

Funcons

reusable components of language specifications

Peter D. Mosses

Swansea University (emeritus)
TU Delft (visitor)

LangDev Meet-Up at CWI, Amsterdam, 8–9 March 2018

Contents

Motivation	<u>3</u>
Funcons	<u>7</u>
Component-based semantics	<u>13</u>
Tool support	<u>22</u>
Demo	<u>25</u>
Conclusion	<u>35</u>
References	<u>47</u>

Note: Slides 31–34 and 40–47 were not presented

Specification vs implementation

Suppose you were developing a new software language...

- ▶ would *you* specify a *formal semantics* of it?

Yes: a few languages from the 80s and 90s

- ▶ ADA, SCHEME, STANDARD ML, CONCURRENT ML

No: all other major programming languages

- ▶ HASKELL, OCAML, SCALA, JAVA, C#, ... (and most DSLs)

Funcons

Make formal semantics *easier* than BNF !

Encourage language developers to use formal semantics for:

- ***documentation***

- language features, design decisions

- ***implementation***

- rapid prototyping, exploration of design alternatives

Conjecture

Using a ***component-based*** semantic meta-language can significantly reduce the effort of language specification

Meta-language engineering

Meta-language requirements

- ▶ clear, concise, expressive *notation*
- ▶ solid *foundations*
- ▶ *tool support* for browsing, checking, validating
- ▶ ease of *co-evolution* of languages and specifications
- ▶ *reusable components*

Funcons

Funcons

- ▶ correspond to *fundamental programming concepts*
- ▶ language-*independent*
- ▶ have *fixed* behaviour
- ▶ specified *independently*
- ▶ *new* funcons can be added

Kinds of funcons

Computations

- ▶ **Normal:** flowing, giving, binding, storing, linking, generating, interacting, ...
- ▶ **Abnormal:** failing, throwing, returning, controlling, ...
- ▶ (Concurrent: not yet specified)

Kinds of funcons

Values *(some types are built-in)*

- ▶ **Primitive:** atoms, bools, ints, floats, chars, strings
- ▶ **Composite:** algebraic datatypes, tuples, lists, vectors, sets, multisets, maps, pointers, references, variants, ...
- ▶ **Abstractions:** closures, thunks, functions, patterns, ...
- ▶ none : no-value

