Verification condition generator

Third Year Project

Burhanuddin Salim

February 2021

Contents

Abstract

We develop a program which mechanizes program verification by using hoare logic to produce verification conditions and then using compiler theory to parse and transform the produced verification conditions to be expressed in SMT-LIB. This enables us to leverage SMT-LIB compliant solvers to verify said verification conditions and in case the verification conditions fail we can use those solvers to obtain examples (models) that fail the conditions. The solver that is of focus in this project is Z3 and the language in which the programs will be written is Typescript. The program focuses on a subset of the typescript language which is strong enough to express most programs, namely: declarations, conditionals, arrays, loops, and a subset of function calls. Additionally, this program can also be used as an aid in Hoare logic teaching to annotate existing programs.

Acknowledgements

I would like to acknowledge the efforts of my supervisor, Giles Reger, in guiding and shaping this project and his efforts in the first year module Fundamentals of computation which were the foundations of this project.

Chapter 1

Introduction

1.1 Project Description

This project is about establishing axiomatic correctness of programs. The idea is to take a program annotated with pre/post conditions and loop invariants and to generate verification conditions whose validity indicate the correctness of the annotated program. These verification conditions should then be passed to an SMT solver to establish correctness or otherwise.

1.2 Requirements and Specification

The points below list a more concrete and testable list of requirements that will be assessed to measure the success of this project.

- Program takes in typescript source code and outputs an SMT-LIB file that encodes the code's verification conditions
- Program cna be triggered by a command line interface with optional and positional arguments that enable certain features of the program
- Program can output an annotated version of the source code that prefixes every statement with a weakest precondition

1.3 Motivation

A large part of the motivation for this project comes from the first year module Fundamentals of Computation. The module introduced us to the subject of Hoare logic[1] and Hoare triples.

The process for validating these Hoare triples often required both mechanical algorithmic work and cognitive work which lead me to think if the mechanical work could be automated and with the aid of SMT solvers how much of the cognitive work could be automated.

Chapter 2

Background

In this chapter, I will be outlining the prerequisite knowledge that is required to comprehend the achievements of this project. I will begin by visiting the notion of correctness in programs and in computer science and then proceed to talk about Hoare logic as a means to rigorously define correctness for programs. Thereafter, we shall discuss Satisfiability Modulo Theory and what it brings to the table.

2.1 Correctness

The notion of correctness with respect to programs is to verify that a certain program fulfills its specification. One way this can be done is by writing tests for the programs, checking the input-output pairs for certain cases. There are many popular testing suites that enable users to do this, namely, in the JavaScript/Typescript world there are suites like Jest and Mocha.

The Listing 1 contains code in Typescript for a binary search algorithm and Listing 2 contains a corresponding spec file that tests the binary search algorithm. We can see that the testing only depends on the signature of the function and is agnostic of the implemntation of the function, however, it is also worthy to note passing all the provided tests does not mean that the function is correct but only that it is correct for the given tests.

The mentioned above technique does not allow to assert absolute correctness because the specification is built by checking certain outputs. For example if the Listing 1 had a bug that would return -1 for every array greater than 20 elements or for arrays with a square size, then the code will still pass the specification but be incorrect nonetheless. The idea of specification is widely used in industry to ensure that changes to the codebase doesn't cause any old existing code to stray from its intended purpose. Software verification is a popular use for correctness.

```typescript
function binarySearch(key: number, sortedArray: number[]): number {
    let start = 0;
    let end = sortedArray.length - 1;

    while (start <= end) {
        let middle = (start + end) // 2;
        if (sortedArray[middle] === key) {
            return middle;
        } else if (sortedArray[middle] < key) {
            start = middle + 1;
        } else {
            end = middle - 1;
        }
    }
    return -1;
}
```

Listing 1: Binary search source code

2.2 Hoare Logic

Now that we have explored the problems that arise due to simple case testing when asserting correctness, it begs the need for a formal rigorous method which can be used to undeniable prove the correctness of a program with respect to its specification. This is where Hoare Logic comes in play. Developed by British computer scientist Tony Hoare in 1969, this logic defines a set of axioms and inferences that allow us to produce logical formulas, the validity of which prove the correctness of our program.

We shall discuss the details of Hoare logic in the While language that is often used to demonstrate Hoare logic. This language has the grammar as defined in Appendix A.1.

In his paper[1] Hoare introduce the following notation to describe the specification of a program known as a Hoare triple:

$$\{P\}\, C\, \{Q\}$$

Where C is a program in the While language defined above and P and Q are conditions on the program variable that are in C. Now, a Hoare triple is true if and only if C is executed in a program state that satisfies the condition P implies the condition Q is true on the program

```
describe('Binary Search', () => {
  test('Empty list returns -1', () => {
    expect(binarySearch(0, [])).toBe(-1);
  });
  test('List with key returns the index of the key', () => {
    expect(binarySearch(0, [0,1,2])).toBe(0);
  });
  test('List without key returns -1', () => {
    expect(binarySearch(0, [1,2,3])).toBe(-1);
  });
});
```

Listing 2: Binary search testing code using Jest framework

state that is after the execution of C.

Another important piece of notation is the proof tree (also known as deduction tree) which is the fraction like looking thing that will show up further into the discussion. The proof tree is a way to show the reasoning trace why a certain statement is true. The numerator bit contains the premises and the denominator bit hold the deduction from those premises.

The examples and rules below are adapted from a lecture notes document provided by Lecturer Mike Gordon on Hoare Logic[4].

The following rules and axioms allow us to generate conditions that will assert the correctness of out program C.

Skip Rule

$$\vdash \{P\}\,\text{skip};\,\{P\}$$

Hoare Assignment Axiom

$$\vdash \{P[E/V]\}\,V := E\,\{P\}$$

Where P[E/V] represents substituting all occurences of V in P with E

Example 2.2.1. Some examples of the assignment axiom are:

- $\vdash \{X := 0\}\, X := X + 1\, \{X = 1\}$
- $\vdash \{Y = 2\}\, X := 2\, \{Y = X\}$

□

There is also a forward version of the axiom which requires the existential quantifier and may make reasoning much harder when it comes to proving the verification conditions. We will, however, mention it here for completeness sake.

$$\vdash \{P\}\, V := E\, \{\exists v.(V = E[v/V]) \wedge P[v/V]\}$$

Consequence Rule

$$\frac{P' \Rightarrow P, \quad \{P'\}\, C\, \{Q'\}\,, \quad Q \Rightarrow Q'}{\{P\}\, C\, \{Q\}}$$

These premises known as precondition weakening and postcondition strengthening, will be vital in genertaing verification conditions. It allows us to work in one direction starting from the precondition or the postcondition and going forwards or backwards repectively, and then proving an implication to prove the statement.

