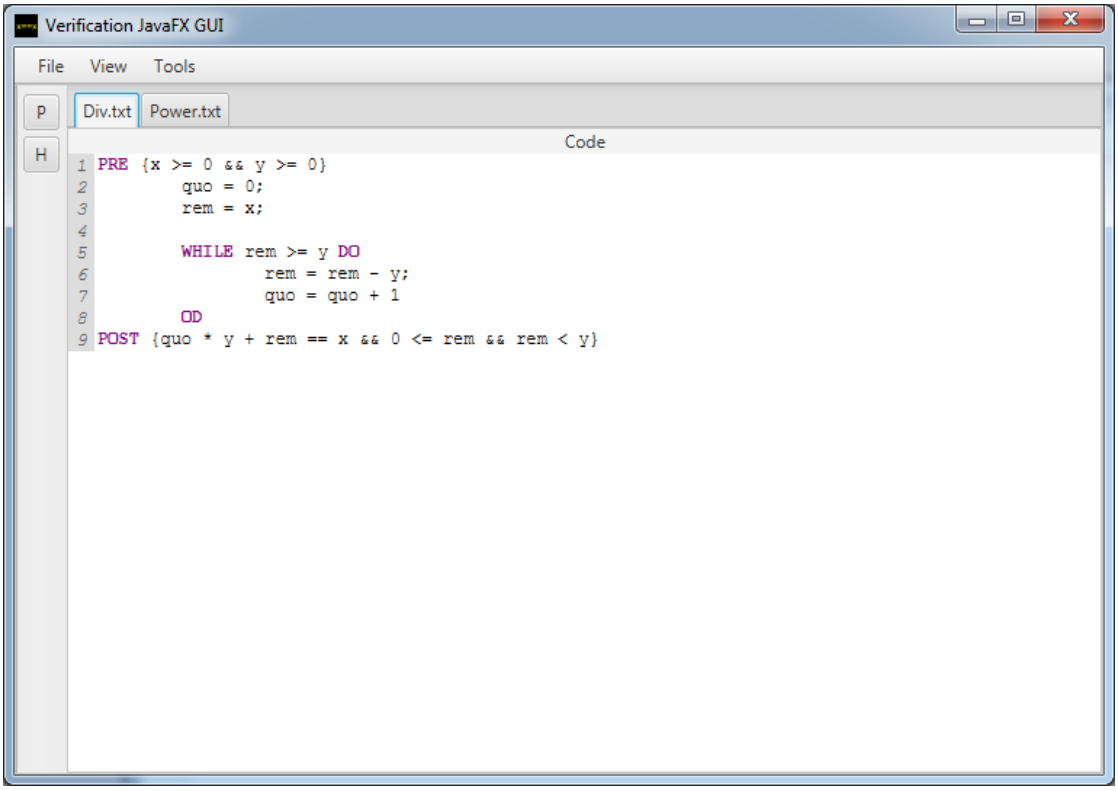

Figure 5.1: Main window

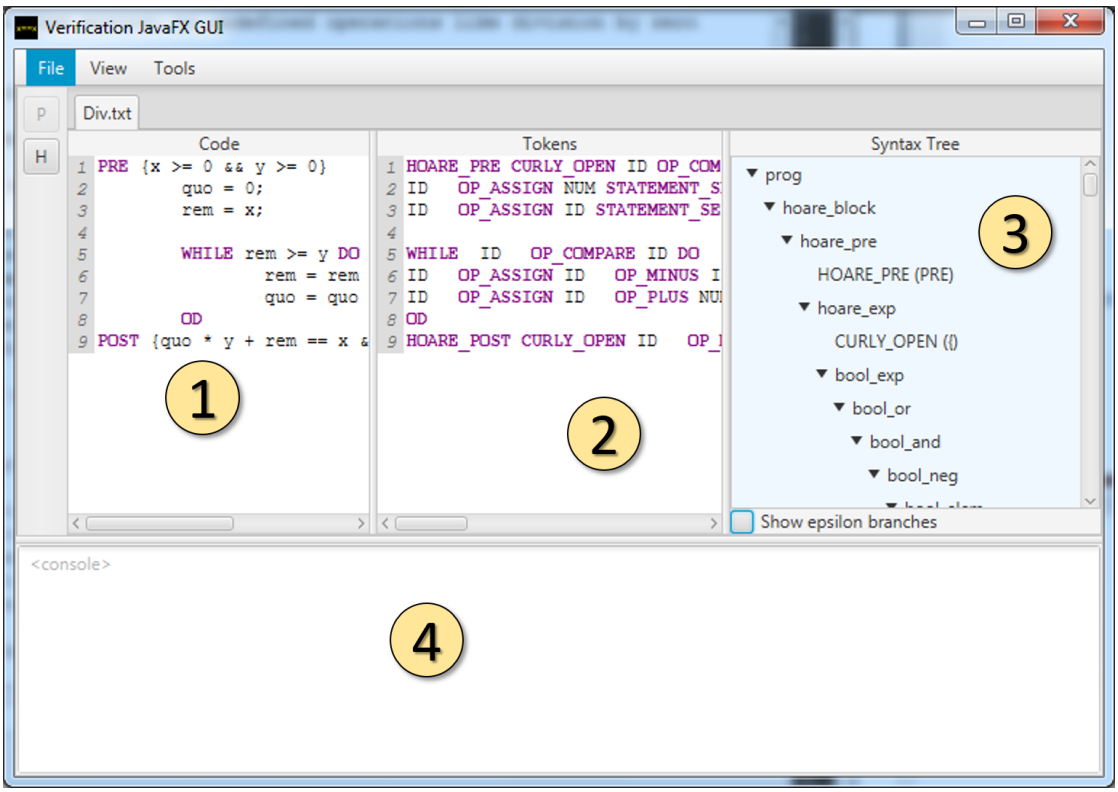

Figure 5.2: Views

5.2 Grammar, Lexer, parser

A grammar hosts terminals and non terminal symbols, which come with their respective rules. Terminals are defined by lexer rules. Those are either regular expressions or fixed terms especially used for keywords. Parser rules for non terminals consist of an ordered list of a combination of non terminals and terminals. Therefore, *Symbol* was chosen as an abstract class generalizing *Terminal* and *NonTerminal*. The relationship is depicted in figure 5.3 and an example for the ⟨exp⟩ grammar is found in listing L1 .

