5. Name Analysis

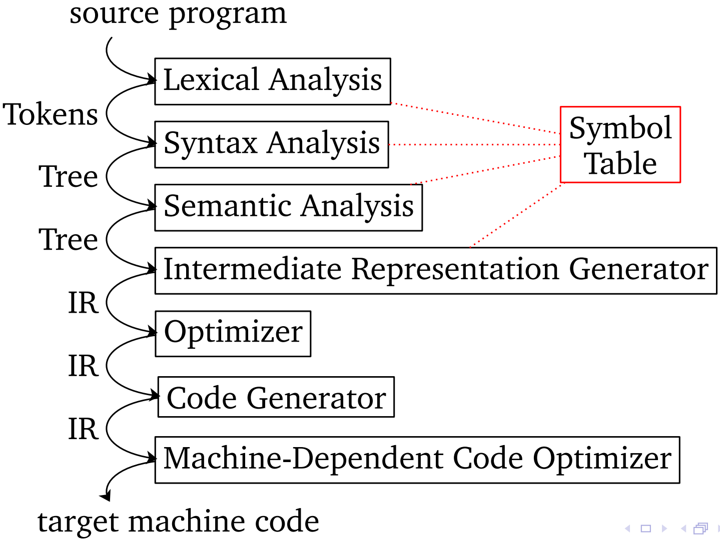
Eva Rose Kristoffer Rose

NYU Courant Institute

Compiler Construction (CSCI-GA.2130-001)
`http://cs.nyu.edu/courses/fall14/CSCI-GA.2130-001/lecture-5.pdf`

October 2, 2014

Context



source program

Lexical Analysis

Tokens

Syntax Analysis

Tree

Semantic Analysis

Tree

Intermediate Representation Generator

IR

Optimizer

IR

Code Generator

IR

Machine-Dependent Code Optimizer

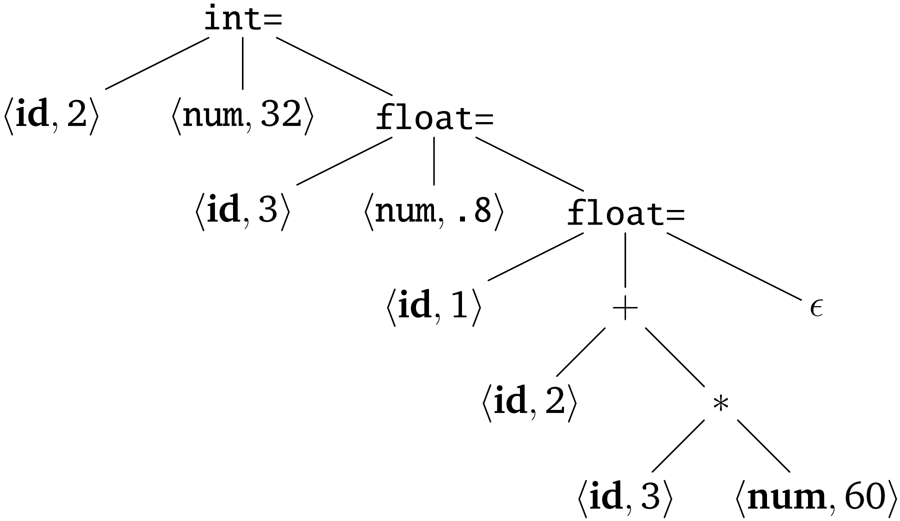
target machine code

Symbol Table

Example Code

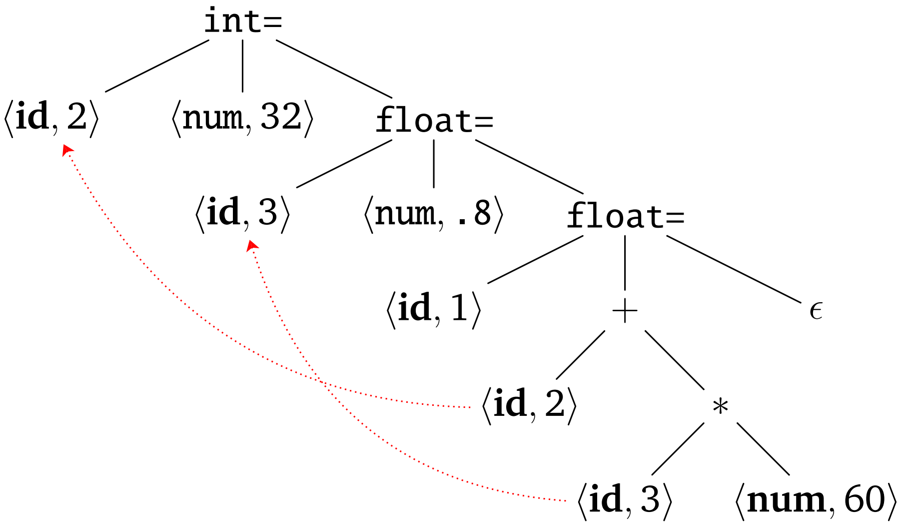
```
int initial = 32;
float rate = .8;
float position = initial + rate * 8;
```

Example Abstract Syntax Tree (AST)



id	*lexeme*
1	`position`
2	`initial`
3	`rate`

Example Abstract Syntax Tree (AST) + “def-use”



id	*lexeme*
1	`position`
2	`initial`
3	`rate`

1. Introduction
2. Programming Language Basics
3. Symbol Tables = Environments
4. HACS
5. Project Milestone 1

What’s in a Name?