Example 2.2.2. We know from arithmetic fact that $\vdash X \geq 0 \Rightarrow X + 1 > 0$, we can use this to show that the triple

$$\{X \geq 0\}\, X := X + 1\, \{X > 0\}$$

is a valid triple. By the assignment axiom and the arithmetic fact we have

$$\vdash \{X + 1 > 0\}\, X := X + 1\, \{X > 0\}\,, \quad \vdash X \geq 0 \Rightarrow X + 1 > 0$$

then by using the consequence rule we can get to

$$\{X \geq 0\}\, X := X + 1\, \{X > 0\}$$

□

The power of the consequence rule is not fully explainable yet but it will become evident how vital it is to the technical success of the project. Notice that the proof of the implication is one that requires intelligence and unlike most rules requires much more effort to achieve. This is where the SMT solver would come in.

Sequences Rule

$$\frac{\vdash \{P\}\, C_1\, \{Q\}\,, \quad \vdash \{Q\}\, C_2\, \{R\}}{\vdash \{P\}\, C_1; C_2\, \{R\}}$$

Most code will consist of more than one statement which is why the sequences rule is important as it dictates how we reason about when blocks of code are involved. We can also combine this with the consequence rule to get the derived sequencing rule which goes

Derived Sequencing Rule

$$\frac{\begin{array}{cc} & \vdash P \Rightarrow P_1 \\ \vdash \{P_1\}\, C_1\, \{Q_1\} & \vdash Q_1 \Rightarrow P_2 \\ \vdots & \vdots \\ \vdash \{P_n\}\, C_n\, \{Q_n\} & \vdash Q_n \Rightarrow Q \end{array}}{\vdash \{P\}\, C_1; \ldots ; C_n; \{Q\}}$$

Example 2.2.3. By the assignment axiom we have the following:

1. $\vdash \{A = a \wedge B = b\}\, C := A\, \{C = a \wedge B = b\}$
2. $\vdash \{C = a \wedge B = b\}\, A := B\, \{C = a \wedge A = b\}$
3. $\vdash \{C = a \wedge A = b\}\, B := C\, \{B = a \wedge A = b\}$

By the sequencing rule on 1 and 2 we can get the following:

4. $\vdash \{A = a \wedge B = b\}\, C := A; A := B\, \{C = a \wedge A = b\}$

And by the sequenceing rule on 4 and 3 we get:

5. $\vdash \{A = a \wedge B = b\}\, C := A; A := B; B := C\, \{B = a \wedge A = b\}$

□

$$\dfrac{\dfrac{\text{ass.}\ \overline{\{C = a \wedge A = b\}\, B := C\, \{B = a \wedge A = b\}} \qquad \dfrac{\text{ass.}\ \overline{\{A = a \wedge B = b\}\, C := A\, \{C = a \wedge B = b\},} \qquad \text{ass.}\ \overline{\{C = a \wedge B = b\}\, A := B\, \{C = a \wedge A = b\}}}{\{A = a \wedge B = b\}\, C := A; A := B\, \{C = a \wedge A = b\}}}{\{A = a \wedge B = b\}\, C := A; A := B; B := C\, \{B = a \wedge A = b\}}$$

Figure 2.1: Proof tree for Sequence rule example

Conditional Rule

$$\frac{\{P \wedge S\}\, C_1\, \{Q\}\,, \quad \{P \wedge \neg S\}\, C_2\, \{Q\}}{\{P\}\, \text{IF } S \text{ THEN } C_1; \text{ ELSE } C_2; \{Q\}}$$

Example 2.2.4. Let us try and prove the simple triple for finding the max of two elements,

$$\{X = x \wedge Y = y\}\, \text{IF } X > Y \text{ THEN } Z := X; \text{ ELSE } Z := Y; \{Z = max(x, y)\}$$

We want to show that the following two are valid triples

$$\{X = x \wedge Y = y \wedge X > Y\}\, Z := X\, \{Z = max(x, y)\}\,,$$
$$\{X = x \wedge Y = y \wedge \neg(X > Y)\}\, Z := Y\, \{Z = max(x, y)\}$$

We know by mathematical reasoning $\vdash x > y \Rightarrow x \geq y \Rightarrow max(x, y) = x$, so now we can employ the consequence rule and the assignment axiom to show the validity of the triple. By the assignment axiom we have

$$\{X = max(x, y)\}\, Z := X\, \{Z = max(x, y)\}\,,$$
$$\{Y = max(x, y)\}\, Z := Y\, \{Z = max(x, y)\}$$

and we also have the implications

$$X = x \wedge Y = y \wedge X > Y \Rightarrow X = max(x, y)$$
$$X = x \wedge Y = y \wedge Y \geq X \Rightarrow Y = max(x, y)$$

which by the consequence rule allows us to get to

$$\vdash \{X = x \wedge Y = y \wedge X > Y\}\, Z := X\, \{Z = max(x, y)\}\,,$$
$$\vdash \{X = x \wedge Y = y \wedge \neg(X > Y)\}\, Z := Y\, \{Z = max(x, y)\}$$

Therefore, the premises have been proven to be valid which then allows us to deduce that the conculsion is valid too. □

While Rule

$$\frac{\{I \wedge S\}\, C_1\, \{I\}\,, \quad I \wedge \neg S \Rightarrow Q, \quad P \Rightarrow I}{\{P\}\, \text{WHILE } S \text{ DO } C; \{Q\}}$$

Where I is an invariant to C such that $\{I\}\, C\, \{I\}$ is true,

Unlike all the other rules we have, the while rule is slightly different. It introduces the need for an invariant, a condition that is satisfied both before and after a piece of code has ran. However, an everyday invariant will not do the job the requirement for this invariant is much stronger than that, it requires that it must be a necessary condition to the precondition and while in conjunction with the negation of thw loop condition it must be a sufficient condition for the postcondition.

It is worth noting that this choice of invariant is also a task that requires intelligence and will more often than not require the user/developer/programmer to feed it into the system.

