Programming Language Concepts, cs2104 Lecture 04 (2003-08-29)

Seif Haridi
Department of Computer Science, NUS
haridi@comp.nus.edu.sg

Overview

- Organization
- Course overview
- Introduction to programming concepts

Organization

Reading Suggestions

- Chapter 2
  - Sections 2.1 – 2.3 [careful]
  - Section 2.4 – 2.5 [careful]
  - Section 2.6 [careful]
- And of course the handouts!

Organizational

- First assignment is due on Monday
- We could have a consultation session on Saturday from 10:00-13:00 in PC Lab2?

Reading Suggestions

- Chapter 2
  - Sections 2.1 – 2.5 [careful]
- And of course the handouts!

Last Lecture

Our Roadmap

- Introduction FINISHED
  - Functions
  - Data structures
  - Recursion
  - Language syntax
- Declarative programming model
  - kernel language
  - language semantics

Languages

- Kernel language
  - few constructs
  - sequence of instructions (statements)
  - geared at execution by machine
  - simple semantics, simple explanation

- Programming language
  - many constructs useful for programming
  - nested structure
  - geared at programming by humans
  - concise programming, rich language

Kernel Language

- Statements
- Single assignment store
- Values and types
- Abstract machine
- Programming language translated to kernel language

Kernel Language Approach

- Provides useful abstractions for programmer
- Can be extended with linguistic abstractions

practical language

translation ↓

kernel language

- Easy to understand and reason with
- Has a precise (formal) semantics

```
fun {Sqr X} X*X end
B = {Sqr {Sqr A}}
```

↓

```
proc {Sqr X Y}
     { * X X Y}
end
local T in
     {Sqr A T}
     {Sqr T B}
end
```


Linguistic Abstractions ⇔ Syntactic Sugar

- Linguistic abstractions:
  provide higher level concepts
- Syntactic sugar:
  short cuts and conveniences to improve readability

```
if N==1 then [1]
else
   local L in
     ...
   end
end
```

```
if N==1 then [1]
else L in
   ...
end
```

Sequential Declarative Computation Model

- *Single assignment store*
  - declarative (dataflow) variables and values (together called entities)
  - values and their types
- *Kernel language syntax*
- *Environment*
  - maps textual variable names (variable identifiers) into entities in the store
- *Execution* of kernel language statements
  - execution stack of statements (defines control)
  - store
  - transforms store by sequence of steps

Language Syntax

- Defines *legal* programs
  - programs that can be executed by machine
- Defined by *grammar rules*
  - define how to make ‘sentences’ out of ‘words’
- For programming languages
  - sentences are called *statements*
  - words are called *tokens*
  - *grammar rules* describe tokens and statements

Grammar Rules Constructs

- ⟨*x*⟩ — nonterminal *x*
- ⟨*x*⟩ ::= *Body* — ⟨*x*⟩ is defined by *Body*
- ⟨*x*⟩ | ⟨*y*⟩ — either ⟨*x*⟩ or ⟨*y*⟩ (choice)
- ⟨*x*⟩ ⟨*y*⟩ — the sequence ⟨*x*⟩ followed by ⟨*y*⟩
- { ⟨*x*⟩ } — sequence of zero or more occurrences of ⟨*x*⟩
- { ⟨*x*⟩ }+ — sequence of one or more occurrences of ⟨*x*⟩
- [⟨*x*⟩] — zero or one occurrence of ⟨*x*⟩

Kernel Language Syntax

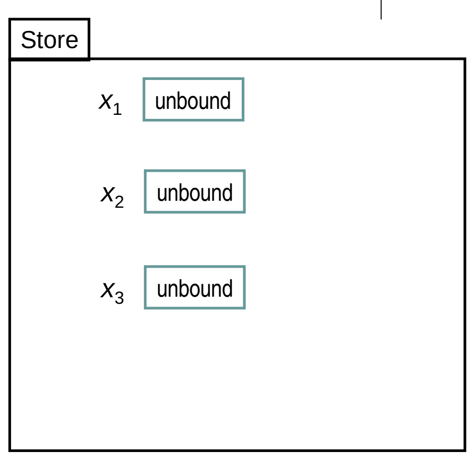
⟨s⟩ denotes a statement

⟨s⟩ ::=	skip	*empty statement*
|	⟨x⟩ = ⟨y⟩	*variable-variable binding*
|	⟨x⟩ = ⟨v⟩	*variable-value binding*
|	$\langle s_1\rangle \langle s_2\rangle$	*sequential composition*
|	local ⟨x⟩ in $\langle s_1\rangle$ end	*declaration*
|	if ⟨x⟩ then $\langle s_1\rangle$ else $\langle s_2\rangle$ end	*conditional*
|	{ ⟨x⟩ $\langle y_1\rangle \ldots \langle y_n\rangle$ }	*procedural application*
|	case ⟨x⟩ of ⟨pattern⟩ then $\langle s_1\rangle$ else $\langle s_2\rangle$ end	*pattern matching*
⟨v⟩ ::=	...	*value expression*
⟨pattern⟩ ::=	...	