Identifier	name that identifies an entity
Variable	abstract notion referring to particular storage location

What's in a Name?

Identifier	name that identifies an entity
Variable	abstract notion referring to particular storage location

What Is Known?

Declaration gives type (*etc.*) of name.

Definition gives value (*etc.*) of name.

What Is Known?

Declaration gives type (*etc.*) of name.

Definition gives value (*etc.*) of name.

What to Call?

Procedure	any callable entity.
Function	callable entity that “returns” a value.
Method	callable entity tied to class or object.

What to Call?

Procedure	any callable entity.
Function	callable entity that “returns” a value.
Method	callable entity tied to class or object.

What to Call?

Procedure	any callable entity.
Function	callable entity that “returns” a value.
Method	callable entity tied to class or object.

What is Passed?

Actual parameters the values that occur in a call.

Formal parameters the variables used to refer to the parameters inside procedure.

What is Passed?

Actual parameters the values that occur in a call.

Formal parameters the variables used to refer to the parameters inside procedure.

How is it Passed?

Call-by-Value actual parameter values computed before call.

Call-by-Reference actual parameter must be variable which is aliased with formal parameter.

Call-by-Name actual parameter text executed in context of formal parameter.

Call-by-Need like Call-by-Value but evaluation delayed until first use.

Lazy just evaluate sufficiently to create observed parts of data structures.

How is it Passed?

Call-by-Value actual parameter values computed before call.

Call-by-Reference actual parameter must be variable which is aliased with formal parameter.

Call-by-Name actual parameter text executed in context of formal parameter.

Call-by-Need like Call-by-Value but evaluation delayed until first use.

- **Lazy** just evaluate sufficiently to create observed parts of data structures.

How is it Passed?

Call-by-Value actual parameter values computed before call.

Call-by-Reference actual parameter must be variable which is aliased with formal parameter.

Call-by-Name actual parameter text executed in context of formal parameter.

Call-by-Need like Call-by-Value but evaluation delayed until first use.

- **Lazy** just evaluate sufficiently to create observed parts of data structures.

How is it Passed?

Call-by-Value actual parameter values computed before call.

Call-by-Reference actual parameter must be variable which is aliased with formal parameter.

Call-by-Name actual parameter text executed in context of formal parameter.

Call-by-Need like Call-by-Value but evaluation delayed until first use.

Lazy just evaluate sufficiently to create observed parts of data structures.

How is it Passed?

Call-by-Value	actual parameter values computed before call.
Call-by-Reference	actual parameter must be variable which is aliased with formal parameter.
Call-by-Name	actual parameter text executed in context of formal parameter.
Call-by-Need	like Call-by-Value but evaluation delayed until first use.
Lazy	just evaluate sufficiently to create observed parts of data structures.

Static *vs* Dynamic

Declarations Address is fixed *vs* runtime allocated.

Classes Shared *vs* per instance.

Scopes Tied to program “blocks” (lexical) *vs* runtime stack.

Static *vs* Dynamic

Declarations Address is fixed *vs* runtime allocated.

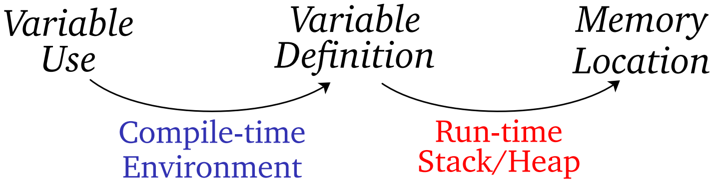
Classes Shared *vs* per instance.

Scopes Tied to program “blocks” (lexical) *vs* runtime stack.

Static *vs* Dynamic

Declarations	Address is fixed *vs* runtime allocated.
Classes	Shared *vs* per instance.
Scopes	Tied to program “blocks” (lexical) *vs* runtime stack.

