Friedrich-Alexander-Universität
Erlangen-Nürnberg

T.cs
Chair for Computer Science 8
Theoretical Computer Science

Development of a Programming Environment and Interpreter for LOOP and WHILE

Bachelor Thesis in Computer Science

Michael Gebhard
michael.gebhard@fau.de

Supervised by:

Stefan Milius Tadeusz Litak

Erlangen, November 22, 2018

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Abstract

In this thesis, we provide an intuitive programming environment for the languages LOOP and WHILE. Our tool is designed for educational use in introductory lectures on theoretical computer science such as ThInfWiL at FAU. We define syntactic extensions to both languages (Section 2) in order to make them practically usable in programming exercises, in particular we introduce macros which allow a programmer to reuse code (Section A.6 and Subsection A.9.16) and arbitrary identifieres for storage locations (Section A.5). Then we build an interpreter for these extended languages (Section 3) along with a plugin for the integrated development environment IntelliJ IDEA (Section 4), in order to allow students to solve programming exercises in a familiar environment.

Abstract (Deutsch)

In dieser Arbeit stellen wir eine intuitive Programierumgebung für die Sprachen LOOP und WHILE bereit. Die Umgebung soll für pädagogische Zwecke in Einführungsorlesungen zur theoretischen Informatik, wie ThInfWiL an der FAU, verwendet werden. Zunächst definieren wir syntaktische Erweiterungen beider Sprachen (Section 2), um sie praktisch verwendbar in Programmierübungen zu machen, ins besondere führen wir Makros, welche Programierenden erlauben Quelltext wiederzuverwenden (Section A.6 und Subsection A.9.16), und freie Wahl von Namen für Speicherstellen (Section A.5) ein. Für diese erweiterten Sprachen implementieren wir dann einen Interpreter (Section 3) sowie ein Zusatzmodul für die Entwicklungsumgebung IntelliJ IDEA (Section 4), um Studierenden das Bearbeiten von Programieraufgaben in einer gewohnten Umgebung zu ermöglichen.

Contents

1 Introduction

This work provides a programming environment for the LOOP and WHILE languages to assist students in a lecture on theoretical computer science [13]. The lecture uses LOOP and WHILE to illustrate the distinction between *recursion* and *primitive recursion*. It is widely accepted that learning programming languages is assisted by an intuitive programming environment (see e.g., [3]). We provide such an environment including a language interpreter which illustrates the steps taken when executing a LOOP or WHILE program.

In Section 2, we first discuss syntax and semantics and computational properties of the LOOP and WHILE languages. Then we give syntactic extensions to both in order to turn them into practically usable programming languages.

In Section 3, we examine different implementations for a language processor, i.e. a tool which executes LOOP/WHILE programs. We come to the conclusion that an interpreter is the best choice for our didactic use case. We complete Section 3 with an in-depth discussion of the interpreter implementation, including debugging functionality.

In Section 4, we elaborate on the implementation of an IntelliJ plugin which provides the intuitive programming environment we aim for. We first present IntelliJ's plugin API and the structure of the plugin, as prescribed by the API. Then we examine the necessary interactions between IntelliJ, our plugin and our interpreter to provide most of IntelliJ's debugging features.

In Section 5, we compare our interpreter and plugin to already existing projects that address similar issues.

Finally in Section 6 we summarize the achievements of our project and outline further features that could be implemented in future work.

Furthermore, in Appendix A, we provide informal semantics for the extended LOOP and WHILE syntax, along with textual substitution rules, to transform any LOOP or WHILE program from extended to basic syntax.

In Appendix B, we give a step-by-step guide to the features of IntelliJ's GUI and our interpreter's command line interface.

2 LOOP and WHILE languages

LOOP was introduced by Meyer and Ritchie [12] as a register machine to investigate their complexity. In later references such as Schöning [15] and Hoffmann [6], LOOP was presented as an imperative programming language. We are following the same route using Hoffmann [6] as our main reference.

The syntax of LOOP programs is shown in Figure 2.1. The LOOP language operates on storage locations [17], here called program variables or just variables. *Variable identifiers* are arbitrary strings in contrast to Schöning's definition [15], where variables are numbered `A0` to `Am` with $m \in \mathbb{N}$. This allows programmers to use meaningful variable identifiers, in order to write readable code.

Definition 2.1 (LOOP State)**.** The *state* s of a loop program `P` is a function $s : Var \to \mathbb{N}$, where Var is the set of strings consisting of all variable identifiers occurring in `P`. We call $s(X)$ the *value* of, *content* of, or *number stored* in X.

Initially the state s maps all variable identifiers to 0, except for variables designated as input (Definition 2.4). Each instruction in a LOOP program alters s as described in Definition 2.2. Saying some instruction alters the value of X to a number n means that the instruction alters the current state s into a state s' such that $s'(X) = n$ and $s'(Y) = s(Y)$ for all variables $Y \neq X$.

Definition 2.2 (LOOP Language Semantics)**.** The LOOP language consists of 4 types of instructions.

- `X := 0` : Sets the content of variable `X` to 0.
- `X := Y` : Copies the content of variable `Y` into variable `X`.
- `X := X + 1` : Increments the number stored in `X` by one.
- `LOOP X DO P ENDDO` : Repeats LOOP program `P` n times, where n is the number stored in `X` before the first execution of `P`.

The WHILE language, following the definition of Schöning [15], is an extension of the LOOP language to achieve greater computability. We will go further into the expressive power of both languages in Section 2.1.

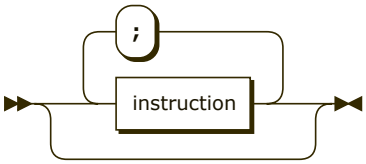
The syntax of WHILE programs is shown in Figure 2.2.

Definition 2.3 (WHILE Language Semantics)**.** The WHILE language follows Definition 2.2 and has one additional instruction type.

- `WHILE X != 0 DO P ENDDO` : Repeats the WHILE program P until x is equal to 0, where x is the value stored in `X` before each execution of P.

One could make the definition of semantics of WHILE fully formal using fixpoints and partial functions, following the denotational semantics of IMP given by Winskel [17]. The closest matching semantics for our definition of WHILE are those given by Hoffmann [6, Ch. 6.1.2].

program:



instruction:

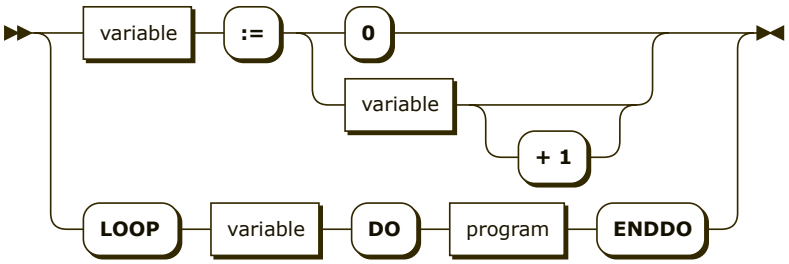
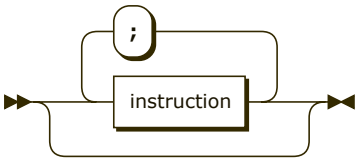


Figure 2.1: Syntax of LOOP Programs

program:



instruction:

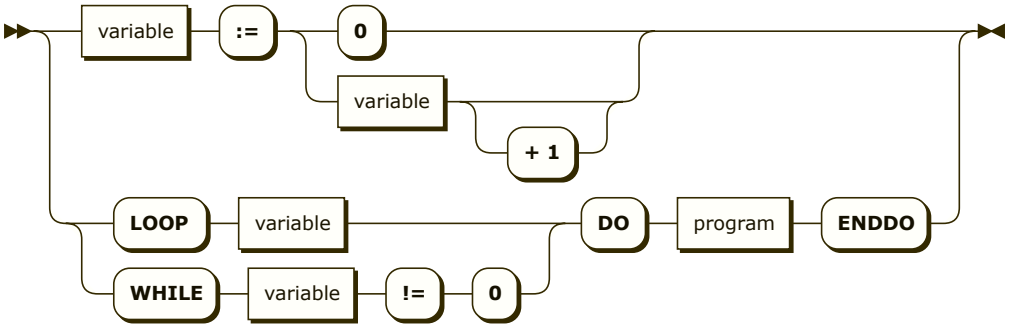


Figure 2.2: Syntax of WHILE Programs

2.1 Properties

We define the *computability* of a programming language by means of the class of functions which can be computed by a program in that language.

Definition 2.4 (Computing Functions)**.** Let $f(x_1, x_2, ..., x_n)$ be a function $\mathbb{N}^n \mapsto \mathbb{N}$ with $n \in \mathbb{N}$. A LOOP or WHILE program `P` *computes* f, if variables `X1`, `X2` through `Xn` initially contain x_1, x_2 through x_n and after execution of `P` variable `Y` contains $f(x_1, x_2, ..., x_n)$. `X1`, `X2` through `Xn` are called input variables, `Y` is called the output variable.

In Schöning's definition of the LOOP language [15], variables have fixed names with indices, `A0` through `Am`, $m > n, m \in \mathbb{N}$. He statically defines `A0` as the output variable and `A1` through `An` as input variables. In our approach we allow input and output variables to be freely declared in the extended syntax (Definition A.7) since we allow arbitrary variable identifiers.

Listing 2.3: LOOP Example

```
LOOP X DO
        X := X + 1
ENDDO
Y := X
```

The LOOP program in Listing 2.3 increments `X` x times and stores the result in `Y`, where x is the initial value of `X`. If `X` is designated as single input and `Y` as output variable, the program computes the function $f(x) = 2x$.

The class of functions computable by LOOP programs is exactly the class of the primitive recursive functions [12].

All LOOP programs terminate, as every `LOOP X DO P END` block is executed a finite number of times. The WHILE language introduces non-terminating programs, e.g. Listing 2.4.

Listing 2.4: Infinite loop

```
X := 0;
X := X + 1;
WHILE X != 0 DO
ENDDO
```

The class of functions computable by WHILE programs are the μ-recursive functions, making the WHILE language Turing complete [6].

2.2 Syntactic Sugar

In order to practically use LOOP and WHILE to solve algorithmic problems, while observing their differences in computability, we introduce several syntactic extensions to both languages (Figure 2.5). These extensions must not alter the expressive power of either language. To ensure this, all programs with extended syntax can be transformed into traditional LOOP and WHILE programs through textual substitution. In order to simplify writing code, the keywords of LOOP and WHILE are made lower case in the extended syntax. The semantic definitions and textual substitution rules for the extended syntax are listed in A. The most noteworthy syntactic addition is macros, see Section A.6 and Subsection A.9.16. A macro is a LOOP/WHILE program

with a name, which can be called through its name within another program. This allows a programmer to reuse the program in the macro without duplicating it. We will discuss the behaviour of macros in detail throughout Section 3.4 and Section 3.5.

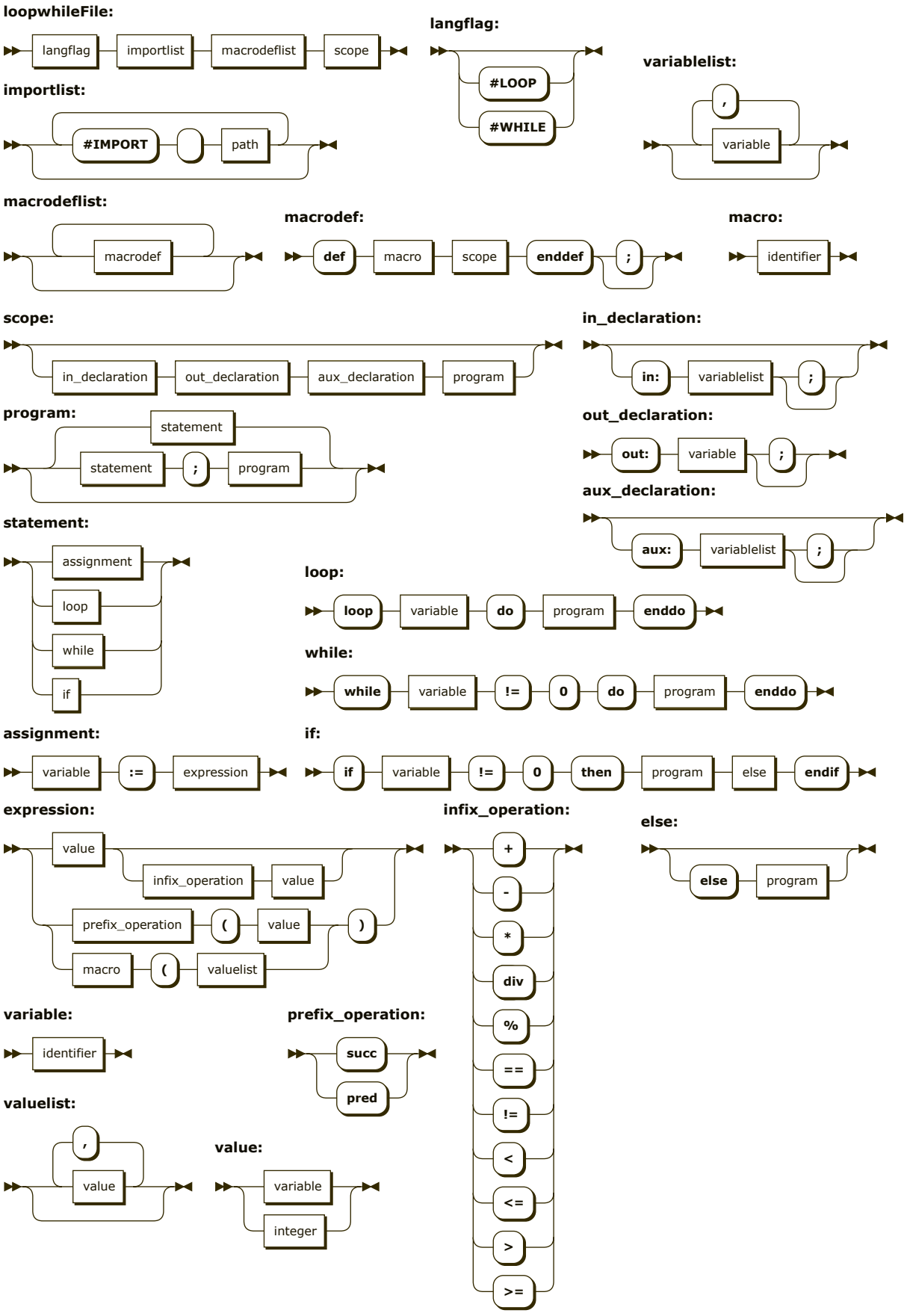


Figure 2.5: Extended Syntax of WHILE Programs

3 Programming Language Processor

In this chapter we will discuss different options to implement a tool which can execute a LOOP or WHILE file (Definition A.1). We will compare the advantages and disadvantages of each option and outline the implementation details of our chosen approach.

Watt [16] defines broadly a programming language processor as any system that manipulates programs expressed in some particular programming language. This includes anything from a simple text editor to a compiler. For our terms we will redefine programming language processors to the tools directly related to program execution.

Definition 3.1 (Programming Language Processor)**.** We define a *programming language processor* to be one of the following:

- *Translators* and *compilers* translate a program from one language to another. A translator produces a program in another high-level language. A compiler translates a high-level program into a low-level program.
- *Interpreters* skip compilation and directly execute a high-level program, by simulating one instruction after another.

3.1 Input Specification

Both compilers and interpreters typically prepare the program by using a lexer and a parser to generate an *abstract syntax tree* (*AST*, Subsection 3.4.3) from the source file. A compiler then translates this AST into machine code which is executable by the target computer. An interpreter operates directly on the AST after optional preprocessing, simulating the instructions at each node. A translator generates source code in a second high-level language in order to use existing tools for that language. The translator can work on an AST of the initial source code or if the syntax of the target language is close enough to that of the source language, it can be implemented as textual substitution directly on the source code.

3.2 Output Specification

A translator or compiler for a LOOP/WHILE file produces an executable which asks values for the input variables from the user and prints the value of the variable declared as output after computing the function described by the LOOP/WHILE file (Definition 2.4).

An interpreter executes a LOOP/WHILE file in the same manner as the executable produced by a compiler. The interpreter asks the user for values for the input variables and prints the value of the variable declared as output after computing the function described by the LOOP/WHILE file (Definition 2.4).

Listing 3.1: Simulation of a `while loop` type node

```
function execute(while_node) {
        while (get_value(while_node.variable) != 0) {
                execute(while_node.program)
        }
}
```

3.3 Comparison of Language Processors

In this section we will compare intuitive implementation approaches for an interpreter, a compiler and a translator. The discussion of the interpreter is inspired by Watt [16]. The presentation of the compiler is based on Sarda [14]. The subsection on the translator follows Watt [16], as well as our own attempt to implement a translator through textual substitution.

These three language processors all differ in performance and complexity of implementation. The implementation details can be split into three relevant parts for our project.

- Execution of code in case of an interpreter and preparation of code in case of a translator or compiler.
- Pausing execution at breakpoints
- Serialising state of execution when pausing, to present the current state in a debugger.

All of these language processors must either parse a given program into a processable data structure or in case of a textual substitution translator, at least check for syntax errors in the program. The most common approach for this is to use a lexer and a parser generated from the syntax definition of the programming language. Syntax errors are produced by this lexer and parser, as described in Subsection 3.4.1 and Subsection 3.4.2, therefore the implementation complexity for parsing and syntax checking is equal for all of these language processors and will not be taken into account in this discussion.

No errors can occur at runtime in a syntactically correct LOOP or WHILE program, therefore, just like parsing errors, we do not have to discuss implementation of runtime errors either.

3.3.1 Interpreter

The disadvantage of an interpreter is its speed of execution. The interpreter has to analyse every instruction before executing it. The time spent on analysis can be decreased by first parsing the program into an AST. In this case instructions do not have to be parsed from the source code every time they are executed. Regardless of optimisation an interpreter cannot be faster in execution than running compiled code. Even if no time at all were spent on analysing instructions, the interpreter still has to execute the same functionality as the compiled code, thus all interpreters have worse execution performance than running compiled code.

An interpreter operating on an AST, such as Figure 3.8 on page 21, needs a piece of code for each type of node in the AST. This piece of code simulates execution of the source code represented by the node. Listing 3.1 shows an example for a node of type `while loop`, where a `while loop` node has two children, one of type variable and one of type program. The complexity of implementing these simulation functions depends on the complexity of each instruction in the given programming language. The LOOP and WHILE languages contain no intrinsically complex instructions, therefore simulating the instructions is also simple.

Breakpoints at a certain line of code can be implemented by searching through the AST and marking the first node that was parsed from the given line as a breakpoint. If the interpreter attempts to execute a node that is marked as a breakpoint, it pauses execution and serialises the state of execution.

Definition 3.2 (State of execution)**.** The *state of execution* consists of the current position in the source code and the bindings of all currently valid variables. The binding of a variable consists of the variable identifier and the value assigned to this variable.

The interpreter has to keep track of all variable bindings and can achieve this by embedding the state of execution into its internal state. The interpreter can read the current position in the source code from the position saved in the AST node which it is currently executing. To serialise the state of execution, the interpreter can then print out its internal state and the position from its current AST node.

3.3.2 Compiler

A compiler translates a high-level program into a low-level program. Implementing a compiler to directly generate low-level programs for all major hardware architectures is a tedious and unnecessary task. Instead, a compiler implementation can translate the source code into an intermediate low-level language such as LLVM intermediate representation (IR) [9]. LLVM IR is independent of the target hardware and highly developed tools for optimising LLVM IR code already exist and can be used by the compiler.

Producing LLVM IR code from LOOP and WHILE programs is fairly simple. There are two issues to consider. First, LOOP and WHILE operate on natural numbers, not integers of limited size. This can be solved by linking in a big integer library and replacing operations on numbers by calls to this library in the LLVM IR code. Second, LLVM IR has no concept of loops, therefore `loop` and `while` instructions have to be rewritten by the compiler into `jump`, `compare` and `goto` representation.

Debugging code compiled through LLVM can be done with the debugger LLDB [11]. LLVM IR code can be annotated with source code positions which eliminates the need to manually translate a breakpoint line position into position of an LLVM IR instruction. LLDB can then set breakpoints following these annotations and stop execution at the first LLVM IR instructions parsed from the given line.

To serialise the state (Definition 3.2) of execution LLVM supports annotations for variable identifiers. LLDB can then produce the desired serialised variable bindings and execution position.

3.3.3 Translator

A translator translates a high-level program into another high-level program. If the translation operates on an AST, all nodes must be translated into source code for equivalent instructions in the target language. If the target language does not natively support natural numbers, a big integer library must be included and operations on numbers must be replaced by calls to this library. Instead of operation on an AST, a translator can also be implemented by textual substitution if the syntax of the target language is very similar to that of the source language. For example LOOP and WHILE programs can easily be rewritten to code in the programming language Python because it shares the only two uncommon features of LOOP and WHILE.

- Python natively supports big integers.
- Python allows booleans to be cast into integers to behave like comparison results in LOOP and WHILE.

The remaining syntax differences are trivially solved by textual substitution.

For a textual substitution translator all syntax checks must be done before translating. This can be done by running a parser but discarding the resulting AST if all checks succeed. Both types of translators must ensure that no parsing errors can occur after translation when the translated program is processed by the tools of the target language. Especially the translators must rename variables that conflict with keywords in the target language. This can be done by prefixing every variable name with e.g. `var_` in case of Python as target language.

Line breakpoint support can be implemented in the translator from AST by keeping a mapping from lines in source language to lines in target language. The textual substitution translator to Python can be implemented to create exactly one line of Python code for each line of LOOP or WHILE code. Line numbers can then be passed to the target language tools unchanged.

In the serialised state (Definition 2.1) of the target language processor, variable names must be translated back by removing the prefix described above. In case of the translator from AST the execution position must also be translated into the respective position in the source language.

3.3.4 Our Chosen Approach

The language processor we aim to provide is intended to ease learning of the LOOP and WHILE languages. This makes the ability to view and understand the internal state of execution far more important than the speed of preparation and execution of a program. All three approaches can produce suitably understandable output about their state of execution, but differ in implementation complexity and maintainability. The compiler implementation requires understanding of low-level programming concepts, in order to correctly use the LLVM API. The translator introduces a dependency on an entirely different programming language. Every change in the translators target language or the tools and libraries of that language can require adaptation in the translator. The interpreter remains with an intuitive implementation and without additional external dependencies, apart from a lexer and a parser. In order to provide an easily maintainable language processor we chose to implement an interpreter operating on an AST.

3.4 Interpreter Implementation

In this section we will describe the structure of our LOOP/WHILE interpreter including the lexer and parser along with the most important implementation choices we made. The interpreter only supports the extended LOOP/WHILE syntax (Figure 2.5), because a program without a declared output variable (Definition A.7) would provide no feedback to the user and therefore be of little didactic use. We have chosen the programming language `Go` [5] along with the tools `Golex` [1] and `Goyacc` [2] for the implementation of this interpreter. `Golex` and `Goyacc` are versions of the Unix utilities `Lex` and `Yacc` [10] for the `Go` programming language.

Figure 3.2 shows the structure and interactions of the different components we chose for the interpreter. The lexer (Subsection 3.4.1) and parser (Subsection 3.4.2) process a LOOP/WHILE file into an AST (Subsection 3.4.3). The core of the interpreter executes the instructions encoded in the AST, asks the user for values of input variables, if there are any, and shows the output variable to the user after execution. The debugger described in Section 3.5 allows the user

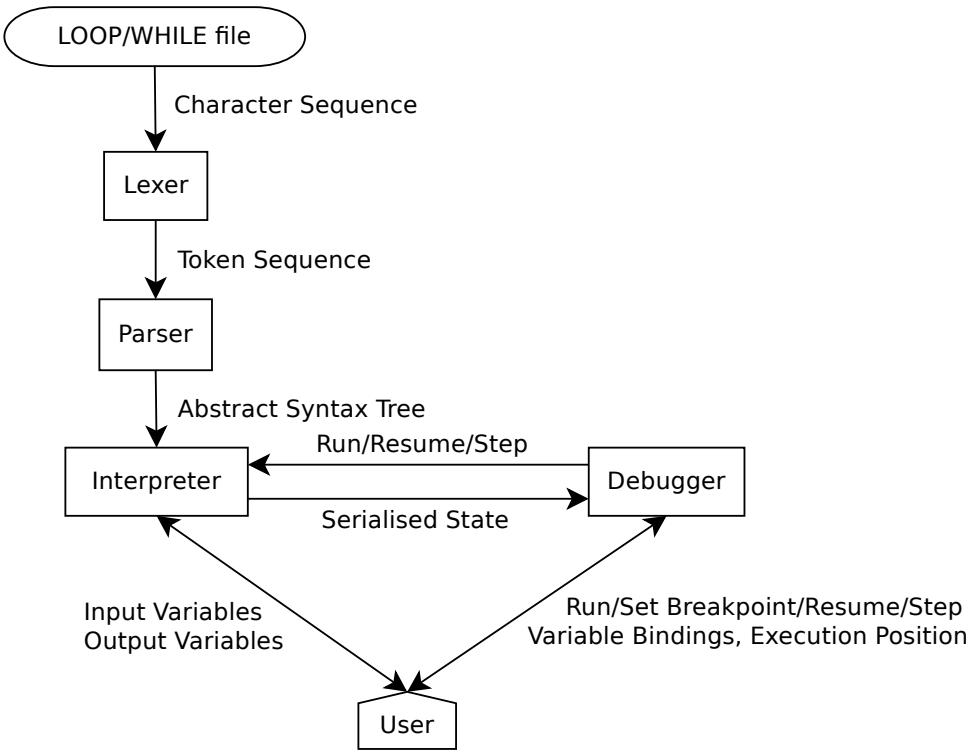


Figure 3.2: Structure of the Interpreter

to control the interpreter through inserting breakpoints into the AST, stepping the execution, i.e. pausing after every instruction, and resuming normal execution.

In case of errors every individual component can halt the interpreter and print information about the error to the user.

3.4.1 Lexer

Definition 3.3 (Lexer)**.** A *lexer* takes a sequence of characters in a source code file as input, splits it into substrings called *tokens* and assigns a token type to each of these tokens.

Definition 3.4 (Lexicon)**.** A *token type* is an arbitrary identifier which declares a token to belong to a specific group of tokens, e.g. **variable identifier** or **assignment operator**. Token types are defined by an ordered sequence of regular expressions, called *lexicon*, where each expression represents one token type.

The tool `Golex` [1] generates a lexer from a lexical definition of token types. Listing 3.3 shows simplified example definitions of the token types for the **while** keyword, the assignment operator, and identifiers. Note that the lexer cannot distinguish between syntax elements that share the same regex for their token type, such as variable identifiers and macro identifiers. This distinction must be done by the parser.

If the generated lexer encounters an input sequence that cannot be matched by any regular expression from the lexicon, it produces an error message containing the position of the first unexpected character in the **unrecognised** token as shown in Listing 3.4.

In addition, the lexer exposes an interface that allows the parser to pass an error. The lexer then adds information about its current position to the error. This allows the parser to produce errors containing a source code position, without the need to keep track of the source code position in the parser itself.

Listing 3.3: Example Lexicon

```
"while"                      { return token{text, WHILE} }
":="                         { return token{text, ASSIGN} }
[a-zA-Z_]([a-zA-Z_]|[0-9])* { return token{text, IDENTIFIER} }
```

This example is only an extract of the lexicon we use in the interpreter. For the full lexicon, see the source code.

