



CSCI-344

# Programming Language Concepts (Section 3)

---

Lecture 28

Reduction Semantics for Control Structures

Instructor: Hossein Hojjat

November 18, 2016

## Done:

- Control Operators
  - (break)
  - (continue)
  - (return *exp*)
  - (try-catch *exp exp*)
  - (throw *exp*)

## This session:

- Reduction Semantics for Control Structures

- Consider a very simple language for addition

$$\begin{array}{l} \text{Exp} = \text{LIT} \quad (\text{Value}) \\ | \quad \text{VAR} \quad (\text{Name}) \\ | \quad \text{ADD} \quad (\text{Exp}, \text{Exp}) \end{array}$$

# Operational Semantics

- Judgment  $\langle e, \rho \rangle \Downarrow n$  means  $e$  evaluates to  $n$  in environment  $\rho$
- Judgment specifies the entire transition from a configuration to a final value
- $\text{ADD}(e_1, e_2)$ : is  $e_1$  evaluated first or  $e_2$ ?

$$\overline{\langle \text{LIT}(n), \rho \rangle \Downarrow n}$$

$$\overline{\langle \text{VAR}(x), \rho \rangle \Downarrow \rho(x)}$$

$$\frac{\langle e_1, \rho \rangle \Downarrow n_1 \quad \langle e_2, \rho \rangle \Downarrow n_2}{\langle \text{ADD}(e_1, e_2), \rho \rangle \Downarrow n_1 + n_2}$$

# Big-step vs. small-step Operational Semantics

- Big-step operational semantics: transition encodes all computation steps

$$\langle e, \rho \rangle \Downarrow n$$

- Small-step operational semantics: transition encodes only one step of computation

$$\langle e, \rho \rangle \rightarrow \langle e', \rho \rangle$$

Evaluation of an expression  $e$  in environment  $\rho$ :

- Big-step:  $\langle e, \rho \rangle \Downarrow n$
- Small-step:  $\langle e, \rho \rangle \rightarrow \langle e', \rho \rangle \rightarrow \langle e'', \rho \rangle \rightarrow \dots \rightarrow n$

Big-step semantics corresponds to the transitive closure of small steps

# Small-step Operational Semantics

$$\overline{\langle \text{LIT}(n), \rho \rangle \rightarrow n}$$

$$\overline{\langle \text{VAR}(x), \rho \rangle \rightarrow \rho(x)}$$

$$\frac{\langle e_1, \rho \rangle \rightarrow \langle e'_1, \rho \rangle}{\overline{\langle \text{ADD}(e_1, e_2), \rho \rangle \rightarrow \langle \text{ADD}(e'_1, e_2), \rho \rangle}}$$

$$\frac{\langle e_2, \rho \rangle \rightarrow \langle e'_2, \rho \rangle}{\overline{\langle \text{ADD}(\text{LIT}(n_1), e_2), \rho \rangle \rightarrow \langle \text{ADD}(\text{LIT}(n_1), e'_2), \rho \rangle}}$$

$$\overline{\langle \text{ADD}(\text{LIT}(n_1), \text{LIT}(n_2)), \rho \rangle \rightarrow n_1 + n_2}$$

- Fixed evaluation order
- Example:  $\text{ADD}(\underbrace{\text{ADD}(\text{LIT}(2), \text{LIT}(5))}_{\text{first}}, \underbrace{\text{ADD}(\text{LIT}(6), \text{LIT}(3))}_{\text{second}})$

- Operational Semantics: how to compute a program on some abstract machine
- How does the abstract machine compute a step in small-step semantics?
- To evaluate  $\text{ADD}(e_1, e_2)$  a small-step machine needs to:
  1. Evaluate  $e_1$ ,
  2. Remember to return and evaluate  $e_2$ ,
  3. Finally compute the result of  $\text{ADD}$ .

- Small-step machine maintains a “TO-DO” stack of contexts (continuations)
- Context is an expression with a **hole** in the place of a sub-expression
- Example: to evaluate

$$\text{ADD}(\text{ADD}(\text{LIT}(1), \text{LIT}(2)), \text{LIT}(3))$$

machine needs to compute

$$\text{ADD}(\text{LIT}(1), \text{LIT}(2))$$

in the context

$$\text{ADD}(\bullet, \text{LIT}(3))$$



## Example

Expression	Context Stack
ADD (ADD(LIT(1), LIT(3)), LIT(5))	[ ]

# Example

Expression

ADD (ADD(LIT(1), LIT(3)), LIT(5))

ADD(LIT(1), LIT(3))

Context Stack

[ ]

ADD(●, LIT(5)) :: [ ]

## Example

Expression

ADD (ADD(LIT(1), LIT(3)), LIT(5))

ADD(LIT(1), LIT(3))

LIT(1)

Context Stack

[ ]

ADD(●, LIT(5)) :: [ ]

ADD(●, LIT(3)) :: ADD(●, LIT(5)) :: [ ]

