Extensions to Simply-Typed Lambda Calculus

Basic Types

- Practical programming needs numerical and Boolean values and types. (Of course these can be encoded in lambda calculus.)

```
e ::=  …
     | v
     | e1 bop e2  (binary ops: +, -, *, /, <, >, =, <=, >=, and, or)
     | uop e      (unary ops: ~, not, pred, succ)

v ::= \x.e
     | …, -1, 0, 1, 2, … [all integers]
     | true | false

t ::= …
     | int
     | bool
```

- Semantics and typing rules for all the binary ops and unary ops are straight forward
- We dropped the type annotation from abstraction for brevity

Associativity and Precedence

- A grammar can be used to define associativity and precedence among the operators in an expression.
  - E.g., + and - are left-associative operators in mathematics;
  - * and / have higher precedence than + and - .
  - a + b + c = (a + b) + c;   a ** b ** c = a ** (b ** c)

- Consider the more interesting grammar G_1 for arithmetic:

```
Expr ::=        Expr + Term
              | Expr – Term
              | Term

Term ::=        Term * Factor
              | Term / Factor
              | Term % Factor
              | Factor

Factor ::=      Primary ** Factor
              | Primary

Primary ::= 0 | … | 9 | ( Expr )
```

Quiz: How would you change the definition of Expr if we want + and – to be “right associative?”

An Ambiguous Expression Grammar G_2

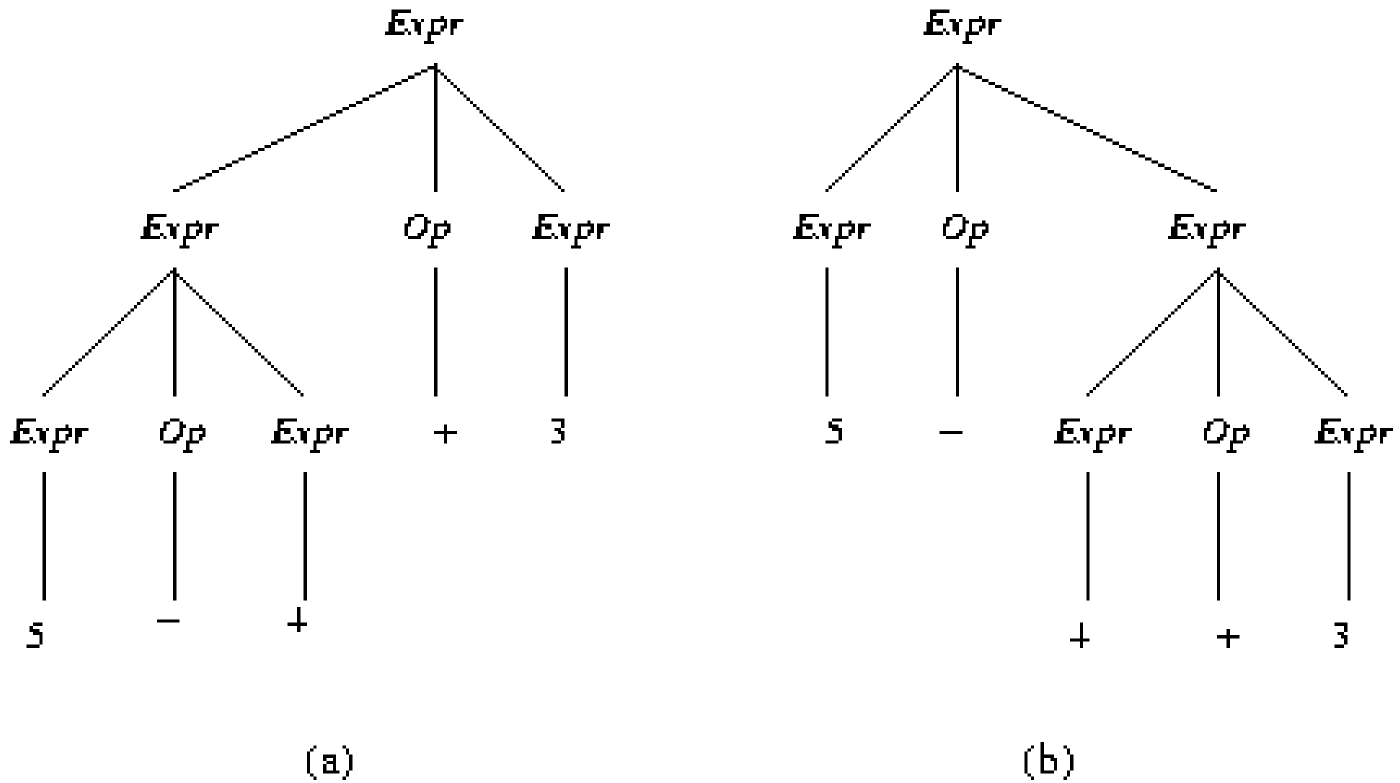
$Expr \rightarrow Expr \;\; Op \;\; Expr \mid (\, Expr \,) \mid \; Integer$