Example 2.2.5. Lets go through the divide program to demonstrate how the while rule functions,

```
⊢{X=x  ∧ Y=y}
R:=X;
Q:=0;
WHILE (Y ≤ R) DO (R:=R-Y; Q:=Q+1)
{x=qy+R  ∧  R<Y}
```

We can infer that the following is a valid triple,

```
⊢{X=x  ∧ Y=y}
R:=X;
Q:=0;
{R=x  ∧  Y=y  ∧  Q=0}
```

This leaves us to prove that the following is a valid triple and then we can use the sequence rule to combine the two,

```
⊢{R=x  ∧ Y=y  ∧  Q=0}
WHILE (Y≤R) DO (R:=R-Y; Q:=Q+1)
{x=qy+R  ∧  R<Y}
```

We will use the invariant $X = Q \times Y + R$, it is easy to verify the follwing by the asignment axiom and precondition strengthening,

```
⊢{X=Q×Y+R  ∧ Y≤R}
R:=R-Y;
Q:=Q+1;
{X=Q×Y+R}
```

We can also see that the implication

$$X = Q \times Y + R \wedge \neg(Y \leq R) \Rightarrow x = q \times y + R \wedge R < y$$

is true trivially as X and Y have been unchanged through the program. Lastly, we can also see that the implication

$$R = x \wedge Y = y \wedge Q = 0 \Rightarrow X = Q \times Y + R$$

is also true trivially as X and Y are unchanged in the program. We have now proved the 3 premises that must hold for a while statement to be proven valid. □

Moreover, before discussing loops we have not had to worry about termination of programs because all non-loop statements execute but once and are sure to terminate. When a program achieves a specification without guaranteeing termination it is known as those that have achieved partial correctness and those that also guarantee termination are known as having achieved total correctness. We will be dealing with partial correctness in the scope of this project.

Array Assignment Axiom

$$\vdash \{P[A\{E_1 \leftarrow E_2\}/A]\}\, A[E_1] := E_2\, \{P\}$$

Where A is an array variable and E_1 is an integer valued expression and $A\{E_1 \leftarrow E_2\}$ is an array identical to A except that at the E_1-th position holds the value E_2

To be able to more rigourously define the store notation, $\{E_1 \leftarrow E_2\}$, we shall define a few axioms around it.

Array Axioms

$$\vdash A\{E_1 \leftarrow E_2\}[E_1] = E_2$$

$$\vdash E_1 \neq E_3 \Rightarrow A\{E_1 \leftarrow E_2\}[E_3] = A[E_3]$$

Where A is an array variable and E_1 is an integer valued expression and $A\{E_1 \leftarrow E_2\}$ is an array identical to A except that at the E_1-th position holds the value E_2

This simply says that if you store a value at a position and then access that position then the value at that position must be what was stored at it. Secondly, If a store has not been performed at a particular position then its value must be equal to the value at that position before the store.

Example 2.2.6. We shall use the array version of the variable swap to demonstrate the above axioms

```
⊢ {A[X]=x  ∧  A[Y]=y}
R:=A[X];
A[X]:=A[Y];
A[Y]:=R;
{A[X]=y  ∧  A[Y]=x}
```

We will work backwards to constructing a pair of triple and implication in the fashion of the derived sequencing rule, By the assignment axiom we have:

```
⊢ {A{Y<-R}[X]=y  ∧  A{Y<-R}[Y]=x}
A[Y]:=R;
{A[X]=y  ∧  A[Y]=x}
```

Then we use the array axioms to strengthen the precondition:

```
⊢ {A{Y<-R}[X]=y  ∧  A[Y]=R}
A[Y]:=R;
{A[X]=y  ∧  A[Y]=x}
```

Resuming the backwards flow we have:

```
⊢ {A{Y<-R}{X<-A[Y]}[X]=y  ∧  A{X<-A[Y]}[Y]=R}
A[X]:=A[Y];
{A{Y<-R}[X]=y  ∧  A[Y]=R}
```

```
⊢ {A{Y<-A[X]}{X<-A[Y]}[X]=y  ∧  A{X<-A[Y]}[Y]=A[X]}
R:=A[X];
{A{Y<-R}{X<-A[Y]}[X]=y  ∧  A{X<-A[Y]}[Y]=R}
```

Now lets use the array axioms once again to show that the original precondition implies this final precondition. We could work through the two cases where $X = Y$ and $X \neq Y$ and reach the conclusion that the implication in question is a valid statement.

□

Using the above rules and axioms we can form proof trees the leaves of which can allow us to arrive at verification conditions which when asserted can be used to claim the validity of the program.

An example of this proof tree was provided in Figure 2.1 when discussing the sequence rule, we branched the tree upwards starting from the original program and then using the axioms and rules in a backwards fashion until we couldn't go any higher (we hit an axiom or an implication), the leaves of this tree are the implications and conditions that need to be proven true by mathematical inference or other non-trivial inferences.

2.3 Satisfiability Modulo Theory

To undestand SMT we must first understand SAT or the boolean satisfiability problem as it is more formally known. SAT asks the question that given a particular boolean formula is there an evaluation of variables such that that formula evaluates to true. SAT comes in a variety of flavours like 3SAT, XORSAT, Linear SAT, etc. which all put different constraints on what the formula can be in terms of what operators can be used or what syntax it must be in and some even alter the passing condition of the problem like MAJSAT which asks if the majority of evaluations evaluate to SAT.

Let us define some terminology that will be useful further into the report. A formula or a set of formulas are known as **sat** when there exists an evaluation of the variables in the formula that cause the whole formula or set of formulas to evaluate to true. A formula is **unsat** when no such evaluation exists. We should also define what is meant by when we call a formula **valid**, A formula is valid if and only if its negation is unsat, which means that a valid formula will be true for any evaluation of its variable, eg. $\mathtt{a} \vee \neg \mathtt{a}$.

SMT or Satisfiability modulo theory is one of the extension of SAT. It is somewhat like a supercharged SAT where we are still trying to find if for a given set of formulas we can find an evaluation that satisfies them but now our formulas are equipped with the knowledge of theories like Ints, Reals and Arrays. These theories allow for us to reason about variables that can be non boolean. For example the theory of ints has logic built in for the common operations like addition, multiplication, modulo remainder which means we can ask a SMT solver to solve for us problems like simultaneous equations.

SMT has a standardised language that comes along with it which is known as SMT-LIB[5] which can be interpreted by SMT solvers and used to give us models that satisfy our conditions. Here's a basic SMT-LIB program that solves a simultaneous equation in the integers.

```
(declare-const x Int)
(declare-const y Int)
(assert (= (- (* 2 x) (* 3 y)) 21))
(assert (= (+ x (* 2 y)) (- 7)))
(check-sat)
(get-model)
```

Note that SMT-LIB takes its conditions in a prefix manner instead of infix manner where the operator preceeds the operands. Here is the output that is produced when running the above SMT-LIB,

```
sat
(
  (define-fun y () Int
    (- 5))
  (define-fun x () Int
    3)
)
```

which tells us that the conditions are satisfiable and that their evaluations are -5 and 3 for y and x respectively. We see that an ordinary SAT solver would've been unable to reason about the Int logic but SMT equipped with the theory of Ints has no problem. The types are known as sorts and a user can define their own types along with any axioms of the type. Some common examples of these are mentioned on the SMT-LIB site.

Note the inner workings or SMT solvers is irrelavent to us because this project does not deal with proving verification conditions but rather just generating them and representing them in a way that SMT solvers can do all the proving.