Listing 3.4: Example Lexer Error

```
out: o0

o0 :- 5;
o0 := o0 * 2
```

```
Unrecognised token in line 3, pos 4:
o0 :- 5;
    ^ unexpected character
```

3.4.2 Parser

Definition 3.5 (Parser)**.** A *parser* takes a sequence of tokens as input and organises it into an AST according to a syntax definition.

The most intuitive approach for defining a parser is to supply a syntax definition in Backus-Naur Form (BNF). A production rule in BNF is written as $N ::= \alpha$, where N is a non-terminal symbol and α is a sequence of terminal and non-terminal symbols or an empty sequence. All terminal symbols are token types. An AST for a BNF syntax definition can be constructed by creating the root node representing the start symbol of the definition. For each symbol on the right hand side of the matching production rule, a child is added to the node representing the left hand side. If the symbol from the right hand side is a terminal symbol, a token of the type specified by this symbol must be read from the lexer. The last step is repeated until all leaves in the AST represent terminal symbols.

The tool `Goyacc` [2] generates a parser from a BNF. In order to create a tree each BNF production rule can be annotated with `Go` code as seen in Listing 3.5. This piece of code is executed when its production rule matches. The code can return an AST node by storing it in `$$`. The code for a production rule with n symbols on the right hand side receives the parameters `$1` through `$n`. If symbol i is a terminal symbol, `$i` contains an AST node, wrapping the token returned by the lexer for this terminal symbol. Otherwise `$i` contains the value stored in `$$` by the annotated code for the production rule of this non-terminal symbol. This code constructs an AST by adding `$1` through `$n` as children to `$$`. We will go further into the construction of the AST in Subsection 3.4.3.

Listing 3.5: Simplified Example Production Rule for Goyacc

```
assignment: variable ":=" expression
              {
                $$ = astNode{type: ASSIGNMENT, line: $1.line}
                $$.children = {$1, $3}
              }
          ;
```

Listing 3.6: Example Parser Error

```
out: o0

o0 := 5
o0 := o0 * 2
```

```
unexpected identifier, expected ';' in line 4, pos 1.
o0:= o0 * 2
^ unexpected token
```

The parser produces an error if it receives a token from the lexer, which can not be matched by any production rule in the BNF syntax definition. The error informs the programmer that an **unexpected** token was encountered. The parser can pass this error to the lexer, allowing it to add the current source code position to the error message. The parser can also make attempts to guess a solution for an error, e.g. if an error occurs after a complete instruction, the parser can suggest that a semicolon is missing before the unrecognised token (Listing 3.6). Producing meaningful parser error messages for *all* possible errors is not a generally solved problem and goes beyond the scope of this thesis.

3.4.3 Abstract Syntax Tree

In the AST most syntactic symbols are represented as tree nodes. Terminal symbols are represented by leaves, non-terminal symbols by inner nodes or leaves, if its BNF production rule has an empty right hand side. Leaf nodes created from non-terminal symbols are omitted in the following examples. Some symbols, e.g. semicolons, serve no purpose after parsing and therefore are not represented at all in the AST. Sequencing of instructions in a program with semicolons, as shown in Figure 2.5, creates a degenerate tree. Instead, we flatten instruction sequences into children of one single `program` node, see Figure 3.7. For brevity we will only discuss a representative example here. For a full list of syntactic symbols and their translation into AST nodes view the source of the parser.

Figure 3.8 shows an example program and the AST generated from it. The AST has one scope node, called the *root scope*. This scope limits visibility of variables to the partial tree below it, as described in Definition A.6 on page 44. We will only need this scope later in Subsection 3.4.6, to support macros. Each instruction of the LOOP/WHILE program is represented by a partial tree and grouped with other instructions to form a program. Note that the `enddo` node is only necessary for the debugger, so that execution can be stopped after the last iteration of a loop, see Section 3.5.

```
instruction1;
instruction2
```

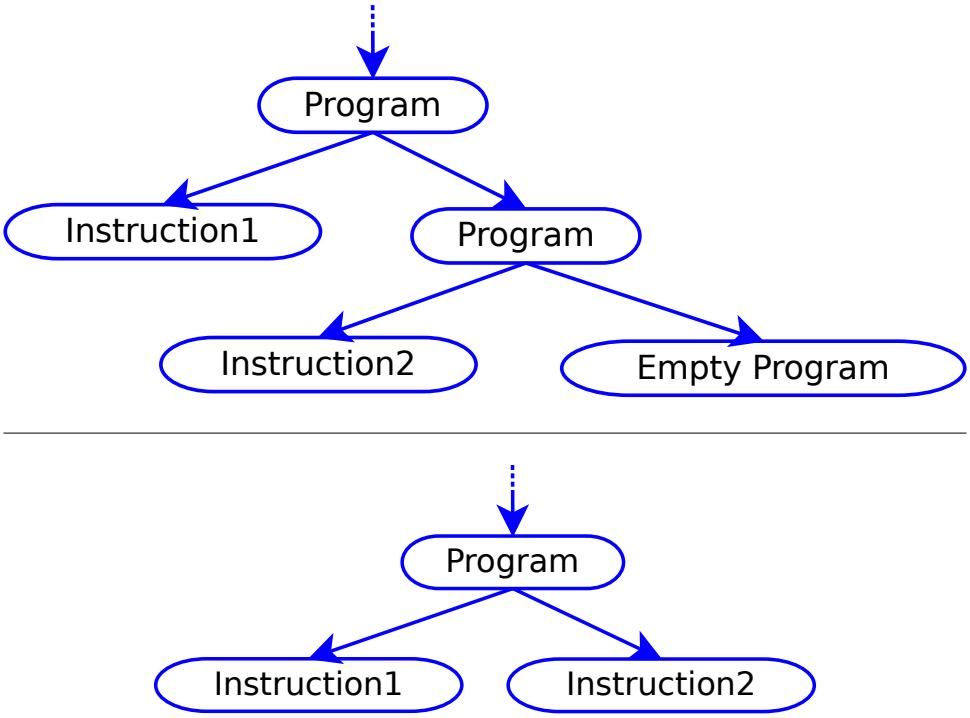


Figure 3.7: Example for Flattened Instruction Sequences

3.4.4 Variable Bindings

The interpreter has to keep track of all variables and their assigned values. This can be done through a global *variable binding* map of `variable identifier` to `value`. This is sufficient for interpretation of a LOOP/WHILE program, but requires variable renaming in macros as described in Definition A.45. It is unsuitable for a debugger because all alterations in the executed code have to be reverted when reporting execution positions and variable bindings. Instead we group the variable bindings into scopes.

Definition 3.6 (Dynamic Scope)**.** A *dynamic scope* is a triple (P, B, N) that consists of a source code position P, a map of variable bindings $B : identifier \to \mathbb{N}$ and a reference N to the scope AST node (Definition A.6) of the macro which this scope belongs to.

Note that there are three types of scopes: static scopes as described in Definition A.6, AST scope nodes which represent a static scope in an AST and dynamic scopes used by the interpreter to keep track of variable bindings and the execution position.

3.4.5 Execution

The execution operates directly on the AST. Execution starts at the root scope node. The interpreter keeps track of the state described in Definition 2.1. It then iterates over all instruction trees below the program node child of this scope. Figure 3.9 shows in green the path which the execution takes through an AST. Figure 3.10 through Figure 3.14 show the path through the AST taken by the interpreter to simulate execution of each instruction type. Listing 3.15 and Listing 3.16 additionally show simulation pseudocode for the assignment and loop instructions used in the example.

```
out: o0

o0 := 5;
loop o0 do
        o0 := o0 * 2
enddo
```

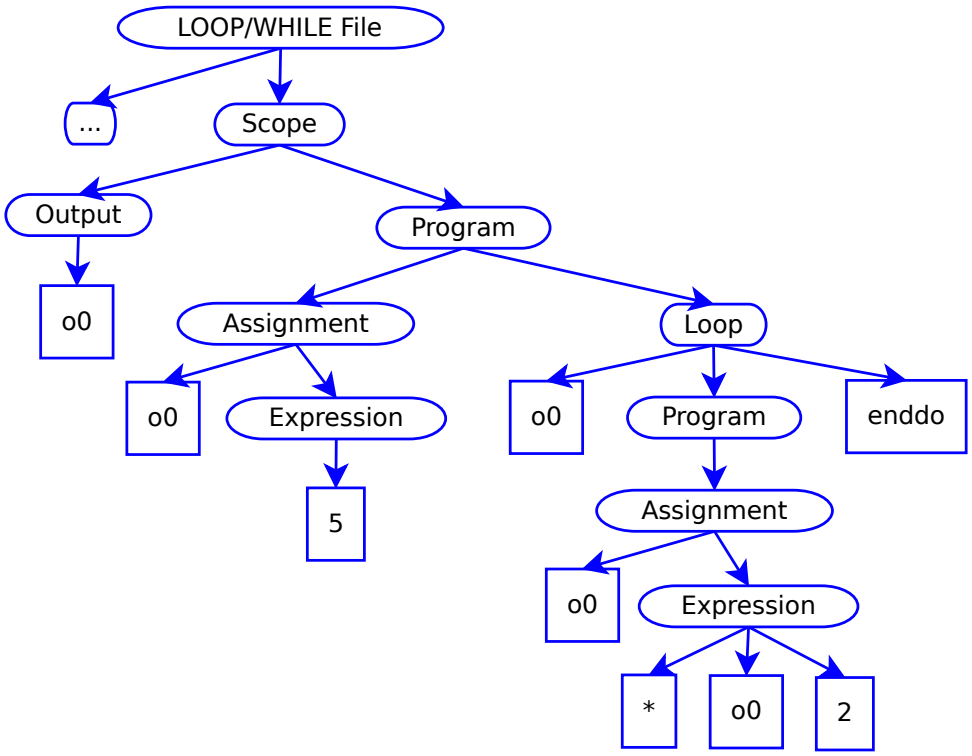
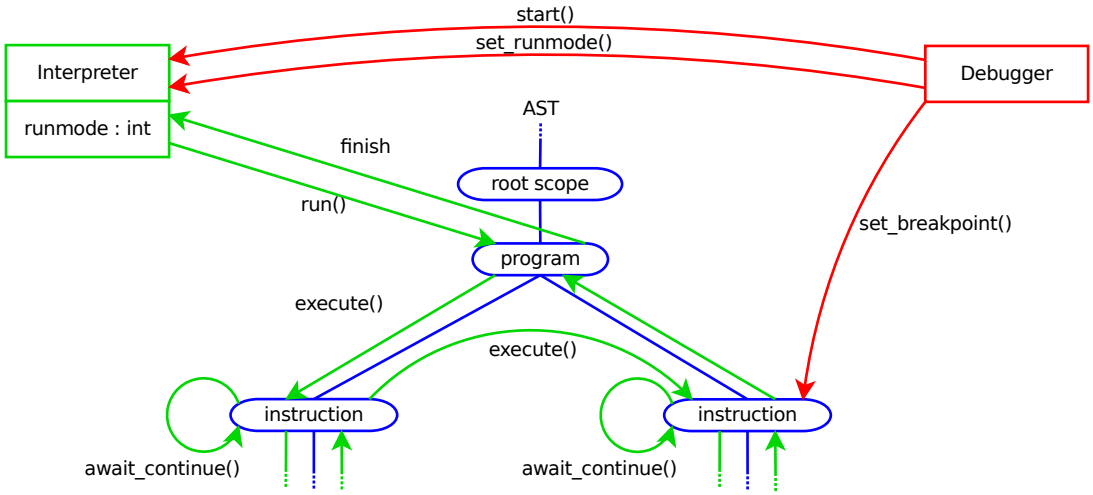
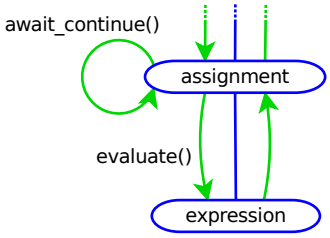


Figure 3.8: Example AST



Blue edges and nodes represent the AST of a LOOP/WHILE file (Definition A.1). Red arrows show intervention of the debugger. Green arrows, followed counterclockwise around the tree, illustrate the path taken by the interpreter when executing the AST. Execution paths for all instruction types are shown in Figure 3.10 through Figure 3.14

Figure 3.9: Interpreter Execution: Generic

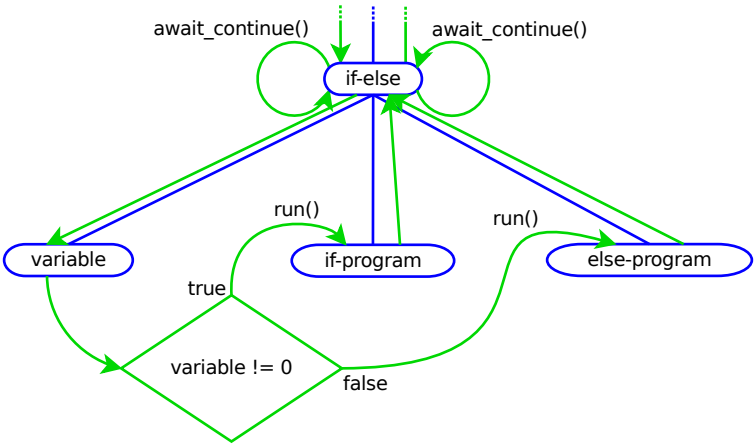


See Figure 3.9 for context.

Figure 3.10: Interpreter Execution: Assignment

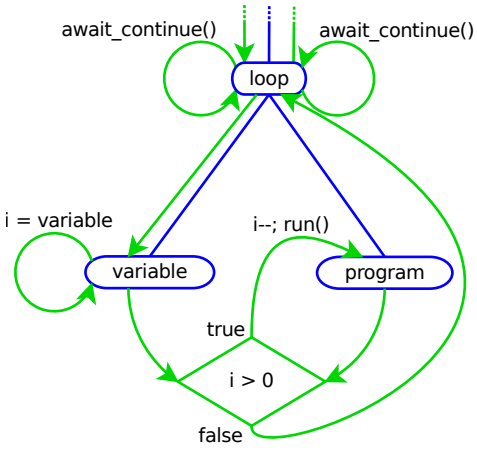
3.4.6 Macros

In order to support macros, as described in Section A.6 and /autorefmacrocallsec, the interpreter could apply their textual substitution rules to produce a LOOP/WHILE program which can then be treated as discussed in the previous sections. However, this would introduce a difference between the code being executed and the code written by the user. When we later implement a debugger in Section 3.5, it would have to revert those changes, in order to present the state of execution at a breakpoint to the user. Instead we will treat macros like functions, i.e. inside a macro the interpreter hide all variables of the surrounding program and only operate on the variables declared within the macro. This is done by creating a new scope corresponding to this macro's scope node when execution enters a macro and copying input values to the macro's input variables. When enters leaves the macro, we use the value of its output variable in the assignment in which the macro was called. Finally we discard the new scope and continue execution with the old scope. Since this separation of scopes is only necessary to provide accurate debugging information, we will go into its detailed implementation in Subsection 3.5.1 of the Debugger Implementation Section. Figure 3.17 shows an example program containing macros and the AST generated from it. Note that the AST root and every macro have their own scope node.



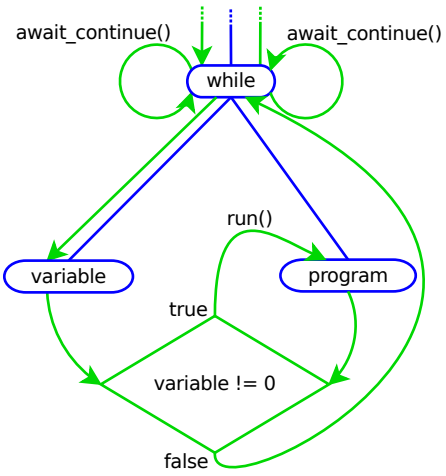
See Figure 3.9 for context.

Figure 3.11: Interpreter Execution: If-Else



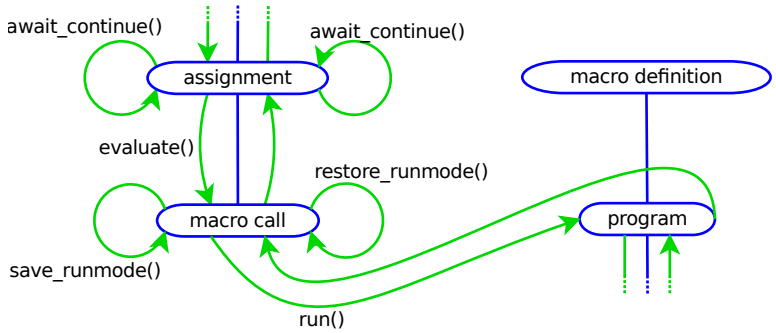
See Figure 3.9 for context.

Figure 3.12: Interpreter Execution: Loop



See Figure 3.9 for context.

Figure 3.13: Interpreter Execution: While



See Figure 3.9 for context.

Figure 3.14: Interpreter Execution: Assignment with Macro Call

Listing 3.15: Simulation of an assignment instruction

```
function execute(assignment) {
        variable = variable type child node of assignment
        expression = expression type child node of assignment
        store evaluate(expression) in variable
}
```

Listing 3.16: Simulation of a loop instruction

```
function execute(loop) {
        variable = variable type child node of loop
        program = program type child node of loop

        for(i = copy value(variable); i>0; i--) {
                run(program)
        }
}
```

```
def macro0
        out: o0
    ...;
enddef

def macro1
        out: o0
        ...;
enddef

out: o0
...;
```

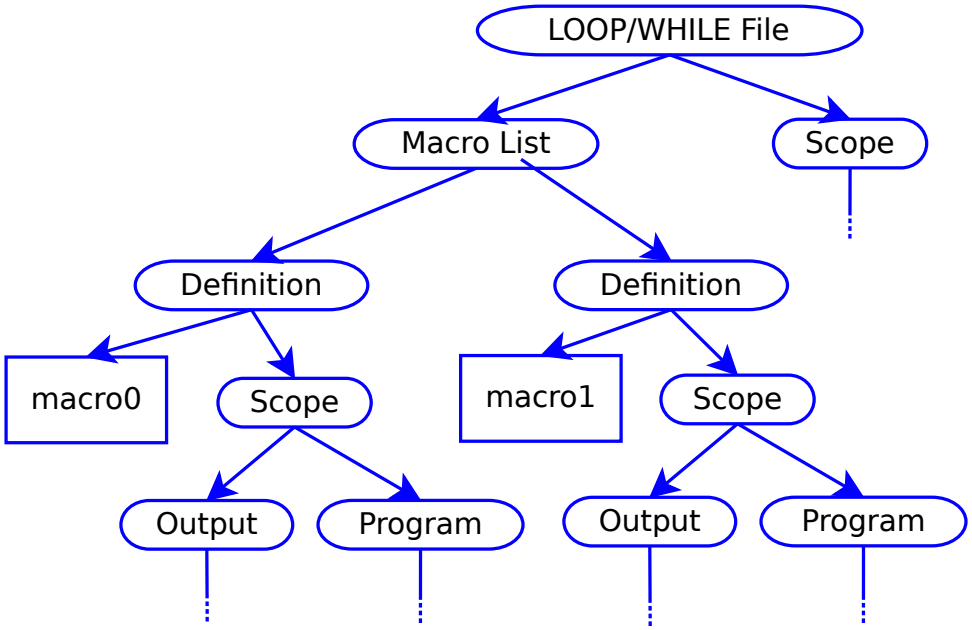


Figure 3.17: Example AST with Macros

3.4.7 Recursive Macro Checking

Macros must not be recursive, as discussed in Section A.6. This means no macro can call itself directly or through calling other macros. We can create a directed graph from macro calls inside macros, as shown in Figure 3.19. In this *macro dependency graph* each macro definition is represented by one node. For every macro call inside a macro, a directed edge is added to the graph, from the caller node to the callee node. If there exists a macro that calls itself through any chain of macro calls, then the macro dependency graph contains a cycle. Implementing an algorithm to check for cycles in a directed graph is standard and goes beyond the scope of this thesis.

3.5 Debugger Implementation

The main application of this project is not just to execute LOOP or WHILE code, but also to allow the programmer to look into every step the interpreter takes when executing a program. We enable this through a debugger which can set, remove, and stop at breakpoints and single step through execution. At every stop the debugger shows the current position of the execution along with all variables and their values.

Listing 3.18: Iterative Integer Square Root

```
def sqrt
    in: i0
    out: o0

    o0 := sqrt_helper(i0, i0);
enddef

def sqrt_helper
    in: x0, n
    out: x1
    aux: fin

    x1 := n div x0;
    x1 := x0 + x1;
    x1 := x1 div 2;

    fin := x0 != x1;
    if fin != 0 then
        x1 := sqrt_helper(x1, n);
    endif;
enddef

in: i0
out: o0

o0 := sqrt(i0);
```

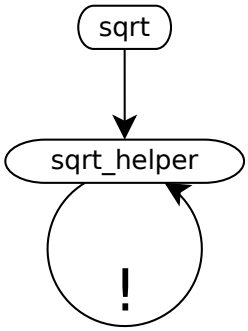


Figure 3.19: Example Macro Dependency Graph

Listing 3.20: Iterative Integer Square Root

```
1  def macro2
2      in: i0
3      out: o0
4
5       o0 := i0 / 2;
6  enddef
7
8  in: i0
9  out: o0
10
11 o0 := macro2(i0);
12 if o0 != 0 then
13     o0 := macro2(o0);
14 endif
```

Figure 3.21: Example with multiple Macro Calls

Figure 3.14 shows the path of execution for a macro call. Note that execution jumps from the AST subtree which contains the call to the subtree of the macro definition. In this case the execution position cannot only consist of the line number in the source code because the same position could be reached from different macro calls. In Figure 3.21 the macro `macro2` is called several times. If we stop execution at line 5, we also need to know where this macro is called from. Not only positions inside a macro pose a problem, also variable identifiers may overlap with those outside of the macro. We can solve both problems by using a stack of scopes.

3.5.1 Scope Stack

When execution enters a macro, we push a new *scope* onto a stack, called *scope stack*, and remove it, when execution leaves the macro again. The scope on top of the stack is called the *active scope*.

Recall that a scope is a triple (P, B, N), as described in Definition 3.6. The source position P always points to an instruction below the node N belonging to this scope. During a macro call execution leaves the subtree under the current scope, as shown in Figure 3.14. We create a new scope for the called macro and leave the position in the previous scope unchanged until the execution returns from the macro call. We achieve this by only updating the position in the scope on top of the stack. Figure B.33 illustrates this behaviour, where each `stack trace` element corresponds to a scope on the scope stack.

The variable bindings B are treated similarly. Assignments inside a macro only affect the variables in the scope of this macro, as described in Definition A.9. Values for input variables of a macro are copied from the previously active scope into the new scope when executing a macro call. Values of output variables are copied back from the scope of the macro to the scope in which the macro call happened.

3.5.2 Breakpoints

The debugger can set breakpoints via marking an instruction node in the AST, illustrated by the `set_breakpoint()` call in Figure 3.9. The interpreter checks this breakpoint marker with

the `await_continue()` call and pauses execution in front of the instruction if the marker is set.

If execution is paused the debugger can send a signal to wake up the interpreter. It then resumes execution until the next breakpoint.

3.5.3 Stepping

The debugger can put the interpreter into stepping mode through the `set_runmode()` call, shown in Figure 3.9. Possible *run modes* are `run`, `pause`, `step into`, `step over`, and `step out`.

When the interpreter issues an `await_continue()` call, as depicted by Figure 3.9 through Figure 3.14, it checks for a breakpoint marker and the current `run mode` and acts accordingly.

- `run`: Continue execution.
- `breakpoint set`, `pause`, `step into/over/out`: Pause execution and wait for a command from the debugger.

If execution is paused, we can send a `resume`, `step into`, `step over`, or `step out` command to the interpreter. Each command sets the `run mode` to `run`, `step into`, `step over` and `step out` respectively and continues execution. Note that `step into`, `step over`, and `step out` each name a command as well as a `run mode`. In the following each refers to the command, unless it is explicitly stated, that the `run mode` is set to that value.

- `resume`: Resume normal execution, only stop at next breakpoint.
- `step into`: Execute the current instruction node and stop at the next one.
- `step over`: Same as `step into`, unless the current instruction is a macro call. In this case execute the entire macro and stop before the first instruction after the call.
- `step out`: Execute the remainder of the current macro and stop before the first instruction after the call to it.

We will take a closer look at stepping over and out of macros. Implementation of this behaviour gets slightly more complex when execution can reach a breakpoint while taking a step.

For `stepping over`, the interpreter can store the current `run mode` (`save_runmode()`) at the node of a macro call, set it to `run`, resume normal execution until it is back at this macro call node and restore it (`restore_runmode()`), as illustrated by Figure 3.14. Only if a breakpoint is reached intermediately, the interpreter should not pause when returning to the macro call node. To do this, the interpreter registers the step over mark in a global list. When the interpreter reaches a breakpoint, it sets a notification flag on all entries in this list. Upon returning to the macro call, the interpreter checks the matching notification flag and only pauses if the flag is not set.

`Stepping out` behaves like stepping over for all instructions in the current macro and pauses if execution is at the end of a macro after a step. This is only the case at the end of the current macro, not at the end of intermediate macro calls, as the `run mode` is reset to `run` for the duration of the intermediate macros.

3.5.4 Serialisation

Whenever execution is paused, the debugger serialises the scope stack (Subsection 3.5.1) into a `JSON` string. This string consists of a list of quadruples (F, L, M, B) each representing one scope,

where M is `root` or the macro to which the scope belongs (see Subsection 3.4.3), F is the path to the file containing the macro M, L is the line at which the execution inside M currently is and B is a map of variable bindings as described in Definition 3.6, with each variable annotated whether it is a input, output or auxiliary variable. Listing 3.22 shows the serialised scope stack corresponding to Figure B.33.

Listing 3.22: Example Serialised Scope Stack

```
[
        {"file":"/home/[...]/src/loop_while_exercises.lw",
         "line":6,
         "macro":"exp",
         "bindings":[
                {"VarType":"input",
                 "Ident":"base",
                 "Val":5
                },
                {"VarType":"input",
                 "Ident":"exp",
                 "Val":7
                },
                {"VarType":"output",
                 "Ident":"res",
                 "Val":1
                }
         ]
        },
        {"file":"/home/[...]/src/loop_while_exercises.lw",
         "line":13,
         "macro":"root",
         "bindings":[
                {"VarType":"input",
                 "Ident":"i0",
                 "Val":5
                },
                {"VarType":"input",
                 "Ident":"i1",
                 "Val":7
                },
                {"VarType":"output",
                 "Ident":"o0",
                 "Val":0
                }
         ]
        }
]
```

4 Integrated Development Environment

So far we have discussed the tools working in the background for the programmer. In this chapter we will focus on the user interface providing access to these tools. We will discuss implementation of this user interface as a plugin to the IntelliJ IDEA *integrated development environment (IDE)* [8].

4.1 Necessity

An IDE can combine the functionality of our interpreter and debugger, with that of an editor. This allows us to provide easily understandable errors and feedback to programmers by embedding error messages and debugging information into the source code shown in the editor. The immediate feedback on syntax errors shown while typing in new code greatly assist learning the syntax of a new programming language. When using the debugger, highlighting the current position in the editor allows the programmer to follow the execution, without the need to manually look up source positions.

In order to use the wide range of features existing IDEs have, we build a plugin for IntelliJ, instead of implementing our own IDE.

