The SLANG Semantics-Based Language Generator

Wolfgang Schreiner, William Steingartner

September 2023

RISC Report Series No. 23-13

ISSN: 2791-4267 (online)

Available at https://doi.org/10.35011/risc.23-13

Supported by: Supported by the Slovak Academic Information Agency SAIA project 2023-03-15-001 “Semantics-Based Rapid Prototyping of Domain-Specific Languages”

The SLANG Semantics-Based Language Generator*

Tutorial and Reference Manual (Version 1.0.*)

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
Wolfgang.Schreiner@risc.jku.at

William Steingartner
Department of Computers and Informatics
Technical University of Košice, Slovak Republic
William.Steingartner@tuke.sk

September 20, 2023

Abstract

This report documents the SLANG semantics-based language generator. SLANG is a software for generating rapid prototype implementations of programming languages from their formal specifications. Its input is a text file that describes the abstract syntax of a language and its concrete text representation; from this, a parser is generated (utilizing the ANTLR4 tool) that transforms the text representation of a program into its abstract syntax tree and a printer that generates from the abstract syntax tree its text representation. Furthermore, one can equip the language with a formal type system (by logical inference rules) from which a type checker is generated. Finally, one can give the language a formal semantics, in the denotational style (by function equations) and/or in the big-step operational style (by transition steps); from this, a language interpreter is generated. SLANG is implemented in Java and produces Java source code; it should be easy to extend the software also to other target languages.

This document will be continuously revised; its most recent version can be found at the following URL: `https://www.risc.jku.at/research/formal/software/SLANG`

*Supported by the Slovak Academic Information Agency SAIA project 2023-03-15-001 “Semantics-Based Rapid Prototyping of Domain-Specific Languages”.

Contents

1 Introduction

SLANG is a tool for the rapid prototyping of programming languages by automatically generating implementations of these languages (as Java source code) from their formal specifications. There already exist numerous tools for the implementation of programming languages, most notably the class of tools variously called "compiler-compilers" or "compiler generators" or, more accurately, "parser generators" and "lexical analyzer generators"; a prominent example is ANTLR (ANother Tool for Language Recognition) [7, 1]. However, these tools mainly support the syntactic process of language recognition by the generation of lexical analyzers from regular expressions and parsers from context-free grammars. The actual implementation of a program's behavior is essentially left to the human software developer, which typically requires a lot of effort.

Traditionally these developers have been experts on the design and implementation of flexible general-purpose programming languages; nowadays, however, more and more also professionals in other fields have to develop specialized languages for specific tasks, i.e., *Domain-Specific Languages* (DSLs) [6, 3, 15]. For this target group, it would be very helpful if the complete process of language development could be guided and supported by systematic processes and supporting tools, where the focus should be on the soundness of the design and the correctness of the implementation of a language rather than on the performance of its execution.

Actually such a systematic implementation process is suggested by approaches to formally define the semantics of programming languages as, e.g., described in [9, 10]. Here the starting point of language design is the definition of a domain of abstract syntax trees that represent syntactically well-formed programs; such trees are generated from a textual representation by a parser in combination with a lexical analyzer. The core of the design then consists of the definition of mathematical functions or logical relations by structural induction on the abstract syntax trees. In particular, the type system of the language can be represented as a relation defined by a logical inference system; its semantics can be represented as a function that maps syntax trees to functions that map program inputs to program outputs ("denotational semantics") or alternatively as relations that describe the reduction of the current state of a program to its final result ("operational semantics"). The focus of SLANG is to support the formalization of syntactic domains and the definition of functions and relations on these domains by structural induction. Ultimately, these mathematical functions and logical relations are translated to Java methods, thus yielding an executable implementation of the language.

There is already quite some history of tool support for the formal semantics of programming languages: the Maude MSOS tool [2] is an executable framework for specifications in *modular structural operational semantics* (MSOS, a version of small-step operational semantics) via translation to the language of the Maude rewriting system. The Ott tool [13] implements a meta-language for semantic definitions that are translated to specifications for the proof assistants Coq, HOL, and Isabelle/HOL; the focus of Ott is not on executing the programming language semantics but on mechanized proofs of soundness results. The K semantic framework [8] is a rewrite-based execution framework for small-step operational semantics based on an internal interpreter and theorem prover. Especially the K framework has been used to formalize the semantics of (subsets of) various real-life programming languages such as C, Java, or Python.

The SLANG tool differs from these tools in some central aspects. First, above tools focus mostly on small-step operational semantics where essentially an abstract syntax tree is gradually rewritten to a final result; this requires special support by a rewrite engine or interpreter (the Ott tool produces proof assistant code from which executable code can be extracted, but this seems not to have been substantially exploited). In contrast, SLANG focuses on denotational semantics and big-step operational semantics both of which can be straight-forwardly implemented in any programming language by structural recursion. In particular, while in above tools the semantics is only executable in separate systems (Maude or the K interpreter), SLANG generates executable Java code that can be freely imported into existing application frameworks.

Furthermore, above tools completely describe the language semantics in specific meta-languages; thus all the mathematical entities (types and operations) on which a semantics depends has to be completely formalized in the meta-language. While this allows formal reasoning within the respective frameworks or via external proof assistants, it also requires substantial efforts and may also lead to prohibitively inefficient executions. SLANG on the other side only provides a minimal meta-language to express structurally recursive definitions of functions or relations; all the computations are expressed by embedded Java code. While this prevents formal reasoning about the specifications, it allows to utilize external (standard or user-defined) libraries for the mathematical entities required in the semantics; the effort to develop a semantics is thus greatly reduced and its implementation is potentially much more efficient.

All in all, SLANG is thus a *pragmatic* tool whose goal is (rather than being theoretically elegant and complete) to minimize the effort of developing a rapid prototype of a language and to maximize the usefulness of this prototype by making it easy to integrate in existing software. The software is freely available as open source at the following URL:

https://www.risc.jku.at/research/formal/software/SLANG

The core of this document is Section 2 that essentially represents a tutorial of SLANG by showing how to use the tool for the implementation of a simple imperative programming language. Section 3 outlines our further plans for the development of the software. Appendix A describes the installation and use of the SLANG software. Appendix B gives the grammar of SLANG in ANTLR syntax; consult this in case of syntax problems and also to learn about aspects of the language not described by the examples. Appendix C lists examples of language definitions that are distributed with the software.

2 An Imperative Programming Language

In this section, we are going to use SLANG to implement a simple imperative programming language which we call `Imp`. A denotational semantics of this language was defined in Chapter 9 “Programming Languages” of [11]; similar languages have been formalized, e.g., in [16, 5].

The following is an example of a program in this language, which is expected to compute by repeated addition for $n = 5$ the square $a = n^2 = 25$.

```
var n; var a; var i;
n:=5; while ~(i = n) do { var j; j := 2*i+1; a:=a+j; i:=i+1 }
```

In the final Subsection 2.8, we will see that this goal is indeed achieved by the SLANG implementation of `Imp`.

Since SLANG specifications embed Java source code, we assume in the following sections familiarity with this programming language.

2.1 Specification Files

SLANG specifications are written in UTF-8 encoded text files and may make use of non-ASCII Unicode symbols; alternatively one may also use multi-character ASCII tokens instead:

Unicode symbol	ASCII token
⊢	`|-`
⟨	`<<`
⟩	`>>`
⟦	`[[`
⟧	`]]`
→	`->`
×	`*`

The full SLANG specification file `Imp.txt` for the implementation of this language is given in Subsection C.2; its content has the following structure:

```
// ----------------------------------------------------------------------------
// Imp.txt
// SLANG version of the *denotational* semantics of a simple imperative language
// (c) 2023 https://www.risc.jku.at/research/formal/software/SLANG
// Wolfgang.Schreiner <Wolfgang.Schreiner@risc.jku.at>
// William Steingartner <william.steingartner@tuke.sk>
// ----------------------------------------------------------------------------
language Imp
{
  target java
  {
    header
    {#
      package lang.imp;
      import java.util.*;
      import static lang.imp.Imp.*;
    #}
  }
  ...
}
```

Language specifications may include line comments starting with `//` or block comments `/*...*/`. The clause `target java` indicates that the generated source code will be in the programming language Java (the only supported target language at the moment). The clause `header` lists code that is added *verbatim* at the beginning of every generated Java source file. This code is surrounded by parentheses of the form `{# #}` which may embed arbitrary source code in the target language. Actually, such source code may be embedded in one of the following forms:

```
{#...#}
{##...##}
#...#
##...##
```

The two forms with double occurrences of the token `##` can be used to embed source code that itself contains individual occurrences of the character `#`. As a convention, we use the forms with curly braces `{...}` to embed multiple source code commands and the forms without these braces to embed single commands or just parts of commands (but technically there is no distinction between these two kinds of forms). The syntactic and semantic correctness of the embedded code is not checked by the SLANG software itself; errors will become apparent only when the source code files generated by SLANG are compiled with a Java compiler.

From the part of file `Imp.txt` that was abbreviated as `...` in the specification snippet above, the SLANG software generates the following entities:

- the syntactic domains of the language, i.e., the types of its abstract syntax trees, according to a specified context-free grammar,
- a printer that transforms an abstract syntax tree according to a specified concrete syntax into a linear text form,
- a parser that checks whether a given text is well-formed according the concrete syntax and transforms it into an abstract syntax tree,
- a type checker that determines whether the abstract syntax tree is well-typed according to a specified type system,
- an interpreter that evaluates the abstract syntax tree with respect to a specified formal semantics.

In the following, we will elaborate the corresponding parts of the specification in turn and then describe the generation, compilation, and execution of the Java code that implements the entities listed above.

2.2 Syntactic Domains

The syntactic domains of the language are described by the following `domains` clause:

```
// ----------------------------------------------------------------------
// syntactic domains
// ----------------------------------------------------------------------
domains
{
  Exp = Num[NUM] + Id[ID] + Plus[Exp,Exp] + Times[Exp,Exp];
  BoolExp = Eq[Exp,Exp] + Not[BoolExp] + And[BoolExp,BoolExp];
  Command = Assign[ID,Exp] + Var[ID,Command] + Seq[Command,Command]
          + If2[BoolExp,Command,Command] + If1[BoolExp,Command]
          + While[BoolExp,Command];
  Program = Prog[Command];
}
```

The clause consists of a sequence of declarations each of which is terminated by a `;` and introduces a syntactic domain:

- the domain `Exp` of integer (numeric) expressions,
- the domain `BoolExp` of Boolean (truth) expressions,
- the domain `Command` of program commands,
- the domain `Program` of programs.

Each element of these domains (that must have all different names) is represented by an abstract syntax tree whose possible forms are denoted by the options on the right-hand side of the equality symbols; each option has form `Tag[`D_1`,...`D_n`]` where *Tag* is a fresh identifier (different from all other identifiers in all options in all domains) and $D_1,\ldots D_n$ is a sequence of $n \geq 0$ domain names separated by commas; each domain name is either defined in the clause or represents a predefined domain (see below) or is of the form *#T#* where T is an embedded Java class (thus the syntax tree may accommodate an object that can be modified by the type checker to annotate the abstract syntax tree with information determined by static analysis).

For instance, in above example, the option `Num[NUM]` denotes the possible construction of a tree from domain `Exp` whose root has label `Num` and a single child from the pre-defined domain `NUM`. The option `Eq[Exp,Exp]` denotes the possible construction of a tree from domain `BoolExp` whose root has label `Eq` and two children from domain `Exp`; the option `If1[BoolExp,Command]` denotes the possible construction of a tree from domain `Command` that has label `If1` and two children from domain `BoolExp` and `Command`, respectively.

In addition to the user-defined syntactic domains, we have various pre-defined domains; the elements from these domains are not trees but particular strings of characters:

- the domain `ID` of identifiers; an identifier is a non-empty sequence of characters each of which may be an (upper-case or lower-case) letter, the underscore character _, or also a decimal digit (but the first character of the identifier must not be a digit).
- the domain `NUM` of decimal numbers; a decimal number is a non-empty sequence of decimal digits;
- the domain `STR` of character strings (not used in the definitions above); a string is an arbitrary (possibly empty) sequence of characters.

A tree from a user-defined domain with option `Tag[`D_1`,...`D_n`]` can be constructed by a term of form `Tag[`t_1`,...,`t_n`]` where the $t_1,\ldots,t_n$ are terms that denote subtrees from domains

$D_1, \ldots D_n$, respectively. The elements of the predefined domains are typically only constructed by application of the parser explained later. However, they may be also constructed by embedded Java string literals `#"..."#` with the text of the element.

