SLANG

A Semantics-Based Language Generator

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University Linz, Austria

Joint work with William Steingartner, Department of Computers and Informatics, Technical University of Košice, Slovak Republic. Supported by the Slovak Academic Information Agency SAIA project 2023-03-15-001 “Semantics-Based Rapid Prototyping of Domain-Specific Languages”.

JOHANNES KEPLER UNIVERSITY LINZ

The SLANG Semantics-Based Language Generator

A tool for the rapid prototyping of programming languages.

- Syntactic domains.
  - Context-free grammar: a definition of types of abstract syntax trees.
- Printer & parser.
  - Mappings of abstract syntax trees to/from character strings.
- Type checker.
  - Formal type system: a logical inference system on abstract syntax trees.
- Interpreter.
  - Denotational semantics: a mathematical function on abstract syntax trees.
  - Big-step operational semantics: a relation on abstract syntax trees.

David A. Schmidt: *The Structure of Typed Programming Languages*, 1994.
Dana Scott and Christopher Strachey: *Towards a Mathematical Semantics for Computer Languages*, 1971.
Gilles Kahn: *Natural Semantics*, 1987.

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

Running Example: A Simple Imperative Language

```
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
```

Syntactic Domains

$N \in Numeral$, $I \in Identifier$

$P \in Program$, $C \in Command$, $E \in Expression$, $B \in BoolExp$

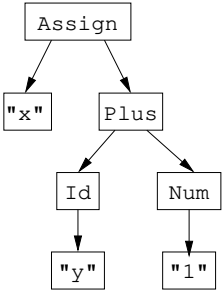
$P := C.$

$C := I\texttt{:=}E \mid \texttt{var}\ I\texttt{;}C \mid C_1\texttt{;}C_2 \mid \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \mid \texttt{while}\ B\ \texttt{do}\ C.$

$E := N \mid I \mid E_1\texttt{+}E_2 \mid E_1\texttt{*}E_2.$

$B := E_1\texttt{=}E_2 \mid \texttt{\~{}}B \mid B_1\texttt{/\backslash}B_2.$

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



Types of abstract syntax trees.

SLANG: Printer

```
printer {
  domain Program {
    case Prog[c] → # _result = c.toString(); #;
  }
  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 If[b,c1,c2] → # _result = "if " + b + " then " + c1 + " else " + c2; #;
    case While[b,c] → # _result = "while " + b + " do " + c; #;
  }
  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 + ")"; #;
  }
}
```

SLANG: Parser

```
parser antlr4 {
  domain Program {
    case # c=dCommand EOF # → Prog[c];
  }
  domain Command {
    case # i=dID ':=' e=dExp # → Assign[i,e];
    case # 'if' b=dBoolExp 'then' c1=dCommand 'else' c2=dCommand # → If[b,c1,c2];
    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 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
  }
}
```

Type System

Rules for $\vdash P$: program:

$$\frac{te = \emptyset \quad te \vdash C : \text{command}}{\vdash C : \text{program}}$$

Rules for $te \vdash C$: command:

$$\frac{I \in te \quad te \vdash E : \text{exp}}{te \vdash I := E : \text{command}} \qquad \frac{te_0 = te \cup \{I\} \quad te_0 \vdash C : \text{command}}{te \vdash \texttt{var}\ I\,;C : \text{command}} \qquad \frac{te \vdash C_1 : \text{command} \quad te \vdash C_2 : \text{command}}{te \vdash C_1\,;C_2 : \text{command}}$$

$$\frac{te \vdash B : \text{boolexp} \quad te \vdash C_1 : \text{command} \quad te \vdash C_2 : \text{command}}{te \vdash \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2 : \text{command}} \qquad \frac{te \vdash B : \text{boolexp} \quad te \vdash C : \text{command}}{te \vdash \texttt{while}\ B\ \texttt{do}\ \texttt{C} : \text{command}}$$

Rules for $te \vdash E$: exp:

...

Rules for $te \vdash B$: boolexp:

...

SLANG: Judgments and Inferences

```
judgment ⊢ Program: program {
  inference ⊢ Prog[c]: program {
    te: #Env<Type># = #new Env<Type>()# ; te ⊢ c: command;
  }
}
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 ⊢ If[b,c1,c2]: command {
    te ⊢ b: bexp; te ⊢ c1: command; te ⊢ c2: command;
  }
  inference te ⊢ While[b,c]: command {
    te ⊢ b: bexp; te ⊢ c: command; }
}
judgment #Env<Type># ⊢ Exp: exp { ... }
judgment #Env<Type># ⊢ BoolExp: bexp { ... }
```

SLANG: Auxiliary Code

```
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); }
#}
```

Throwing an exception signals a type error.

Denotational Semantics

$[\![\mathbf{P}]\!] : \mathbf{ValueEnv}$

$[\![C]\!] = \text{let } ve = \emptyset \text{ in} [\![C]\!](ve)$

$[\![\mathbf{C}]\!] : \mathbf{ValueEnv} \to \mathbf{ValueEnv}$

$[\![I \texttt{:=} E]\!](ve) = ve[I \mapsto [\![E]\!](ve)]$

$[\![\texttt{var}\ I\texttt{;}C]\!](ve) = \text{let } ve_0 = ve[I \mapsto 0], ve_1 = [\![C]\!](ve_0) \text{ in } ve_1[I \mapsto ve(I)]$

$[\![C_1\texttt{;}C_2]\!](ve) = \text{let } ve_0 = [\![C_1]\!](ve) \text{ in } [\![C_2]\!](ve_0)$

$[\![\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!](ve) = \text{if } [\![B]\!](ve) = \text{true then } [\![C_1]\!](ve) \text{ else } [\![C_2]\!](ve)$

$[\![\texttt{while}\ B\ \texttt{do}\ C]\!](ve) = \text{let } w = \text{lfp } w.\ \lambda ve_0.\ \text{if } [\![B]\!](ve_0) = \text{false then } ve_0 \text{ else } w([\![C]\!](ve_0)) \text{ in } w(ve)$

$[\![\mathbf{E}]\!] : \mathbf{ValueEnv} \to \mathbb{Z}$

...