4.2 IntelliJ Plugin

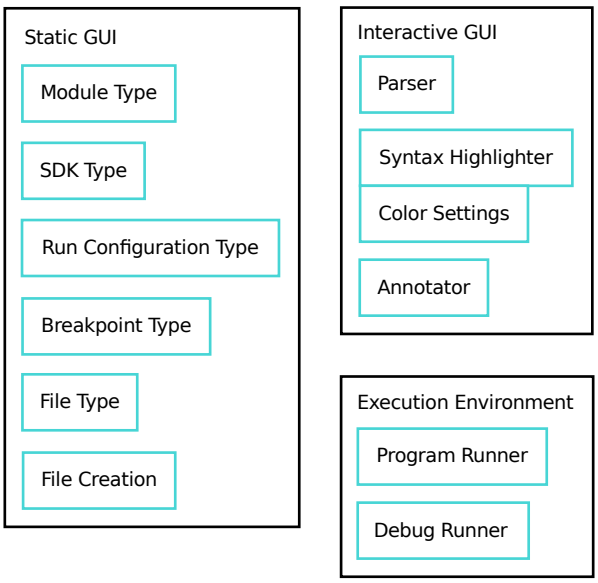
IntelliJ is a widely used IDE for Java with differently named versions for many other languages, e.g. C++, Python. IntelliJ provides a plugin API which allows us to use all existing features of the IDE. Using this API we only need to implement language specific interfaces.

An IntelliJ language plugin consists of an XML file stating which parts of the plugin API are implemented and, with some exceptions, Java classes implementing these API parts. The API ranges from a replication of the language grammar, in order to provide syntax highlighting, to a server sending commands to our debugger through a TCP socket. Figure 4.1 shows the API elements we implemented, grouped by their functionality.

- Static GUI: This group handles the non-interactive graphical appearance of our plugin, such as icons and buttons labels.
- Interactive GUI: This group provides dynamic feedback on user input, such as highlighting errors and correct syntax.
- Execution Environment: This group handles communication between the IDE and the interpreter/debugger.

For details on the static GUI group, see the source code.

The core of the interactive GUI group is a `parser` which is a duplicate of the parser used in the interpreter, built with IntelliJ's own parser generator. The `syntax highlighter` provides a mapping from syntax elements produced by the parser to text colors, in order to make the LOOP/WHILE code more readable. IntelliJ invokes the parser with every new input in its



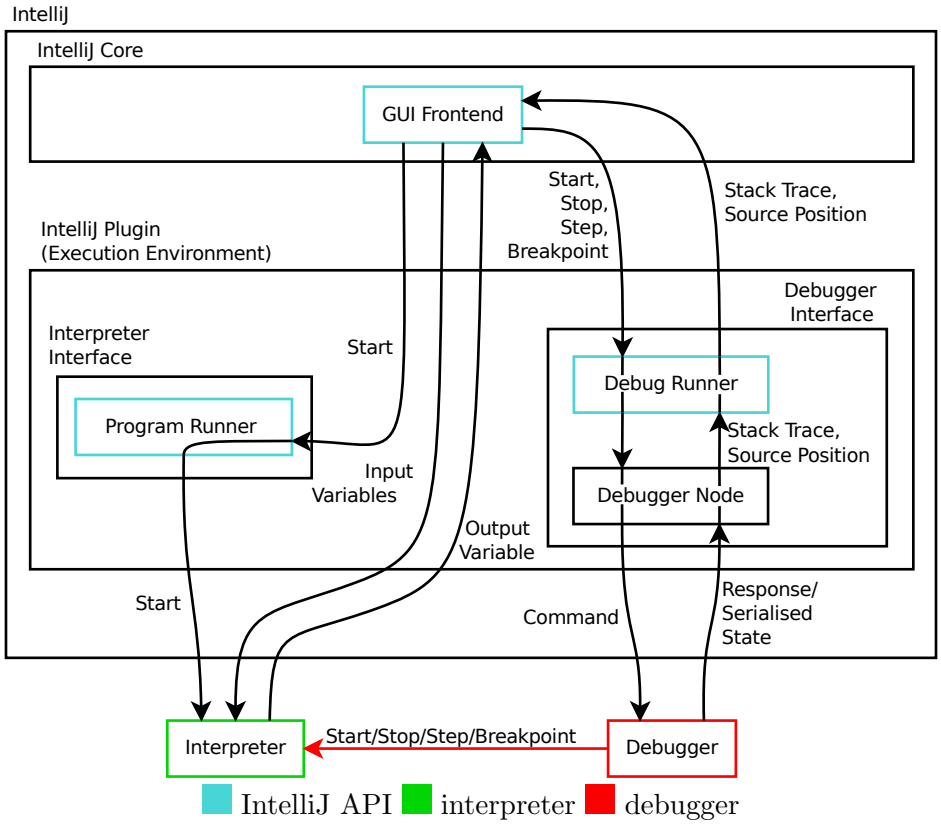
This Figure show the elements of IntelliJ's API which our plugin implements. In addition to green and red highlighted interpreter and debugger components, shown in Figure 3.9 through Figure 3.14, Figure 4.1 through Figure 4.6 show elements highlighted in turquoise. These are parts of IntelliJ's API.

Figure 4.1: IntelliJ Plugin API

editor. If the parser encounters an error, the faulty source code is marked while the user types. For the user's convenience, all colors used by the highlighter can be customised in a `color settings` page. In addition to syntactic checks, the `annotator` can be used for semantic checks. IntelliJ invokes the `annotator` on every successfully parsed input token. We use the `annotator` to mark undeclared variables by walking the AST upwards until we encounter a scope node and check if a declaration for the variable identifier in the input token exists.

The execution environment group is, despite implementing only two API elements, the most complex of the three. We will discuss it in detail in the following section.

Due to lack of documentation for IntelliJ language plugins, the source code for our plugin is in large parts taken from an existing plugin for the programming language Erlang [7] and modified to work with LOOP and WHILE.



This Figure shows interaction of IntelliJ's core with the interpreter and debugger through the execution environment part of the plugin, shown in Figure 4.1. All queries from IntelliJ originate in its GUI. We discuss the various interactions in detail in Subsection 4.3.1 and Subsection 4.3.2.

Figure 4.2: IntelliJ API: Execution Environment

4.3 Execution Environment

The task of the execution environment is to pass input and commands from the GUI to our interpreter and present the responses in the GUI. Figure 4.2 shows the simplified interactions of the plugin elements, the GUI, the interpreter and the debugger. The elements in the plugin box will be explained in Subsection 4.3.1 and Subsection 4.3.2. The execution environment interacts with various elements of IntelliJ's GUI frontend.

- The `editor` contains the source code for the execution. We pass this code to the interpreter and mark the current execution position at breakpoints in the editor.
- The `debugger menu` contains buttons to issue debugger commands, such as `start`, `stop`, `step` and `set breakpoint`, as well as a window to show a `stack trace` and `variable bindings`. We translate the debugger commands from IntelliJ's internal representation to the command strings which our debugger understands and translate `serialised states` from the debugger into IntelliJ's internal `stack trace` data type.
- The `run configuration` settings page allows the user to set the path to the interpreter executable used by our plugin, and the command line arguments which we pass to the interpreter.

Figure 4.5 shows IntelliJ's GUI and highlights parts that interact with our plugin.

Listing 4.3: LOOP File Template

```
#LOOP

in: i0;
out: o0;

o0 := i0;
```

Listing 4.4: WHILE File Template

```
#WHILE

in: i0;
out: o0;

o0 := i0;
```

4.3.1 Interpreter Interface

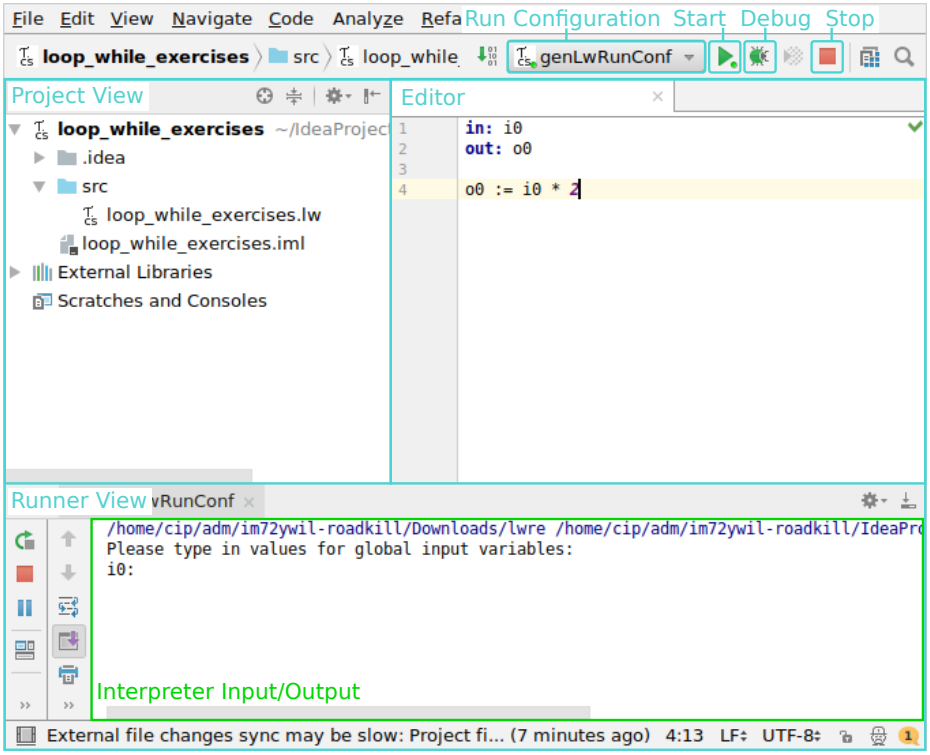
The `interpreter interface`, shown in Figure 4.2, is responsible for starting our interpreter and setting up its visual representation in IntelliJ's GUI. It assembles the command line of the interpreter from the run-configuration, stored in the `program runner`, starts execution and connects input and output to the `runner view` (Figure 4.5). When the interpreter now asks for input variable values or prints the output variable, it directly interacts with the GUI.

In order to build the command line, we simply take the executable path and command line arguments from the run-configuration, but we also have to find a LOOP/WHILE file to execute. Many bigger programming languages, e.g. `C` or `go`, have a concept of a `main` function which is the starting point for execution. This `main` function is typically only present once per project and *running a project* means *running the main function*. Similarly, LOOP/WHILE files contain either only macro definitions or macro definitions and exactly one functions which can be viewed as its `main` function. However our use case for a project is not to have a large collection of macro definitions which form one single executable, but rather to have a collection of stand-alone exercises where each LOOP/WHILE file solves a certain task and can be run on its own. Therefore we need to determine which file to execute. We leave this choice to the user by executing the file which is currently open in IntelliJ's editor. If there is no suitable open file, we ask the user to open one.

4.3.2 Debugger Interface

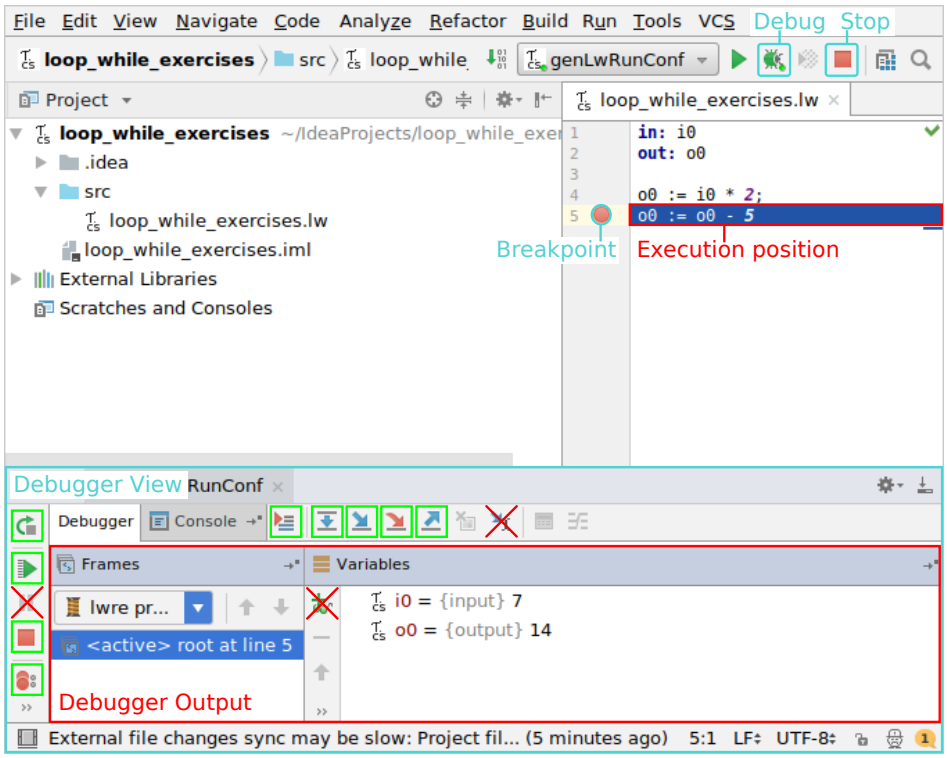
The `debugger interface` handles communication between IntelliJ's debug menu (Figure 4.6) and our debugger. It translates IntelliJ's debug commands into commands understood by our debugger and converts the serialised state from our debugger into IntelliJ's internal `stack trace` format.

IntelliJ calls functions of our plugin representing debugger commands, e.g. `set breakpoint`, and passes a source position as parameter if it is required for the command. Our plugin has a predefined string for each debugger command, e.g. `"setbreakpoint"`, concatenates this with the ASCII representation of the line number from the source position and enqueues this string to be sent to our debugger.



- The `project view` lists the files in the current project. A user can add, open and delete files in this view. When the user adds a file, IntelliJ calls our plugin through the `file creation` interface shown in Figure 4.1. Our plugin then creates a new LOOP/WHILE file from a template. We provide an empty template and the two shown in Listing 4.3 and Listing 4.4.
- The editor lets the user alter LOOP/WHILE files in the current project. The file in focus in the editor is also the one being run, when the user presses the `start` button.
- The `runner view` displays input and output directly from our interpreter.
- The `run configuration` button opens a menu to edit the parameters passed to the interpreter and the path to its executable.
- The `start`, `debug` and `stop` buttons trigger the plugin to execute or terminate interpreter and debugger.

Figure 4.5: IntelliJ Editing/Running GUI



IntelliJ API Debugger Button implemented Button not implemented The debugger can be started and stopped with the `debug` and `stop` buttons. During debugging, the `runner view`, seen in Figure 4.5, is replaced by a `debugger view`.

- Input and output from the interpreter is then shown in the `console` tab of the `debugger view`.
- The `debugger` tab shows a `stack trace` and `variable bindings` as described in detail in Subsection 4.3.2.
- The execution position and breakpoints are shown in the editor.
- Features of IntelliJ's debugger interface which are supported by our debugger, are marked in green. Ones that require further work are marked in red.

Figure 4.6: IntelliJ Debugging GUI

When the debugger responds to one of IntelliJ's debug commands, it either states that execution has been resumed or that it has been stopped.

If execution is resumed after the command, our plugin informs IntelliJ that the program is running again. IntelliJ then hides any previously shown `stack traces` or `variable bindings` and instead displays a `"program is running"` message.

If execution is stopped after a debug command, the debugger also sends a `JSON` encoded serialisation of the interpreter's execution state. The plugin then decodes this `JSON` string into a temporary Java object which closely resembles the internal data type in our interpreter. We use this temporary Java object to construct a `stack trace` object of IntelliJ's internal data type. Our plugin passes that object to IntelliJ, which presents the `stack trace` along with the contained `variable bindings` to the user.

5 Related Projects

We set out to provide an intuitive programming environment for the LOOP and WHILE languages. Alexander Dietsch created a project called `lw-simulator` [4] which addresses similar issues.

The simulator is a standalone interpreter like ours, but its implementation follows closer to Meyer and Ritchie's register machine than we do. The simulator enforces register names (`x0` through `xn`) in the root scope (Subsection 3.4.3). The output register is hard-wired to `x0` . Input registers are not marked as such, instead the user can edit the initial value of all registers, before starting execution. Input, output and auxiliary registers of macros must have the identifiers `i0` through `i0` , `o0` through `on` and `a0` through `an` respectively. These register names do not correspond to any actual registers, instead every macro call assigns a `xi` register to each register used in the macro. Our interpreter improved on this by operating on variables. This especially means, that variable names can be freely chosen and macros can declare new variables without the need to pass a register name for every internal register.

The simulator checks for recursive macros at runtime, which means syntactically incorrect programs only trigger an error if the recursive code path is taken with the given input. Our interpreter does all checks before execution. This makes it easier for a user to understand errors, since any errors will always appear regardless of input values for the program.

The simulator has no debugging functionality, instead it allows the user to set a stepping speed, i.e. it waits for a specified amount of time after every instruction, as well as pausing at any time. When the simulator is paused it shows the execution position in the editor and the user can edit values stored in all registers. Our interpreter includes a debugger which allows setting breakpoints at any line of code, single stepping and stepping over and out of macros. When the interpreter is paused, it also shows the current execution position in the editor, but also includes a stack trace showing the chain of macro calls that led to the current position. Variables can also be edited and are grouped by macro scope, to provide an overview over which variables are used in the current scope.

Finally the simulator includes a self-made GUI which lacks responsiveness in syntax highlighting has no instant error feedback. Our GUI is `IntelliJ` which comes with a wide range of features, including instant syntax highlighting and marking of syntax errors as the user types.

6 Conclusion

As discussed in Section 5, our interpreter exceeds previous projects in three aspects. The first aspect is syntactic extensions to LOOP and WHILE, especially the ability to give arbitrary identifiers to variables and macro support. The second aspect is `IntelliJ` as a well-known and responsive front-end. The third aspect is a set of debugging functionalities, consisting of setting breakpoints, stepping, viewing stack traces and editing variable values at runtime. The combination of these features allow users to focus on implementing algorithmic exercises without being held back the simplicity of these languages.

In future work our interpreter and plugin could still be extended with various features. The plugin can be extended with further `IntelliJ` API parts, such as refactoring for variable names. One could also implement the remaining debugging features of `IntelliJ` in our debugger, these are running to the current cursor position, pausing at any time without a breakpoint and setting watch-points on variables, to stop if the value of that variable changes. The command line interface of the debugger could be equipped with completion for debug commands and human-readable stack trace output.

A Extended Syntax

In this appendix, we give informal definitions of the semantics of all extended LOOP/WHILE syntax elements, in the order in which they may appear in a syntax tree. We also give textual substitution rules to transform all LOOP/WHILE programs in extended syntax into basic LOOP or WHILE programs, in order to show that these extensions do not alter the computability of both languages.

A.1 LOOP/WHILE File

Definition A.1. A *LOOP/WHILE file* consists of a language constraint (Definition A.2), a list of imports (Definition A.4), a list of macro definitions (Definition A.9), variable declarations (Definition A.7), and finally a LOOP or WHILE program.

No additional textual substitution is necessary to transform a LOOP or WHILE file into a traditional LOOP or WHILE program if all elements of the file are already transformed.

A.2 Language Constraint

Definition A.2 (Language Constraint)**.** The `#LOOP` and `#WHILE` flags declare whether the given file is a LOOP or a WHILE file. If present, the language flag must be the first statement in the LOOP or WHILE file.

Definition A.3 (Language Constraint: Substitution Rule)**.** If the file starts with `#LOOP`, but contains at least one `while X != 0 do P enddo` instruction, the file is invalid. Otherwise remove the language constraint.

A.3 Import

Definition A.4 (Import)**.** `#IMPORT <path>` allows importing macro definitions (Definition A.9) from other LOOP/WHILE files. `<path>` must be a valid path to a file in the file system. All macros defined in the file at `<path>` may be called from this file. Imports must be substituted before checking the language constraint (Definition A.2), before checking validity of macro definitions (Definition A.9) and before substituting any macro calls (Definition A.45). Imports are allowed to be nested. All imports that have already been substituted once in this LOOP/WHILE file will be ignored. Imports must occur right after the optional language constraint and before any macro definitions.

Definition A.5 (Import: Substitution Rule)**.** If the `<path>` is not a valid path in the file system, the LOOP/WHILE file is invalid. If the file at `<path>` is not a valid LOOP/WHILE file, this file is also invalid. If the file at `<path>` has already been imported into this file, remove this import. Otherwise replace `#IMPORT <path>` by all imports in the file at `<path>`,

followed by all macro definitions from that file. Apply this substitution rule to all remaining and new imports.

A.4 Scope

Definition A.6 (Static Scope)**.** A static scope is a 4-tuple (I, O, A, P) that consists of an output variable declaration O, a LOOP/WHILE program P, and an input and an auxiliary variable declaration I and A. I and A can also be empty. The variable declarations I, O, or A are only binding within P.

A.5 Variable Declaration

Definition A.7 (Variable Declaration)**.** Every LOOP or WHILE file must contain a declaration of the output variable and may contain declarations of input and auxiliary variables before the first instruction. Input and output variables are treated in the sense of Definition 2.4. All variables that are neither input nor output must be declared as auxiliary variables.

- `in: Xi1, Xi2, ... Xin` : Declares `Xi1` through `Xin` as input variables, with $n \in \mathbb{N}$.
- `out: Xo` : Declares `Xo` as output variable.
- `aux: Xa1, Xa2, ... Xam` : Declares `Xa1` through `Xam` as auxiliary variables, with $m \in \mathbb{N}$.

All variables used in the file must occur in exactly one of these three declarations within the scope (Definition A.6) of the LOOP/WHILE program they are part of. Variable identifiers do not have to be enumerated the way they are in this example, instead arbitrary identifiers are allowed, if they do not conflict with keywords of LOOP and WHILE.

Definition A.8 (Variable Declaration: Substitution Rule)**.** If the file contains any variables that are not declared in the matching scope, the file is invalid. Otherwise remove the declarations.

A.6 Macro Definition

A macro defines a LOOP or WHILE program that can be reused by calling the macro (Definition A.45) to avoid copying code.

Definition A.9 (Macro Definition)**.** `def F D P enddef` : Defines a macro `F` with variable declarations `D`
(`in: Xi1, Xi2, ... Xin out: Xo aux: Xa1, Xa2, ... Xam`), such that a
macro call `F(X1, X2, ... Xn)` results in the value x_o, where x_o is the value of `Xo` after execution of the program `P`, with the contents of `X1` through `Xn` copied to `Xi1` through `Xin`. `P` must only contain variables declared in `D` in the sense of Definition A.7. Variables declared in `D` are only valid in the scope of `F`, i.e. a variable `X` declared in `D` is a separate variable from the variable `X` declared in the file or macro from which `F` is called. Therefore altering the value of `X` within `P` does not alter the value of `X` outside of the macro `F`.

Textual substitution on a recursive macro would lead to an infinitely large LOOP/WHILE program, therefore we do not allow macros to contain calls to themselves, directly or through a chain of other macros.

Listing A.1: Conditional: Substitution

```
Y1 := 0;
Y2 := 0;
Y2 := Y2 + 1;
loop X do
        Y2 := 0;
        Y1 := Y1 + 1
enddo;
loop Y1 do
        P1
enddo;
loop Y2 do
        P2
enddo
```

Definition A.10 (Macro Definition: Substitution Rule)**.** If the macro contains a variable that is not declared in the macro's variable declarations, the entire LOOP/WHILE file is invalid. If the macro is recursive, the file is invalid. If there is more than one macro definition for the name `F`, this file is invalid. Otherwise substitute all calls to this macro as described in Definition A.46. After all calls to this macro are substituted, remove this macro definition.

A.7 Conditional

Definition A.11 (If Else Conditional)**.** `if X != 0 then P1 else P2 endif` : Let x be the value of `X`. If x equals 0, execute `P2`, otherwise execute `P1`. A conditional may also be of the form `if X != 0 then P endif` which is treated as `if X != 0 then P else Pe endif`, where `Pe` is the empty instruction sequence.

Definition A.12 (Conditional: Substitution Rule)**.** Replace `if X != 0 then P1 else P2 endif` by the program in Listing A.1, where `Y1` and `Y2` are variables that do not occur in the LOOP/WHILE file.

A.8 Assignment

Definition A.13 (Assignment)**.** Assignments in extended syntax are `X := <expression>` where the resulting value of `<expression>` is copied into `X`. `<expression>` may be any expression defined in Section A.9.

Definition A.14 (Assignment: Substitution Rule)**.** If `<expression>` matches no expression described in Section A.9, the LOOP/WHILE file is invalid. Otherwise apply the substitution specified for the instance of `<expression>`.

A.9 Expression

Definition A.15 (Expression)**.** An expression is either the right hand side of one of the assignment instructions of the LOOP language (Definition 2.2), or one of the expressions defined in this section.

Listing A.2: Addition: Substitution

```
Y := <atom2>;
X := <atom1>;
loop Y do
        X := X + 1
enddo
```

In the following `<integer>` refers to any sequence of decimal digits, `<atom>` refers to an `<integer>` or a variable. The value stored in `<atom>` refers to the number represented by the `<integer>`, if `<atom>` is an `<integer>`, and refers to the value stored in the variable, if `<atom>` is a variable.

A.9.1 Constant

Definition A.16 (Constant)**.** `X := <integer>` : Stores the number represented by `<integer>` in `X`.

Definition A.17 (Constant: Substitution Rule)**.** Replace `X := <integer>` by `X := 0` followed by i instructions `X := X + 1`, where i is the number represented by `<integer>`.

A.9.2 Successor

Definition A.18 (Successor)**.** `X := succ(<atom>)` : Stores $a + 1$ in `X`, where a is the value stored in `<atom>`.

Definition A.19 (Successor: Substitution Rule)**.** Replace `X := succ(<atom>)` by `X := <atom>` followed by `X := X + 1`. If `<atom>` is an `<integer>`, replace the assignment of `<atom>` following the substitution rule in Definition A.17.

A.9.3 Addition

Definition A.20 (Addition)**.** `X := <atom1> + <atom2>` : Stores the sum of `<atom1>` and `<atom2>` in `X`.

Definition A.21 (Addition: Substitution Rule)**.** Replace `X := <atom1> + <atom2>` by the program in Listing A.2, where `Y` is a variable that does not occur in the LOOP/WHILE file. If either `<atom>` is an `<integer>`, replace assignments of this `<atom>` following the substitution rule in Definition A.17.

A.9.4 Predecessor

Definition A.22 (Predecessor)**.** `X := pred(<atom>)` : Stores $a \dot{-} 1$ in `X`, where a is the value stored in `<atom>` and $a \dot{-} 1$ equals $a - 1$ if $a \geqslant 1$ and is zero otherwise.

Definition A.23 (Predecessor: Substitution Rule)**.** Replace `X := pred(<atom>)` by the program in Listing A.3, where `A` and `Y` are variables that do not occur in the LOOP/WHILE file. If `<atom>` is an `<integer>`, replace the assignment of `<atom>` following the substitution rule in Definition A.17.

Listing A.3: Predecessor: Substitution

```
A := <atom>;
Y := 0;
X := 0;
loop A do
        X := Y;
        Y := Y + 1
enddo
```

Listing A.4: Subtraction: Substitution

```
Y := <atom2>;
X := <atom1>;
loop Y do
        X := pred(X)
enddo
```

A.9.5 Subtraction

Definition A.24 (Subtraction)**.** `X := <atom1> - <atom2>` : Stores $a1 \dot{-} a2$ in `X`, where $a1$ and $a2$ are the values stored in `<atom1>` and `<atom2>` respectively. $a1 \dot{-} a2$ equals $a1 - a2$ if $a1 \geqslant a2$ and is zero otherwise.