Funcon library

```
# Values

## Types
[
  Type universe
  Type values      Alias vals
  Type types
  Type empty-type
  Funcon is-in-type      Alias is
  Type datatype-values
  Type no-value
  Funcon none
  Type cast-to-type      Alias cast
  Type is-defined         Alias is-def
  Type ground-values     Alias ground-vals
  Funcon is-equal        Alias is-eq
]

## Primitive values

### The null value
[
  Datatype unit-type
  Funcon null-value      Alias null
]

### Booleans
[
  Datatype booleans      Alias bools
  Funcon true
  Funcon false
  Funcon not
  Funcon implies
  Funcon and
  Funcon or
  Funcon exclusive-or    Alias xor
]

## Bits and bit vectors
[
  Type bits
  Datatype bit-vectors
  Type bytes      Alias octets
  Funcon bit-vector-not
  Funcon bit-vector-and
  Funcon bit-vector-or
  Funcon bit-vector-xor
  Funcon bit-vector-shift-left
  Funcon bit-vector-logical-shift-right
  Funcon bit-vector-arithmetic-shift-right
  Funcon integer-to-bit-vector
  Funcon bit-vector-to-integer
  Funcon bit-vector-to-natural
  Funcon unsigned-bit-vector-maximum
  Funcon signed-bit-vector-maximum
  Funcon signed-bit-vector-minimum
  Funcon is-in-signed-bit-vector
  Funcon is-in-unsigned-bit-vector
]

## Integers
[
  Type integers      Alias ints
  Type integers-from      Alias from
  Type integers-up-to      Alias up-to
  Type bounded-integers      Alias bounded-ints
  Type positive-integers      Alias pos-ints
  Type negative-integers      Alias neg-ints
  Type natural-numbers      Alias nats
  Funcon natural-successor      Alias nat-succ
  Funcon natural-predecessor    Alias nat-pred
  Funcon integer-add            Alias int-add
  Funcon integer-subtract       Alias int-sub
  Funcon integer-multiply       Alias int-mul
  Funcon integer-divide         Alias int-div
  Funcon integer-modulo         Alias int-mod
  Funcon integer-power          Alias int-pow
  Funcon integer-absolute-value Alias int-abs
  Funcon integer-negate         Alias int-neg
  Funcon integer-is-less        Alias is-less
  Funcon integer-is-less-or-equal Alias is-less-or-equal
  Funcon integer-is-greater      Alias is-greater
  Funcon integer-is-greater-or-equal Alias is-greater-or-equal
  equal
  Funcon binary-natural      Alias binary
  Funcon octal-natural       Alias octal
  Funcon decimal-natural     Alias decimal
  Funcon hexadecimal-natural Alias hexadecimal
]

## Floats
[
  Datatype float-formats
  Funcon binary32
  Funcon binary64
  Funcon binary128
  Funcon decimal164
  Funcon decimal128
  Type floats
  Type float
]

Funcon quiet-not-a-number      Alias qNaN
Funcon signaling-not-a-number  Alias sNaN
Funcon positive-infinity       Alias pos-inf
Funcon negative-infinity       Alias neg-inf
Funcon float-convert
Funcon float-equal
Funcon float-is-less
Funcon float-is-less-or-equal
Funcon float-is-greater
Funcon float-is-greater-or-equal
Funcon float-negate
Funcon float-absolute-value
Funcon float-add
Funcon float-subtract
Funcon float-multiply
Funcon float-multiply-add
Funcon float-divide
Funcon float-remainder
Funcon float-sqrt
Funcon float-integer-power
Funcon float-round-power
Funcon float-round-ties-to-even
Funcon float-round-ties-to-infinity
Funcon float-floor
Funcon float-ceiling
Funcon float-truncate
Funcon float-pi
Funcon float-e
Funcon float-log
Funcon float-log10
Funcon float-exp
Funcon float-sin
Funcon float-cos
Funcon float-tan
Funcon float-asin
Funcon float-acos
Funcon float-atan
Funcon float-sinh
Funcon float-cosh
Funcon float-tanh
Funcon float-asinh
Funcon float-acosh
Funcon float-atanh
Funcon float-atan2

Funcon datatype-value-atom
Funcon datatype-value-elements

]

## Tuples
[
  Datatype tuples
  Funcon tuple-elements
  Funcon tuple-zip
]

## Lists
[
  Datatype lists
  Funcon list
  Funcon list-elements
  Funcon list-nil      Alias nil
  Funcon list-cons      Alias cons
  Funcon list-head      Alias head
  Funcon list-tail      Alias tail
  Funcon list-length
  Funcon list-append
]

## Vectors
[
  Datatype vectors
  Funcon vector
  Funcon vector-elements
]

## Sets
[
  Type sets
  Funcon set
  Funcon set-elements
  Funcon is-in-set
  Funcon is-subset
  Funcon set-insert
  Funcon set-unite
  Funcon set-intersect
  Funcon set-difference
  Funcon set-size
  Funcon some-element
  Funcon element-not-in
]

## Maps
[
  Type maps
  Funcon map
  Funcon map-elements
  Funcon map-lookup      Alias lookup
  Funcon map-domain      Alias dom
  Funcon map-override
  Funcon map-unite
  Funcon map-delete
]

## Multisets (bags)
[
  Type multisets
  Funcon multiset
  Funcon multiset-elements
  Funcon multiset-occurrences
  Funcon multiset-insert
  Funcon multiset-delete
  Funcon is-submultiset
]

## Graphs
[
  Type directed-graphs
  Funcon is-cyclic
  Funcon topological-sort
]

## References and pointers
[
  Datatype references
  Funcon reference
  Type pointers
  Funcon dereference
]

## Records
[
  Datatype records
  Funcon record
  Funcon record-select
]

## Variants
[
  Datatype variants
  Funcon variant
  Funcon variant-id
  Funcon variant-value
]

// Further types of composite values to be added

## Abstraction values

### Generic abstractions
[
  Type abstractions
  Funcon abstraction
  Funcon closure
]

### Thunks
[
  Datatype thunks
  Funcon thunk
  Funcon force
]

## Functions
[
  Datatype functions
  Funcon function
  Funcon apply
  Funcon supply
  Funcon compose
  Funcon uncurry
  Funcon curry
  Funcon partial-apply
]

## Patterns
[
  Datatype patterns
  Funcon pattern
  Funcon match
  Funcon case
  Funcon pattern-any
  Funcon pattern-bind
  Funcon pattern-type
  Funcon pattern-prefer
  Funcon pattern-unite
  Funcon structural-match
  Funcon structural-pattern-prefer
  Funcon structural-pattern-unite
]

## Computations

## Types of computation
[
  Funcon computation-types
]

## Normal computation

### Flowing
[
  Funcon interleave
  Funcon atomic
  Funcon left-to-right      Alias l-to-r
  Funcon sequential      Alias seq
  Funcon effect
  Funcon choice
  Funcon if-then-else
  Funcon while
  Funcon do-while
]

## Giving
[
  Entity given-value
  Funcon give
  Funcon given
  Funcon no-given
  Funcon left-to-right-map
  Funcon interleave-map
  Funcon left-to-right-filter
  Funcon interleave-filter
  Funcon fold-left
  Funcon fold-right
]

## Binding
[
  Type environments      Alias envs
  Datatype identifiers      Alias ids
  Funcon identifier-tagged      Alias id-tagged
  Funcon fresh-identifier
  Entity environment      Alias env
  Funcon bind-value      Alias bind
  Funcon unbind
  Funcon bound-value      Alias bound
  Funcon bound-link
  Funcon closed
  Funcon scope
  Funcon accumulate
  Funcon bind-recursively
  Funcon recursive
]

### Generating
[
  Type atoms
  Entity used-atom-set
  Funcon fresh-atom
  Funcon use-atom-not-in
]

### Storing
[
  Datatype variables      Alias vars
  Funcon variable      Alias var
  Type stores
  Entity store
  Funcon clear-store
  Funcon fresh-variable      Alias fresh-var
  Funcon allocate-variable      Alias alloc-var
  Funcon de-allocate-variable      Alias de-alloc-var
  Funcon initialise-variable      Alias init
  Funcon allocate-initialised-variable
  Funcon assign-variable-value      Alias assign
  Funcon assigned-variable-value      Alias assigned
  Funcon current-value
  Funcon un-assign
  Funcon structural-allocate
  Funcon structural-assign
  Funcon structural-assigned
]

### Linking
[
  Datatype links
  Funcon link
  Funcon fresh-link
  Funcon fresh-initialised-link      Alias fresh-init-link
  Funcon set-link
  Funcon follow-if-link
]

### Interacting

#### Input
[
  Entity standard-in
  Funcon read
]

#### Output
[
  Entity standard-out
  Funcon print
]

## Abnormal computation

### Sticking
[
  Funcon stuck
]

### Failing
[
  Datatype signals
  Funcon signal
  Entity failure
  Funcon fail
  Funcon else
  Funcon check-true      Alias def
  Funcon definitely
]

## Throwing
[
  Entity thrown
  Funcon throw
  Funcon handle-thrown
  Funcon handle-recursively
  Funcon finally
]

## Continuing
[
  Entity control-signal
  Entity resume-signal
  Funcon control
  Funcon prompt
  Funcon hole
  Funcon plug
]

// Concurrent computation to be added
```