Single Assignment Store

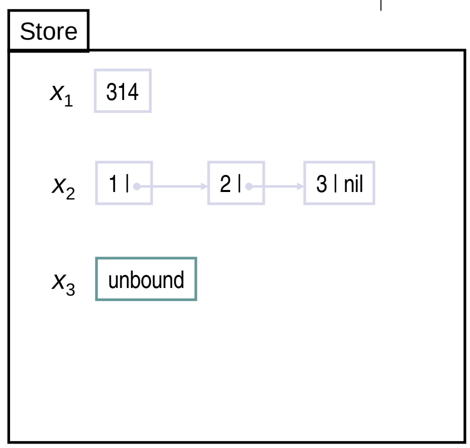
Single Assignment Store

- Single assignment store is store (set) of variables
- Initially variables are unbound



Declarative (Single-Assignment) Variables

- Created as being *unbound*
- Can be *bound* to exactly one value
- Once bound, stays bound
  - indistinguishable from its value

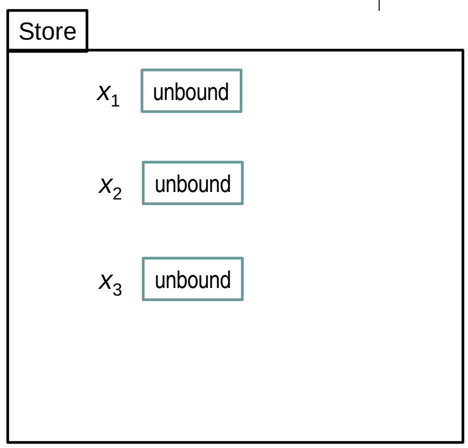


Store Operations

- Single assignment $\langle x \rangle = \langle v \rangle$
  - constructs value $\langle v \rangle$ in store
  - binds variable $\langle x \rangle$ to constructed value
  - if already bound, tests for compatibility
    - if not compatible, error raised
- Variable-variable binding $\langle x \rangle = \langle y \rangle$
  - binds variables to each other
  - variables form equivalence classes

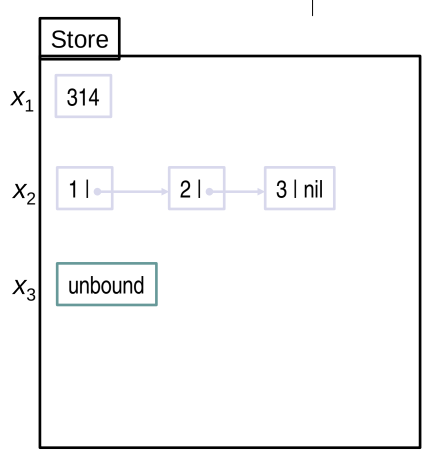
Single Assignment Store

- Single assignment store is store (set) of variables
- Initially variables are unbound
- Example: store with three variables, x_1, x_2, and x_3
- `{x1,x2,x3}`



Single Assignment Store (3)

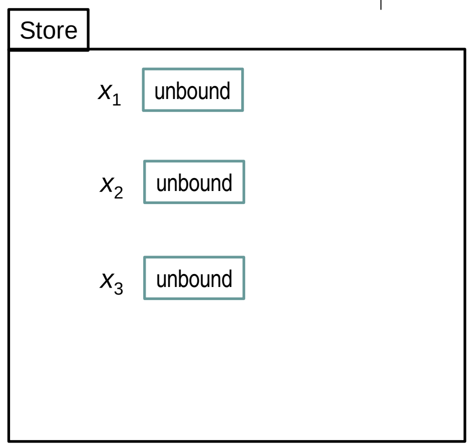
- Variables in store may be bound to values
- Assume we allow as values
- Example:
  - x_1 is bound to integer 314
  - x_2 is bound to list [1 2 3]
  - x_3 is still unbound
- `{x1=314,x2=[1 2 3],x3}`



Store Operations: Single Assignment

$\langle x \rangle = \langle v \rangle$

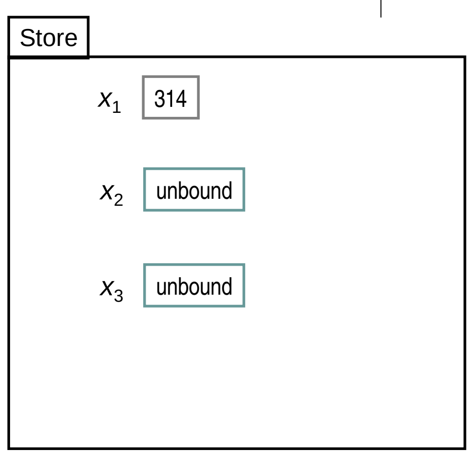
- x_1 = 314
- x_2 = [1 2 3]
- Assumes that $\langle x \rangle$ is unbound