Compile time *vs* Runtime Values



Variable Use

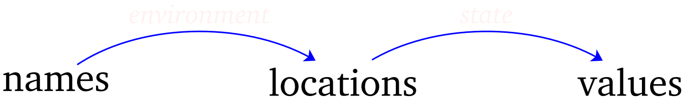
Variable Definition

Memory Location

Compile-time Environment

Run-time Stack/Heap

Environment *vs* State



environment

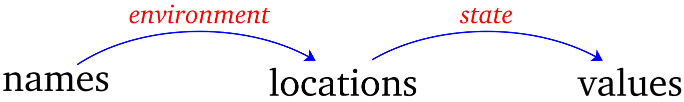
state

names

locations

values

Environment *vs* State



environment

state

names

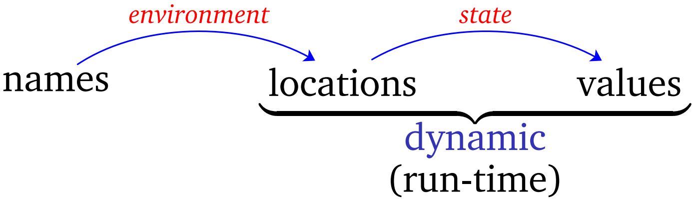
locations

values

Environment *vs* State

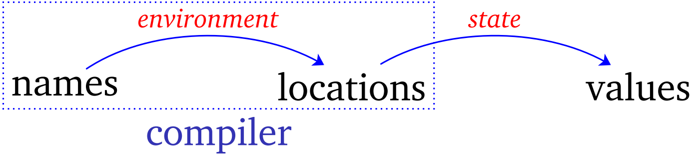
$$\underbrace{\text{names} \xrightarrow{\textit{environment}} \text{locations}}_{\substack{\text{static} \\ \text{(compile-time)}}} \xrightarrow{\textit{state}} \text{values}$$

Environment *vs* State



$$\text{names} \xrightarrow{\textit{environment}} \underbrace{\text{locations} \xrightarrow{\textit{state}} \text{values}}_{\substack{\text{dynamic} \\ \text{(run-time)}}}$$

Environment *vs* State



environment

state

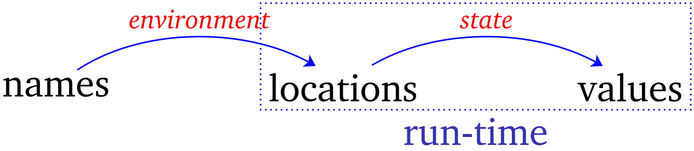
names

locations

values

compiler

Environment *vs* State



Focus of Class. . .

We are concerned with. . .

- Static scoping.
- Environments.
- Planning runtime state.

Focus of Class. . .

We are concerned with. . .

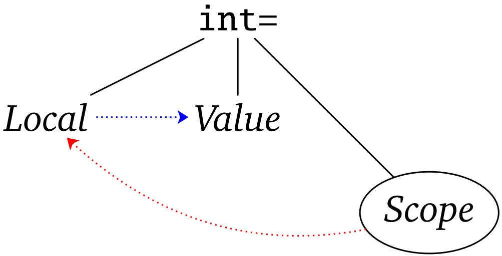
- Static scoping.
- Environments.
- Planning runtime state.

Focus of Class. . .

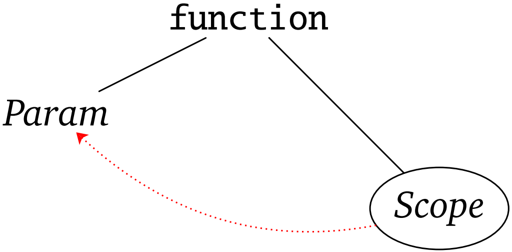
We are concerned with. . .

- Static scoping.
- Environments.
- Planning runtime state.

Static (Lexical) Scoping



Formal Parameters



Exercise 1.6.1

```
int w, x, y, z;
int i = 4; int j = 5;
{
    int j = 7; i = 6; w = i + j;
}
x = i + j;
{
    int i = 8; y = i + j;
}
z = i + j;
```

Exercise 1.6.4

```
#define a (x+1)
int x = 2;

void b() {
    x = a; printf("%d\n", x);
}
void c() {
    int x = 1; printf("%d\n", a);
}
void main() { b(); c(); }
```

Symbol Tables

- Traditional method for managing binders in system.
- Logically one symbol table per scope.
    - Really a tree: common ancestor blocks can be shared.
    - Can get messy if hash-table is used.
- Interferes with semantic rules/actions.
- *We shall fix this!* It is a little different from the book.

Symbol Tables

- Traditional method for managing binders in system.
- Logically one symbol table per scope.
  - Really a tree: common ancestor blocks can be shared.
  - Can get messy if hash-table is used.
- Interferes with semantic rules/actions.
- *We shall fix this!* It is a little different from the book.

Symbol Tables

- Traditional method for managing binders in system.
- Logically one symbol table per scope.
    - Really a tree: common ancestor blocks can be shared.
    - Can get messy if hash-table is used.
- Interferes with semantic rules/actions.
- *We shall fix this!* It is a little different from the book.

Symbol Tables

- Traditional method for managing binders in system.
- Logically one symbol table per scope.
  - Really a tree: common ancestor blocks can be shared.
  - Can get messy if hash-table is used.
- Interferes with semantic rules/actions.
- *We shall fix this!* It is a little different from the book.

Symbol Tables

- Traditional method for managing binders in system.
- Logically one symbol table per scope.
  - Really a tree: common ancestor blocks can be shared.
  - Can get messy if hash-table is used.
- Interferes with semantic rules/actions.
- *We shall fix this!* It is a little different from the book.

