Principles of Programming Languages

Lecture 02 While

Albert-Ludwigs-Universität Freiburg

Peter Thiemann

University of Freiburg, Germany

thiemann@informatik.uni-freiburg.de

23 Apr 2018

The While Programming Language

- Core **imperative** programming language
- Part of most programming languages
- Imperative
  - program state
  - statement: state transformation

Syntax of WHILE

Expressions

$e \quad ::= \quad x \mid e+e \mid \ldots$

Statements

s	$::=$	`x:=e`	assignment
	$\mid$	`skip`	empty statement
	$\mid$	`s;s`	sequence
	$\mid$	`if e then s else s`	conditional
	$\mid$	`while e do s`	repetition

Modeling the state

$$
\begin{aligned}
\sigma \ni \quad \mathsf{Store} &= \mathsf{Var} \hookrightarrow \mathsf{Val} \\
y \ni \quad \mathsf{Val} &= \mathbf{Z}
\end{aligned}
$$

- Variables contain integers
- Some variables may not be defined

$\Rightarrow$ store is a partial function from variables to values

Big-Step Semantics

- Two syntactic categories e and s with different outcomes

$\Rightarrow$ two evaluation relations needed

Evaluation of expressions

- input: current state and expression
- output: value of expression
- need relation $\sigma, e \hookrightarrow y$

Evaluation of statements

- input: current state and statement
- output: state after executing statement
- need relation $\sigma, s \hookrightarrow \sigma'$

Big-Step Evaluation of Expressions

$$\sigma, x \hookrightarrow \sigma(x) \quad \text{if } \sigma(x) \text{ defined}$$

$$\frac{\sigma, e_1 \hookrightarrow n_1 \qquad \sigma, e_2 \hookrightarrow n_2}{\sigma, e_1\texttt{+}e_2 \hookrightarrow m} \quad \text{if } m = n_1 + n_2 \in \mathbf{Z}$$

Big-Step Execution of Statements I

Assign

$$\frac{\sigma, e \hookrightarrow n}{\sigma, x{:=}e \hookrightarrow \sigma[x \mapsto n]}$$

Skip

$$\sigma, \texttt{skip} \hookrightarrow \sigma$$

Seq

$$\frac{\sigma, s_1 \hookrightarrow \sigma' \qquad \sigma', s_2 \hookrightarrow \sigma''}{\sigma, s_1 \, ; s_2 \hookrightarrow \sigma''}$$

Big-Step Execution of Statements II

IfTrue

$$\frac{\sigma, e \hookrightarrow n \qquad \sigma, s_1 \hookrightarrow \sigma'}{\sigma, \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \hookrightarrow \sigma'} \quad \text{if } n \neq 0$$

IfFalse

$$\frac{\sigma, e \hookrightarrow 0 \qquad \sigma, s_2 \hookrightarrow \sigma'}{\sigma, \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \hookrightarrow \sigma'}$$

Big-Step Execution of Statements III

WhileTrue

$$\frac{\sigma, e \hookrightarrow n \qquad \sigma, s \hookrightarrow \sigma' \qquad \sigma', \texttt{while}\ e\ \texttt{do}\ s \hookrightarrow \sigma''}{\sigma, \texttt{while}\ e\ \texttt{do}\ s \hookrightarrow \sigma''} \quad \text{if } n \neq 0$$

WhileFalse

$$\frac{\sigma, e \hookrightarrow 0}{\sigma, \texttt{while}\ e\ \texttt{do}\ s \hookrightarrow \sigma}$$

Small-Step Semantics

Reduction of expressions

- (machine) state: current state and expression
- need transition relation: $\sigma, e \longrightarrow e'$ (no output state needed)

Reduction of Statements

- (machine) state: current state and statement
- but there is no statement in the last step
- **two** transition relations
  - $\sigma, s \longrightarrow \sigma', s'$ if there is a nested step to take
  - $\sigma, s \longrightarrow \sigma'$ if there is only one step

Small-Step Reduction of Expressions

EVar

$$\sigma, x \longrightarrow \sigma(x) \quad \text{if } \sigma(x) \text{ defined}$$

EOp

$$\sigma, n_1 + n_2 \longrightarrow m \quad \text{where } m = n_1 + n_2 \in \mathbf{Z}$$

EOpL

$$\frac{\sigma, e_1 \longrightarrow e_1'}{\sigma, e_1 + e_2 \longrightarrow e_1' + e_2}$$

EOpR

$$\frac{\sigma, e_2 \longrightarrow e_2'}{\sigma, n + e_2 \longrightarrow n + e_2'}$$

