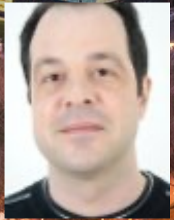
Domain Specific Language Design (2IMP20): Static and Dynamic Semantics

Loek Cleophas



Ivan Kurtev

Department of Mathematics and Computer Science — Software Engineering and Technology cluster

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Background information

Reading:

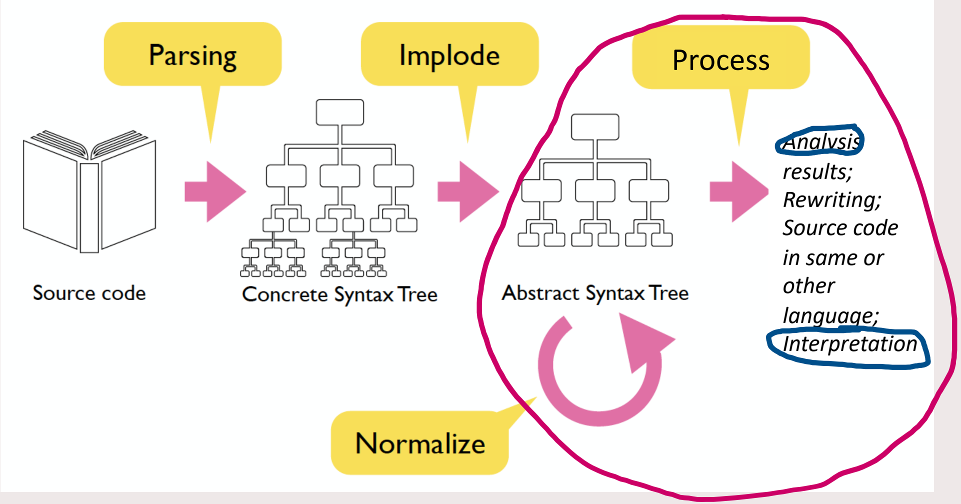
- Chapter 3, 4 and 5 of *Introduction to Compiler Design*
  Covers scoping, interpretation, type checking
- Chapter 8 and 9 of *Software Languages*
  Covers operational semantics + type checking

Implementation of a DSL

Definition of a (programming) language involves:

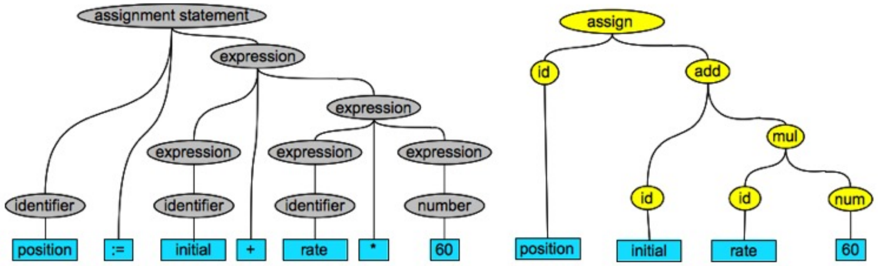
- abstract syntax
- concrete syntax:
  - textual syntax
  - graphical syntax
- semantics:
  - static
  - dynamic

Semantics



For the input sentence

the parse tree (left) and abstract syntax tree (right) may look as follows:



(Source: https://www.rascal-mpl.org/docs/Rascalopedia/ParseTree/)

TU/e

Static semantics

Symbol table/type environment for block structured languages in Rascal:

```
alias SENV = tuple[map[Id, TYPE] symbols];
alias TABLES = tuple[list[SENV] blocks];

TABLES enterBlock(TABLES tbls, SENV senv) = <push(senv, tbls.blocks)>;
TABLES leaveBlock(TABLES tbls) = <tail(tbls.blocks)>;
```

Per block a separate type environment is required

- Created when entering a block
- Destroyed when leaving a block

Finding a declared variable involves inspecting all type environments

Static semantics

Scope and visibility

- A *binding occurrence* of identifier `I` is an occurrence where `I` is bound to entity `X`
- An *applied occurrence* of `I` is an occurrence where use is made of entity `X` to which `I` is bound
- Each applied occurrence of `I` should correspond to exactly one binding occurrence of `I`
  - An identifier `I` may be defined in multiple blocks
- Nested blocks, some outer block contains a declaration of `I`:
  - Inner block does not contain a declaration of `I` → declaration is visible throughout outer and inner blocks
  - Inner block contains a declaration of `I` → inner block declaration hides outer block declaration
  - This leads to “nested” symbol tables