As an example, the following term constructs an abstract syntax tree from domain `Program`, i.e., an `Imp` program:

```
Prog[Var[#"x"#,Seq[
  Assign[#"x"#,Plus[Num[#"2"#],Num[#"3"#]]],
  If[Eq[Id[#"x"#],Num[#"5"#]],
    Assign[#"x"#,Times[Num[#"2"#]],Id[#"x"#]]]]]]
```

In the concrete syntax described in Subsection 2.3, this program is represented as follows:

```
var x; x := 2+3; if x = 5 then x := 2*x;
```

2.3 Printer

The clause `printer` describes the concrete text representation of the abstract syntax trees of the user-specified domains:

```
// ------------------------------------------------------------------------
// printer
// ------------------------------------------------------------------------
printer
{
  domain Exp
  {
    case Num[n] → # _result = n; #;
    case Id[i] → # _result = i; #;
    case Plus[e1,e2] → # _result = "(" + e1 + "+" + e2 + ")"; #;
    case Times[e1,e2] → # _result = "(" + e1 + "*" + e2 + ")"; #;
  }
  domain BoolExp
  {
    case Eq[e1,e2] → # _result = e1 + " = " + e2; #;
    case Not[b] → # _result = "(~" + b + ")"; #;
    case And[b1,b2] → # _result = "(" + b1 + " /\\ " + b2 + ")"; #;
  }
  domain Command
  {
    case Assign[i,e] → # _result = i + " := " + e; #;
    case Var[i,c] → # _result = "{var " + i + "; " + c + "}"; #;
    case Seq[c1,c2] → # _result = "{" + c1 + "; " + c2 + "}"; #;
    case If2[b,c1,c2] → # _result = "if " + b + " then " + c1 + " else " + c2; #;
    case If1[b,c] → # _result = "if " + b + " then " + c; #;
    case While[b,c] → # _result = "while " + b + " do " + c; #;
  }
  domain Program
  {
    case Prog[c] → # _result = c.toString(); #;
  }
}
```

The clause has one section for each user-specified domain and one rule for each option of the domain of the following form (please note the terminating semicolon):

case *Tag*[x_1,...,x_n] → # ... # ;

Each rule is distinguished by the *Tag* of the corresponding root node parameterized by variables x_1,...,x_n each of which represents one subtree of the node. The right side of the rule embeds Java code that sets the `String` variable `_result` to the string representation of the tree using the following interpretation of every variable x_i:

- If x_i represents an element of a predefined syntactic domain, in the embedded Java code the variable denotes an object of type `String` which holds the text content of the element.
- If x_i represents an abstract syntax tree from a user-defined syntactic domain, in the embedded Java code it represents an object whose standard method `toString()` returns the representation of the corresponding subtree as a value of type `String`; as usual, this method is implicitly called if x_i is an argument of the Java string concatenation operator +.

The embedded Java code may be a single command or a sequence of commands; in the later case, it should be grouped in a block `{...}` to avoid any potentially unintended parsing of the ultimately generated code.

2.4 Parser

The clause `parser antlr4` describes the transformation of plain text into abstract syntax trees utilizing the syntax of the ANTLR4 tool for language recognition [7, 1] (no other tool is supported at the moment):

```
// --------------------------------------------------------------------------
// parser
// --------------------------------------------------------------------------
parser antlr4
{
  domain Exp
  {
    case # n=dNUM # → Num[n];
    case # i=dID # → Id[i];
    case # e1=dExp '*' e2=dExp # → Times[e1,e2]; // higher priority first
```

```
      case # e1=dExp '+' e2=dExp # → Plus[e1,e2];
      case # '(' e=dExp ')' # → # $_result = $e._result; #; // parenthesing
    }
    domain BoolExp
    {
      case # e1=dExp '=' e2=dExp # → Eq[e1,e2];
      case # '~' b=dBoolExp # → Not[b];
      case # b1=dBoolExp '/\\' b2=dBoolExp # → And[b1,b2];
      case # '(' b=dBoolExp ')' # → # $_result = $b._result; #; // parenthesing
    }
    domain Command
    {
      case # i=dID ':=' e=dExp # → Assign[i,e];
      case # 'if' b=dBoolExp 'then' c1=dCommand 'else' c2=dCommand # →
        If2[b,c1,c2];
      case # 'if' b=dBoolExp 'then' c=dCommand # → If1[b,c];
      case # 'while' b=dBoolExp 'do' c=dCommand # → While[b,c];
      case # c1=dCommand ';' c2=dCommand # → Seq[c1,c2];
        // binds weaker than if/while
      case # 'var' i=dID ';' c=dCommand # → Var[i,c];
        // binds weaker than sequence
      case # '{' c=dCommand '}' # → # $_result = $c._result; #; // parenthesing
    }
    domain Program
    {
      case # c=dCommand EOF # → Prog[c];
    }
  }
```

The clause has one section for each user-specified domain each of which contains rules of the following form (please note the terminating semicolon):

```
  case # ... # → ... ;
```

In a rule for `domain` *`Domain`*, the left-hand side of the rule describes an option of a ANTLR4 parser rule for a concrete syntax domain `d`*`Domain`* (please note the leading character `d`; ANTLR4 domains have to start with lower-case letters). Typically this option mimics an option of the abstract syntax definition, by having for the abstract syntax domain D_i of every subtree a phrase x_i=dD_i that assigns to some variable x_i the corresponding abstract syntax tree. Otherwise the left side of the rule contains concrete syntax tokens of form `'...'` that define the linear text representation of an abstract syntax tree from domain *`Domain`*. The right-hand side of the rule is then typically a term $Tag[x_1,...,x_n]$ that describes the construction of this abstract syntax tree.

The generated parser attempts the rules in order, i.e., rules that appear earlier in the listing have higher precedence. Therefore, for example, the rule for `If2` is given before the rule for `If1`; this ensures that in an `if` command a succeeding `else` branch is immediately considered to be part of the command. Similarly, the rule for the command composition operator `;` appears after the rules for the atomic commands; the rule for the variable declaration operator `var` appears even later.

The concrete grammar may have additional rules that ensure by the application of corresponding parentheses that nested syntactic phrases are parsed as intended, see, e.g., the last rules of domains `Exp`, `BoolExp`, and `Command` above. In order to avoid the necessity for corresponding nodes in the abstract syntax trees (which already defined the unique structure by their shape), the right side of a rule may be a Java embedded command that sets the variable `_result` of type *`Domain`* to the value of the subtree surrounded by the parentheses; if this subtree has been on the left side of the rule assigned to variable *`x`*, we may refer to it on the right side as `$`*`x`*`._result`.

2.5 Type Checker

Not every program that can be correctly parsed to an abstract syntax tree indeed makes sense; to eliminate meaningless programs, we have to ensure that programs follow some typing discipline. We follow the approach stated in [10] and also described in Chapter 1 "Syntax and Semantics" of [11]: there a type system is a logical inference system where every abstract syntax domain is equipped with a judgment that states whether a phrase is well-typed, potentially depending on the context of the phrase (from which information is passed to the judgment), and potentially also determining extra information that may be passed to other judgments. Furthermore, we may annotate well-typed phrases with this extra information for later utilization by the interpreter that executes the phrases.

To represent such extra information, our SLANG specification file contains a clause with auxiliary definitions embedded in a `code` clause:

```
// ----------------------------------------------------------------------
// type system
// ----------------------------------------------------------------------
code
{#
  public enum Type { Int }
  public static class Env<T> extends HashMap<String,T>
  {
    public Env() { super(); }
    public Env(Env<T> e) { super(e); }
    public T put(String key, T value)
    { return super.put(key.toString(), value); }
    public T get(String key)
    { return super.get(key.toString()); }
  }
  public static class TypeError extends RuntimeException
  { public TypeError(String msg) { super(msg); } }
  public static void check(boolean b, String msg)
  { if (!b) throw new TypeError(msg); }
#}
```

Such clauses may be added freely (also multiple times) to the specification file; all the embedded code is collected in a Java file and made accessible to all entities of the implementation. In the example above, this code consists of a type `Type` whose values will denote types of expressions; since in our language we only have integer expressions, the only value of this type is the constant `Int`. Furthermore, we have a generic type `Env`<*T*> that represents a partial map of variable names (strings) to values of type T, for arbitrary T. Finally, there is a runtime exception type `TypeError`. An exception of this type is thrown whenever a type error is detected; the exception carries a message that identifies this error. The auxiliary method `check(`*b*`,`*msg*`)` tests whether the boolean value *b* is true; if this is the case, the method returns normally, otherwise it throws a `TypeError` exception that carries the message *msg*.

With the help of these Java entities, we now define the type checking of integer expressions by the following clause:

```
judgment #Env<Type># ⊢ Exp: exp
{
  inference te ⊢ Id[i]: exp
  {
    # check(te.get(i) != null, "undeclared variable " + i); #
  }
  inference te ⊢ Num[n]: exp
  {
  }
  inference te ⊢ Plus[e1,e2]: exp
  {
    te ⊢ e1: exp;
    te ⊢ e2: exp;
  }
  inference te ⊢ Times[e1,e2]: exp
  {
    te ⊢ e1: exp;
    te ⊢ e2: exp;
  }
}
```

The header of the clause introduces a judgment *te* ⊢ *e*:`exp` which states that in type environment *te* the integer expression *e* is well-typed. Here *te* is of type `Env<Type>`, i.e., a partial mapping of variable names to types; for a given variable name *i*, *te* has a value if and only if *i* has been previously declared. Furthermore, *e* is an abstract syntax tree of syntactic domain `Exp`; thus for each option of `Exp` there is a logical inference rule that describes when a syntax tree constructed according to this option is well-typed:

- A variable with name *i* is well typed, if *i* is in the domain of *te*.
- A number literal *n* is always well typed.
- A sum expression and a product expression with subexpressions e_1 and e_2 are well typed if both subexpressions are well typed integer expressions.

The last two rules contain as prerequisites applications of the currently defined judgments, but the applications are to subphrases of the current phrase. The judgment can be thus interpreted as a recursively defined procedure that is guaranteed to terminate; the various rules represent the subprocedures that cover every possible case of the syntactic phrase passed to the procedure as an argument.

In a similar style, we can define the type checker for Boolean expressions:

```
judgment #Env<Type># ⊢ BoolExp: bexp
{
  inference te ⊢ Eq[e1,e2]: bexp
  {
    te ⊢ e1: exp;
    te ⊢ e2: exp;
  }
  inference te ⊢ Not[b]: bexp
  {
    te ⊢ b: bexp;
  }
  inference te ⊢ And[b1,b2]: bexp
  {
    te ⊢ b1: bexp;
    te ⊢ b2: bexp;
  }
}
```

Here the first inference rule checks the well-typedness of the equality of two integer expressions e_1 and e_2; this expression is a well-typed Boolean expressions if both e_1 and e_2 are well-typed integer expressions. However, the negation of b is a well-typed Boolean expression if also b is a well-typed Boolean expression. Likewise, the conjunction of b_1 and b_2 is a well-typed Boolean expression if both subexpressions are well-typed Boolean expressions.

The core of the type checker, however, is the judgment for commands:

```
judgment #Env<Type># ⊢ Command: command
{
  inference te ⊢ Assign[i,e]: command
  {
    # check(te.get(i) != null, "undeclared variable " + i); #
    te ⊢ e: exp;
  }
  inference te ⊢ Var[i,c]: command
  {
    te0: #Env<Type># = # new Env<Type>(te) #; # te0.put(i, Type.Int); #
    te0 ⊢ c: command;
  }
  inference te ⊢ Seq[c1,c2]: command
```

```
    {
      te ⊢ c1: command;
      te ⊢ c2: command;
    }
    inference te ⊢ If2[b,c1,c2]: command
    {
      te ⊢ b: bexp;
      te ⊢ c1: command;
      te ⊢ c2: command;
    }
    inference te ⊢ If1[b,c]: command
    {
      te ⊢ b: bexp;
      te ⊢ c: command;
    }
    inference te ⊢ While[b,c]: command
    {
      te ⊢ b: bexp;
      te ⊢ c: command;
    }
  }
```

Here an assignment of the value of integer expression *e* to a variable with name *i* is only well-typed if *i* has been previously declared and *e* is a well-typed integer expression.

To type-check a block command which declares a variable *i* and then executes the body command *c*, we declare a new type environment te_0 that is identical to *i* except that it also contains a mapping for *i*; then *c* is type-checked with respect to this environment. This inference rule contains an example of a variable declaration `te0: #Env<Type># = #...#;` SLANG thus has now one additional variable `te0` of type `Env<Type>` in addition to the variables listed in the header of the inference rule. It will check that the use of `te` in the subsequent application of an inference rule is well-typed. SLANG considers to types as equal if and only if their names match *exactly*; thus care must be taken the the content of the code expression `#...#` is exactly the same (without additional spaces etc).

The other cases of command sequences, two- and one-sided conditional statements, and while loops only require to check the well-typedness of the Boolean expressions and subcommands in the given type environment.

We are ready to type-check a complete program by the following judgment with a single rule:

```
  judgment ⊢ Program: program
  {
    inference ⊢ Prog[c]: program
    {
      te: #Env<Type># = #new Env<Type>()# ;
      te ⊢ c: command;
    }
    }
```

This inference rule checks that a program with body command c is well typed by constructing an empty type environment *te* and checking c within that environment; thus every variable that is used within the program must be appropriately declared.

Above examples demonstrate the main features of the type checker by inference rules that may contain applications of the current judgment or other judgments, using the variables that are declared as input parameters in the header or as locally declared variables in the body of the inference rules, as well as utilizing embedded Java type annotations, expressions, and commands. However, two features that have not yet been demonstrated are:

- Type judgments may also have output parameters whose values are set by the body of the inference rule; as an example, see Subsection C.1 where the rules for a judgment *te* $\vdash$ e`:exp(`t`)` first check whether in type environment *te* the expression e is well-typed and then set output parameter t to the type of e.

- Type judgments may annotate abstract syntax trees with information derived from the type checking; for this the corresponding option of the syntax domain has to be defined as `Tag[...,#C#]` where `C` is some (potentially user-defined) Java class. The parser constructing the corresponding syntax node has then to initialize the corresponding object of this class; the rule for type-checking this node may then update the content of this object.

If a program is not well-typed, the type checker raises a runtime exception and (unless this exception is caught and handled) aborts the processing of the program. If the program, however, is well typed, we are ready to execute it on the basis of its formal semantics, as described in the following subsections.