Chapter 3

Implementation

3.1 Architecture

The architecture of the project consists of three main building blocks: the annotations which allow the user to input preconditions and postconditions as well as any other annotations like loop invariants, the heart of the system the verification condition generator that generates the verification conditions for the input program, lastly, the transpiler that takes verification conditions and yeilds a `.smt` file which can be used in conjunction with SMT solvers to verify the verification conditions.

3.2 Design choices

Some vital design choices in this project are the choice of language for developing the project (development language) as well as the language that the verification programs would be written in (program language). Below are some considerations for certain languages that were taken into account while making this choice.

- **Python**: This has always been a popular language when it has come to scripting, it has access for a multitude of libraries generally speaking as well as in the context of SMT solvers. Z3 (Microsoft's SMT solver) features an integration library with python which makes it very easy to use Z3 reasoning directly from within python. However, Python lacks an important feature that makes development difficult once the project scales, this is its lack of type safety. The latest version of python has implemented and supports type annotations but it is yet to be widely adopted. This is why Python was overlooked as it would have aided development within the last leg of the project which is SMT solving bit but would've hindered development otherwise.

- **C, C++**: These languages have been staples of the computer science industry when it comes to dealing with lower level tasks. The multitude of binaries that are available when dealing with compiler based tasks like parsing and lexing would largely aid development when it came to dealing with the annotation language or even parsing the program language.Also, Vampire, the solver that was recommended by my supervisor has better support when accessed through C++. However, most programs are best suited to a linux based machine and often require quite a few hurdles to jump through to make work with a windows machine.

- **Custom**: Another idea was to implement one's own primitive programming language. This would allow for extensive control over the language and allow for the language to be optimised for verification generation. However, this would be a large task that would take away from the project at hand. A parser and lexer would have to implemented for the whole language and even then the language wouldn't have any utility as it would require the backend of the compiler to be able to run on the computer. Having to spend a lot of time designing and implementing the appropriate elements to add utility to the language would be quite a drain on the time of the project bringing down the value per time for the project.

- **Typescript**: This is Microsoft's extension to ECMAscript (aka Javascript) to include type annotations and a compiler to enforce type safety. Due to the popularity of Node.js and TS backward compatibility with javascript, TS has a large variety of tools available to it and the distance of a single command. Moreover, like JS TS is also an interpreted language which means setting up the environment to get started is minimal.

As the typescript compiler is written in javascript itself, using typescript as program language will also be ideal as this will eliminate the need to harmonize the development code with an external language library. The program language will be a subset to the typescript language holding language features that allow it to express as many language features as possible albeit with more lines of code than native typescript would take.

3.3 Annotation language

The annotation language is the interface for the user, it is where the user can provide useful metadata about the program such as the precondition and postcondtion for the program as well as any loop invariants. This particular language needs to have a syntax quite simillar to the expression syntax of the program language which in this case is typescript. This is

to make it easy to generate the verification conditions as one does not have to worry about writing middleware to translate between the annotation syntax and the typescript syntax.

We make use of a parser combinator, known as nearley[7], to generate both, our lexer as well as our parser for the annotation language. The following grammar expresses the annotation language:

```
bool_exp
    → bool_exp "or" bool_exp | bool_exp "and" bool_exp
    | bool_exp "⇒" bool_exp | "not" bool_exp
    | "(" bool_exp ")" | bool_term
bool_term
    → math_exp relational_op math_exp
    | array_id "=" array_id
math_exp
    → math_exp "%" math_exp | math_exp "+" math_exp
    | math_exp "-" math_exp | math_exp "*" math_exp
    | math_exp "/" math_exp
    | integer | id | array_selection
array_selection
    → array_id array_store_pairs "[" math_exp "]"
```

Listing 3.1: "Grammar for annotation language"

For brevity, this is a gross simplification of the language grammar. It doesn't include precedence for the logical operators as well as the mathematical operators which renders the grammar as an ambiguous one, a more detailed version can be found in the Appendix A.2.

3.4 Verification condition generation

Let us get to the core implementation of the project. Generation of verification conditions can be derived from the qualitative analysis of the Hoare logic rules. Walking through mechanically, through what conditions would have to be verified if we performed the proof of the triple by hand, gives us quite a bit of insight into the intuition for the verification conditions.

Let us start with the most simple of the rules, the rule of the empty case,

$$\{P\}\,\text{skip};\,\{Q\}$$

Verification Conditions:

1. $P \Rightarrow Q$

Then we will define the rules for Assignments, Conditionals and Loops,

$$\{P\}\,V := E\,\{Q\}$$

Verification Conditions:

1. $P \Rightarrow Q[V/E]$

$$\{P\}\,\text{IF } B \text{ THEN } C_1;\,\text{ELSE } C_2;\,\{Q\}$$

Verification Conditions:

1. Conditions generated by $\{P \wedge B\}\,C_1\,\{Q\}$
2. Conditions generated by $\{P \wedge \neg B\}\,C_2\,\{Q\}$

$$\{P\}\,\text{WHILE } B \text{ DO } C_1;\{Q\}$$

Verification Conditions:

1. $P \Rightarrow I$
2. $I \wedge \neg B \Rightarrow Q$
3. Conditions generated by $\{I \wedge B\}\, C_1\, \{I\}$

Then the sequencing rule which comes in two flavours

$$\{P\}\, C_1; C_2; \ldots ; C_{n-1}; C_n;\, \{Q\}$$

Verification Conditions:

1. Conditions generated by $\{P\}\, C_1; C_2; \ldots ; C_{n-1}; \{R\}$
2. Conditions generated by $\{R\}\, C_n;\, \{Q\}$

$$\{P\}\, C_1; C_2; \ldots ; C_{n-1}; V := E;\, \{Q\}$$

Verification Conditions:

1. Conditions generated by $\{P\}\, C_1; C_2; \ldots ; C_{n-1}; \{Q[V/E]\}$

The sequencing rule is the entry point to our algorithm, we take a block of statements then then traverse in a backwards fashion until we reach an empty block. The keen amongst you must've noticed that the sequencing rule introduces a condition R from seemingly nowhere. This R is not provided by the user in our development, we can infer the R by using the (approximate) weakest precondition of the statement in the triple inspired by Dijkstra's implementation of the weakest precondition[3].

This inference is outlined by the function `f(C,Q)` where `C` is the program and `Q` is the post condition of the statement.

```
f(skip, Q)                    = Q
f(V:=E, Q)                    = Q[V/E],
```

```
f(IF B THEN C1 ELSE C2, Q) = (f(C1, Q) ∧ B)  ∨  (f(C2, Q) ∧ ¬ B)
f(C1;C2, Q)                = f(C1, f(C2, Q))
f(WHILE B DO C, Q)         = I
```

Remark. The I in the while rule is the invariant of the while loop

This is somewhat similar to the weakest precondition that is defined by Dijkstra with the exception of the while definition where we return the invariant as the verification of the invariant is done by the other verification conditions generated as shown above in the while verification conditions.