Static semantics

Static vs dynamic scoping

- A language is ***statically scoped*** if the body of a procedure is executed in the environment of the procedure's **definition**
  - compile-time binding of identifiers

- A language is ***dynamically scoped*** if the body of a procedure is executed in the environment of the procedure **call**
  - run-time binding of identifiers

- Nearly all programming languages (C, C++, Java, etc.) are statically scoped

Static semantics

Static vs dynamic scoping

- **Consider:**

```
const int s = 2;
int f(int x) { return s * x;}

void p(int y) { print(f(y));}
p(3)                                                    (1)

void q(int z) { const int s = 3; print(f(z));}
q(3)                                                    (2)
```

- What is the value printed at (1) and (2)?

Questions?

Static semantics — Declarations

A *declaration* is a language construct to produce a binding

Types of simple declarations:

- type
- constant
- variable
- procedure

Procedure and function definitions

A *procedure definition* binds an identifier to a procedure

- function
- procedure

- A function definition:
```
bool even(int n) {return (n % 2 == 0);}
```
- A procedure definition:
```
void double(int& n) { n *= 2; }
```

Procedure and function definitions

A function/*procedure* is an entity that embodies a computation

- a *function* embodies an expression to be evaluated
- a *procedure* embodies a command to be executed

Both implement procedural abstraction

Parameters and arguments

Parameters and arguments

- An *argument* is a value or other entity that is passed to a procedure
- An *actual parameter* is an expression that yields an argument
- A *formal parameter* is an identifier through which a procedure can access an argument
- Association between formal parameter and argument is called *parameter mechanism*
  - two basic concepts:
    - copy parameter mechanism
    - reference parameter mechanism

Parameters and arguments

Copy parameter mechanism allows the transfer of values to and from a procedure

- *Pass by value*
- A formal parameter `FP` denotes a local variable of the procedure

Reference parameter mechanism: A *reference parameter* allows for the formal parameter `FP` to be bound directly to the argument

- *Pass by reference*
- Modifications of the value may be visible outside the body of the procedure

Overloading

An identifier is *overloaded* if it denotes two or more distinct procedures in the same scope

- In many programming languages operators for certain built-in functions are overloaded
- "-" operator:
  - integer negation (`Integer → Integer`)
  - floating-point negation (`Float → Float`)
  - integer subtraction (`Integer×Integer → Integer`)
  - floating-point subtraction (`Float×Float → Float`)

Overloading

Identifier $\mathtt{F}$ denotes

- function ($\mathtt{f}_1$) of type $\mathtt{S}_1 \rightarrow \mathtt{T}_1$
- function ($\mathtt{f}_2$) of type $\mathtt{S}_2 \rightarrow \mathtt{T}_2$

- *context-independent overloading* requires $\mathtt{S}_1$ and $\mathtt{S}_2$ are non-equivalent
  - if actual parameter $\mathtt{E}$ of $\mathtt{F(E)}$ is of type $\mathtt{S}_1$ then $\mathtt{F}$ denotes $\mathtt{f}_1$
  - if $\mathtt{E}$ is of type $\mathtt{S}_2$ then $\mathtt{F}$ denotes $\mathtt{f}_2$

- with context-independent overloading *the function can be uniquely identified by the type of the actual parameter*

Overloading

Identifier $\mathtt{F}$ denotes

- function ($\mathtt{f}_1$) of type $\mathtt{S}_1 \rightarrow \mathtt{T}_1$
- function ($\mathtt{f}_2$) of type $\mathtt{S}_2 \rightarrow \mathtt{T}_2$

- *context-dependent overloading* requires $\mathtt{S}_1$ and $\mathtt{S}_2$ are non-equivalent or $\mathtt{T}_1$ and $\mathtt{T}_2$ are non-equivalent
  - if $\mathtt{S}_1$ and $\mathtt{S}_2$ are non-equivalent, see previous slide
  - if $\mathtt{S}_1$ and $\mathtt{S}_2$ are equivalent, but $\mathtt{T}_1$ and $\mathtt{T}_2$ are non-equivalent the *context must be taken into consideration*
    - if the context $\mathtt{F(E)}$ is of type $\mathtt{T}_1$ then $\mathtt{F}$ denotes $\mathtt{f}_1$
    - if the context is of type $\mathtt{T}_2$ then $\mathtt{F}$ denotes $\mathtt{f}_2$

- with context-dependent overloading, it is possible to formulate expressions which cannot be uniquely identified (How?)