2.6 Interpreter from Denotational Semantics

In this subsection, we formalize the semantics of `Imp` in the style of a *denotational semantics* as it was introduced by Scott and Strachey [12] and is described in numerous textbooks [9, 10, 16, 5], i.e., by "semantic functions" that map syntactic domains to semantic domains. These semantic domains are typically (not always) function domains, i.e., the semantics of a syntactic phrase is typically a function on semantic domains. In SLANG, a semantic function F is thus usually applied as $v = F[\![p]\!](v_1, \dots, v_n)$ where p is the syntactic phrase to which F is applied and $v_1, \dots, v_n$ are the semantic values to which $F[\![p]\!]$ is applied; this application yields a semantic value v that denotes the semantics of p. A semantic function may also have multiple result values; in this case it is invoked as $r_1, \dots, r_m = F[\![p]\!](v_1, \dots, v_n)$.

The names of semantic functions may be overloaded with respect to the domain of their syntactic arguments. It is therefore often convenient to define semantic functions with *empty* names ("anonymous" functions) that may be applied as $v = [\![p]\!](v_1, \dots, v_n)$ (read: v is "the semantics of p in the context of $v_1, \dots, v_n$"). Since the function is anonymous, here the syntactic domain of p alone determines which function definition is to be looked up.

A semantic function is in general defined by multiple equations, one equation for each construction option of the syntactic domain on which it is defined. For instance, the following clause defines the semantics of integer expressions by an anonymous function from integer environments (“stores”) to integer values:

```
// -------------------------------------------------------------------
// denotational semantics
// -------------------------------------------------------------------
function ⟦Exp⟧: #Env<Integer># → #Integer#
{
  equation ⟦Num[n]⟧(ve) = v
  {
    v = #Integer.valueOf(n)#;
  }
  equation ⟦Id[i]⟧(ve) = v
  {
    v = #ve.get(i)#;
  }
  equation ⟦Plus[e1,e2]⟧(ve) = v
  {
    v1 = ⟦e1⟧(ve);
    v2 = ⟦e2⟧(ve);
    v = #v1+v2#;
  }
  equation ⟦Times[e1,e2]⟧(ve) = v
  {
    v1 = ⟦e1⟧(ve);
    v2 = ⟦e2⟧(ve);
    v = #v1*v2#;
  }
}
```

Thus the semantics of an integer numeral n evaluated in input store ve is (ignoring ve) the integer value v resulting from the decimal interpretation of n. The semantics of an identifier i evaluated in store ve is the integer value v to which ve maps v (if the program is well-typed according to the typing rules of the previous session, ve indeed maps i to some value). The semantics of a sum or product of two integer expressions e_1 and e_2 is the sum or product of the integer values v_1 and v_2 denoted by e_1 and e_2, respectively.

As above example demonstrates, each equation defines the semantic value v of the semantics of some kind of syntactic phrase applied to semantic arguments. The definition may consist of multiple equational definitions each of which introduces on the left side some auxiliary variable (v_1 and v_2 in above example) or defines the value of the final result v. The right side may be either the application of a semantic function or some embedded Java expression. The type of each variable is determined either from the point of its introduction (the semantic input parameter ve has type `Env<Integer>`, the semantic output parameter v has type `Integer`) or from the application of the semantic function on the right side (v_1 and v_2 have both type `Integer`). If an auxiliary variable x is introduced without a defining value or by a defining Java expression e, it must be explicitly given an embedded Java type t:

```
x: #t#;
x: #t# = #e#;
```

The semantics of Boolean expressions can be defined analogously:

```
function ⟦BoolExp⟧: #Env<Integer># → #Boolean#
{
  equation ⟦Eq[e1,e2]⟧(ve) = v
  {
    v1 = ⟦e1⟧(ve);
    v2 = ⟦e2⟧(ve);
    v = # v1.equals(v2) #;
  }
  equation ⟦Not[b]⟧(ve) = v
  {
    v0 = ⟦b⟧(ve);
    v = # !v0 #;
  }
  equation ⟦And[b1,b2]⟧(ve) = v
  {
    v1 = ⟦b1⟧(ve);
    v2 = ⟦b2⟧(ve);
    v = # v1 && v2 #;
  }
}
```

Thus the semantics of the equality of two integer expressions e_1 and e_2 is “true” if and only if both expressions yield equal integer values v_1 and v_2, respectively. The negation of a Boolean expression b yields the result of the semantic negation of the truth value v_0 denoted by b. The conjunction of two Boolean expressions b_1 and b_2 yields the result of the semantic conjunction of the truth values v_1 and v_2 denoted by b_1 and b_2, respectively.

The semantics of commands is a function on integer environments (i.e., a mapping from “input stores” to “output stores”):

```
function ⟦Command⟧: #Env<Integer># → #Env<Integer>#
{
  equation ⟦Assign[i,e]⟧(ve) = ve0
  {
    v = ⟦e⟧(ve);
    ve0 = #new Env<Integer>(ve)#; # ve0.put(i,v); #
  }
  after  {# System.out.println("Assignment " + i + " = " + ve0.get(i)); #}
```

```
    equation ⟦Var[i,c]⟧(ve) = ve0
    {
      ve1: #Env<Integer># = # new Env<Integer>(ve) #;
      v: #Integer# = # ve1.put(i,0) #;
      ve2 = ⟦c⟧(ve1);
      ve0 = # new Env<Integer>(ve2) #; # ve0.put(i,v); #
    }
    equation ⟦Seq[c1,c2]⟧(ve) = ve0
    {
      ve1 = ⟦c1⟧(ve);
      ve0 = ⟦c2⟧(ve1);
    }
    equation ⟦If2[b,c1,c2]⟧(ve) = ve0
    {
      v = ⟦b⟧(ve);
      # if (v) #
        ve0 = ⟦c1⟧(ve);
      # else #
        ve0 = ⟦c2⟧(ve);
    }
    equation ⟦If1[b,c]⟧(ve) = ve0
    {
      v = ⟦b⟧(ve);
      ve0 = ve ;
      # if (v) # ve0 = ⟦c⟧(ve);
    }
    equation ⟦While[b,c]⟧(ve) = ve0
    {
      ve0 = ve;
      # while (true) { #
        v = ⟦b⟧(ve0);
        # if (!v) break; #
        ve0 = ⟦c⟧(ve0);
      # } #
    }
  }
```

Here the equation for each kind of command must define from the "input store" *ve* the "output store" ve_0. If the command is an assignment of an expression *e* to a variable with name *i*, then *e* is evaluated in *ve*, which yields the integer value *v*, and ve_0 is a copy of *ve* that is updated by mapping *i* to *v*.

The equation for assignments is extended by an `after` clause with embedded Java code. This code will be executed in the generated interpreter after every evaluation of the equation defining the semantics of a assignment; it may be used to trace the execution of the program by external side effects, in this case by exhibiting every variable update. Likewise, the code embedded in a corresponding `before` clause will be executed before the evaluation of the defining equation. Therefore the code in a `before` clause may only refer to the input variables of the equation while the code in an `after` clause may also refer to the output variable and the auxiliary variables introduced in its definition.

The semantics of a block statement with the declaration of a variable named i and body command c first clones the input store ve to an intermediate store ve_1 and updates this store by mapping i to initial value 0 remembering in variable v the original value of i (if any). Then c is executed with respect to ve_1 resulting its output store ve_2. The output store ve_0 is then a clone of ve_2 where, however, i is reset to its original value v (if any).

The semantics of a sequence of two commands c_1 and c_2 is determined by first executing c_1 in input store ve; this yields an intermediate store ve_1 in which c_2 is executed which yields the overall output store ve_0.

The semantics of the two-sided or of the one-sided conditional command is defined by making use of an embedded Java conditional: if the semantics v of the boolean expression b yields "true", the first branch is executed in input store ve, which yields the output store ve_0; otherwise, the second branch is executed to determine the output store or (in the case of the one-sided conditional the output store is identical to the input store).

Similarly the semantics of the while loop command is defined by making use of the corresponding Java command[1]. We start by setting the output store ve_0 to the input store ve and repeatedly evaluate the boolean expression b with respect to ve_0. As long as this yields "true", we execute c in ve_0 and update ve_0 to the result of the execution. When the evaluation yields "false", ve_0 represents the overall result store.

Finally we can define the semantics of a program as follows:

```
function ⟦Program⟧: #Void#
{
  equation ⟦Prog[c]⟧ = none
  {
    ve0: #Env<Integer># = # new Env<Integer>() #;
    ve1 = ⟦c⟧(ve0);
    none = # null # ;
  }
}
```

The semantic function for programs creates an empty input store ve_0 and executes in that store the program command c, which yields the output store ve_1. The semantics function could now return this output store, our definition, however just returns a dummy value. The execution of the program semantics, however, will from the `after` clause for the assignment command print all variable updates to the standard output stream, which is the effect we desire for our programming language.

[1] It is also possible to define the loop semantics by recursive application of the semantic function being defined, which would be more in line with the usual formalization of loop semantics; Subsection 2.7 will demonstrate how such recursive applications become possible.

2.7 Interpreter from Big-Step Operational Semantics

In this subsection, we present an alternative formalization of the semantics of `Imp` as a *big-step operational semantics* (also called *natural semantics*) which was introduced by Kahn [4] and is also described in, e.g., [16, 5]. Here the semantics is described by a logical inference system whose judgments describe the transition of a program configuration (consisting of a syntactic phrase and the context in which the phrase is evaluated) to a result value. Big-step operational semantics is in this sense similar to denotational semantics but conceptually more general, because the judgments denote relations rather than functions; thus from a given configuration multiple outcomes are possible, i.e., program execution may be non-deterministic (however, SLANG implements a single deterministic execution mechanism that is applied to both kinds of semantic specifications).

A transition relation named R has generally the form $\langle p, v_1, \ldots, v_n\rangle \rightarrow R\ v$ where p is a syntactic phrase and $v_1, \ldots, v_n, v$ are semantic values (the angle brackets $\langle\ \rangle$ are optional); it can be read as "phrase p can in context $v_1, \ldots, v_n$ be reduced to value v". A transition relation may also yield multiple result values; in this case it is invoked as $\langle p, v_1, \ldots, v_n\rangle \rightarrow R\ \langle r_1, \ldots, r_m\rangle$ (again the angle brackets $\langle\ \rangle$ are optional). Analogously to the semantic functions, the name of a transition relations can be overloaded on the domain of its syntactic phrase; we can also define *anonymous* relations where R is the empty name and the appropriate definition of the relation is looked up from the domain of p only.

The following operational definition of `Imp` is stored in a separate specification file `Imp2.txt` that is given in Subsection C.3. This file is identical to `Imp.txt` except that the definitions of the semantic functions have been replaced by the definitions of transition relations[2].

Our big-step semantics depends on the following Java declarations:

```
// ----------------------------------------------------------------------
// big-step operational semantics ("natural semantics")
// ----------------------------------------------------------------------
code
{#
  public static class Failure extends RuntimeException { }
  public static void check(boolean b) { if (!b) throw new Failure(); }
#}
```

The method `check(`b`)` returns normally if the Boolean value b is true and raises a runtime exception otherwise. This enables to define multiple variants of an inference rule for a transition; if one fails by throwing a runtime exception, the next one will be selected (see below).

The big-step semantics of integer expressions can then be defined as follows:

```
transition ⟨Exp,#Env<Integer>#⟩ → #Integer#
{
```

[2]SLANG considers type checker judgments, semantic functions, and transition relations simply as different forms of *operations* whose fundamental behavior is identical and that are stored in the same namespace; thus we cannot have simultaneously in a single specification file an anonymous semantic function with application $v' = [\![p]\!](v)$ and an anonymous transition relation with application $\langle p, v\rangle \rightarrow v'$.

```
  step ⟨Num[n],ve⟩ → v
  {
    v = #Integer.valueOf(n)#;
  }
  step ⟨Id[i],ve⟩ → v
  {
    v = #ve.get(i)#;
  }
  step ⟨Plus[e1,e2],ve⟩ → v
  {
    ⟨e1,ve⟩ → v1;
    ⟨e2,ve⟩ → v2;
    v = #v1+v2#;
  }
  step ⟨Times[e1,e2],ve⟩ → v
  {
    ⟨e1,ve⟩ → v1;
    ⟨e2,ve⟩ → v2;
    v = #v1*v2#;
  }
}
```

This definition of the transition relation consists of multiple inference rules that describe the reduction of a configuration with an integer expression to an integer value. It matches line for line the corresponding definition of a denotational semantics by equations given in Subsection 2.6; the core difference is just the syntactic replacement of equational definitions by applications of inference rules.