Small-Step Reduction of Statements I

SAssign

$$\sigma, x{:=}n \longrightarrow \sigma[x \mapsto n]$$

SAssignStep

$$\frac{\sigma, e \longrightarrow e'}{\sigma, x{:=}e \longrightarrow \sigma, x{:=}e'}$$

SSkip

$$\sigma, \texttt{skip} \longrightarrow \sigma$$

SSeqL

$$\frac{\sigma, s_1 \longrightarrow \sigma'}{\sigma, s_1;s_2 \longrightarrow \sigma', s_2}$$

SSeqStep

$$\frac{\sigma, s_1 \longrightarrow \sigma', s_1'}{\sigma, s_1;s_2 \longrightarrow \sigma', s_1';s_2}$$

Small-Step Reduction of Statements II

SIfTrue

$$\sigma, \texttt{if } n \texttt{ then } s_1 \texttt{ else } s_2 \longrightarrow \sigma, s_1 \quad \text{if } n \neq 0$$

SIfFalse

$$\sigma, \texttt{if } 0 \texttt{ then } s_1 \texttt{ else } s_2 \longrightarrow \sigma, s_2$$

SIfStep

$$\frac{\sigma, e \longrightarrow e'}{\sigma, \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \longrightarrow \sigma, \texttt{if } e' \texttt{ then } s_1 \texttt{ else } s_2}$$

Small-Step Reduction of Statements III

SWhile

$$\sigma, \texttt{while}\ e\ \texttt{do}\ s \longrightarrow \sigma, \texttt{if}\ e\ \texttt{then}\ (s\texttt{;while}\ e\ \texttt{do}\ s)\ \texttt{else skip}$$

- while is handled by **unfolding**

Plan

Exceptions

If there is an exception (e.g. division by zero)

- Small step semantics: reduction gets stuck
- Big step semantics: so far undefined

Solution

- Small step: Judgment with combined result
- Big step: Separate judgments for normal result and exceptional results

Big-Step Exceptions: Separate Judgments

$$r ::= \texttt{div0} \mid \ldots$$

Two Judgments

normal evaluation $e \hookrightarrow y$ as before

exceptional evaluation $e \Uparrow r$ (very schematic definition)

$$\frac{e_1 \hookrightarrow n_1 \qquad e_2 \hookrightarrow 0}{e_1 / e_2 \Uparrow \texttt{div0}} \qquad \frac{e_1 \Uparrow r}{e_1 - e_2 \Uparrow r} \qquad \frac{e_1 \Uparrow r}{e_1 / e_2 \Uparrow r}$$

$$\frac{e_1 \hookrightarrow n_1 \qquad e_2 \Uparrow r}{e_1 - e_2 \Uparrow r} \qquad \frac{e_1 \hookrightarrow n_1 \qquad e_2 \Uparrow r}{e_1 / e_2 \Uparrow r}$$

Results

For each expression e

- $e \hookrightarrow n$, for some n or
- $e \Uparrow r$, for some r.

Exceptions for Statements

- Similar approach: judgments $\sigma, s \hookrightarrow \sigma'$ and $\sigma, s \Uparrow \sigma', r$.
- But σ, s may not terminate, so we do not have a result corresponding to expressions.

Language Extension: Catch and Throw Exceptions

$$s ::= \cdots \mid \texttt{throw}\ r \mid \texttt{try}\ s\ \texttt{catch}\ r\ \texttt{then}\ s$$

$$\sigma, \texttt{throw}\ r \Uparrow \sigma, r \qquad \frac{\sigma, s_1 \hookrightarrow \sigma'}{\sigma, \texttt{try}\ s_1\ \texttt{catch}\ r\ \texttt{then}\ s_2 \hookrightarrow \sigma'}$$

$$\frac{\sigma, s_1 \Uparrow \sigma', r \qquad \sigma', s_2 \hookrightarrow \sigma''}{\sigma, \texttt{try}\ s_1\ \texttt{catch}\ r\ \texttt{then}\ s_2 \hookrightarrow \sigma''}$$

- what else could happen?

Language Extension: Catch and Throw Exceptions II

Uncaught exception

$$\frac{\sigma, s_1 \Uparrow \sigma', r'}{\sigma, \texttt{try}\ s_1\ \texttt{catch}\ r\ \texttt{then}\ s_2 \Uparrow \sigma', r'} \quad \text{where } r \neq r'$$

Exception in exception handler

$$\frac{\sigma, s_1 \Uparrow \sigma', r \qquad \sigma', s_2 \Uparrow \sigma'', r'}{\sigma, \texttt{try}\ s_1\ \texttt{catch}\ r\ \texttt{then}\ s_2 \Uparrow \sigma'', r'}$$

Plan

WHILE with Procedures

Programs

p	$::=$	s	main program
	$\mid$	$\texttt{proc}\ f(\overline{x})s; p$	procedure definition

Statements (ext)

s	$::=$	$\dots$	
	$\mid$	$f(\overline{e})$	procedure call

- Useful extension?

