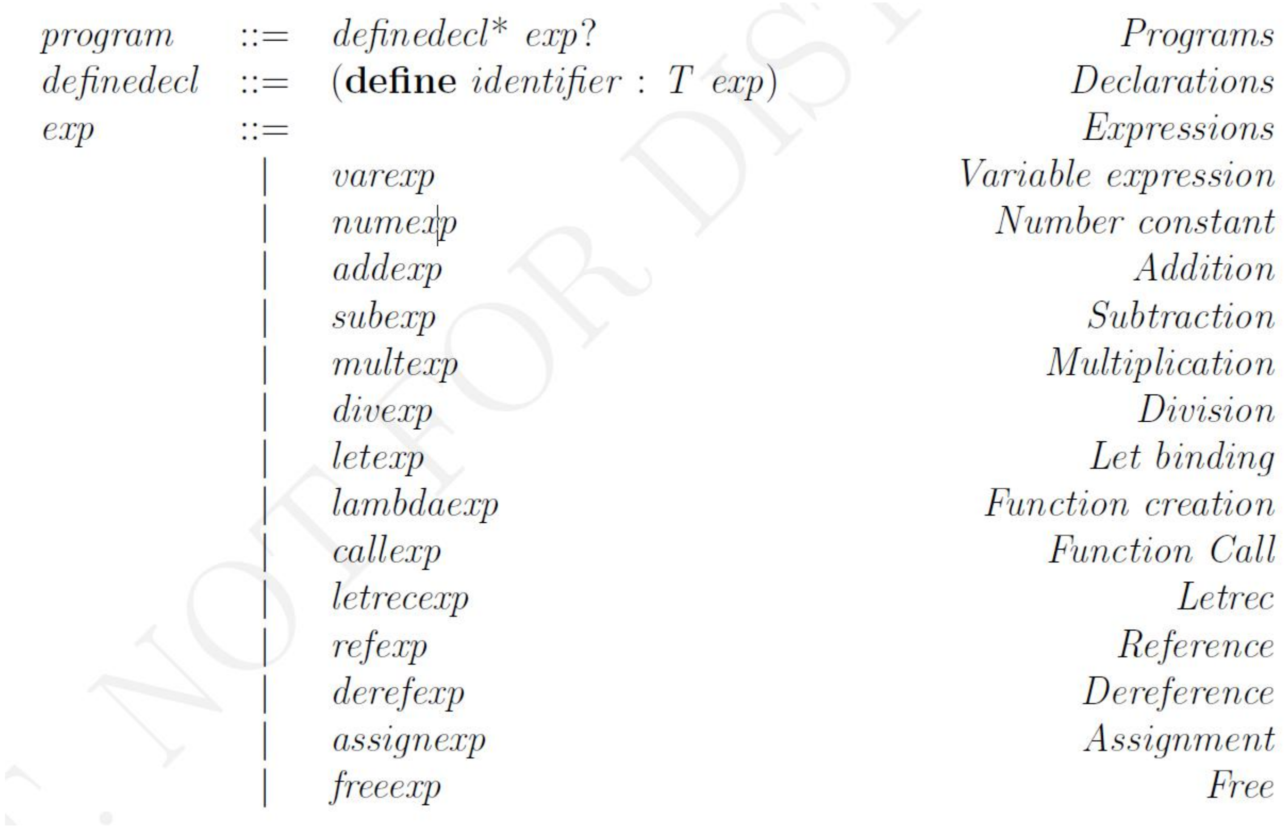
COti S 342

Recitation 11/18/2019 – 11/20/2019

Topic

- Typelang
- Q&A

Typelang



program	::=	*definedecl* exp?*	*Programs*
definedecl	::=	(**define** *identifier* : *T* *exp*)	*Declarations*
exp	::=		*Expressions*
	|	*varexp*	*Variable expression*
	|	*numexp*	*Number constant*
	|	*addexp*	*Addition*
	|	*subexp*	*Subtraction*
	|	*multexp*	*Multiplication*
	|	*divexp*	*Division*
	|	*letexp*	*Let binding*
	|	*lambdaexp*	*Function creation*
	|	*callexp*	*Function Call*
	|	*letrecexp*	*Letrec*
	|	*refexp*	*Reference*
	|	*derefexp*	*Dereference*
	|	*assignexp*	*Assignment*
	|	*freeexp*	*Free*

Typelang

T	::=		*Types*
	|	`unit`	*Unit Type*
	|	`num`	*Number Type*
	|	`bool`	*Boolean Type*
	|	`(` T^* `->` T `)`	*Function Type*
	|	`Ref` T	*Reference Type*

- Base Type
- Recursively-defined types, i.e. their definition makes use of other types: reference, function types

