CS 516: COMPILERS

Lecture 13

Topics

- First Class Functions: Lambda Calculus, Substitution, Evaluation
- Implementing an Interpreter

Materials

- lec14.zip

Announcements

- HW6: Frontend Compilation (OAT v. 1.0)
  – Parsing & basic code generation
  – **Due Thursday March 31 at 11:59**
  – **Oat v1 Definitions PDF**

- **Next Quiz topic**:
  – Parsing (In class, March 29th)
  – One-page, letter-sized, double-sided "cheat sheet" of notes permitted (must hand it in)

Wrapping up Parsing

- Lec12: Parsing (finding derivations in a grammar)
  - LR Grammars
  - Shift/Reduce parsing
  - LR(0) Grammars
  - Menhir

OAT and Homework 6

- Simple C-like Imperative Language
  - supports 64-bit integers, arrays, strings
  - top-level, mutually recursive procedures
  - scoped local, imperative variables
- See examples in hw4programs directory

- Homework 6 tasks:
  - Improve the parser to support all of hw5programs/*
  - Compile Oat AST to LLVM:

```
let rec cmp_exp (c:Ctxt.t) (exp:Ast.exp node)
  : Ll.ty * Ll.operand * stream =

let rec cmp_stmt (c:Ctxt.t) (rt:Ll.ty) (stmt:Ast.stmt node)
  : Ctxt.t * stream =


...
```

OAT and Homework 6

- To compile OAT variables, we maintain a mapping of source identifiers to the corresponding LLVMlite operands.

- Bindings are added for:
  - global OAT variables
  - local variables that are in scope.

```
module Ctxt = struct

  type t = (Ast.id * (Ll.ty * Ll.operand)) list

. . .
```

Compilation in a Nutshell

Source Code
(Character stream)

```
if (b == 0) { a = 1; }
```

Lexical Analysis

Token stream:

if	(	b	==	0	)	{	a	=	0	;	}

Compilation in a Nutshell

Source Code
(Character stream)

```
if (b == 0) { a = 1; }
```

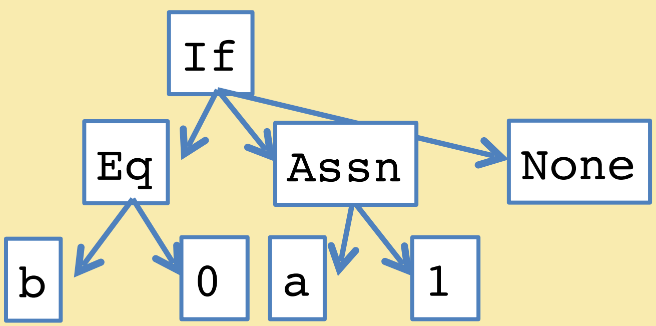
Lexical Analysis

Token stream:

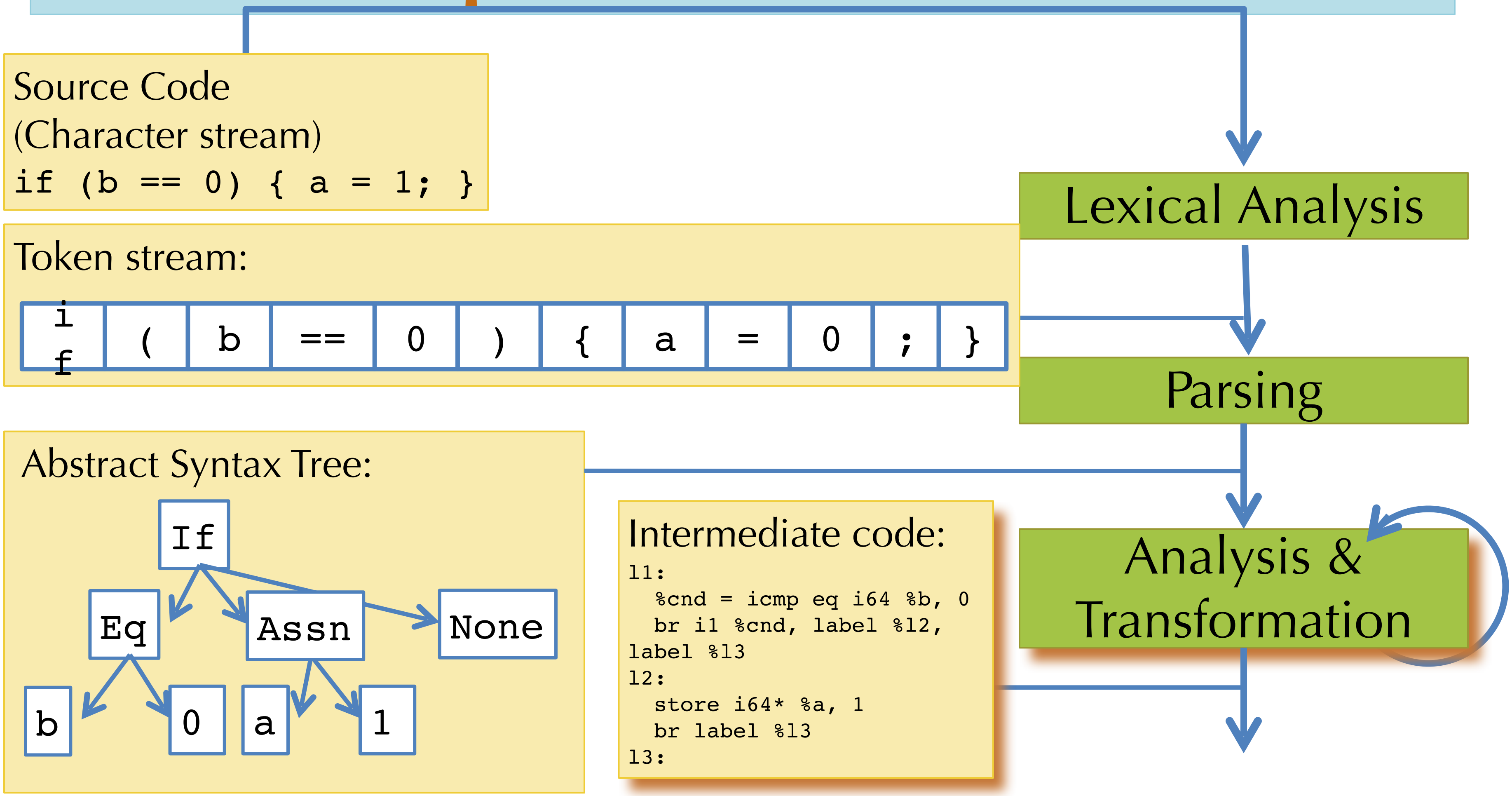
if	(	b	==	0	)	{	a	=	0	;	}

Parsing

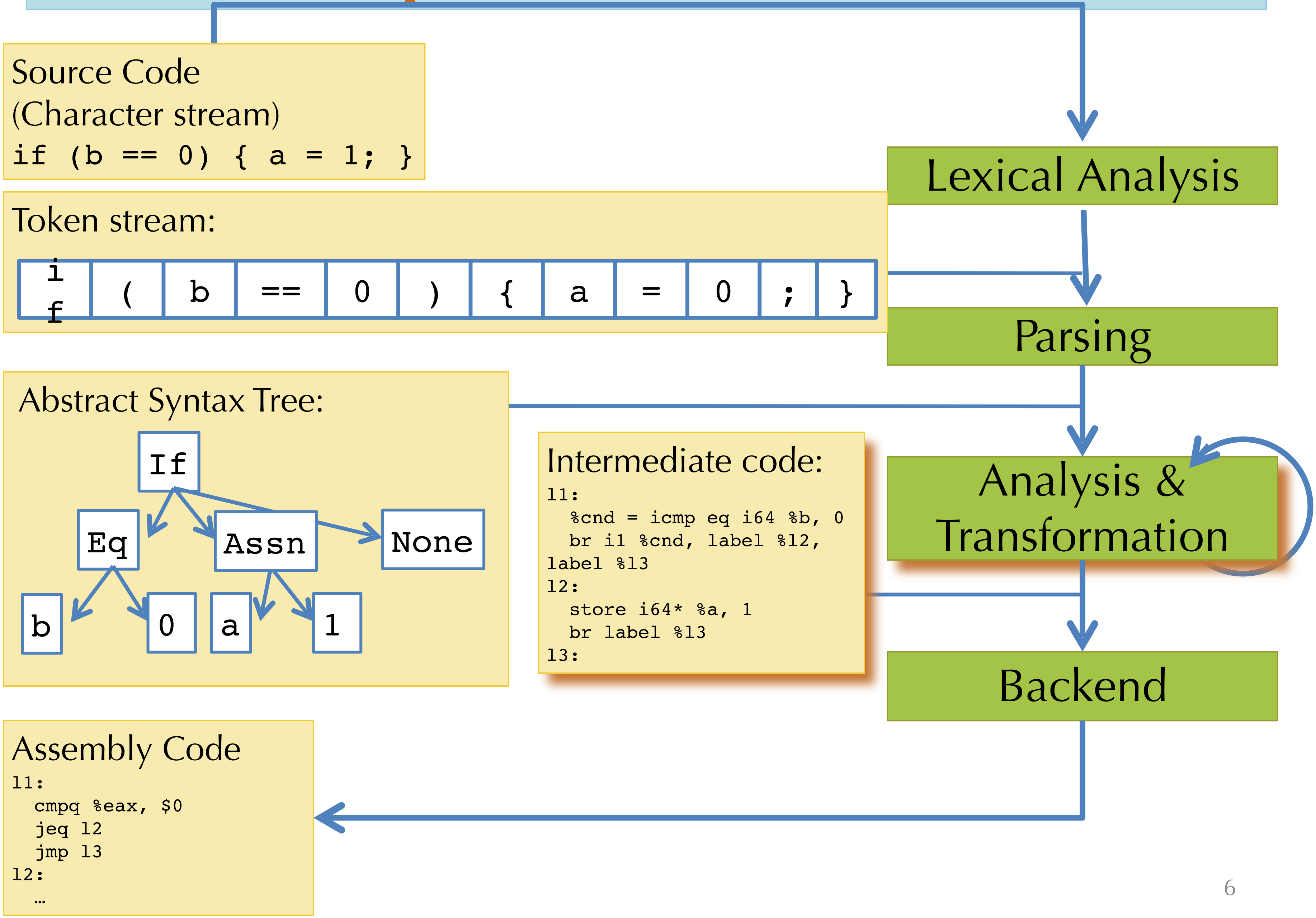
Abstract Syntax Tree:



Compilation in a Nutshell



Compilation in a Nutshell



Untyped lambda calculus
Substitution
Evaluation

FIRST-CLASS FUNCTIONS

“Functional” languages

- Languages like ML, Haskell, Scheme, Python, C#, Java 8, Swift
- Functions can be passed as arguments (e.g. map or fold)
- Functions can be returned as values (e.g. compose)
- Functions nest: inner function can refer to variables bound in the outer function

```
let add = fun x -> fun y -> x + y
let inc = add 1
let dec = add -1

let compose = fun f -> fun g -> fun x -> f (g x)
let id = compose inc dec
```

- How do we implement such functions?
  - in an interpreter? in a compiled language?

(Untyped) Lambda Calculus

- The lambda calculus is a minimal programming language.
  - Note: we're writing `(fun x -> e)` lambda-calculus notation: λ x. e
- It has variables, functions, and function application.
  - That's it!
  - It's Turing Complete.
  - It's the foundation for a *lot* of research in programming languages.
  - Basis for "functional" languages like Scheme, ML, Haskell, etc.

