2024 한국소프트웨어종합학술대회 - 한국 프로그래밍언어 역사워크샵
(12월18일, 여수엑스포컨벤션센터)

직접 실행해보는 프로그래밍 언어 수업

최광훈

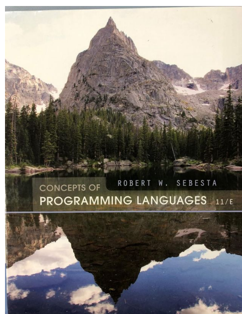
소프트웨어 언어 및 시스템 연구실
전남대학교

수업 자료 준비에는 전남대학교의 이영웅 학생과 이현진 학생이, 발표 자료 준비에는 서강대학교 문현아 박사가 도와주었습니다

프로그래밍 언어 수업

(주제1) 프로그래밍 언어 분류(Taxonomy)

- 객체지향 언어(C++, Java), 함수형 언어(ML, Haskell), 논리 언어(Prolog)

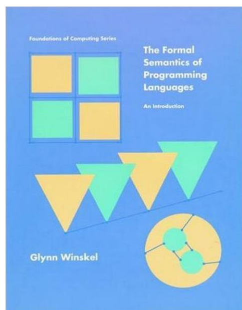


1989- 2015

프로그래밍 언어 수업의 핵심

(주제2) 프로그램의 동작을 엄밀하게 정의하고 이해하는 방법(Semantics)

수학으로
의미 정의



Rules for commands

Atomic commands:

$$\langle \mathbf{skip}, \sigma \rangle \to \sigma$$

$$\frac{\langle a, \sigma \rangle \to m}{\langle X := a, \sigma \rangle \to \sigma[m/X]}$$

Sequencing:

$$\frac{\langle c_0, \sigma \rangle \to \sigma'' \quad \langle c_1, \sigma'' \rangle \to \sigma'}{\langle c_0; c_1, \sigma \rangle \to \sigma'}$$

Conditionals:

$$\frac{\langle b, \sigma \rangle \to \mathbf{true} \quad \langle c_0, \sigma \rangle \to \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \to \sigma'}$$

$$\frac{\langle b, \sigma \rangle \to \mathbf{false} \quad \langle c_1, \sigma \rangle \to \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \to \sigma'}$$

While-loops:

$$\frac{\langle b, \sigma \rangle \to \mathbf{false}}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma \rangle \to \sigma}$$

$$\frac{\langle b, \sigma \rangle \to \mathbf{true} \quad \langle c, \sigma \rangle \to \sigma'' \quad \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma'' \rangle \to \sigma'}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma \rangle \to \sigma'}$$

“이 책은 프로그래밍 언어의 의미 정의를 처음 공부하는 학생들에게 필요한 기본적인 수학적 기법들을 제공합니다.

이 기법들은 학생들이 다양한 프로그래밍 언어에 대해 이해할 수 있는 규칙을 새로 만들고, 형식화하는데 사용할 수 있습니다.”

Proposition 2.8 *Let* $w \equiv \mathbf{while}\ b\ \mathbf{do}\ c$ *with* $b \in \mathbf{Bexp}, c \in \mathbf{Com}$. *Then*

$$w \sim \mathbf{if}\ b\ \mathbf{then}\ c; w\ \mathbf{else}\ \mathbf{skip}.$$

Proof: We want to show

$$\langle w, \sigma \rangle \to \sigma' \text{ iff } \langle \mathbf{if}\ b\ \mathbf{then}\ c; w\ \mathbf{else}\ \mathbf{skip}, \sigma \rangle \to \sigma',$$

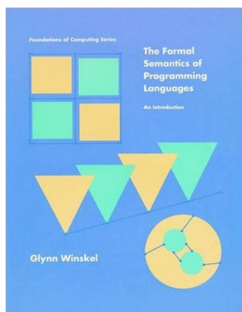
for all states σ, σ'.