$[\![\mathbf{B}]\!] : \mathbf{ValueEnv} \to \mathbb{B}$

...

SLANG: Functions and Equations

```
function ⟦Program⟧: #Void# {
  equation ⟦Prog[c]⟧ = none
  { ve0: #Env<Integer># = # new Env<Integer>() #; ve1 = ⟦c⟧(ve0); none = # null # ; }
}
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 ⟦If[b,c1,c2]⟧(ve) = ve0
  { v = ⟦b⟧(ve); # if (v) # ve0 = ⟦c1⟧(ve); # else # ve0 = ⟦c2⟧(ve); }
  equation ⟦While[b,c]⟧(ve) = ve0
  { ve0 = ve; # while (true) { # v = ⟦b⟧(ve0); # if (!v) break; # ve0 = ⟦c⟧(ve0); # } # }
}
```

Alternative: Big-Step Operational Semantics

Rules for $P \to ve_0$

$$\frac{ve = \emptyset \quad \langle C, ve\rangle \to ve_0}{C \to ve_0}$$

Rules for $\langle C, ve\rangle \to ve_0$

$$\frac{\langle E, ve\rangle \to v \quad ve_0 = ve[I \mapsto v]}{\langle I\texttt{:=}E, ve\rangle \to ve_0} \qquad \frac{ve_1 = ve[I \mapsto 0] \quad \langle C, ve_1\rangle \to ve_2 \quad ve_0 = ve_2[I \mapsto ve(I)]}{\langle \texttt{var}\ I\texttt{;}C, ve\rangle \to ve_0}$$

$$\frac{\langle C_1, ve\rangle \to ve_1 \quad \langle C_2, ve_1\rangle \to ve_0}{\langle C_1\texttt{;}C_2, ve\rangle \to ve_0}$$

$$\frac{\langle B, ve\rangle \to v \quad v = \text{true} \quad \langle C_1, ve\rangle \to ve_0}{\langle \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2, ve\rangle \to ve_0} \qquad \frac{\langle B, ve\rangle \to v \quad v = \text{false} \quad \langle C_2, ve\rangle \to ve_0}{\langle \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2, ve\rangle \to ve_0}$$

$$\frac{\langle B, ve\rangle \to v \quad v = \text{false} \quad ve_0 = ve}{\langle \texttt{while}\ B\ \texttt{do}\ C, ve\rangle \to ve_0}$$

$$\frac{\langle B, ve\rangle \to v \quad v = \text{true} \quad \langle C, ve\rangle \to ve_1 \quad \langle \texttt{while}\ B\ \texttt{do}\ C, ve_1\rangle \to ve_0}{\langle \texttt{while}\ B\ \texttt{do}\ C, ve\rangle \to ve_0}$$

SLANG: Transitions and Steps

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

SLANG: Auxiliary Code

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

Throwing an exception lets the current inference fail and attempts the next one.

SLANG: The Complete Specification File

```
language Imp { // Imp.txt with denotational semantics, Imp2.txt with operational one
  // code generation information
  target java {
    header {#
      package lang.imp;
      import static lang.imp.Imp.*;
      import java.util.*;
    #}
  }

  // abstract and concrete syntax -> printer/parser
  domains { ... }
  printer { ... }
  parser antlr4 { ... }

  // formal type system -> type checker
  code {# ... #}
  judgment ⊢ Program: program { ... }
  judgment #Env<Type># ⊢ Command: command
  ...
  // formal (denotational/operational) semantics -> interpreter
  code {# ... #}
  function ⟦Program⟧: #Void# { ...} // transition ⟨Program⟩ ⟶ #Void# { ...}
  function ⟦Command⟧: #Env<Integer># ⟶ #Env<Integer># { ...} // transition ⟨Command,#Env<Integer>#⟩ ⟶ #Env<Integer># { ...}
  ...
}
```

Generation of Java Classes and Functions

```
> SLANG -d lang/imp Imp.txt
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.
```

Integration into Application Context

```
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());
    }
  }
}
```

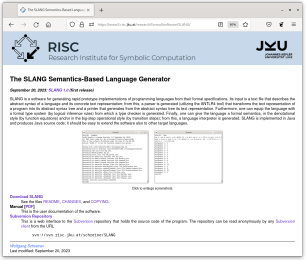
Compilation and Execution of Application

```
> antlr4 lang/imp/Imp.g4
> javac -cp ".:/software/SLANG/lib/*" ImpMain.java
> java -cp ".:/software/SLANG/lib/*" ImpMain

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
```

Summary



- Goal: a pragmatic tool for semantics-based language prototyping.
  - Definitions of operations by structural induction on syntactic domains.
  - Generation of source code that can be easily integrated into applications.
  - Computations based on embedded code in target language.
  - Supports execution but not really formal reasoning.
- Next: application to the rapid prototyping of domain-specific languages.

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