# Example

Expression

ADD (ADD(LIT(1), LIT(3)), LIT(5))

ADD(LIT(1), LIT(3))

LIT(1)

1

Context Stack

[ ]

ADD(●, LIT(5)) :: [ ]

ADD(●, LIT(3)) :: ADD(●, LIT(5)) :: [ ]

ADD(●, LIT(3)) :: ADD(●, LIT(5)) :: [ ]

# Example

Expression

ADD (ADD(LIT(1), LIT(3)), LIT(5))

ADD(LIT(1), LIT(3))

LIT(1)

1

LIT(3)

Context Stack

[ ]

ADD(●, LIT(5)) :: [ ]

ADD(●, LIT(3)) :: ADD(●, LIT(5)) :: [ ]

ADD(●, LIT(3)) :: ADD(●, LIT(5)) :: [ ]

ADD(1, ●) :: ADD(●, LIT(5)) :: [ ]

# Example

Expression

Context Stack

ADD (ADD(LIT(1), LIT(3)), LIT(5))

[ ]

ADD(LIT(1), LIT(3))

ADD(●, LIT(5)) :: [ ]

LIT(1)

ADD(●, LIT(3)) :: ADD(●, LIT(5)) :: [ ]

1

ADD(●, LIT(3)) :: ADD(●, LIT(5)) :: [ ]

LIT(3)

ADD(1, ●) :: ADD(●, LIT(5)) :: [ ]

3

ADD(1, ●) :: ADD(●, LIT(5)) :: [ ]

# Example

Expression

Context Stack

ADD (ADD(LIT(1), LIT(3)), LIT(5))

[ ]

ADD(LIT(1), LIT(3))

ADD(●, LIT(5)) :: [ ]

LIT(1)

ADD(●, LIT(3)) :: ADD(●, LIT(5)) :: [ ]

1

ADD(●, LIT(3)) :: ADD(●, LIT(5)) :: [ ]

LIT(3)

ADD(1, ●) :: ADD(●, LIT(5)) :: [ ]

3

ADD(1, ●) :: ADD(●, LIT(5)) :: [ ]

4

ADD(●, LIT(5)) :: [ ]

# Example

Expression	Context Stack
ADD (ADD(LIT(1), LIT(3)), LIT(5))	[ ]
ADD(LIT(1), LIT(3))	ADD(•, LIT(5)) :: [ ]
LIT(1)	ADD(•, LIT(3)) :: ADD(•, LIT(5)) :: [ ]
1	ADD(•, LIT(3)) :: ADD(•, LIT(5)) :: [ ]
LIT(3)	ADD(1, •) :: ADD(•, LIT(5)) :: [ ]
3	ADD(1, •) :: ADD(•, LIT(5)) :: [ ]
4	ADD(•, LIT(5)) :: [ ]
LIT(5)	ADD(4, •) :: [ ]



# Example

Expression	Context Stack
ADD (ADD(LIT(1), LIT(3)), LIT(5))	[ ]
ADD(LIT(1), LIT(3))	ADD(●, LIT(5)) :: [ ]
LIT(1)	ADD(●, LIT(3)) :: ADD(●, LIT(5)) :: [ ]
1	ADD(●, LIT(3)) :: ADD(●, LIT(5)) :: [ ]
LIT(3)	ADD(1, ●) :: ADD(●, LIT(5)) :: [ ]
3	ADD(1, ●) :: ADD(●, LIT(5)) :: [ ]
4	ADD(●, LIT(5)) :: [ ]
LIT(5)	ADD(4, ●) :: [ ]
5	ADD(4, ●) :: [ ]

# Example

Expression	Context Stack
ADD (ADD(LIT(1), LIT(3)), LIT(5))	[ ]
ADD(LIT(1), LIT(3))	ADD(●, LIT(5)) :: [ ]
LIT(1)	ADD(●, LIT(3)) :: ADD(●, LIT(5)) :: [ ]
1	ADD(●, LIT(3)) :: ADD(●, LIT(5)) :: [ ]
LIT(3)	ADD(1, ●) :: ADD(●, LIT(5)) :: [ ]
3	ADD(1, ●) :: ADD(●, LIT(5)) :: [ ]
4	ADD(●, LIT(5)) :: [ ]
LIT(5)	ADD(4, ●) :: [ ]
5	ADD(4, ●) :: [ ]
9	[ ]

## Reduction semantics

An alternative representation of small-step operational semantics with explicit representation of the reduction contexts

- The technique is useful to model control operators
- In reduction semantics we have a direct control over what to execute and when

Judgment:  $\langle e/v, \rho, \sigma, S \rangle \rightarrow \langle e'/v', \rho', \sigma', S' \rangle$