Typing

Statically typed programming languages insist on explicit declaration of the type of entity in programs

- `integer I := E`
- In *Python* we can write a definition `I = E` where type of I is not explicitly stated, but *inferred* from `E`

Type inference is a process where the type of a declared entity is inferred instead of explicitly stated

TU/e

Types and Data

What is the relation between data and types?

- Values are entities manipulated by programs
- Different types of (programming) languages support different types of values

- Values are grouped into types
- A *type* is a set of values with operations that can be applied uniformly to all these values

Type Systems

A *type system* of a (programming) language is a set of rules

- Groups values into types
- Prevents illegal operations, like multiplication of strings by booleans: *type error*

- *Statically typed* language: each variable and expression has a fixed type
  - all operands can be type-checked at *compile-time*
- *Dynamically typed* language: values have fixed type, but variables and expressions have no fixed type.
  - operands can be only type-checked at *run-time*

Type Inferencing

Inference rules:

```
P₁      …       Pₙ
-------------          [l]
       C
```

where P_1, …, P_n and C are *judgements* (*premises + conclusion*) and l is simply a *label* used as reference.

If n = 0, the judgement is an *axiom*

```
C                      [l]
```

Type Inferencing for Type Checking

Typing rules can be defined as inference rules

- Given the following context-free grammar:

```
syntax
  Expr = “true” | “false” | “zero” |
         “succ” Expr | “pred” Expr | “iszero” Expr |
         “if” Expr “then” Expr “else” Expr
```

For type checking, *inference rules* as typing rules have to be defined

Type Inferencing

Typing rules can be defined as inference rules

- The following typing rules as inference rules must be defined

```
true  : booltype        [true]
false : booltype        [false]
zero  : nattype         [zero]

   e : nattype                          e : nattype
   -----------          [succ]          -----------          [pred]
succ(e) : nattype                     pred(e) : nattype

   e : nattype
   -----------          [iszero]
iszero(e) : booltype

e0 : booltype   e1 : T   e2 : T
-------------------------------     [if]
      if(e0, e1, e2) : T
```

A type checker for PICO

- The types are **`natural`** and **`string`**
- All variables should be declared before use

- Lhs and Rhs of assignment should have equal type
- The test in while and if-then should be natural

- Operands of $+$ and $-$ should be natural; result is natural
- Operands of `||` should be string; result string

Syntax directed definitions

- For every construct in the abstract syntax
  define a rule to describe
  the semantics/translation of that language construct

Recall PICO abstract syntax

```
public data PROGRAM =
     program(list[DECL] decls, list[STATEMENT] stats);

public data DECL =
  decl(PicoId name, TYPE tp);

public data EXP =
       id(PicoId name)
     | natCon(int iVal)
     | strCon(str sVal)
     | add(EXP left, EXP right)
     | sub(EXP left, EXP right)
     | conc(EXP left, EXP right);

public data STATEMENT =
       asgStat(PicoId name, EXP exp)
     | ifElseStat(EXP exp, list[STATEMENT] thenpart,
                           list[STATEMENT] elsepart)
     | whileStat(EXP exp, list[STATEMENT] body);
```

Some preliminaries: type environment and error handling

```
alias TENV = tuple[ map[PicoId, TYPE] symbols,
                              list[tuple[loc l, str msg]] errors];

TENV addError(TENV env, loc l, str msg) =
               env[errors = env.errors + <l, msg>];

str required(TYPE t, str got) = "Required <getName(t)>, got <got>";
str required(TYPE t1, TYPE t2) = required(t1, getName(t2));
```