Properties

A procedure

- can take parameters
- cannot return a value
- can modify fixed (global) variables
- cannot modify arbitrary variables
- can be called recursively
- cannot be nested in another procedure

The other alternatives

also make sense — stay tuned

Modeling Procedure Calls

- procedure parameters are **values of expressions**
- stored in **local** variables, i.e., new variables for each procedure call

$\Rightarrow$ store is now structured into **local store** and **global store**

$\Rightarrow$ use indirection to implement

Parameter passing

Here: Call-by-value

- parameter expressions are evaluated before the procedure call

Alternative: Call-by-name

- parameter expressions are passed **unevaluated**

Alternative: Call-by-reference

- parameters must be variable names
- assignments to parameters in the procedure are visible at the call site

The Store Model

$$
\begin{array}{lcl}
\mu \ni \text{Memory} & = & \text{Address} \hookrightarrow \text{Val} \\
\sigma \ni \text{Store} & = & \text{Var} \hookrightarrow \text{Address}
\end{array}
$$

Big-step evaluation of expressions

$$\mu, \sigma \vdash n \hookrightarrow n \qquad \frac{\sigma(x) = a \qquad \mu(a) = n}{\mu, \sigma \vdash x \hookrightarrow n}$$

$$\frac{\mu, \sigma \vdash e_1 \hookrightarrow n_1 \qquad \mu, \sigma \vdash e_2 \hookrightarrow n_2}{\mu, \sigma \vdash e_1 \text{-} e_2 \hookrightarrow n} \quad \text{where } n = n_1 - n_2$$

Big-step execution of statements
Procedure call

SCall-Provisional

$$\frac{\mu, \sigma \vdash \overline{e} \hookrightarrow \overline{y} \qquad \texttt{proc}\ f(\overline{x})s \in p \qquad \overline{a} \cap dom(\mu) = \emptyset \qquad \mu[\overline{a \mapsto y}], \sigma[\overline{x \mapsto a}] \vdash s \hookrightarrow \mu'}{\mu, \sigma \vdash f(\overline{e}) \hookrightarrow \mu'}$$

- Judgment adapted to indirection: $\mu, \sigma \vdash s \hookrightarrow \mu'$
  - memory μ changes and is threaded through computation
  - store (environment) σ propagated to the leaves
  - σ updated for duration of procedure call

Passing Parameters by Reference

- The previous rule SCall implemented call-by-value
- To implement call-by-reference
  - actual parameters must be variable names
  - pass the address of the variable instead of its value
  - no new variables allocated

SCallRef-Provisional

$$\frac{\sigma(\overline{z}) = \overline{a} \qquad \texttt{proc}\ f(\overline{\&x})s \in p \qquad \mu, \sigma[\overline{x \mapsto a}] \vdash s \hookrightarrow \mu'}{\mu, \sigma \vdash f(\overline{z}) \hookrightarrow \mu'}$$

Plan

Scope and Visibility

UNI FREIBURG

Scope of identifier x

- part of program where x refers to some entity (variable, procedure, type, ...)
- static (lexical) scope: program structure determines the scope — next enclosing definition is referred to
- dynamic scope: program execution determines the scope — last executed definition is referred to

Example: Scope in WHILE with Procedures

- global scope
  - all variables in the main program
  - variables used in procedures, but not declared as parameters
- procedure scope
  - variables declared as parameters
- lexical scope desired: identifier x in procedure $p(\overline{x})s$ refers to
  - parameter x, if present in $\overline{x}$;
  - global variable x, otherwise.
- rule SCall
  - defines scoping by manipulation of environment σ
  - global visibility of procedure names p (enables mutually recursive procedure calls)

Static Scope vs Dynamic Scope
A While-Proc program

$$p(x, y)\{q(y)\} \tag{1}$$
$$q(y)\{z := x + y\} \tag{2}$$
$$x := 42; \tag{3}$$
$$p(17, 4) \tag{4}$$

static scope terminates with $z = 46$

dynamic scope (hypothetical) in q, the last executed definition of x is setting the parameter x of p to 17. Hence, $z = 21$ in the end.

- The first Lisp interpreter accidentally implemented dynamic scope.
- Emacs Lisp uses dynamic scope.
- SCall-Provisional implements dynamic scope!