Single Assignment

$\langle x \rangle = \langle \text{value} \rangle$

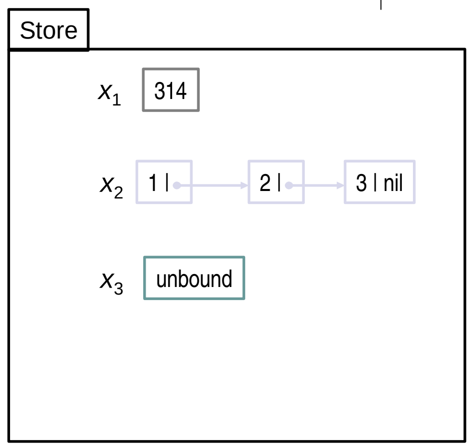
- **x_1 = 314**
- x2 = [1 2 3]



Single Assignment

⟨x⟩ = ⟨v⟩

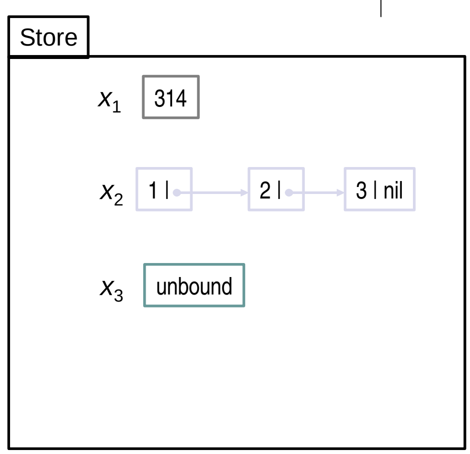
- x_1 = 314
- x_2 = [1 2 3]
- *Single assignment operation* (‘=‘)
  - constructs ⟨v⟩ in store
  - binds variable ⟨x⟩ to this value
- If variable already bound, operation tests compatibility of values
  - if test fails an error is raised



Single Assignment

⟨x⟩ = ⟨v⟩

- x_1 = 314
- x_2 = [1 2 3]
- *Single assignment operation* (‘=‘)
  - constructs ⟨v⟩ in store
  - binds variable ⟨x⟩ to this value
- {x_1=314,x_2=[1 2 3],x_3}

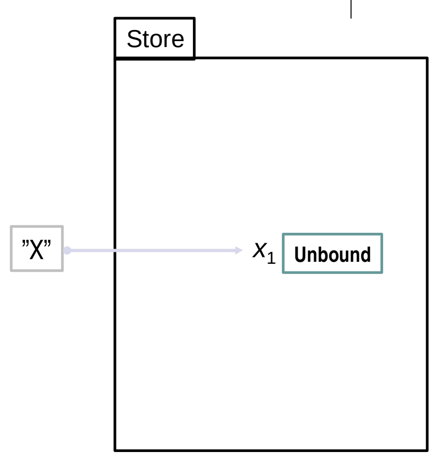


Single Assignment Store

- Can contain partial values
  - data structure with unbound variables
- Once variable is bound, indistinguishable from its value
  - *dereferencing*: traversing variable cell to get value
  - automatic and transparent to programmer

Variable Identifiers

- Refer to store entities
- Environment maps variable identifiers to variables
  - declare X
  - local X in ...
- ”X” is variable identifier
- Corresponds to ’environment’ {”X” $\rightarrow x_1$}

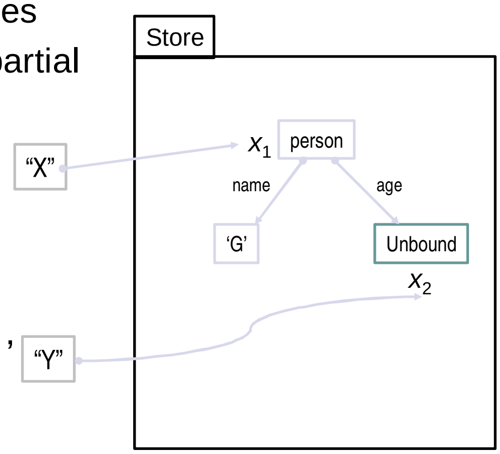


Partial Values

- Data structure that may contain unbound variables
- The store contains the partial value:

  person(name: 'G' age: x_2)
- declare Y X

  X = person(name: george age: Y)
- The identifier 'Y' refers to x_2
- `S={x1=person(name:'G' age:x2 ), x2 }`

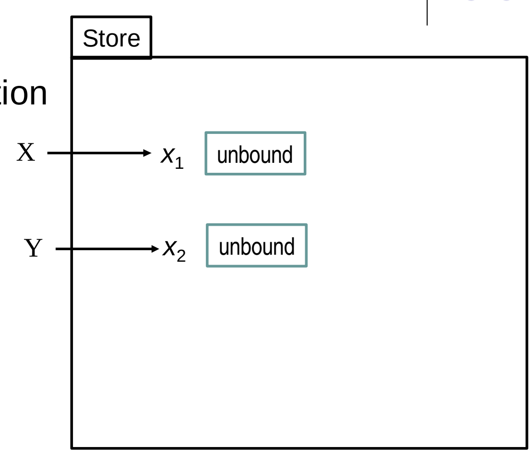


Variable-variable Binding

$\langle x_1 \rangle = \langle x_2 \rangle$

- Performs bind operation between variables
- Example:

  X = Y
- Operation equates (merges) the two variables
- S = $\{x_1, x_2\}$

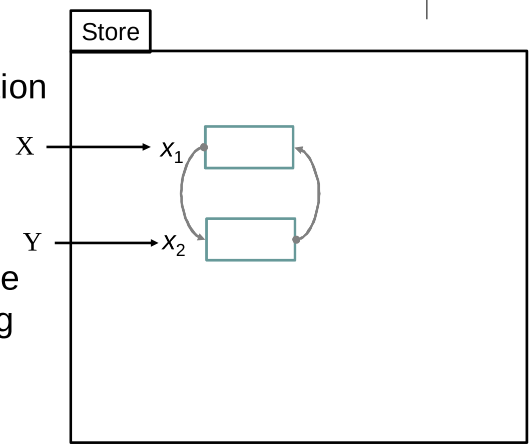


Variable-variable Binding

$\langle x_1 \rangle = \langle x_2 \rangle$

- Performs bind operation between variables
- Example:

  X = Y
- Operation equates the two variables: forming an equivalence class
- $S=\{x_1=x_2\}$



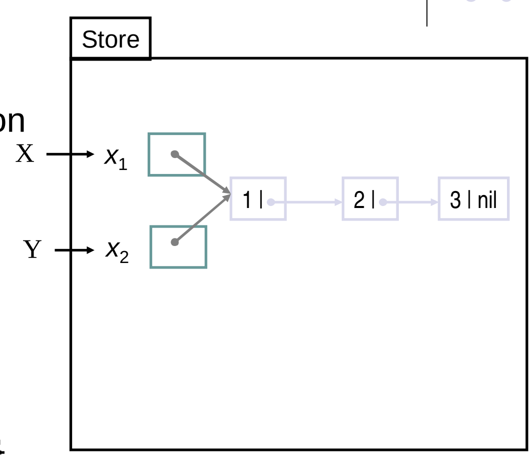
Variable-variable Binding

$\langle x_1 \rangle = \langle x_2 \rangle$

- Performs bind operation between variables
- Example:

  X = Y

  X = [1 2 3]
- All variables (X and Y) are bound to **[1 2 3]**
- `S={X1=X2=[1 2 3]}`



Summary: Variables and Partial Values

- Declarative variable
  - resides in single-assignment store
  - is initially unbound
  - can be bound to exactly one (partial) value
  - can be bound to several (partial) values as long as they are compatible with each other
- Partial value
  - data-structure that may contain unbound variables
  - when one of the variables is bound, it is replaced by the (partial) value it is bound to
  - a complete value, or value for short is a data-structure that does not contain any unbound variable

Kernel Language Syntax

Variable Identifiers

- ⟨*x*⟩ , ⟨*y*⟩, ⟨*z*⟩ stand for variables
- Concrete kernel language variables
  - begin with upper-case letter
  - followed by (possibly empty) sequence of alphanumeric characters or underscore
- Any sequence of characters within backquote
- Examples:
  - X, Y1
  - Hello_World
  - `hello this is a $5 bill` (backquote)

Kernel Language Syntax

⟨s⟩ denotes a statement

⟨s⟩	::=	skip	*empty statement*
	|	⟨*x*⟩ = ⟨y⟩	*variable-variable binding*
	|	⟨x⟩ = ⟨v⟩	*variable-value binding*
	|	⟨s_1⟩ ⟨s_2⟩	*sequential composition*
	|	local ⟨x⟩ in ⟨s_1⟩ end	*declaration*
	|	if ⟨x⟩ then ⟨s_1⟩ else ⟨s_2⟩ end	*conditional*
	|	{ ⟨x⟩ ⟨y_1⟩ ... ⟨y_n⟩ }	*procedural application*
	|	case ⟨x⟩ of ⟨pattern⟩ then ⟨s_1⟩ else ⟨s_2⟩ end	*pattern matching*
⟨v⟩	::=	...	*value expression*
⟨pattern⟩	::=	...	

Value Expressions

$\langle v \rangle$ ::= $\langle$record$\rangle$ | $\langle$number$\rangle$ | $\langle$procedure$\rangle$

$\langle$record$\rangle$,
$\langle$pattern$\rangle$ ::= $\langle$literal$\rangle$
| $\langle$literal$\rangle$ ($\langle$feature$_1\rangle$: $\langle x_1 \rangle$ … $\langle$feature$_n\rangle$: $\langle x_n \rangle$)

$\langle$literal$\rangle$ ::= $\langle$atom$\rangle$ | $\langle$bool$\rangle$
$\langle$feature$\rangle$ ::= $\langle$int$\rangle$ | $\langle$atom$\rangle$ | $\langle$bool$\rangle$