A type checker for PICO

Checking basic expressions

```
TENV checkExp(exp:natCon(int N), TYPE req, TENV env)=
  req == natural() ? env :
          addError(env, exp@location, required(req, "natural"));

TENV checkExp(exp:strCon(str S), TYPE req, TENV env) =
 req == string() ? env :
         addError(env, exp@location, required(req, "string"));

TENV checkExp(exp:id(PicoId Id), TYPE req, TENV env){
  if(!env.symbols[Id]?)
     return addError(env, exp@location, "Undeclared variable <Id>");
  tpid = env.symbols[Id];
  return req == tpid ? env :
                 addError(env, exp@location, required(req, tpid));
}
```

A type checker for PICO

Checking binary expressions

```
TENV checkExp(exp:add(EXP E1, EXP E2), TYPE req, TENV env)=
  req == natural() ? checkExp(E1, natural(), checkExp(E2, natural(), env))
                   : addError(env, exp@location, required(req, "natural"));

TENV checkExp(exp:sub(EXP E1, EXP E2), TYPE req, TENV env) =
  req == natural() ? checkExp(E1, natural(), checkExp(E2, natural(), env))
                   : addError(env, exp@location, required(req, "natural"));

TENV checkExp(exp:conc(EXP E1, EXP E2), TYPE req, TENV env) =
  req == string() ? checkExp(E1, string(), checkExp(E2, string(), env))
                   : addError(env, exp@location, required(req, "string"));
```

A type checker for PICO

Checking simple statements

```
TENV checkStat(stat:asgStat(PicoId Id, EXP Exp), TENV env) {
  if(!env.symbols[Id]?)
     return addError(env, stat@location, "Undeclared variable <Id>");
  tpid = env.symbols[Id];
  return checkExp(Exp, tpid, env);
}

TENV checkStat(stat:ifElseStat(EXP Exp,
               list[STATEMENT] Stats1, list[STATEMENT] Stats2),
               TENV env) {
    env0 = checkExp(Exp, natural(), env);
    env1 = checkStats(Stats1, env0);
    env2 = checkStats(Stats2, env1);
    return env2;
}
```


TU/e

A type checker for PICO

Checking simple statements

```
TENV checkStat(stat:whileStat(EXP Exp,
                 list[STATEMENT] Stats1),
                   TENV env){
    env0 = checkExp(Exp, natural(), env);
    env1 = checkStats(Stats1, env0);
    return env1;
}

TENV checkStats(list[STATEMENT] Stats1, TENV env){
 for(S <- Stats1){
      env = checkStat(S, env);
 }
 return env;
}
```


TU/e

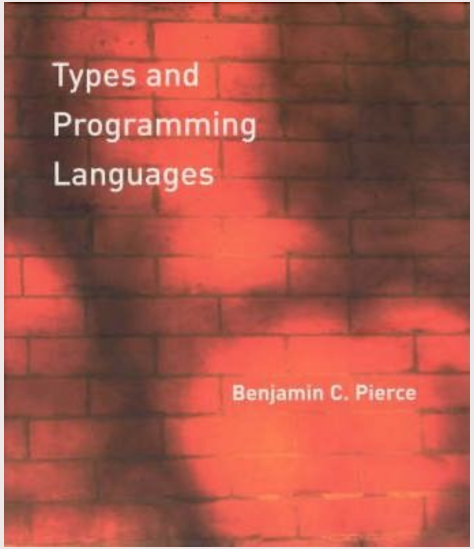
A type checker for PICO

Checking declarations and a program

```
TENV checkDecls(list[DECL] Decls) =
  <( Id : tp | decl(PicoId Id, TYPE tp) <- Decls), []>;

public TENV checkProgram(PROGRAM P){
  if(program(list[DECL] Decls, list[STATEMENT] Series) := P){
     TENV env = checkDecls(Decls);
     return checkStats(Series, env);
  } else
    throw "Cannot happen";
}
```

The typing bible



Questions?

Dynamic semantics

Definition of a (programming) language involves:

- **abstract syntax, so-called signature**
- **concrete syntax:**
  - **textual syntax**
  - **graphical syntax**

- **semantics:**
  - **static semantics**
  - **dynamic semantics**

Dynamic semantics

Semantics *represents the intended meaning* of a language and language constructs

Semantics is a *means to represent our understanding* of a model/program (**what it does**) and

To *communicate our understanding* to other entities

To understand what happens in a computer/machine *when a program/model is executed*

Dynamic semantics

Not every language is suitable for describing dynamic semantics

- C and Java are not really suited for describing the dynamic semantics, because of lack of a formal description