Symbol Tables

- Traditional method for managing binders in system.
- Logically one symbol table per scope.
    - Really a tree: common ancestor blocks can be shared.
    - Can get messy if hash-table is used.
- Interferes with semantic rules/actions.
- *We shall fix this!* It is a little different from the book.

HACS is *Higher-order* Attribute Contraction Schemes

- Traditional:

$$P \to S^*$$

$$S \to \texttt{int}\ V = E\,\texttt{;} \mid \texttt{print}\ V\,\texttt{;}$$

- Combine Scoping and Grammar:

$$P \to S$$

$$S \to \texttt{int}\ V = E\,\texttt{;}\ S \mid \texttt{print}\ V\,\texttt{;}\ S \mid \epsilon$$

HACS is *Higher-order* Attribute Contraction Schemes

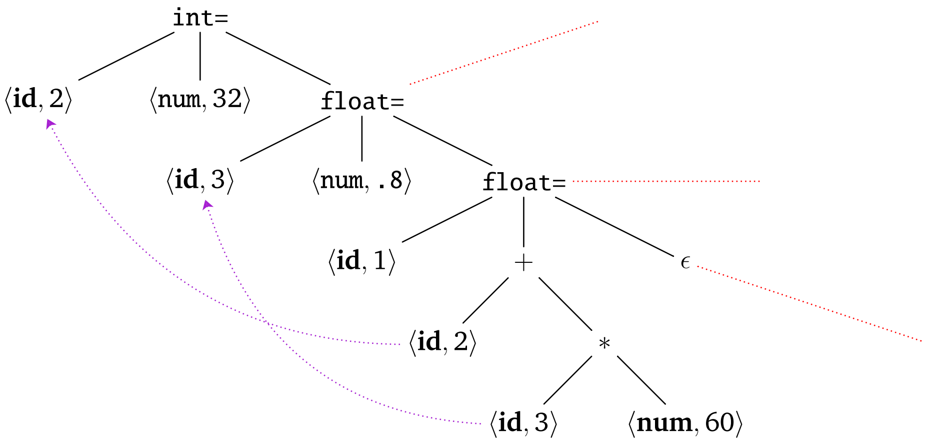
- Traditional:

$$\begin{aligned} P &\to S^* \\ S &\to \texttt{int}\ V = E\,;\ \mid\ \texttt{print}\ V\,; \end{aligned}$$

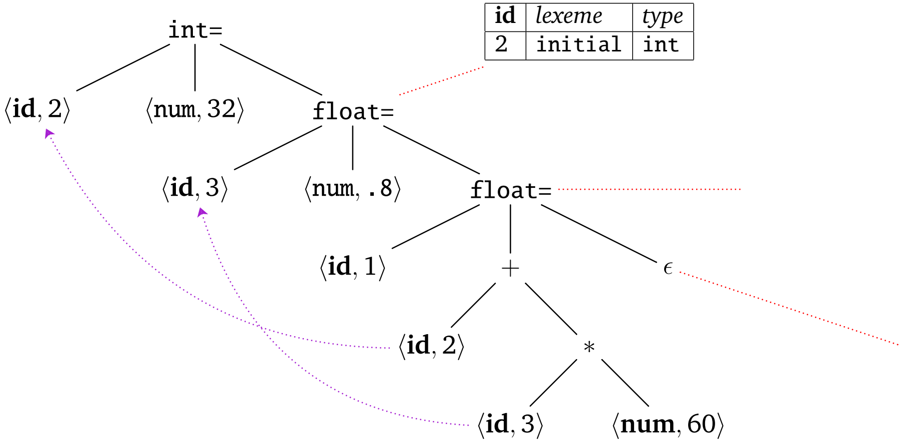
- Combine Scoping and Grammar:

$$\begin{aligned} P &\to S \\ S &\to \texttt{int}\ V = E\,;\ S\ \mid\ \texttt{print}\ V\,;\ S\ \mid\ \epsilon \end{aligned}$$