Definition A.25 (Subtraction: Substitution Rule)**.** Replace `X := <atom1> - <atom2>` by the program in Listing A.4, where `Y` is a variable that does not occur in the LOOP/WHILE file. If either `<atom>` is an `<integer>`, replace assignments of this `<atom>` following the substitution rule in Definition A.17. Replace `pred(X)` following the substitution rule in Definition A.23.

A.9.6 Multiplication

Definition A.26 (Multiplication)**.** `X := <atom1> * <atom2>` : Stores the product of $a1$ and $a2$ in `X`, where $a1$ and $a2$ are the values stored in `<atom1>` and `<atom2>` respectively.

Definition A.27 (Multiplication: Substitution Rule)**.** Replace `X := <atom1> * <atom2>` by the program in Listing A.5, where `Y1` and `Y2` are variables that do not occur in the LOOP/WHILE file. If either `<atom>` is an `<integer>`, replace assignments of this `<atom>` following the substitution rule in Definition A.17.

Listing A.5: Multiplication: Substitution

```
Y1 := <atom1>;
Y2 := <atom2>;
X := 0;
loop Y1 do
        loop Y2 do
                X := X + 1
        enddo
enddo
```

Listing A.6: Division: Substitution

```
Y1 := <atom1>;
Y2 := <atom2>;
X := 0;
if Y2 != 0 then
        loop Y1 do
                if Y1 != 0 then
                        X := X + 1;
                        Y1 := Y1 - Y2
                endif
        enddo;
        X := pred(X)
endif
```

A.9.7 Division

Definition A.28 (Division)**.** `X := <atom1> div <atom2>` : Stores the integer quotient of $a1$ and $a2$ in `X`, if $a2 > 0$, and 0 otherwise. $a1$ and $a2$ are the values stored in `<atom1>` and `<atom2>` respectively.

Definition A.29 (Division: Substitution Rule)**.** Replace `X := <atom1> div <atom2>` by the program in Listing A.6, where `Y1` and `Y2` are variables that do not occur in the LOOP/WHILE file. If either `<atom>` is an `<integer>`, replace assignments of this `<atom>` following the substitution rule in Definition A.17. Replace `if Y != 0 then P endif` and `Y1 := Y1 - Y2` following the substitution rules in Definition A.12 and Definition A.25 respectively.

A.9.8 Modulo

Definition A.30 (Modulo)**.** `X := <atom1> % <atom2>` : Stores the remainder of $a1/a2$ in `X`, if $a2 > 0$, and 0 otherwise. $a1$ and $a2$ are the values stored in `<atom1>` and `<atom2>` respectively.

Definition A.31 (Modulo: Substitution Rule)**.** Replace `X := <atom1> % <atom2>` by the program in Listing A.7, where `Y1` and `Y2` are variables that do not occur in the LOOP/WHILE file. If either `<atom>` is an `<integer>`, replace assignments of this `<atom>` following the substitution rule in Definition A.17. Replace `if Y != 0 then P endif` and `Y1 := Y1 - Y2` following the substitution rules in Definition A.12 and Definition A.25 respectively.

A.9.9 Comparison

Comparing variables and integers is implemented in the extended syntax as expressions, which evaluate to 1, if the condition is true, and to 0 otherwise.

Definition A.32 (Comparison: Substitution Rule)**.** All comparison expressions can be substituted by the program in Listing A.8, with `<greater>`, `<less>`, and `<equal>` each replaced by 1 or 0 according to the substitution rule of the expression instance. After the instance specific substitutions, if either `<atom>` is an `<integer>`, replace assignments of this `<atom>` following the substitution rule in Definition A.17 and replace `if X != 0 then P1 else P2 endif`, `X := Y1 - Y2` and `X := 1` following the substitution rules in Definition A.12, Definition A.25 and Definition A.17 respectively.

Listing A.7: Modulo: Substitution

```
Y1 := <atom1>;
Y2 := <atom2>;
X := 0;
if Y2 != 0 then
        loop Y1 do
                if Y1 != 0 then
                        X := Y1;
                        Y1 := Y1 - Y2
                endif
        enddo
endif
```

Listing A.8: Comparison: Substitution

```
Y1 := <atom1>;
Y2 := <atom2>;
X := Y1 - Y2;
if X != 0 then
        X := <greater>
else
        X := Y2 - Y1;
        if X != 0 then
                X := <less>
        else
                X := <equal>
        endif
endif
```

A.9.10 Equal

Definition A.33 (Equal)**.** `X := <atom1> == <atom2>` : Stores 1 in `X` if $a1$ equals $a2$ and 0 otherwise. $a1$ and $a2$ are the values stored in `<atom1>` and `<atom2>` respectively.

Definition A.34 (Equal: Substitution Rule)**.** Replace `X := <atom1> == <atom2>` by the program in Listing A.8. Replace `<greater>`, `<less>`, and `<equal>` with `0`, `0`, and `1` respectively. Follow the substitution rule in Definition A.32.

A.9.11 Unequal

Definition A.35 (Unequal)**.** `X := <atom1> != <atom2>` : Stores 1 in `X` if $a1$ does not equal $a2$ and 0 otherwise. $a1$ and $a2$ are the values stored in `<atom1>` and `<atom2>` respectively.

Definition A.36 (Unequal: Substitution Rule)**.** Replace `X := <atom1> != <atom2>` by the program in Listing A.8. Replace `<greater>`, `<less>`, and `<equal>` with `1`, `1`, and `0` respectively. Follow the substitution rule in Definition A.32.

A.9.12 Less

Definition A.37 (Less)**.** `X := <atom1> < <atom2>` : Stores 1 in `X` if $a1$ is less than $a2$ and 0 otherwise. $a1$ and $a2$ are the values stored in `<atom1>` and `<atom2>` respectively.

Definition A.38 (Less: Substitution Rule)**.** Replace `X := <atom1> < <atom2>` by the program in Listing A.8. Replace `<greater>`, `<less>`, and `<equal>` with `0`, `1`, and `0` respectively. Follow the substitution rule in Definition A.32.

A.9.13 Less or Equal

Definition A.39 (Less or Equal)**.** `X := <atom1> <= <atom2>` : Stores 1 in `X` if $a1$ is less than or equal to $a2$ and 0 otherwise. $a1$ and $a2$ are the values stored in `<atom1>` and `<atom2>` respectively.

Definition A.40 (Less or Equal: Substitution Rule)**.** Replace `X := <atom1> <= <atom2>` by the program in Listing A.8. Replace `<greater>`, `<less>`, and `<equal>` with `0`, `1`, and `1` respectively. Follow the substitution rule in Definition A.32.

A.9.14 Greater

Definition A.41 (Greater)**.** `X := <atom1> > <atom2>` : Stores 1 in `X` if $a1$ is greater than $a2$ and 0 otherwise. $a1$ and $a2$ are the values stored in `<atom1>` and `<atom2>` respectively.

Definition A.42 (Greater: Substitution Rule)**.** Replace `X := <atom1> > <atom2>` by the program in Listing A.8. Replace `<greater>`, `<less>`, and `<equal>` with `1`, `0`, and `0` respectively. Follow the substitution rule in Definition A.32.

A.9.15 Greater or Equal

Definition A.43 (Greater or Equal)**.** `X := <atom1> >= <atom2>` : Stores 1 in `X` if $a1$ is greater than or equal to $a2$ and 0 otherwise. $a1$ and $a2$ are the values stored in `<atom1>` and `<atom2>` respectively.

Definition A.44 (Greater or Equal: Substitution Rule)**.** Replace `X := <atom1> >= <atom2>` by the program in Listing A.8. Replace `<greater>`, `<less>`, and `<equal>` with `1`, `0`, and `1` respectively. Follow the substitution rule in Definition A.32.

A.9.16 Macro Call

Definition A.45 (Macro Call)**.** `X := <macro>(<atom1>, <atom2>,... <atom_n>)` : Calls the macro `<macro>` and stores then resulting value, as defined in Definition A.9, in `X`. `<macro>` must be a macro defined in a macro definition in this LOOP/WHILE file.

Definition A.46 (Macro Call: Substitution Rule)**.** If `<macro>` is defined anywhere in the LOOP/WHILE file as `def <macro> D P enddef`, remove the calling assignment `X := <macro>(<atom1>, <atom2>,... <atom_n>)`. For each input variable `Xi` declared in `D`, insert `Xi := <atomi>`. Insert `P`, followed by `X := Xo`, where `Xo` is the variable declared as output in `D`. Rename all `Xi` in the first n inserted lines, `Xo` and all variables in `P` to names that do not occur in the LOOP/WHILE file. If `<macro>` is not defined in this manner, the file is invalid. If the number n of parameters `<atom1>` through `<atom_n>` is not equal to the number of input variables declared in `D`, the file is invalid.

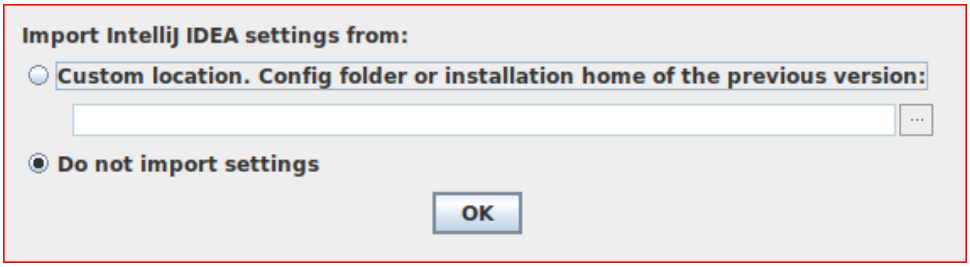
B Manual - How to use

In this chapter we will discuss how to work with our IntelliJ plugin and our interpreter. For general instructions on IntelliJ, refer to the website of the IntelliJ project [8].

This manual contains examples of IntelliJ version 2018.1.3 on a Linux system. Installation and usage on a Windows system will slightly differ in representation of file system paths. Usage on both macOS and Windows might also differ in the placement and labeling of buttons in IntelliJ's GUI, even at the same version which is used in this manual.

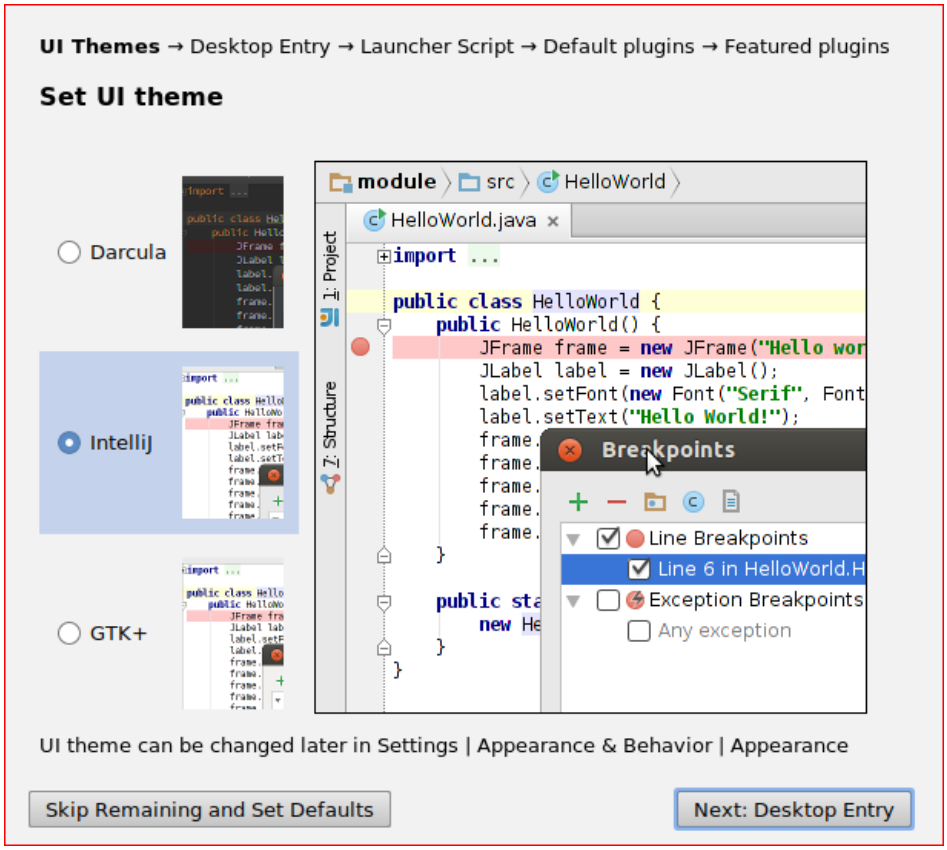
B.1 Installation

Download and install IntelliJ, following their installation guide [8]. Download the IntelliJ plugin (`LoopWhileLanguagePlugin.jar`) and the LOOP/WHILE interpreter executable (`lwre`) from `https://www8.cs.fau.de/tools:loopwhile`. In addition, the sources can be found at `https://gitlab.cs.fau.de/i8/intellij-lw-plugin` and `https://gitlab.cs.fau.de/i8/LoopWhile-yacc-interpreter`. Start IntelliJ and follow the screenshots and descriptions in Figure B.1 through Figure B.21



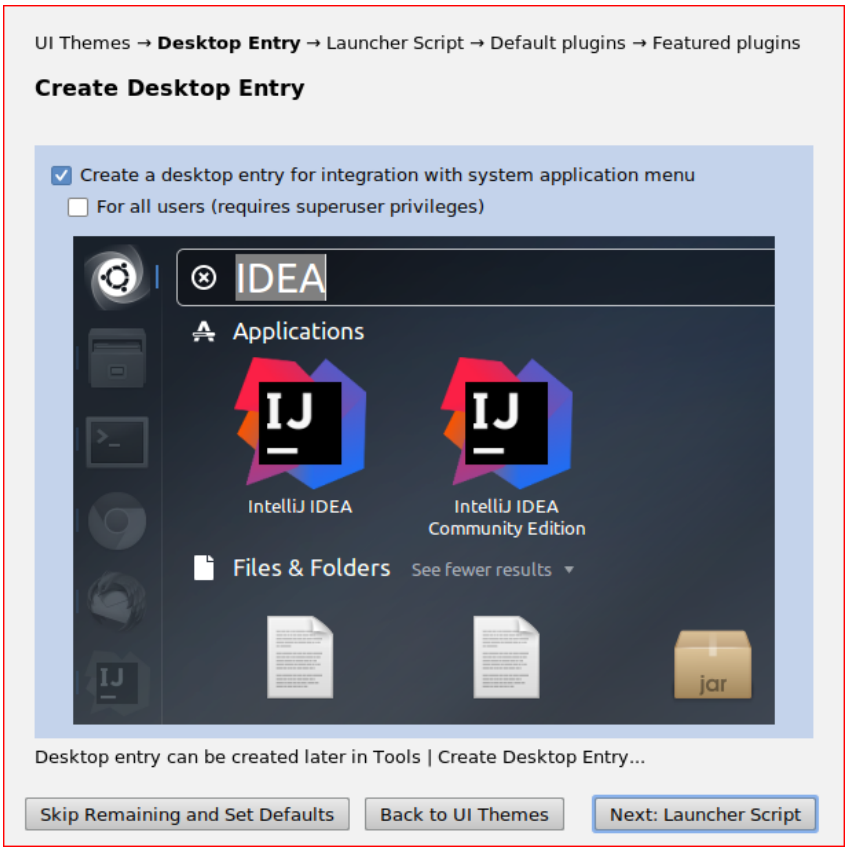
If this is the very first start of IntelliJ, Figure B.1 through Figure B.9 will guide your through the initial setup. If you have used IntelliJ before, skip to Figure B.10.

Figure B.1



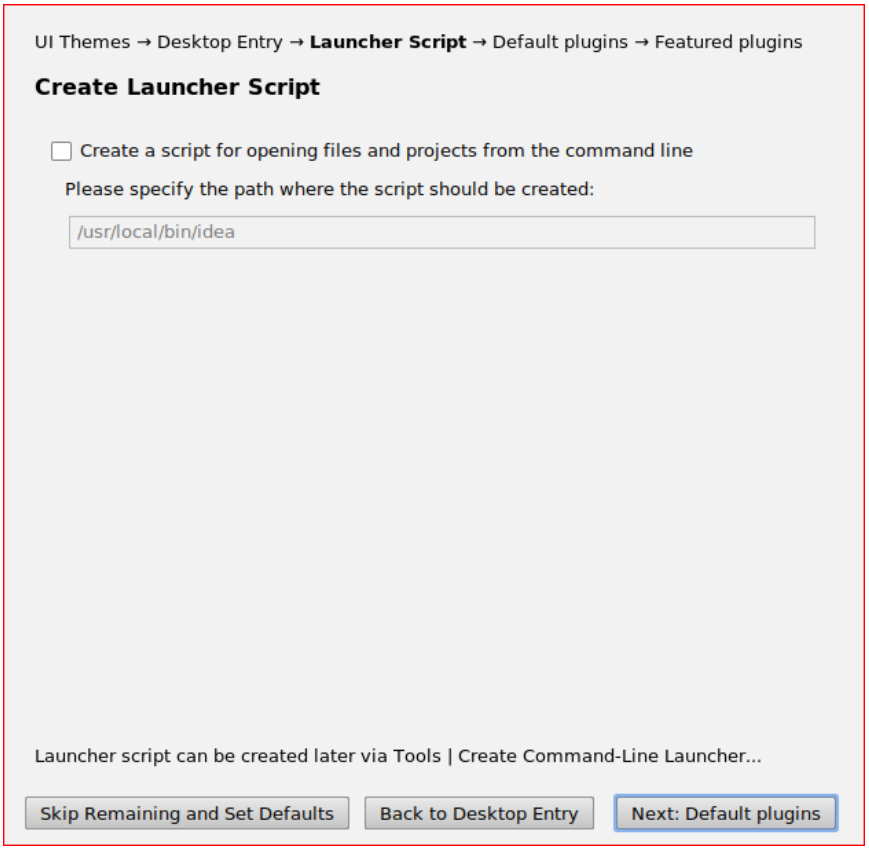
Select a color scheme and click `Next`.

Figure B.2



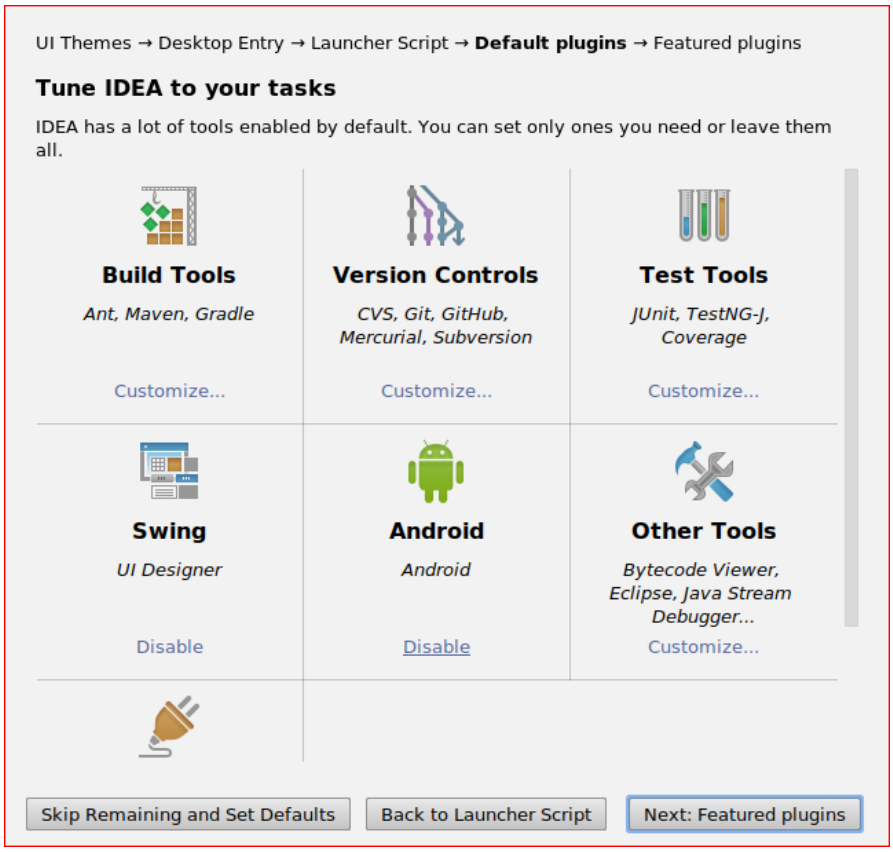
Click `Next`.

Figure B.3



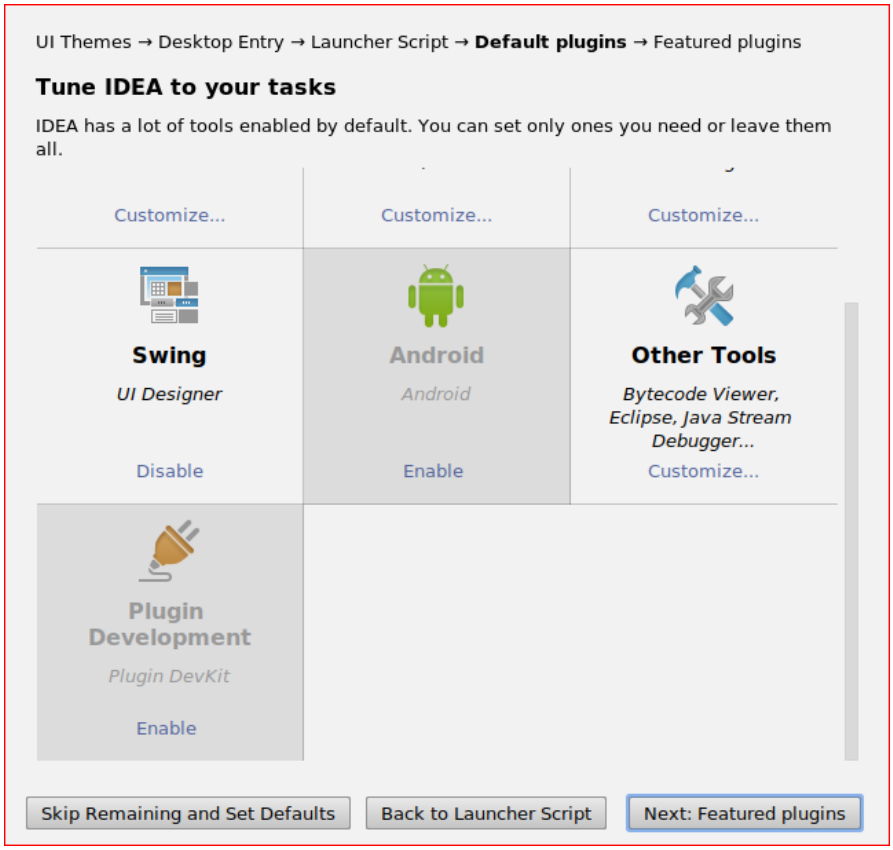
Click `Next`.

Figure B.4



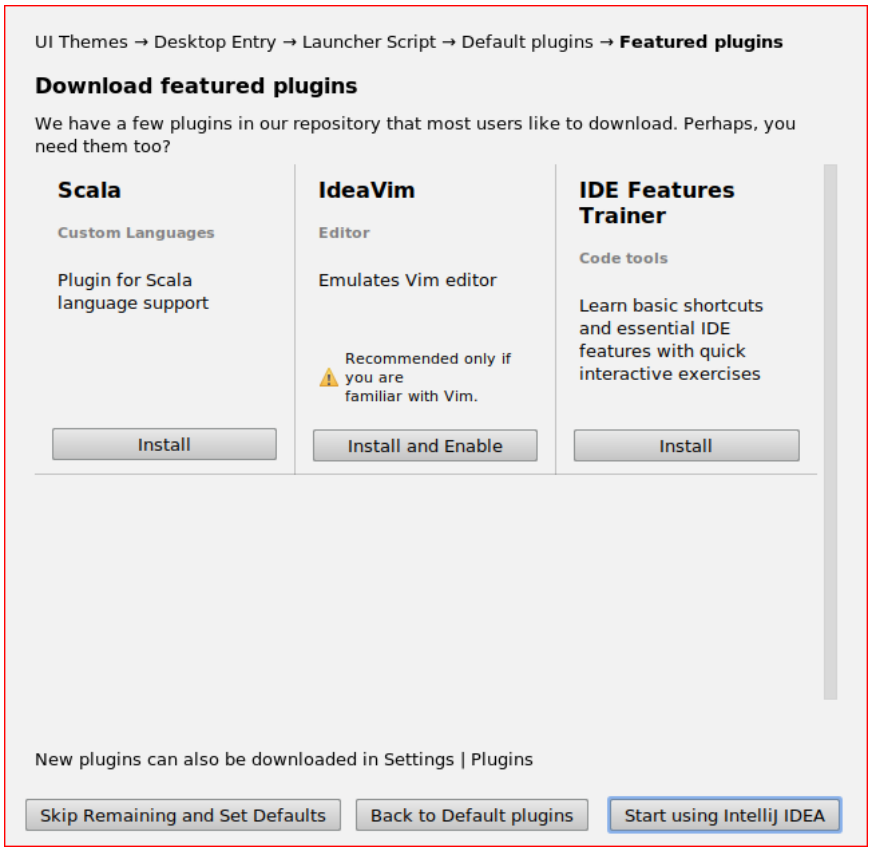
Optional: Disable `Android` and `Plugin Development` to save disk space.

Figure B.5



Click `Next`.

Figure B.6

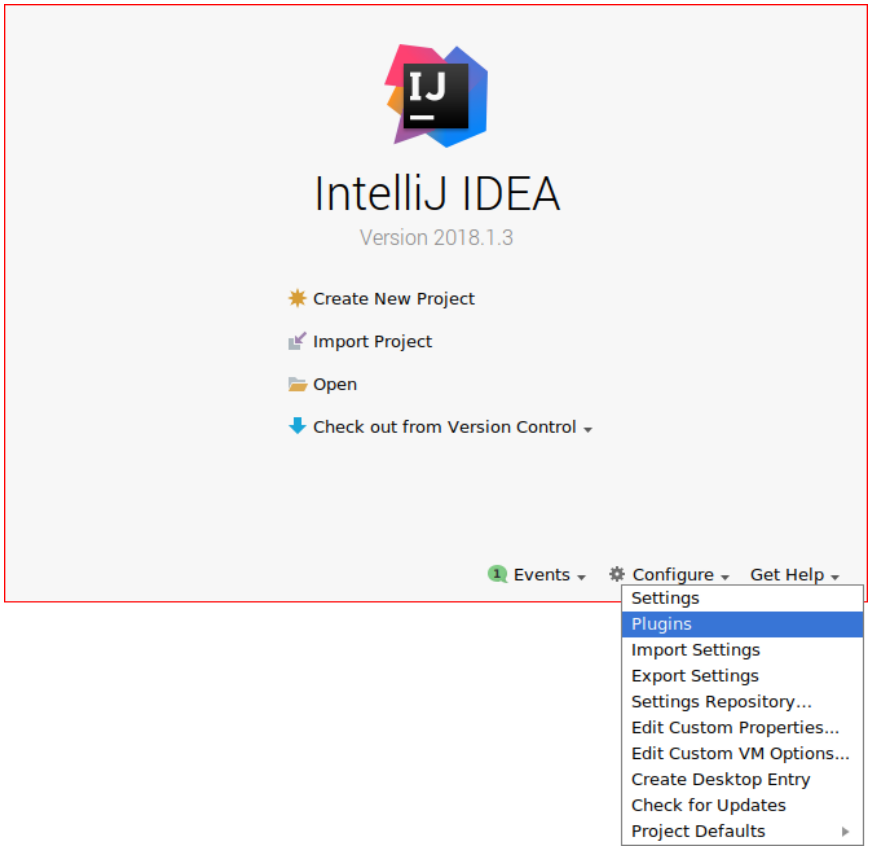


Click `Start using Intelij IDEA`.