$\langle$bool$\rangle$ ::= true | false

$\langle$number$\rangle$ ::= $\langle$int$\rangle$ | $\langle$float$\rangle$

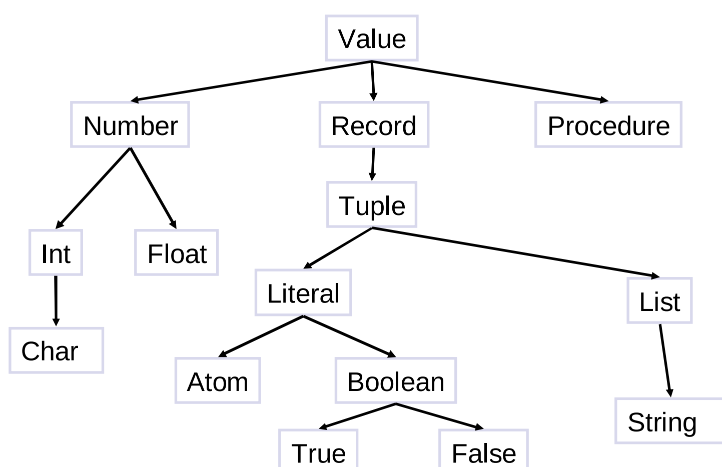
$\langle$procedure$\rangle$ ::= proc {\$ $\langle y_1 \rangle$ … $\langle y_n \rangle$} $\langle s \rangle$ end

Values and Types

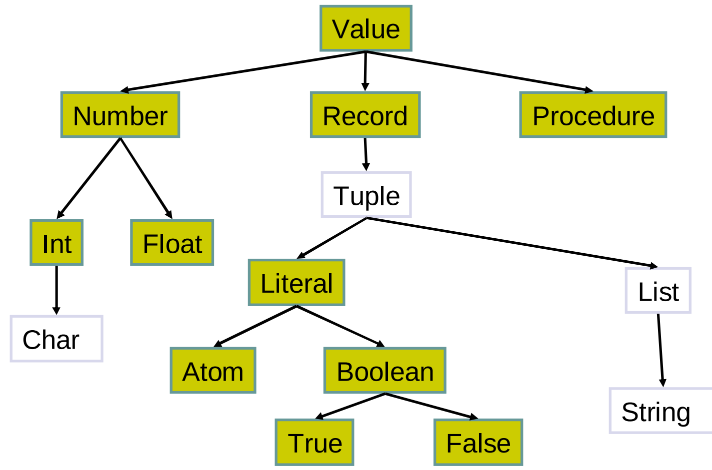
Values and Types

- *Data type*
  - set of values
  - set of associated operations
- Example: Int is data type ”Integer”
  - set of all integer values
  - 1 is *of type* Int
  - has set of operations including +,-,*,div, etc
- Model comes with a set of basic types
- Programs can define other types
  - for example: *abstract data types* ADT

Data Types



Primitive Data Types



Value Expressions

⟨v⟩ ::= ⟨record⟩ | ⟨number⟩ | ⟨procedure⟩

⟨record⟩,
⟨pattern⟩ ::= ⟨literal⟩
| ⟨literal⟩ (⟨feature$_1$⟩ : ⟨x$_1$⟩ … ⟨feature$_n$⟩ : ⟨x$_n$⟩)

⟨literal⟩ ::= ⟨atom⟩ | ⟨bool⟩
⟨feature⟩ ::= ⟨int⟩ | ⟨atom⟩ | ⟨bool⟩

⟨bool⟩ ::= true | false

⟨number⟩ ::= ⟨int⟩ | ⟨float⟩

⟨procedure⟩ ::= proc {$ ⟨y_1⟩ … ⟨y_n⟩} ⟨s⟩ end

Numbers

- Integers
  - 314, 0
  - ~10 (minus 10)
- Floats
  - 1.0, 3.4, 2.0e2, 2.0E2 (2×10^2)
- Number: either Integer or Float

Atoms and Booleans

- A sequence starting with a lower-case character followed by characters or digits, …
  - person, peter
  - 'Seif Haridi'
- Booleans
  - true
  - false
- Literal: atom or boolean

Records

- Compound representation (data-structures)
  - ⟨*l*⟩(⟨f_1⟩ : ⟨x_1⟩ … ⟨f_n⟩ : ⟨x_n⟩)
  - ⟨*l*⟩ is a literal
- Examples
  - person(age:X1 name:X2)
  - person(1:X1 2:X2)
  - 'l'(1:H 2:T)
- Syntactic sugar
  - tuples f(a b)
  - lists a|Xr [a b c]
  - pairs a#b

Strings

- Is list of character codes enclosed with double quotes
  - example "E=mc^2"
  - same as [69 61 109 99 94 50]

Procedure Declarations

- Kernel language

  ⟨x⟩ = proc {$ ⟨y_1⟩ … ⟨y_n⟩} ⟨s⟩ end

  is legal statement
  - binds ⟨x⟩ to procedure value
  - *declares* (introduces) a procedure

- Familiar syntactic variant

  ```
  proc {⟨x⟩ ⟨y1⟩ … ⟨yn⟩}
     ⟨s⟩
  end
  ```

  introduces (declares) the procedure ⟨x⟩

Operations on Basic Types

- Numbers
  - floats: +,-,*, /
  - integers: +,-,*,div, mod
- Records
  - Arity, Label, Width, and ”.”
  - X = person(name:”George” age:25)
  - {Arity X} = [age name]
  - {Label X} = person, X.age = 25
- Comparisons
  - equality: ==, \=
  - order: =<, <, >=

    integers, floats, and atoms

Value Expressions

⟨v⟩ ::= ⟨procedure⟩ | ⟨record⟩ | ⟨number⟩ | ⟨basicExpr⟩

⟨basicExpr⟩ ::= ... | ⟨numberExpr⟩ | ...