$Op \rightarrow + \mid - \mid * \mid / \mid \% \mid **$

Notes:

- G_2 is equivalent to G_1, *i.e.*, its language is the same.
- G_2 has fewer productions and non-terminals than G_1.
- However, G_2 is ambiguous.
- Ambiguity can be resolved using the associativity and precedence table

Ambiguous Parse of 5-4+3 Using Grammar G_2



Let Binding

- It is useful to bind intermediate results of computations to variables:

New syntax:

```
e ::= x                           (a variable)
    | true | false                (a boolean value)
    | if e1 then e2 else e3       (conditional)
    | \x.e                        (a nameless function)
    | e1 e2                       (function application)
    | let x = e1 in e2            (let expression)
```

x is bound in e2 (which is the scope of x)

Call-by-value Semantics and Typing

$$\frac{e1 \rightarrow e1'}{\text{let } x{=}e1 \text{ in } e2 \rightarrow \text{let } x{=}e1' \text{ in } e2}\ [\text{e-let}]$$

$$\frac{}{\text{let } x{=}v \text{ in } e2 \rightarrow e2\,[v/x]}\ [\text{e-letv}]$$

$$\frac{G \vdash e1 : t1 \qquad G, x{:}t1 \vdash e2 : t2}{G \vdash \text{let } x{=}e1 \text{ in } e2 : t2}\ [\text{t-let}]$$

Implementation of Let Expressions

- Question: can we implement this idea in pure lambda calculus?

source = lambda calculus + let

translate/compile

target = lambda calculus

Let Expressions

- Question: can we implement this idea in the lambda calculus?

  translate (let x = e1 in e2) =

  (\x.e2) e1

Let Expressions

- Question: can we implement this idea in the lambda calculus?

  translate (let x = e1 in e2) =

          (\x. translate e2) (translate e1)

Let Expressions

- Question: can we implement this idea in the lambda calculus?

```
translate (let x = e1 in e2) =
      (\x. translate e2) (translate e1)
translate (x) = x
translate (\x.e) = \x.translate e
translate (e1 e2) = (translate e1) (translate e2)
```

The Principle of "Bound Variable Names Don't Matter"

When you write

```
let x = \z.z z in
    let y = \w.w in (x y)
```

you assume you can change the declaration of y to a declaration of v (or another name) provided you systematically change the uses of y. E.g.:

```
let x = \z.z z in
    let v = \w.w in (x v)
```

provided that the name you pick doesn't conflict with the free variables of the expression. E.g.:

```
let x = \z.z z in
    let x = \w.w in (x x)
```

bad, original x captured

Static vs. Dynamic Scoping

- The ***scope*** of a name is the collection of expressions and/or statements which can access the name binding.
- In static scoping, a name is bound for a collection of statements according to its position in the source program → determined at compile time (static)
- In dynamic scoping, the valid association for a name X, at any point P of a program, is the most ***recent*** (in the temporal sense) association created for X which is still active when control flow arrives at P → determined at run time (dynamic)
- Most modern languages use static (or *lexical*) scoping.