Figure B.7

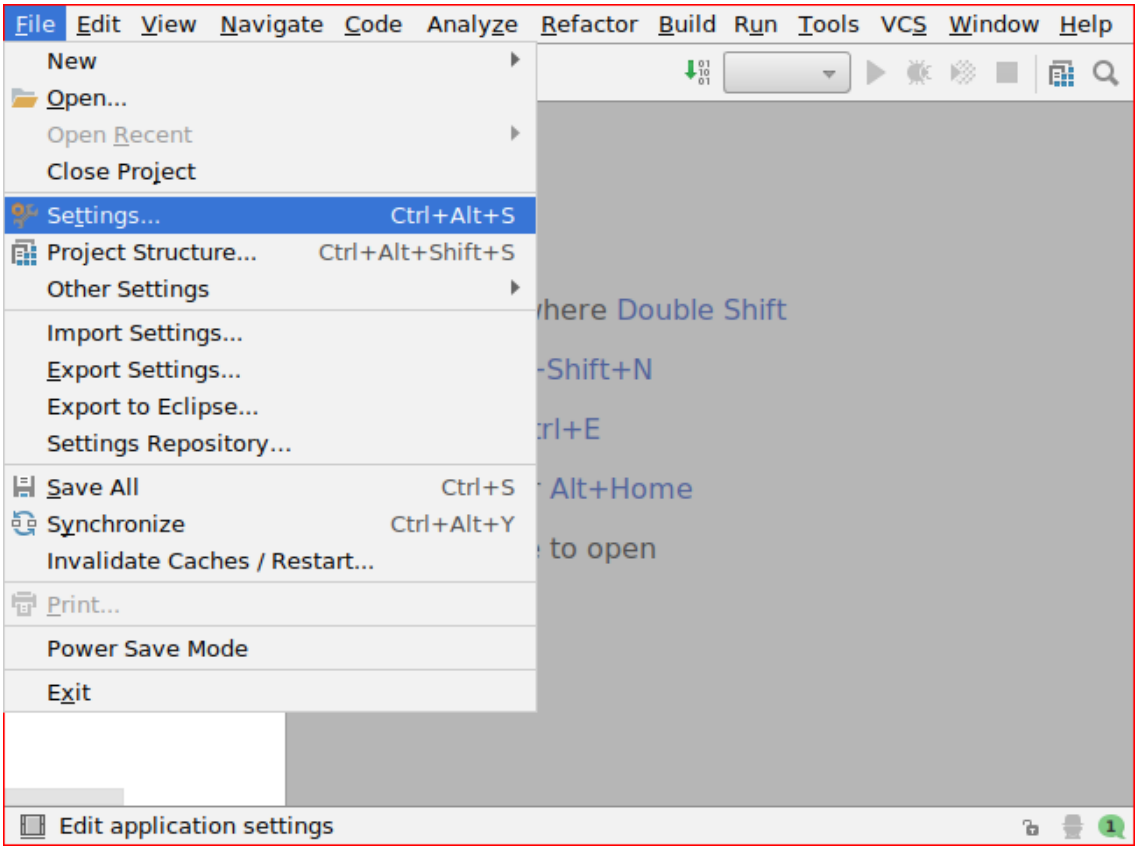
Wait for the loading screen to finish.

Figure B.8



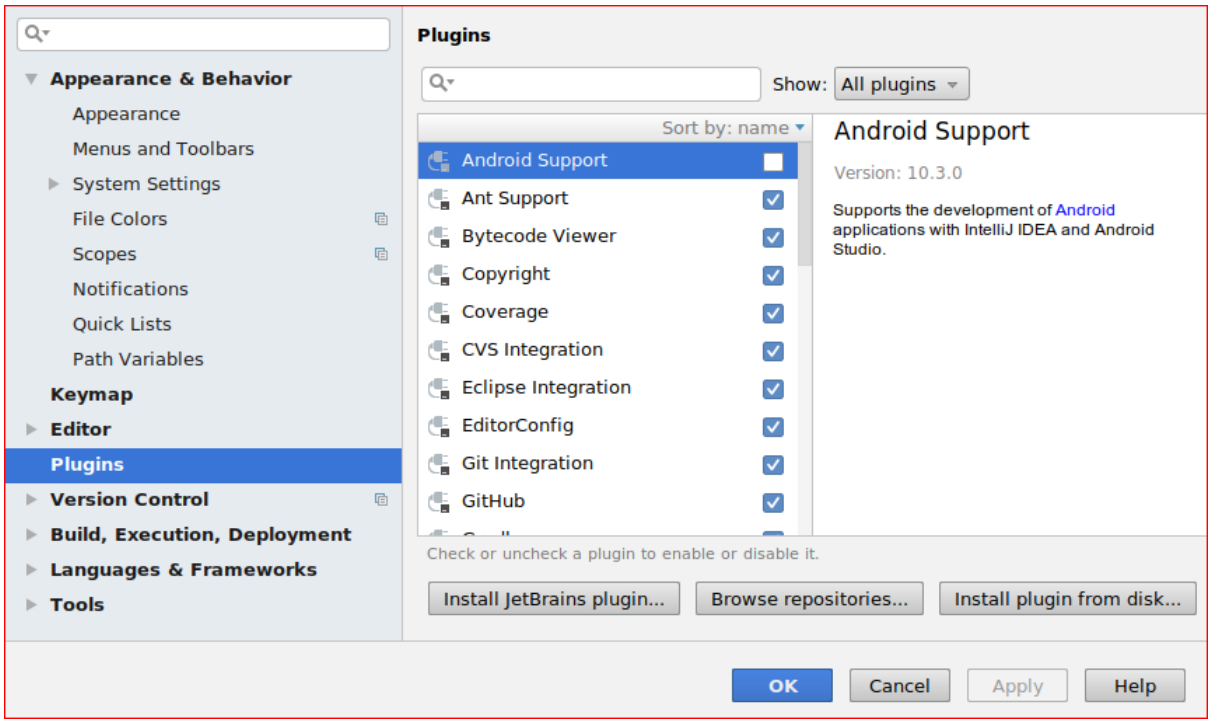
Click `Configure`, then click `Plugins`. Skip to Figure B.11.

Figure B.9



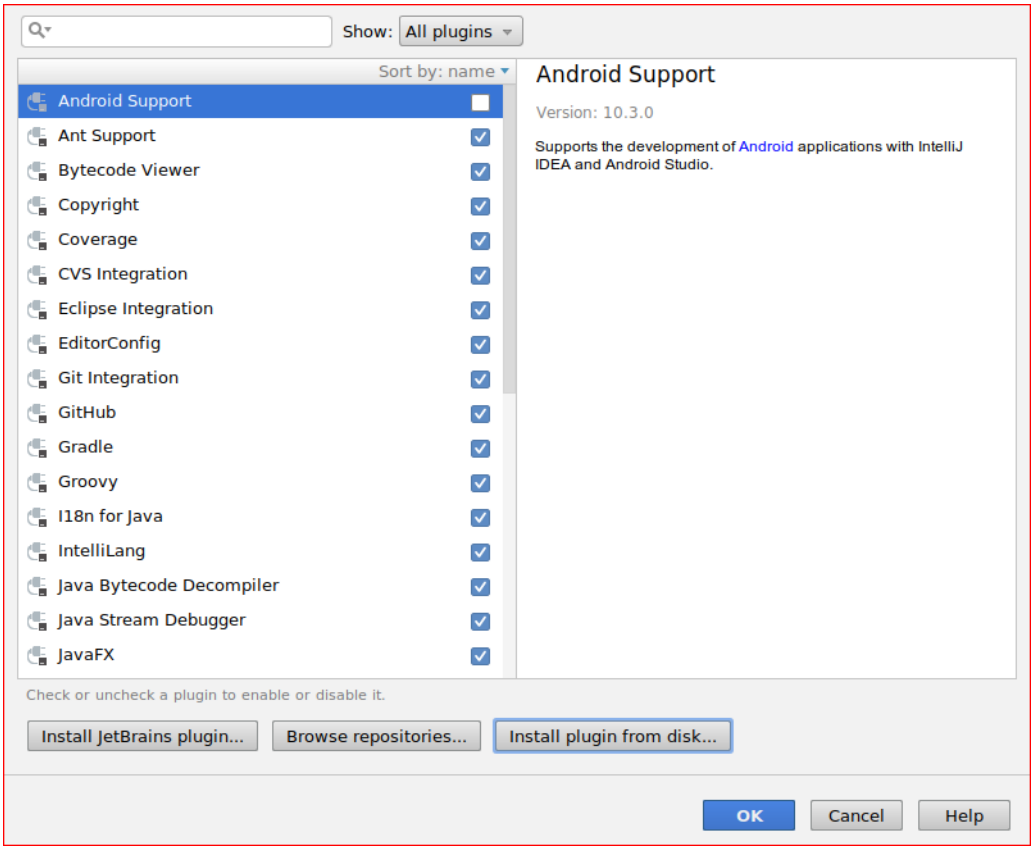
This step differs between IntelliJ versions. Mouse over `File` and click `Settings...`. If you have an `IntelliJ IDEA` button in the top bar, the `Settings...` will be in that menu, instead of in the `File` menu.

Figure B.10



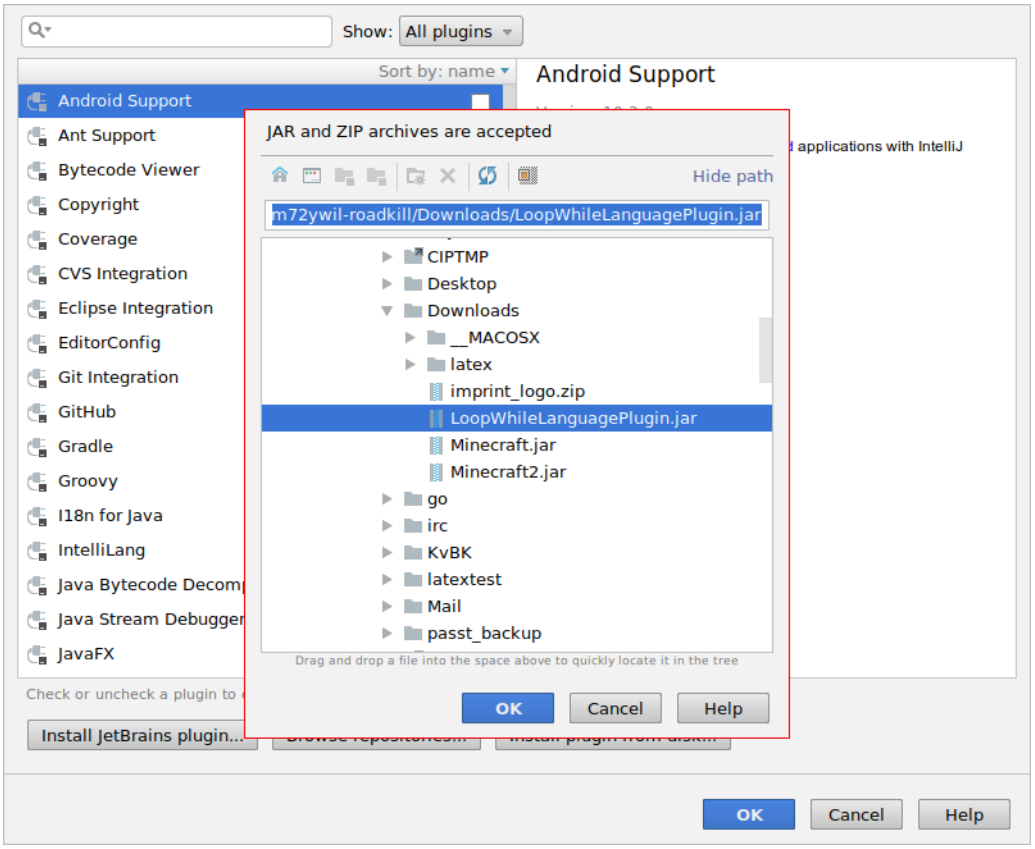
Click `Plugins` in the left column.

Figure B.11



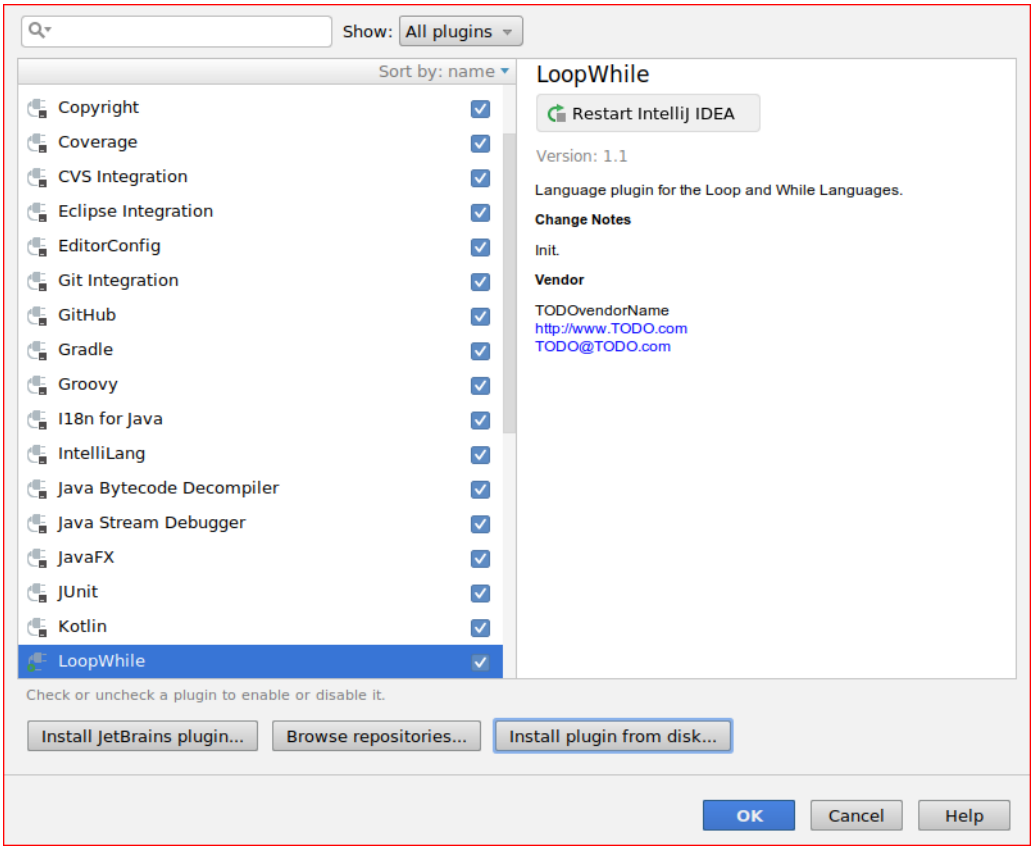
Click `Install plugin from disk...`.

Figure B.12



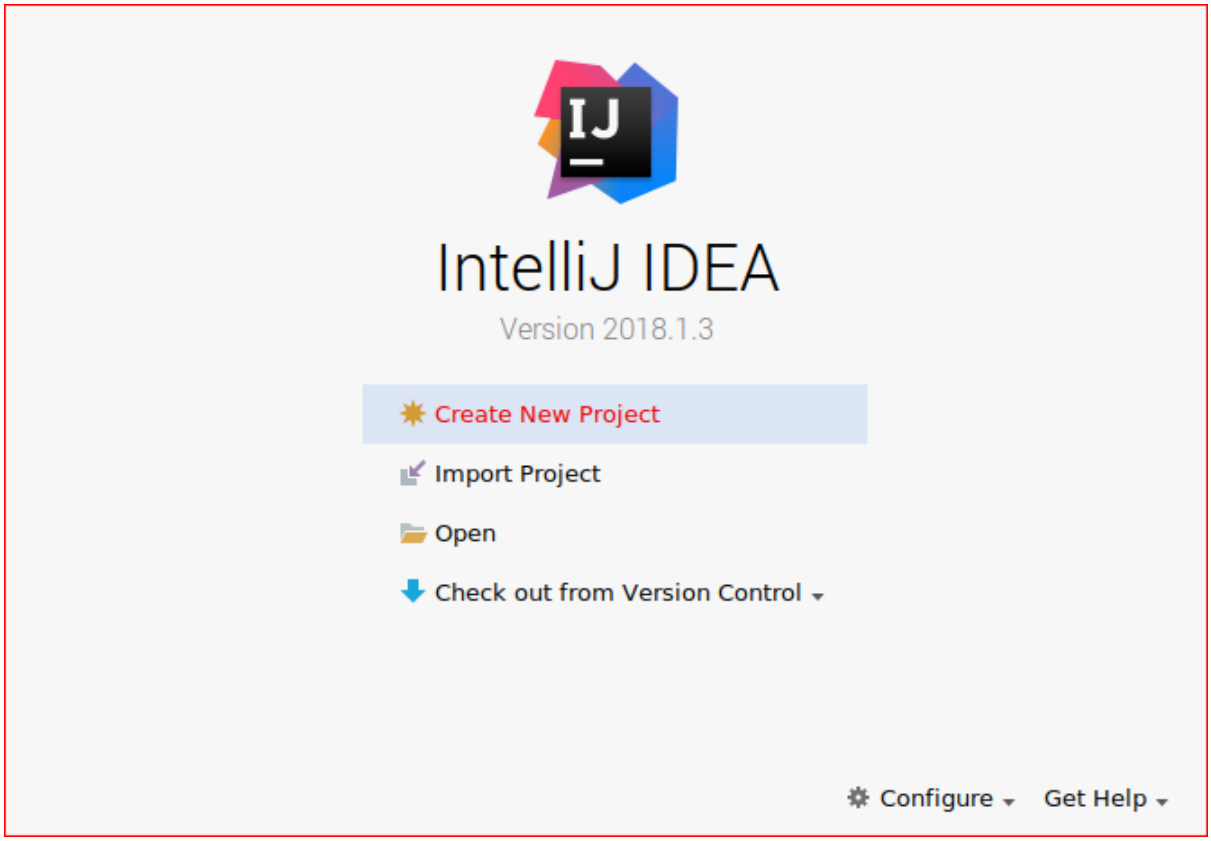
Navigate to the `LoopWhileLanguagePlugin.jar` file which you downloaded in Section B.1 and click `OK`.

Figure B.13



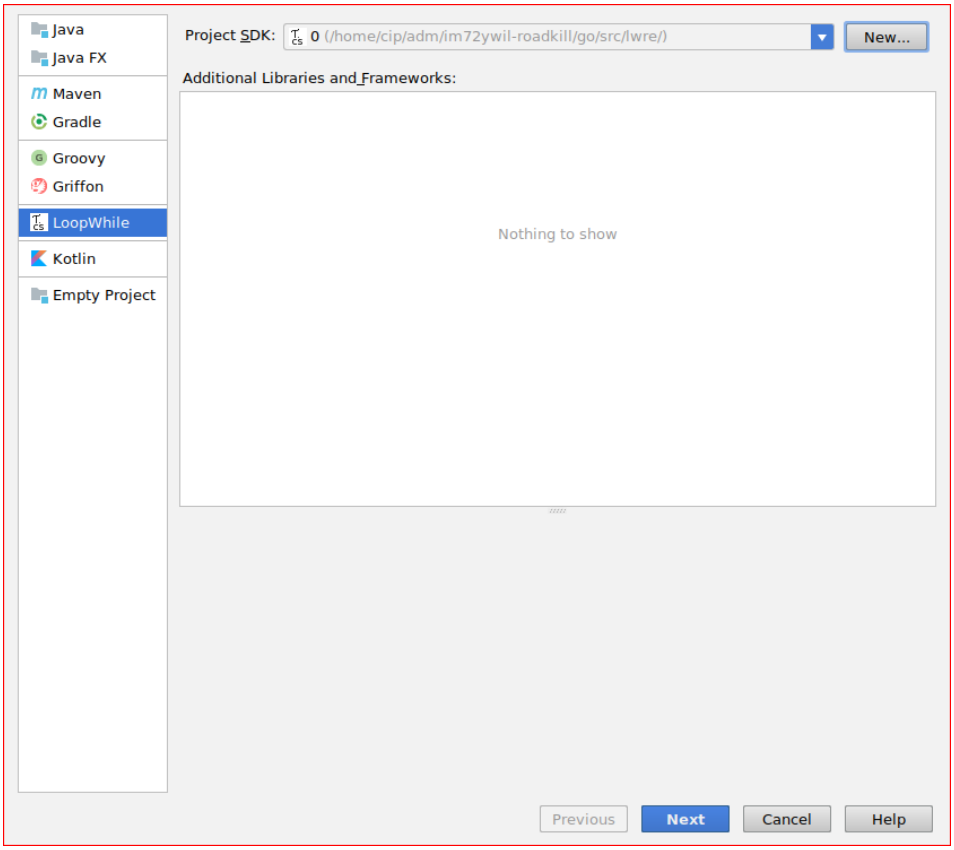
Click `Restart IntelliJ IDEA` in the top right corner and wait for IntelliJ to restart.

Figure B.14



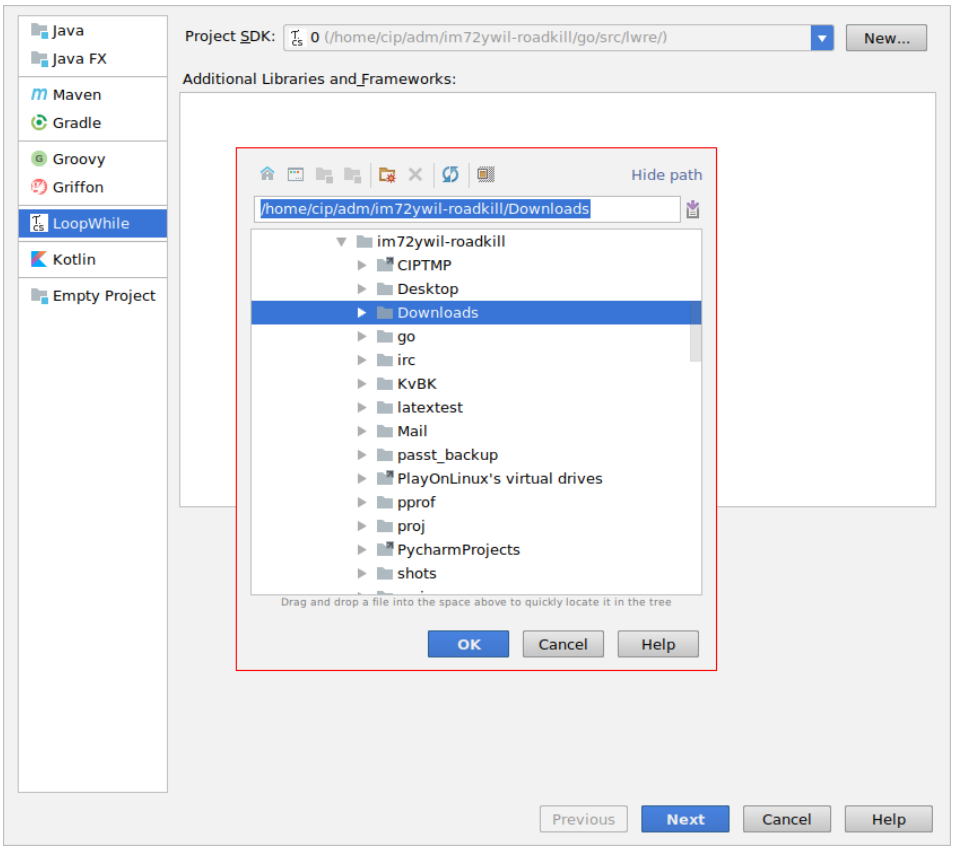
Click `Create New Project`.

Figure B.15



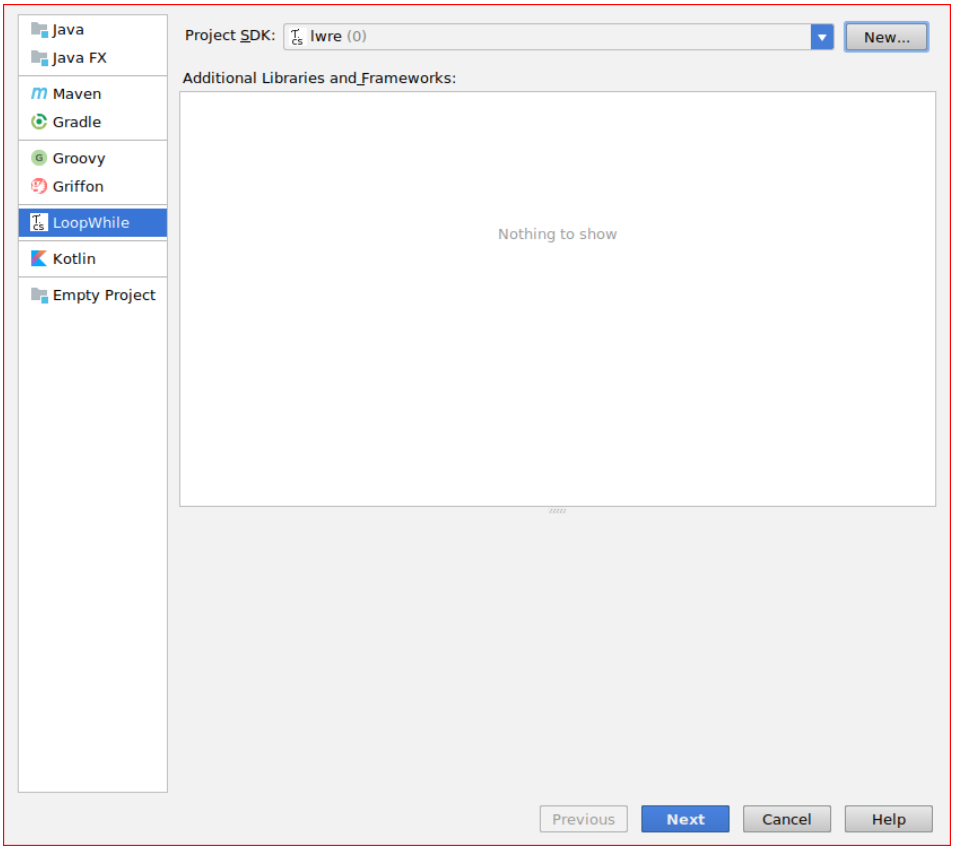
Select `LoopWhile` in the left column and click `New...` in the top right corner.