A good question at this point is, if the function above can calculate the weakest precondition why go through the bother of having to produce more than one verification condition for a program. This is a very valid question and one that I had right up until the time I had to start dealing with WHILE loops. You see, the function above correctly evaluates the weakest preconditions for programs that do not have WHILE loops and produces one verification condition which is the implication from the precondition to the weakest precondition of the program. The weakest precondition for while loops, however, is not as simple as the invariant. Assuming so means we assume that the invariant is valid, is implied by the precondition and implies the post condition.

The observation above allows for an ad-hoc optimization, if the block of statements `C` contain no loops then we execute the function `f(C,Q)` to find its verification condition.

Let's write some psuedocode to show how the verfication condition generator would work.

```
def generateVC(P,C,Q) {
    if (C is empty) {
        return P ⇒ Q;
    }

    last := getLastStatementInBlock(C);
    blockWithoutLast := getBlockWithoutLastStatement(C);

    if (last is assignment) {
        right := getRightSide(last);
        left := getLeftSide(last);
        return generateVC(P, blockWithoutLast, Q[right/left]);
    }

    if (last is IF) {
        thenBlock := getThenBlock(last);
        elseBlock := getElseBlock(last);
        return [
            generateVC(f(thenBlock,Q), thenBlock, Q),
            generateVC(f(elseBlock,Q), elseBlock, Q),
```

```
            generateVC(P, getBlockWithoutLastStatement(C), f(
               ↪ last, Q))
        ];
    }

    if (last is WHILE) {
        whileBlock := getWhileBlock(last);
        I := getInvariant(last);
        B := getCondition(last);
        return [
            I ∧ ¬ B ⇒ Q,
            generateVC(I ∧ B, whileBlock, I),
            generateVC(P, blockWithoutLastStatement, I)
        ];
    }

    throw Error;
}
```

Let us discuss the correctness of this code here. Firstly, we can assure that the code terminates this is because each recursive call will at some point in time end up passing in a block that has no conditionals or loops. This block free of loops and conditional will emulate a sort of tail recursion passing a block each time that is 1 smaller than the input block, and as the length of the block is bounded by 0 and the base case exists for a block at length 0 the function will terminate.

We can also add the optimization we mentioned above which then allows us to recurse through the function less times and decrease the number of verification conditions.

```
def generateVC(P,C,Q) {
    if (C is empty) {
        return P ⇒ Q;
    }

    if (C has no loops) {
        return P ⇒ f(C, Q);
    }

    ...
}
```

Up until now it has been smooth sailing apart from the implementation of the WHILE logic which was a head scratcher but not too bad. Now let us tackle Arrays. The array rule for assignment is awfully similar to the rule for regular assignment. It is:

$$\{P\}\, A[V] := E\, \{Q\}$$

Verification Conditions:

1. $P \Rightarrow Q[A/A\{V \leftarrow E\}]$

In plain English it says replace every occurrence of the array A with the array AV←E which is identical to the array A except for at position V where is store E. A small caveat here is that we should be able to carry over these store terms as our condition generator cannot reason about the store terms (this is the solver's job). So if we have a condition `A{V<-E}[X]=Y` and we need to replace all `A` with `A{T<-D}` we must be aware of the fact that the store of `V<-E` happens before `T<-D` and appropriately deal with it. We have chosen to put store terms one after the other the stores happen from left to right. In reference to the example our condition would now look like `A{V<-E}{T<-D}[X]=Y` this is to aid readability.

The array assignment rule also adds another flavour to the sequencing rule

$$\{P\}\, C_1; C_2; \ldots ; C_{n-1}; A[V] := E; \{Q\}$$

Verification Conditions:

1. Conditions generated by $\{P\}\, C_1; C_2; \ldots ; C_{n-1}; \{Q[A/A\{V \leftarrow E\}]\}$

Let's add this to our function above:

```
def generateVC(P,C,Q) {
    ...
    if (last is an array assignment) {
        arrayId := getArrayId(last);
        storeTerms := getArrayStoreTerms(last);
        index := getArrayIndex(last);
        expression := getRightSide(last);

        array := concatenate(arrayId, storeTerms)

        return generateVC(P, blockWithoutLastStatement, Q[array
          ↪ /concatenate(array, {index<-expression})]);
    }
}
```

Lastly we will be implementing another feature to the project, the ability to make function calls in your program. This feature unlike others doesn't have any rules for the triple rather this will just behave as a number valued variable. For example, `{P} V:=foo(X){Q}` will have the verification condition `P` $\Rightarrow$ `Q[V/foo(X)]` where `foo(X)` just behaves as an expression like in the assignment rule. This means that no additional logic needs to be added to the `generateVC` function.

The challenge here is generating the definition of the function programmatically. We only show a first order implementation of this feature, where we only allow for called functions to be pure functions ie. functions that only depend on their parameters and have no side effects. This does mean that we need to add logic for a return statement and abrupt termination of that kind.

A naive implementation here would be to give the return statement it's own special variable that can't be assigned to in a program and any statement `return x;` is equivalent to `@ret := x` where `@ret` is the special return variable. The problem with this approach is that it does not account for the short-circuiting behaviour of the return statement. If we would want to go with this approach we would have to add restrictions to how the return statement is used like it can only be used once and it can have no code after the return statement.

For a more fuller approach we would like to implement some sort of short-circuiting behaviour when evaluating the verification conditions. So let us use a different implementation as follows:

```
if (C has a return statement) {
    return generateVC(P, C upto return statement but excluding,
       ↪  globalPostcondition);
}
```

In plain English this approach is to perform the naive implementation mentioned above but also then performing the recursive call with the global postcondition which is equivalently assuming the case that the return statement is the last statement of the program. This implementation too is not perfect, it kind of falls apart when we get to loops because return statements in loops break out of the loop without any certainty to whether the loop condition is false or not. Take for example the following program:

```
{I}
WHILE (B) DO
    return 5;
{Q}
```

We cannot any longer assume that `B` is false after and in this case it most probably isn't and this kind of breaks down how we use the consequence rule to reason about the WHILE

loop. For this reason we will disallow the user to use return statements in loops. This does not however hinder the strength of expression of the language and one can still represent an equivalent program by using conditionals and apending to the loop condition, although, this may greatly increase the complexity of constructing the loop invariant but this is a problem for another day.

