Programming Languages

Lec 3: Design of PLs - Expressions, Procedures

김재호

Designing a Programming Language

- 프로그래밍 언어를 설계하고, 만든다는 것은 굉장히 엄밀하면서도 아름다운 일
  - 프로그램을 기술할 때, 어떤 직관과 철학을 담은 언어를 사용하는지에 따라 기술 방식은 천차만별 (e.g., Imperative vs Declarative, Procedural vs OOP vs Functional)
  - 또 얼마나 달콤한 문법적 설탕(syntactic sugar)을 첨가하는지에 따라 같은 의미의 코드가 훨씬 간결하게 표현되기도, 혹은 너무 달콤해져서 처음 보는 사람이 납득하기 어려워질지도 모른다

Designing a Programming Language

- 잘 짜인 프로그래밍 언어는, 마치 실제 언어처럼 두 가지 요소를 명확히 정의해야만 한다:
  - **Syntax(문법):** 어떻게 프로그램을 작성할 것인가
  - **Semantics(의미):** 그 프로그램이 무엇을 뜻하는가
- 그리고 우리는 이 두 요소를 inductive definitions를 통해 formal하게 정의할 수 있다

Let: Our First Language

Let Language

Syntax of Let

$$
\begin{array}{lcl}
P & \to & E \\
E & \to & n \\
 & | & x \\
 & | & E + E \\
 & | & E - E \\
 & | & \texttt{iszero } E \\
 & | & \texttt{if } E \texttt{ then } E \texttt{ else } E \\
 & | & \texttt{let } x = E \texttt{ in } E \\
 & | & \texttt{read}
\end{array}
$$

Let Language

Semantics of Let

- 언어의 semantics를 정의하기 위해서는, values(값)와 environments(환경) 정의 필요
- **Values:** 말 그대로 우리가 프로그램에서 다룰 수 있는 모든 “값”을 의미
  - e.g., 1, 2, 3, true, false, …
- **Environments:** Variable-Value Mapping
  - 어떤 코드를 실행하는 과정에서, 특정 상황에 존재하는 variables(변수)와 그것이 가지고 있는 values를 함수의 형태로 정의

Let Language

Semantics of Let

- Let Lang에서 values의 집합은 다음과 같이 표현:

$$v \in \boldsymbol{Val} = \mathbb{Z} + \boldsymbol{Bool}$$

- Environment는 variables 집합에서 values 집합으로의 function으로 표현:

$$\rho \in \boldsymbol{Env} = \boldsymbol{Var} \to \boldsymbol{Val}$$

- 기억해야 할 점: Semantics를 정의하기 위해서는 여러 “집합”이 사용된다!! 왜?

Let Language

Notation of Env

- []
- $[x \to v]\rho$ (or $\rho[x \to v]$)

Let Language

Evaluation of Expressions

- **Expressions:** 프로그래밍 언어에서, 연산을 거쳐 어떤 value를 가지게 되는 문법 요소
- **Evaluation:** Env를 통해 어떤 expression의 값을 알아내는 과정; 평가
  - e.g., 1 is evaluates to 1, if Env is $[x \rightarrow 1]$, then x + 1 is evaluates to 2
- Env ρ가 주어졌을 때, 어떤 exp e가 value v로 eval된다면, 다음과 같이 표현

$$\rho \vdash e \Rightarrow v$$

- Env에 따라 eval이 안 되는 exp도 있다!

Let Language

Evaluation of Expressions

- 이를 활용해 semantics 집합 $S = Env \times Exp \times Val$ 정의 가능
- 즉, 주어진 ρ에 대해 e가 “의미”를 가짐 $\Leftrightarrow (\rho, e, v) \in S$인 v가 존재

- 우리는 어떤 구조가 집합에 존재하는지 증명하는 도구로써, rule of inference를 배움!