Static vs. Dynamic Scoping (II)

```
let x = v1 in
let y = (let x = v2 in x)
in x
```

- This expression evaluates to
  - v1 (static scoping)
  - v2 (dynamic scoping)

PAIRS

- Programming languages offer compound types.
- Simplest is *pairs*, or 2-*tuples*.
- We introduce one new value {v1, v2}
- One new product type: t1 * t2.

Pairs (syntax)

e ::= …	expressions:
	{e1, e2}	pair
	e.1	first projection
	e.2	second projection
v ::= …	values:
	{v1, v2}	pair value
t ::= …	types:
	t1 * t2	product type (or $t_1 \times t_2$)

PAIRS (EVALUATION)

Quiz: Change (E-Pair1) and (E-Pair2) so that the evaluation follows a right-to-left call by value operational semantics?

[e → e']

$$\frac{}{\{v_1, v_2\}.1 \rightarrow v_1} \text{ (E-PairBeta1)} \qquad \frac{}{\{v_1, v_2\}.2 \rightarrow v_2} \text{ (E-PairBeta2)}$$

$$\frac{\mathrm{e} \rightarrow \mathrm{e}'}{e.1 \rightarrow e'.1} \text{ (E-Proj1)} \qquad \frac{\mathrm{e} \rightarrow \mathrm{e}'}{e.2 \rightarrow e'.2} \text{ (E-Proj2)}$$

$$\frac{e_1 \rightarrow e_1'}{\{e_1, e_2\} \rightarrow \{e_1', e_2\}} \text{ (E-Pair1)} \qquad \frac{e_2 \rightarrow e_2'}{\{v_1, e_2\} \rightarrow \{v_1, e_2'\}} \text{ (E-Pair2)}$$

Example Evaluations

Left to right evaluation:

{if 3+2 > 0 then true else false, succ 0}.1

→ {if 5 > 0 then true else false, succ 0}.1

→ {if true then true else false, succ 0}.1

→ {true, succ 0}.1

→ {true, 1}.1

→ true

Pairs must be evaluated to values before passing to functions:

(\x:int*int. x.2) {pred 1, 6/2}

→ (\x:int*int. x.2) {0, 6/2}

→ (\x:int*int. x.2) {0, 3}

→ {0, 3}.2

→ 3

PAIRS (TYPING)

$[\Gamma \vdash e : t]$

$$\frac{\Gamma \mid\!- e_1 : t_1 \quad \Gamma \mid\!- e_2 : t_2}{\Gamma \mid\!- \{e_1, e_2\} : t_1 \times t_2} \quad \text{(T - Pair)}$$

$$\frac{\Gamma \mid\!- e : t_1 \times t_2}{\Gamma \mid\!- e.1 : t_1} \quad \text{(T - Proj1)}$$

$$\frac{\Gamma \mid\!- e : t_1 \times t_2}{\Gamma \mid\!- e.2 : t_2} \quad \text{(T - Proj2)}$$

Tuples

- Tuples generalize from pairs: binary product → n-ary product

e ::= …	expressions:
| {e1, …, en} (or $\{e_i^{i \in 1..n}\}$)	tuple
| e.i	i^{th} projection
v ::= …	values:
| {v1, …, vn}	tuple value
t ::= …	types:
| t1 * …* tn (or $\{t_i^{i \in 1..n}\}$)	tuple type