프로그래밍 언어 수업의 핵심

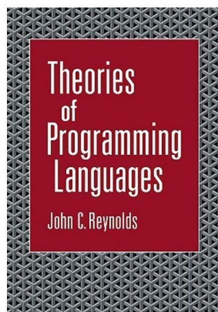
(주제2) 프로그램의 동작을 엄밀하게 정의하고 이해하는 방법

수학으로
의미 정의

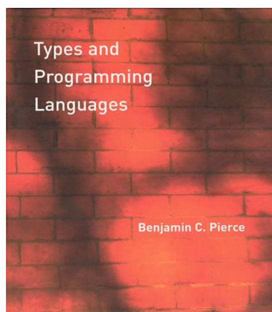
동작하는
실행기



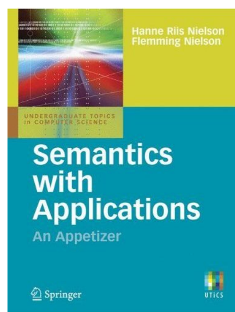
1993



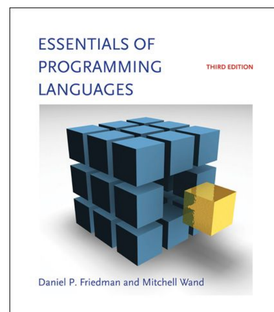
1998



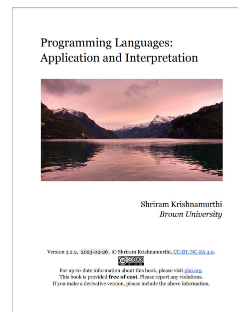
2002



1992-2007

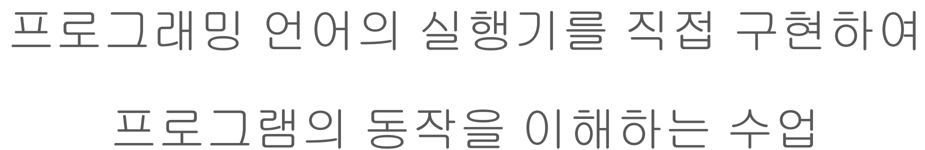


1983-2001-2008



2000~07-2023

직접 실행해보는 프로그래밍 언어 수업



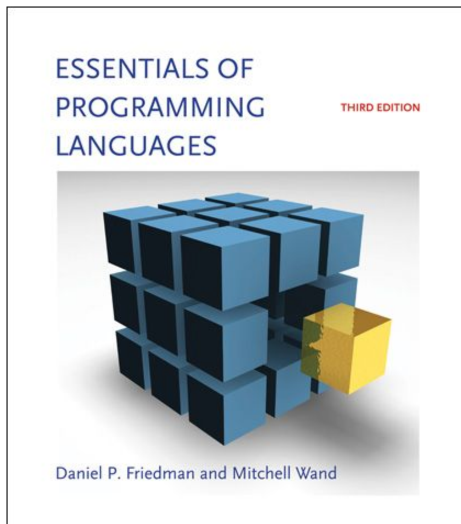
직접 실행해보는 프로그래밍 언어 수업

프로그래밍 언어의 실행기를 직접 구현하여

프로그램의 동작을 이해하는 방법

- Essentials of Programming Languages (EOPL)

  D. P. Friedman and M. Wand



EOPL 소스 코드(Scheme)
: https://github.com/mwand/eopl3

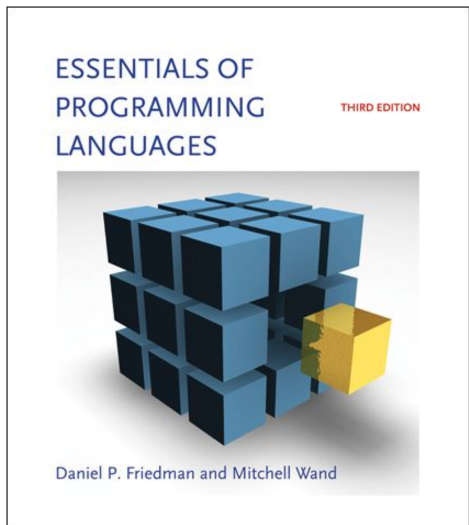
직접 실행해보는 프로그래밍 언어 수업

프로그래밍 언어 주제	이름	실행기	타입 검사기	주요 특징	Note
식 (Expression)	LET	O		프로그램 텍스트 기준 이름의 범위 (Lexical scope)	람다 계산법 (Lambda calculus)
	PROC	O		함수 정의와 호출	
	LETREC	O		재귀 함수와 Immutable 구현	
상태 (State)	EXPLICIT-REFS	O		주소를 직접 다루는 포인터 방식	C/C++ 포인터
	IMPLICIT-REFS	O		주소를 간접적으로 다루는 레퍼런스 방식	Java 레퍼런스
제어 흐름 (Continuation)	LETREC-CPS	O		'다음 실행' 개념으로 실행기 실행 순서를 구성	
	EXCEPTIONS	O		예외 처리	
	THREADS	O		멀티스레드와 뮤텍스	
타입 (Types)	CHECKED	O	O	타입 검사	
	INFERRED	O	O	유니피케이션 알고리즘 활용 간단 타입 유추 방법	
모듈 (Modules)	SIMPLE-MODULES	O	O	간단한 형태의 모듈과 인터페이스	Structure, opaque types, functors (SML and O'Caml)
	OPAQUE-TYPES	O	O	모듈 내 타입 선언 (Opaque and transparent types)	
	PROC-MODULES	O	O	모듈을 인자로 받고 반환하는 모듈 함수 (Module procedure)	
객체와 클래스 (Objects and Classes)	CLASSES	O		클래스, 객체, 인터페이스, 상속, super, self, new, overriding	Java-style inheritance
	TYPED-OO	O	O	객체지향 프로그램 타입 검사	

