

# Unstable-Languages-beta: LD-Start \*

The PPlanCompS Project

LD-Start.cbs | PLAIN | PRETTY

## OUTLINE

- 1 Lexical constructs**
  - 2 Call-by-value lambda-calculus**
  - 3 Arithmetic and Boolean expressions**
  - 4 References and imperatives**
  - 5 Multithreading**
  - 6 Programs**
- 

*Language* "LD"

- [ **1** Lexical constructs
- 2** Call-by-value lambda-calculus
- 3** Arithmetic and Boolean expressions
- 4** References and imperatives
- 5** Multithreading
- 6** Programs
- A** Disambiguation ]

Lexical syntax:

```
Lexis  X : id ::= ('a'-'z') ('a'-'z' | '0'-'9')*
        N : int ::= ('0'-'9')+
        keyword ::= 'do' | 'else' | 'fork' | 'if'
                  | 'in' | 'join' | 'lambda' | 'let'
                  | 'ref' | 'spawn' | 'then' | 'while'
```

Context-free syntax:

---

\*Suggestions for improvement: [plancomps@gmail.com](mailto:plancomps@gmail.com).  
Reports of issues: <https://github.com/plancomps/CBS-beta/issues>.

*Syntax*  $E : \text{exp} ::=$

- $\text{int}$
- $\text{id}$
- $\text{'lambda' id '.' exp}$
- $\text{exp exp}$
- $\text{'let' id '=' exp 'in' exp}$
- $\text{'(' exp ')}$
- $\text{exp '+' exp}$
- $\text{exp '*' exp}$
- $\text{exp '/' exp}$
- $\text{exp '<=' exp}$
- $\text{exp '&\&' exp}$
- $\text{'if' exp 'then' exp 'else' exp}$
- $\text{'ref' exp}$
- $\text{exp ':=' exp}$
- $\text{'!' exp}$
- $\text{exp ';' exp}$
- $\text{'(' ')}$
- $\text{'while' exp 'do' exp}$
- $\text{'spawn' exp}$
- $\text{'join' exp}$

Expression evaluation:

*Type*  $\text{Id-values}$

- $\rightsquigarrow \text{functions}(\text{values}, \text{values})$
- $\text{integers}$
- $\text{booleans}$
- $\text{variables}$
- $\text{null-type}$
- $\text{thread-ids}$

*Semantics*  $\text{eval}[\![ \_ : \text{exp} ]\!] : \Rightarrow \text{Id-values}$

## 1 Lexical constructs

*Rule*  $\text{eval}[\![ N ]\!] = \text{decimal } "N"$

*Rule*  $\text{eval}[\![ X ]\!] = \text{bound } "X"$

## 2 Call-by-value lambda-calculus

Rule  $\text{eval}[\![ \text{'lambda' } X \text{'.' } E ]\!] =$   
function closure  
scope(  
bind("X", given),  
 $\text{eval}[\![ E ]\!]$ )

Rule  $\text{eval}[\![ E_1 E_2 ]\!] =$   
apply( $\text{eval}[\![ E_1 ]\!]$ ,  $\text{eval}[\![ E_2 ]\!]$ )

Rule  $\text{eval}[\![ \text{'let' } X \text{'=' } E_1 \text{'in' } E_2 ]\!] =$   
scope(  
bind("X",  $\text{eval}[\![ E_1 ]\!]$ ),  
 $\text{eval}[\![ E_2 ]\!]$ )

Desugaring (alternative to the above rule):

Rule  $\llbracket \text{'let' } X \text{'=' } E_1 \text{'in' } E_2 \rrbracket : \text{exp} =$   
 $\llbracket \text{'(' } \text{'lambda' } X \text{'.' } E_2 \text{' )' ' ( ' } E_1 \text{' )' } \rrbracket$

Rule  $\text{eval}[\![ \text{'(' } E \text{' )' } ]\!] = \text{eval}[\![ E ]\!]$

## 3 Arithmetic and Boolean expressions

Rule  $\text{eval}[\![ E_1 \text{'+' } E_2 ]\!] =$   
int-add( $\text{eval}[\![ E_1 ]\!]$ ,  $\text{eval}[\![ E_2 ]\!]$ )

Rule  $\text{eval}[\![ E_1 \text{'*'} E_2 ]\!] =$   
int-mul( $\text{eval}[\![ E_1 ]\!]$ ,  $\text{eval}[\![ E_2 ]\!]$ )

Rule  $\text{eval}[\![ E_1 \text{'/' } E_2 ]\!] =$   
checked int-div( $\text{eval}[\![ E_1 ]\!]$ ,  $\text{eval}[\![ E_2 ]\!]$ )

Rule  $\text{eval}[\![ E_1 \text{'<=' } E_2 ]\!] =$   
is-less-or-equal l-to-r( $\text{eval}[\![ E_1 ]\!]$ ,  $\text{eval}[\![ E_2 ]\!]$ )

Rule  $\text{eval}[\![ E_1 \text{'\&\&'} E_2 ]\!] =$   
if-true-else( $\text{eval}[\![ E_1 ]\!]$ ,  $\text{eval}[\![ E_2 ]\!]$ , false)

Rule  $\text{eval}[\![ \text{'if' } E_1 \text{'then' } E_2 \text{'else' } E_3 ]\!] =$   
if-true-else( $\text{eval}[\![ E_1 ]\!]$ ,  $\text{eval}[\![ E_2 ]\!]$ ,  $\text{eval}[\![ E_3 ]\!]$ )

## 4 References and imperatives

Rule  $\text{eval}[\![ \text{'ref' } E ]\!] =$   
allocate-initialised-variable(lid-values,  $\text{eval}[\![ E ]\!]$ )

Rule  $\text{eval}[\![ E_1 \text{' := ' } E_2 ]\!] =$   
assign( $\text{eval}[\![ E_1 ]\!]$ ,  $\text{eval}[\![ E_2 ]\!]$ )

Rule  $\text{eval}[\![ \text{'!' } E ]\!] = \text{assigned}(\text{eval}[\![ E ]\!])$

Rule  $\text{eval}[\![ E_1 \text{' ; ' } E_2 ]\!] =$   
sequential(effect( $\text{eval}[\![ E_1 ]\!]$ ),  $\text{eval}[\![ E_2 ]\!]$ )

Rule  $\text{eval}[\![ \text{'(' } \text{' )' } ]\!] = \text{null-value}$

Rule  $\text{eval}[\![ \text{'while' } E_1 \text{'do' } E_2 ]\!] =$   
while-true( $\text{eval}[\![ E_1 ]\!]$ , effect( $\text{eval}[\![ E_2 ]\!]$ ))

## 5 Multithreading

N.B. The funcons for multithreading have not yet been fully validated, so they are defined in Unstable-Funcons-beta instead of Funcons-beta.

*Rule*  $\text{eval}[\text{'spawn'}\ E] =$   
     $\text{thread-activate thread-joinable thunk closure eval}[E]$   
*Rule*  $\text{eval}[\text{'join'}\ E] = \text{thread-join}(\text{eval}[E])$

## 6 Programs

*Syntax*  $START : \text{start} ::= \text{exp}$

*Semantics*  $\text{start}[_ : \text{start}] : \Rightarrow \text{values}$

*Rule*  $\text{start}[E] =$   
     $\text{initialise-binding}$   
     $\text{initialise-storing}$   
     $\text{finalise-failing}$   
     $\text{multithread}$   
     $\text{eval}[E]$