Environment Example

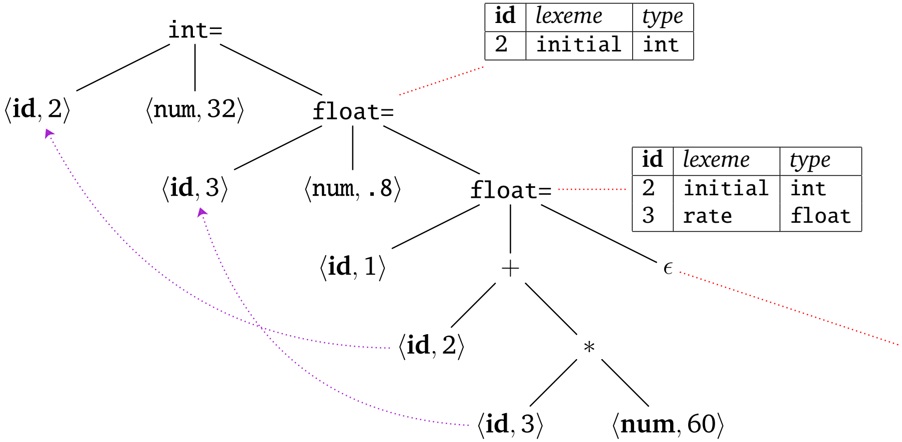


Environment Example



id	*lexeme*	*type*
2	`initial`	`int`

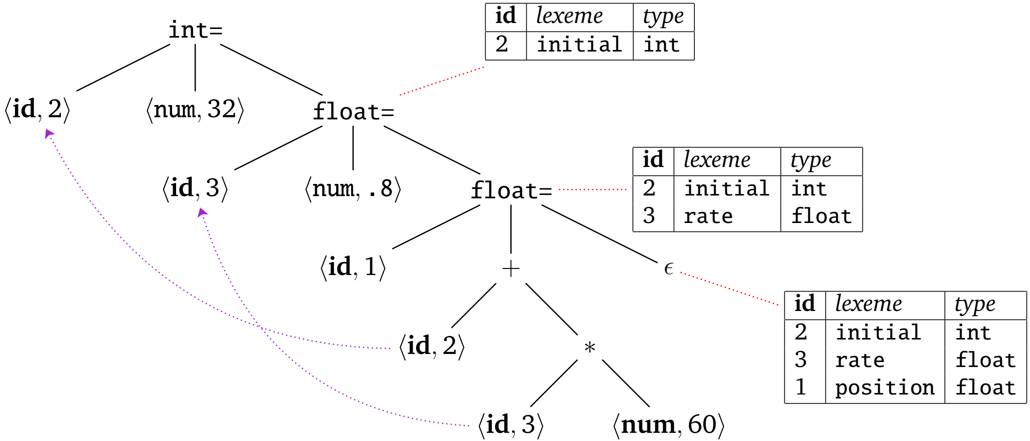
Environment Example



id	*lexeme*	*type*
2	`initial`	`int`

id	*lexeme*	*type*
2	`initial`	`int`
3	`rate`	`float`

Environment Example

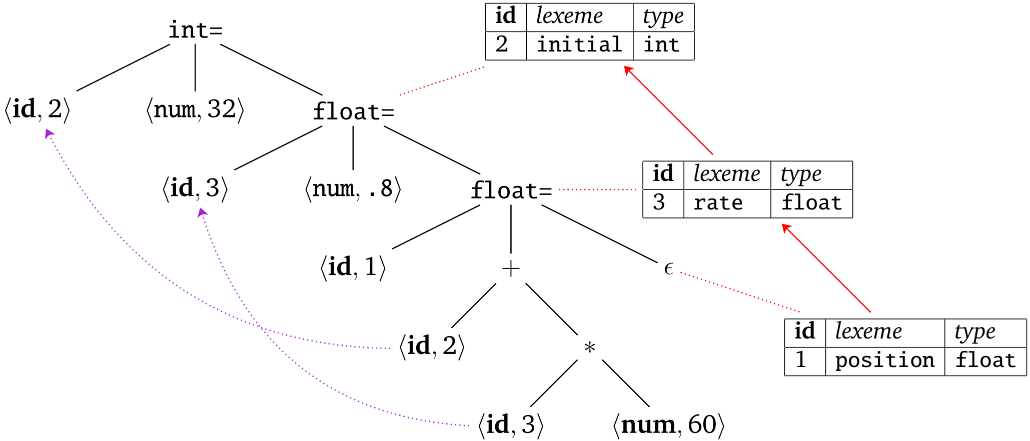


id	*lexeme*	*type*
2	`initial`	`int`

id	*lexeme*	*type*
2	`initial`	`int`
3	`rate`	`float`

id	*lexeme*	*type*
2	`initial`	`int`
3	`rate`	`float`
1	`position`	`float`

Environment Example with *Stack*