Visibility

- A variable may be in scope, but not visible
- It can be **shadowed** by a closer redefinition

Example

- In While-Proc, global variables can be shadowed by parameters
- Shadowing is purely lexical as shown in Example (1)

Plan

Extension: Local Variables

- Parameters in While-Proc are assignable, but that's not a good programming style
- Introduce **local variables**
- Local variables are freshly allocated for each procedure call, they can be accessed during the call, and are unavailable afterwards
- Their scope is the body of the procedure

Revised abstract syntax of programs

p	$::=$	s	main program
	$\mid$	$\texttt{proc}\ f(\overline{x})\{\texttt{var}\ \overline{z}; s\}; p$	procedure definition

Implementation: Local Variables

- only procedure call rule affected

SCallLocal-Provisional

$$\frac{\mu, \sigma \vdash \overline{e} \hookrightarrow \overline{y} \qquad \texttt{proc}\ f(\overline{x})\{\texttt{var}\ \overline{z}; s\} \in p \qquad \overline{a}, \overline{b} \cap dom(\mu) = \emptyset \qquad \mu[\overline{a \mapsto y}, \overline{b \mapsto 0}], \sigma[\overline{x \mapsto a}, \overline{z \mapsto b}] \vdash s \hookrightarrow \mu'}{\mu, \sigma \vdash f(\overline{e}) \hookrightarrow \mu'}$$

Terminology: Activation Block

- Management of variable storage in memory μ is key part of rules for procedure call
- A sequence of active procedure calls gives rise to an environment of this form

$$\mu_0 \underbrace{[\overline{a_1 \mapsto y_1}]}_{\text{call to } f_1} \underbrace{[\overline{a_2 \mapsto y_2}]}_{\text{call to } f_2} \dots \underbrace{[\overline{a_n \mapsto y_n}]}_{\text{call to } f_n}$$

- Each part $[\overline{a_i \mapsto y_i}]$ is an **activation block** for procedure f_i, which consists of
  - values of the parameters
  - values of the local variables
  - (other local structures; on a machine: return address)
- It describes all local information needed to execute the procedure call and to return from it.
- Typically allocated on the machine stack.

Extension: Nested Procedures

- In many languages, procedures, function, methods can be **nested**, that is, a procedure f can have **local** procedure declarations whose scope is just the body of f
- A **block** is a scope unit consisting of
  - variable declarations,
  - procedure declarations, and
  - a statement
- The main program is just the top-level block

blocks

$$
\begin{array}{lcll}
b & ::= & \texttt{var}\ \overline{z}; s & \text{block} \\
 & \mid & \texttt{proc}\ f(\overline{x})\{b\}; b & \text{procedure definition}
\end{array}
$$

Modeling Nested Procedures and Blocks

- Now procedure declarations are lexically scoped, too!
- Need to adjust access to procedure declarations
- Requires a procedure environment π which maps procedure names to procedure declarations

$$\text{BProcDef}\quad \frac{\mu, \sigma, \pi[f \mapsto \texttt{proc } f(\overline{x})\{b_1\}] \vdash b_2 \hookrightarrow \mu'}{\mu, \sigma, \pi \vdash \texttt{proc } f(\overline{x})\{b_1\}; b_2 \hookrightarrow \mu'}$$

$$\text{BExec}\quad \frac{\overline{a} \cap dom(\mu) = \emptyset \qquad \mu[\overline{a \mapsto 0}], \sigma[\overline{z \mapsto a}], \pi \vdash s \hookrightarrow \mu'}{\mu, \sigma, \pi \vdash \texttt{var } \overline{z}; s \hookrightarrow \mu'}$$

Modeling Nested Procedures and Blocks II

SCallBlock-Provisional

$$\frac{\mu, \sigma \vdash \overline{e} \hookrightarrow \overline{y} \qquad \pi(f) = \texttt{proc}\ f(\overline{x})\{b\} \qquad \overline{a} \cap dom(\mu) = \emptyset \qquad \mu[\overline{a \mapsto y}], \sigma[\overline{x \mapsto a}], \pi \vdash b \hookrightarrow \mu'}{\mu, \sigma, \pi \vdash f(\overline{e}) \hookrightarrow \mu'}$$

Nested Procedures and Blocks

Consequences of the rules

- Variables and procedures live in different **namespaces**
  - variables are governed by environment σ
  - procedures by environment π
- Inner declarations shadow enclosing declarations
- Nested procedures can access
  - procedures at the same level or higher up
  - variables . . .

A Pitfall

$$\texttt{proc}\ f(x)\{g(42)\}$$
$$\texttt{proc}\ g(y)\{x := 42\}$$
$$x := 0; f(0)$$

- What is the value of global variable x in the end?