Tuple Evaluation and Typing

$$\frac{}{\{v_i^{\,i\in 1..n}\}.j \rightarrow v_j} \quad \text{(E - ProjTuple)}$$

$$\frac{\mathrm{e} \rightarrow \mathrm{e}'}{e.i \rightarrow e'.i} \quad \text{(E - ProjTuple1)}$$

$$\frac{\mathrm{e_j} \rightarrow \mathrm{e_j}'}{\{v_1,\ldots,v_{j-1},e_j,\ldots,e_n\} \rightarrow \{v_1,\ldots,v_{j-1},e_j',\ldots,e_n\}} \quad \text{(E - Tuple)}$$

$$\frac{\text{for each } i : \Gamma \mid\!- e_i : t_i}{\Gamma \mid\!- \{e_i^{\,\mathrm{i}\in 1..\mathrm{n}}\} : \{t_i^{\,i\in 1..n}\}} \quad \text{(T - Tuple)}$$

$$\frac{\Gamma \mid\!- e : \{t_i^{\,i\in 1..n}\}}{\Gamma \mid\!- e.j : t_j} \quad \text{(T - Proj)}$$

- Note that order of elements in tuple is significant.
- Evaluation is from left to right.
- Projection is done after tuple becomes value.

Records

- Straightforward to extend tuples into records
- Elements are indexed by labels:
  - {y=10}
  - {id=1, salary=50000, active=true}
- The order of the record fields is often insignificant in most PL
  - {y=10, x= 5} is the same as {x=5, y=10}
- To access fields of a record:
  - a.id
  - b.salary
- Syntax and semantic rules left as an exercise.

SUMS

- Program needs to deal with heterogeneous collection of values – values that can take different shapes:
  - A binary tree node can be:
    - A leaf node, or
    - An interior node
  - An abstract syntax tree node of λ-calculus can be:
    - A variable
    - A function abstraction, or
    - An application, etc.
- *Sum* type: union of two types
- More generally, *variant* type: union of n types.

Sum (Syntax)

e ::= …	expressions:
	inl e	injection (left)
	inr e	injection (right)
	case e of inl x =>e1	inr x => e2	case
v ::= …	values:
	inl v	injection value (left)
	inr v	injection value (right)
t ::= …	types:
	t1 + t2	sum type

Sums (Example)

- There are two types:
  - faculty = {empid: int, position: string}
  - student = {stuid: int, level: int}
- Define a sum type:
  - personnel = faculty + student
- We can "inject" element of *faculty* or *student* type into *personnel* type. Think of inl and inr as functions:
  - inl : faculty → personnel
  - inr: student → personnel
- To use a elements of sum type, we use the case expression:

```
getid = \p : personnel .
  case p of
      inl x => x.empid
    | inr x => x.stuid
```

Sums (semantics)

$$\frac{}{\text{case (inl v) of inl } \mathrm{x}_1 \Rightarrow \mathrm{e}_1 \mid \text{inr } \mathrm{x}_2 \Rightarrow \mathrm{e}_2 \rightarrow e_1[v/x_1]} \text{ (E-CaseInl)}$$

$$\frac{}{\text{case (inr v) of inl } \mathrm{x}_1 \Rightarrow \mathrm{e}_1 \mid \text{inr } \mathrm{x}_2 \Rightarrow \mathrm{e}_2 \rightarrow e_2[v/x_2]} \text{ (E-CaseInr)}$$

$$\frac{e \rightarrow e'}{\begin{array}{l}\text{case e of inl } \mathrm{x}_1 \Rightarrow \mathrm{e}_1 \mid \text{inr } \mathrm{x}_2 \Rightarrow \mathrm{e}_2 \\ \rightarrow \text{case e' of inl } \mathrm{x}_1 \Rightarrow \mathrm{e}_1 \mid \text{inr } \mathrm{x}_2 \Rightarrow \mathrm{e}_2\end{array}} \text{ (E-Case)}$$

$$\frac{\mathrm{e} \rightarrow \mathrm{e'}}{\text{inl e} \rightarrow \text{inl e'}} \text{ (E-Inl)} \qquad \frac{\mathrm{e} \rightarrow \mathrm{e'}}{\text{inr e} \rightarrow \text{inr e'}} \text{ (E-Inr)}$$