Figure B.16



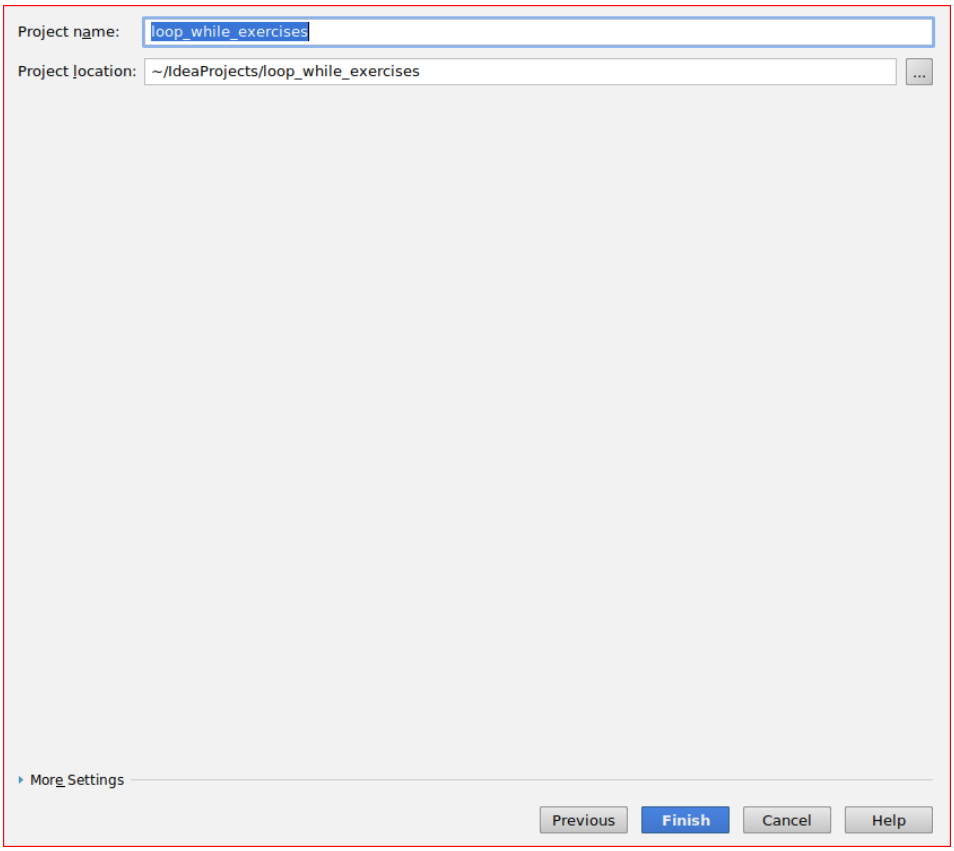
This step differs between IntelliJ versions. Navigate to the interpreter executable (`lwre`) which you downloaded in Section B.1 and select it. If you only see directories in this dialog, but not files, just select the directory which contains the `lwre` executable. Click `OK`.

Figure B.17



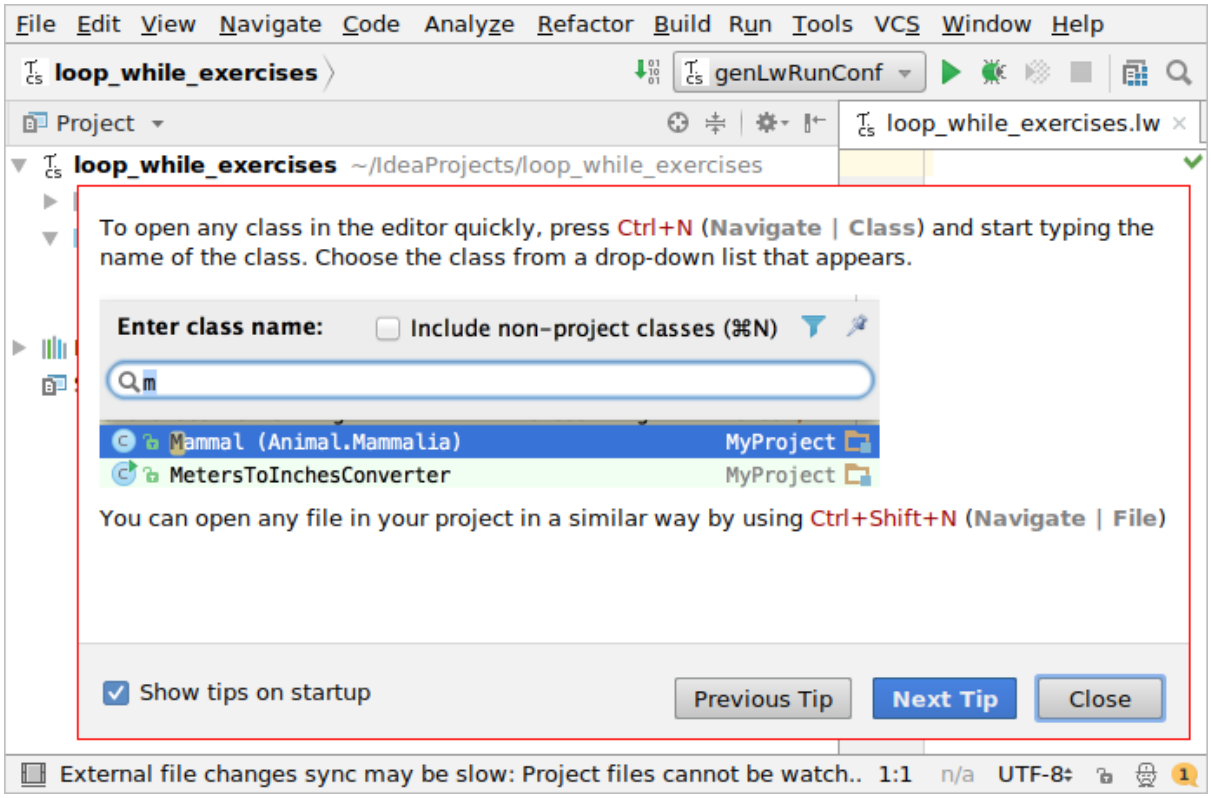
Click `Next`.

Figure B.18



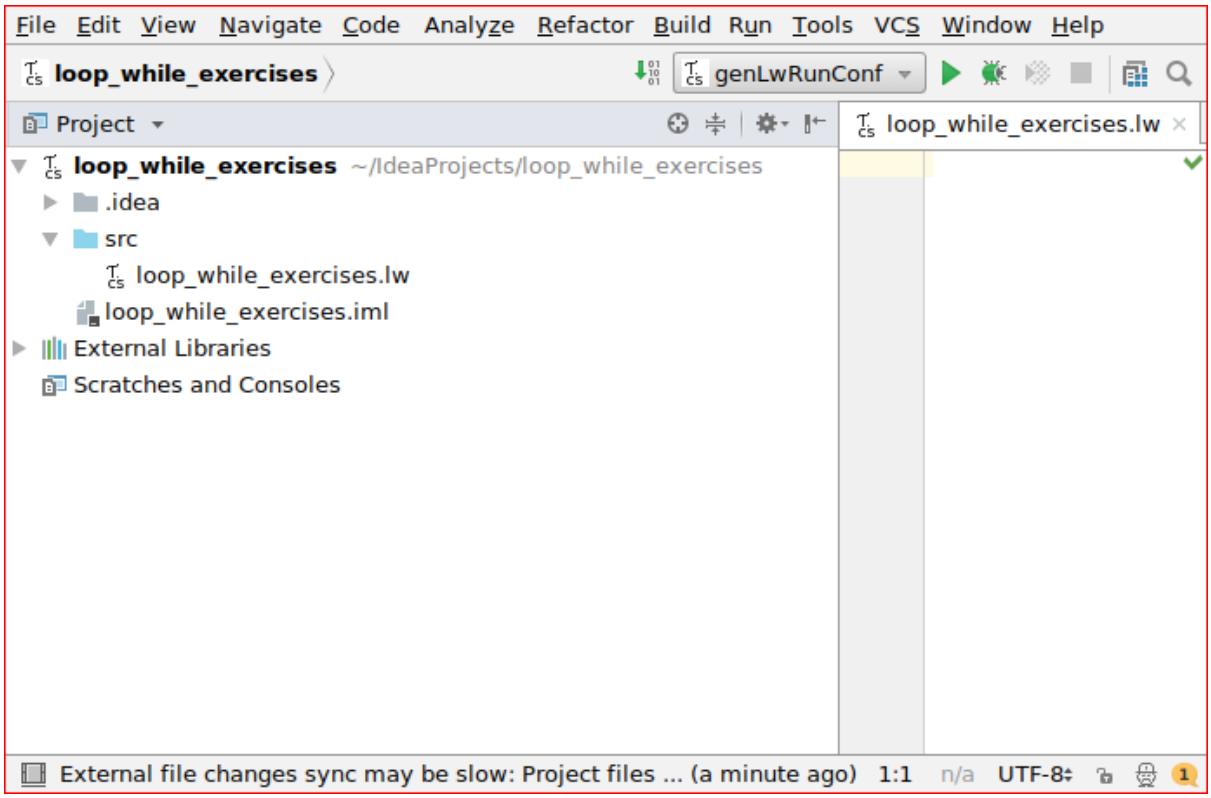
Optional: Select a suitable name for this project. Click `Finish`.

Figure B.19



Optional: Untick `Show tips on startup`. Click `Close`.

Figure B.20

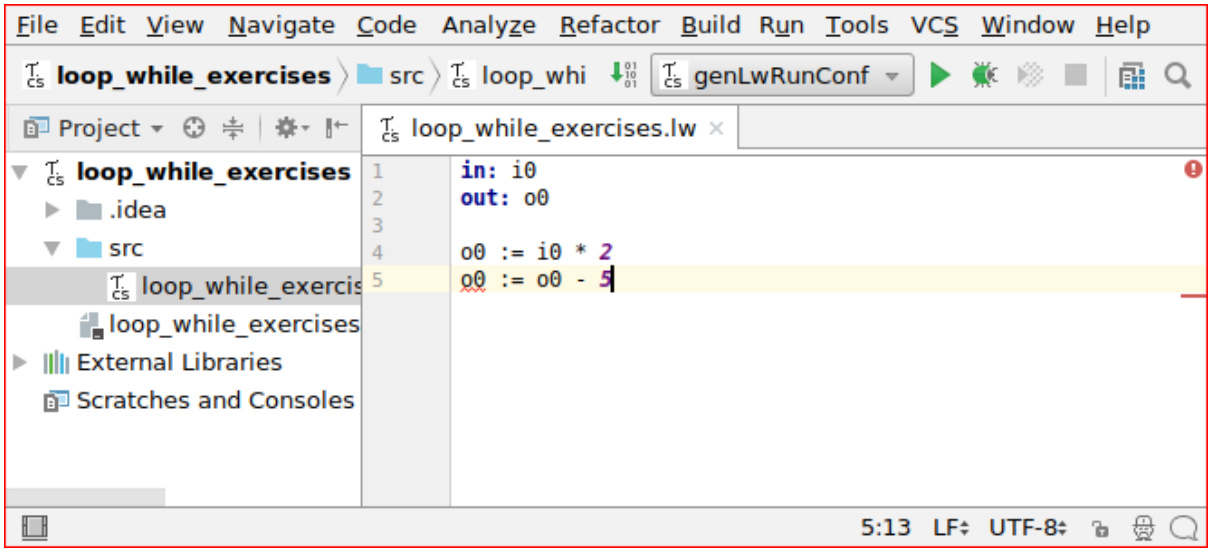


You are ready to write your first LOOP/WHILE program.

Figure B.21

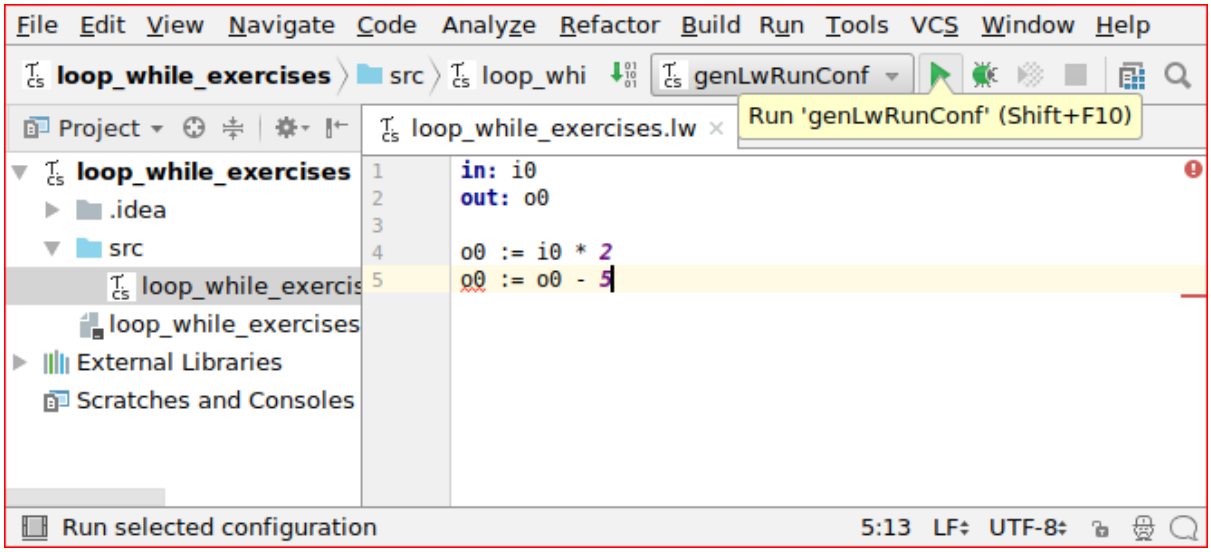
B.2 Programming

After completing the installation (Section B.1) you can start writing code. Figure B.22 through Figure B.28 guide you to your first working LOOP program.



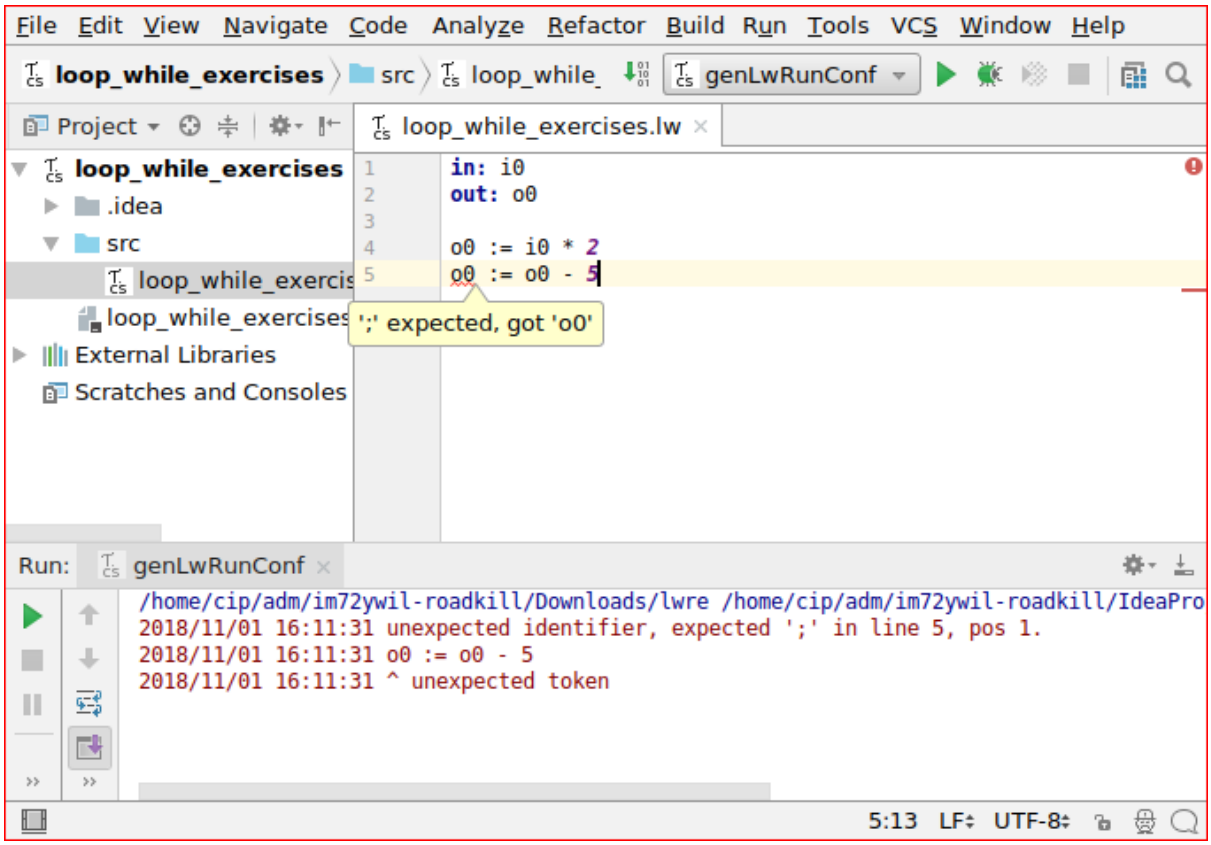
Type your program into the editor. The editor is the window on the right.

Figure B.22



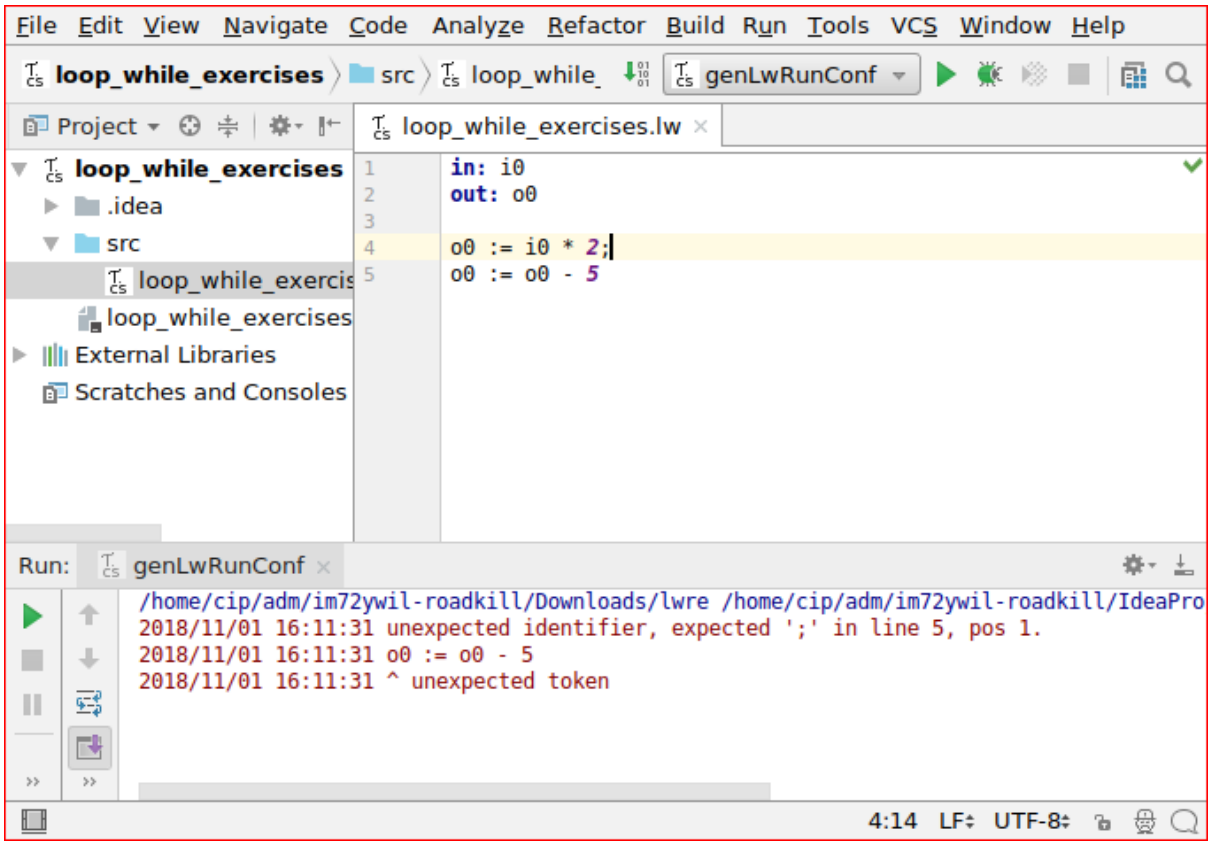
Hit the Run button .

Figure B.23



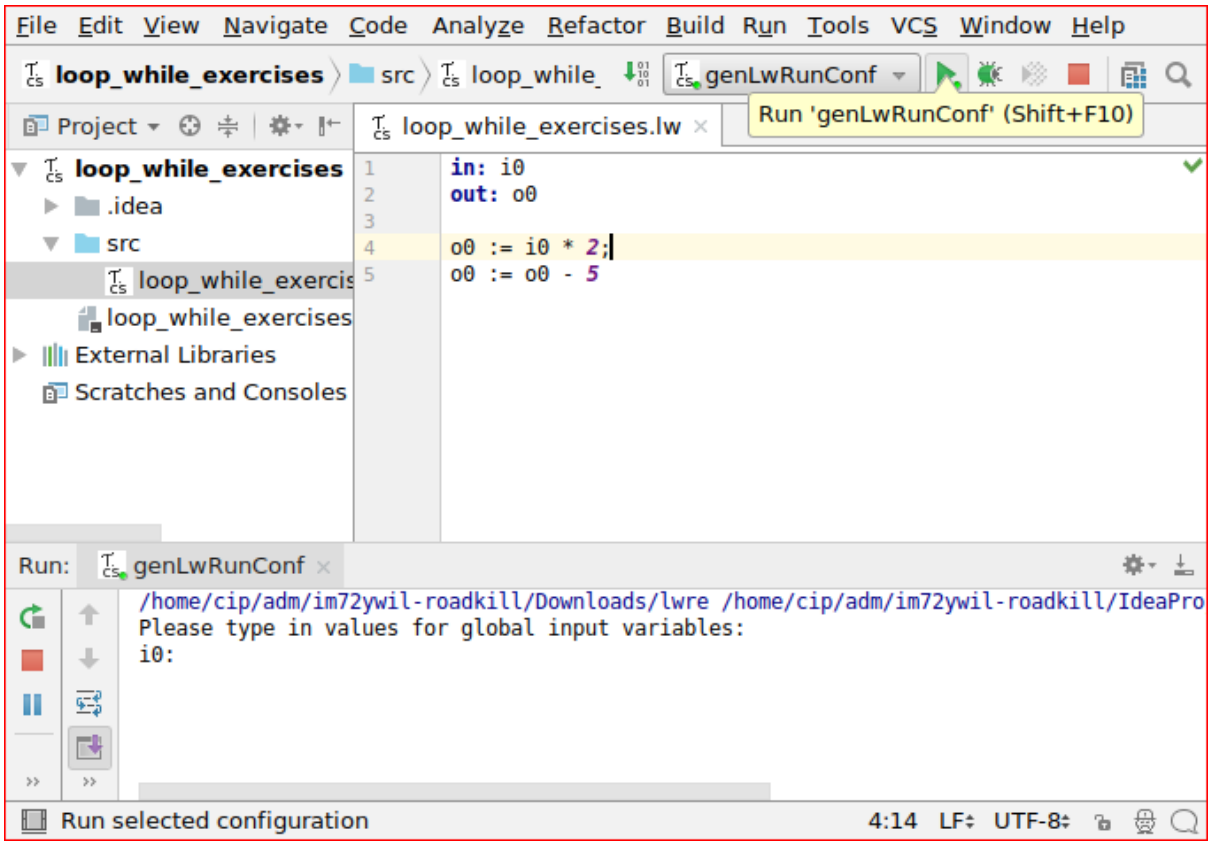
We have made a mistake. The window on the bottom shows the error message from the interpreter. In addition, you can hover the mouse over the underlined piece of code to view an error message from the IntelliJ plugin. We need to put a ';' between the two statements.

Figure B.24



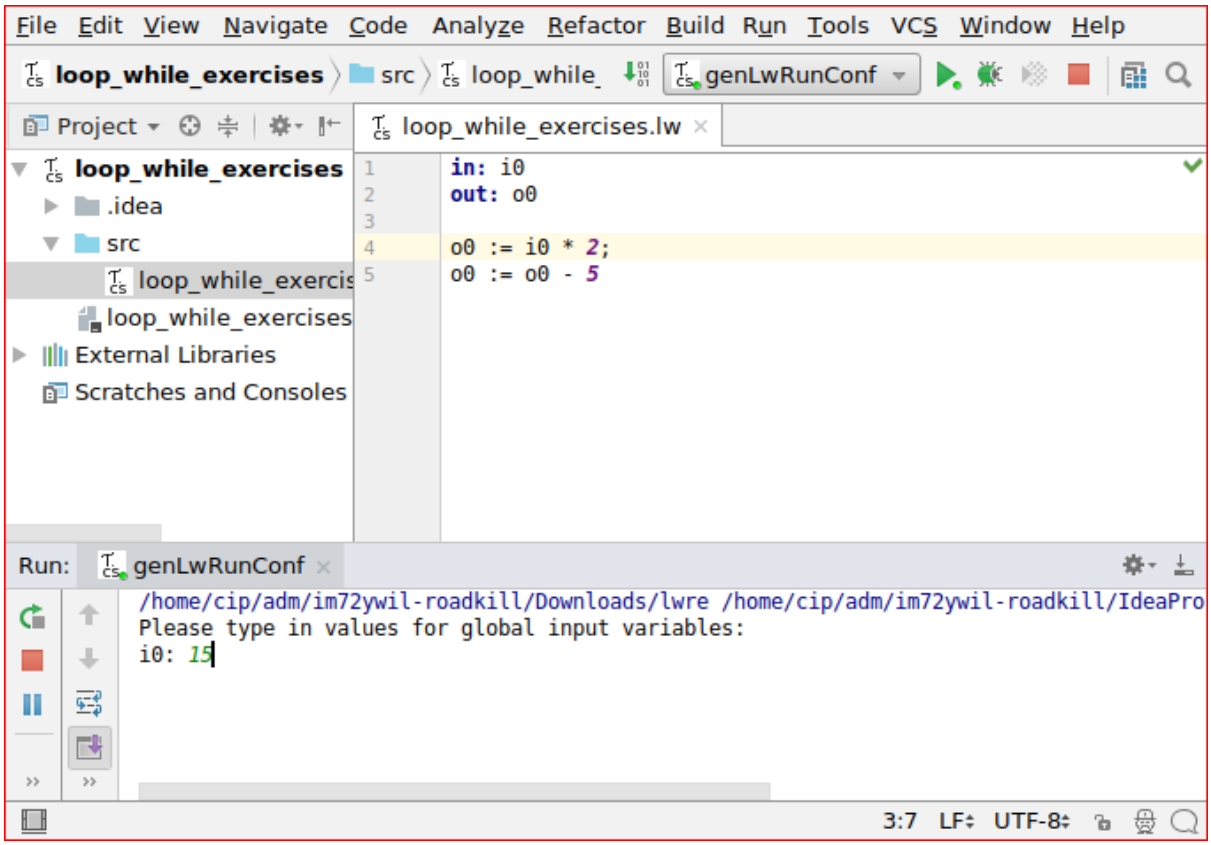
Correct the error and hit ▶.