Likewise, the big-step operational semantics of Boolean expressions is just a translation of the corresponding denotational one:

```
transition ⟨BoolExp,#Env<Integer>#⟩ → #Boolean#
{
  step ⟨Eq[e1,e2],ve⟩ → v
  {
    ⟨e1,ve⟩ → v1;
    ⟨e2,ve⟩ → v2;
    v = # v1.equals(v2) #;
  }
  step ⟨Not[b],ve⟩ → v
  {
    ⟨b,ve⟩ → v0;
    v = # !v0 #;
  }
  step ⟨And[b1,b2],ve⟩ → v
```

```
    {
      ⟨b1,ve⟩ → v1;
      ⟨b2,ve⟩ → v2;
      v = # v1 && v2 #;
    }
  }
```

However, more fundamental differences arise in the definition of the big-step operational semantics of some commands:

```
  transition ⟨Command,#Env<Integer>#⟩ → #Env<Integer>#
  {
    step ⟨Assign[i,e],ve⟩ -> ve0
    {
      ⟨e,ve⟩ → v;
      ve0 = #new Env<Integer>(ve)#; # ve0.put(i,v); #
    }
    after  {# System.out.println("Assignment " + i + " = " + ve0.get(i)); #}
    step ⟨Var[i,c],ve⟩ → ve0
    {
      ve1: #Env<Integer># = # new Env<Integer>(ve) #;
      v: #Integer# = # ve1.put(i,0) #;
      ⟨c,ve1⟩ → ve2;
      ve0 = # new Env<Integer>(ve2) #; # ve0.put(i,v); #
    }
    step ⟨Seq[c1,c2],ve⟩ → ve0
    {
      ⟨c1,ve⟩ → ve1;
      ⟨c2,ve1⟩ → ve0;
    }
    step ⟨If2[b,c1,c2],ve⟩ → ve0
    {
      ⟨b,ve⟩ → v; # check(v); #
      ⟨c1,ve⟩ → ve0;
    }
    or
    {
      ⟨b,ve⟩ → v; # check(!v); #
      ⟨c2,ve⟩ → ve0;
    }
    step ⟨If1[b,c],ve⟩ → ve0
    {
      ⟨b,ve⟩ → v; # check(v); #
      ⟨c,ve⟩ → ve0;
    }
    or
    {
      ⟨b,ve⟩ → v; # check(!v); #
      ve0 = ve;
    }
    step ⟨w=While[b,c],ve⟩ → ve0
    {
      ⟨b,ve⟩ → v; # check(!v); #
      ve0 = ve;
    }
    or
    {
      ⟨b,ve⟩ → v; # check(v); #
      ⟨c,ve⟩ → ve1;
      ⟨w,ve1⟩ → ve0;
    }
  }
```

While the definitions of the transition relations for assignment, variable declaration block, and command sequence still mimic the definitions of the corresponding semantic functions, the definitions for the two-sided and one-sided conditional and for the while loop differ in that we handle the two cases of the boolean expression b evaluated to truth value “true” and to truth value “false” by two separate inference rules separated by the keyword `or`. For the conditionals, the first rule checks whether the value is “true”; if this check fails, a runtime exception is thrown which signals to attempt the next rule (which for symmetry checks the dual case, even if this is guaranteed to succeed). The computation of the corresponding output stores thus becomes more transparent, since the various cases are syntactically separated.

As for the while loop command, the first rule checks whether the Boolean expression b evaluates to “false”; if this is the case, the output store ve_0 is identical to the input store ve; in the other case, we execute the body command c in ve which yields an intermediate store ve_1; what now remains to be done is to evaluate the while loop again in that state. For this purpose, the header of the rule includes the declaration w=`While[`b`,`c`]` which assigns name w to the loop; this name may be used to describe the remaining evaluation of the loop in intermediate store ve_1 to the overall output store ve_0.

This is an example of a case where the semantics of the currently considered phrase is itself utilized in its definition, i.e., the definition becomes (generally) *recursive*. In all other examples so far, semantic functions were only applied to structurally simpler phrases, i.e., recursive definitions were restricted to the pattern of *structural induction*, which guarantees the termination of the direct evaluation of these definitions. SLANG does not allow to construct new syntactic phrases in semantic definitions, however, by above naming it allows to not only base the semantics not only on the meanings of subphrases but also on the meaning of the whole phrase. We thus might also define the denotational semantics of while commands given in Subsection 2.6 in a recursive style (by translating above definition of a transition relation into a corresponding definition of an semantic function; we may also have in such function definitions multiple cases separated by `or` or use embedded Java conditionals to express the case distinction).

Finally, the big-step operational semantics of a program again mimics the previously given denotational semantics:

```
transition ⟨Program⟩ ⟶ #Void#
{
  step ⟨Prog[c]⟩ ⟶ none
  {
    ve0: #Env<Integer># = # new Env<Integer>() #;
    ⟨c,ve0⟩ ⟶ ve1;
    none = # null # ;
  }
}
```

Again, from the `after` clause of the assignment command, the execution of this program semantics traces all variable updates that occur in the program.

2.8 Code Generation, Compilation, and Execution

In this section, we are going to explain the steps that are needed to generate from the SLANG specification file `Imp.txt` an executable implementation of the language `Imp`. We assume that the `SLANG` software has been appropriately installed as described in Subsection A.1.

The individual steps can be best explained by investigating the shell script `ImpMake` included in the software distribution; by running this script we generate an executable Java program `ImpMain`:

```
#!/bin/sh

# JAVA
JAVA_HOME=/software/java21
JAVA_OPTIONS=
JAVA=$JAVA_HOME/bin/java
JAVAC=$JAVA_HOME/bin/javac

# ANTLR4: complete tool for compilation, not just the runtime binaries
ANTLR4_JAR=/software/SLANG/lib/antlr4.jar
ANTLR4="$JAVA -cp $ANTLR4_JAR org.antlr.v4.Tool"

# SLANG
SLANG=/software/SLANG/bin/SLANG

# execute SLANG, ANTLR4, JAVAC
$SLANG -d lang/imp Imp.txt
$ANTLR4 lang/imp/Imp.g4
$JAVAC $JAVA_OPTIONS -cp "$ANTLR4_JAR:." ImpMain.java
```

After the configuration of the various paths and command, the main steps executed by the last three commands are:

- Run SLANG to produce ANTLR4 source code for the parser and Java source code for the syntactic domains, type checker, and interpreter.
- Run ANTLR4 to translate ANTLR4 source code into Java source code for the parser.
- Compile a user-defined Java program `ImpMain.java` (given below) that calls the parser, prints the abstract syntax tree, type-checks the tree, and interprets it.

Running the script shows the terminal output produced by SLANG (the execution of ANTLR4 and the Java compiler is silent):

```
> ./ImpMake
SLANG Semantics-Based Language Generator 1.0 (September 20, 2023)
(c) 2023 https://www.risc.jku.at/research/formal/software/SLANG
This is free software distributed under the terms of the GNU GPL.
Execute "SLANG -h" to see the available command line options.
-----------------------------------------------------------------
Reading file /usr2/software/SLANG-1.0/languages/Imp.txt.
Generating files in directory /usr2/software/SLANG-1.0/languages/lang/imp.
Generating code class file Imp.java.
Generating ANTLR4 grammar Imp.g4.
Generating parser class file Imp_parser.java.
Generating for domain Exp interface file Exp.java.
Generating for domain BoolExp interface file BoolExp.java.
Generating for domain Command interface file Command.java.
Generating for domain Program interface file Program.java.
Generating for operation exp[Exp] class file Exp_exp.java.
Generating for operation bexp[BoolExp] class file BoolExp_bexp.java.
Generating for operation command[Command] class file Command_command.java.
Generating for operation program[Program] class file Program_program.java.
Generating for operation ␣[Exp] class file Exp_.java.
Generating for operation ␣[BoolExp] class file BoolExp_.java.
Generating for operation ␣[Command] class file Command_.java.
Generating for operation ␣[Program] class file Program_.java.
SUCCESS: execution successfully completed.
```

We see that a Java file `Imp.java` is generated that contains definitions used in the other Java files. Also an ANTLR4 file `Imp.g4` is generated that contains the ANTLR4 source code of the parser from which the execution of ANTLR4 generates the basic Java source code of the parser. SLANG generates in `Imp_parser.java` a high-level interface to the parser that can be later utilized in the main program `ImpMain`. Then SLANG generates one Java interface file for every syntactic domain and one Java class file for every operation specified in the SLANG file, i.e., for every type system judgment, every semantic function, and every transition relation (if such operations exist).

The content of the sample file `ImpMain.java` included in the distribution now utilizes the generated definitions as follows in order to construct an executable implementation of `Imp`:

```
import lang.imp.*;

public class ImpMain
{
  public static void main(String[] args)
  {
    try
    {
      // parsing
      Program program = Imp_parser.parseProgram();
      // pretty-printing
      System.out.println(program);
      // type-checking: inferring judgment ⊢ program:program()
      Program_program.operation(program).apply();
      // executing: evaluating anonymous function ⟦program⟧()
      Program_.operation(program).apply();
    }
    catch(Exception e)
    {
      // type-checking errors are caught here
      System.out.println(e.getMessage());
    }
  }
}
```

This Java file imports the content of package `lang.imp` declared in the `header` section of the SLANG specification file `Imp.txt`. As explained above, we get from this package (among others) the following entities:

- A type (interface) `Program` that represents the domain of abstract syntax trees of `Imp` programs (such interfaces exist for every syntactic domain); each such interface contains a method `toString()` that allows to print the text representation (concrete syntax) of the abstract syntax tree.
- A method `Imp_parser.parseProgram()` that parses the standard input for such a program (such methods exist for every syntactic domain and for multiple input mediums: the standard input, a Java `String`, a Java `File`, and a Java `Reader`). If the parsing fails, a runtime exception is thrown.
- For the type system judgment `program` on syntactic domain `Program` a Java function `Program_program.operation()` that can be applied to the abstract syntax tree of the

program and returns an anonymous Java function that can be (via method `apply`) invoked on the semantic input values of the judgment and returns its semantic output value (in our case just a dummy value which is ignored).

- For the anonymous semantic function on syntactic domain `Program` a Java function `Program_.operation()` that can be applied to the abstract syntax tree of the program and returns an anonymous Java function that can be (via method `apply`) invoked on the semantic input values of the function and returns its semantic output value (in our case just a dummy value which is ignored).

If an operation *op* on a syntactic domain *D* returns multiple values, the class *D_op* contains a class `Result` that encapsulates these values.

The compiled program `Imp` can be executed as shown in the shell script `Imp` included in the distribution:

```
#!/bin/sh
JAVA_HOME=/software/java21
JAVA_OPTIONS=
JAVA=$JAVA_HOME/bin/java
$JAVA $JAVA_OPTIONS -cp ".:/software/SLANG/lib/*" ImpMain $*
```

A sample execution on the `Imp` program given at the beginning of this section is shown below:

```
> ./Imp
var n; var a; var i;
n:=5; while ~(i = n) do { var j; j := 2*i+1; a:=a+j; i:=i+1 }
{var n; {var a; {var i; {n := 5; while (~i = n) do
  {var j; {{j := ((2*i)+1); a := (a+j)}; i := (i+1)}}}}}}
Assignment n = 5
Assignment j = 1
Assignment a = 1
Assignment i = 1
Assignment j = 3
Assignment a = 4
Assignment i = 2
Assignment j = 5
Assignment a = 9
Assignment i = 3
Assignment j = 7
Assignment a = 16
Assignment i = 4
Assignment j = 9
Assignment a = 25
Assignment i = 5
```

As we see, the program is parsed correctly (the output shows by additional parentheses the unique parsing structure), can be type-checked without error, and indeed computes in an iterative way for input $n = 5$ the output $a = n^2 = 25$.

3 Future Work

The SLANG language is quite minimalist and would profit from some extensions. In particular, SLANG specifications are monolithic and must be provided in single source files; in the future we may investigate the modularization of the language. Also, the only targets supported at the moment are Java and ANTLR4; it should be, however, quite easy to also support other programming languages and parser generators.

Our focus in the immediate future, however, will be the formalization of concrete languages with SLANG, in particular some domain-specific languages such as the robot control language described in [14]. The further evolution of SLANG will depend on the experience gained with its practical use.

References

[1] ANTLR, May 2023. `https://www.antlr.org`.

[2] Fabricio Chalub and Christiano Braga. Maude MSOS Tool. *Electronic Notes in Theoretical Computer Science*, 176(4):133–146, 2007. doi:10.1016/j.entcs.2007.06.012.

[3] Martin Fowler and Rebecca Parsons. *Domain-Specific Languages*. Professional Computing Series. Addison-Wesley, 2010. `https://martinfowler.com/books/dsl.html`.

[4] Gilles Kahn. Natural Semantics. In Franz J. Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS 87: 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19—21, 1987*, volume 1987 of *Lecture Notes in Computer Science*, pages 22–39. Springer, Berlin, Germany, 1987. doi:10.1007/BFb0039592.

[5] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. Undergraduate Topics in Computer Science. Springer, London, UK, 2007. doi:10.1007/978-1-84628-692-6.

[6] Terence Parr. *Language Implementation Patterns — Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 2009. `https://pragprog.com/titles/tpdsl/language-implementation-patterns`.

[7] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, January 2013. `https://pragprog.com/titles/tpantlr2/the-definitive-antlr-4-reference`.

[8] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010. doi:10.1016/j.jlap.2010.03.012.

[9] David A. Schmidt. *Denotational Semantics — A Methodology for Language Development*. Allyn and Bacon, Boston, MA, USA, 1986. http://people.cis.ksu.edu/~schmidt/text/densem.html.

[10] David A. Schmidt. *The Structure of Typed Programming Languages*. MIT Press, Cambridge, MA, USA, 1994. https://mitpress.mit.edu/books/structure-typed-programming-languages.

[11] Wolfgang Schreiner. *Thinking Programs — Logical Modeling and Reasoning about Languages, Data, Computations, and Executions*. Texts & Monographs in Symbolic Computation. Springer, Cham, Switzerland, 2021. doi:10.1007/978-3-030-80507-4.

[12] Dana Scott and Christopher Strachey. Towards a Mathematical Semantics for Computer Languages. In J. Fox, editor, *Proceedings of the Symposium on Computers and Automata*, volume 21 of *Microwave Research Institute Symposia Series*, pages 19–46, Polytechnic Institute of Brooklyn Press, New York, NY, USA, 1971. Also: Technical Monograph PRG-6, Oxford University Computing Laboratory, Programming Research Group, Oxford, UK, https://www.researchgate.net/publication/237107559_Towards_a_Mathematical_Semantics_for_Computer_Languages.