Granularity

Funcons are *individual* programming constructs

- ▶ *not deltas*
- ▶ *not language features*
- ▶ *not language extensions*

Funcons can be *freely* combined

- ▶ *independent, unordered, no constraints*

CBS: component-based
semantics

Component-based semantics

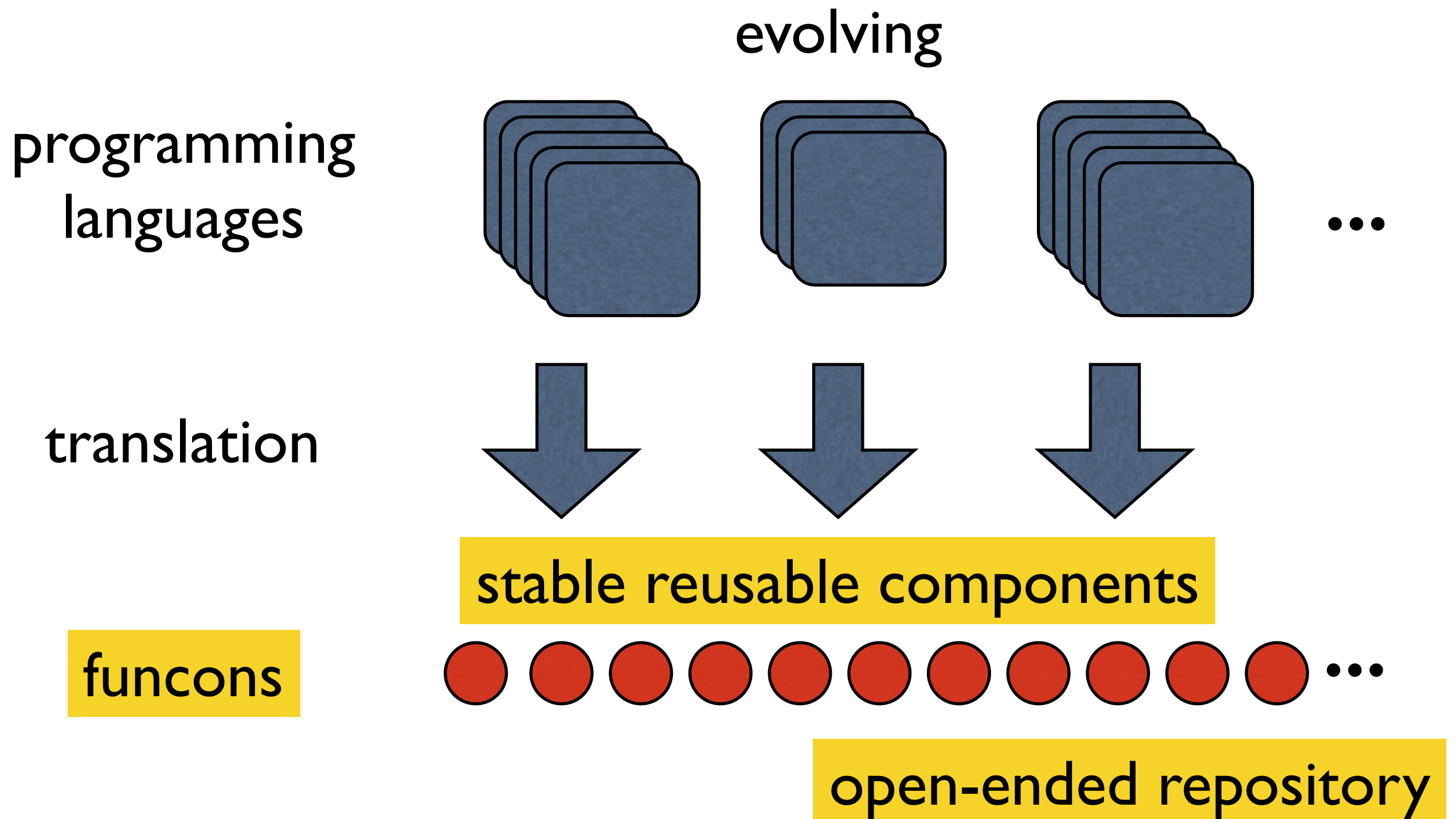
Language semantics by reduction

- *translation*: language constructs \rightarrow funcon terms, ***hence***:
- *derivation*: funcon semantics \rightarrow language semantics

Reusable components of language specifications

- *funcon specifications*

Component-based semantics



Conjecture

Using a ***component-based*** semantic meta-language can significantly reduce the effort of language specification

Language specification in CBS

Syntax

$E : \text{exp} ::= \dots \mid \text{'let' id '=' exp 'in' exp} \mid \dots$

Semantics

$\text{eval}[[_ : \text{exp}]] : \Rightarrow \text{values}$

Rule

$\text{eval}[[\text{'let' } I \text{'=' } E1 \text{'in' } E2]] =$
 $\text{scope} (\text{bind} (I, \text{eval}[[E1]]), \text{eval}[[E2]])$

Language specification in CBS

Syntax

$S : \text{stm} ::= \dots \mid \text{'while' '(' exp ')'} \text{stm} \mid \dots$

Semantics

$\text{exec}[[_ : \text{stm}]] : \Rightarrow \text{no-value}$

Rule

$\text{exec}[[\text{'while' '(' } E \text{ ')'} S]] =$
 while-true (
 $\text{eval}[[E]],$
 $\text{exec}[[S]] \text{)}$

Language specification in CBS

co-evolution

Syntax

$S : \text{stm} ::= \dots \mid \text{'while' '(' exp ')'} \text{stm} \mid \dots$

Semantics

$\text{exec}[[_ : \text{stm}]] : \Rightarrow \text{no-value}$

Rule

$\text{exec}[[\text{'while' '(' } E \text{ ')'} S]] =$
 while-true (
 $\text{not is-eq (eval}[[E]], \emptyset \text{),}$
 $\text{exec}[[S]] \text{)}$

Language specification in CBS

co-evolution

Syntax

$S : \text{stm} ::= \dots \mid \text{'while' ' (' exp ')' stm} \mid \text{'break' } \mid \dots$

Semantics

$\text{exec}[[_ : \text{stm}]]: \Rightarrow \text{no-value}$

Rule

$\text{exec}[[\text{'while' ' (' E ')' S }]] =$
 $\text{handle-break while-true (}$
 $\quad \text{not is-eq (} \emptyset, \text{eval}[[E]] \text{),}$
 $\quad \text{exec}[[S]] \text{)}$

Rule

$\text{exec}[[\text{'break' }]]= \text{break}$

Version control?

Funcons never change!

- ▶ *no versioning*
- ▶ *optimal reusability*

Languages evolve

- ▶ *ordinary version control*
- ▶ *no support for language reuse/extension/composition*

Tool support

CBS tool support

implemented in SPOOFAX (v2.4)

CBS-Editor

- ▶ for developing language and funcon specifications
- ▶ CBS parser (in SDF3)
- ▶ CBS name resolution, arity-checking (in NABL2)
- ▶ HTML generation (in STRATEGO, CSS)
- ▶ language-editor generation (in STRATEGO)

CBS tool support

main features

Generated Language-Editors

- program parsing and translation to funcons

Integrated external **HASKELL** tools

- generation of funcon interpreters

Internal **DYNSEM** tools

- generation of funcon interpreters

Demo

Demo of current CBS tool

Browsing/editing CBS specifications

- languages and funcons

Translating programs to funcons

- using generated STRATEGO code

‘Running’ programs by interpreting funcons

- using generated HASKELL or specified DYNSEM

Demo: language specification

The screenshot shows the CBS-Editor IDE with a project tree on the left and a code editor on the right. The project tree includes folders for Computations, Funcons-Index, Values, and SL-Tests. The code editor displays the 'SL-Start.cbs' file, which contains the following text:

```
Language "SL"

/*
The SimpleLanguage, abbreviated "SL", is a dynamic demonstration language.
It was built using Truffle for the GraalVM at Oracle Labs
[https://github.com/graalvm/simplelanguage].

A DynSem specification of SL in Spoofax has been given by Vlad Vergu
[https://github.com/MetaBorgCube/metaborg-sl/].

This CBS specification of SL has been prototyped using a DynSem specification
of the required funcons. The CBS grammar of SL is annotated with the SDF3
productions used for the DynSem of SL.

The SL programs used to test the DynSem of SL are in SL-Tests/examples,
together with their expected output and the funcon terms produced by the CBS
of SL. Running the funcon terms using the DynSem of the funcons gives the
same results as running the SL programs using the DynSem of SL, except for
programs involving the following built-in SL functions, which have not yet
been implemented:
- defineFunction (the argument is a string in SL, requiring parsing)
- nanoTime
- stackTrace
Moreover, some of the literal numbers in the examples have been reduced
(mostly by a factor of 10) to avoid potential stack overflow.

Acknowledgement: Vlad Vergu provided helpful advice regarding SL and DynSem.
*/

[
#1 Function definitions
#2 Expressions
#3 Statements
#A Disambiguation
]

Syntax
Program : program
      ::= fun-def*
// Program.Program = [[{FunDef "\n"}*]]

Semantics
```

The status bar at the bottom indicates 'Writable', 'Insert', and '33 : 1'.