Figure B.25



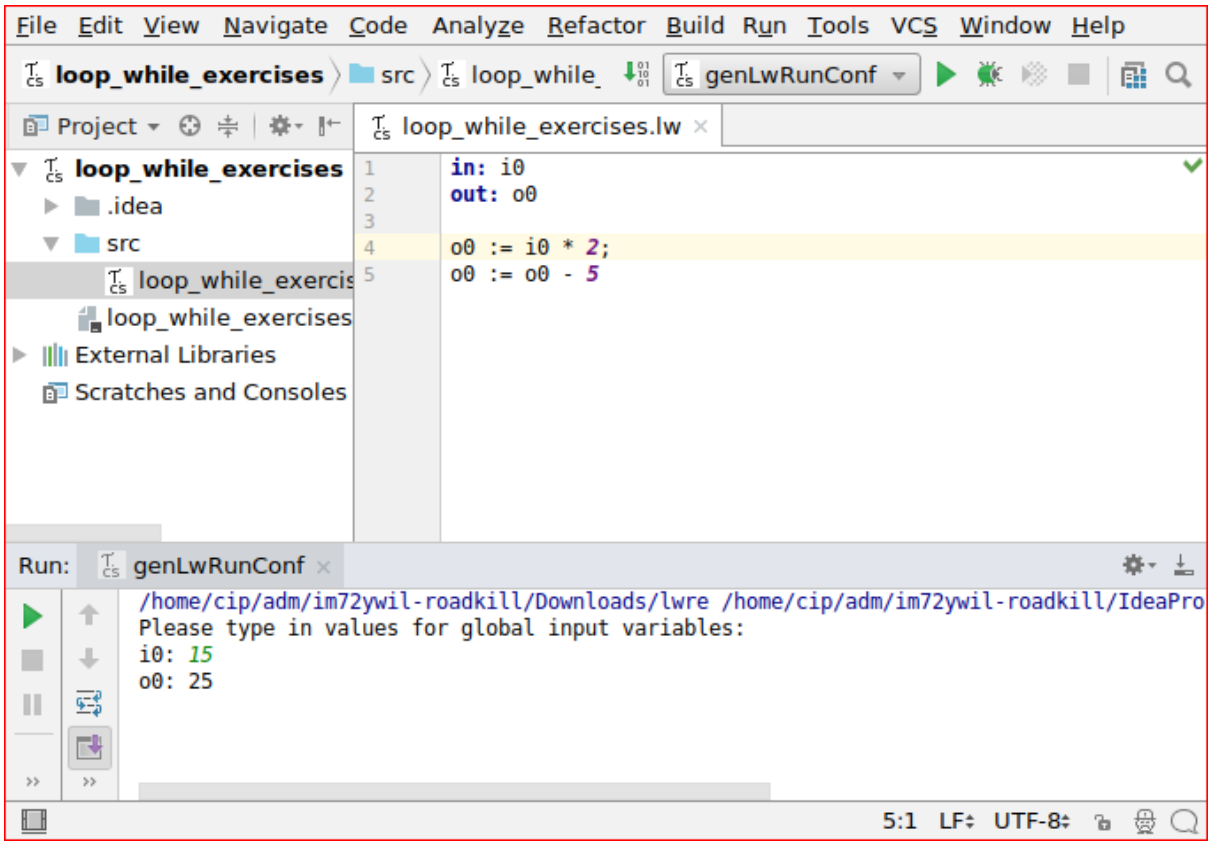
Success! The program is running, as indicated by the green dot next to the ▶ button. The interpreter asks you for a value for `i0`.

Figure B.26



Enter a value for `i0` and hit `Return`.

Figure B.27

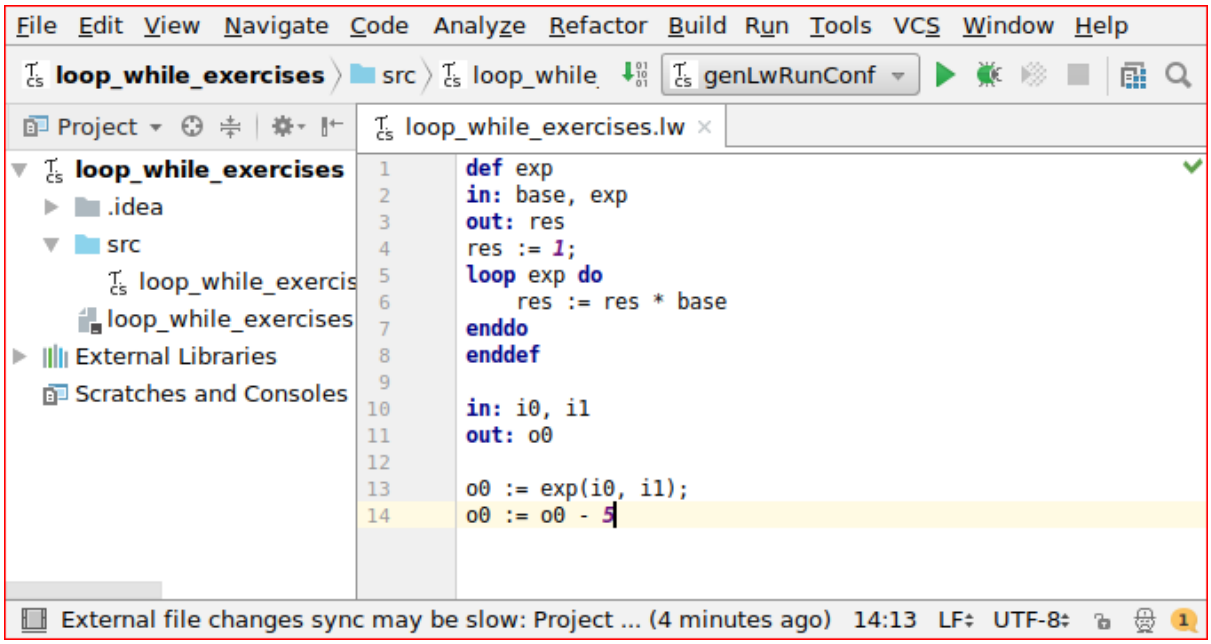


The program finished successfully and shows the resulting value of `o0`.

Figure B.28

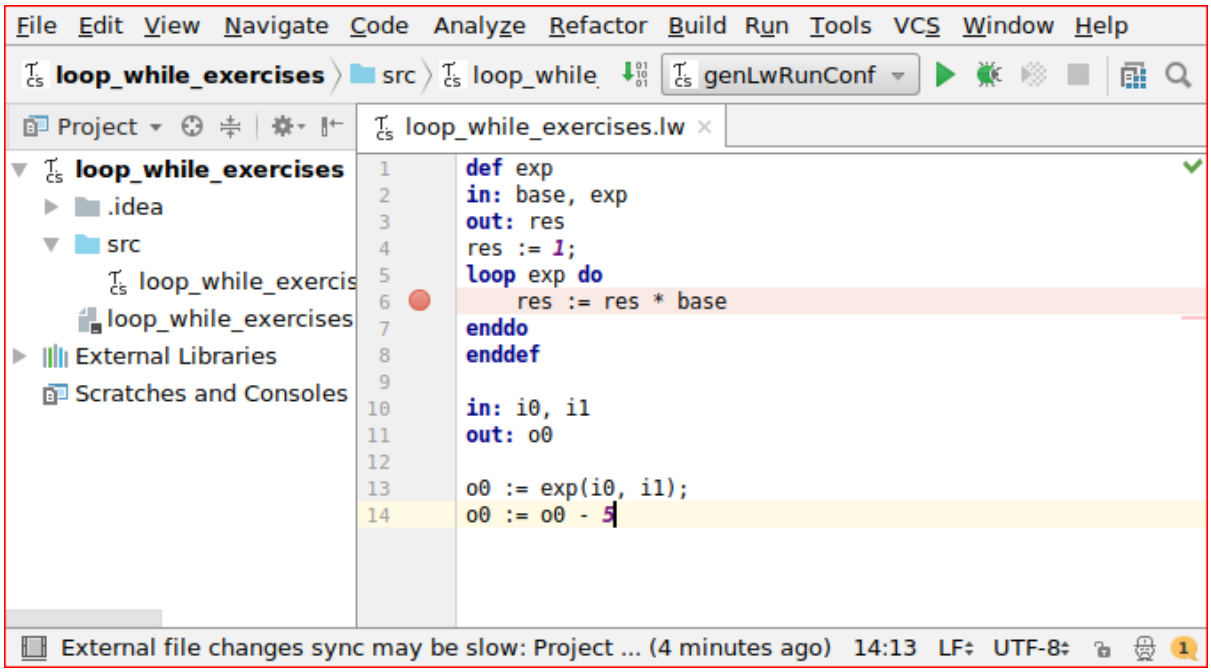
B.3 Debugging

IntelliJ's debugger interface can help you both to investigate logic errors in a program and to analyse the control flow of a LOOP or WHILE program. This section demonstrates the available features through our IntelliJ plugin and our debugger.



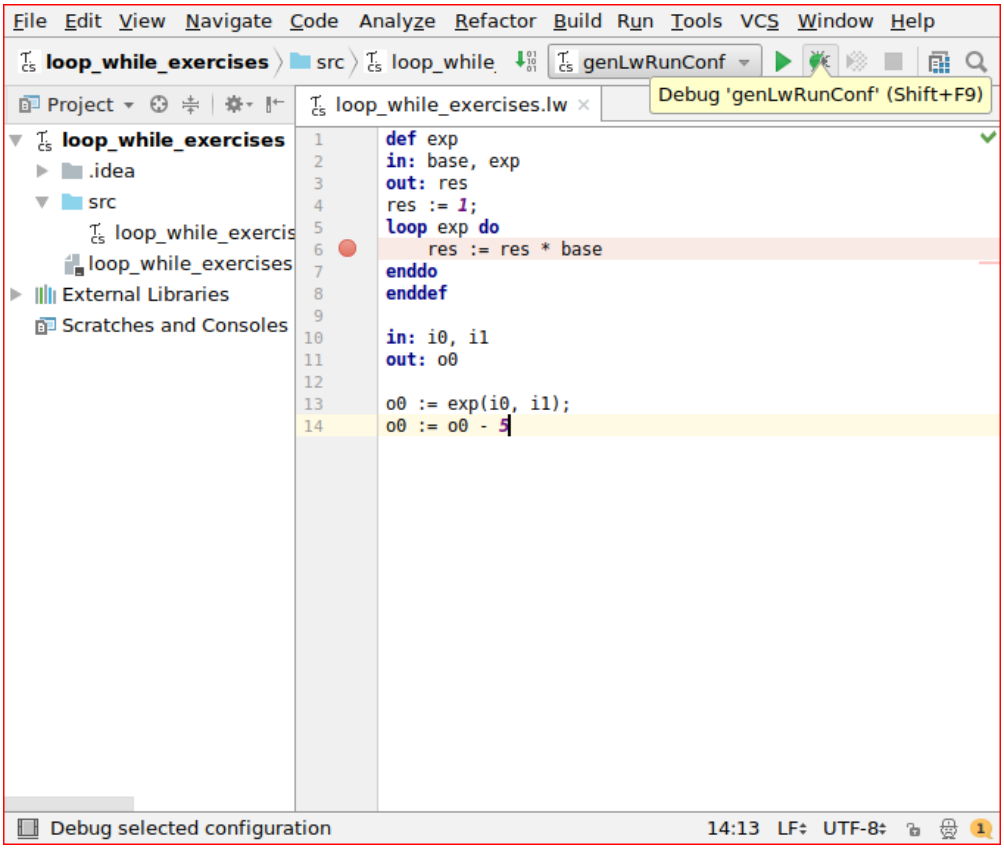
Type your program into the editor. The editor is the window on the right. Figure B.29 through Figure B.45 demonstrate the use of the debugger in IntelliJ.

Figure B.29



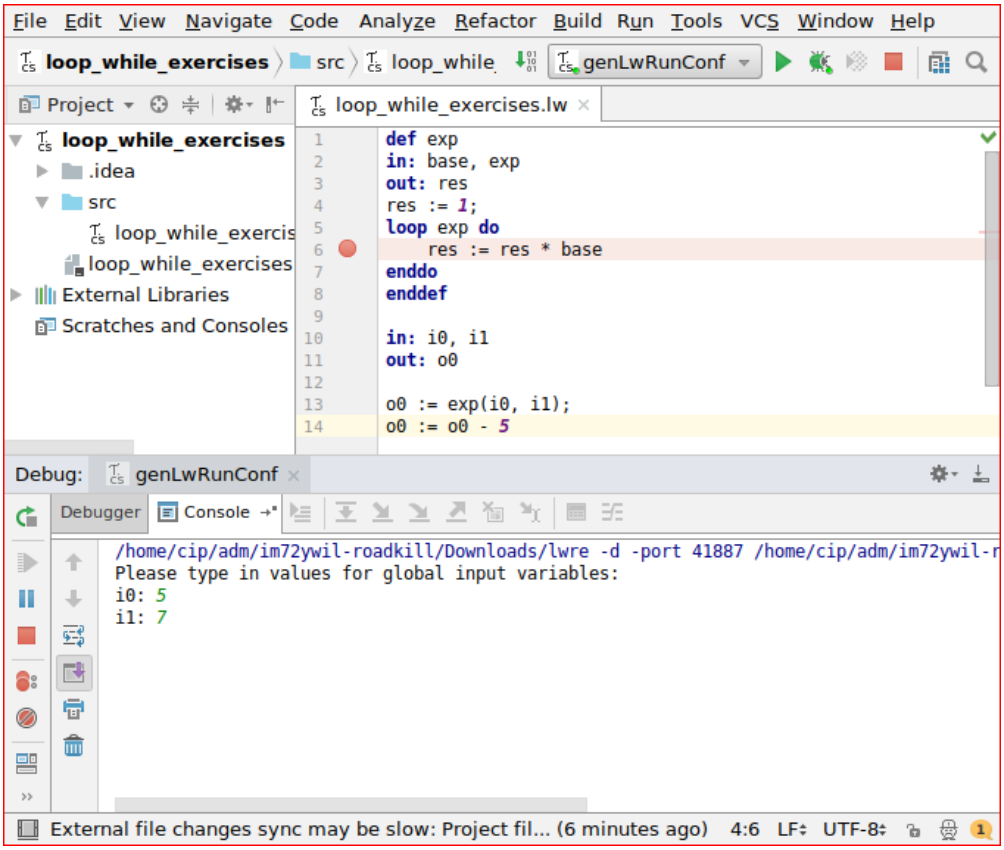
Set a breakpoint in line 6, by clicking directly to the right of the line-number '6'.

Figure B.30



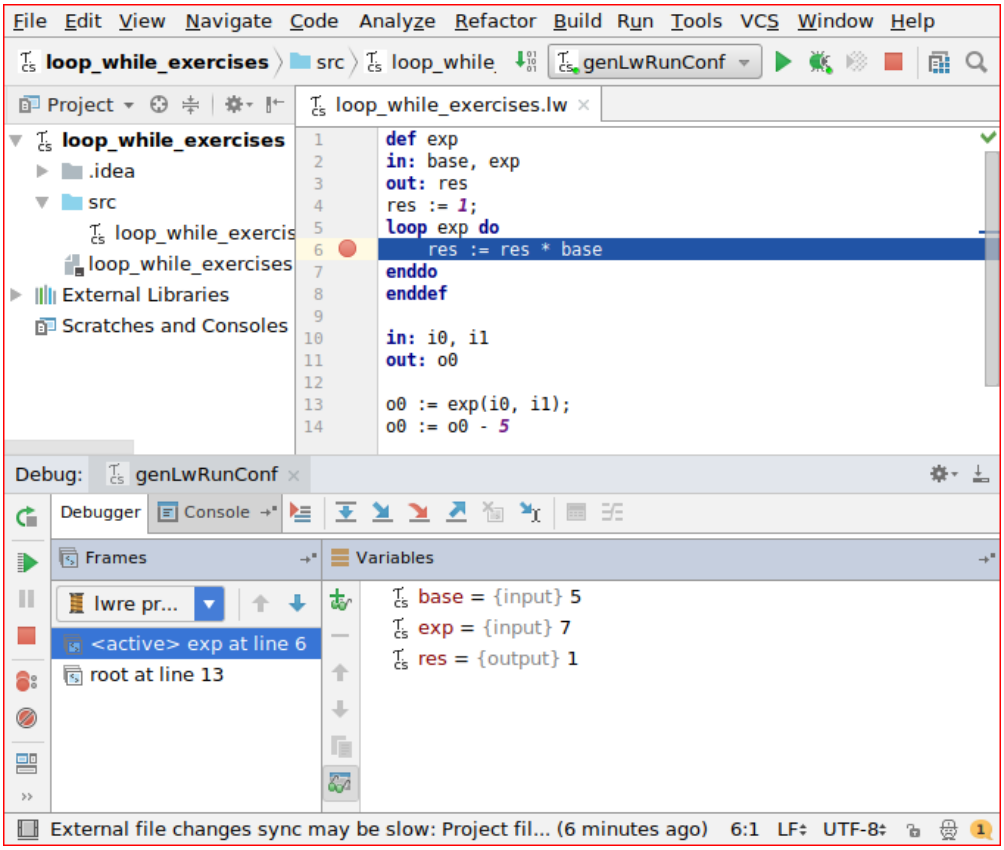
Hit the Debug button , to start the debugger.

Figure B.31



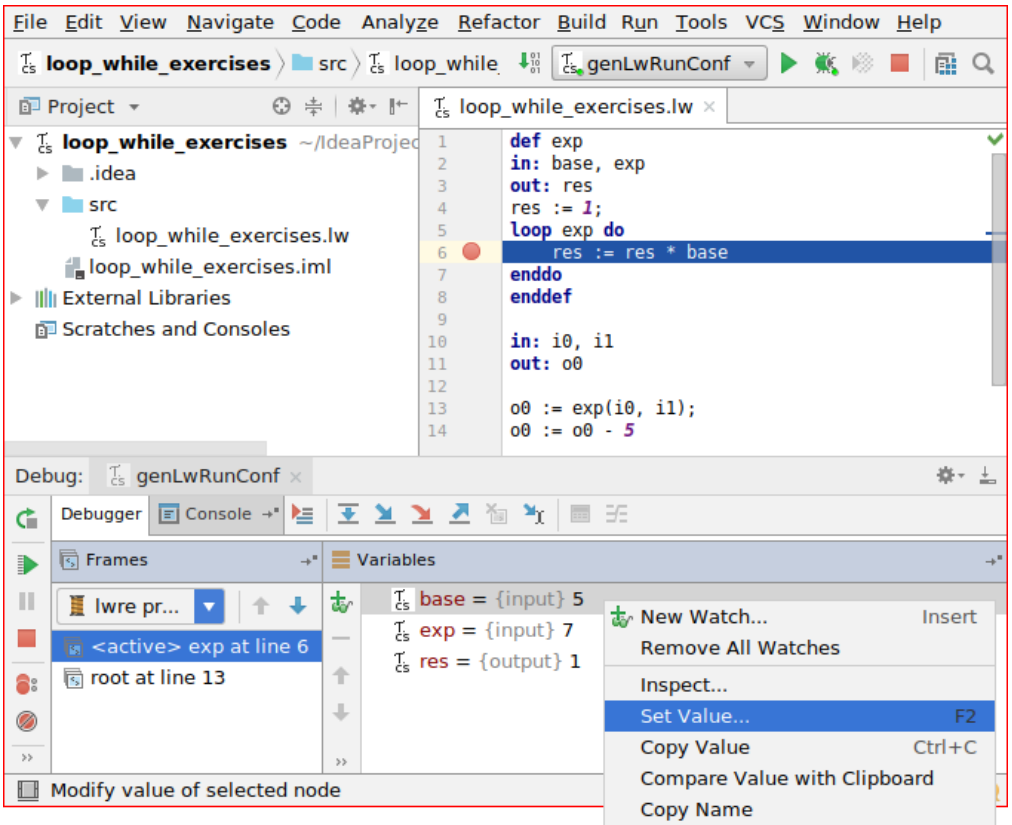
The debugger started. Enter values for the input variables.

Figure B.32



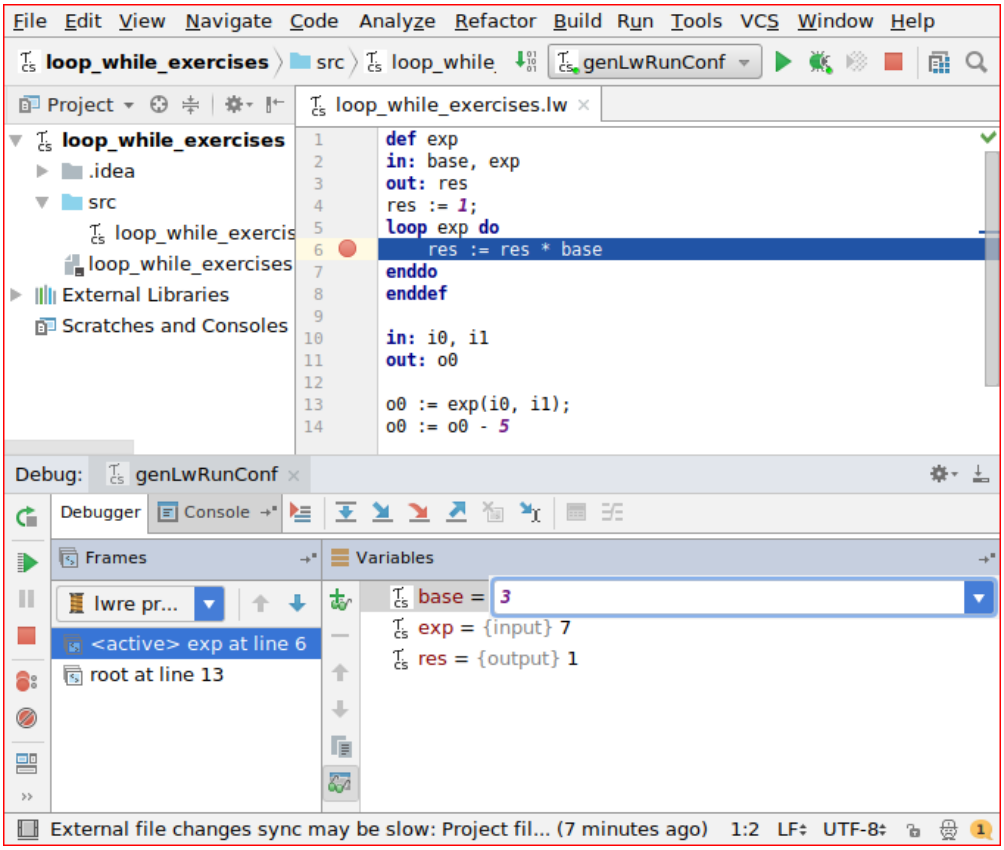
The debugger reached your breakpoint. The `debugger view` (Figure 4.6) shows the `stack trace` and `variable bindings`. The `stack trace` on the left shows through which macro calls the execution reached the current line. In our case, the current macro `exp` was called from the root program at line 13. The `variables view` on the right show all declared variables and their values for the scope which is selected in the `stack trace`.

Figure B.33



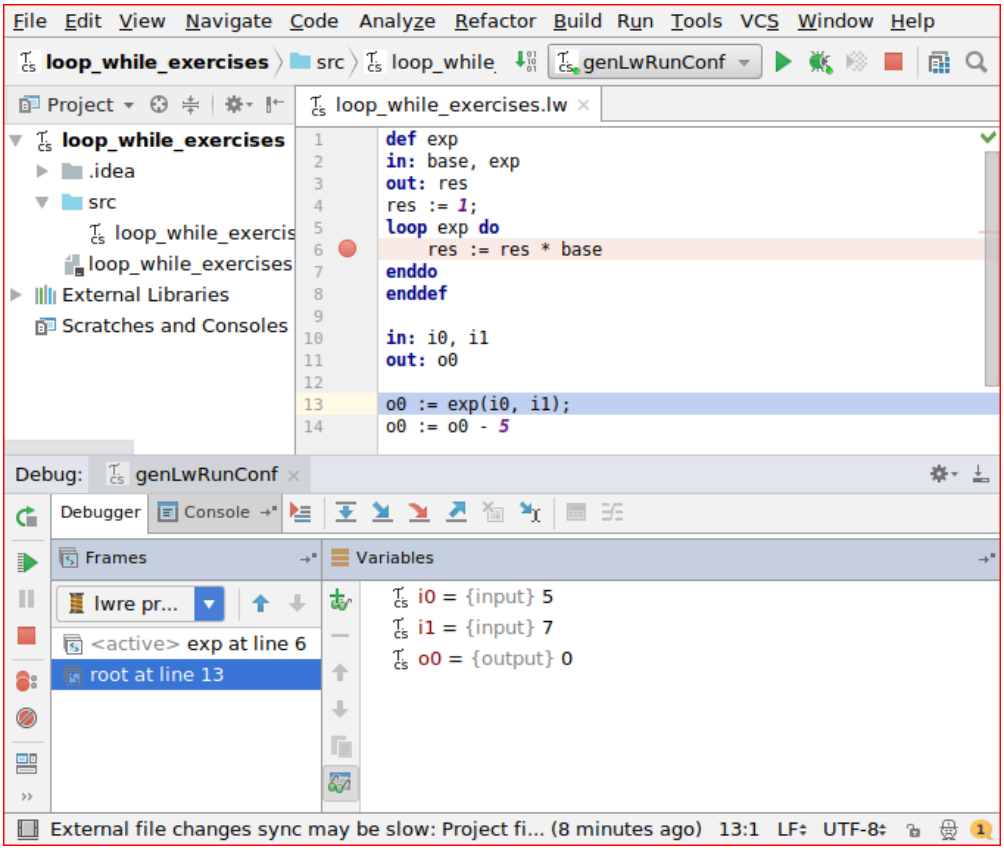
In the `variables view` you can right click on a variable and change its value, by clicking on `Set Value...`.

Figure B.34



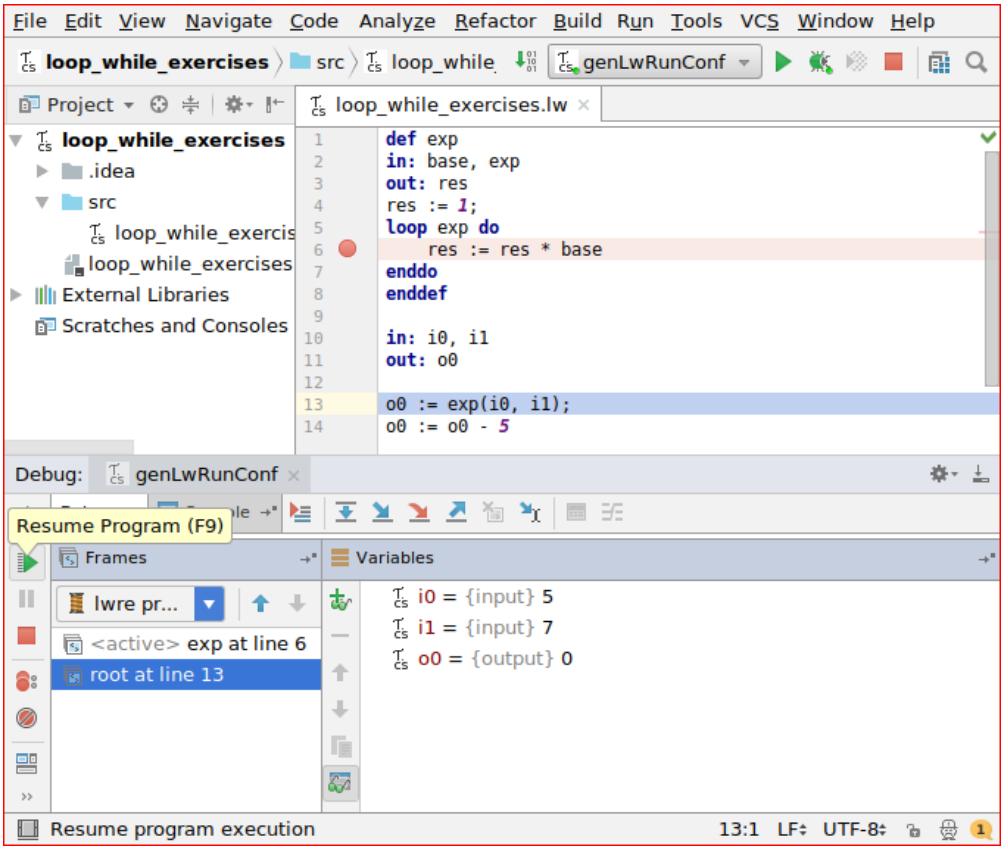
Enter a new value for `base`.