[13] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1):71–122, 2010. doi:10.1017/S0956796809990293.

[14] William Steingartner, Davorka Radaković, and Richard Zsiga. Some Aspects about Visualization of Natural Semantics for a Selected Domain-Specific Language. *IPSI Transactions on Internet Research*, 19(1):46–54, 2023. doi:10.58245/ipsi.tir.2301.08.

[15] Markus Voelter. *DSL Engineering — Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. https://voelter.de/data/books/markusvoelter-dslengineering-1.0.pdf.

[16] Glynn Winskel. *The Formal Semantics of Programming Languages — An Introduction*. MIT Press, Cambridge, MA, USA, 1994. https://mitpress.mit.edu/books/formal-semantics-programming-languages.

A The SLANG Software

In the following sections, we describe the software that implements the SLANG language.

A.1 Installing the Software

The README file of the installation is included below.

```
-----------------------------------------------------------------------------
README
Information on SLANG.

(c) 2023 https://www.risc.jku.at/research/formal/software/SLANG
Wolfgang.Schreiner <Wolfgang.Schreiner@risc.jku.at>
William Steingartner <william.steingartner@tuke.sk>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <https://www.gnu.org/licenses/>.
-----------------------------------------------------------------------------

The SLANG Semantics-Based Language Generator
===========================================
https://www.risc.jku.at/research/formal/software/SLANG

SLANG is a software for generating rapid prototype implementations of
programming languages from their formal specifications. Its input is a text file
that describes the abstract syntax of a language and its concrete text
representation; from this, a parser is generated (utilizing the ANTLR4 tool)
that transforms the text representation of a program into its abstract syntax
tree and a printer that generates from the abstract syntax tree its text
representation. Furthermore, one can equip the language with a formal type
system (by logical inference rules) from which a type checker is generated.
Finally, one can give the language a formal semantics, in the denotational style
(by function equations) and/or in the big-step operational style (by transition
steps); from this, a language interpreter is generated. SLANG is implemented in
Java and produces Java source code; it should be easy to extend the software
also to other target languages.

The Distribution
================
The distribution has the following contents:

  README      ... this file
  COPYING     ... the GNU General Public Licence Version 3
  CHANGES     ... the version history of the software
  bin/
    SLANG     ... the execution script
  lib/
    antl4.jar ... the ANTLR4 library (complete version)
    SLANG.jar ... the SLANG library
  doc/
    main.pdf  ... the manual
  languages/
    <L>.txt       ... a sample language definition of language <L>
    <L>Main.java  ... the main program for executing the language
    <L>Make       ... a script to generate the executable program
    <L>           ... a script to run the executable program
    lang/*/*      ... the *.java and *.g4 files generated by SLANG
  src/
    slang/*.java  ... the source code of SLANG

Installation
============
First make sure that you have installed the Java Development Kit Version 21
or newer (see below).

Then copy file bin/SLANG to a directory in your PATH and adapt in this file the
variable JAVA to point to the Java executable "java". Adapt SLANG to point to
the directory "lib" of the SLANG distribution.

You should then be able to execute

  SLANG -h

See the examples in directory "languages" (start with the scripts "<L>Make")
on how to generate an executable version from a SLANG language definition.

Third Party Software That You Have to Install
=============================================
SLANG assumes that the following third party software is installed on your
computer (if it is not already provided by your GNU/Linux distribution, you have
to downlad and install it manually).

Java Development Kit 21 or newer (Oracle JDK 21 recommended)
https://www.oracle.com/java/technologies/downloads/
------------------------------------------------------------------
Go to the "Downloads" section to download the JDK.

An installation of JDK 21 or newer is required, Oracle JDK 21 is recommended.
Older versions of Java will not work (SLANG utilizes the "record patterns"
and "pattern matching for switch" introduced in JDK 21).

On a Debian 13 "trixie" GNU/Linux distribution, just install the package
"openjdk-21-jdk" by executing (as superuser) the command

  apt-get install openjdk-21-jdk
```

```
On a Debian 12 "bookworm" GNU/Linux distribution, this package may be available
as a backport via https://backports.debian.org.

Third Party Software That Comes with SLANG
==========================================
SLANG also uses the the following open source software developed by third
parties. This software is already included in the distribution, but if you want
or need, you can download the source code from the denoted locations and compile
it on your own. Many thanks to the respective developers for making this great
software freely available!

ANTLR 4.13.0
https://www.antlr.org
--------------------
This is a framework for constructing parsers and lexical analyzers used for
processing the SLANG language and the ANTLR4 files generated by SLANG.

On a Debian 12 "bookworm" GNU/Linux distribution, just install the package
"antlr4" by executing (as superuser) the command

  apt-get install antlr4

-------------------------------------------------------------------------------
End of README.
-------------------------------------------------------------------------------
```

A.2 Running the Software

The SLANG software is executed by the shell script

```
SLANG
```

which prints out the copyright message

```
SLANG Semantics-Based Language Generator 1.0 (September 20, 2023)
(c) 2023 https://www.risc.jku.at/research/formal/software/SLANG
This is free software distributed under the terms of the GNU GPL.
Execute "SLANG -h" to see the available command line options.
-----------------------------------------------------------------
Reading standard input.
```

If we then press <CTRL>-D to close the standard input stream, this terminates the program. However, if we execute (as indicated in above message)

```
SLANG -h
```

we get the following output:

```
SLANG [ <options> ] [ <path> ]
<path>: path of language file (if none, read from stdin)
<options>: the following command line options
```

```
-h: print this message and exit
-ast: print abstract syntax tree of language specification
-d <path>: path of directory in which to generate code
```

A typical execution is therefore

```
SLANG -d <dir> <language>
```

where `<language>` denotes the language specification file and `<dir>` denotes the directory in which the software shall generate the source code of the implementation of the language.

B The SLANG Language

The SLANG grammar (for both lexical analysis and syntax analysis) is formally defined below as an ANTLR4 grammar file. Please note that the order of options in a grammar rules determine precedences; options that come earlier have higher precedence than others.

```
// --------------------------------------------------------------------------
// SLANG.g4
// Semantics Language Generator ANTLR4 Grammar
// $Id: SLANG.g4,v 1.1 2023/08/02 09:01:07 schreine Exp $
//
// (c) 2023 https://www.risc.jku.at/research/formal/software/SLANG
// Wolfgang.Schreiner <Wolfgang.Schreiner@risc.jku.at>
// William Steingartner <william.steingartner@tuke.sk>
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program.  If not, see <http://www.gnu.org/licenses/>.
// --------------------------------------------------------------------------

grammar SLANG;

options
{
  language=Java;
}

@header
{
  package slang.parser;
}
```

```
// ----------------------------------------------------------------------------
// problems and declarations
// ----------------------------------------------------------------------------

// languages
language: 'language' id '{'( ';' )* ( clause ( ';' )* )* '}' EOF ;

// clauses
clause:
  'target' 'java' '{' 'header' literal '}'    #Target
| 'code' literal                              #Code
| 'domains' '{' ( domaindef )* '}'            #Domains
| 'printer' '{' ( printerdomain )* '}'        #Printer
| 'parser' 'antlr4' '{' ( parserdomain )* '}' #Parser
| ( 'judgment' | 'judgement' ) ( in=types )? ( '⊢' | '|-' )
  domain=id ':' fun=id ( '(' out=types ')' )? '{' ( inference )* '}'
  ( 'before' ( bin=ids )? ( '⊢' | '|-' ) bid=id ':'
     before=id ( '(' bout=ids ')' )? bliteral=literal )?
  ( 'after' ( ain=ids )? ( '⊢' | '|-' ) aid=id ':'
     after=id ( '(' aout=ids ')' )? aliteral=literal )?   #Judgment
| 'transition' ( '⟨' | '<<' )? domain=id ( ',' in=types )? ( '⟩' | '>>' )?
    ( '→' | '->' ) ( fun=id )?
    ( '⟨' | '<<' )? out=types ( '⟩' | '>>' )? '{' ( step )* '}'
  ( 'before' ( bin=ids )? ( '⊢' | '|-' ) bid=id ':'
     before=id ( '(' bout=ids ')' )? bliteral=literal )?
  ( 'after' ( ain=ids )? ( '⊢' | '|-' ) aid=id ':'
     after=id ( '(' aout=ids ')' )? aliteral=literal )?   #Transition
| 'function' ( fun=id )? ( '⟦' | '[[' ) domain=id ( '⟧' | ']]' )
  ':' ( in=types ( '→' | '->' ) )? out=types '{' ( equation )* '}'
  ( 'before' ( before=id )? ( '⟦' | '[[' ) bid=id ( '⟧' | ']]' )
  ( '(' bin=ids ')' )? bliteral=literal )?
  ( 'after' ( after=id )? ( '⟦' | '[[' ) aid=id ( '⟧' | ']]' )
  ( '(' ain=ids ')' )? '=' aout=ids aliteral=literal )?   #Function
;

// domains and their constructors
domaindef: id '=' domaincon ( '+' domaincon )* ';' ;
domaincon: id ( '[' ( domainname ( ',' domainname )* ) ']' )? ;
domainname:
  id      #DomainId
| literal #DomainLiteral
;

// printers
printerdomain: 'domain' id '{' ( printercase )* '}' ;
printercase: 'case' domainpattern ( '→' | '->' ) literal ';' ;

// parsers
parserdomain: 'domain' id ( 'prefix' literal )? '{' ( parsercase )* '}' ;
parsercase: 'case' literal ( '→' | '->' ) domainexp ';' ;

// inferences, steps, and equations
inference :
  'inference' ( in=ids )? ( '⊢' | '|-' ) domainpattern ':' id ( '(' out=ids ')' )?
  bodies ( 'before' before=literal )? ( 'after' after=literal )?
;
step :
  'step' ( '⟨' | '<<' )? domainpattern ( ',' in=ids )? ( '⟩' | '>>' )?
  ( '→' | '->' ) ( id )? ( '⟨' | '<<' )? out=ids ( '⟩' | '>>' )?
  bodies ( 'before' before=literal )? ( 'after' after=literal )?
;
equation :
  'equation' ( id )? ( '⟦' | '[[' ) domainpattern ( '⟧' | ']]' )
  ( '(' in=ids ')' )? '=' out=ids
  bodies ( 'before' before=literal )? ( 'after' after=literal )?
;

// bodies and commands
body: '{' ( command )* '}' ;
command :
  literal                                           #CommandLiteral
| ids ( '⊢' | '|-' ) id ':' id ( '(' ids ')' )? ';' #CommandInference
| ( '⟨' | '<<' )? id ( ',' ids )? ( '⟩' | '>>' )? ( '→' | '->' ) ( id )?
  ( '⟨' | '<<' )? ids ( '⟩' | '>>' )? ';'           #CommandStep
| variables ( '=' value )? ';'                      #Assignment
;

// auxiliaries
variable: id ( ':' type )?;
value:
  id                        #ValueId
| literal                   #ValueLiteral
| ( fun=id )? ( '⟦' | '[[' ) domain=id ( '⟧' | ']]' )
  ( '(' values ')' )?       #Application
;
domainpattern: ( res=id '=' )? con=id ( '[' ids ']' )? ;
domainexp:
  literal             #DomainExpLiteral
| id ( '[' ids ']' )? #DomainExpApplication
;
type: literal ;

// sequences
types: type ( ( '×' | '*' | ',' ) type )* ;
bodies: body ( 'or' body )* ;
values : value ( ',' value )* ;
variables: variable ( ',' variable )* ;
ids: id ( ',' id )* ;

// wrappers for lexer domains
id: ID ;
literal: LITA | LITB | LITC | LITD ;

// ---------------------------------------------------------------------------
// lexical rules
// ---------------------------------------------------------------------------
```

```
ID   : [a-zA-Z_][a-zA-Z_0-9]* ;

LITA : '{##' .*? '##}' ;
LITB : '##' .*? '##' ;
LITC : '{#' .*? '#}' ;
LITD : '#' .*? '#' ;

WHITESPACE  : [ \t\r\n\f]+ -> skip ;
LINECOMMENT : '//' .*? '\r'? ('\n' | EOF) -> skip ;
COMMENT     : '/*' .*? '*/' -> skip ;

// matches any other character
ERROR : . ;

// ---------------------------------------------------------------------------
// end of file
// ---------------------------------------------------------------------------
```

C Example Languages Generated with SLANG

In the following, we present some example languages that have been implemented in SLANG and are distributed with the software in subdirectory `languages` (the shell scripts for generating and executing the implementations are omitted). In all cases, the implementations consist of a parser, printer, type checker, and interpreter.