We would also need to amend the `f(C,Q)` to account for the short circuiting of the return statement. which will end up being the following rule:

```
f(C;return E;C', Q) = f(C, Q[@ret/E])
```

Now that we have added the logic for the return statement lets go back to discussing the implementation for function calls. For our implementation of function calls we require the function being called to be a pure function that return a value which means in terms of our annotation language the postcondition should be a condition only on auxiliary variables or the return variable. For example, let us take the max function with its specification:

```
{x=X ∧ y=Y}
if (x < y) {
    z := y;
} else {
    z := x;
}
{z ≥ X ∧ z ≥ Y ∧ (z=X ∨ z=Y)}
```

This is not a valid function that we can call in our implementation, it rather assigns the max to `z` which is then used to read the output. This means it expects the value of `z` to live on even after the function terminates which is side-effect of the function and therefore is invalid in our implementation. A valid version of the function would be something like:

```
{x=X ∧ y=Y}
if (x < y) {
    return y;;
}
return x;
{@ret ≥ X ∧ @ret ≥ Y ∧ (@ret=X ∨ @ret=Y)}
```

Given such a function in the body of our program, we treat the function as a program on its own and its correctness can be verified by verifying the verification conditions generated by it. The bigger question is how do we deal with showing the correctness of the function that is the caller. This is done after the verification conditions are generated, we scan through the verification conditions and find function calls, and then generate function call definitions that will be asserted along with the verification conditions.

Let us see an example:

```
{a=A}
func absolute(a) {
    return max(a, -a);
}
{(@ret=A ∧ A ≥ 0) ∨ (@ret=-A ∧ A<0)}

{x=X ∧ y=Y}
func max(x,y){
    if (x < y) {
        return y;;
    }
    return x;
}
{@ret ≥ X ∧ @ret ≥ Y ∧ (@ret=X ∨ @ret=Y)}
```

We will generate the verification condition

```
a=A ⇒ ((max(a,-a)=A ∧ A ≥ 0) ∨ (max(a, -a)=-A ∧ A<0))
```

Keep in mind the solver has no information about what the max function is or what it's properties are and we need to give it what the properties are and we will construct that based on the specification of the function. In terms of `max` we will have the definition as follows

```
forall (x,X,y,Y)
x=X ∧ y=Y ⇒ max(X,Y) ≥ X ∧ max(X,Y) ≥ Y ∧ (max(X,Y)=X ∨ max
   ↪ (X,Y)=Y)
```

One can extrapolate from what we have done here, we took the function parameters and fixed their value to some aux variables and then constructed the implication by implying the postcondition with `@ret` replaced with `max(X,Y)`. So in general given a function`foo` with with n parameters $(x_1, \ldots, x_n)$, precondition `P`, and postcondition `Q` we have the definition,

```
forall (x1, ..., xn, X1, ..., Xn)
(P ∧ (x1=X1 ∧... xn=Xn)) ⇒Q[@ret/foo(X1, ..., Xn)]
```

This concludes how we generate verification conditions for our input programs along with the extensions of arrays and function calls.

3.5 Transpilation to SMT-LIB

Now we shall look at how we take the given verification conditions and produce a valid SMT-LIB file that our solver can use to prove (or disprove) the validity of the verification conditions. The general structure of a file which verifies the verification conditions is:

1. Declarations for values, these may be integers, arrays, or functions within the scope of this program.

2. Definitions and properties of the above mentioned variables which may be things like definitions of functions, constraints on values, initial states of the values.

3. Assertions for the validation of verification conditions.

Another remark that we shall make is that the syntax in SMT-LIB for operators and function is that of prefix instead of the regular infix. For example to assert that $a > 5 \wedge a < n$ we would have the condition `(and (> a 5)(< a n))` in SMT-LIB. So this introduces the additional task of converting our conditions from infix to prefix.

Let us mention what inputs does the transpiler have in terms of generating the SMT-LIB file. It has 3 inputs: the precondition and function definitions, verification conditions, and the source code for the program. We will use the verification conditions to identify all the ids that need to be declared.

We have the luxury of having a parser made available to us by the work we did when constructing and parsing the annotation language. The parser allows us to construct an AST (Abstract syntax tree) for a given input string, which we will traverse all the nodes to find and store all the ids mentioned in the verification conditions, this includes integer valued ids, array valued ids and function ids.

We now need a way to differentiate between the ids, as their declarations are all different. function ids will easily be differentiable mostly due to the syntax as a function is always followed by brackets which contain parameters, as our functions are not first class variables where they can be assigned to. Between array and integer valued ids we can also differentiate easily as array ids will be succeeded by brackets with an integer valued expression. However, we want to be able to have our annotation language be able to fix arrays and allow for essentially auxiliary arrays so it can be asserted that the array is unchanged in the postcondition. This introduces the problem of how do we differentiate if `a=A` is a array equal to an auxiliary array or an integer id equal to to an auxiliary integer id.

We solve this by requiring the lexer to differentiate between an array and integer id. Let us assert that all array ids must be preceeded by a `!`, this way we know `a=A` are integer ids and `!a=!A` are array ids.

For example a (dummy) verification condition as follows:

`a` $\geq$ `0` $\Rightarrow$ `!x[a]>0` $\wedge$ `foo(a,b) > foo(b,a)` $\wedge$ `b=B` $\wedge$ `!x=!X`

will produce the following classification of ids:

```
int ids       → a, b, B
```

```
array ids    → !x, !X
function ids → foo
```

this will in turn produce the following declaration statements in SMT-LIB

```
(declare-const a Int)
(declare-const b Int)
(declare-const B Int)
(declare-const !x (Array Int Int))
(declare-const !X (Array Int Int))
(declare-fun foo (Int Int) Int)
```

Notice we assume that the parameters to the function are integers this is because we assume that all functions except for the main one are pure functions and passing in arrays may result in side-effects. We avoid the need to have checks on what has been done with the array by simply restricting our functions to not have arrays. However, a very straight forward extension is to ensure that the array is unchanged by requiring that the array preserves its initial state.

Now we shall assert the constraints on the declared variables, these are of two kind, firstly, the precondition that determines what values a particular id is allowed to take and then the function definitions which asset the properties of a function based on what the function specification and signature is. We discussed the generation of function definitions and here we can simply transform them to prefix and add them to the file.