⟨numberExpr⟩ ::= $\langle x \rangle_1 + \langle x \rangle_2$ | ...

.....

Summary: Values and Types

- For kernel language
  - numbers
  - literals
  - records
  - procedures
- Created by value expressions

Abstract Machine (Semantics)

Concepts

- Single-assignment store
- Environment
- Semantic statement
- Execution state
- Computation

Abstract Machine

- Performs a computation
- *Computation* is sequence of execution states
- *Execution state*
  - stack of semantic statements
  - single assignment store
- *Semantic statement*
  - statement
  - environment
- *Environment* maps variable identifiers to store entities

Single Assignment Store

- Single assignment store σ
  - set of store variables
  - partitioned into
    - sets of variables that are equal but unbound
    - variables bound to value
- Example store $\{x_1, x_2{=}x_3, x_4{=}a|x_2\}$
  - x_1 unbound
  - x_2, x_3 equal and unbound
  - x_4 bound to partial value $a|x_2$

Environment

- Environment E
  - maps variable identifiers to entities in store σ
  - written as set of pairs $X \rightarrow x$
    - variable identifier X
    - store variable x
- Example environment $\{ X \rightarrow x, Y \rightarrow y \}$
  - maps identifier X to store variable x
  - maps identifier Y to store variable y

Environment and Store

- Given: environment E, store σ
- Looking up value for variable identifier X:
  - find store variable in environment $E(X)$
  - take value from σ for $E(X)$
- Example:

  $\sigma=\{x_1, x_2=x_3, x_4=a|x_2\}$ $E = \{ X \rightarrow x_1, Y \rightarrow x_4 \}$
  - $E(X) = x_1$ and no information in σ on x_1
  - $E(Y) = x_4$ and σ binds x_4 to $a|x_2$

Calculating with Environments

- Program execution looks up values
  - assume store σ
  - given variable identifier $\langle x \rangle$
  - $E(\langle x \rangle)$ is value in store σ
- Program execution modifies environments
  - for example: declaration
  - adding new mappings from identifiers
  - overwrite existing mappings
  - restricting mappings to sets of variables

Environment Adjunction

- Given: Environment E

  $$E + \{\langle x \rangle_1 \rightarrow x_1, \ldots, \langle x \rangle_n \rightarrow x_n\}$$

  is new environment E' with mappings added:
  - always take store entity from new mappings
  - might overwrite old mappings
- Obs: no name given in book!

Environment Projection

- Given: Environment E

  $E \mid \{\langle x\rangle_1, \ldots, \langle x\rangle_n\}$

  is new environment E' where only mappings for $\{\langle x\rangle_1, \ldots, \langle x\rangle_n\}$ are retained from E

Adjunction Example

- $E_0 = \{ Y \rightarrow 1 \}$
- $E_1 = E_0 + \{ X \rightarrow 2 \}$
  - corresponds to $\{ X \rightarrow 2, Y \rightarrow 1 \}$
  - $E_1(X) = 2$
- $E_2 = E_1 + \{ X \rightarrow 3 \}$
  - corresponds to $\{ X \rightarrow 3, Y \rightarrow 1 \}$
  - $E_2(X) = 3$

Why Adjunction?

E_0

```
local X in
   X = 2
   local X in
      X = 3
      {... Y}
   end
   ...
end
```

E_2 E_1

E_0

Semantic Statements

- To actually execute statement:
  - environment to map identifiers
    - modified with execution of each statement
    - each statement has its own environment
  - store to find values
    - all statements modify same store
    - single store
- Semantic statement **(⟨s⟩, *E*)**
  - pair of (statement, environment)

Stacks of Statements

- Execution maintains stack of semantic statements *ST*

  $[(\langle s \rangle_1, E_1), \ldots, (\langle s \rangle_n, E_n)]$

  - always topmost statement $(\langle s \rangle_1, E_1)$ executes first
  - rest of stack: what needs to be done

- Also called: *semantic stack*

Execution State

- *Execution state* $(\, ST, \sigma \,)$
  - pair of (stack of semantic statements, store)

- *Computation*

  $(ST_1, \sigma_1) \Rightarrow (ST_2, \sigma_2) \Rightarrow (ST_3, \sigma_3) \Rightarrow \ldots$

  - sequence of execution states

Program Execution

- Initial execution state

  ([(⟨s⟩,∅)] , ∅)

  - empty store ∅
  - stack with semantic statement [(⟨s⟩,∅)]
    - single statement ⟨s⟩, empty environment ∅
- At each execution step
  - pop topmost element of semantic stack
  - execute according to statement
- If semantic stack empty, execution stops