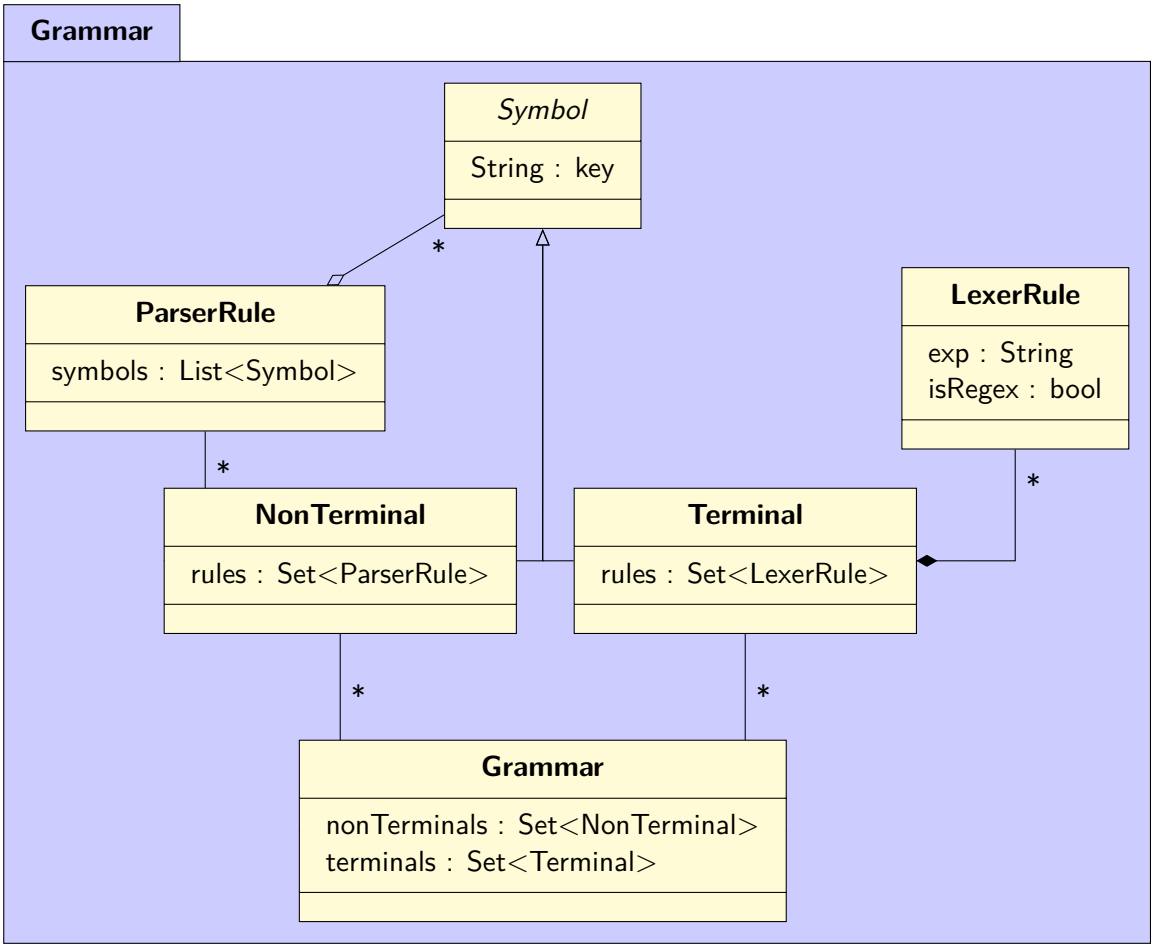

Figure 5.3: grammar-related class diagram

```
1  public ExpGrammar() {
2    super();
3
4    //lexer rules
5    TERMINAL_NUM = createTerminal("NUM");
6    TERMINAL_PAREN_OPEN = createTerminal("PAREN_OPEN");
7    TERMINAL_PAREN_CLOSE = createTerminal("PAREN_CLOSE");
8    TERMINAL_OP_PLUS = createTerminal("OP_PLUS");
9    TERMINAL_OP_MINUS = createTerminal("OP_MINUS");
10   TERMINAL_OP_MULT = createTerminal("OP_MULT");
11   TERMINAL_OP_DIV = createTerminal("OP_DIV");
12   TERMINAL_OP_POW = createTerminal("OP_POW");
13   TERMINAL_OP_FACTORIAL = createTerminal("OP_FACTORIAL");
14   TERMINAL_ID = createTerminal("ID");
```

```
15
16      TERMINAL_NUM.addRuleRegEx("[1-9][0-9]*");
17      TERMINAL_NUM.addRuleRegEx("0");
18
19      TERMINAL_PAREN_OPEN.addRule("(");
20      TERMINAL_PAREN_CLOSE.addRule(")");
21      TERMINAL_OP_PLUS.addRule("+");
22      TERMINAL_OP_MINUS.addRule("-");
23      TERMINAL_OP_MULT.addRule("*");
24      TERMINAL_OP_DIV.addRule("/");
25      TERMINAL_OP_POW.addRule("^");
26      TERMINAL_OP_FACTORIAL.addRule("!");
27
28      TERMINAL_ID.addRuleRegEx("[a-zA-Z][a-zA-Z0-9]*");
29
30      //parser rules
31      NON_TERMINAL_EXP = createNonTerminal("exp");
32      NON_TERMINAL_EXP_ = createNonTerminal("exp'");
33      NON_TERMINAL_FACTOR = createNonTerminal("factor");
34      NON_TERMINAL_FACTOR_ = createNonTerminal("factor'");
35      NON_TERMINAL_POW = createNonTerminal("pow");
36      NON_TERMINAL_POW_ = createNonTerminal("pow'");
37      NON_TERMINAL_FACTORIAL = createNonTerminal("factorial");
38      NON_TERMINAL_FACTORIAL_ = createNonTerminal("factorial'");
39      NON_TERMINAL_EXP_ELEMENTARY = createNonTerminal("exp_elementary");
40
41      createRule(NON_TERMINAL_EXP, "factor␣exp'");
42
43      createRule(NON_TERMINAL_EXP_, "OP_PLUS␣factor␣exp'");
44      createRule(NON_TERMINAL_EXP_, "OP_MINUS␣factor␣exp'");
45      createRule(NON_TERMINAL_EXP_, Terminal.EPSILON);
46
47      createRule(NON_TERMINAL_FACTOR, "pow␣factor'");
48
49      createRule(NON_TERMINAL_FACTOR_, "OP_MULT␣pow␣factor'");
50      createRule(NON_TERMINAL_FACTOR_, "OP_DIV␣pow␣factor'");
51      createRule(NON_TERMINAL_FACTOR_, Terminal.EPSILON);
52
53      createRule(NON_TERMINAL_POW, "factorial␣pow'");
54
55      createRule(NON_TERMINAL_POW_, "OP_POW␣pow");
56      createRule(NON_TERMINAL_POW_, Terminal.EPSILON);
57
58      createRule(NON_TERMINAL_FACTORIAL, "exp_elementary␣factorial'");
59
60      createRule(NON_TERMINAL_FACTORIAL_, "OP_FACTORIAL");
61      createRule(NON_TERMINAL_FACTORIAL_, Terminal.EPSILON);
62
63      createRule(NON_TERMINAL_EXP_ELEMENTARY, "ID");
64      createRule(NON_TERMINAL_EXP_ELEMENTARY, "NUM");
65      createRule(NON_TERMINAL_EXP_ELEMENTARY, "PAREN_OPEN␣exp␣PAREN_CLOSE");
66
67      //finalize
68      setStartParserRule(NON_TERMINAL_EXP);
69
70      updatePredictiveParserTable();
71  }
```

Listing L1: Implementation of ⟨exp⟩ grammar (Java)

The *Grammar* class can be extended, e.g. *ExpGrammar* is inherited by *WhileGrammar*

(the grammar that describes while programs). This modularization approach concedes further flexibility when converting between string and syntax tree. An instance of *Grammar* both is used by the lexer and the parser. The *Lexer* class splits a string into a stream of tokens and the *Parser* class transforms the tokens into a syntax tree. *Token* is an aggregate of *Terminal* and contains additional details like the position it occupies relating to the input string. That information can be used for error reporting when checking the syntax or for other output arrangement.

The inner workings of the lexer can be looked at in listing B . The method first removes comments and sanitizes the input from unnecessary line breaks. The current position is memorized and incorporated into the regular expression pattern. All of the terminals and rules are iterated over in order to find the longest match. Before doing that, the rules get sorted because the language for '`id`' is infact a superset of most keywords like '`IF`' or '`WHILE`'. In that case, the keywords should be prioritized. If no match is found after trying everything, an exception with the current position will be thrown. Otherwise, the longest match will generate a new instance of *Token* and the lexer appends it to the output list. The pointer advances and the process is repeated until it transcends the end of the input string.

Using the terminals and non terminals along with the information about the parser rules, a grammar can construct a predictive parser table. This table makes a direct assignment between a pair of *NonTerminal* and *Terminal* and *ParserRule* and is used by the parser to select the next rule. The parsing algorithm is depicted in listing B . First the terminator symbol ('$') is annexed to the token list and the iterator is pointed to the first token. Beginning with the designated starting rule of the grammar, the recursively invoked *getNode* method is called. It is supposed to create a single node of the syntax tree (paying attention to the current non terminal and token) but triggers the construction of all children nodes in one fell swoop. The *selectRule* method fetches the rule to pursue from the predictive parser table and will report an error if there is no entry. Then the symbols of the rule are gone over: If a symbol is a terminal, it will be compared against the current token, added as a child and the iterator will take a step to point to the next token. In case of the empty word, the child is a special ϵ node. Lastly, non terminals amount to a child as well but call the *getNode* method again to acquire their own respective children. Since the last case does not effectuate a change in the token iterator, the grammar is expected to indeed be a LL(1) grammar as a different scenario would be prone to induce an infinite loop.

