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.)

\[
e ::= \ldots
\]

- \(v\) ::= \(\\langle x.e\rangle\) \hspace{2cm} \(t ::= \ldots\)
  - \(\ldots, -1, 0, 1, 2, \ldots \) [all integers]
  - \(\text{true} | \text{false}\)

- Semantics and typing rules for all the binary ops and unary ops are straightforward.
- 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 - .
  - \( a + b + c = (a + b) + c; \quad a ** b ** c = a ** ( b ** c ) \)

- Consider the more interesting grammar \( G_1 \) for arithmetic:

\[
\begin{align*}
\text{Expr} & ::= \quad \text{Expr} + \text{Term} \\
& \quad | \quad \text{Expr} - \text{Term} \\
& \quad | \quad \text{Term} \\
\text{Term} & ::= \quad \text{Term} * \text{Factor} \\
& \quad | \quad \text{Term} / \text{Factor} \\
& \quad | \quad \text{Term} \% \text{Factor} \\
& \quad | \quad \text{Factor} \\
\text{Factor} & ::= \quad \text{Primary} ** \text{Factor} \\
& \quad | \quad \text{Primary} \\
\text{Primary} & ::= \quad 0 \mid \ldots \mid 9 \mid ( \text{Expr} )
\end{align*}
\]
AN AMBIGUOUS EXPRESSION GRAMMAR $G_2$

$Expr \rightarrow Expr \ Op \ Expr \ | \ (\ Expr) \ | \ Integer$

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

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 $G_2$

(a)  
```
Expr  |  Op  |  Expr  
Expr  |  Op  |  Expr  +  3
Expr  |  -   |  4    
S     |      |      
```

(b)  
```
Expr  |  Op  |  Expr  
S     |  -   | Expr  
Expr  |  Op  |  Expr  +  3
Expr  |      |      
```

5
**Let Binding**

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

New syntax:

\[ e ::= x \]

(a variable)

\[ \text{true} | \text{false} \]

(a boolean value)

\[ \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \]

(conditional)

\[ \text{x.e} \]

(a nameless function)

\[ e_1 \; e_2 \]

(function application)

\[ \text{let } x = e_1 \text{ in } e_2 \]

(let expression)

\[ x \text{ is bound to } e_2 \text{ (which is the scope of } x) \]
CALL-BY-VALUE SEMANTICS AND TYPING

\[
e_1 \rightarrow e_1'
\]

\[\text{[e-let]}\]

\[
\text{let } x=e_1 \text{ in } e_2 \rightarrow \text{let } x=e_1' \text{ in } e_2
\]

\[\text{[e-letv]}\]

\[
\text{let } x=v \text{ in } e_2 \rightarrow e_2 \ [v/x]
\]

\[\text{[t-let]}\]

\[
G \ |- \ e_1 : t_1 \quad G, x:t_1 \ |- \ e_2 : t_2
\]

\[
G \ |- \ \text{let } x=e_1 \text{ in } e_2 : t_2
\]
**IMPLEMENTATION OF LET EXPRESSIONS**

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

```
source = lambda calculus + let
```

```
target = lambda calculus
```

translate/compile
Let Expressions

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

\[ \text{translate } (\text{let } x = e_1 \text{ in } e_2) = (\\lambda x. e_2) \ e_1 \]
Let Expressions

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

\[ \text{translate (let } x = e1 \text{ in } e2) = \]
\[ (\lambda x. \text{translate } e2) \text{ (translate } e1) \]
**LET EXPRESSIONS**

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

  translate (let x = e1 in e2) =
  \( (\lambda x. \text{translate } e2) \text{ (translate } e1) \)
  translate (x) = x
  translate (\x.e) = \x.\text{translate } e
  translate (e1 e2) = (\text{translate } e1) \text{ (translate } e2)
THE PRINCIPLE OF “BOUND VARIABLE NAMES DON’T MATTER”

When you write

```ml
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 other name) provided you systematically change the uses of `y`. eg:

```ml
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. eg:

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

bad, capturing
**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: \( t1 \times t2 \).
PAIRS (SYNTAX)

\[
e ::= \ldots
\ |
\{e_1, e_2\}
\ |
e.1
\ |
e.2
\]

\[
v ::= \ldots
\ |
\{v_1, v_2\}
\]

\[
t ::= \ldots
\ |
t_1 \ast t_2
\]

expressions:
pair
first projection
second projection
values:
pair value
types:
product type
PAIRS (evaluation)

\[ e \rightarrow e' \]

\[
\begin{align*}
\{v_1, v_2\} \cdot 1 & \rightarrow v_1 & (E - \text{PairBeta1}) \\
\{v_1, v_2\} \cdot 2 & \rightarrow v_2 & (E - \text{PairBeta2}) \\
\end{align*}
\]

\[
\begin{align*}
e & \rightarrow e' & (E - \text{Proj1}) \\
e.1 & \rightarrow e'.1 \\
\end{align*}
\]

\[
\begin{align*}
e & \rightarrow e' & (E - \text{Proj2}) \\
e.2 & \rightarrow e'.2 \\
\end{align*}
\]