Let Language

Evaluation Rules

$$\frac{}{\rho \vdash n \Rightarrow n} \qquad \frac{}{\rho \vdash x \Rightarrow \rho(x)}$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 + E_2 \Rightarrow n_1 + n_2} \qquad \frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 - E_2 \Rightarrow n_1 - n_2}$$

$$\frac{}{\rho \vdash \texttt{read} \Rightarrow n} \qquad \frac{\rho \vdash E \Rightarrow 0}{\rho \vdash \texttt{iszero}\ E \Rightarrow \mathit{true}} \qquad \frac{\rho \vdash E \Rightarrow n}{\rho \vdash \texttt{iszero}\ E \Rightarrow \mathit{false}}\ n \neq 0$$

$$\frac{\rho \vdash E_1 \Rightarrow \mathit{true} \qquad \rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 \Rightarrow v} \qquad \frac{\rho \vdash E_1 \Rightarrow \mathit{false} \qquad \rho \vdash E_3 \Rightarrow v}{\rho \vdash \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 \Rightarrow v}$$

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \qquad [x \mapsto v_1]\rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{let}\ x = E_1\ \texttt{in}\ E_2 \Rightarrow v}$$

Let Language

Evaluation Rules - Examples

$$[] \vdash \texttt{let x = 1 in x + 2} \Rightarrow$$

Let Language

Evaluation Rules - Examples

$$[] \vdash \texttt{let x = 1 in let y = 2 in x + y} \Rightarrow$$

Let Language

Evaluation Rules - Examples

$$[] \vdash \texttt{let x = (let y = 2 in y + 1) in x + 3} \Rightarrow$$

Let Language

Evaluation Rules - Examples

$$[] \vdash \texttt{let x = 1 in let y = 2 in let x = 3 in x + y} \Rightarrow$$

Let Language

Evaluation Rules - Examples

$$[] \vdash \texttt{let x = 1 in let y = (let x = 2 in x + x) in x + y} \Rightarrow$$

Let Language

Evaluation Rules - Examples

When $\boldsymbol{\rho}$ is $[\mathtt{x} \mapsto \mathbf{1}, \mathtt{y} \mapsto \mathbf{2}]$:

$$\boldsymbol{\rho} \vdash \texttt{if iszero (x - 1) then y - 1 else y + 1} \Rightarrow$$

Let Language

- Let 언어는 굉장히 작고 간단하지만, 프로그래밍 언어로써 충분히 기능함
- 그러나, 프로그래밍 언어에서 핵심이 되는 “함수”가 아직 없음
- 함수의 핵심 키워드는 macro와 parameter
  - 마치 macro처럼 코드 묶음을 단위화 해서 간편하게 사용할 수 있음
  - 또한, parameter가 존재하여 어떤 값을 넣느냐에 따라 다른 결과를 만들 수 있음
- 이제 우리의 언어에 함수를 추가해 보자!

Proc: Let + Procedures

Proc Language

Syntax of Proc

$$
\begin{array}{lcl}
P & \to & E \\
E & \to & n \\
 & | & x \\
 & | & E + E \\
 & | & E - E \\
 & | & \texttt{iszero } E \\
 & | & \texttt{if } E \texttt{ then } E \texttt{ else } E \\
 & | & \texttt{let } x = E \texttt{ in } E \\
 & | & \texttt{read}
\end{array}
$$

Proc Language

Syntax of Proc

$$
\begin{array}{lcl}
P & \to & E \\
E & \to & n \\
 & | & x \\
 & | & E + E \\
 & | & E - E \\
 & | & \texttt{iszero } E \\
 & | & \texttt{if } E \texttt{ then } E \texttt{ else } E \\
 & | & \texttt{let } x = E \texttt{ in } E \\
 & | & \texttt{read} \\
 & | & \texttt{proc } x\ E \\
 & | & E\ E
\end{array}
$$

Proc Language

Semantics of Proc

- Proc 언어에서 Value, Environment를 다음과 같이 정의하자:

$$\begin{aligned} \boldsymbol{Val} &= \mathbb{Z} + \boldsymbol{Bool} + \boldsymbol{Procedure} \\ \boldsymbol{Procedure} &= \\ \boldsymbol{Env} &= \boldsymbol{Var} \rightarrow \boldsymbol{Val} \end{aligned}$$

- 기억해야 할 점: Proc 역시 value!!

- 그럼 이제 $Procedure$ 집합은 어떻게 정의해야 할까?