5.2.1 First, follow

The algorithms for the concept of *First* and *Follow* were already outlined in chapter 3: *Introduction of language* . Specific *Java* implementations are provided in listing L2 and listing L3 . The predictive parser table is pieced together in listing A ;

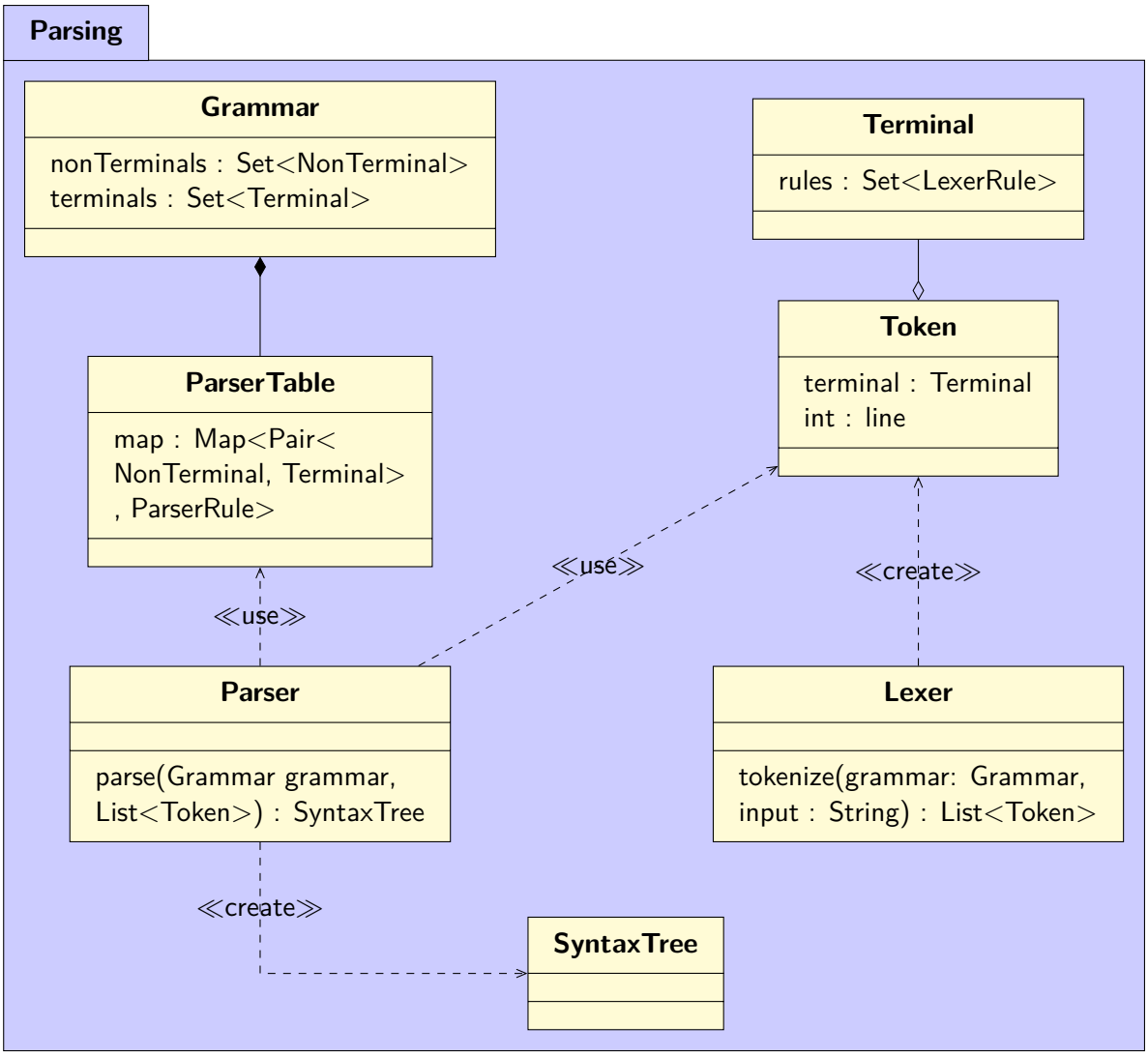

Figure 5.4: parser-related class diagram

```
1  public Set<Terminal> getFirst(List<Symbol> symbols, Set<NonTerminal> recursiveSet)
       {
2    if (symbols.contains(Terminal.EPSILON)) return new LinkedHashSet<>(Arrays.asList
       (new Terminal[]{Terminal.EPSILON}));
3
4    Set<Terminal> ret = new LinkedHashSet<>();
5
6    for (int i = 0; i < symbols.size(); i++) {
7      Symbol symbol = symbols.get(i);
8
9      if (symbol instanceof Terminal) {
10       ret.add((Terminal) symbol); break;
11     } else if (symbol instanceof NonTerminal) {
12       Set<Terminal> setSub = getFirst((NonTerminal) symbol, recursiveSet);
13
14       if (i < symbols.size() - 1 && setSub.contains(Terminal.EPSILON)) {
15         setSub.remove(Terminal.EPSILON); ret.addAll(setSub);
16       } else {
17         ret.addAll(setSub); break;
18       }
```

```
    }
  }

  return ret;
}

public Set<Terminal> getFirst(NonTerminal nonTerminal, Set<NonTerminal>
    recursiveSet) {
  Set<Terminal> ret = new LinkedHashSet<>();

  if (recursiveSet.contains(nonTerminal)) return ret;

  recursiveSet.add(nonTerminal);

  for (ParserRule p : nonTerminal.getRules()) {
    ret.addAll(getFirst(p.getSymbols(), recursiveSet));
  }

  return ret;
}

public Set<Terminal> getFirst(NonTerminal nonTerminal) {
  return getFirst(nonTerminal, new LinkedHashSet<>());
}
```

Listing L2: Implementation of First (Java)

```
public Set<Terminal> getFollow(NonTerminal nonTerminal, Grammar grammar, Set<
    NonTerminal> recursiveSet) {
  Set<Terminal> ret = new LinkedHashSet<>();

  if (recursiveSet.contains(nonTerminal)) return ret;

  recursiveSet.add(nonTerminal);

  if (grammar.getStartParserRule().equals(nonTerminal)) ret.add(Terminal.
      TERMINATOR);

  for (NonTerminal p : grammar.getNonTerminals()) {
    for (ParserRule rule : p.getRules()) {
      List<Symbol> symbols = rule.getSymbols();

      for (int i = 0; i < symbols.size(); i++) {
        Symbol symbol = rule.getSymbols().get(i);

        if (symbol.equals(nonTerminal)) {
          List<Symbol> restSymbols = (i < symbols.size() - 1) ? symbols.subList(i
              + 1, symbols.size()) : new ArrayList<>();

          if (restSymbols.isEmpty()) {
            ret.addAll(getFollow(p, grammar, recursiveSet));
          } else {
            Set<Terminal> subSet = getFirst(restSymbols, new LinkedHashSet<
                NonTerminal>());

            if (subSet.contains(Terminal.EPSILON)) {
              subSet.remove(Terminal.EPSILON);

              ret.addAll(subSet);

              ret.addAll(getFollow(p, grammar, recursiveSet));
```

```
            } else {
              ret.addAll(subSet);
            }
          }
        }
      }
    }

    return ret;
}

public Set<Terminal> getFollow(NonTerminal nonTerminal, Grammar grammar) {
    return getFollow(nonTerminal, grammar, new LinkedHashSet<NonTerminal>());
}
```

Listing L3: Implementation of Follow (Java)

```
public PredictiveParserTable(Grammar g) {
    Map<NonTerminal, Set<Terminal>> firstMap = new LinkedHashMap<>();
    Map<NonTerminal, Set<Terminal>> followMap = new LinkedHashMap<>();

    for (NonTerminal p : g.getNonTerminals()) {
      firstMap.put(p, getFirst(p));
      followMap.put(p, getFollow(p, g));
    }

    for (NonTerminal p : g.getNonTerminals()) {
      for (ParserRule r : p.getRules()) {
        List<Symbol> symbols = r.getSymbols();

        if (symbols.contains(Terminal.EPSILON))
          for (Terminal terminal : followMap.get(p)) {
            set(p, terminal, r);
          }
        else {
          Symbol symbol = r.getSymbols().get(0);

          if (symbol instanceof Terminal) {
            set(p, (Terminal) symbol, r);
          } else if (symbol instanceof NonTerminal) {
            for (Terminal terminal : firstMap.get((NonTerminal) symbol)) {
              set(p, terminal, r);
            }
          }
        }
      }
    }
}
```

Listing L4: Parser table (Java)

```
public LexerResult tokenize(String s) throws LexerException {
    s = removeComments(s);

    Vector<Terminal> terminals = new Vector<>(_grammar.getTerminals());

    terminals.sort(new Comparator<Terminal>() {
      private boolean isRegEx(Terminal terminal) {
        for (LexerRule rule : terminal.getRules()) if (rule.isRegEx()) return true;