Lastly we have the verification conditions which we will negate and then convert to prefix by a pre-order traversal of the AST of that verification condition. Here is some pseudo-code to that effect,

```
func toPrefix(node) {
    if (node is null) return;
    if (node is terminal) return node.value;

    if (node is binary op) {
        return (node.op toPrefix(node.left) toPrefix(node.right
            ↪ ));
    }
    if (node is unary op) {
        return (node.op toPrefix(node.right));
    }
}
```

this will turn to prefix all statements in our annotation language grammar. Some special statements that we need to keep in mind are function calls and array operations. We are utilising the array theory with the SMT solver which has two operations built into it that

satisfy the array axioms that we mentioned in Section 2.2. These two operations have the following signatures depicted in SMT-LIB:

```
(declare-fun select ((Array Int Int) (Int)) Int)
(declare-fun store ((Array Int Int) Int Int) (Array Int Int))
```

and each of the statement have the following properties,

```
- (forall ((a (Array s1 s2)) (i s1) (e s2))
  (= (select (store a i e) i) e))

- (forall ((a (Array s1 s2)) (i s1) (j s1) (e s2))
  (⇒ (distinct i j)
          (= (select (store a i e) j) (select a j))))

- (forall ((a (Array s1 s2)) (b (Array s1 s2)))
  (⇒ (forall ((i s1)) (= (select a i) (select b i)))
          (= a b)))
```

The first two are the array axioms we have already mentioned and the last one says that if for every index over an array, if the value at that array is the same as another array then the two arrays are equal. This might not hold true when it comes to programming as arrays that may be equal in all indices may not be equal as they are different references in memory but when it comes to reasoning that two arrays are equal, this axiom makes sense.

So a statement `!A[a]` would be transformed to `(select A a)` and a statement like `!A{a ↪ <-b}{c<-d}[e]` would be represented as `(select (store (store A c d)a b)e)` which suggests that our implementation first evaluates the stores from a left to right fashion and then evaluates the select on the array returned at the end of those store.

Let us go through the absolute function on page 23 and produce an SMT-LIB file for it. The verification conditions for the `absolute` function are,

$$a=A \Rightarrow ((max(a,-a)=A \wedge A \geq 0) \vee (max(a,-a)=-A \wedge A<0))$$

the classification of ids are as,

```
int ids      → a, A
function ids → max
```

which produce the following declare statements,

```
(declare-const a Int)
(declare-const A Int)
(declare-fun max (Int Int) Int)
```

Then we have the precondition and then the definition for max,

```
(assert (= a _A_))
```

```
(assert (forall ((x Int) (_X_ Int) (y Int) (_Y_ Int)) (⇒ (and
   ↪ (= x _X_) (= y _Y_) (= x _X_) (= y _Y_)) (and (≥ (max
   ↪ _X_ _Y_) _X_) (≥ (max _X_ _Y_) _Y_) (or (= (max _X_ _Y_)
   ↪  _X_) (= (max _X_ _Y_) _Y_)))))
```

and then we negate the verification conditions and prefix them to produce,

```
(assert (not (⇒ (= a _A_) (or (and (= (max a (- a)) _A_) (>
   ↪ _A_ 0)) (and (= (max a (- a)) (- _A_)) (≤ _A_ 0))))))
```

and lastly we add the `(check-sat)` statement to tell the solver to evaluate the satifiability of the conditions.

This concludes the transpilation of verification conditions to SMT-LIB

3.6 Usability and Interactivity

Now let us discuss the usability and interactive of the development. The verification generator is delivered as a command line tool, it takes in one positional argument which is the filename of the program that the verification conditions will be generated for and then optional arguments that enable other features of the generator.

An important one to mention is the ability to annotate programs. This is often an exercise that 1st year students perform where they go through a program and annotate it with preconditions starting from the postcondition. As we do this anyways for individual statements when we are evaluating the verification conditions, I have chosen to allow to output an annotated version of the source code. For example, for the max program,

```
//? x = _A_ AND y = _B_
function max(x: number, y: number) {
  let z = 0;
  if (x > y) {
    z = x;
  } else {
    z = y;
  }
  return z;
} //? @ret ≥ _A_ AND @ret ≥ _B_ AND (@ret = _A_ OR @ret = _B_
   ↪ )
```

would get annotated to,

```
//? x = _A_ AND y = _B_
function max(x: number, y: number) {
    //? ((((x)) ≥ _A_ and ((x)) ≥ _B_ and (((x)) = _A_ or ((x
       ↪ )) = _B_)) and (x > y)) or ((((y)) ≥ _A_ and ((y))
```

```
        ↪ ≥ _B_ and (((y)) = _A_ or ((y)) = _B_)) and not(x >
        ↪ y))
    let z = 0;
    //? ((((x)) ≥ _A_ and ((x)) ≥ _B_ and (((x)) = _A_ or ((x
        ↪ )) = _B_)) AND (x > y)) OR ((((y)) ≥ _A_ and ((y))
        ↪ ≥ _B_ and (((y)) = _A_ or ((y)) = _B_)) AND NOT(x >
        ↪ y))
    if (x > y) {
        //? ((x)) ≥ _A_ and ((x)) ≥ _B_ and (((x)) = _A_ or
            ↪ ((x)) = _B_)
        z = x;
    }
    else {
        //? ((y)) ≥ _A_ and ((y)) ≥ _B_ and (((y)) = _A_ or
            ↪ ((y)) = _B_)
        z = y;
    }
    //? (z) ≥ _A_ and (z) ≥ _B_ and ((z) = _A_ or (z) = _B_)
    return z;
} //? @ret ≥ _A_ AND @ret ≥ _B_ AND (@ret = _A_ OR @ret = _B_
    ↪ )
```

Furthermore, we have also added the ability for users to define custom functions in SMT-LIB that they can then reference in their annotations. This aids verification as we are able to define functions that more accurately represent the condition we are trying to describe, also we can make use of first order logic quantifiers because SMT-LIB supports them where as the annotation language doesn't. There also a version of overriding implemented where it prefers functions defined in the source code to those defined in SMT-LIB this is to allow for a user to care less about if the function is already defined and will cause errors.

Chapter 4

Conclusion and Reflection

4.1 Further Work

The program language is limited in terms of the features that it has, things like custom data types and functions with parameters that are passed by reference. Most of this can be implemented by implementing pointers and implementing the logic that is required to reason about logic. This requires diving into something that is known as separation logic[6] where a heap store is added to the state of a program and then the reasoning is now a function of both the program and the heap store.

Another important construct is the construct of a FOR loop. Naively, one would think that the FOR loop is just a WHILE loop with an additional assignment statement, however, a major difference is that a for loop will always terminate (At least how it is used in most conditions). This condition of termination affects how we reason about the FOR loop and introduces a new type of condition known as the variant.

Furthermore, the idea of scope is much more of a quality of life improvement but one that has a major impact on how code it written and the readability of the code. Currently, the script assumes that all variables in a function are globally scoped to the function, which means variables declared within IF blocks or WHILE blocked are accessible throughout the function. This idea is not one that is difficult to implement but is tedious indeed. One implementation is to pre-process the file and prefix or suffix each variable with a string that uniquely identifies its block. Also, this string may encode the parent scope of the current scope as well allowing the child scope to access the parent scope declaration but preventing it the other way around.