Abstract syntax in OCaml:

```
type exp =
  | Var of var            (* variables              *)
  | Fun of var * exp      (* functions: fun x -> e *)
  | App of exp * exp      (* function application  *)
```

Concrete syntax:

exp ::=	
| x	*variables*
| `fun` x -> exp	*functions*
| exp_1 exp_2	*function application*
| (exp)	*parentheses*

Values and Substitution

- The only values of the lambda calculus are (closed) functions:

```
val ::=
      | fun x -> exp      functions are values
```

- To *substitute* a (closed) value `v` for some variable `x` in an expression `e`
  - Replace all *free occurrences* of `x` in `e` by `v`.
  - In OCaml: written `subst v x e`
  - In Math: written e{v/x}

- Function application is interpreted by *substitution*:

```
  (fun x -> fun y -> x + y) 1
= subst 1 x (fun y -> x + y)
= (fun y -> 1 + y)
```

Lambda Calculus Operational Semantics

- Substitution function (in Math):

x{v/x}	= v	*(replace the free x by v)*
y{v/x}	= y	*(assuming $y \neq x$)*
(fun x -> exp){v/x}	= (fun x -> exp)	*(x is bound in exp)*
(fun y -> exp){v/x}	= (fun y -> exp{v/x})	*(assuming $y \neq x$)*
(e_1 e_2){v/x}	= (e_1{v/x} e_2{v/x})	*(substitute everywhere)*

- Examples:

x y {(fun z ->z)/y} ⇒ Answer...

(fun x -> x y){(fun z -> z) / y} ⇒ Answer...

(fun x -> x){(fun z -> z) / x} ⇒ Answer...

Lambda Calculus Operational Semantics

- Substitution function (in Math):

x{v/x}	= v	*(replace the free x by v)*
y{v/x}	= y	*(assuming $y \neq x$)*
(fun x -> exp){v/x}	= (fun x -> exp)	*(x is bound in exp)*
(fun y -> exp){v/x}	= (fun y -> exp{v/x})	*(assuming $y \neq x$)*
(e_1 e_2){v/x}	= (e_1{v/x} e_2{v/x})	*(substitute everywhere)*

- Examples:

  x y {(fun z ->z)/y} ⇒ x (fun z -> z)

  (fun x -> x y){(fun z -> z) / y} ⇒ Answer...

  (fun x -> x){(fun z -> z) / x} ⇒ Answer...

Lambda Calculus Operational Semantics

- Substitution function (in Math):

x{v/x}	= v	*(replace the free x by v)*
y{v/x}	= y	*(assuming $y \neq x$)*
(fun x -> exp){v/x}	= (fun x -> exp)	*(x is bound in exp)*
(fun y -> exp){v/x}	= (fun y -> exp{v/x})	*(assuming $y \neq x$)*
(e_1 e_2){v/x}	= (e_1{v/x} e_2{v/x})	*(substitute everywhere)*

- Examples:

x y {(fun z ->z)/y} ⇒ x (fun z -> z)

(fun x -> x y){(fun z -> z) / y} ⇒ (fun x -> x (fun z -> z))

(fun x -> x){(fun z -> z) / x} ⇒ Answer…

Lambda Calculus Operational Semantics

- Substitution function (in Math):

x{v/x}	= v	*(replace the free x by v)*
y{v/x}	= y	*(assuming $y \neq x$)*
(fun x -> exp){v/x}	= (fun x -> exp)	*(x is bound in exp)*
(fun y -> exp){v/x}	= (fun y -> exp{v/x})	*(assuming $y \neq x$)*
(e_1 e_2){v/x}	= (e_1{v/x} e_2{v/x})	*(substitute everywhere)*

- Examples:

  x y {(fun z ->z)/y} ⇒ x (fun z -> z)

  (fun x -> x y){(fun z -> z) / y} ⇒ (fun x -> x (fun z -> z))

  (fun x -> x){(fun z -> z) / x} ⇒ (fun x -> x) // x is not free!

Free Variables and Scoping

```
let add = fun x -> fun y -> x + y
let inc = add 1
```

- The result of `add 1` is a function
- After calling `add`, we can't throw away its argument (or its local variables) because those are needed in the function returned by add.
- We say that the variable `x` is *free* in `fun y -> x + y`
  - Free variables are defined in an outer scope
- We say that the variable `y` is *bound* by "`fun y`" and its scope is the body "`x + y`" in the expression `fun y -> x + y`

- A term with no free variables is called *closed*.
- A term with one or more free variables is called *open*.