```

```
      return false;
    }

    @Override
    public int compare(Terminal terminalA, Terminal terminalB) {
      if (terminalA.getRules().isEmpty() || terminalB.getRules().isEmpty()) return
          0;

      if (isRegEx(terminalA)) return 1;
      if (isRegEx(terminalB)) return -1;

      return 0;
    }
  });

  Terminal wsRule = new Terminal(new SymbolKey("WS"), true);

  wsRule.addRule(new LexerRule("\\s+", true));

  terminals.add(wsRule);

  int curPos = 0; Vector<Token> tokens = new Vector<>(); int x = 0; int y = 0;

  while (curPos < s.length()) {
    if ((s.length() - curPos >= System.lineSeparator().length()) && s.substring(
        curPos, curPos + System.lineSeparator().length()).equals(System.
        lineSeparator())) {
      curPos += System.lineSeparator().length(); x = 0; y++;

      continue;
    }

    int curLen = 0; LexerRule curRule = null; Terminal curTerminal = null;

    for (int i = 0; i < terminals.size(); i++) {
      Terminal terminal = terminals.get(i);

      for (LexerRule rule : terminal.getRules()) {
        String ruleS = (curPos > 0) ? String.format("^.{%d}(%s)", curPos, rule.
            getRegEx()) : String.format("^(%s)", rule.getRegEx());

          Pattern adjustedPattern = Pattern.compile(ruleS, Pattern.DOTALL);

        Matcher matcher = adjustedPattern.matcher(s);

        if (matcher.find() && (matcher.start(1) == curPos)) {
          int newLen = (matcher.end(1) - 1) - matcher.start(1) + 1;

            if (newLen > curLen) {
            curTerminal = terminal; curRule = rule; curLen = newLen;
          }
        }
      }
    }

    if (curRule == null) throw new LexerException(y, x, curPos, s);
    else {
      String text = s.substring(curPos, curPos + curLen);

      for (int i = 0; i < text.length();) {
        if ((text.length() - i >= System.lineSeparator().length()) && text.
```

```
            substring(i, i + System.lineSeparator().length()).equals(System.
            lineSeparator())) {
          i += System.lineSeparator().length();
        } else {
          i++;
        }
      }

      Token token = createToken(curTerminal, curRule, text, y, x, curPos);

      if (!token.getTerminal().isSkipped()) tokens.add(token);

      curPos += curLen;
      x += curLen;
    }
  }

  return new LexerResult(tokens);
}
```

Listing L5: Lexer (Java)

```
private ParserRule selectRule(NonTerminal nonTerminal, Token terminal) throws
    ParserException {
  try {
    ParserRule rule = _ruleMap.get(nonTerminal, terminal.getTerminal());

    if (rule == null) throw new Exception();

    return rule;
  } catch (Exception e) {
    if (terminal == null) throw new NoMoreTokensException(nonTerminal);
    else throw new NoRuleException(terminal, nonTerminal);
  }
}

private SyntaxTreeNode getNode(NonTerminal rule) throws ParserException {
  ParserRule nextRule = selectRule(rule, _token);

  SyntaxTreeNode thisNode = new SyntaxTreeNode(rule, nextRule);

  for (Symbol symbol : nextRule.getSymbols()) {
    if (symbol instanceof NonTerminal) {
      thisNode.addChild(getNode((NonTerminal) symbol));
    } else {
      if (symbol.equals(Terminal.EPSILON)) {
        thisNode.addChild(new SyntaxTreeNodeTerminal(null));

        continue;
      }

      if (_token == null) throw new NoMoreTokensException(rule, (Terminal) symbol)
          ;
      if (!_token.getTerminal().equals(symbol)) throw new WrongTokenException(
          _token, rule, symbol);

      thisNode.addChild(new SyntaxTreeNodeTerminal(_token));

      _token = _tokensItr.hasNext() ? _tokensItr.next() : null;
    }
  }
```

```
37
38    return thisNode;
39  }
40
41  public SyntaxTree parse(Vector<Token> tokens) throws ParserException {
42    _tokens = tokens;
43
44    if (_tokens.isEmpty()) throw new NoMoreTokensException(_grammar.
          getStartParserRule());
45
46    _tokens.add(Token.createTerminator(tokens));
47
48    _tokensItr = _tokens.iterator();
49
50    _token = _tokensItr.next();
51
52    SyntaxTreeNode root = getNode(_grammar.getStartParserRule());
53
54    if (!_token.getTerminal().equals(Terminal.TERMINATOR)) throw new
          SuperfluousTokenException(_token);
55
56    return new SyntaxTree(_grammar, root);
57  }
```

Listing L6: Parser (Java)

5.3 Hoare

5.4 Exception handling

5.5 GUI

5.5.1 Editor

5.5.2 Display of tokens/syntax tree

5.5.3 Syntax chart with Hoare decoration

Chapter 6

6 Fazit

6.1 Summerization

6.2 Remaining problems

6.3 Extendabilities

[Rav17]

Bibliography

[APdBO10] K. Apt, A. Pnueli, F.S. de Boer, and E.R. Olderog. *Verification of Sequential and Concurrent Programs*. Texts in Computer Science. Springer London, 2010. URL: https://books.google.de/books?id=9BGPVwLTkh4C.

[Gri82] David Gries. A note on a standard strategy for developing loop invariants and loops. *Science of Computer Programming*, 2(3):207 – 214, 1982. URL: http://www.sciencedirect.com/science/article/pii/0167642383900151, doi:http://dx.doi.org/10.1016/0167-6423(83)90015-1.

[Kam16] Anya Kamenetz. The president wants every student to learn computer science. how would that work? http://www.npr.org/sections/ed/2016/01/12/462698966/the-president-wants-every-student-to-learn-computer-science-how-would-that-work, 2016.

[Kle09] S. Kleuker. *Formale Modelle der Softwareentwicklung: Model-Checking, Verifikation, Analyse und Simulation*. Vieweg Studium. Vieweg+Teubner Verlag, 2009. URL: https://books.google.de/books?id=Gql3pv5V85cC.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society. URL: http://dx.doi.org/10.1109/SFCS.1977.32, doi:10.1109/SFCS.1977.32.

[Rav17] Ravindrababu Ravula. Compiler design online lectures. https://www.youtube.com/playlist?list=PLEbnTDJUr_IcPtUXFy2b1sGRPsLFMghhS, 2017. [Online; accessed 31-August-2017].

[Wik17a] Wikipedia. Fehlerquotient — wikipedia, die freie enzyklopädie. https://de.wikipedia.org/w/index.php?title=Fehlerquotient, 2017. [Online; accessed 24-August-2017].

[Wik17b] Wikipedia. Shape analysis (program analysis) — wikipedia, the free encyclopedia.

https://en.wikipedia.org/w/index.php?title=Shape_analysis_(program_analysis), 2017. [Online; accessed 24-August-2017].

Appendix A

A First, Follow, Parser Table listings

```java
public Set<Terminal> getFirst(List<Symbol> symbols, Set<NonTerminal> recursiveSet) {
  if (symbols.contains(Terminal.EPSILON)) return new LinkedHashSet<>(Arrays.asList(new Terminal[]{Terminal.EPSILON}));

  Set<Terminal> ret = new LinkedHashSet<>();

  for (int i = 0; i < symbols.size(); i++) {
    Symbol symbol = symbols.get(i);

    if (symbol instanceof Terminal) {
      ret.add((Terminal) symbol); break;
    } else if (symbol instanceof NonTerminal) {
      Set<Terminal> setSub = getFirst((NonTerminal) symbol, recursiveSet);

      if (i < symbols.size() - 1 && setSub.contains(Terminal.EPSILON)) {
        setSub.remove(Terminal.EPSILON); ret.addAll(setSub);
      } else {
        ret.addAll(setSub); break;
      }
    }
  }

  return ret;
}

public Set<Terminal> getFirst(NonTerminal nonTerminal, Set<NonTerminal> recursiveSet) {
  Set<Terminal> ret = new LinkedHashSet<>();

  if (recursiveSet.contains(nonTerminal)) return ret;

  recursiveSet.add(nonTerminal);

  for (ParserRule p : nonTerminal.getRules()) {
    ret.addAll(getFirst(p.getSymbols(), recursiveSet));
  }

  return ret;
}

public Set<Terminal> getFirst(NonTerminal nonTerminal) {
  return getFirst(nonTerminal, new LinkedHashSet<>());
}
```

```
1  public Set<Terminal> getFollow(NonTerminal nonTerminal, Grammar grammar, Set<
       NonTerminal> recursiveSet) {
2    Set<Terminal> ret = new LinkedHashSet<>();
3
4    if (recursiveSet.contains(nonTerminal)) return ret;
5
6    recursiveSet.add(nonTerminal);
7
8    if (grammar.getStartParserRule().equals(nonTerminal)) ret.add(Terminal.
         TERMINATOR);
9
10   for (NonTerminal p : grammar.getNonTerminals()) {
11     for (ParserRule rule : p.getRules()) {
12       List<Symbol> symbols = rule.getSymbols();
13
14       for (int i = 0; i < symbols.size(); i++) {
15         Symbol symbol = rule.getSymbols().get(i);
16
17         if (symbol.equals(nonTerminal)) {
18           List<Symbol> restSymbols = (i < symbols.size() - 1) ? symbols.subList(i
                 + 1, symbols.size()) : new ArrayList<>();
19
20           if (restSymbols.isEmpty()) {
21             ret.addAll(getFollow(p, grammar, recursiveSet));
22           } else {
23             Set<Terminal> subSet = getFirst(restSymbols, new LinkedHashSet<
                   NonTerminal>());
24
25             if (subSet.contains(Terminal.EPSILON)) {
26               subSet.remove(Terminal.EPSILON);
27
28               ret.addAll(subSet);
29
30               ret.addAll(getFollow(p, grammar, recursiveSet));
31             } else {
32               ret.addAll(subSet);
33             }
34           }
35         }
36       }
37     }
38   }
39
40   return ret;
41 }
42
43 public Set<Terminal> getFollow(NonTerminal nonTerminal, Grammar grammar) {
44   return getFollow(nonTerminal, grammar, new LinkedHashSet<NonTerminal>());
45 }
```

```
public PredictiveParserTable(Grammar g) {
  Map<NonTerminal, Set<Terminal>> firstMap = new LinkedHashMap<>();
  Map<NonTerminal, Set<Terminal>> followMap = new LinkedHashMap<>();

  for (NonTerminal p : g.getNonTerminals()) {
    firstMap.put(p, getFirst(p));
    followMap.put(p, getFollow(p, g));
  }

  for (NonTerminal p : g.getNonTerminals()) {
    for (ParserRule r : p.getRules()) {
      List<Symbol> symbols = r.getSymbols();

      if (symbols.contains(Terminal.EPSILON))
        for (Terminal terminal : followMap.get(p)) {
          set(p, terminal, r);
        }
      else {
        Symbol symbol = r.getSymbols().get(0);

        if (symbol instanceof Terminal) {
          set(p, (Terminal) symbol, r);
        } else if (symbol instanceof NonTerminal) {
          for (Terminal terminal : firstMap.get((NonTerminal) symbol)) {
            set(p, terminal, r);
          }
        }
      }
    }
  }
}
```