Dynamic semantics ideally is described in a *formal language* (*formal semantics*) because:

- *ambiguities and inconsistencies can be detected* in a model which appears to be “ok”
- this is the basis for *analysis*, *validation* and *verification*, but also *implementation*

TU/e

Dynamic semantics

Dynamic semantics in practice

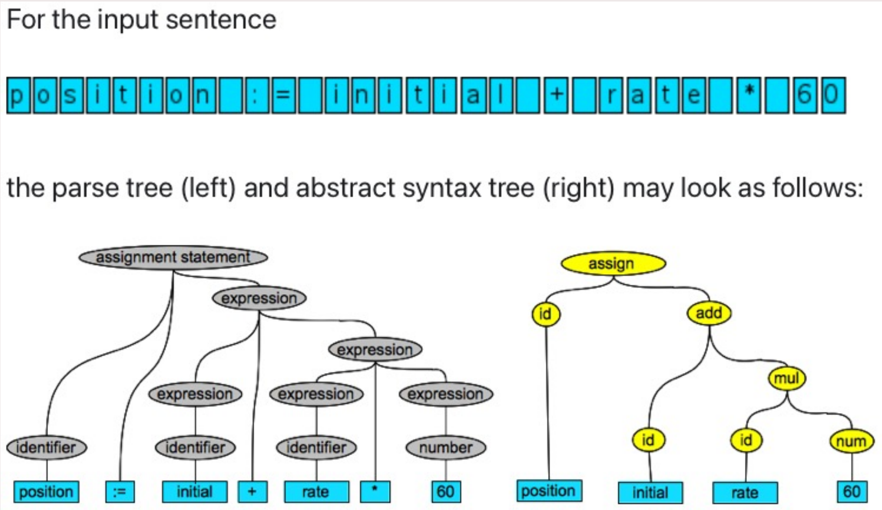
- Semantics of languages is often defined by *translation* or *interpretation*
- Languages evolve over time and semantic ambiguities and conflicts may be introduced

Dynamic semantics

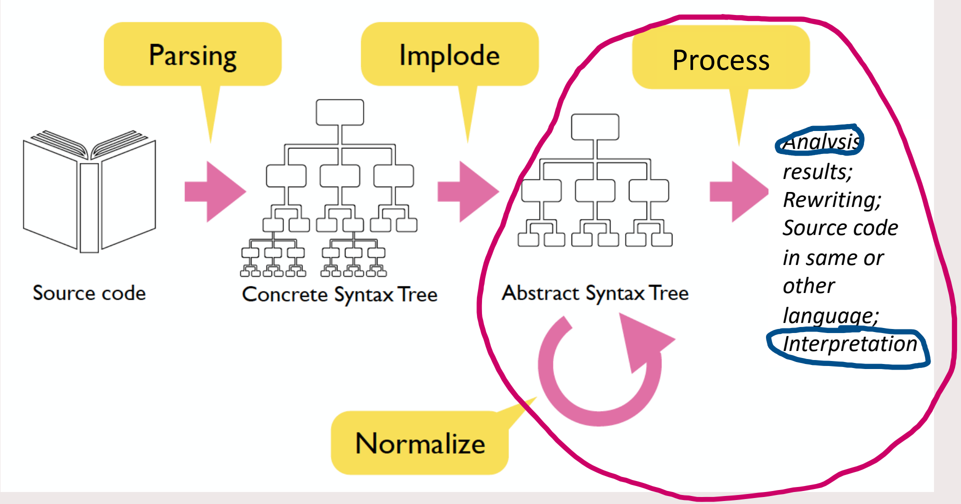
Formal dynamic semantics

- Modern systems models are
  - complex
  - (possibly) at a high level of abstraction
- Reasoning about the models using rigorous methods
  - need to find existing *ambiguities and inconsistencies*
  - need to keep a (modeling) language *“clean and simple”*
- Formal semantics allows for
  - analysis, validation and verification, but also (correct) implementation
  - model/program comparison, thus optimization, modification

Semantics



(Source: https://www.rascal-mpl.org/docs/Rascalopedia/ParseTree/)



TU/e

Operational semantics

- specifies how (step-by-step) program/model executes
- specifies how states/configurations are modified during the execution
- all possible executions are generated
- underlying model is the model of *transition systems* (a program/model execution is turned into a transition system)

Operational semantics

Big-step operational semantics deals with *relations* over program fragments and data structures to represent *values* and *stores*:

- *Values* represent values of evaluation
- *Stores* are maps from variable names to values

Judgements are claims of relationships, a judgement is a basic formula (a claim to be proven based on the operational semantics specification):

- `evaluate(e,v)`: a judgement of the evaluation of **expression** e to value v;
- `execute(s,m,m')`: a judgement of the execution of an **imperative statement** s for a store m and a resulting store m';
- `evaluate(fs,m,e,v)`: a judgement of the evaluation of expression e in a **functional program** to value v relative to a given collection of functions fs and an environment m.

