Principles of Programming Languages
Lecture 4: Side-effects. I/O. Local variables.

Andrei Arusoaie[1]

[1]Department of Computer Science

October 24, 2017

Outline

Outline

Outline

Side-effects

Input/Output

Local variables

Side-effects

- What are side-effects in computer programming?
- observable change over the *“outside world”*
- Example: expression `2 + 3`
  - evaluates to `5`
- Example: expression `++x`
  - evaluates to `x + 1` **and** changes the state

Side-effects

- What are side-effects in computer programming?
- observable change over the *“outside world”*
- Example: expression `2 + 3`
  - evaluates to `5`
- Example: expression `++x`
  - evaluates to `x + 1` **and** changes the state

Side-effects

- What are side-effects in computer programming?
- observable change over the *“outside world”*
- Example: expression `2 + 3`
  - evaluates to `5`
- Example: expression `++x`
  - evaluates to `x + 1` **and** changes the state

Side-effects

- What are side-effects in computer programming?
- observable change over the *“outside world”*
- Example: expression `2 + 3`
  - evaluates to `5`
- Example: expression `++x`
  - evaluates to `x + 1` **and** changes the state

Side-effects

- What are side-effects in computer programming?
- observable change over the *“outside world”*
- Example: expression `2 + 3`
  - evaluates to `5`
- Example: expression `++x`
  - evaluates to `x + 1` **and** changes the state

Side-effects

- What are side-effects in computer programming?
- observable change over the *"outside world"*
- Example: expression `2 + 3`
  - evaluates to `5`
- Example: expression `++x`
  - evaluates to `x + 1` **and** changes the state

Increment in K

- Syntax:

```
syntax AExp ::= "++" Id
```

- Semantics:

```
rule <k> ++X => I +Int 1 ...</k>
     <env>... X |-> (I => I +Int 1) ...</env>
```

- Demo

Increment in K

- Syntax:

```
syntax AExp ::= "++" Id
```

- Semantics:

```
rule <k> ++X => I +Int 1 ...</k>
    <env>... X |-> (I => I +Int 1) ...</env>
```

- Demo

Increment in K

- Syntax:

```
syntax AExp ::= "++" Id
```

- Semantics:

```
rule <k> ++X => I +Int 1 ...</k>
   <env>... X |-> (I => I +Int 1) ...</env>
```

- Demo

Input/Output

- Semantics of I/O
- Input: read an integer
  - `read()`
- Output: print an expression
  - `print(x);`

Input/Output

- Semantics of I/O
- Input: read an integer
  - `read()`
- Output: print an expression
  - `print(x);`

Input/Output

- Semantics of I/O
- Input: read an integer
  - `read()`
- Output: print an expression
  - `print(x);`

Input

Steps:

1. Add syntax declaration:

```
syntax AExp ::= "read" "(" ")"
```

2. Add a new cell connected to `stdin`:

```
<in stream="stdin"> .List </in>
```

3. Add a rule for reading a value:

```
rule <k> (read() => I) ...</k>
     <in> (ListItem(I:Int) => .List) ...</in>
```

DEMO

Input

Steps:

1. Add syntax declaration:

```
syntax AExp ::= "read" "(" ")"
```

2. Add a new cell connected to `stdin`:

```
<in stream="stdin"> .List </in>
```

3. Add a rule for reading a value:

```
rule <k> (read() => I) ...</k>
  <in> (ListItem(I:Int) => .List) ...</in>
```

DEMO

Input

Steps:

1. Add syntax declaration:

```
syntax AExp ::= "read" "(" ")"
```

2. Add a new cell connected to `stdin`:

```
<in stream="stdin"> .List </in>
```

3. Add a rule for reading a value:

```
rule <k> (read() => I) ...</k>
  <in> (ListItem(I:Int) => .List) ...</in>
```

DEMO

Input

Steps:

1. Add syntax declaration:

```
syntax AExp ::= "read" "(" ")"
```

2. Add a new cell connected to `stdin`:

```
<in stream="stdin"> .List </in>
```

3. Add a rule for reading a value:

```
rule <k> (read() => I) ...</k>
  <in> (ListItem(I:Int) => .List) ...</in>
```

DEMO

Input

Steps:

1. Add syntax declaration:

```
syntax AExp ::= "read" "(" ")"
```

2. Add a new cell connected to `stdin`:

```
<in stream="stdin"> .List </in>
```

3. Add a rule for reading a value:

```
rule <k> (read() => I) ...</k>
  <in> (ListItem(I:Int) => .List) ...</in>
```

DEMO

Output

Steps:

1. Add syntax declaration:

```
syntax Stmt ::= "print" "(" AExp ")"
```

2. Add a new cell connected to `stdout`:

```
<out stream="stdout"> .List </out>
```

3. Add a rule that outputs a value:

```
rule <k> (print(I:Int); => .) ...</k>
  <out>...  (.List => ListItem(I)) </out>
```

DEMO

Output

Steps:

1. Add syntax declaration:

```
syntax Stmt ::= "print" "(" AExp ")"
```

2. Add a new cell connected to `stdout`:

```
<out stream="stdout"> .List </out>
```

3. Add a rule that outputs a value:

```
rule <k> (print(I:Int); => .)  ...</k>
  <out>...  (.List => ListItem(I)) </out>
```

DEMO

Output

Steps:

1. Add syntax declaration:

```
syntax Stmt ::= "print" "(" AExp ")"
```

2. Add a new cell connected to `stdout`:

```
<out stream="stdout"> .List </out>
```

3. Add a rule that outputs a value:

```
rule <k> (print(I:Int); => .)  ...</k>
  <out>...  (.List => ListItem(I)) </out>
```

DEMO

Output

Steps:

1. Add syntax declaration:

```
syntax Stmt ::= "print" "(" AExp ")"
```

2. Add a new cell connected to `stdout`:

```
<out stream="stdout"> .List </out>
```

3. Add a rule that outputs a value:

```
rule <k> (print(I:Int); => .)  ...</k>
  <out>...  (.List => ListItem(I)) </out>
```

DEMO

Output

Steps:

1. Add syntax declaration:

```
syntax Stmt ::= "print" "(" AExp ")"
```

2. Add a new cell connected to `stdout`:

```
<out stream="stdout"> .List </out>
```

3. Add a rule that outputs a value:

```
rule <k> (print(I:Int); => .)  ...</k>
  <out>...  (.List => ListItem(I)) </out>
```

DEMO

Output

Print strings:

1. Add syntax declaration:

```
syntax AExp ::= String
```

2. Add a new **sort** to group the printable expressions:

```
syntax Printable ::= Int | String
syntax AExp ::= Printable
```

3. Modify the output rule:

```
rule <k> (print(I:Printable); => .)  ...</k>
     <out>...  (.List => ListItem(I)) </out>
```

DEMO

Output

Print strings:

1. Add syntax declaration:

```
syntax AExp ::= String
```

2. Add a new **sort** to group the printable expressions:

```
syntax Printable ::= Int | String
syntax AExp ::= Printable
```

3. Modify the output rule:

```
rule <k> (print(I:Printable); => .)  ...</k>
     <out>...  (.List => ListItem(I)) </out>
```

DEMO

Output

Print strings:

1. Add syntax declaration:

```
syntax AExp ::= String
```

2. Add a new **sort** to group the printable expressions:

```
syntax Printable ::= Int | String
syntax AExp ::= Printable
```

3. Modify the output rule:

```
rule <k> (print(I:Printable); => .)  ...</k>
     <out>...  (.List => ListItem(I)) </out>
```

DEMO

Output

Print strings:

1. Add syntax declaration:

```
syntax AExp ::= String
```

2. Add a new **sort** to group the printable expressions:

```
syntax Printable ::= Int | String
syntax AExp ::= Printable
```

3. Modify the output rule:

```
rule <k> (print(I:Printable); => .)  ...</k>
     <out>...  (.List => ListItem(I)) </out>
```

DEMO

Output

Print strings:

1. Add syntax declaration:

```
syntax AExp ::= String
```

2. Add a new **sort** to group the printable expressions:

```
syntax Printable ::= Int | String
syntax AExp ::= Printable
```

3. Modify the output rule:

```
rule <k> (print(I:Printable); => .)   ...</k>
     <out>...   (.List => ListItem(I)) </out>
```

DEMO

Output

Print strings:

1. Add syntax declaration:

```
syntax AExp ::= String
```

2. Add a new **sort** to group the printable expressions:

```
syntax Printable ::= Int | String
syntax AExp ::= Printable
```