Demo: funcon reference

The screenshot displays the CBS-Editor interface. On the left, a project tree shows the structure of the 'Funcons-beta' project, including folders for 'Computations', 'Normal', 'Funcons-Index', 'Values', and 'SL-Tests'. The 'SL' folder is expanded, showing files like 'SL-1-Function-Definitions.cbs', 'SL-2-Expressions.cbs', 'SL-3-Statements.cbs', 'SL-Disambiguation.cbs', 'SL-Funcons.cbs', 'SL-Funcons-Index.cbs', and 'SL-Start.cbs'. The main editor window is open to 'SL-3-Statements.cbs', showing the definition of the 'SL' language. The code is organized into sections: 'Language "SL"', '#3 Statements', 'Syntax', 'Rule', 'Semantics', and 'Rule'. The 'Syntax' section defines the grammar for statements, including expressions, blocks, while loops, if-then-else statements, return statements, break statements, and continue statements. The 'Rule' section defines the semantics for these statements, including the execution of blocks, while loops, if-then-else statements, return statements, break statements, and continue statements. The 'Semantics' section defines the execution of statements, including the execution of blocks, while loops, if-then-else statements, return statements, break statements, and continue statements. The 'Rule' section defines the execution of statements, including the execution of blocks, while loops, if-then-else statements, return statements, break statements, and continue statements.

```
Language "SL"

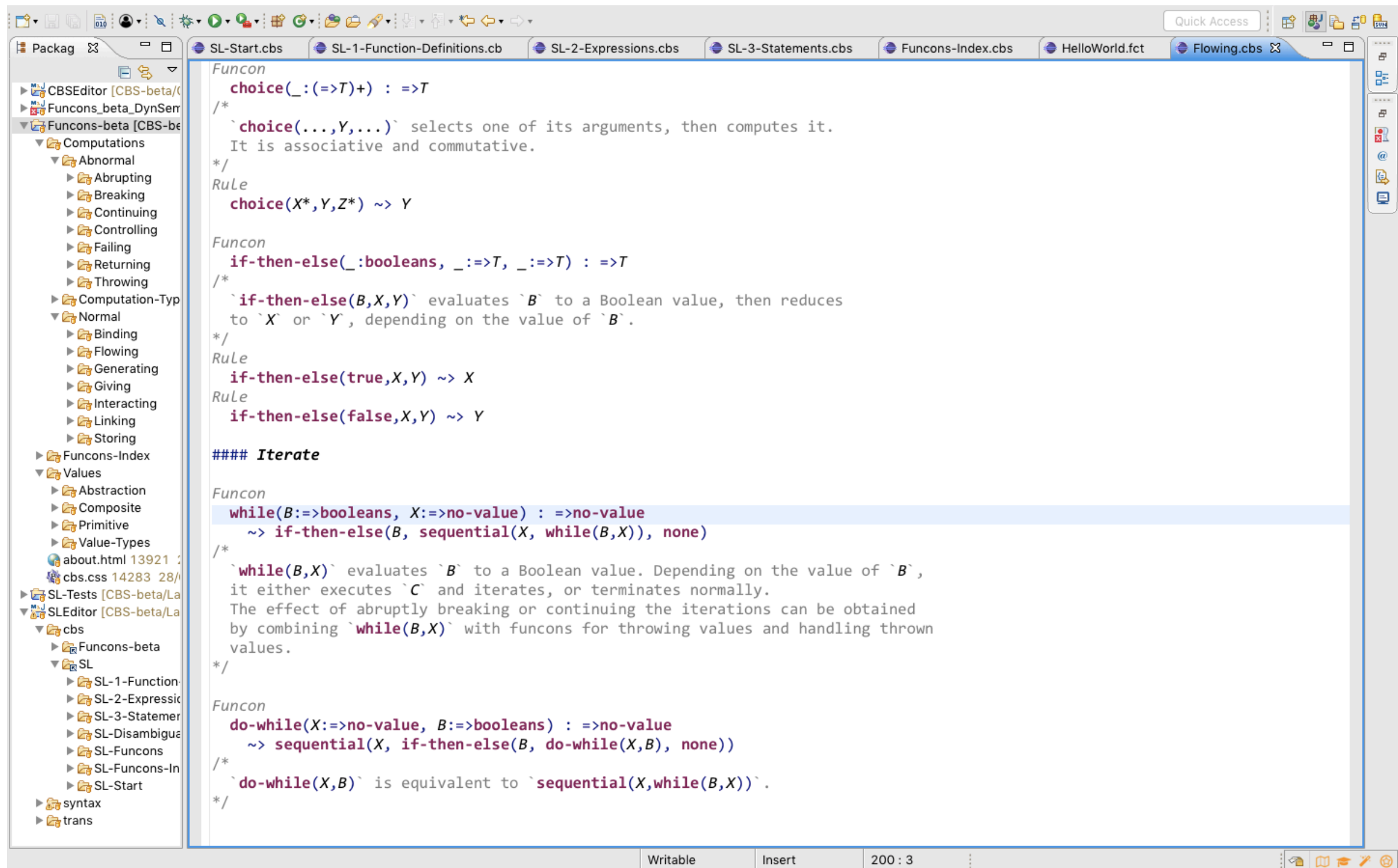
#3 Statements

Syntax
  Stmt : stmt
  ::= expr ';' // Stmt.Stmt = [[Expr]];
  | block // Stmt = Block
  | 'while' '(' expr ')' block // Stmt.While = [while([Expr]) [Block]]
  | 'if' '(' expr ')' block // Stmt.IfNoElse = [if ([Expr]) [Block]]
  | 'if' '(' expr ')' block 'else' block // Stmt.IfThenElse = [if ([Expr]) [Block]
  // else [Block]]
  | 'return' expr ';' // Stmt.Return = [return [Expr]];
  | 'return' ';' // Stmt.ReturnVoid = [return;]
  | 'break' ';' // Stmt.Break = [break;]
  | 'continue' ';' // Stmt.Continue = [continue;]

Rule
  [[ 'if' '(' Expr ')' Block ]] : stmt =
  [[ 'if' '(' Expr ')' Block 'else' '{' '}' ]]

Semantics
  exec[[ _:stmt ]] : => no-value
Rule
  exec[[ Expr ';' ]] = effect(eval[[Expr]])
/*
  exec[[ Block ]] defined below
*/
Rule
  exec[[ 'while' '(' Expr ')' Block ]] =
  handle-break(while(eval[[Expr]], handle-continue(exec[[Block]])))
Rule
  exec[[ 'if' '(' Expr ')' Block1 'else' Block2 ]] =
  if-then-else(eval[[Expr]], exec[[Block1]], exec[[Block2]])
Rule
  exec[[ 'return' Expr ';' ]] = return(eval[[Expr]])
Rule
  exec[[ 'return' ';' ]] = return(none)
Rule
  exec[[ 'break' ';' ]] = break
Rule
```