Proc Language
함수 파헤치기

- 프로그래밍 언어를 자연어와 대응해 보면
- **Expression**은 어떤 가치를 가지는 구절 (단어 하나가 될 수도, 문장이 될 수도)
- **Value**는 그 구절이 가지는 가치
- **Evaluation**은 그 구절이 어떤 가치를 가지는지 의미를 해석하는 과정
- **Procedure**(또는 **Function**)는?
  - 자주 사용하는 문장 구조를, 원하는 부분에 구멍 뚫어서 만들어 뒀다가 사용하는 것
  - e.g., Factorial 함수 - 1부터 OO까지 모든 자연수의 곱

Proc Language

Free/Bound Variables

- Procedure에는 두 가지의 variable이 존재할 수 있음
  - **Bound Variables:** proc의 formal parameter로 쓰여, 그 proc의 body에서는 새로 정의되지 않는 variable
  - **Free Variables:** proc에서 bound variable이 아닌 모든 variable

- `proc (y) (x+y)`
- `proc (x) (let y = 1 in x + y + z)`
- `proc (x) (proc (y) (x+y))`
- `let x = 1 in proc (y) (x+y)`
- `let x = 1 in proc (y) (x+y+z)`

Proc Language

Scoping Rule

- Procedure는 그 내용을 결정하는 정의와 실제로 실행하는 호출이 분리되어 있음
- 이에 따라 proc의 의미가 결정되는 시점이 정의인지, 호출인지에 따라 다른 결과가 나옴
- 예를 들어, 다음 코드는 어떻게 eval될 수 있을지 생각해 보자

```
let x = 1
in let f = proc (y) (x+y)
   in let x = 2
      in let g = proc (y) (x+y)
         in (f 1) + (g 1)
```

Proc Language

Scoping Rule

- **Static Scoping (Lexical Scoping)**
  - Proc이 정의되는 시점에 proc의 body가 어떤 의미를 가지는지 결정, 이후 불변
  - 이 때 의미가 결정되기 위해, 해당 시점의 Env를 따로 저장해둘 필요가 있음
- **Dynamic Scoping**
  - Proc이 정의되는 시점에는, 정말 그 body expression만이 저장될 뿐
  - Proc이 호출되는 시점의 Env를 이용해 body expression을 실행

Proc Language

Scoping Rule - Exercises

```
let a = 3
in let p = proc (z) a
   in let f = proc (x) (p 0)
      in let a = 5
         in (f 2)
```

- **Static Scoping:**
- **Dynamic Scoping:**

Proc Language

Scoping Rule - Exercises

```
let a = 3
in let p = proc (z) a
   in let f = proc (a) (p 0)
      in let a = 5
         in (f 2)
```

- **Static Scoping:**
- **Dynamic Scoping:**

Proc Language

Why Static Scoping?

- 언뜻 보기엔 dynamic scoping이 구현하기도, 사용하기도 간편해 보일 수 있다
- 그러나 최근 대부분의 프로그래밍 언어들은 static scoping을 사용
- 왜?

```
let x = 1
in let f = proc (y) (x+y)
   in let x = 2
      in let g = proc (y) (x+y)
         in (f 1) + (g 1)
```

Proc Language

Semantics of Proc - Static Scoping

- **Domain:**

$$\begin{aligned} \boldsymbol{Val} &= \mathbb{Z} + \boldsymbol{Bool} + \boldsymbol{Procedure} \\ \boldsymbol{Procedure} &= \boldsymbol{Var} \times \boldsymbol{E} \times \boldsymbol{Env} \\ \boldsymbol{Env} &= \boldsymbol{Var} \to \boldsymbol{Val} \end{aligned}$$

- **Semantic rules:**

$$\frac{}{\boldsymbol{\rho} \vdash \texttt{proc}\ \boldsymbol{x}\ \boldsymbol{E} \Rightarrow (\boldsymbol{x}, \boldsymbol{E}, \boldsymbol{\rho})}$$

$$\frac{\boldsymbol{\rho} \vdash \boldsymbol{E_1} \Rightarrow (\boldsymbol{x}, \boldsymbol{E}, \boldsymbol{\rho'}) \qquad \boldsymbol{\rho} \vdash \boldsymbol{E_2} \Rightarrow \boldsymbol{v} \qquad [\boldsymbol{x} \mapsto \boldsymbol{v}]\boldsymbol{\rho'} \vdash \boldsymbol{E} \Rightarrow \boldsymbol{v'}}{\boldsymbol{\rho} \vdash \boldsymbol{E_1}\ \boldsymbol{E_2} \Rightarrow \boldsymbol{v'}}$$