B Appendix B
Lexer, Parser listings

```
1  public LexerResult tokenize(String s) throws LexerException {
2    s = removeComments(s);
3
4    Vector<Terminal> terminals = new Vector<>(_grammar.getTerminals());
5
6    terminals.sort(new Comparator<Terminal>() {
7      private boolean isRegEx(Terminal terminal) {
8        for (LexerRule rule : terminal.getRules()) if (rule.isRegEx()) return true;
9
10       return false;
11     }
12
13     @Override
14     public int compare(Terminal terminalA, Terminal terminalB) {
15       if (terminalA.getRules().isEmpty() || terminalB.getRules().isEmpty()) return
             0;
16
17       if (isRegEx(terminalA)) return 1;
18       if (isRegEx(terminalB)) return -1;
19
20       return 0;
21     }
22   });
23
24   Terminal wsRule = new Terminal(new SymbolKey("WS"), true);
25
26   wsRule.addRule(new LexerRule("\\s+", true));
27
28   terminals.add(wsRule);
29
30   int curPos = 0; Vector<Token> tokens = new Vector<>(); int x = 0; int y = 0;
31
32   while (curPos < s.length()) {
33     if ((s.length() - curPos >= System.lineSeparator().length()) && s.substring(
         curPos, curPos + System.lineSeparator().length()).equals(System.
         lineSeparator())) {
34       curPos += System.lineSeparator().length(); x = 0; y++;
35
36       continue;
37     }
38
39     int curLen = 0; LexerRule curRule = null; Terminal curTerminal = null;
40
41     for (int i = 0; i < terminals.size(); i++) {
42       Terminal terminal = terminals.get(i);
43
44       for (LexerRule rule : terminal.getRules()) {
45         String ruleS = (curPos > 0) ? String.format("^.{%d}(%s)", curPos, rule.
               getRegEx()) : String.format("^(%s)", rule.getRegEx());
46
```

```
          Pattern adjustedPattern = Pattern.compile(ruleS, Pattern.DOTALL);

        Matcher matcher = adjustedPattern.matcher(s);

        if (matcher.find() && (matcher.start(1) == curPos)) {
          int newLen = (matcher.end(1) - 1) - matcher.start(1) + 1;

            if (newLen > curLen) {
            curTerminal = terminal; curRule = rule; curLen = newLen;
          }
        }
      }
    }

    if (curRule == null) throw new LexerException(y, x, curPos, s);
    else {
      String text = s.substring(curPos, curPos + curLen);

      for (int i = 0; i < text.length();) {
        if ((text.length() - i >= System.lineSeparator().length()) && text.
            substring(i, i + System.lineSeparator().length()).equals(System.
            lineSeparator())) {
          i += System.lineSeparator().length();
        } else {
          i++;
        }
      }

      Token token = createToken(curTerminal, curRule, text, y, x, curPos);

      if (!token.getTerminal().isSkipped()) tokens.add(token);

      curPos += curLen;
      x += curLen;
    }
  }

  return new LexerResult(tokens);
}
```

```
private ParserRule selectRule(NonTerminal nonTerminal, Token terminal) throws
    ParserException {
  try {
    ParserRule rule = _ruleMap.get(nonTerminal, terminal.getTerminal());

    if (rule == null) throw new Exception();

    return rule;
  } catch (Exception e) {
    if (terminal == null) throw new NoMoreTokensException(nonTerminal);
    else throw new NoRuleException(terminal, nonTerminal);
  }
}

private SyntaxTreeNode getNode(NonTerminal rule) throws ParserException {
  ParserRule nextRule = selectRule(rule, _token);

  SyntaxTreeNode thisNode = new SyntaxTreeNode(rule, nextRule);

  for (Symbol symbol : nextRule.getSymbols()) {
    if (symbol instanceof NonTerminal) {
```

```
21        thisNode.addChild(getNode((NonTerminal) symbol));
22      } else {
23        if (symbol.equals(Terminal.EPSILON)) {
24          thisNode.addChild(new SyntaxTreeNodeTerminal(null));
25
26          continue;
27        }
28
29        if (_token == null) throw new NoMoreTokensException(rule, (Terminal) symbol)
              ;
30        if (!_token.getTerminal().equals(symbol)) throw new WrongTokenException(
              _token, rule, symbol);
31
32        thisNode.addChild(new SyntaxTreeNodeTerminal(_token));
33
34        _token = _tokensItr.hasNext() ? _tokensItr.next() : null;
35      }
36    }
37
38    return thisNode;
39  }
40
41  public SyntaxTree parse(Vector<Token> tokens) throws ParserException {
42    _tokens = tokens;
43
44    if (_tokens.isEmpty()) throw new NoMoreTokensException(_grammar.
          getStartParserRule());
45
46    _tokens.add(Token.createTerminator(tokens));
47
48    _tokensItr = _tokens.iterator();
49
50    _token = _tokensItr.next();
51
52    SyntaxTreeNode root = getNode(_grammar.getStartParserRule());
53
54    if (!_token.getTerminal().equals(Terminal.TERMINATOR)) throw new
          SuperfluousTokenException(_token);
55
56    return new SyntaxTree(_grammar, root);
57  }
```