- $e/v$ : either an expression  $e$  or a value  $v$
- $\rho$ : environment
- $\sigma$ : store
- $S$ : stack of evaluation context

$$\frac{x \in \text{dom}(\rho)}{\langle \text{SET}(x, e), \rho, \sigma, S \rangle \rightarrow \langle e, \rho, \sigma, \text{SET}(x, \bullet) :: S \rangle} \text{ ASSIGN}$$

$$\frac{\rho(x) = \ell}{\langle v, \rho, \sigma, \text{SET}(x, \bullet) :: S \rangle \rightarrow \langle v, \rho, \sigma \{ \ell \mapsto v \}, S \rangle} \text{ FINISH - ASSIGN}$$

$$\frac{}{\langle \text{IF}(e_1, e_2, e_3), \rho, \sigma, S \rangle \rightarrow \langle e_1, \rho, \sigma, \text{IF}(\bullet, e_2, e_3) :: S \rangle} \text{IF}$$

$$\frac{v \neq \text{BOOLV}(\#f)}{\langle v, \rho, \sigma, \text{IF}(\bullet, e_2, e_3) :: S \rangle \rightarrow \langle e_2, \rho, \sigma, S \rangle} \text{IF - TRUE}$$

$$\frac{v = \text{BOOLV}(\#f)}{\langle v, \rho, \sigma, \text{IF}(\bullet, e_2, e_3) :: S \rangle \rightarrow \langle e_3, \rho, \sigma, S \rangle} \text{IF - FALSE}$$

## while loop

$$\frac{}{\langle \text{WHILE}(e_1, e_2), \rho, \sigma, S \rangle \rightarrow \langle e_1, \rho, \sigma, \text{WHILE}(e_1, e_2) :: S \rangle} \text{WHILE}$$

$$\frac{v \neq \text{BOOLV}(\#f)}{\langle v, \rho, \sigma, \text{WHILE}(e_1, e_2) :: S \rangle \rightarrow \langle e_2, \rho, \sigma, \text{EVAL\_BODY}(e_1, e_2) :: S \rangle} \text{WHILE - TRUE}$$

$$\frac{v = \text{BOOLV}(\#f)}{\langle v, \rho, \sigma, \text{WHILE}(e_1, e_2) :: S \rangle \rightarrow \langle \text{BOOLV}(\#f), \rho, \sigma, S \rangle} \text{WHILE - FALSE}$$

$$\frac{}{\langle v, \rho, \sigma, \text{EVAL\_BODY}(e_1, e_2) :: S \rangle \rightarrow \langle e_1, \rho, \sigma, \text{WHILE}(e_1, e_2) :: S \rangle} \text{WHILE - BODY}$$

`(try-catch eb eh)`

- Evaluate handler  $e_h$  which must produce a function  $f$
- Install  $f$  as the most recent handler
- Evaluate body  $e_b$  and proceed as follows:
  1. If  $e_b$  evaluates to  $v$  without throwing any exception
    - Uninstall the handler  $f$
    - Produce  $v$  as the result of the `try-catch` expression
  2. If during the evaluation of  $e_b$  a value  $v$  is thrown
    - Uninstall the handler  $f$
    - Produce  $(f\ v)$  as the result of the `try-catch` expression



$$\frac{}{\langle \text{THROW}(e), \rho, \sigma, S \rangle \rightarrow \langle e, \rho, \sigma, \text{THROW}(\bullet) :: S \rangle} \text{THROW}$$

$$\frac{}{\langle v, \rho, \sigma, \text{THROW}(\bullet) :: \text{TRYCATCH}(\bullet, v_h) :: S \rangle \rightarrow \langle \text{APPLY}(v_h, v), \rho, \sigma, S \rangle}$$

THROW – TRANSFER

$$\frac{F \text{ does not have the form } \text{TRYCATCH}(\bullet, v)}{\langle v, \rho, \sigma, \text{THROW}(\bullet) :: F :: S \rangle \rightarrow \langle v, \rho, \sigma, \text{THROW}(\bullet) :: S \rangle} \text{THROW – UNWIND}$$

## try-catch expression

$$\frac{}{\langle \text{TRYCATCH}(e_b, e_h), \rho, \sigma, S \rangle \rightarrow \langle e_h, \rho, \sigma, \text{TRYCATCH}(e_b, \bullet) :: S \rangle} \text{TRY - CATCH}$$

$v$  is a function

$$\frac{}{\langle v, \rho, \sigma, \text{TRYCATCH}(e_b, \bullet) :: S \rangle \rightarrow \langle e_b, \rho, \sigma, \text{TRYCATCH}(\bullet, v) :: \text{LETENV}(\rho) :: S \rangle}$$

TRY - CATCH - HANDLER

$$\frac{}{\langle v, \rho, \sigma, \text{TRYCATCH}(\bullet, v_h) :: S \rangle \rightarrow \langle v, \rho, \sigma, S \rangle} \text{TRY - CATCH - FINISH}$$

## **This Lecture**

- Reduction Semantics for Control Structures

## **Next Lecture**

- Garbage Collection