C.1 An Evaluator Language

Generating the Implementation

```
> ./EvaluatorMake
SLANG Semantics-Based Language Generator 1.0 (September 20, 2023)
(c) 2023 https://www.risc.jku.at/research/formal/software/SLANG
This is free software distributed under the terms of the GNU GPL.
Execute "SLANG -h" to see the available command line options.
-----------------------------------------------------------------
Reading file /usr2/software/SLANG-1.0/languages/Evaluator.txt.
Generating files in directory /usr2/software/SLANG-1.0/languages/lang/eval.
Generating code class file Evaluator.java.
Generating ANTLR4 grammar Evaluator.g4.
Generating parser class file Evaluator_parser.java.
Generating for domain Exp interface file Exp.java.
Generating for domain Stat interface file Stat.java.
Generating for domain StatSeq interface file StatSeq.java.
Generating for domain Session interface file Session.java.
Generating for operation exp[Exp] class file Exp_exp.java.
Generating for operation stat[Stat] class file Stat_stat.java.
Generating for operation statseq[StatSeq] class file StatSeq_statseq.java.
Generating for operation session[Session] class file Session_session.java.
Generating for operation ␣[Exp] class file Exp_.java.
Generating for operation ␣[Stat] class file Stat_.java.
Generating for operation ␣[StatSeq] class file StatSeq_.java.
Generating for operation s[StatSeq] class file StatSeq_s.java.
Generating for operation ␣[Session] class file Session_.java.
SUCCESS: execution successfully completed.
```

Executing the Implementation

```
> ./Evaluator
i = (1+2)*3; b = !(i == 6); c = b ? i*i+1 : 0;
i = ((1+2)*3);
b = !i == 6;
c = (b ? ((i*i)+1) : 0);
Before execution of statement i = ((1+2)*3);
Assignment i = ((1+2)*3)
Variable value 9
After execution of statement i = ((1+2)*3);
Before execution of statement b = !i == 6;
Assignment b = !i == 6
Variable value true
After execution of statement b = !i == 6;
Before execution of statement c = (b ? ((i*i)+1) : 0);
Assignment c = (b ? ((i*i)+1) : 0)
Variable value 82
After execution of statement c = (b ? ((i*i)+1) : 0);
i = 9
b = true
c = 82
```

Main Program

```
import lang.eval.*;

public class EvaluatorMain
{
  public static void main(String[] args)
  {
    try
    {
    // parsing
    Session session = Evaluator_parser.parseSession();
    // pretty-printing
    System.out.print(session);
    // type-checking: inferring judgment ⊢ session:session()
    Session_session.operation(session).apply();
    // executing: evaluating anonymous function ⟦session⟧()
    Evaluator.Env<Object> env = Session_.operation(session).apply();
    // printing the resulting variable environment
    for (var entry : env.entrySet())
      System.out.println(entry.getKey() + " = " + entry.getValue());
    }
    catch(Exception e)
    {
      // type-checking errors are caught here
      System.out.println(e.getMessage());
    }
  }
}
```

Language Specification

```
// ---------------------------------------------------------------------------
// SLANG: a semantics-based language generator
// (c) 2023 https://www.risc.jku.at/research/formal/software/SLANG
// Wolfgang.Schreiner <Wolfgang.Schreiner@risc.jku.at>
// William Steingartner <william.steingartner@tuke.sk>
// ---------------------------------------------------------------------------

// automatic generation of parser, printer, typechecker, semantic evaluator

// supports the following kinds of operations
// (which are overloaded on the syntactic domain of "phrase"):
// * judgments: inferences of form "in ⊢ phrase:name(out)"
// * transitions: steps of form "<phrase,in> →name out"
// * functions: equations of form "out = name⟦phrase⟧(in)"
// (for transitions and functions "name" is optional)

// internally all three kinds are treated alike
// (actually each operation can be invoked in each syntax)

// the language specification grammar is agnostic of target language
// code in the target language may be embedded as #...# or {#...#}
// or ##...## or {##...##} (which allows to embed the # character itself)

// Unicode symbols and their ASCII alternatives:
// "⊢"=>"|-"  "⟨"=>">>" "⟩"=>">>" "⟦"=>"[[" "⟧"=>"]]" "→"=>"->" "×"=>"*"

// ---------------------------------------------------------------------------
// a language for the evaluation of a sequence of assignments
// ---------------------------------------------------------------------------
language Evaluator
{
  target java
  {
    header
    {#
      package lang.eval;
      import java.util.*;
      import static lang.eval.Evaluator.*;
    #}
  }

  // -------------------------------------------------------------------------
  // syntactic domains
  // -------------------------------------------------------------------------
  domains
  {
    // builtin domains ID, NUM, STR (represented as strings)
    Exp = Id[ID] + Num[NUM] + Plus[Exp,Exp] + Times[Exp,Exp]
        + True + Not[Exp] + Equal[Exp,Exp] + If[Exp,Exp,Exp];
    Stat = Assign[ID,Exp];
    StatSeq = Empty + Seq[StatSeq,Stat];
    Session = Eval[StatSeq];
  }

  // --------------------------------------------------------------------
  // printer
  // --------------------------------------------------------------------
  printer
  {
    domain Exp
    {
      case Id[i] → # _result = i; #;
      case Num[n] → # _result = n; #;
      case Plus[e1,e2] → # _result = "(" + e1 + "+" + e2 + ")"; #;
      case Times[e1,e2] → # _result = "(" + e1 + "*" + e2 + ")"; #;
      case True → # _result = "true"; #;
      case Not[e] → # _result = "!" + e; #;
      case Equal[e1,e2] → # _result = e1 + " == " + e2; #;
      case If[e,e1,e2] → # _result = "(" + e + " ? " + e1 + " : " + e2 + ")"; #;
    }
    domain Stat
    {
      case Assign[i,e] → # _result = i + " = " + e + ";\n"; #;
    }
    domain StatSeq
    {
      case Empty → # _result = ""; #;
      case Seq[seq,stat] → # _result = "" + seq + stat; #;
    }
    domain Session
    {
      case Eval[seq] → # _result = seq.toString(); #;
    }
  }

  // --------------------------------------------------------------------
  // parser
  // --------------------------------------------------------------------
  code
  {#
    public static StatSeq statSeq(List<Stat> stats)
    {
      StatSeq session = new StatSeq.Empty();
      for (Stat stat : stats) session = new StatSeq.Seq(session, stat);
      return session;
    }
  #}

  parser antlr4
  {
    domain Exp
    {
      case # i=dID # → Id[i];
      case # n=dNUM # → Num[n];
      case # e1=dExp '*' e2=dExp # → Times[e1,e2]; // higher priority first
      case # e1=dExp '+' e2=dExp # → Plus[e1,e2];
      case # 'true' # → True;
      case # '!' e=dExp # → Not[e];
      case # e1=dExp '==' e2=dExp # → Equal[e1,e2];
      case # e=dExp '?' e1=dExp ':' e2=dExp # → If[e,e1,e2];
      case # '(' e=dExp ')' # → # $_result = $e._result; #; // parenthesing
    }
    domain Stat
    {
      case # i=dID '=' e=dExp ';' # → Assign[i,e];
    }
    domain StatSeq
    prefix
    {#
      locals [ List<Stat> stats = new ArrayList<Stat>() ]
    #}
    {
      case # ( s=dStat { $stats.add($s._result); } )* # →
           # $_result = statSeq($stats); #;
    }
    domain Session
    {
      case # s=dStatSeq # → Eval[s];
    }
  }

  // ------------------------------------------------------------------------
  // type system
  // ------------------------------------------------------------------------
  code
  {#
    public enum Type { Int, Bool }
    public static class Env<T> extends HashMap<String,T>
    {
      public Env() { super(); }
      public Env(Env<T> e) { super(e); }
      public T put(String key, T value)
      { return super.put(key.toString(), value); }
      public T get(String key)
      { return super.get(key.toString()); }
    }
    public static class TypeError extends RuntimeException
    { public TypeError(String msg) { super(msg); } }
    public static void check(boolean b, String msg)
    { if (!b) throw new TypeError(msg); }
    public static void checkType(Exp e, Type type1, Type type2)
    { if (type1 != type2) throw new TypeError("expression " + e +
        " has type " + type1 + " but must have type " + type2); }
  #}
```

```
// overloading of judgment names based on domain parameters is allowed
judgment #Env<Type># ⊢ Exp: exp(#Type#)
{
  inference te ⊢ Id[i]: exp(t)
  {
    t = #te.get(i)#; # check (t != null, "undeclared variable " + i); #
  }
  inference te ⊢ Num[n]: exp(t)
  {
    t = #Type.Int#;
  }
  inference te ⊢ Plus[e1,e2]: exp(t)
  {
    te ⊢ e1: exp(t1); # checkType(e1, t1, Type.Int); #
    te ⊢ e2: exp(t2); # checkType(e2, t2, Type.Int); #
    t = #Type.Int#;
  }
  inference te ⊢ Times[e1,e2]: exp(t)
  {
    te ⊢ e1: exp(t1); # checkType(e1, t1, Type.Int); #
    te ⊢ e2: exp(t2); # checkType(e2, t2, Type.Int); #
    t = #Type.Int#;
  }
  inference te ⊢ True: exp(t)
  {
    t = #Type.Bool#;
  }
  inference te ⊢ Not[e]: exp(t)
  {
    te ⊢ e: exp(t0); # checkType(e, t0, Type.Bool); #
    t = #Type.Bool#;
  }
  inference te ⊢ Equal[e1,e2]: exp(t)
  {
    te ⊢ e1: exp(t1); # checkType(e1, t1, Type.Int); #
    te ⊢ e2: exp(t2); # checkType(e2, t2, Type.Int); #
    t = #Type.Bool#;
  }
  inference te ⊢ If[e,e1,e2]: exp(t)
  {
    te ⊢ e:  exp(t0); # checkType(e, t0, Type.Bool); #
    te ⊢ e1: exp(t1); # checkType(e1, t1, Type.Int); #
    te ⊢ e2: exp(t2); # checkType(e2, t2, Type.Int); #
    t = #Type.Bool#;
  }
}

judgment #Env<Type># ⊢ Stat: stat(#Env<Type>#)
{
  inference te ⊢ Assign[i,e]: stat(te0)
  {
    te ⊢ e: exp(t);
    te0 = #new Env<Type>(te)#; #te0.put(i,t);#
  }
```

```
}

judgment #Env<Type># ⊢ StatSeq: statseq(#Env<Type>#)
{
  inference te ⊢ Empty: statseq(te0)
  {
    te0 = te;
  }
  inference te ⊢ Seq[seq,stat]: statseq(te0)
  {
    te ⊢ seq: statseq(te1);
    te1 ⊢ stat: stat(te0);
  }
}

judgment ⊢ Session: session(#Env<Type>#)
{
  inference ⊢ Eval[seq]: session(te)
  {
    te0: #Env<Type># = #new Env<Type>()#;
    te0 ⊢ seq: statseq(te);
  }
}

// -----------------------------------------------------------------------
// denotational semantics
// -----------------------------------------------------------------------
code
{#
  public static class Failure extends RuntimeException { }
  public static void check(boolean b) { if (!b) throw new Failure(); }
#}

// functions may be named ⟦⟧ or id⟦⟧
// overloading based on domain parameters is allowed
function ⟦Exp⟧: #Env<Object># → #Object#
{
  equation ⟦Id[i]⟧(ve) = v
  {
    v = #ve.get(i)#;
  }
  equation ⟦Num[n]⟧(ve) = v
  {
    v = #Integer.valueOf(n)#;
  }
  equation ⟦Plus[e1,e2]⟧(ve) = v
  {
    v1 = ⟦e1⟧(ve); # Integer i1 = (Integer)v1; #
    v2 = ⟦e2⟧(ve); # Integer i2 = (Integer)v2; #
    v = #i1+i2#;
  }
  equation ⟦Times[e1,e2]⟧(ve) = v
  {
    v1 = ⟦e1⟧(ve); # Integer i1 = (Integer)v1; #
    v2 = ⟦e2⟧(ve); # Integer i2 = (Integer)v2; #
    v = #i1*i2#;
  }
  equation ⟦True⟧(ve) = v
  {
    v = #true#;
  }
  equation ⟦Not[e]⟧(ve) = v
  {
    v0 = ⟦e⟧(ve); # Boolean b = (Boolean)v0; #
    v = #!b#;
  }
  equation ⟦Equal[e1,e2]⟧(ve) = v
  {
    v1 = ⟦e1⟧(ve);
    v2 = ⟦e2⟧(ve);
    v = #v1 == v2#;
  }

  // definition via handling of conditional in target code
  /*
  equation ⟦If[e,e1,e2]⟧(ve) = v
  {
    v0  = ⟦e⟧(ve); # Boolean b = (Boolean)v0; #
    # Object vr; #
    # if (b) { #
      v1 = ⟦e1⟧(ve); # vr = v1; #
    # } else { #
      v2 = ⟦e2⟧(ve); # vr = v2; #
    # } #
    v = #vr#;
  }
  */

  // definition with multiple cases:
  equation ⟦If[e,e1,e2]⟧(ve) = v
  {
     v0  = ⟦e⟧(ve);
     # check((Boolean)v0); # // case fails if condition is false
     v = ⟦e1⟧(ve);
   }
   or
   {
     v0  = ⟦e⟧(ve);
     # check(!(Boolean)v0); # // case fails if condition is true
     v = ⟦e2⟧(ve);
   }
}

function ⟦Stat⟧: #Env<Object># → #Env<Object>#
{
  equation ⟦Assign[i,e]⟧(ve) = ve0
  {
    v = ⟦e⟧(ve);
```

```
      ve0 = #new Env<Object>(ve)#; # ve0.put(i,v); #
    }
    // annotations for producing side effects for one equation/inference
    before {# System.out.println("Assignment " + i + " = " + e); #}
    after  {# System.out.println("Variable value " + ve0.get(i)); #}
  }
  // annotations for producing side effects for whole function/judgment
  before ⟦stat⟧(ve)
  {# System.out.print("Before execution of statement " + stat); #}
  after ⟦stat⟧(ve) = ve0
  {# System.out.print("After execution of statement " + stat); #}

  function ⟦StatSeq⟧: #Env<Object># → #Env<Object>#
  {
    equation ⟦Empty⟧(ve) = ve0
    {
      ve0 = ve;
    }
    equation ⟦Seq[seq,stat]⟧(ve) = ve0
    {
      ve1 = ⟦seq⟧(ve);
      ve0 = ⟦stat⟧(ve1);
    }
  }

  // alternative big-step operational semantics
  transition ⟨StatSeq,#Env<Object>#⟩ →s #Env<Object>#
  {
    step ⟨Empty,ve⟩ →s ve0
    {
      ve0 = ve;
    }
    // seq0 denotes whole phrase, useful in semantics of loops
    step ⟨seq0=Seq[seq,stat],ve⟩ →s ve0
    {
      ⟨seq,ve⟩ →s ve1;
      ⟨stat,ve1⟩ → ve0; // uses ve0 = ⟦stat⟧(ve)
    }
  }

  function ⟦Session⟧: #Env<Object>#
  {
    equation ⟦Eval[seq]⟧ = ve
    {
      ve0 : #Env<Object># = #new Env<Object>()#;
      ve = ⟦seq⟧(ve0);
    }
  }
}
// -------------------------------------------------------------------------
// end of file
// -------------------------------------------------------------------------
```