Operational semantics

An operational semantic specification consists of inference rules:

```
P₁     …       Pₙ
-------------         [l]
       C
```

where P_1, …, P_n and `C` are judgements (*premises + conclusion*) and `l` is simply a label used as reference.

If n = 0, the judgement is an axiom

```
C                     [l]
```


TU/e

Operational semantics

The definition of operational semantics is syntax-driven:

- the conclusion applies to a specific syntactic construct
- the premises may apply to subterms of this construct or terms formed over it

Example:

```
evaluate(zero, zero)          [zero]

       evaluate(e, n)
==========================    [succ]
evaluate(succ(e), succ(n))
```

Operational semantics

Operational semantics of BTL (from book of Lämmel):

```
evaluate(true, true)          [true]

evaluate(false, false)        [false]

evaluate(zero, zero)          [zero]

       evaluate(e, n)
==========================    [succ]
evaluate(succ(e), succ(n))

       evaluate(e, zero)
==========================    [pred1]
evaluate(pred(e), zero)

       evaluate(e, succ(n))
==========================    [pred2]
evaluate(pred(e), n)
```

Operational semantics

```
      evaluate(e, zero)
==========================  [iszero1]
evaluate(iszero(e), true)

      evaluate(e, succ(n))
==========================  [iszero2]
evaluate(iszero(e), false)


evaluate(e0, true)       evaluate(e1, v1)
========================================  [if1]
evaluate(if(e0, e1, e2), v1)

evaluate(e0, false)      evaluate(e2, v2)
========================================  [if2]
evaluate(if(e0, e1, e2), v2)
```

Operational semantics

Derivation tree for evaluating a BTL expression:

```
   evaluate(zero, zero)
  ==========================  [iszero1]
  evaluate(iszero(zero), true)            evaluate(succ(zero), succ(zero))
======================================================================== [if1]
         evaluate(if(iszero(zero),succ(zero),zero),succ(zero))
======================================================================  [pred2]
        evaluate(pred(if(iszero(zero),succ(zero),zero)), zero)
```

Operational semantics

Derivation tree for evaluating a BTL expression:

```
   evaluate(zero, zero)
  ==========================  [iszero1]
  evaluate(iszero(zero), true)             evaluate(succ(zero), succ(zero))
========================================================================= [if1]
       evaluate(if(iszero(zero),succ(zero),zero),succ(zero))
=======================================================================  [pred2]
      evaluate(pred(if(iszero(zero),succ(zero),zero)), zero)
```

Operational semantics

Derivation tree for evaluating a BTL expression:

```
   evaluate(zero, zero)                     evaluate(zero, zero)
  ==========================  [iszero1]   ===========================  [succ]
  evaluate(iszero(zero), true)            evaluate(succ(zero), succ(zero))
========================================================================= [if1]
         evaluate(if(iszero(zero),succ(zero),zero),succ(zero))
======================================================================  [pred2]
        evaluate(pred(if(iszero(zero),succ(zero),zero)), zero)
```

Operational semantics

Proof of a big-step judgement has a tree-like shape, so called *derivation trees*:

- Each node in a derivation tree is an instance of a conclusion of an inference rule
- Subtrees of a node are instances of the premises of the same rule
- Leaf nodes of the tree are instance of axioms
- Instances are metavariables in the rules consistently replaced by phrases of data structures of the appropriate sort

Questions?