Demo: funcon specification



The screenshot shows the CBS-Editor IDE with a project tree on the left and a code editor on the right. The project tree includes folders for Computations, Normal, Funcons-Index, Values, and SL-Tests. The code editor displays the Funcon specification for 'choice' and 'while'.

```
Funcon
choice(_:(=>T)+) : =>T
/*
`choice(...,Y,...)` selects one of its arguments, then computes it.
It is associative and commutative.
*/
Rule
choice(X*,Y,Z*) ~> Y

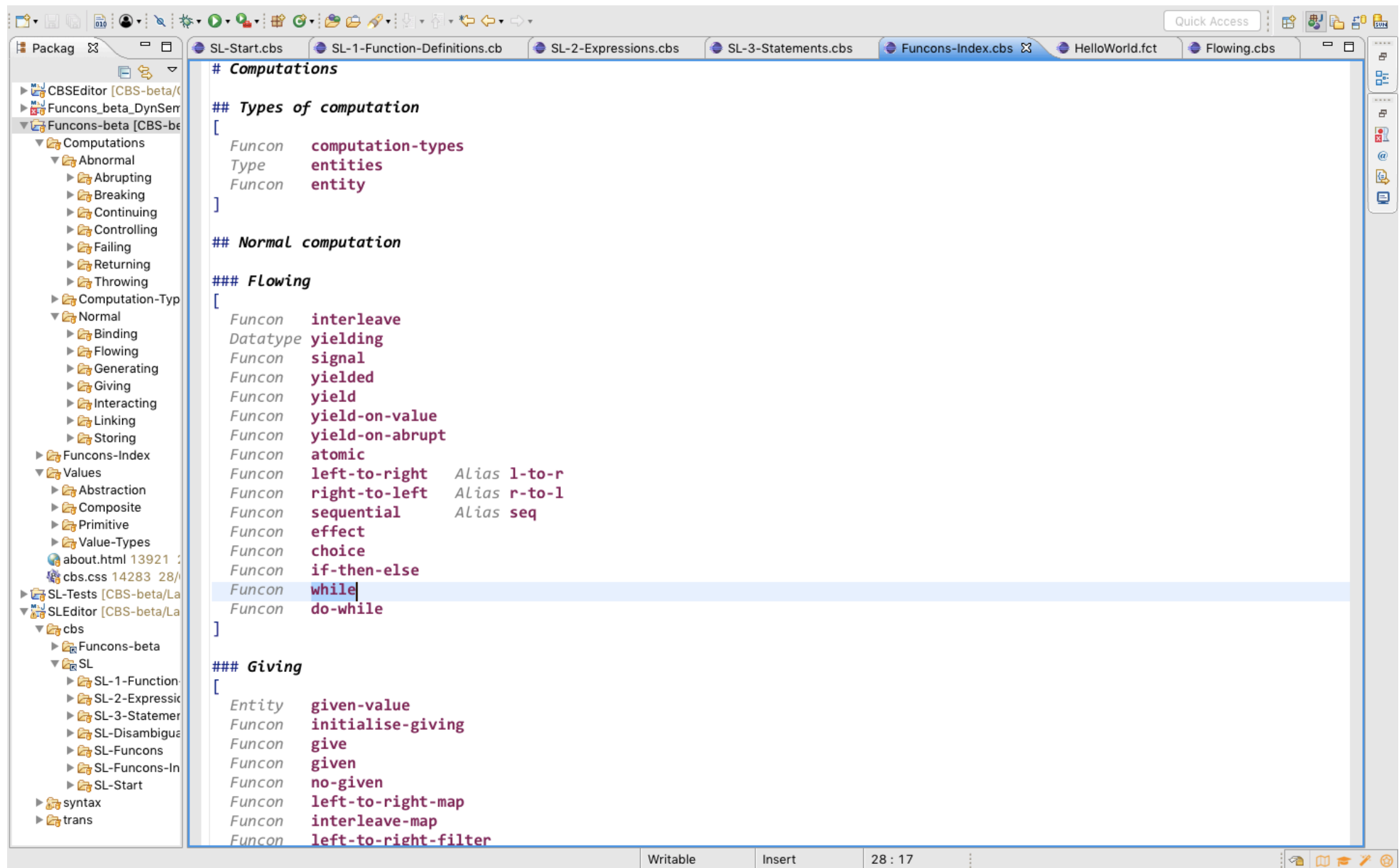
Funcon
if-then-else(_:booleans, _:=>T, _:=>T) : =>T
/*
`if-then-else(B,X,Y)` evaluates `B` to a Boolean value, then reduces
to `X` or `Y`, depending on the value of `B`.
*/
Rule
if-then-else(true,X,Y) ~> X
Rule
if-then-else(false,X,Y) ~> Y

#### Iterate

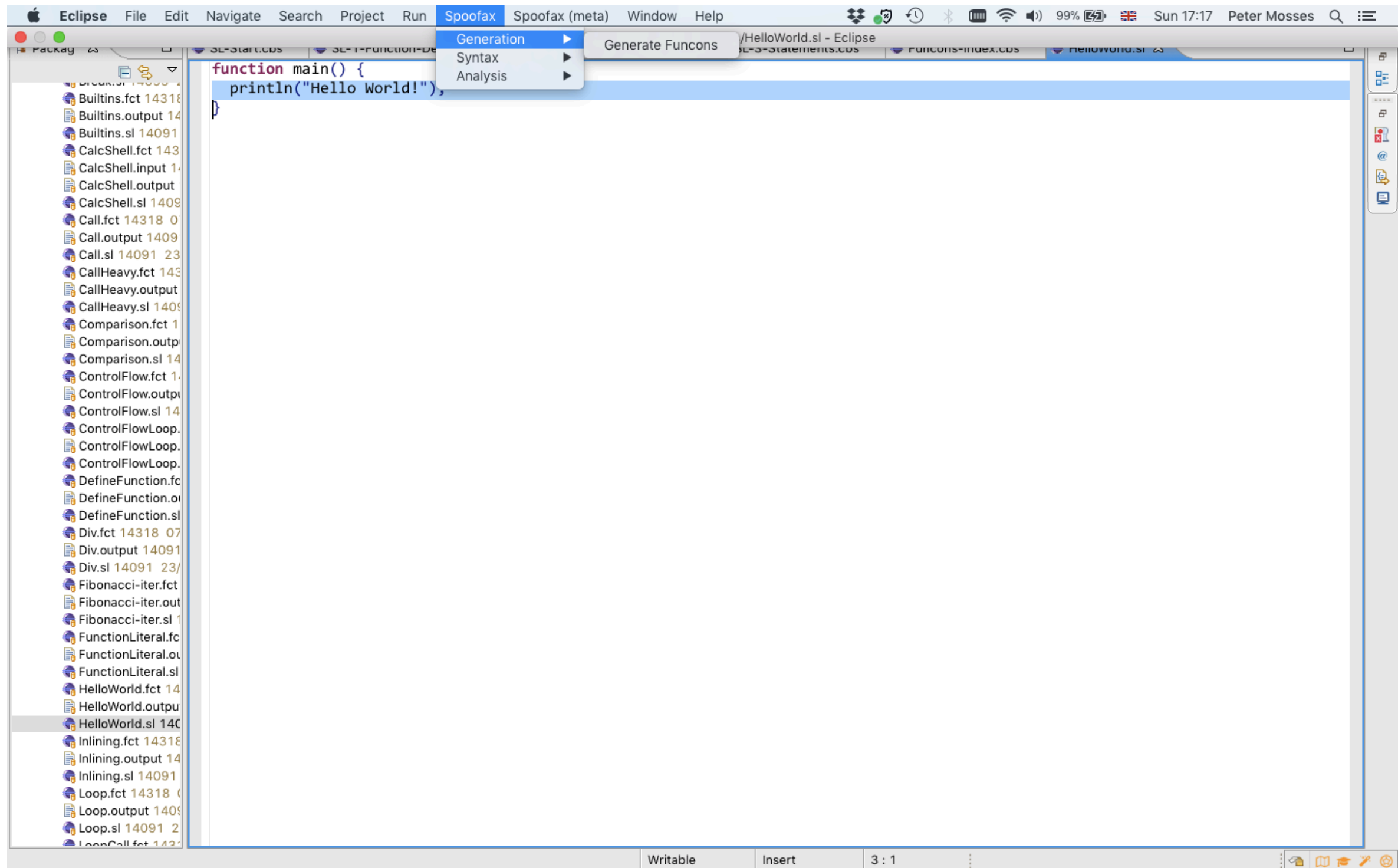
Funcon
while(B:=>booleans, X:=>no-value) : =>no-value
~> if-then-else(B, sequential(X, while(B,X)), none)
/*
`while(B,X)` evaluates `B` to a Boolean value. Depending on the value of `B`,
it either executes `C` and iterates, or terminates normally.
The effect of abruptly breaking or continuing the iterations can be obtained
by combining `while(B,X)` with funcons for throwing values and handling thrown
values.
*/

Funcon
do-while(X:=>no-value, B:=>booleans) : =>no-value
~> sequential(X, if-then-else(B, do-while(X,B), none))
/*
`do-while(X,B)` is equivalent to `sequential(X,while(B,X))`.
*/
```