C.2 An Imperative Language (Denotational Semantics)

Generating the Implementation

```
> ./ImpMake
SLANG Semantics-Based Language Generator 1.0 (September 20, 2023)
(c) 2023 https://www.risc.jku.at/research/formal/software/SLANG
This is free software distributed under the terms of the GNU GPL.
Execute "SLANG -h" to see the available command line options.
-----------------------------------------------------------------
Reading file /usr2/software/SLANG-1.0/languages/Imp.txt.
Generating files in directory /usr2/software/SLANG-1.0/languages/lang/imp.
Generating code class file Imp.java.
Generating ANTLR4 grammar Imp.g4.
Generating parser class file Imp_parser.java.
Generating for domain Exp interface file Exp.java.
Generating for domain BoolExp interface file BoolExp.java.
Generating for domain Command interface file Command.java.
Generating for domain Program interface file Program.java.
Generating for operation exp[Exp] class file Exp_exp.java.
Generating for operation bexp[BoolExp] class file BoolExp_bexp.java.
Generating for operation command[Command] class file Command_command.java.
Generating for operation program[Program] class file Program_program.java.
Generating for operation ␣[Exp] class file Exp_.java.
Generating for operation ␣[BoolExp] class file BoolExp_.java.
Generating for operation ␣[Command] class file Command_.java.
Generating for operation ␣[Program] class file Program_.java.
SUCCESS: execution successfully completed.
```

Executing the Implementation

```
> ./Imp
var n; var a; var i; n:=5; while ~(i = n) do { var j; j := 2*i+1; a:=a+j; i:=i+1 }
{var n; {var a; {var i; {n := 5; while (~i = n) do {var j; {{j := ((2*i)+1); a := (a+j)}; i := (i+1)}}}}}}
Assignment n = 5
Assignment j = 1
Assignment a = 1
Assignment i = 1
Assignment j = 3
Assignment a = 4
Assignment i = 2
Assignment j = 5
Assignment a = 9
Assignment i = 3
Assignment j = 7
Assignment a = 16
Assignment i = 4
Assignment j = 9
Assignment a = 25
Assignment i = 5
```

Main Program

```
import lang.imp.*;
```

```
public class ImpMain
{
  public static void main(String[] args)
  {
    try
    {
      // parsing
      Program program = Imp_parser.parseProgram();
      // pretty-printing
      System.out.println(program);
      // type-checking: inferring judgment ⊢ program:program()
      Program_program.operation(program).apply();
      // executing: evaluating anonymous function ⟦program⟧()
      Program_.operation(program).apply();
    }
    catch(Exception e)
    {
      // type-checking errors are caught here
      System.out.println(e.getMessage());
    }
  }
}
```

Language Specification

```
// ----------------------------------------------------------------------------
// Imp.txt
// SLANG version of the *denotational* semantics of a simple imperative language
// (c) 2023 https://www.risc.jku.at/research/formal/software/SLANG
// Wolfgang.Schreiner <Wolfgang.Schreiner@risc.jku.at>
// William Steingartner <william.steingartner@tuke.sk>
// ----------------------------------------------------------------------------

language Imp
{
  target java
  {
    header
    {#
      package lang.imp;
      import java.util.*;
      import static lang.imp.Imp.*;
    #}
  }

  // --------------------------------------------------------------------------
  // syntactic domains
  // --------------------------------------------------------------------------
  domains
  {
    Exp = Num[NUM] + Id[ID] + Plus[Exp,Exp] + Times[Exp,Exp];
    BoolExp = Eq[Exp,Exp] + Not[BoolExp] + And[BoolExp,BoolExp];
    Command = Assign[ID,Exp] + Var[ID,Command] + Seq[Command,Command]
            + If2[BoolExp,Command,Command] + If1[BoolExp,Command]
            + While[BoolExp,Command];
    Program = Prog[Command];
  }

  // ----------------------------------------------------------------------
  // printer
  // ----------------------------------------------------------------------
  printer
  {
    domain Exp
    {
      case Num[n] → # _result = n; #;
      case Id[i] → # _result = i; #;
      case Plus[e1,e2] → # _result = "(" + e1 + "+" + e2 + ")"; #;
      case Times[e1,e2] → # _result = "(" + e1 + "*" + e2 + ")"; #;
    }
    domain BoolExp
    {
      case Eq[e1,e2] → # _result = e1 + " = " + e2; #;
      case Not[b] → # _result = "(~" + b + ")"; #;
      case And[b1,b2] → # _result = "(" + b1 + " /\\ " + b2 + ")"; #;
    }
    domain Command
    {
      case Assign[i,e] → # _result = i + " := " + e; #;
      case Var[i,c] → # _result = "{var " + i + "; " + c + "}"; #;
      case Seq[c1,c2] → # _result = "{" + c1 + "; " + c2 + "}"; #;
      case If2[b,c1,c2] → # _result = "if " + b + " then " + c1 + " else " + c2; #;
      case If1[b,c] → # _result = "if " + b + " then " + c; #;
      case While[b,c] → # _result = "while " + b + " do " + c; #;
    }
    domain Program
    {
      case Prog[c] → # _result = c.toString(); #;
    }
  }

  // ----------------------------------------------------------------------
  // parser
  // ----------------------------------------------------------------------
  parser antlr4
  {
    domain Exp
    {
      case # n=dNUM # → Num[n];
      case # i=dID # → Id[i];
      case # e1=dExp '*' e2=dExp # → Times[e1,e2]; // higher priority first
      case # e1=dExp '+' e2=dExp # → Plus[e1,e2];
      case # '(' e=dExp ')' # → # $_result = $e._result; #; // parenthesing
    }
    domain BoolExp
    {
      case # e1=dExp '=' e2=dExp # → Eq[e1,e2];
      case # '~' b=dBoolExp # → Not[b];
      case # b1=dBoolExp '/\\' b2=dBoolExp # → And[b1,b2];
      case # '(' b=dBoolExp ')' # → # $_result = $b._result; #; // parenthesing
    }
    domain Command
    {
      case # i=dID ':=' e=dExp # → Assign[i,e];
      case # 'if' b=dBoolExp 'then' c1=dCommand 'else' c2=dCommand # → If2[b,c1,c2];
      case # 'if' b=dBoolExp 'then' c=dCommand # → If1[b,c];
      case # 'while' b=dBoolExp 'do' c=dCommand # → While[b,c];
      case # c1=dCommand ';' c2=dCommand # → Seq[c1,c2]; // binds weaker than if/while
      case # 'var' i=dID ';' c=dCommand # → Var[i,c];    // binds weaker than sequence
      case # '{' c=dCommand '}' # → # $_result = $c._result; #; // parenthesing
    }
    domain Program
    {
      case # c=dCommand EOF # → Prog[c];
    }
  }

  // ----------------------------------------------------------------------
  // type system
  // ----------------------------------------------------------------------
  code
  {#
    public enum Type { Int }
    public static class Env<T> extends HashMap<String,T>
    {
      public Env() { super(); }
      public Env(Env<T> e) { super(e); }
      public T put(String key, T value)
      { return super.put(key.toString(), value); }
      public T get(String key)
      { return super.get(key.toString()); }
    }
    public static class TypeError extends RuntimeException
    { public TypeError(String msg) { super(msg); } }
    public static void check(boolean b, String msg)
    { if (!b) throw new TypeError(msg); }
  #}
  judgment #Env<Type># ⊢ Exp: exp
  {
    inference te ⊢ Id[i]: exp
    {
      # check(te.get(i) != null, "undeclared variable " + i); #
    }
    inference te ⊢ Num[n]: exp
    {
    }
    inference te ⊢ Plus[e1,e2]: exp
    {
      te ⊢ e1: exp;
      te ⊢ e2: exp;
    }
    inference te ⊢ Times[e1,e2]: exp
    {
      te ⊢ e1: exp;
      te ⊢ e2: exp;
    }
  }
  judgment #Env<Type># ⊢ BoolExp: bexp
  {
    inference te ⊢ Eq[e1,e2]: bexp
    {
      te ⊢ e1: exp;
      te ⊢ e2: exp;
    }
    inference te ⊢ Not[b]: bexp
    {
      te ⊢ b: bexp;
    }
    inference te ⊢ And[b1,b2]: bexp
    {
      te ⊢ b1: bexp;
      te ⊢ b2: bexp;
    }
  }
  judgment #Env<Type># ⊢ Command: command
  {
    inference te ⊢ Assign[i,e]: command
    {
      # check(te.get(i) != null, "undeclared variable " + i); #
      te ⊢ e: exp;
    }
    inference te ⊢ Var[i,c]: command
    {
      te0: #Env<Type># = # new Env<Type>(te) #; # te0.put(i, Type.Int); #
      te0 ⊢ c: command;
    }
    inference te ⊢ Seq[c1,c2]: command
    {
      te ⊢ c1: command;
      te ⊢ c2: command;
    }
    inference te ⊢ If2[b,c1,c2]: command
    {
      te ⊢ b: bexp;
      te ⊢ c1: command;
      te ⊢ c2: command;
    }
    inference te ⊢ If1[b,c]: command
    {
      te ⊢ b: bexp;
      te ⊢ c: command;
    }
    inference te ⊢ While[b,c]: command
    {
      te ⊢ b: bexp;
      te ⊢ c: command;
    }
  }
  judgment ⊢ Program: program
  {
    inference ⊢ Prog[c]: program
    {
      te: #Env<Type># = #new Env<Type>()# ;
      te ⊢ c: command;
    }
  }

  // -------------------------------------------------------------------
  // denotational semantics
  // -------------------------------------------------------------------
  code
  {#
    public static class Failure extends RuntimeException { }
    public static void check(boolean b) { if (!b) throw new Failure(); }
  #}
  function ⟦Exp⟧: #Env<Integer># → #Integer#
  {
    equation ⟦Num[n]⟧(ve) = v
    {
      v = #Integer.valueOf(n)#;
    }
    equation ⟦Id[i]⟧(ve) = v
    {
      v = #ve.get(i)#;
    }
    equation ⟦Plus[e1,e2]⟧(ve) = v
    {
      v1 = ⟦e1⟧(ve);
      v2 = ⟦e2⟧(ve);
      v = #v1+v2#;
    }
    equation ⟦Times[e1,e2]⟧(ve) = v
    {
      v1 = ⟦e1⟧(ve);
      v2 = ⟦e2⟧(ve);
      v = #v1*v2#;
    }
  }
  function ⟦BoolExp⟧: #Env<Integer># → #Boolean#
  {
    equation ⟦Eq[e1,e2]⟧(ve) = v
    {
      v1 = ⟦e1⟧(ve);
      v2 = ⟦e2⟧(ve);
      v = # v1.equals(v2) #;
    }
    equation ⟦Not[b]⟧(ve) = v
    {
      v0 = ⟦b⟧(ve);
      v = # !v0 #;
    }
    equation ⟦And[b1,b2]⟧(ve) = v
    {
      v1 = ⟦b1⟧(ve);
      v2 = ⟦b2⟧(ve);
      v = # v1 && v2 #;
    }
  }
  function ⟦Command⟧: #Env<Integer># → #Env<Integer>#
  {
    equation ⟦Assign[i,e]⟧(ve) = ve0
    {
      v = ⟦e⟧(ve);
      ve0 = #new Env<Integer>(ve)#; # ve0.put(i,v); #
    }
    after  {# System.out.println("Assignment " + i + " = " + ve0.get(i)); #}
    equation ⟦Var[i,c]⟧(ve) = ve0
    {
      ve1: #Env<Integer># = # new Env<Integer>(ve) #;
      v: #Integer# = # ve1.put(i,0) #;
      ve2 = ⟦c⟧(ve1);
      ve0 = # new Env<Integer>(ve2) #; # ve0.put(i,v); #
    }
    equation ⟦Seq[c1,c2]⟧(ve) = ve0
    {
      ve1 = ⟦c1⟧(ve);
      ve0 = ⟦c2⟧(ve1);
    }
    equation ⟦If2[b,c1,c2]⟧(ve) = ve0
    {
      v = ⟦b⟧(ve);
      # if (v) #
        ve0 = ⟦c1⟧(ve);
      # else #
        ve0 = ⟦c2⟧(ve);
    }
    equation ⟦If1[b,c]⟧(ve) = ve0
    {
      v = ⟦b⟧(ve);
      ve0 = ve ;
      # if (v) # ve0 = ⟦c⟧(ve);
    }
    equation ⟦While[b,c]⟧(ve) = ve0
    {
      ve0 = ve;
      # while (true) { #
        v = ⟦b⟧(ve0);
        # if (!v) break; #
        ve0 = ⟦c⟧(ve0);
      # } #
    }
  }
```

```
  function ⟦Program⟧: #Void#
  {
    equation ⟦Prog[c]⟧ = none
    {
      ve0: #Env<Integer># = # new Env<Integer>() #;
      ve1 = ⟦c⟧(ve0);
      none = # null # ;
    }
  }
}
// ------------------------------------------------------------------------
// end of file
// ------------------------------------------------------------------------
```