Free Variable Calculation

- An OCaml function to calculate the set of free variables in a lambda expression:

```
let rec free_vars (e:exp) : VarSet.t =
  begin match e with
    | Var x          -> VarSet.singleton x
    | Fun(x, body) -> VarSet.remove x (free_vars body)
    | App(e1, e2)  -> VarSet.union (free_vars e1) (free_vars e2)
  end
```

- A lambda expression `e` is *closed* if `free_vars e` returns `VarSet.empty`
- In mathematical notation:

$$\begin{aligned} fv(x) &= \{x\} \\ fv(\text{fun } x \text{ -> } exp) &= fv(exp) \setminus \{x\} \quad \textit{('x' is a bound in exp)} \\ fv(exp_1\ exp_2) &= fv(exp_1) \cup fv(exp_2) \end{aligned}$$

Variable Capture

- Note that if we try to naively "substitute" an open term, a bound variable might capture the free variables:

```
     (fun x -> (x y))  {(fun z -> x) / y}        Note:  x is free in (fun x -> x)
=    fun x -> (x (fun z -> x))                   free x is captured!!
```

- Usually *not* the desired behavior
  - This property is sometimes called "dynamic scoping"
    The meaning of "x" is determined by where it is bound dynamically, not where it is bound statically.
  - Some languages (e.g. emacs lisp) are implemented with this as a "feature"
  - But, leads to hard to debug scoping issues

Alpha Equivalence

- Note that the names of bound variables don't matter.
  - i.e. it doesn't matter which variable names you use, as long as you use them consistently

    (fun x -> y x)   is the "same" as   (fun z -> y z)

    the choice of "x" or "z" is arbitrary, as long as we consistently rename them

    Two terms that differ only by consistent renaming of bound variables are called *alpha equivalent*

- The names of free variables do matter:

    (fun x -> y x)   is *not* the "same" as   (fun x -> z x)

  Intuitively: y an z can refer to different things from some outer scope

Fixing Substitution

- Consider the substitution operation:

$$\{e_2/x\}\ e_1$$

- To avoid capture, define substitution to pick an alpha equivalent version of e_1 such that the bound names of e_1 don't mention the free names of e_2.
  - Then do the "naïve" substitution.

For example: (fun x → (x y)) {(fun z → x) / y}

= (fun x′ → (x′ y)) {(fun z → x) / y} *rename* x to x'

= (fun x′ → (x′ (fun z → x)) *substitute*

Operational Semantics

- We write the "operational semantics" (i.e. the meaning of the language) in the following notation:

$$\mathbf{exp} \Downarrow \mathbf{v}$$

- Read this notation as "program **exp** evaluates to value **v**"

Operational Semantics

exp $\Downarrow$ v

- Specified using just two inference rules
- This is *call-by-value* semantics: function arguments are evaluated before substitution

Rule 1

$$\frac{}{v \Downarrow v}$$

"Values evaluate to themselves"

Rule 2

$$\frac{exp_1 \Downarrow (\text{fun } x \text{ -> } exp_3) \qquad exp_2 \Downarrow v \qquad exp_3\{v/x\} \Downarrow w}{exp_1\ exp_2 \Downarrow w}$$

"To evaluate function application: Evaluate the function to a value, evaluate the argument to a value, and then substitute the argument for the function. "

code demo

LAMBDA CALCULUS

1. **Implementing Lambda Calculus Interpreter**

```
open fun.ml
```

Adding Integers to Lambda Calculus

exp ::=
  | …
  | n — *constant integers*
  | $\text{exp}_1 + \text{exp}_2$ — *binary arithmetic operation*

val ::=
  | `fun` x `->` exp — *functions are values*
  | n — *integers are values*

$n\{v/x\} = n$		*constants have no free vars.*
$(e_1 + e_2)\{v/x\} = (e_1\{v/x\} + e_2\{v/x\})$		*substitute everywhere*

$$\frac{\text{exp}_1 \Downarrow n_1 \quad \text{exp2} \Downarrow n_2}{\text{exp}_1 + \text{exp}_2 \ \Downarrow (\text{n1} \ [\![+]\!] \ \text{n2})}$$

Object-level '+'   Meta-level '+'