4.2 Testing

Most of our testing has been writing sample-programs that can be found in the code archive submitted. Some of those samples do make appearences in the report when trying to explain the background or the implementation challenges. These include the max program, the absolute program, the euclidean division algorithm.

We also setup somewhat of an automated E2E testing system by testing the output of the SMT solver when fed the SMT-LIB file the is produced by the sample programs. This was done in an attempt to have some degree of regression testing and allow for new features to be implemented without having to tiptoe around code and be afraid of messing up already working examples. The testing framework Jest[8] was used to achieve this.

4.3 Evaluation

Let us list again the requirements of the project

- Program takes in typescript source code and outputs an SMT-LIB file that encodes the code's verification conditions
- Program can be triggered by a command line interface with optional and positional arguments that enable certain features of the program
- Program can output an annotated version of the source code that prefixes every statement with a weakest precondition

We can see that we have manged to tick all of the mentioned requirements. However, I believe we have achieved more than what we have set out to do to produce an interface that is user friendly and can be understood relatively quickly.

In hindsight, I believe the project should've broadened it's scope to allow for more implementations and a more accessible UI, something along the lines of a webservice that would've been able to give verifications on the go without users having to set up the runtime environment for the project.

Also, I would consider to opt for a language that has a library to access a SMT solver programmatically. This would allow for a more whole experience where the SMT-LIB would be generated optionally and verification could be performed in the code it self which would also allow for counter examples to be displayed in a more user-friendly way.

Bibliography

[1] Hoare, C. A. R. (October 1969). "An axiomatic basis for computer programming"

[2] R. W. Floyd. "Assigning meanings to programs." Proceedings of the American Mathematical Society Symposia on Applied Mathematics. Vol. 19, pp. 19–31. 1967.

[3] Dijkstra, Edsger W. (1976). A Discipline of Programming. Prentice Hall. ISBN 978-0-613-92411-5. — A systematic introduction to a version of the guarded command language with many worked examples

[4] Gordon, M., 2015. Background reading on Hoare Logic. [ebook] Cambridge: University of Cambridge. Available at: <https://www.cl.cam.ac.uk/archive/mjcg/Teaching/2015/Hoare/Notes/Notes.pdf>.

[5] Smtlib.cs.uiowa.edu. 2021. SMT-LIB The Satisfiability Modulo Theories Library. [online] Available at: <https://smtlib.cs.uiowa.edu/index.shtml>.

[6] Reynolds, John C. (2002). "Separation Logic: A Logic for Shared Mutable Data Structures"

[7] Nearley.js.org. n.d. Home - nearley.js - JS Parsing Toolkit. [online] Available at: <https://nearley.js.org/>.

[8] Jest. 2021. Jest.io. [online] Available at: <https://jestjs.io/>.

Appendices

Appendix A

Grammars

A.1 WHILE Language

$$
\begin{array}{lcl}
S & ::= & x := a \\
 & | & \text{skip} \\
 & | & S_1;\ S_2 \\
 & | & \text{if } P \text{ then } S_1 \text{ else } S_2 \\
 & | & \text{while } P \text{ do } S \\
a & ::= & x \\
 & | & n \\
 & | & a_1\ op_a\ a_2 \\
op_a & ::= & + \mid - \mid * \mid / \\
P & ::= & \text{true} \\
 & | & \text{false} \\
 & | & \text{not } P \\
 & | & P_1\ op_b\ P_2 \\
 & | & a_1\ op_r\ a_2 \\
op_b & ::= & \text{and} \mid \text{or} \\
op_r & ::= & < \mid \leq \mid = \mid > \mid \geq
\end{array}
$$

Figure A.1: Grammar for WHILE language

A.2 Annotation Language

```
@{%
const moo = require('moo');
// Order is important
const lexer = moo.compile({
    implication_op: '⇒',
    replace_op: '<-',
    rel_op: ['>', '≥', '<', '≤', '=', '!=', '==', '==='],
    or_word: {match: ['or', 'OR'], value: s ⇒ 'or'},
    and_word: {match:['and', 'AND'], value: s ⇒'and'},
    not_word: {match:['not', 'NOT'], value: s ⇒'not'},
    left_para: '(',
    right_para: ')',
    plus_op: '+',
    minus_op: '-',
    mul_op: '*',
    list_sep: ',',
    div_op: '//',
    mod_op: '%',
    return_id: '$ret',
    left_array_bracket: '[',
    right_array_bracket: ']',
    left_replace_bracket: '{',
    right_replace_bracket: '}',
    truth: 'true',
    array_id: /![a-zA-Z]+/,
    array_id_aux: /!_[a-zA-Z]+_/,
    id: /[a-zA-Z]+/,
    id_aux: {match: /_[a-zA-Z]+_/},
    integer: /\d+/,
    ws: /[ \t]/,
});
%}

@lexer lexer

b_exp
    → _ or_exp _
# OR expression
or_exp
    → or_exp (__ %or_word __) and_exp
    |  and_exp
# AND expression
```

```
and_exp
    → and_exp (__ %and_word __) impl_exp
    |  impl_exp
# Implication expression
impl_exp
    → impl_exp (_ %implication_op _) not_exp
    |  not_exp
# NOT expression
not_exp
    → (%not_word _ "(" _) or_exp (_ ")")
    |  bool_term
    |  "(" (_ or_exp _) ")"
# A term that returns true or false
bool_term
    → rel_exp
# Relational expression that compares two math expressions
rel_exp
    → math_exp (_ %rel_op _) math_exp
    |  array_term (_ "=" _) array_term
    |  %id
# Math exprssion
math_exp → mod_term

mod_term
    → mod_term (_ %mod_op _) sum_term
    |  sum_term
sum_term
    → sum_term (_ ( %plus_op | %minus_op ) _) mul_term
    |  mul_term
mul_term
    → mul_term (_ (%div_op | %mul_op) _) math_exp
    |  %minus_op mod_term
    |  "(" mod_term ")"
    |  function_call
    |  term
function_call
    → %id "(" (_ arg_list _) ")"
arg_list
    → arg_list (_ "," _) mod_term
    |  mod_term # should return a single element array
term
    → (%id | %integer | %id_aux | %return_id | array_selection
      ↪ )
array_selection
    → array_term (_ %left_array_bracket _) mod_term (_ %
```

```
      ↪ right_array_bracket)

array_term
    → (%array_id | %array_id_aux) array_store_term:?

array_store_term
    → array_store_term (_ "{" _) mod_term (_ %replace_op _)
      ↪ mod_term (_ "}" _)
    | (_ "{" _) mod_term (_ %replace_op _) mod_term (_ "}" _)
_ → %ws:* # optional whitespace
__ → %ws:+ # mandatory whitespace
```