C.3 An Imperative Language (Big-Step Operational Semantics)

Generating the Implementation

```
> ./Imp2Make
SLANG Semantics-Based Language Generator 1.0 (September 20, 2023)
(c) 2023 https://www.risc.jku.at/research/formal/software/SLANG
This is free software distributed under the terms of the GNU GPL.
Execute "SLANG -h" to see the available command line options.
-----------------------------------------------------------------
Reading file /usr2/software/SLANG-1.0/languages/Imp2.txt.
Generating files in directory /usr2/software/SLANG-1.0/languages/lang/imp2.
Generating code class file Imp.java.
Generating ANTLR4 grammar Imp.g4.
Generating parser class file Imp_parser.java.
Generating for domain Exp interface file Exp.java.
Generating for domain BoolExp interface file BoolExp.java.
Generating for domain Command interface file Command.java.
Generating for domain Program interface file Program.java.
Generating for operation exp[Exp] class file Exp_exp.java.
Generating for operation bexp[BoolExp] class file BoolExp_bexp.java.
Generating for operation command[Command] class file Command_command.java.
Generating for operation program[Program] class file Program_program.java.
Generating for operation ␣[Exp] class file Exp_.java.
Generating for operation ␣[BoolExp] class file BoolExp_.java.
Generating for operation ␣[Command] class file Command_.java.
Generating for operation ␣[Program] class file Program_.java.
SUCCESS: execution successfully completed.
```

Executing the Implementation

```
> ./Imp2
var n; var a; var i; n:=5; while ~(i = n) do { var j; j := 2*i+1; a:=a+j; i:=i+1 }
{var n; {var a; {var i; {n := 5; while (~i = n) do {var j; {{j := ((2*i)+1); a := (a+j)}; i := (i+1)}}}}}}
Assignment n = 5
Assignment j = 1
Assignment a = 1
Assignment i = 1
Assignment j = 3
Assignment a = 4
Assignment i = 2
Assignment j = 5
Assignment a = 9
Assignment i = 3
Assignment j = 7
Assignment a = 16
Assignment i = 4
Assignment j = 9
Assignment a = 25
Assignment i = 5
```

Main Program

```
import lang.imp2.*;

public class Imp2Main
{
  public static void main(String[] args)
  {
    try
    {
      // parsing
      Program program = Imp_parser.parseProgram();
      // pretty-printing
      System.out.println(program);
      // type-checking: inferring judgment ⊢ program:program()
      Program_program.operation(program).apply();
      // executing: evaluating anonymous function ⟦program⟧()
      Program_.operation(program).apply();
    }
    catch(Exception e)
    {
      // type-checking errors are caught here
      System.out.println(e.getMessage());
    }
  }
}
```

Language Specification

```
// ------------------------------------------------------------------------
// Imp.txt
// SLANG version of the *operational* semantics of a simple imperative language
// (c) 2023 https://www.risc.jku.at/research/formal/software/SLANG
// Wolfgang.Schreiner <Wolfgang.Schreiner@risc.jku.at>
// William Steingartner <william.steingartner@tuke.sk>
// ------------------------------------------------------------------------

language Imp
{
  target java
  {
```

```
  header
  {#
    package lang.imp2;
    import java.util.*;
    import static lang.imp2.Imp.*;
  #}
}

// -----------------------------------------------------------------------
// syntactic domains
// -----------------------------------------------------------------------
domains
{
  Exp = Num[NUM] + Id[ID] + Plus[Exp,Exp] + Times[Exp,Exp];
  BoolExp = Eq[Exp,Exp] + Not[BoolExp] + And[BoolExp,BoolExp];
  Command = Assign[ID,Exp] + Var[ID,Command] + Seq[Command,Command]
          + If2[BoolExp,Command,Command] + If1[BoolExp,Command]
          + While[BoolExp,Command];
  Program = Prog[Command];
}

// ------------------------------------------------------------------------
// printer
// ------------------------------------------------------------------------
printer
{
  domain Exp
  {
    case Num[n] → # _result = n; #;
    case Id[i] → # _result = i; #;
    case Plus[e1,e2] → # _result = "(" + e1 + "+" + e2 + ")"; #;
    case Times[e1,e2] → # _result = "(" + e1 + "*" + e2 + ")"; #;
  }
  domain BoolExp
  {
    case Eq[e1,e2] → # _result = e1 + " = " + e2; #;
    case Not[b] → # _result = "(~" + b + ")"; #;
    case And[b1,b2] → # _result = "(" + b1 + " /\\ " + b2 + ")"; #;
  }
  domain Command
  {
    case Assign[i,e] → # _result = i + " := " + e; #;
    case Var[i,c] → # _result = "{var " + i + "; " + c + "}"; #;
    case Seq[c1,c2] → # _result = "{" + c1 + "; " + c2 + "}"; #;
    case If2[b,c1,c2] → # _result = "if " + b + " then " + c1 + " else " + c2; #;
    case If1[b,c] → # _result = "if " + b + " then " + c; #;
    case While[b,c] → # _result = "while " + b + " do " + c; #;
  }
  domain Program
  {
    case Prog[c] → # _result = c.toString(); #;
  }
}
```

```
// ---------------------------------------------------------------------
// parser
// ---------------------------------------------------------------------
parser antlr4
{
  domain Exp
  {
    case # n=dNUM # → Num[n];
    case # i=dID # → Id[i];
    case # e1=dExp '*' e2=dExp # → Times[e1,e2]; // higher priority first
    case # e1=dExp '+' e2=dExp # → Plus[e1,e2];
    case # '(' e=dExp ')' # → # $_result = $e._result; #; // parenthesing
  }
  domain BoolExp
  {
    case # e1=dExp '=' e2=dExp # → Eq[e1,e2];
    case # '~' b=dBoolExp # → Not[b];
    case # b1=dBoolExp '/\\' b2=dBoolExp # → And[b1,b2];
    case # '(' b=dBoolExp ')' # → # $_result = $b._result; #; // parenthesing
  }
  domain Command
  {
    case # i=dID ':=' e=dExp # → Assign[i,e];
    case # 'if' b=dBoolExp 'then' c1=dCommand 'else' c2=dCommand # → If2[b,c1,c2];
    case # 'if' b=dBoolExp 'then' c=dCommand # → If1[b,c];
    case # 'while' b=dBoolExp 'do' c=dCommand # → While[b,c];
    case # c1=dCommand ';' c2=dCommand # → Seq[c1,c2]; // binds weaker than if/while
    case # 'var' i=dID ';' c=dCommand # → Var[i,c];    // binds weaker than sequence
    case # '{' c=dCommand '}' # → # $_result = $c._result; #; // parenthesing
  }
  domain Program
  {
    case # c=dCommand EOF # → Prog[c];
  }
}

// ---------------------------------------------------------------------
// type system
// ---------------------------------------------------------------------
code
{#
  public enum Type { Int }
  public static class Env<T> extends HashMap<String,T>
  {
    public Env() { super(); }
    public Env(Env<T> e) { super(e); }
    public T put(String key, T value)
    { return super.put(key.toString(), value); }
    public T get(String key)
    { return super.get(key.toString()); }
  }
  public static class TypeError extends RuntimeException
  { public TypeError(String msg) { super(msg); } }
  public static void check(boolean b, String msg)
  { if (!b) throw new TypeError(msg); }
#}
judgment #Env<Type># ⊢ Exp: exp
{
  inference te ⊢ Id[i]: exp
  {
    # check(te.get(i) != null, "undeclared variable " + i); #
  }
  inference te ⊢ Num[n]: exp
  {
  }
  inference te ⊢ Plus[e1,e2]: exp
  {
    te ⊢ e1: exp;
    te ⊢ e2: exp;
  }
  inference te ⊢ Times[e1,e2]: exp
  {
    te ⊢ e1: exp;
    te ⊢ e2: exp;
  }
}
judgment #Env<Type># ⊢ BoolExp: bexp
{
  inference te ⊢ Eq[e1,e2]: bexp
  {
    te ⊢ e1: exp;
    te ⊢ e2: exp;
  }
  inference te ⊢ Not[b]: bexp
  {
    te ⊢ b: bexp;
  }
  inference te ⊢ And[b1,b2]: bexp
  {
    te ⊢ b1: bexp;
    te ⊢ b2: bexp;
  }
}
judgment #Env<Type># ⊢ Command: command
{
  inference te ⊢ Assign[i,e]: command
  {
    # check(te.get(i) != null, "undeclared variable " + i); #
    te ⊢ e: exp;
  }
  inference te ⊢ Var[i,c]: command
  {
    te0: #Env<Type># = # new Env<Type>(te) #; # te0.put(i, Type.Int); #
    te0 ⊢ c: command;
  }
  inference te ⊢ Seq[c1,c2]: command
  {
    te ⊢ c1: command;
    te ⊢ c2: command;
  }
  inference te ⊢ If2[b,c1,c2]: command
  {
    te ⊢ b: bexp;
    te ⊢ c1: command;
    te ⊢ c2: command;
  }
  inference te ⊢ If1[b,c]: command
  {
    te ⊢ b: bexp;
    te ⊢ c: command;
  }
  inference te ⊢ While[b,c]: command
  {
    te ⊢ b: bexp;
    te ⊢ c: command;
  }
}
judgment ⊢ Program: program
{
  inference ⊢ Prog[c]: program
  {
    te: #Env<Type># = #new Env<Type>()# ;
    te ⊢ c: command;
  }
}

// ----------------------------------------------------------------------
// big-step operational semantics ("natural semantics")
// ----------------------------------------------------------------------
code
{#
  public static class Failure extends RuntimeException { }
  public static void check(boolean b) { if (!b) throw new Failure(); }
#}
transition ⟨Exp,#Env<Integer>#⟩ → #Integer#
{
  step ⟨Num[n],ve⟩ → v
  {
    v = #Integer.valueOf(n)#;
  }
  step ⟨Id[i],ve⟩ → v
  {
    v = #ve.get(i)#;
  }
  step ⟨Plus[e1,e2],ve⟩ → v
  {
    ⟨e1,ve⟩ → v1;
    ⟨e2,ve⟩ → v2;
    v = #v1+v2#;
  }
  step ⟨Times[e1,e2],ve⟩ → v
  {
    ⟨e1,ve⟩ → v1;
    ⟨e2,ve⟩ → v2;
    v = #v1*v2#;
  }
}
transition ⟨BoolExp,#Env<Integer>#⟩ → #Boolean#
{
  step ⟨Eq[e1,e2],ve⟩ → v
  {
    ⟨e1,ve⟩ → v1;
    ⟨e2,ve⟩ → v2;
    v = # v1.equals(v2) #;
  }
  step ⟨Not[b],ve⟩ → v
  {
    ⟨b,ve⟩ → v0;
    v = # !v0 #;
  }
  step ⟨And[b1,b2],ve⟩ → v
  {
    ⟨b1,ve⟩ → v1;
    ⟨b2,ve⟩ → v2;
    v = # v1 && v2 #;
  }
}
transition ⟨Command,#Env<Integer>#⟩ → #Env<Integer>#
{
  step ⟨Assign[i,e],ve⟩ -> ve0
  {
    ⟨e,ve⟩ → v;
    ve0 = #new Env<Integer>(ve)#; # ve0.put(i,v); #
  }
  after  {# System.out.println("Assignment " + i + " = " + ve0.get(i)); #}
  step ⟨Var[i,c],ve⟩ → ve0
  {
    ve1: #Env<Integer># = # new Env<Integer>(ve) #;
    v: #Integer# = # ve1.put(i,0) #;
    ⟨c,ve1⟩ → ve2;
    ve0 = # new Env<Integer>(ve2) #; # ve0.put(i,v); #
  }
  step ⟨Seq[c1,c2],ve⟩ → ve0
  {
    ⟨c1,ve⟩ → ve1;
    ⟨c2,ve1⟩ → ve0;
  }
  step ⟨If2[b,c1,c2],ve⟩ → ve0
  {
    ⟨b,ve⟩ → v; # check(v); #
    ⟨c1,ve⟩ → ve0;
  }
  or
  {
    ⟨b,ve⟩ → v; # check(!v); #
    ⟨c2,ve⟩ → ve0;
    }
    step ⟨If1[b,c],ve⟩ → ve0
    {
      ⟨b,ve⟩ → v; # check(v); #
      ⟨c,ve⟩ → ve0;
    }
    or
    {
      ⟨b,ve⟩ → v; # check(!v); #
      ve0 = ve;
    }
    step ⟨w=While[b,c],ve⟩ → ve0
    {
      ⟨b,ve⟩ → v; # check(!v); #
      ve0 = ve;
    }
    or
    {
      ⟨b,ve⟩ → v; # check(v); #
      ⟨c,ve⟩ → ve1;
      ⟨w,ve1⟩ → ve0;
    }
  }
  transition ⟨Program⟩ → #Void#
  {
    step ⟨Prog[c]⟩ → none
    {
      ve0: #Env<Integer># = # new Env<Integer>() #;
      ⟨c,ve0⟩ → ve1;
      none = # null # ;
    }
  }
}
// ----------------------------------------------------------------------------
// end of file
// ----------------------------------------------------------------------------
```