Semantic Stack States

- Semantic stack can be in run-time states
  - *terminated* stack is empty
  - *runnable* can do execution step
  - *suspended* stack not empty, no execution step possible
- Statements
  - *non-suspending* can always execute
  - *suspending* need values from store dataflow behavior

Summary

- Single assignment store — σ
- Environments — E
  - adjunction, projection — $E + \{\ldots\}$ $\quad E\,|_{\{\ldots\}}$
- Semantic statements — $(\langle s\rangle, E)$
- Semantic stacks — $[(\langle s\rangle, E)\ \ldots\ ]$
- Execution state — (ST, σ)
- Program execution
  - runnable, terminated, suspended
- Statements
  - suspending, non-suspending

Statement Execution

Plan

- Simple statements
  - skip and sequential composition
  - variable declaration
  - store manipulation
  - conditional
- Computing with procedures (next lecture)
  - lexical scoping
  - closures
  - procedures as values
  - procedure call

Simple Statements

⟨s⟩ denotes a statement

⟨s⟩	::=	skip	*empty statement*
	|	⟨x⟩ = ⟨y⟩	*variable-variable binding*
	|	⟨x⟩ = ⟨v⟩	*variable-value binding*
	|	$\langle s_1 \rangle \langle s_2 \rangle$	*sequential composition*
	|	local ⟨x⟩ in $\langle s_1 \rangle$ end	*declaration*
	|	if ⟨x⟩ then $\langle s_1 \rangle$ else $\langle s_2 \rangle$ end	*conditional*
⟨v⟩	::=	...	*value expression* (no records here)

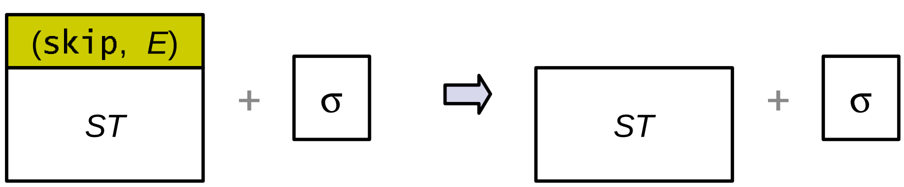
Executing `skip`

- Execution of semantic statement

  (`skip`, *E*)

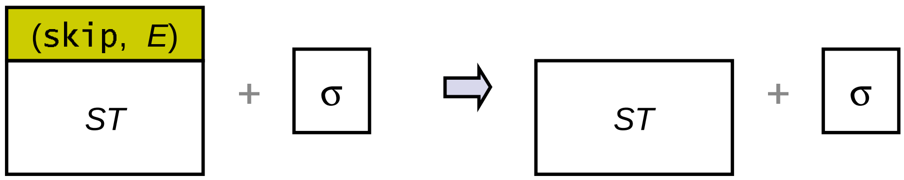
- Do nothing
  - means: continue with next statement
  - non-suspending statement

`skip`



- No effect on store σ
- Non-suspending statement

skip



- Remember: topmost statement is always popped!

Executing Sequential Composition

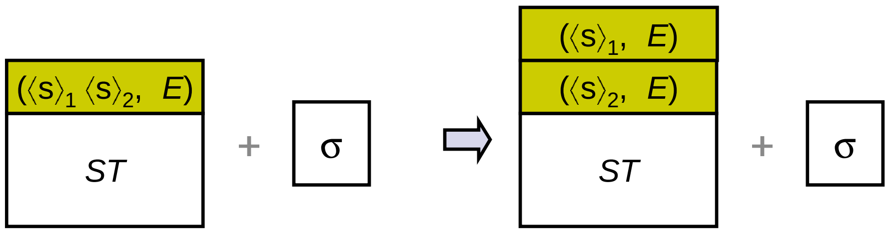
- Semantic statement is

  $(\langle s \rangle_1 \; \langle s \rangle_2, E)$

- Push in following order
  - $\langle s \rangle_2$ executes after
  - $\langle s \rangle_1$ executes next

- Statement is non-suspending

Sequential Composition



- Decompose statement sequences
  - environment is given to both statements

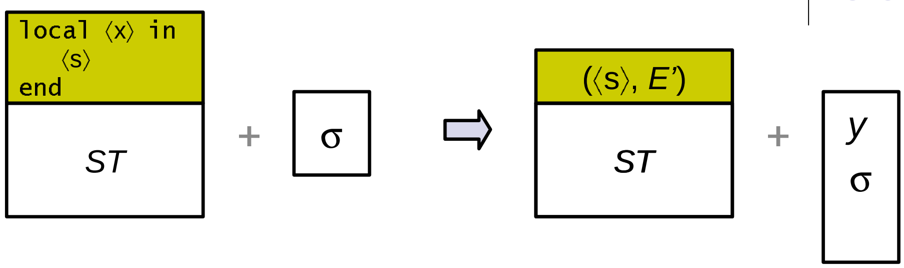
Executing `local`

- Semantic statement is
  (`local` ⟨x⟩ `in` ⟨s⟩ `end`, *E*)