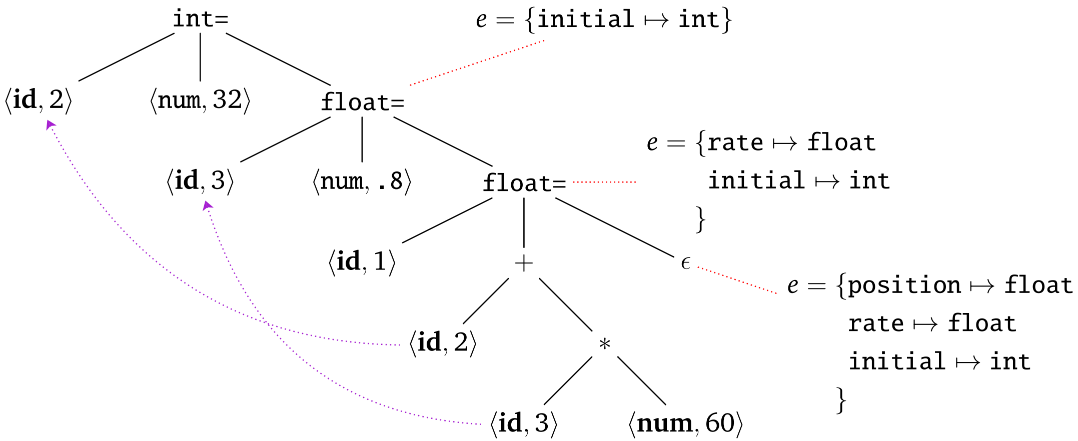


id	*lexeme*	*type*
2	`initial`	`int`

id	*lexeme*	*type*
3	`rate`	`float`

id	*lexeme*	*type*
1	`position`	`float`

Environment Example with *Maps*

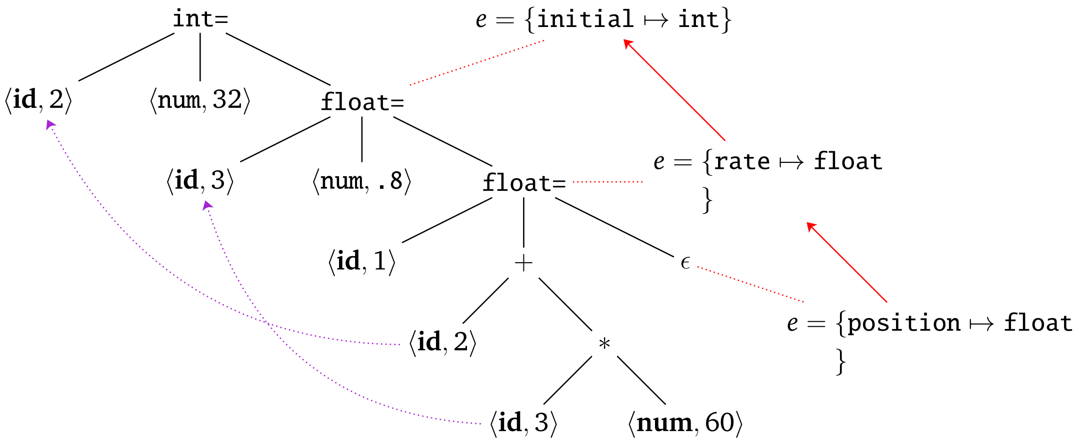


$$e = \{\texttt{initial} \mapsto \texttt{int}\}$$

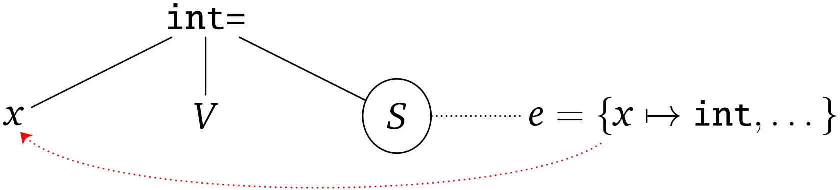
$$e = \{\texttt{rate} \mapsto \texttt{float} \quad \texttt{initial} \mapsto \texttt{int}\}$$

$$e = \{\texttt{position} \mapsto \texttt{float} \quad \texttt{rate} \mapsto \texttt{float} \quad \texttt{initial} \mapsto \texttt{int}\}$$