Proc Language

Examples

$$[] \vdash \texttt{(proc (x) (x)) 1} \Rightarrow$$

Proc Language

Examples

$$[] \vdash \begin{array}{l}\texttt{let x = 1}\\ \texttt{in let f = proc (y) (x+y)}\\ \quad\texttt{in let x = 2}\\ \quad\quad\texttt{in (f 3)}\end{array} \Rightarrow$$

Proc Language

Semantics of Proc - Dynamic Scoping

- **Domain:**

$$\begin{aligned} \boldsymbol{Val} &= \mathbb{Z} + \boldsymbol{Bool} + \boldsymbol{Procedure} \\ \boldsymbol{Procedure} &= \boldsymbol{Var} \times \boldsymbol{E} \\ \boldsymbol{Env} &= \boldsymbol{Var} \to \boldsymbol{Val} \end{aligned}$$

- **Semantic rules:**

$$\frac{}{\boldsymbol{\rho} \vdash \texttt{proc}\ \boldsymbol{x}\ \boldsymbol{E} \Rightarrow (\boldsymbol{x}, \boldsymbol{E})}$$

$$\frac{\boldsymbol{\rho} \vdash \boldsymbol{E_1} \vdash (\boldsymbol{x}, \boldsymbol{E}) \qquad \boldsymbol{\rho} \vdash \boldsymbol{E_2} \Rightarrow \boldsymbol{v} \qquad [\boldsymbol{x} \mapsto \boldsymbol{v}]\boldsymbol{\rho} \vdash \boldsymbol{E} \Rightarrow \boldsymbol{v'}}{\boldsymbol{\rho} \vdash \boldsymbol{E_1}\ \boldsymbol{E_2} \Rightarrow \boldsymbol{v'}}$$

Proc Language

Examples

$$[] \vdash \texttt{let x = 1}\\ \texttt{in let f = proc (y) (x+y)}\\ \texttt{\quad in let x = 2}\\ \texttt{\quad\quad in (f 3)} \Rightarrow$$

Proc Language

cf) Multiple Argument Procedures, Currying

- Proc 언어는 하나의 procedure가 여러 formal parameter를 가질 수 없어 보임
- 하지만 다음과 같이 정의하면, 충분히 여러 argument를 받는 함수처럼 구현 가능

```
let f = proc (x) proc (y) (x+y)
in ((f 3) 4)
```

- 실제 (OCaml과 같은) 함수형 언어에서는 이를 더 간편하게 사용할 수 있도록 syntactic sugar를 제공: `let f = fun x y → x + y in f 3 4`

Proc Language

cf) Multiple Argument Procedures, Currying

- 거꾸로 생각해 보면, 결국 여러 argument를 받는 함수는, 하나의 argument를 받는 함수의 body expression이 하나의 argument를 받는 함수, 또 그 함수의 body expression이 하나의 argument를 받는 함수, … 와 같은 의미를 가진다는 것
- 그렇다면, 결국 n개의 argument를 받는 함수를 호출할 때 k ($< n$)개의 argument만을 넘겨줘도 문제될 게 없다!
- $n - k$개의 argument를 받는 함수를 리턴하면 되기 때문!
- 이를 **Currying**이라 부른다

Summary

- 프로그래밍 언어는 syntax와 semantics로 구성
  - **Syntax:** context-free grammar 이용. 이 과정 역시 멋진 역사가 있지만 PL 수업에서는 스킵
  - **Semantics: Expression**과 이를 **evaluate**해서 얻을 수 있는 **value**를 정의하기 위해 집합을 사용, evaluation 과정에 필요한 **environment** 역시 집합(함수)으로 정의
- **Procedure**를 정의하기 위해 두 가지 **scoping rule**을 다뤄봄
  - **Static Scoping:** Proc definition exp 시점의 env를 저장. Proc call exp는 이 env를 사용
  - **Dynamic Scoping:** Proc definition exp의 env와 무관. Proc call exp 시점의 env를 사용