A PICO interpreter based on big-step “operational” semantics

Definition of an interpreter for PICO:

- Natural variables are initialized to 0
- String variables are initialized to “”

- Variable on lhs of assignment gets value of Rhs
- Variable (in expression) evaluates to its current value

- Test in while and if-then equal to 0 => false
- Test in while and if-then not equal to 0 => true

The Pico interpreter “transforms” a Pico program to the output it generates, by stepwise reduction in a syntax directed manner

This interpreter implements the *big-step “operational” semantics*

A PICO interpreter — Values

```
data PicoValue =
  natval(int n) |
  strval(str s) |
  errorval(loc l, str msg);
```

`errorval` denotes error messages and their locations

A PICO interpreter — Value environment

```
Alias VENV
    map[[PicoId, PicoValue]]
```

A PICO interpreter — Evaluating expressions

```
PicoValue evalExp(exp:natCon(int N), VENV env) = natval(N);

PicoValue evalExp(exp:strCon(str S), VENV env) = strval(S);

PicoValue evalExp(exp:id(PicoId Id), VENV env)  =
    env[Id]?  ? env[Id] : errorval(exp@location, "Uninitialized variable <Id>");

PicoValue evalExp(exp:add(EXP E1, EXP E2), VENV env) =
   (natval(n1) := evalExp(E1, env) &&
    natval(n2) := evalExp(E2, env)) ? natval(n1 + n2)
                                    : errorval(exp@location, "+ requires natural 
arguments");
```

A PICO interpreter — Evaluating expressions

```
PicoValue evalExp(exp:sub(EXP E1, EXP E2), VENV env) =
    (natval(n1) := evalExp(E1, env) &&
     natval(n2) := evalExp(E2, env)) ? natval(n1 - n2)
                                     : errorval(exp@location,
                                          "- requires natural arguments");

PicoValue evalExp(exp:conc(EXP E1, EXP E2), VENV env) =
    (strval(s1) := evalExp(E1, env) &&
     strval(s2) := evalExp(E2, env)) ? strval(s1 + s2)
                                     : errorval(exp@location,
                                          "|| requires string arguments");
```

A PICO interpreter — Evaluating lists of statements

```
VENV evalStats(list[STATEMENT] Stats1, VENV env) {
    for (S <- Stats1){
        env = evalStat(S,env);
    }
    return env;
}
```

A PICO interpreter — Evaluating a statement

```
VENV evalStat(stat:asgStat(PicoId Id, EXP Exp), VENV env) {
  env[Id] = evalExp(Exp, env);
  return env;
}

VENV evalStat(stat:ifElseStat(EXP Exp, list[STATEMENT] Stats1,
                                 list[STATEMENT] Stats2),
              VENV env) =
  evalStats(evalExp(Exp, env) != natval(0) ? Stats1 : Stats2, env);

VENV evalStat(stat:whileStat(EXP Exp, list[STATEMENT] Stats1),
              VENV env) {
    while(evalExp(Exp, env) != natval(0)){
       env = evalStats(Stats1, env);
    }
    return env;
}
```

Evaluating declarations and a Pico program

```
VENV evalDecls(list[DECL] Decls) =
    ( Id : (tp == demo::lang::Pico::Abstract::natural() ?
           natval(0) : strval(""))
    | decl(PicoId Id, TYPE tp) <- Decls);

public VENV evalProgram(PROGRAM P){
  if(program(list[DECL] Decls, list[STATEMENT] Series) := P){
     VENV env = evalDecls(Decls);
     return evalStats(Series, env);
  } else
    throw "Cannot happen";
}
```

Translational Semantics

An alternative route to define the semantics of a DSL is to generate code for some general-purpose programming language:

- Java
- C
- Haskell

No real guarantee with respect to completeness and consistency because of lack of precise semantics of many general-purpose programming languages

Dynamic semantics

Research related topics

- Integration of (static and dynamic) semantics in a language workbench
- Size of semantic building block:
  - on the level of “assembler instructions”
  - on the level of “semantic patterns”: queues, state machines, channels, etc.

- Domain specific languages to describe both static and dynamic semantics

Questions?

Lookahead

- 2023/05/26: Ivan
- 2023/05/31: catch-up timeslot Ivan d/t earlier illness
- 2023/06/02: Ivan
- 2023/06/07 (rest of semantics) + DSL design (the bigger picture)
- 2023/06/09: Ivan
- 2023/06/14: guest lecture Eugen Schindler, Canon Production printing
- 2023/06/16: Ivan
- 2023/06/21: anything left + guest lecture Mauricio Verano Merino, VU

TU/e