Figure B.35



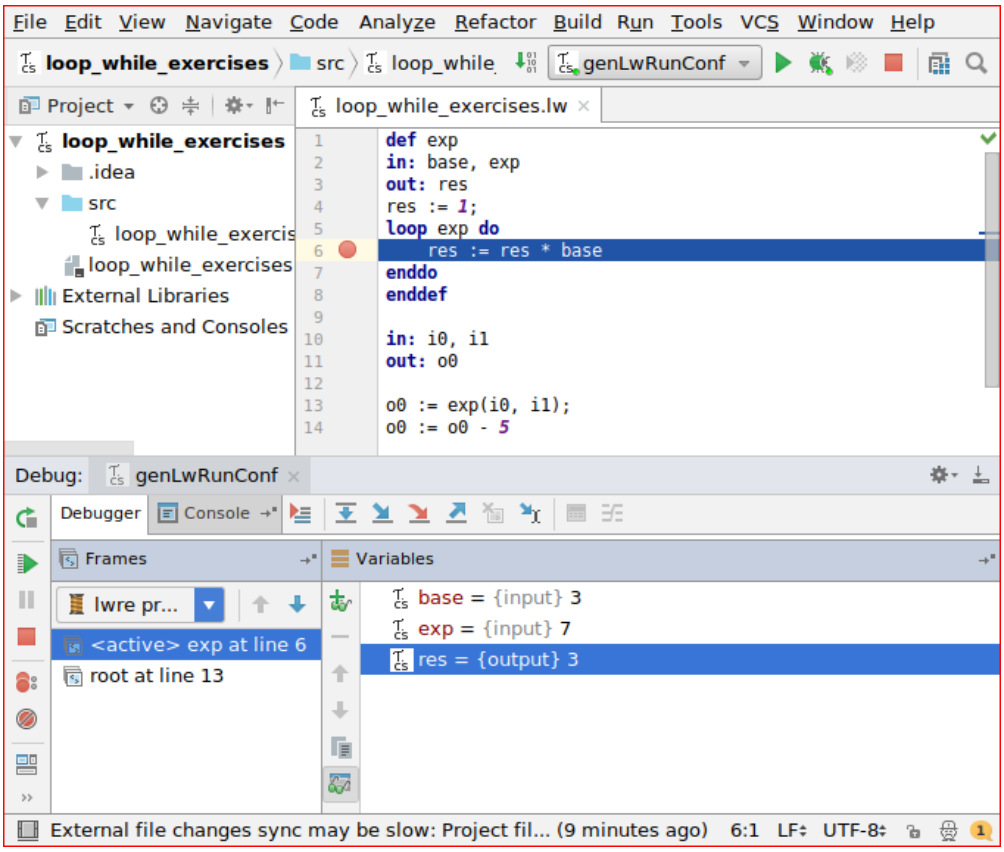
You can select a different scope in the `stack trace` on the left. The editor now highlights the line from which the current macro was called. The `variables view` shows the variables in the selected scope. Note that `i0`, which was used as input for the `base` variable, was not changed along with `base`.

Figure B.36



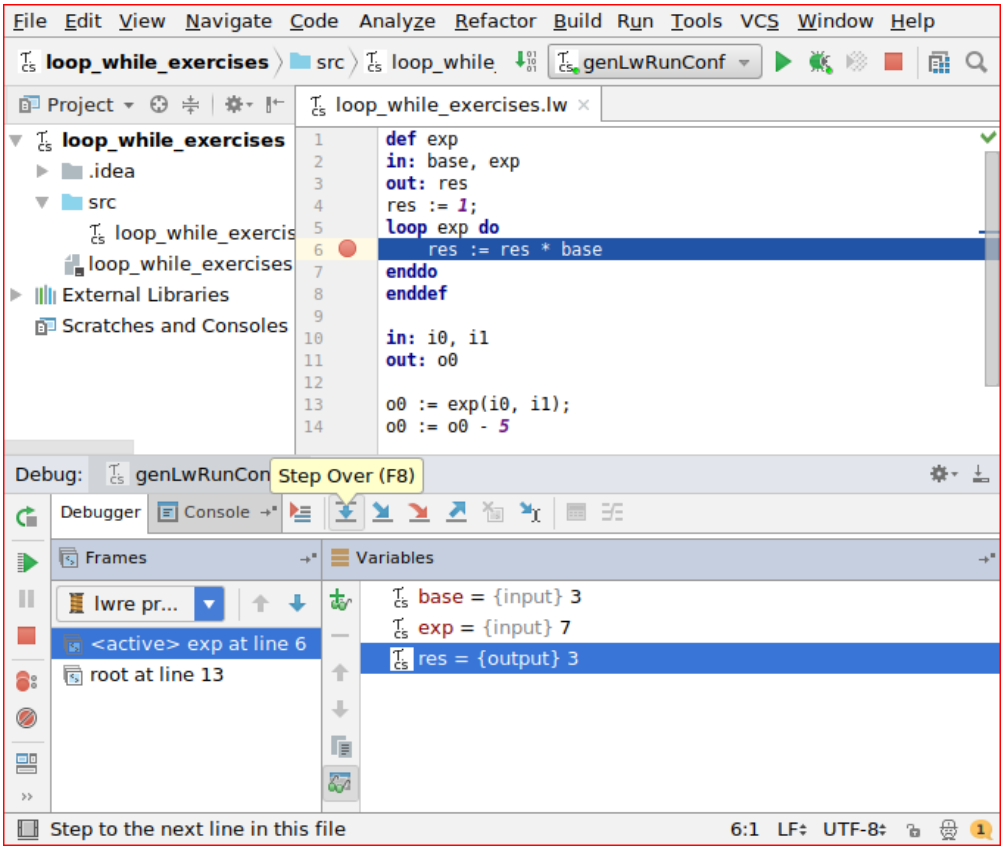
Now hit `Resume` to continue execution.

Figure B.37



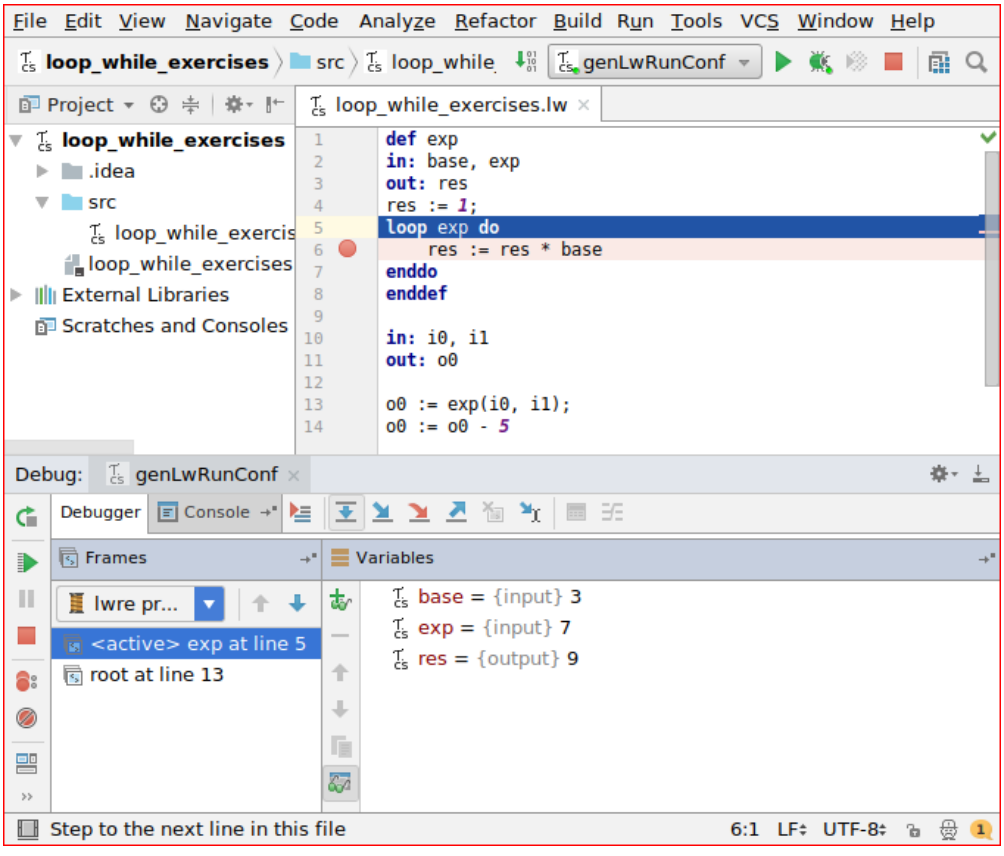
The debugger reached our breakpoint in the next loop iteration. Note that our changed `base` value has now been multiplied onto `res`.

Figure B.38



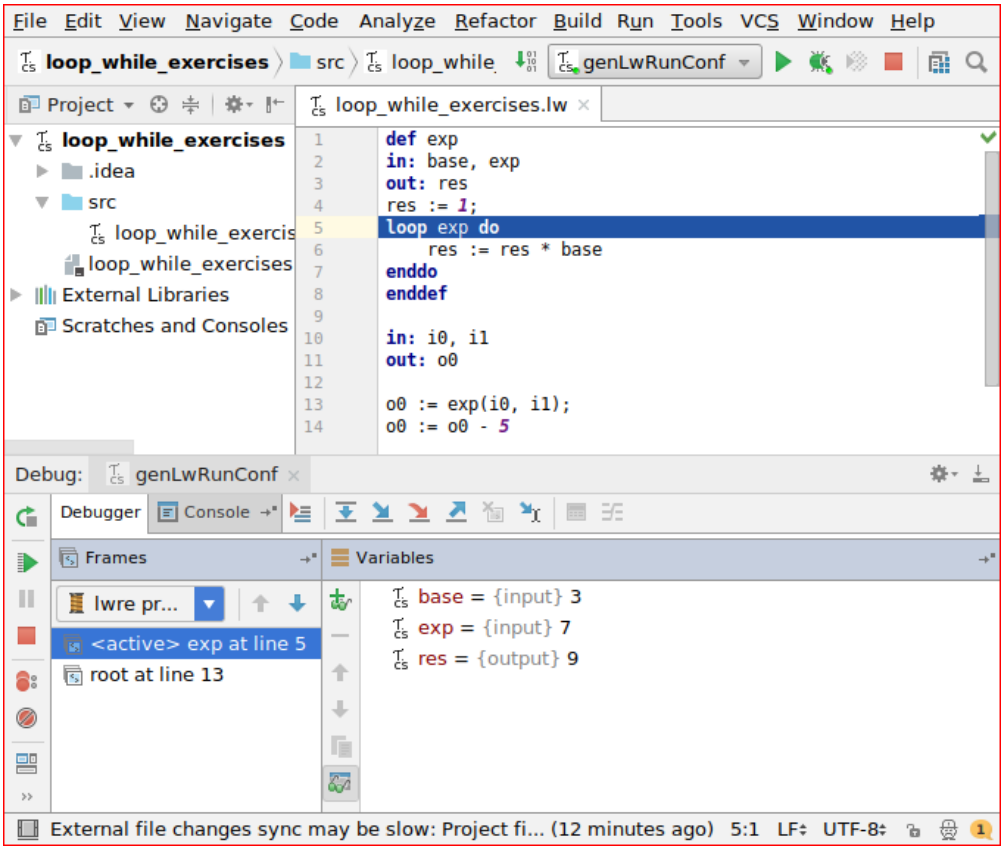
You can hit `Step Over` to move through the current macro line by line. `Step Over` will not descend into any macro calls. You could use `Step Into` for that. With `Step Into` you can execute every single instruction in a program one by one.

Figure B.39



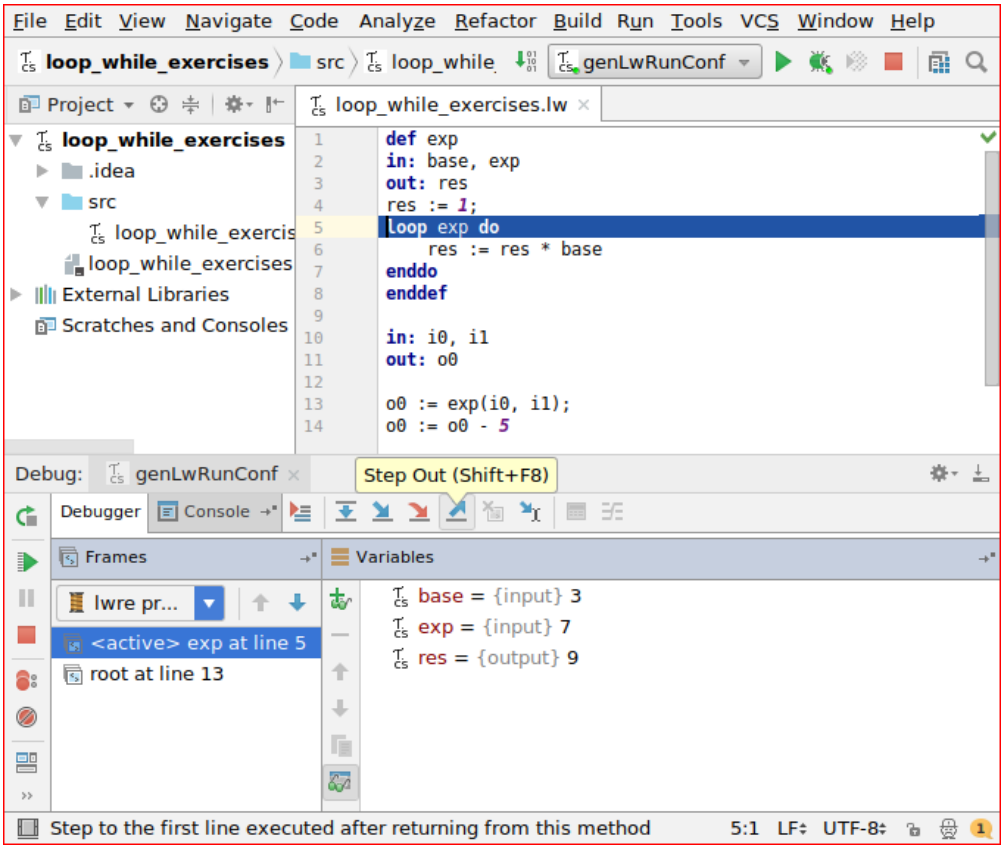
The `stack trace` and the editor now show you, that execution is in line 5, at the loop condition. With another step, execution will go back to line 6, until after the last loop iteration.

Figure B.40



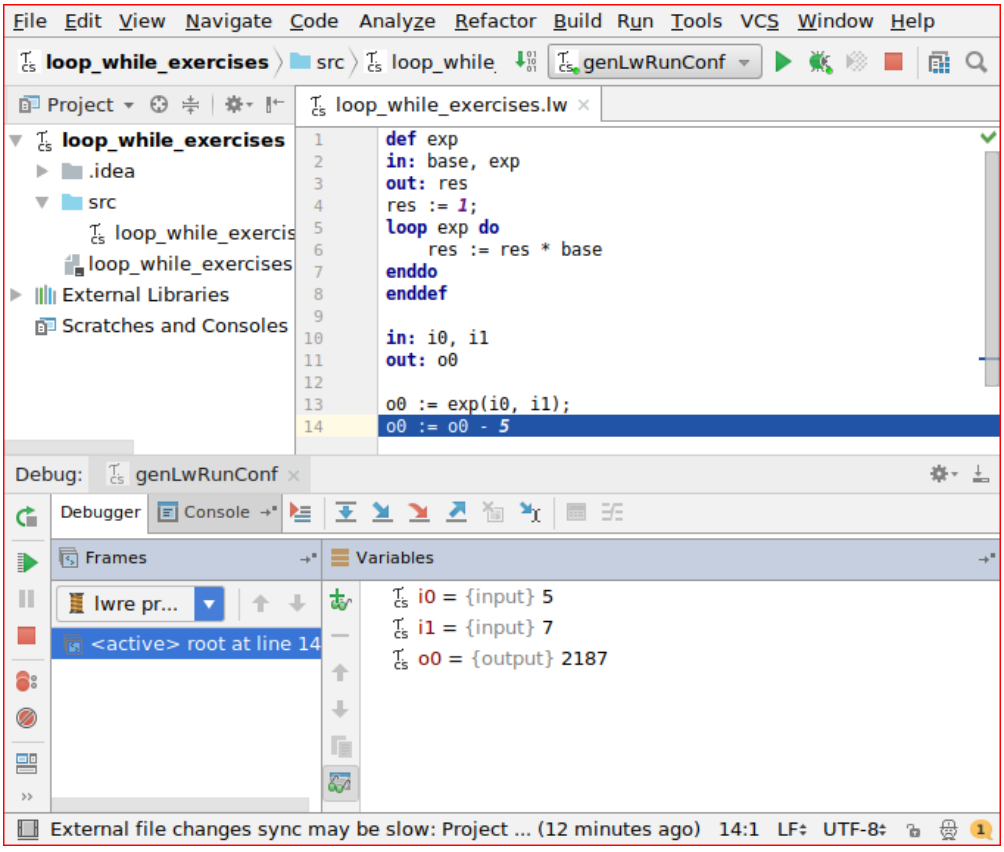
Remove the breakpoint in line 6, by clicking on the breakpoint icon next to the line-number.

Figure B.41



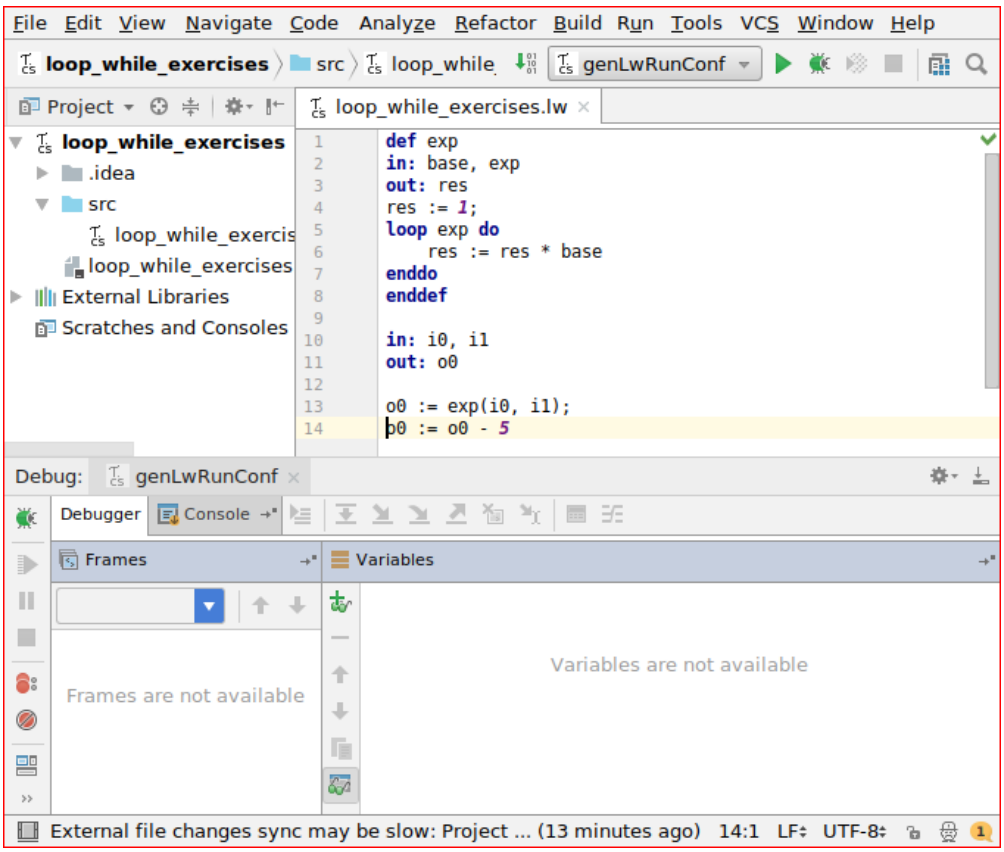
Hit `Step Out` to continue execution up to the first instruction after the call the macro we currently are inside.

Figure B.42



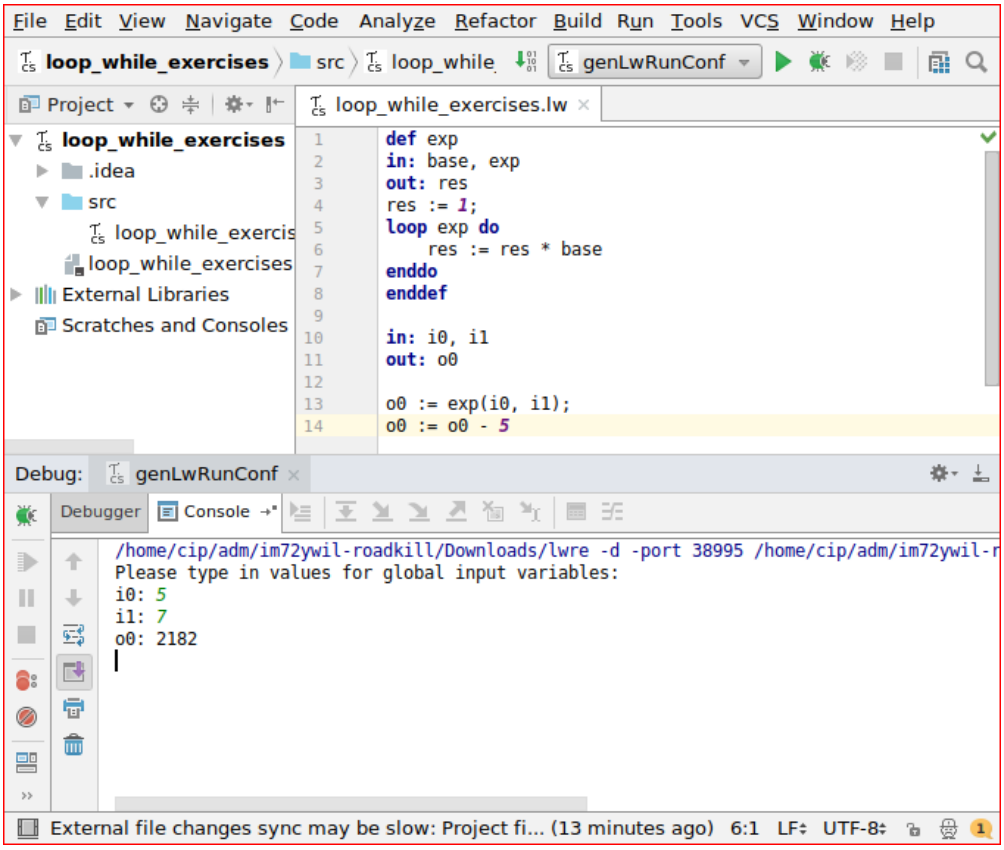
Execution stopped in line 14, right after the call to `exp`. Note that `o0` now contains 3^7, since we altered the value of `base` in `exp`. Hit `Resume` to finish the program.

Figure B.43



Execution has finished. You can click on `Console` to switch back to the `console view` in the `debugger view` and view the output of the interpreter.

Figure B.44



The program successfully computed $3^7 - 5$.

Figure B.45

B.4 Command Line

The interpreter and debugger can also be run without the IntelliJ plugin from command line. The interpreter produces the same output on command line as it does in IntelliJ's `runner view`. Listing B.46 shows how to run the interpreter from command line, along with the output it produces with the example code from Figure B.26.

Listing B.46: Running Interpreter from Command Line

```
> /path/to/lwre /path/to/example/file.lw
Please type in values for global input variables:
i0: 15
o0: 25
```

`text` : user input

Listing B.47 shows the debugging example from Section B.3 run on commandline. The interpreter can be started in debugging mode by adding the `-d` flag. It then waits for debugging commands, such as `setbreakpoint <line> <file>`, until given the `run` command. Now it asks for input variable values if there are any, runs to the first breakpoint, prints a stack trace and awaits more debugging commands. Then it executes each debugging command until the program is finished. Finally it prints the resulting value of the output variable.

Listing B.47: Running Debugger from Command Line

```
> /path/to/lwre -d /path/to/example/file.lw

setbreakpoint 6 file.lw

1 6 0# Breakpoint 0 set.

run

Please type in values for global input variables:
i0: 5
i1: 7

0 6 0 [{"file":"/path/to/example/file.lw","line":6,"macro":"exp",
"bindings":[{"VarType":"input","Ident":"base","Val":5},{"VarType"
:"input","Ident":"exp","Val":7},{"VarType":"output","Ident":"res"
,"Val":1}]},{"file":"/path/to/example/file.lw","line":13,"macro":
"root","bindings":[{"VarType":"input","Ident":"i0","Val":5},{"Var
Type":"input","Ident":"i1","Val":7},{"VarType":"output","Ident":"
o0","Val":0}]}]# Breakpoint 0 reached.

setvariable exp base 3

resume

0 6 0 [{..."Ident":"base","Val":3...}]# Breakpoint 0 reached.

stepover

2 5 -1 [{...}]# Stepped.

clearbreakpoint 6 file.lw

stepout

2 14 -1 [{"file":"/path/to/example/file.lw","line":14,"macro":"ro
ot","bindings":[{"VarType":"input","Ident":"i0","Val":5},{"VarTyp
e":"input","Ident":"i1","Val":7},{"VarType":"output","Ident":"o0"
,"Val":2187}]}]# Stepped.

resume

o0: 2182
```

`text` : user input

Bibliography

[1] Golex. `https://godoc.org/github.com/cznic/golex`.

[2] Goyacc. `https://godoc.org/golang.org/x/tools/cmd/goyacc`.

[3] Marilza Antunes de LEMOS and Leliane Nunes de BARROS. A didactic interface in a programming tutor. In *Proceedings of 11th International Conference on Artificial Intelligence in Education (AIED2003)*, 2003.

[4] Alexander Dietsch. A Simulator for LOOP and WHILE. `https://cal8.cs.fau.de/redmine/projects/lw-simulator`, 2016.

[5] Google. Golang. `https://golang.org/`.

[6] Dirk W Hoffmann. *Theoretische Informatik*. Carl Hanser Verlag GmbH Co KG, 2018.

[7] Sergey Ignatov. Erlang Plugin. `https://github.com/ignatov/intellij-erlang`.

[8] JetBrains. IntelliJ IDEA. `https://www.jetbrains.com/idea/`.

[9] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[10] John R. Levine, Tony Mason, and Doug Brown. *Lex & Yacc (2Nd Ed.)*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1992.

[11] Bruno Cardoso Lopes and Rafael Auler. *Getting Started with LLVM Core Libraries*. Packt Publishing, 2014.

[12] Albert R. Meyer and Dennis M. Ritchie. The complexity of loop programs. In *Proceedings of the 1967 22Nd National Conference*, ACM '67, pages 465–469, New York, NY, USA, 1967. ACM.

[13] Stefan Milius. Theoretische Informatik für Wirtschaftsinformatik und Lehramt. `https://www8.cs.fau.de/ss18:tiet`.

[14] Suyog Sarda and Mayur Pandey. *LLVM Essentials*. Packt Publishing, 2015.

[15] Uwe Schöning. *Theoretische Informatik*. Vorlesungsskript, 1995.

[16] David A. Watt. *Programming Language Processors*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[17] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA, 1993.