A Pitfall

$$\texttt{proc}\ f(x)\{g(42)\}$$
$$\texttt{proc}\ g(y)\{x := 42\}$$
$$x := 0; f(0)$$

- What is the value of global variable x in the end?
- Initial call $f(0)$:

$$\begin{array}{l}[a \mapsto 0][a_1 \mapsto 0],\\ [x \mapsto a][x \mapsto a_1]\end{array} \vdash g(42)\ldots$$

A Pitfall

$$\texttt{proc}\ f(x)\{g(42)\}$$
$$\texttt{proc}\ g(y)\{x := 42\}$$
$$x := 0; f(0)$$

- What is the value of global variable x in the end?
- Initial call $f(0)$:

$$\begin{array}{l}[a \mapsto 0][a_1 \mapsto 0],\\ [x \mapsto a][x \mapsto a_1]\end{array} \vdash g(42) \ldots$$

- Next call $g(42)$:

$$\begin{array}{l}[a \mapsto 0][a_1 \mapsto 0][a_2 \mapsto 42],\\ [x \mapsto a][x \mapsto a_1][y \mapsto a_2]\end{array} \vdash x := 42 \hookrightarrow [a \mapsto 0][a_1 \mapsto 42][a_2 \mapsto 42]$$

A Pitfall

$$\texttt{proc}\ f(x)\{g(42)\}$$
$$\texttt{proc}\ g(y)\{x := 42\}$$
$$x := 0; f(0)$$

- What is the value of global variable x in the end?
- Initial call $f(0)$:

$$\begin{array}{l}[a \mapsto 0][a_1 \mapsto 0], \\ [x \mapsto a][x \mapsto a_1] \quad \vdash g(42) \dots\end{array}$$

- Next call $g(42)$:

$$\begin{array}{l}[a \mapsto 0][a_1 \mapsto 0][a_2 \mapsto 42], \\ [x \mapsto a][x \mapsto a_1][y \mapsto a_2] \quad \vdash x := 42 \hookrightarrow [a \mapsto 0][a_1 \mapsto 42][a_2 \mapsto 42]\end{array}$$

- Updates f's parameter x at a_1 instead of the global variable x at a_0: **dynamic scope!**

What's Wrong?

- dynamic scope
- solution to implement lexical scope: each procedure needs to remember the variable environment in which it was created
- easiest modeling: restructure AST of blocks

blocks

$$b \quad ::= \quad \texttt{var}\ \overline{z};\ \texttt{proc}\ \overline{f(\overline{x})\{b\};}\ s \qquad \text{block}$$

Block

$$\frac{\overline{a} \cap dom(\mu) = \emptyset \qquad \sigma' = \sigma[\overline{z \mapsto a}] \qquad \pi' = \pi[\overline{f \mapsto \sigma', \pi', (\overline{x})\{b\}}] \qquad \mu[\overline{a \mapsto 0}], \sigma', \pi' \vdash s \hookrightarrow \mu'}{\mu, \sigma, \pi \vdash \texttt{var}\ \overline{z};\ \texttt{proc}\ \overline{f(\overline{x})\{b\}};\ s \hookrightarrow \mu'}$$

SCallBlock

$$\frac{\mu, \sigma \vdash \overline{e} \hookrightarrow \overline{y} \qquad \pi(f) = \sigma', \pi', (\overline{x})\{b\} \qquad \overline{a} \cap dom(\mu) = \emptyset \qquad \mu[\overline{a \mapsto y}], \sigma'[\overline{x \mapsto a}], \pi' \vdash b \hookrightarrow \mu'}{\mu, \sigma, \pi \vdash f(\overline{e}) \hookrightarrow \mu'}$$

Proper Implementation of Lexical Scope

- Environments σ and π implement **scope chains**.
- The environment σ' stored in the procedure map always contains the variable environment of the block in which the procedure was declared.
- The environment π' in the procedure map always contains the procedure environment of the block in which the procedure was declared.
- The entry in the procedure environment is **recursive** to enable mutually recursive procedure calls across nested procedures.

Scope Chain and Activation Blocks

```
var x;
proc p(x){... q() ...};
proc q(y){...};
... p(42) ...
```

scope chain

$\sigma_0 = [x \mapsto a_0]$

$\sigma_1 = \sigma_0[x \mapsto a_1]$

$\sigma_2 = \sigma_0[y \mapsto a_2]$

activation blocks

$\mu_0 = [a_0 \mapsto \dots]$

p()

$\mu_1 = \mu_0[a_1 \mapsto \dots]$

q()

$\mu_2 = \mu_1[a_2 \mapsto \dots]$