Environment Example with *Stack of Maps*

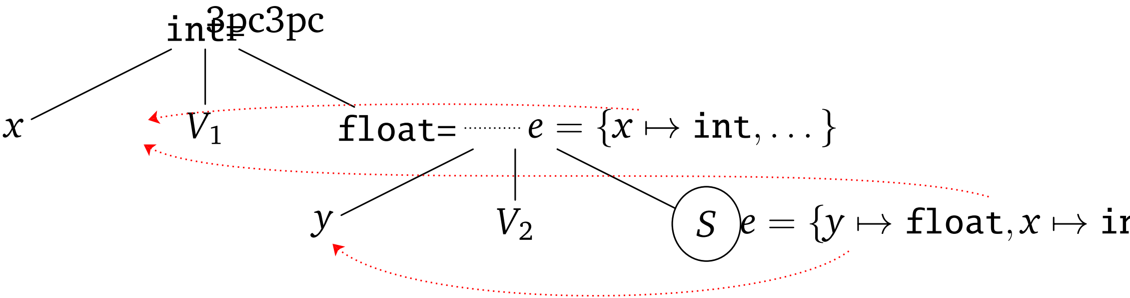


Binding Construct *with Local Symbol Table = Environment*



$e = \{x \mapsto$ `int`$, \ldots\}$

Binding Construct *with Local Symbol Table = Environment* II



`int=`3pc3pc

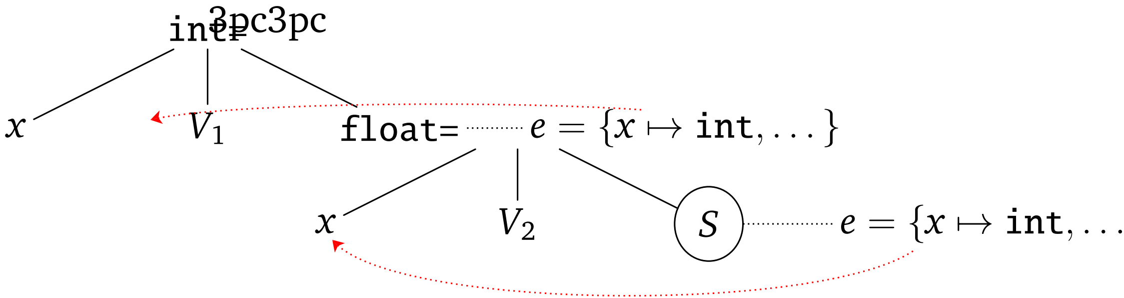
x V_1 `float=` $e = \{x \mapsto \texttt{int}, \ldots\}$

y V_2 S $e = \{y \mapsto \texttt{float}, x \mapsto \texttt{i}$

Shadowing

```
int x = 32;
int y;
{
  float x = .8;
  float y = x + x * 8;
}
y = y + x;
```

Binding Construct *with Local Symbol Table = Environment* III



Example

PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} := E_1; S_2$	$E_1.e = S.e;\; S_2.e = \text{Extend}(S.e, \mathbf{id}.sym, E_1.t)$
$\mid \{\ S_1\ \}\ S_2$	$S_1.e = S.e;\; S_2.e = S.e$
$\mid \epsilon$	
$E \rightarrow E_1 + T_2$	$E_1.e = E.e;\; T_2.e = E.e;\; E.t = \text{Unif}(E_1.t, T_2.t)$
$\mid T_1$	$T_1.e = E.e;\; E.t = T_1.t$
$T \rightarrow T_1 * F_2$	$T_1.e = T.e;\; F_2.e = T.e;\; T.t = \text{Unif}(T_1.t, F_2.t)$
$\mid F_1$	$F_1.e = T.e;\; T.t = F_1.t$
$F \rightarrow \mathbf{id}$	$F.t = \text{Lookup}(F.e, \mathbf{id}.sym)$
$\mid \mathbf{int}$	$E.t = \text{Int}$
$\mid \mathbf{float}$	$E.t = \text{Float}$

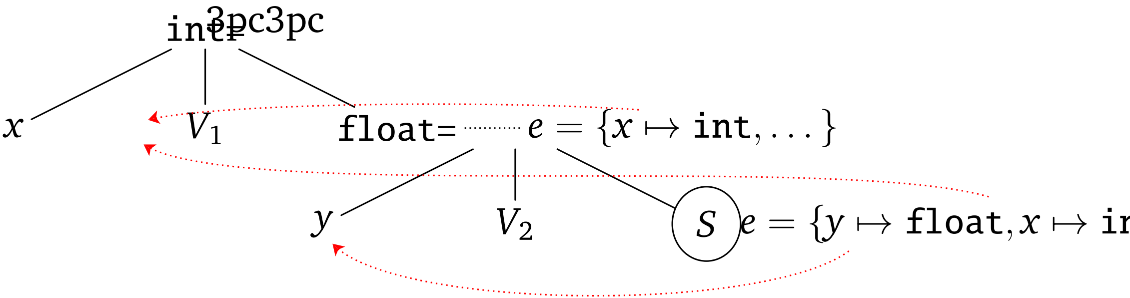
Example

PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} := E_1 ; S_2$	$E_1.e = S.e;\ S_2.e = \mathrm{Extend}(S.e, \mathbf{id}.sym, E_1.t)$
$\mid \{\ S_1\ \}\ S_2$	$S_1.e = S.e;\ S_2.e = S.e$
$\mid \epsilon$	
$E \rightarrow E_1 + T_2$	$E_1.e = E.e;\ T_2.e = E.e;\ E.t = \mathrm{Unif}(E_1.t, T_2.t)$
$\mid T_1$	$T_1.e = E.e;\ E.t = T_1.t$
$T \rightarrow T_1 * F_2$	$T_1.e = T.e;\ F_2.e = T.e;\ T.t = \mathrm{Unif}(T_1.t, F_2.t)$
$\mid F_1$	$F_1.e = T.e;\ T.t = F_1.t$
$F \rightarrow \mathbf{id}$	$F.t = \mathrm{Lookup}(F.e, \mathbf{id}.sym)$
$\mid \mathbf{int}$	$E.t = \mathrm{Int}$
$\mid \mathbf{float}$	$E.t = \mathrm{Float}$