Appendix C

Hoare listing

```
1  private class HoareNode {
2    private SyntaxTreeNode _actualNode;
3
4    private Vector<HoareNode> _children = new Vector<>();
5
6    public Vector<HoareNode> getChildren() {
7      return _children;
8    }
9
10   public void addChild(HoareNode child) {
11     _children.add(child);
12   }
13
14   public HoareNode(SyntaxTreeNode actualNode) {
15     _actualNode = actualNode;
16   }
17 }
18
19 private Vector<HoareNode> collectChildren(SyntaxTreeNode node) {
20   Vector<HoareNode> ret = new Vector<>();
21
22   for (SyntaxTreeNode child : node.getChildren()) {
23     Vector<HoareNode> hoareChildren = collectChildren(child);
24
25     ret.addAll(hoareChildren);
26   }
27
28   if ((node.getSymbol() != null) && node.getSymbol().equals(_grammar.
       nonTerminal_hoare_block)) {
29     HoareNode selfNode = new HoareNode(node);
30
31     for (HoareNode child : ret) {
32       selfNode.addChild(child);
33     }
34
35     ret.clear();
36
37     ret.add(selfNode);
38   }
39
40   return ret;
41 }
42
43 private interface ExecInterface {
44   public void finished() throws HoareException, LexerException, IOException,
       ParserException;
45 }
46
47 public static class Executer {
48   private HoareNode _node;
```

```
49  private int _nestDepth;
50  private HoareWhileGrammar _grammar;
51  private ObservableMap<SyntaxTreeNode, HoareCond> _preCondMap;
52  private ObservableMap<SyntaxTreeNode, HoareCond> _postCondMap;
53  private ExecInterface _callback;
54
55  private Vector<Executer> _execChain = new Vector<>();
56  private Iterator<Executer> _execChainIt;
57
58  public interface ImplicationInterface {
59    public void result(boolean yes) throws HoareException, LexerException,
      IOException, ParserException;
60  }
61
62  public interface InvariantInterface {
63    public void result(HoareCond invariant) throws HoareException, LexerException,
       IOException, ParserException;
64  }
65
66  private boolean check(HoareCond a) throws ScriptException {
67    /*ScriptEngineManager manager = new ScriptEngineManager();
68
69    ScriptEngine engine = manager.getEngineByName("JavaScript");
70
71    System.out.println(engine.eval(a.toString()));
72
73    return true;*/
74    return true;
75  }
76
77  private boolean implicates(HoareCond a, HoareCond b, ImplicationInterface
      callback) throws ScriptException, IOException, HoareException, LexerException,
       ParserException {
78    System.out.println("try␣implication␣" + a + "->" + b);
79
80    //TODO implicit check
81    boolean checkSuccess = false;
82    boolean checkResult = false;
83
84    if (checkSuccess) {
85      callback.result(checkResult);
86    } else {
87      new ImplicationDialog(a, b, callback, false).show();
88    }
89
90    return !check(a) || check(b);
91  }
92
93  private int _wlp_nestDepth = 0;
94  private int _wlp_printDepth = 0;
95
96  private void println_begin() {
97    _wlp_printDepth++;
98  }
99
100 private void println(String s) {
101   System.out.println(StringUtil.repeat("\t", _wlp_printDepth - 1) + s);
102 }
103
104 private void println_end() {
105   _wlp_printDepth--;
```

```
106   }
107
108   private interface wlp_callback {
109     public void result(HoareCond cond) throws IOException, HoareException,
      LexerException, ParserException;
110   }
111
112   private void wlp_assign(HoareCond postCond, String var, SyntaxTreeNode valNode,
      wlp_callback callback) throws IOException, HoareException {
113     println_begin();
114
115     HoareCond preCond = postCond.copy();
116
117     try {
118       Exp val = Exp.fromString(valNode.synthesize());
119
120       preCond.replace(_grammar.TERMINAL_ID, var, val.getBaseEx());
121
122       println("apply␣assignment␣rule:");
123       println("\t" + postCond.toStringEx(var + ":=" + valNode.synthesize()) + "␣"
    + var + "=" + valNode.synthesize() + "␣" + postCond.toStringEx());
124       println("\t->" + preCond.toStringEx() + "␣" + var + "=" + valNode.synthesize
    () + "␣" + postCond.toStringEx());
125
126       println_end();
127
128       callback.result(preCond);
129     } catch (ParserException | LexerException e) {
130       throw new RuntimeException(e);
131     }
132   }
133
134   private void wlp_composite(HoareCond postCond, SyntaxTreeNode first,
      SyntaxTreeNode second, wlp_callback callback) throws HoareException,
      IOException, LexerException, ParserException {
135     println_begin();
136
137     println("applying␣composition␣rule...");
138
139     wlp(second, postCond, new wlp_callback() {
140       @Override
141       public void result(HoareCond midCond) throws IOException, HoareException,
      LexerException, ParserException {
142         wlp(first, midCond, new wlp_callback() {
143           @Override
144           public void result(HoareCond preCond) throws IOException, HoareException
    , LexerException, ParserException {
145             String firstS = first.synthesize().replaceAll("\n", "");
146             String secondS = second.synthesize().replaceAll("\n", "");
147
148             //System.out.println("{" + postCondition + "}" + " -> " + "{" +
      midCondition + "}" + " -> " + "{" + ret + "}");
149             println("apply␣composition␣rule:");
150             println("\t" + preCond.toStringEx() + "␣" + firstS + "␣" + midCond.
      toStringEx() + ",␣" + midCond.toStringEx() + "␣" + secondS + "␣" + postCond.
      toStringEx());
151             println("\t" + "->");
152             println("\t" + preCond.toStringEx() + "␣" + firstS + ";␣" + secondS +
      "␣" + postCond.toStringEx());
153
154             println_end();
```

```
155
156             callback.result(preCond);
157           }
158         });
159       }
160     });
161   }
162
163   private void wlp_alt(HoareCond postCond, HoareCondBoolExpr altCond,
        SyntaxTreeNode first, SyntaxTreeNode second, wlp_callback callback) throws
        IOException, HoareException, LexerException, ParserException {
164     println_begin();
165
166     HoareCond preCond = postCond.copy();
167
168     String firstS = first.synthesize().replaceAll("\n", "");
169     String secondS = second.synthesize().replaceAll("\n", "");
170
171     //TODO
172     println("apply␣alternative␣rule:");
173     println("\t" + new HoareCondAnd(preCond, altCond).toStringEx() + "␣" + firstS
        + "␣" + postCond.toStringEx() + ",␣" + new HoareCondAnd(preCond, new
        HoareCondNeg(altCond)).toStringEx() + "␣" + secondS + "␣" + postCond.
        toStringEx());
174     println("\t" + "->");
175     println("\t" + preCond.toStringEx() + "␣if␣" + "(" + altCond + ")" + "{" +
        firstS + "}" + "␣else␣" + "{" + secondS + "}" + postCond.toStringEx());
176
177     println_end();
178
179     callback.result(preCond);
180   }
181
182   private void wlp_loop_acceptInvariant(HoareCond invariant, SyntaxTreeNode
        loopNode, wlp_callback callback) throws IOException, HoareException,
        LexerException, ParserException {
183     HoareCond loopCond = new HoareCondBoolExpr(loopNode.findChild(_grammar.
        NON_TERMINAL_BOOL_EXP));
184     SyntaxTreeNode body = loopNode.findChild(_grammar.NON_TERMINAL_PROG);
185
186     println("accept␣invariant␣" + invariant);
187
188     HoareCond preCond = new HoareCondAnd(invariant);
189
190     if (preCond==null) throw new RuntimeException("preCond␣null");
191     if (loopCond==null) throw new RuntimeException("loopCond␣null");
192
193     String bodyS = body.synthesize().replaceAll("\n", "");
194
195     println("apply␣loop␣rule:");
196     println("\t" + new HoareCondAnd(preCond, loopCond).toStringEx() + "␣" + bodyS
        + "␣" + preCond.toStringEx());
197     println("\t" + "->");
198     println("\t" + new HoareCondAnd(preCond).toStringEx() + "␣while␣" + "(" +
        loopCond + ")" + "{" + bodyS + "}" + "␣" + new HoareCondAnd(preCond, new
        HoareCondNeg(loopCond)).toStringEx());
199
200     println_end();
201
202     callback.result(preCond);
203   }
```

```
204
205   private void wlp_loop_tryInvariant(SyntaxTreeNode loopNode, HoareCond postCond,
        HoareCond invariantPost, wlp_callback callback) throws HoareException,
        IOException, LexerException, ParserException {
206     //TODO: auto-generate invariants
207
208     if (invariantPost == null) {
209       println("failed␣to␣guess␣invariant:␣ask␣user");
210
211       InvariantDialog diag = new InvariantDialog(_grammar, loopNode, postCond, new
         InvariantInterface() {
212         @Override
213         public void result(HoareCond invariant) throws HoareException, IOException
        , LexerException, ParserException {
214           if (invariant != null) {
215             wlp_loop_acceptInvariant(invariant, loopNode, callback);
216           } else {
217             throw new HoareException("aborted");
218           }
219         }
220       });
221
222       diag.show();
223     } else {
224       println("try␣invariant:␣" + invariantPost);
225
226       wlp(loopNode.findChild(_grammar.NON_TERMINAL_PROG), invariantPost, new
        wlp_callback() {
227         @Override
228         public void result(HoareCond invariantPre) throws HoareException,
        LexerException, IOException, ParserException {
229           println("tried␣invariant␣" + invariantPost + "␣resulted␣in␣" +
        invariantPre);
230
231           ImplicationDialog diag = new ImplicationDialog(invariantPre,
        invariantPost, new ImplicationInterface() {
232             @Override
233             public void result(boolean yes) throws HoareException, LexerException,
         IOException, ParserException {
234               if (yes) {
235                 wlp_loop_acceptInvariant(invariantPre, loopNode, callback);
236               } else {
237                 wlp_loop_tryInvariant(loopNode, postCond, null, callback);
238               }
239             }
240           }, true);
241
242           diag.show();
243         }
244       });
245     }
246   }
247