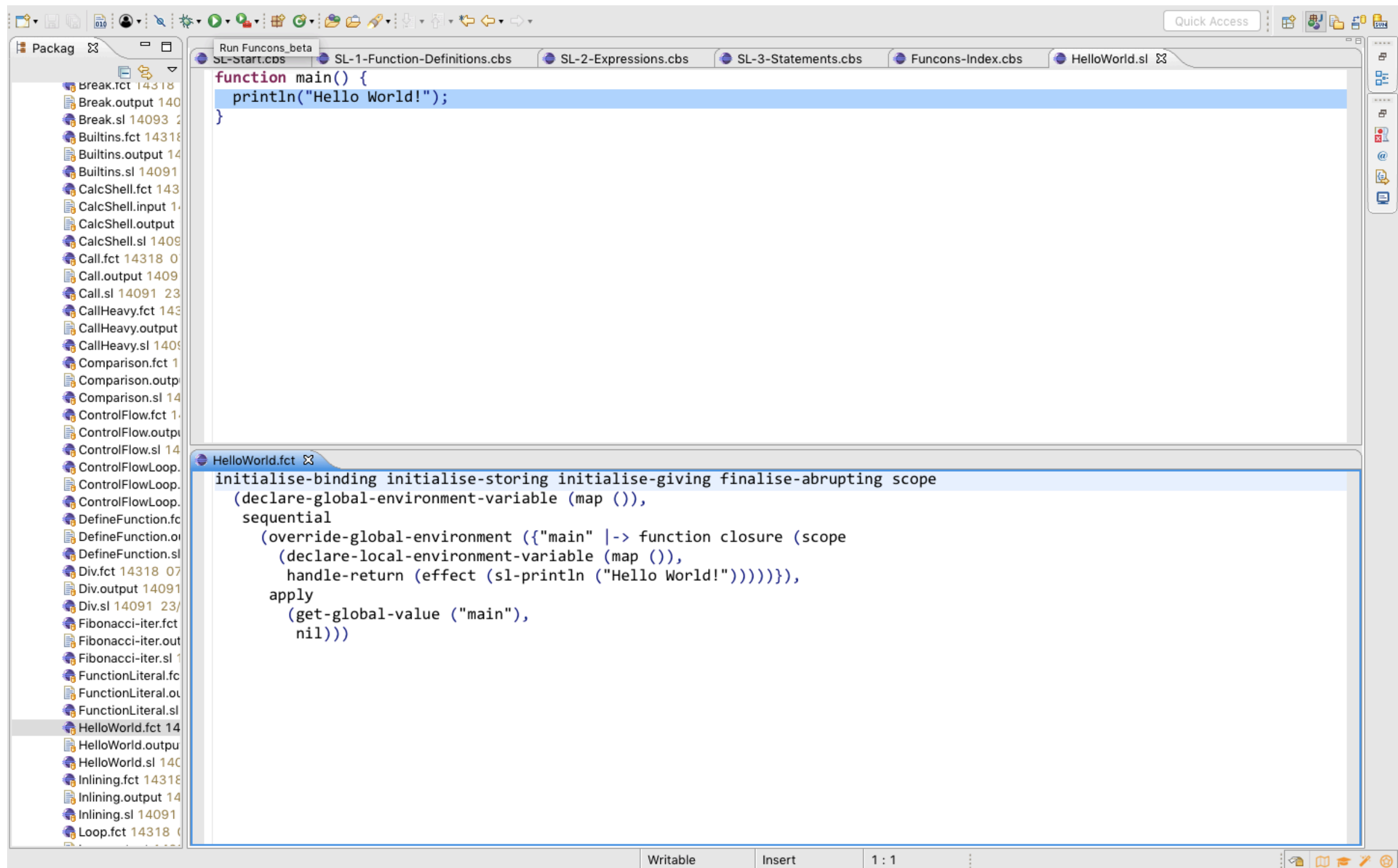
Demo: funcon index



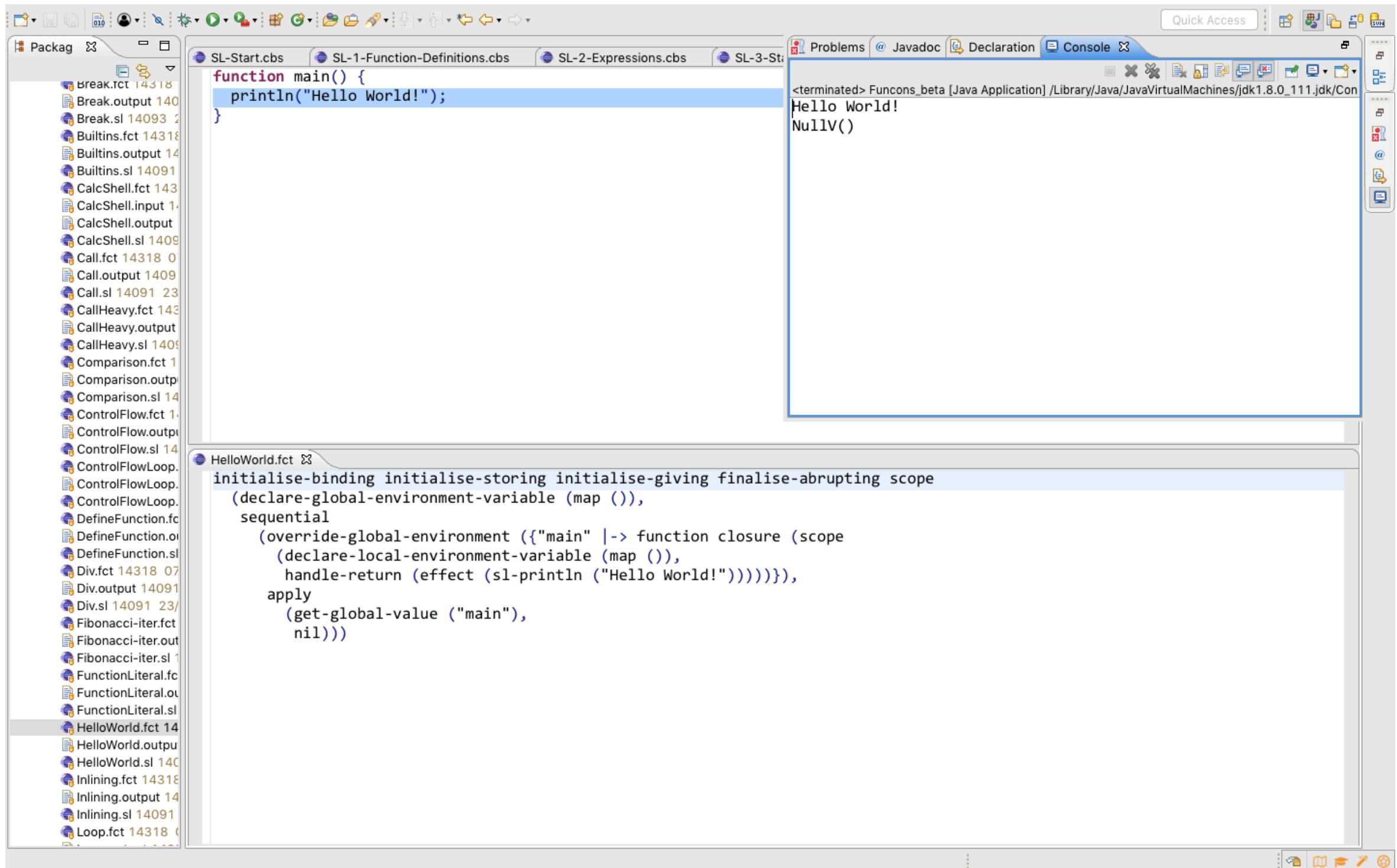
Demo: translation to funcons



Demo: running funcons



Demo: resulting behaviour



Demo: (re)generating

Eclipse File Edit Navigate Search Project Run Spoofax Spoofax (meta) Window Help

Generate
Syntax
Analysis

Language editor
HTML files
Funcon reuse index
All the above
Developer

Language "SL"

#3 Statements

Syntax

```
Stmt : stmt
 ::= expr ';'
    | block
    | 'while' '(' expr ')' block
    | 'if' '(' expr ')' block
    | 'if' '(' expr ')' block 'else' block
    | 'return' expr ';'
    | 'return' ';'
    | 'break' ';'
    | 'continue' ';'

// Stmt.Stmt = [[Expr]];
// Stmt = Block
// Stmt.While = [while([Expr]) [Block]]
// Stmt.IfNoElse = [if ([Expr]) [Block]]
// Stmt.IfThenElse = [if ([Expr]) [Block]
//                      else [Block]]
// Stmt.Return = [return [Expr]];
// Stmt.ReturnVoid = [return;]
// Stmt.Break = [break;]
// Stmt.Continue = [continue;]
```

Rule

```
[[ 'if' '(' Expr ')' Block ]] : stmt =
[[ 'if' '(' Expr ')' Block 'else' '{' '}' ]]
```

Semantics

```
exec[[ _:stmt ]] : => no-value
```

Rule

```
exec[[ Expr ';' ]] = effect(eval[[Expr]])
```

/*
exec[[Block]] defined below
*/

Rule

```
exec[[ 'while' '(' Expr ')' Block ]] =
  handle-break(while(eval[[Expr]], handle-continue(exec[[Block]])))
```

Rule

```
exec[[ 'if' '(' Expr ')' Block1 'else' Block2 ]] =
  if-then-else(eval[[Expr]], exec[[Block1]], exec[[Block2]])
```