3. Modify the output rule:

```
rule <k> (print(I:Printable); => .)   ...</k>
     <out>...   (.List => ListItem(I)) </out>
```

DEMO

Local variables

- How to handle local variables in blocks?
- ```
  int x;
  int y;
  x = 1;
  y = 10;
  {
     int x;
     x = 3;
     print(x); print(y);
  }
  print(x); print(y);
  ```
- **Visibility: variable scope**
- **Hidding: variable shadowing**
  ```

Local variables

- How to handle local variables in blocks?
- ```
  int x;
  int y;
  x = 1;
  y = 10;
  {
     int x;
     x = 3;
     print(x); print(y);
  }
  print(x); print(y);
  ```
- Visibility: variable scope
- Hidding: variable shadowing
  ```

Local variables

- How to handle local variables in blocks?
- ```
  int x;
  int y;
  x = 1;
  y = 10;
  {
     int x;
     x = 3;
     print(x); print(y);
  }
  print(x); print(y);
  ```
- **Visibility**: **variable scope**
- **Hidding**: **variable shadowing**
  ```

Local variables: behavior

- When entering the block:
  - Redeclared variables **hide** previous declarations (e.g., `x`)
  - The undeclared (inherited) variables are made **visible** in the block (e.g., `y`)
- When exiting the block:
  - Redeclared variables are de-activated (eventually de-allocated)
  - The old environment is restored
- DEMO: how does this work in C?
- Solutions?

Local variables: behavior

- When entering the block:
  - Redeclared variables **hide** previous declarations (e.g., `x`)
  - The undeclared (inherited) variables are made **visible** in the block (e.g., `y`)
- When exiting the block:
  - Redeclared variables are de-activated (eventually de-allocated)
  - The old environment is restored
- DEMO: how does this work in C?
- Solutions?

Local variables: behavior

- When entering the block:
  - Redeclared variables **hide** previous declarations (e.g., `x`)
  - The undeclared (inherited) variables are made **visible** in the block (e.g., `y`)
- When exiting the block:
  - Redeclared variables are de-activated (eventually de-allocated)
  - The old environment is restored
- DEMO: how does this work in C?
- Solutions?

Local variables: behavior

- When entering the block:
  - Redeclared variables **hide** previous declarations (e.g., `x`)
  - The undeclared (inherited) variables are made **visible** in the block (e.g., `y`)
- When exiting the block:
  - Redeclared variables are de-activated (eventually de-allocated)
  - The old environment is restored
- DEMO: how does this work in C?
- Solutions?

Local variables: behavior

- When entering the block:
  - Redeclared variables **hide** previous declarations (e.g., `x`)
  - The undeclared (inherited) variables are made **visible** in the block (e.g., `y`)
- When exiting the block:
  - Redeclared variables are de-activated (eventually de-allocated)
  - The old environment is restored
- DEMO: how does this work in C?
- Solutions?

Local variables: behavior

- When entering the block:
  - Redeclared variables **hide** previous declarations (e.g., `x`)
  - The undeclared (inherited) variables are made **visible** in the block (e.g., `y`)
- When exiting the block:
  - Redeclared variables are de-activated (eventually de-allocated)
  - The old environment is restored
- DEMO: how does this work in C?
- Solutions?

Local variables: behavior

- When entering the block:
  - Redeclared variables **hide** previous declarations (e.g., `x`)
  - The undeclared (inherited) variables are made **visible** in the block (e.g., `y`)
- When exiting the block:
  - Redeclared variables are de-activated (eventually de-allocated)
  - The old environment is restored
- DEMO: how does this work in C?
- Solutions?

Local variables: behavior

- When entering the block:
  - Redeclared variables **hide** previous declarations (e.g., `x`)
  - The undeclared (inherited) variables are made **visible** in the block (e.g., `y`)
- When exiting the block:
  - Redeclared variables are de-activated (eventually de-allocated)
  - The old environment is restored
- DEMO: how does this work in C?
- Solutions?

Local variables in K

Memory refinement:

- environment + variable references
- `<env>`: `varname` $\mapsto$ `reference`
- `<store>`: `reference` $\mapsto$ `value`

Solution: keep the environment organised in a stack

1. either in the `k` cell
2. or use a dedicated `<stack>`

Local variables in K

Memory refinement:

- environment + variable references
- `<env>`: `varname` $\mapsto$ `reference`
- `<store>`: `reference` $\mapsto$ `value`

Solution: keep the environment organised in a stack

1. either in the `k` cell
2. or use a dedicated `<stack>`

Local variables in K

Memory refinement:

- environment + variable references
- `<env>`: `varname` $\mapsto$ `reference`
- `<store>`: `reference` $\mapsto$ `value`

Solution: keep the environment organised in a stack

1. either in the `k` cell
2. or use a dedicated `<stack>`

Local variables in K

Memory refinement:

- environment + variable references
- `<env>`: `varname` $\mapsto$ `reference`
- `<store>`: `reference` $\mapsto$ `value`

Solution: keep the environment organised in a stack

1. either in the `k` cell
2. or use a dedicated `<stack>`

Solution 1: stack in the <k> cell

Steps:

1. memory refinement + update rules: assignment, lookup
2. Revise declarations rule:

```
rule <k> int (X, Xs => Xs); ...</k>
       <env> Rho:Map => Rho[X <- !L:Int] </env>
       <store> Sigma:Map (. => (!L |-> 0)) </store>
```

3. Replace the rule for non-empty block with:

```
rule <k> { S:Stmt } => S ↷ Rho ...</k> <env> Rho </env>
rule <k> Rho => . ...</k> <env> _ => Rho </env>
```

Solution 1: stack in the <k> cell

Steps:

1. memory refinement + update rules: assignment, lookup
2. Revise declarations rule:

```
rule <k> int (X, Xs => Xs); ...</k>
       <env> Rho:Map => Rho[X <- !L:Int] </env>
       <store> Sigma:Map (. => (!L |-> 0)) </store>
```

3. Replace the rule for non-empty block with:

```
rule <k> { S:Stmt } => S ↷ Rho ...</k> <env> Rho </env>
rule <k> Rho => . ...</k> <env> _ => Rho </env>
```

Solution 1: stack in the <k> cell

Steps:

1. memory refinement + update rules: assignment, lookup
2. Revise declarations rule:

```
rule <k> int (X, Xs => Xs); ...</k>
       <env> Rho:Map => Rho[X <- !L:Int] </env>
       <store> Sigma:Map (. => (!L |-> 0)) </store>
```

3. Replace the rule for non-empty block with:

```
rule <k> { S:Stmt } => S ↷ Rho ...</k> <env> Rho </env>
rule <k> Rho => . ...</k> <env> _ => Rho </env>
```

Solution 2: stack in dedicated cell

Steps:

1. Append a new cell: `<stack> .List </stack>`
2. Restore point: `syntax KItem ::= "restoreEnv"`
3. Replace the rule for non-empty block with:

```
rule <k> { S:Stmt } => S ↷ restoreEnv ...</k>
     <env> Rho </env>
     <stack> (. => ListItem(Rho)) ...</stack>

rule <k> restoreEnv => . ...</k>
     <env> _ => Rho </env>
     <stack> (ListItem(Rho) => .List) ...</stack>
```

Solution 2: stack in dedicated cell

Steps:

1. Append a new cell: `<stack> .List </stack>`
2. Restore point: `syntax KItem ::= "restoreEnv"`
3. Replace the rule for non-empty block with:

```
rule <k> { S:Stmt } => S ↷ restoreEnv ...</k>
     <env> Rho </env>
     <stack> (. => ListItem(Rho)) ...</stack>

rule <k> restoreEnv => . ...</k>
     <env> _ => Rho </env>
     <stack> (ListItem(Rho) => .List) ...</stack>
```

Solution 2: stack in dedicated cell

Steps:

1. Append a new cell: `<stack> .List </stack>`
2. Restore point: `syntax KItem ::= "restoreEnv"`
3. Replace the rule for non-empty block with:

```
rule <k> { S:Stmt } => S ↷ restoreEnv ...</k>
      <env> Rho </env>
      <stack> (. => ListItem(Rho)) ...</stack>

rule <k> restoreEnv => . ...</k>
      <env> _ => Rho </env>
      <stack> (ListItem(Rho) => .List) ...</stack>
```

Lab this week

- Given some incomplete language specification in K and some informal hand-written specification, you will create an interpreter for a language using K that covers both specifications