248   private void wlp_loop(HoareCond postCond, SyntaxTreeNode loopNode, wlp_callback
        callback) throws HoareException, IOException {
249     println_begin();
250
251     try {
252       println("applying␣loop␣rule...␣needs␣invariant");
253
254       //HoareCondition invariantPost = HoareCondition.fromString("erg==2^(y-x)");
```

```
255       HoareCond invariantPost = null;//HoareCond.fromString("y==z!");
256
257       wlp_loop_tryInvariant(loopNode, postCond, invariantPost, callback);
258     } catch (LexerException | ParserException e) {
259       throw new HoareException(e.getMessage());
260     }
261   }
262
263   private void wlp_consequence_pre(HoareCond origPreCond, HoareCond origPostCond,
      SyntaxTreeNode body, HoareCond newPreCond, HoareCond newPostCond, wlp_callback
       callback) throws IOException, HoareException, LexerException, ParserException
       {
264     println_begin();
265
266     String bodyS = body.synthesize().replaceAll("\n", "");
267
268     System.out.println("apply␣consequence␣rule");
269     System.out.println("\t" + newPreCond + "->" + origPreCond + ",␣" +
      origPostCond.toStringEx() + "␣" + bodyS + "␣" + origPostCond.toStringEx() + ",
      ␣" + origPostCond + "->" + newPostCond);
270     System.out.println("\t" + "->");
271     System.out.println("\t" + newPreCond.toStringEx() + "␣" + bodyS + "␣" +
      newPostCond.toStringEx());
272
273     println_end();
274
275     //TODO: for post as well, merged?
276     callback.result(newPreCond);
277   }
278
279   private void wlp(SyntaxTreeNode node, HoareCond postCondV, wlp_callback callback
      ) throws HoareException, IOException, LexerException, ParserException {
280     _wlp_nestDepth++;
281
282     final HoareCond postCond = postCondV.copy();
283
284     _postCondMap.put(node, postCond);
285
286     //System.out.println(StringUtil.repeat("\t", _wlp_nestDepth) + "postcond " +
      node);
287
288     wlp_callback retCallback = new wlp_callback() {
289       @Override
290       public void result(HoareCond cond) throws IOException, HoareException,
      LexerException, ParserException {
291         _preCondMap.put(node, cond);
292         _wlp_nestDepth--;
293
294         callback.result(cond);
295       }
296     };
297
298     if (node.getSymbol().equals(_grammar.NON_TERMINAL_PROG)) {
299       SyntaxTreeNode firstChild = node.getChildren().firstElement();
300       SyntaxTreeNode lastChild = node.getChildren().lastElement();
301
302       if (lastChild.findChild(_grammar.NON_TERMINAL_PROG) != null) {
303         wlp_composite(postCond, firstChild, lastChild.findChild(_grammar.
      NON_TERMINAL_PROG), retCallback);
304       } else {
305         wlp(firstChild, postCond, retCallback);
```

```
306       }
307     } else if (node.getSymbol().equals(_grammar.NON_TERMINAL_PROG_)) {
308       if (node.getSubRule().equals(_grammar.RULE_PROG__PROG))
309         wlp(node.findChild(_grammar.NON_TERMINAL_PROG), postCond, retCallback);
310       else
311         retCallback.result(postCond);
312     } else if (node.getSymbol().equals(_grammar.NON_TERMINAL_SKIP))
313       retCallback.result(postCond);
314     else if (node.getSymbol().equals(_grammar.NON_TERMINAL_ASSIGN)) {
315       SyntaxTreeNode idNode = node.findChild(_grammar.TERMINAL_ID);
316
317       SyntaxTreeNode expNode = node.findChild(_grammar.NON_TERMINAL_EXP);
318
319       String var = idNode.synthesize();
320       SyntaxTreeNode exp = expNode;
321
322       wlp_assign(postCond, var, exp, retCallback);
323     } else if (node.getSymbol().equals(_grammar.NON_TERMINAL_SELECTION)) {
324       if (node.getSubRule().equals(_grammar.RULE_SELECTION)) {
325         SyntaxTreeNode selectionElseRule = node.findChild(_grammar.
    NON_TERMINAL_PROG);
326
327         if (selectionElseRule.getSubRule().equals(Terminal.EPSILON)) {
328           wlp(node.findChild(_grammar.NON_TERMINAL_PROG), postCond, new
    wlp_callback() {
329             @Override
330             public void result(HoareCond thenCond) throws IOException,
    HoareException, LexerException, ParserException {
331               HoareCond elseCond = postCond;
332
333               retCallback.result(new HoareCondOr(thenCond, elseCond));
334             }
335           });
336         } else if (selectionElseRule.getSubRule().equals(_grammar.
    RULE_SELECTION_ELSE)) {
337           wlp(node.findChild(_grammar.NON_TERMINAL_PROG), postCond, new
    wlp_callback() {
338             @Override
339             public void result(HoareCond thenCond) throws IOException,
    HoareException, LexerException, ParserException {
340               wlp(node.findChild(_grammar.NON_TERMINAL_PROG, 2), postCond, new
    wlp_callback() {
341                 @Override
342                 public void result(HoareCond elseCond) throws IOException,
    HoareException, LexerException, ParserException {
343                   retCallback.result(new HoareCondOr(thenCond, elseCond));
344                 }
345               });
346             }
347           });
348         }
349       }
350     } else if (node.getSymbol().equals(_grammar.NON_TERMINAL_WHILE)) {
351       wlp_loop(postCond, node, retCallback);
352     } else if (node.getSymbol().equals(_grammar.nonTerminal_hoare_block)) {
353       wlp(node.findChild(_grammar.NON_TERMINAL_PROG), postCond, retCallback);
354     } else {
355       throw new HoareException("no␣wlp␣for␣" + node + "␣with␣rule␣" + node.
    getSubRule());
356     }
357   }
```

```
358
359   public void exec() throws IOException, HoareException, LexerException,
        ParserException {
360     SyntaxTreeNode preNode = _node._actualNode.findChild(_grammar.
      nonTerminal_hoare_pre);
361     SyntaxTreeNode postNode = _node._actualNode.findChild(_grammar.
      nonTerminal_hoare_post);
362
363     HoareCond preCondition = HoareCond.fromString(preNode.findChild(_grammar.
      NON_TERMINAL_BOOL_EXP).synthesize());
364     HoareCond postCondition = HoareCond.fromString(postNode.findChild(_grammar.
      NON_TERMINAL_BOOL_EXP).synthesize());
365
366     System.err.println(StringUtil.repeat("\t", _nestDepth) + "checking␣" +
      preCondition + "->" + postCondition + "␣at␣" + _node);
367
368     _wlp_nestDepth = 0;
369     _wlp_printDepth = 0;
370
371     wlp(_node._actualNode, postCondition, new wlp_callback() {
372       @Override
373       public void result(HoareCond finalPreCondition) throws IOException,
      HoareException, LexerException, ParserException {
374         System.out.println("final␣preCondition:␣" + finalPreCondition);
375
376         try {
377           implicates(preCondition, finalPreCondition, new ImplicationInterface() {
378             @Override
379             public void result(boolean yes) throws HoareException, LexerException,
       IOException, ParserException {
380               if (yes) {
381                 System.out.println(preCondition + "->" + postCondition + "␣holds␣
      true␣(wlp:␣" + finalPreCondition + ")");
382               } else {
383                 System.out.println(preCondition + "->" + postCondition + "␣failed␣
      (wlp:␣" + finalPreCondition + ")");
384               }
385
386               _callback.finished();
387             }
388           });
389         } catch (ScriptException e) {
390           e.printStackTrace();
391         }
392       }
393     });
394   }
395
396   public void start() throws IOException, HoareException, LexerException,
        ParserException {
397     _execChainIt.next().exec();
398   }
399
400   public Executer(HoareNode node, int nestDepth, HoareWhileGrammar grammar,
        ObservableMap<SyntaxTreeNode, HoareCond> preCondMap, ObservableMap<
        SyntaxTreeNode, HoareCond> postCondMap, ExecInterface callback) throws
        IOException, HoareException, NoRuleException, LexerException {
401     _node = node;
402     _nestDepth = nestDepth;
403     _grammar = grammar;
404     _preCondMap = preCondMap;
```

```
405     _postCondMap = postCondMap;
406     _callback = callback;
407 
408     ExecInterface childCallback = new ExecInterface() {
409       @Override
410       public void finished() throws HoareException, LexerException, IOException,
    ParserException {
411         Executer next = _execChainIt.next();
412 
413         next.exec();
414       }
415     };
416 
417     for (HoareNode child : node.getChildren()) {
418       _execChain.add(new Executer(child, nestDepth + 1, _grammar, preCondMap,
    postCondMap, childCallback));
419     }
420 
421     _execChain.add(this);
422 
423     _execChainIt = _execChain.iterator();
424   }
425 }
426 
427 private Vector<Executer> _execChain;
428 private Iterator<Executer> _execChainIt;
429 
430 public void exec() throws HoareException, LexerException, IOException,
    ParserException {
431   System.err.println("hoaring...");
432 
433   Vector<HoareNode> children = collectChildren(_tree.getRoot());
434 
435   if (children.isEmpty()) {
436     System.err.println("no␣hoareBlocks");
437   } else {
438     _execChain = new Vector<>();
439 
440     for (HoareNode child : children) {
441       if (children.lastElement().equals(child)) {
442         _execChain.add(new Executer(child, 0, _grammar, _preCondMap, _postCondMap,
     new ExecInterface() {
443           @Override
444           public void finished() throws HoareException, NoRuleException,
    LexerException, IOException {
445             System.err.println("hoaring␣finished");
446           }
447         }));
448       } else {
449         _execChain.add(new Executer(child, 0, _grammar, _preCondMap, _postCondMap,
     new ExecInterface() {
450           @Override
451           public void finished() throws IOException, HoareException,
    LexerException, ParserException {
452             _execChainIt.next().exec();
453           }
454         }));
455       }
456     }
457 
458     Iterator<Executer> execChainIt = _execChain.iterator();
```

```
459
460       execChainIt.next().exec();
461     }
462 }
```