\[
\begin{align*}
e_1 & \rightarrow e_1' & (E - \text{Pair1}) \\
\{e_1, e_2\} & \rightarrow \{e_1', e_2\} \\
\end{align*}
\]

\[
\begin{align*}
e_2 & \rightarrow e_2' & (E - \text{Pair2}) \\
\{v_1, e_2\} & \rightarrow \{v_1, e_2'\} \\
\end{align*}
\]
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:
- (\(\lambda x: \text{int}*\text{int}. x.2\) \{pred 1, 6/2\})
- (\(\lambda x: \text{int}*\text{int}. x.2\) \{0, 6/2\})
- (\(\lambda x: \text{int}*\text{int}. x.2\) \{0, 3\})
- \{0, 3\}.2
- 3
PAIRS (TYPING)

\[
\begin{array}{c}
\Gamma \vdash e_1 : t_1 \\
\Gamma \vdash e_2 : t_2 \\
\hline
\Gamma \vdash \{e_1, e_2\} : t_1 \times t_2
\end{array}
\]  
(T - Pair)

\[
\begin{array}{c}
\Gamma \vdash e : t_1 \times t_2 \\
\hline
\Gamma \vdash e.1 : t_1
\end{array}
\]  
(T - Proj1)

\[
\begin{array}{c}
\Gamma \vdash e : t_1 \times t_2 \\
\hline
\Gamma \vdash e.2 : t_2
\end{array}
\]  
(T - Proj2)
**Tuples**

- Tuples generalize from pairs: binary product $\rightarrow$ n-ary product

  - $e ::= \ldots$
    - $\{e_1, \ldots, e_n\}$ (or $\{e_i \mid i \in 1..n\}$)
    - $e.i$
    - expressions:
    - tuple
    - projection

  - $v ::= \ldots$
    - $\{v_1, \ldots, v_n\}$
    - values:
    - tuple value

  - $t ::= \ldots$
    - $t_1 \ast \ldots \ast t_n$ (or $\{t_i \mid i \in 1..n\}$)
    - types:
    - tuple type
**Tuple Evaluation and Typing**

\[ \{v_i^{i \in 1..n}\}.j \rightarrow v_j \]  
\[ (E - \text{ProjTuple}) \]

\[ e \rightarrow e' \]
\[ e.i \rightarrow e'.i \]  
\[ (E - \text{ProjTuple1}) \]

\[ e_j \rightarrow e_j' \]
\[ \{v_1,\ldots,v_{j-1},e_j,\ldots,e_n\} \rightarrow \{v_1,\ldots,v_{j-1},e_j',\ldots,e_n\} \]  
\[ (E - \text{Tuple}) \]

\[ \frac{e_i}{\Gamma | - e : t_i} \]  
\[ \frac{\Gamma | - \{e_i^{i \in 1..n}\} : \{t_i^{i \in 1..n}\}}{(T - \text{Tuple})} \]
\[ \frac{\Gamma | - e : \{t_i^{i \in 1..n}\}}{(T - \text{Proj})} \]

- 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.
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 $\lambda$-calculus can be:
  - A variable
  - A function abstraction
  - An application, etc.

- **Sum** type: union of two types

- More generally, **variant** type: union of $n$ types.
**Sum (Syntax)**

\[ e ::= \ldots \]
\[ | \ \text{inl } e \]
\[ | \ \text{inr } e \]
\[ | \ \text{case } e \ \text{of } \ \text{inl } x \Rightarrow e_1 \ | \ \text{inr } x \Rightarrow e_2 \]

\[ v ::= \ldots \]
\[ | \ \text{inl } v \]
\[ | \ \text{inr } v \]

\[ t ::= \ldots \]
\[ | \ t_1 + t_2 \]

expressions:
- injection (left)
- injection (right)
- case

values:
- injection value (left)
- injection value (right)

types:
- 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 \rightarrow personnel`
  - `inr: student \rightarrow 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)

\[
\begin{align*}
&\text{(E - CaseInl)} \\
\text{case (inl } v\text{) of inl } x_1 \Rightarrow e_1 | \text{inr } x_2 \Rightarrow e_2 \Rightarrow e_1[v/x_1]
\end{align*}
\]

\[
\begin{align*}
&\text{(E - CaseInr)} \\
\text{case (inr } v\text{) of inl } x_1 \Rightarrow e_1 | \text{inr } x_2 \Rightarrow e_2 \Rightarrow e_2[v/x_2]
\end{align*}
\]

\[
\begin{align*}
&\text{(E - Case)} \\
e \Rightarrow e' \\
\text{case } e\text{ of inl } x_1 \Rightarrow e_1 | \text{inr } x_2 \Rightarrow e_2 \\
\Rightarrow \text{case } e'\text{ of inl } x_1 \Rightarrow e_1 | \text{inr } x_2 \Rightarrow e_2
\end{align*}
\]

\[
\begin{align*}
&\text{(E - Inl)} \\
e \Rightarrow e' \\
\text{inl } e \Rightarrow \text{inl } e'
\end{align*}
\]

\[
\begin{align*}
&\text{(E - Inr)} \\
e \Rightarrow e' \\
\text{inr } e \Rightarrow \text{inr } e'
\end{align*}
\]