Rule

```
exec[[ 'return' Expr ';' ]] = return(eval[[Expr]])
```

Rule

```
exec[[ 'return' ';' ]] = return(none)
```

Rule

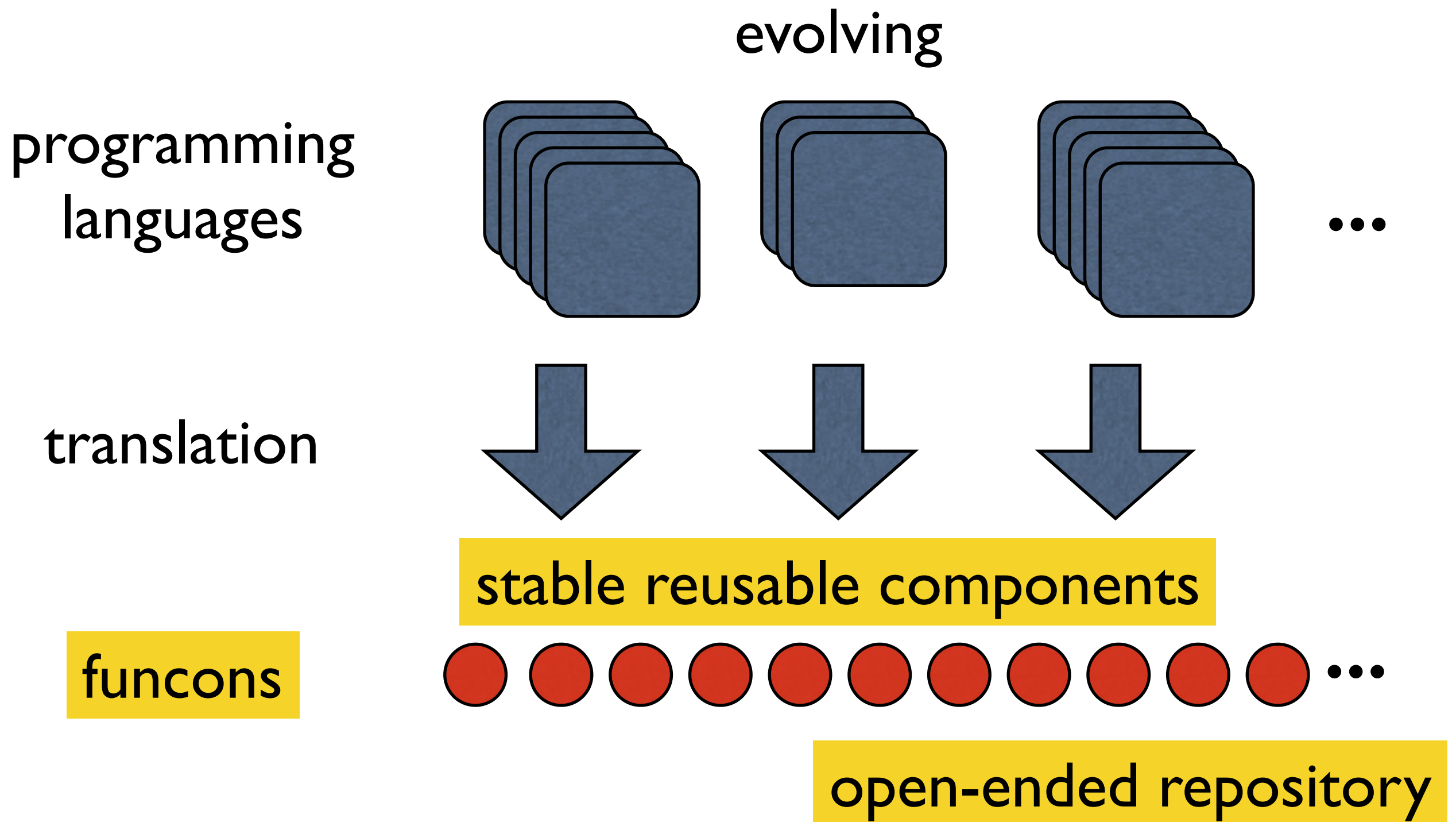
```
exec[[ 'break' ';' ]] = break
```

Rule

Writable Insert 31 : 20

Conclusion

Language specification in CBS



Conjecture

Using a ***component-based*** semantic meta-language can significantly reduce the effort of language specification

Proving the conjecture

Case studies

- ▶ Various small example languages (IMP, SIMPLE, SL, MJ, ...)
- ▶ CAML LIGHT
 - *Reusable components of semantic specifications*
in *Trans. AOSD XII*, Springer, 2015
- ▶ C# (v1.2)
 - ongoing...

Funcons beta-release

(imminent...🙄)

2018(Q1): **Funcons-beta** to be made available **for review**

- ▶ *some details may change !*
- ▶ preliminary tool support, minimal documentation

2018(Q3/Q4): **Funcons** to be **released for reuse**

- ▶ *details will **not** be allowed to change !!*
- ▶ usable tool support, user-level documentation

PLANCOMPS

www.plancomps.org

“Programming Language Components and Specifications”

Funded project 2011–16: 
Engineering and Physical Sciences
Research Council

- ▶ at Swansea, Royal Holloway (RHUL), City, Newcastle

Current participants:

- ▶ A. Johnstone, E.A. Scott, L.T. van Binsbergen (RHUL)
N. Sculthorpe (NTU), C. Bach Poulsen, PDM (Delft)

New participants are welcome !

Current and future work

- ▶ ***modular static semantics*** for funcons
 - *modular* type soundness proofs?
- ▶ improved ***tool support***
- ▶ funcons for ***threads and concurrency***
- ▶ completing a ***major case study: C#***

Appendix

Funcon specification in CBS

Normal computation: **flowing**

Funcon

```
while-true ( _: =>booleans, _: =>no-value ) : =>no-value
```

Rule

```
while-true ( X, Y )  
  ~> if-then-else ( X, seq ( Y, while ( X, Y ) ), none)
```

Funcon specification in CBS

Normal computation: **flowing**

Funcon

if-then-else ($_ : \text{bools}$, $_ : \Rightarrow T$, $_ : \Rightarrow T$) : $\Rightarrow T$

Rule

if-then-else (**true**, X , Y) $\sim\>$ X

Rule

if-then-else (**false**, X , Y) $\sim\>$ Y

Funcon specification in CBS

Normal computation: **flowing**

Funcon

seq ($_$: no-value, $_$: $\Rightarrow T$) : $\Rightarrow T$

Rule

seq (none, Y) $\sim\>$ Y

Funcon specification in CBS

Normal computation: **binding**

Funcon

scope($_ : \text{envs}$, $_ : \Rightarrow T$) : $\Rightarrow T$

Rule

$\text{env}(\text{map-override}(Rho1, Rho0)) \mid - X \dashrightarrow X'$

 $\text{env}(Rho0) \mid - \text{scope} (Rho1 : \text{envs}, X) \dashrightarrow \text{scope} (Rho1, X')$

Rule

scope ($_ : \text{envs}$, $V : T$) $\sim \triangleright V$

References

- PDM: Modular structural operational semantics. *J.LAP*, 2004.
<https://doi.org/10.1016/j.jlap.2004.03.008>
- PDM, M.J. New: Implicit propagation in structural operational semantics. In *SOS'08*, 2009.
<https://doi.org/10.1016/j.entcs.2009.07.073>
- *PLanCompS project home page*.
<http://www.plancomps.org>
- *Spoofax*. <http://spoofax.readthedocs.io/en/latest/>
- M. Churchill, PDM: Modular bisimulation for computations and values. In *FoSSaCS'13*.
https://doi.org/10.1007/978-3-642-37075-5_7
- F. Vesely, PDM. Funkons – Component-based semantics in K. In *WRLA'14*.
https://doi.org/10.1007/978-3-319-12904-4_12
- M. Churchill, PDM, N. Sculthorpe, P. Torrini: Reusable components of semantic specifications. *Trans. AOSD XII*, 2015.
https://doi.org/10.1007/978-3-662-46734-3_4
- V.A. Vergu, P. Neron, E. Visser: DynSem – A DSL for dynamic semantics specification. In *RTA'15*.
<https://doi.org/10.4230/LIPIcs.RTA.2015.365>
- L.T.v. Binsbergen, N. Sculthorpe, PDM: Tool support for component-based semantics. In *Modularity'16 Demos*.
<http://doi.acm.org/10.1145/2892664.2893464>