Appendix D

D Grammmar for while programs

⟨*num*⟩	::=	[1-9][0-9]*
	|	0
⟨*id*⟩	::=	[a-zA-Z][a-zA-Z0-9]*
⟨*exp*⟩	::=	⟨*factor*⟩ ⟨*exp'*⟩
⟨*exp'*⟩	::=	'+' ⟨*factor*⟩ ⟨*exp'*⟩
	|	'−' ⟨*factor*⟩ ⟨*exp'*⟩
	|	ϵ
⟨*factor*⟩	::=	⟨*pow*⟩ ⟨*factor'*⟩
⟨*factor'*⟩	::=	'*' ⟨*pow*⟩ ⟨*factor'*⟩
	|	'/' ⟨*pow*⟩ ⟨*factor'*⟩
	|	ϵ
⟨*pow*⟩	::=	⟨*factorial*⟩ ⟨*pow'*⟩
⟨*pow'*⟩	::=	'^' ⟨*pow*⟩
	|	ϵ
⟨*factorial*⟩	::=	⟨*exp_elem*⟩ ⟨*factorial'*⟩
⟨*factorial'*⟩	::=	'!'
	|	ϵ
⟨*exp_elem*⟩	::=	'id'
	|	'num'
	|	'(' ⟨*exp*⟩ ')'
⟨*bool_exp*⟩	::=	⟨*bool_or*⟩
⟨*bool_or*⟩	::=	⟨*bool_and*⟩ ⟨*bool_or'*⟩
⟨*bool_or'*⟩	::=	'||' ⟨*bool_and*⟩ ⟨*bool_or'*⟩
	|	ϵ
⟨*bool_and*⟩	::=	⟨*bool_neg*⟩ ⟨*bool_and'*⟩
⟨*bool_and'*⟩	::=	'&&' ⟨*bool_neg*⟩ ⟨*bool_and'*⟩
	|	ϵ

⟨*bool_neg*⟩ ::= ⟨*bool_elem*⟩
| '~' ⟨*bool_elem*⟩

⟨*bool_elem*⟩ ::= ⟨*exp*⟩ '<' ⟨*exp*⟩
| 'true'
| '[' ⟨*bool_exp*⟩ ']'

⟨**prog*⟩ ::= ⟨*skip*⟩ ⟨*prog'*⟩
| ⟨*assign*⟩ ⟨*prog'*⟩
| ⟨*selection*⟩ ⟨*prog'*⟩
| ⟨*while*⟩ ⟨*prog'*⟩

⟨*prog'*⟩ ::= ';' ⟨*prog*⟩
| ϵ

⟨*skip*⟩ ::= 'SKIP'

⟨*assign*⟩ ::= 'id' '=' ⟨*exp*⟩

⟨*selection*⟩ ::= 'IF' ⟨*bool_exp*⟩ 'THEN' ⟨*prog*⟩ ⟨*selection_else*⟩ 'FI'

⟨*selection_else*⟩ ::= 'ELSE' ⟨*prog*⟩
| ϵ

⟨*while*⟩ ::= 'WHILE' ⟨*bool_exp*⟩ 'DO' ⟨*prog*⟩ 'OD'

Appendix E

Grammar for Hoare-decorated while programs

⟨*num*⟩	::=	[1-9][0-9]*
	|	0
⟨*id*⟩	::=	[a-zA-Z][a-zA-Z0-9]*
⟨*exp*⟩	::=	⟨*factor*⟩ ⟨*exp'*⟩
⟨*exp'*⟩	::=	'+' ⟨*factor*⟩ ⟨*exp'*⟩
	|	'−' ⟨*factor*⟩ ⟨*exp'*⟩
	|	ϵ
⟨*factor*⟩	::=	⟨*pow*⟩ ⟨*factor'*⟩
⟨*factor'*⟩	::=	'*' ⟨*pow*⟩ ⟨*factor'*⟩
	|	'/' ⟨*pow*⟩ ⟨*factor'*⟩
	|	ϵ
⟨*pow*⟩	::=	⟨*factorial*⟩ ⟨*pow'*⟩
⟨*pow'*⟩	::=	'^' ⟨*pow*⟩
	|	ϵ
⟨*factorial*⟩	::=	⟨*exp_elem*⟩ ⟨*factorial'*⟩
⟨*factorial'*⟩	::=	'!'
	|	ϵ
⟨*exp_elem*⟩	::=	'id'
	|	'num'
	|	'(' ⟨*exp*⟩ ')'
⟨*bool_exp*⟩	::=	⟨*bool_or*⟩
⟨*bool_or*⟩	::=	⟨*bool_and*⟩ ⟨*bool_or'*⟩
⟨*bool_or'*⟩	::=	'||' ⟨*bool_and*⟩ ⟨*bool_or'*⟩
	|	ϵ
⟨*bool_and*⟩	::=	⟨*bool_neg*⟩ ⟨*bool_and'*⟩

⟨*bool_and'*⟩ ::= '&&' ⟨*bool_neg*⟩ ⟨*bool_and'*⟩
| ϵ

⟨*bool_neg*⟩ ::= ⟨*bool_elem*⟩
| '~' ⟨*bool_elem*⟩

⟨*bool_elem*⟩ ::= ⟨*exp*⟩ '<' ⟨*exp*⟩
| 'true'
| '[' ⟨*bool_exp*⟩ ']'

⟨**prog*⟩ ::= ⟨*skip*⟩ ⟨*prog'*⟩
| ⟨*assign*⟩ ⟨*prog'*⟩
| ⟨*selection*⟩ ⟨*prog'*⟩
| ⟨*while*⟩ ⟨*prog'*⟩
| ⟨*hoare_block*⟩ ⟨*prog'*⟩

⟨*prog'*⟩ ::= ';' ⟨*prog*⟩
| ϵ

⟨*skip*⟩ ::= 'SKIP'

⟨*assign*⟩ ::= 'id' '=' ⟨*exp*⟩

⟨*selection*⟩ ::= 'IF' ⟨*bool_exp*⟩ 'THEN' ⟨*prog*⟩ ⟨*selection_else*⟩ 'FI'

⟨*selection_else*⟩ ::= 'ELSE' ⟨*prog*⟩
| ϵ

⟨*while*⟩ ::= 'WHILE' ⟨*bool_exp*⟩ 'DO' ⟨*prog*⟩ 'OD'

⟨*hoare_exp*⟩ ::= '{' ⟨*bool_exp*⟩ '}'

⟨*hoare_pre*⟩ ::= 'PRE' ⟨*hoare_exp*⟩

⟨*hoare_post*⟩ ::= 'POST' ⟨*hoare_exp*⟩

⟨*hoare_block*⟩ ::= ⟨*hoare_pre*⟩ ⟨*prog*⟩ ⟨*hoare_post*⟩

Declaration of Originality

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any sources other than those cited in the text and acknowledgements. This applies also to all graphics, drawings, maps and images included in the thesis.

Merseburg, October 22, 2017

.....	.....
Place and date	*Signature*

Dedication and Acknowledgements