Typelang

- TypeLang assert that all numeric values (constants) have type **num**

(Num)

n : **num**

- TypeLang assert that all Boolean values (constants) have type **bool**

(Num)

n : **bool**

Typelang

- Atomic: no subexpressions

$$\frac{}{\text{(NumExp n) : } \textbf{num}}\ \text{(NumExp)}$$

Typelang

- Conditional assertion: if subexpressions of the addition expression always produce values of type **num**, then the addition expression will produce a value of type **num**.

$$\text{(ADDEXP)}\quad \frac{tenv \vdash e_i : \mathbf{num}, \forall i \in 1..n}{tenv \vdash (AddExp\ \ e_0\ \ e_1\ \ \ldots\ \ e_n) : \mathbf{num}}$$

- if subexpressions e_0 to e_n have type **num**, then the expression (AddExp e_0 , e_1 , . . . , e_n) will have type **num** also

Typelang

(MultExp)

$$\frac{tenv \vdash e_i : \mathtt{num}, \forall i \in 1..n}{tenv \vdash (\mathit{MultExp}\ e_0\ e_1\ \ldots\ e_n) : \mathtt{num}}$$

(SubExp)

$$\frac{tenv \vdash e_i : \mathtt{num}, \forall i \in 1..n}{tenv \vdash (\mathit{SubExp}\ e_0\ e_1\ \ldots\ e_n) : \mathtt{num}}$$

(DivExp)

$$\frac{tenv \vdash e_i : \mathtt{num}, \forall i \in 1..n}{tenv \vdash (\mathit{DivExp}\ e_0\ e_1\ \ldots\ e_n) : \mathtt{num}}$$

Typelang

$$\texttt{get(tenv, v')} = \begin{cases} \mathit{Error} & \mathit{tenv} = \texttt{(EmptyEnv)} \\ t & \mathit{tenv} = \texttt{(ExtendEnv v t tenv')} \\ & \texttt{and } v = v' \\ \texttt{get(tenv', v')} & \text{Otherwise.} \end{cases}$$

$$(\textsc{VarExp}) \quad \frac{get(tenv, var) = t}{tenv \vdash (VarExp \;\; var) : t}$$

Typelang

$letexp \quad ::= \quad (\textbf{let}\ ((identifier : T\ \mathrm{exp})^{+})\ exp)$ *Let expression*

(LetExp)

$$\frac{\begin{array}{c} tenv \vdash e_i : t_i, \forall i \in 0..n \\ tenv_n = (ExtendEnv\ var_n\ t_n\ tenv_{n-1}) \quad \ldots \\ tenv_0 = (ExtendEnv\ var_0\ t_0\ tenv) \\ tenv_n \vdash e_{body} : t \end{array}}{tenv \vdash (LetExp\ var_0, \ldots, var_n, t_0, \ldots, t_n, e_0, \ldots, e_n, e_{body}) : t}$$

Typelang

$$\mathit{lambdaexp} \quad ::= \quad (\textbf{lambda}\ (\{\mathit{identifier} : T\}^{*})\ \mathit{exp}) \qquad \mathit{Lambda}$$

```
(lambda
  (
    x : num      //Argument 1
    y : num      //Argument 2
    z : num      //Argument 3
  )
  (+ x (+ y z))
)
```

- Declares a function with three arguments x, y and z
- Type for this function is,
- num num num -> num
- Return type is num as well
- Type checks!

```
(
  (lambda
    (
      x : num      //Argument 1
      y : num      //Argument 2
      z : num      //Argument 3
    )
    (+ x (+ y z))
  )
  1 2 3
)
```

- Declares the same function and also calls it by passing integer parameters 1, 2 and 3 for arguments x, y and z
- Type checks!

Typelang

(LambdaExp)

$$\frac{\begin{array}{c} tenv_n = (ExtendEnv\ var_n\ t_n\ tenv_{n-1}) \ldots \\ tenv_0 = (ExtendEnv\ var_0\ t_0\ tenv) \qquad tenv_n \vdash e_{body} : t \end{array}}{tenv \vdash (LambdaExp\ var_0\ \ldots\ var_n, t_0\ \ldots\ t_n, e_{body}) : (t_0\ \ldots\ t_n -> t)}$$

(CallExp)

$$\frac{tenv \vdash e_{op} : (t_0\ \ldots\ t_n -> t) \qquad tenv \vdash e_i : t_i, \forall i \in 0..n}{tenv \vdash (CallExp\ e_{op}\ e_0\ \ldots\ e_n) : t}$$

Typelang

T	::=		*Types*
	|	`unit`	*Unit Type*
	|	`num`	*Number Type*
	|	`bool`	*Boolean Type*
	|	`(` T^* `->` T `)`	*Function Type*
	|	`Ref` T	*Reference Type*

type	::=	...	*Types*
	|	`String`	*String Type*
	|	`( T , T )`	*Pair Type*
	|	`List < T >`	*List Type*

Q&A