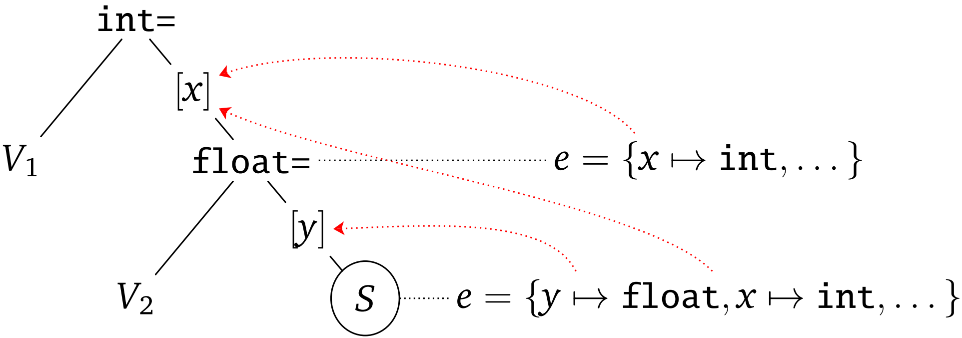
Example

PRODUCTION	SEMANTIC RULES
$S \to \mathbf{id} := E_1 ; S_2$	$E_1.e = S.e;\ S_2.e = \mathrm{Extend}(S.e, \mathbf{id}.sym, E_1.t)$
$\mid \{\ S_1\ \}\ S_2$	$S_1.e = S.e;\ S_2.e = S.e$
$\mid \epsilon$	
$E \to E_1 + T_2$	$E_1.e = E.e;\ T_2.e = E.e;\ E.t = \mathrm{Unif}(E_1.t, T_2.t)$
$\mid T_1$	$T_1.e = E.e;\ E.t = T_1.t$
$T \to T_1 * F_2$	$T_1.e = T.e;\ F_2.e = T.e;\ T.t = \mathrm{Unif}(T_1.t, F_2.t)$
$\mid F_1$	$F_1.e = T.e;\ T.t = F_1.t$
$F \to \mathbf{id}$	$F.t = \mathrm{Lookup}(F.e, \mathbf{id}.sym)$
$\mid \mathbf{int}$	$E.t = \mathrm{Int}$
$\mid \mathbf{float}$	$E.t = \mathrm{Float}$

Binding Construct



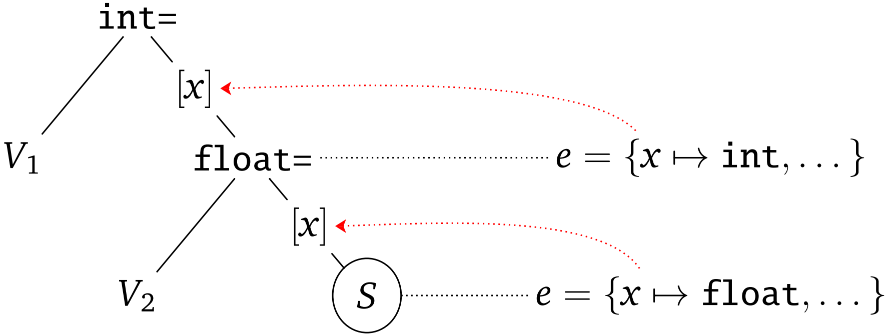
Binding Construct *à la HACS*



$e = \{x \mapsto \texttt{int}, \ldots\}$

$e = \{y \mapsto \texttt{float}, x \mapsto \texttt{int}, \ldots\}$

Binding Construct *à la HACS*



HACS is *Higher-order* Attribute Contraction Schemes II

$$
\begin{aligned}
P &\rightarrow S \\
S &\rightarrow \texttt{int}\ V_{x} = E\texttt{;}\ S^{x} \mid \texttt{print}\ V\texttt{;}\ S \mid \epsilon
\end{aligned}
$$

```
sort V | symbol ⟦⟨ID⟩⟧ ;

sort P | ⟦⟨S⟩⟧ ;

sort S | ⟦ int ⟨V binds x⟩ = ⟨E⟩; ⟨S[x as V]⟩ ⟧
       | ⟦ print ⟨V⟩; ⟨S⟩ ⟧
       | ⟦ ⟧ ;
```

HACS is *Higher-order* Attribute Contraction Schemes II

$$P \to S$$

$$S \to \texttt{int}\ V_x = E;\ S^x \mid \texttt{print}\ V;\ S \mid \epsilon$$

```
sort V | symbol ⟦⟨ID⟩⟧ ;

sort P | ⟦⟨S⟩⟧ ;

sort S | ⟦ int ⟨V binds x⟩ = ⟨E⟩; ⟨S[x as V]⟩ ⟧
       | ⟦ print ⟨V⟩; ⟨S⟩ ⟧
       | ⟦ ⟧ ;
```

1. Introduction
2. Programming Language Basics
3. Symbol Tables = Environments
4. HACS
5. **Project Milestone 1**

Project Milestone 1 Issues

- Lexer issues: `.`, comments, character classes.
- Spaces in **sort** $\llbracket \_ \rrbracket$-declarations.
- Document, document, document...
  - Choices where specification is imprecise.
  - Who you discussed strategy with.
  - How to test your code and what is tested.

Questions?

evarose@cs.nyu.edu krisrose@cs.nyu.edu