- Execute as follows
  - create new variable *y* in store
  - create new environment *E*’ = *E* + {⟨x⟩ → *y*}
  - push (⟨s⟩, *E*’)

- Statement is non-suspending

`local`



- With $E' = E + \{\langle x\rangle \rightarrow y\}$

Variable-variable equality

- Semantic statement is
  $(\langle x\rangle = \langle y\rangle, E)$
- Execute as follows
  - bind $E(\langle x\rangle)$ and $E(\langle y\rangle)$ in store
- Statement is non-suspending

Variable-value equality

- Semantic statement is

  (⟨x⟩ = ⟨v⟩, *E*)

  with ⟨v⟩ number or record
- Execute as follows
  - create value ⟨v⟩ in store
    - use variables as defined by *E*
  - bind *E*(⟨x⟩) and ⟨v⟩ in store
- Statement is non-suspending

Variable-value equality

- Semantic statement is

  (⟨x⟩ = ⟨v⟩, *E*)

  with ⟨v⟩ number or record
- Execute as follows
  - create value ⟨v⟩ in store
    - use variables as defined by *E*
  - bind *E*(⟨x⟩) and ⟨v⟩ in store
- Statement is non-suspending

procedures?
…next lecture

Suspending Statements

- All statements so far can always execute
  - non-suspending (or immediate)
- Conditional?
  - requires condition ⟨x⟩ to be bound variable
  - *activation condition*: ⟨x⟩ is bound (determined)

Executing `if`

- Semantic statement is
  (`if` ⟨x⟩ `then` $\langle s\rangle_1$ `else` $\langle s\rangle_2$ `end`, *E*)
- If activation condition "⟨x⟩ bound" true
  - if *E*(⟨x⟩) bound to `true` push $\langle s\rangle_1$
  - if *E*(⟨x⟩) bound to `false` push $\langle s\rangle_2$
  - otherwise, raise error
- Otherwise, suspend…

An Example

```
local X in
   local B in
      B=true
      if B then X=1 else skip end
   end
end
```


Example: Initial State

```
([(local X in
      local B in
         B=true
         if B then X=1 else skip end
      end
   end, ∅)],
 ∅)
```

- Start with empty store and empty environment

Example: `local`

```
([(local B in
      B=true
      if B then X=1 else skip end
    end,
    {X → x})],
 {x})
```

- Create new store variable *x*
- Continue with new environment

Example: `local`

```
([(B=true
   if B then X=1 else skip end
   ,
   {B → b, X → x})],
 {b,x})
```

- Create new store variable *b*
- Continue with new environment

Example: Sequential Composition

```
([(B=true, {B → b, X → x}),
  (if B then X=1
   else skip end, {B → b, X → x})],
 {b,x})
```

- Decompose to two statements
- Stack has now two semantic statements

Example: Variable-Value Assignment

```
([(if B then X=1
   else skip end, {B → b, X → x})],
 {b=true, x})
```

- Environment maps `B` to *b*
- Bind *b* to true

Example: `if`

`([(X=1, {B` → *b*, `X` → *x*`})],`
 `{`*b*`=true`, *x*`})`

- Environment maps `B` to *b*
- Store binds *b* to `true`, continue with then

Example: Variable-Value Assignment

`([],`
 `{`*b*`=true`, *x*`=1})`

- Environment maps `X` to *x*
- Binds *x* to `1`
- Computation terminates as stack is empty

Summary

- Semantic statement execute by
  - popping itself — always
  - creating environment — `local`
  - manipulating store — `local, =`
  - pushing new statements — `local, if`, sequential composition
- Semantic statement can suspend
  - activation condition
  - read store

Homework

- Be an abstract machine!
- Execute something yourself!
- RTFB!

- See you next week!

Our Roadmap

- Single assignment store
- Kernel language syntax
- Values and types
- Environments
- Execution

Kernel Language Syntax

Kernel Language Syntax

⟨s⟩ denotes a statement

⟨s⟩	::=	skip	*empty statement*
	|	⟨x⟩ = ⟨y⟩	*variable-variable binding*
	|	⟨x⟩ = ⟨v⟩	*variable-value binding*
	|	$\langle s_1 \rangle \langle s_2 \rangle$	*sequential composition*
	|	local ⟨x⟩ in $\langle s_1 \rangle$ end	*declaration*
	|	if ⟨x⟩ then $\langle s_1 \rangle$ else $\langle s_2 \rangle$ end	*conditional*
	|	{ ⟨x⟩ $\langle y_1 \rangle$ … $\langle y_n \rangle$ }	*procedural application*
	|	case ⟨x⟩ of ⟨pattern⟩ then $\langle s_1 \rangle$ else $\langle s_2 \rangle$ end	*pattern matching*
⟨v⟩	::=	...	*value expression*
⟨pattern⟩	::=	...	

2003-09-05 S. Haridi, CS2104, L04 (slides: C. Schulte, S. Haridi) 95