직접 실행해보는 프로그래밍 언어 수업

개인적인 경험담

- 애자일 방식으로 배우기
- 구현 언어로 타입 기반 순수 함수형 언어 사용



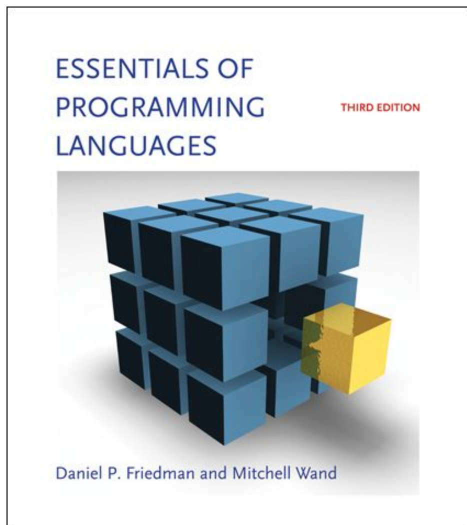
강의 자료 및 소스 코드 (Haskell)
: https://github.com/kwanghoon/Lecture_EOPL

직접 실행해보는 프로그래밍 언어 수업

개인적인 경험담

✔ 애자일 방식으로 배우기

- 구현 언어로 타입 기반 순수 함수형 언어 사용



강의 자료 및 소스 코드 (Haskell)
: https://github.com/kwanghoon/Lecture_EOPL

각 PL에 대한 강의 구조

구문 문법 정의	
핵심 문법 자료 구조 정의 (파서 제공)	
실행 의미 정의 (Dynamic Semantics)	*타입 의미 정의(Static Semantics)*
값 정의	타입 정의
실행기: 핵심 문법 구조 => 값	타입 검사기: 핵심 문법 구조 => 타입
런타임 라이브러리 (실행기는 본 기능에 집중하도록)	도움 라이브러리 (타입 검사기는 본 기능에 집중하도록)

LET 언어

새로운 프로그래밍 언어를 설명하는 전형적인 방법

- 예제 프로그램과 실행 결과를 알려주고,
- 그 이유를 설명

LET 언어

새로운 프로그래밍 언어를 설명하는 전형적인 방법

- 예제 프로그램과 실행 결과를 알려주고,
- 그 이유를 설명

=> 테스트케이스!

LET 언어

Step1. 예제+실행 결과

```
;; simple arithmetic
(positive-const "11" 11)
(negative-const "-33" -33)
(simple-arith-1 "-(44,33)" 11)

;; nested arithmetic
(nested-arith-left "-(-(44,33),22)" -11)
(nested-arith-right "-(55, -(22,11))" 44)

;; simple variables
(test-var-1 "x" 10)
(test-var-2 "-(x,1)" 9)
(test-var-3 "-(1,x)" -9)

;; simple unbound variables
(test-unbound-var-1 "foo" error)
(test-unbound-var-2 "-(x,foo)" error)
```

상수
* positive-const : 테스트케이스 이름
* “11” : 프로그램 텍스트
* 11 : 실행 결과

산술식
* nested-arith-left : 테스트케이스 이름
* “-(-(44,33),22)” : 프로그램 텍스트
* -11 : 실행 결과

변수
* test-var-1 : 테스트 케이스 이름
* “x” : 프로그램 텍스트
* 10 : 실행 결과

LET 언어

Step1. 예제+실행 결과

```
;; simple conditionals
(if-true "if zero?(0) then 3 else 4" 3)
(if-false "if zero?(1) then 3 else 4" 4)

;; test dynamic typechecking
(no-bool-to-diff-1 "-(zero?(0),1)" error)
(no-bool-to-diff-2 "-(1,zero?(0))" error)
(no-int-to-if "if 1 then 2 else 3" error)

;; make sure that the test and both arms get evaluated
;; properly.
(if-eval-test-true "if zero?(-(11,11)) then 3 else 4" 3)
(if-eval-test-false "if zero?(-(11, 12)) then 3 else 4" 4)

;; and make sure the other arm doesn't get evaluated.
(if-eval-test-true-2 "if zero?(-(11, 11)) then 3 else foo" 3)
(if-eval-test-false-2 "if zero?(-(11,12)) then foo else 4" 4)
```

조건식
* if … then … else …

0인지 검사하는 식
* zero?(…)

LET 언어

Step1. 예제+실행 결과

```
;; simple let
(simple-let-1 "let x = 3 in x" 3)

;; make sure the body and rhs get evaluated
(eval-let-body "let x = 3 in -(x,1)" 2)
(eval-let-rhs "let x = -(4,1) in -(x,1)" 2)

;; check nested let and shadowing
(simple-nested-let "let x = 3 in let y = 4 in -(x,y)" -1)
(check-shadowing-in-body "let x = 3 in let x = 4 in x" 4)
(check-shadowing-in-rhs "let x = 3 in let x = -(x,1) in x" 2)
```

지역 변수 선언식(let-in)
 * let v = … in … v …

=> LET 언어의 구문 설명과 직관적인 의미

LET 언어

Step2. 구문 정의 (BNF 문법)

```
1: Expression -> integer_number
2: Expression -> - integer_number
3: Expression -> - ( Expression , Expression )
4: Expression -> zero? ( Expression )
5: Expression -> if Expression then Expression else Expression
6: Expression -> identifier
7: Expression -> let identifier = Expression in Expression
```

[참고] 파싱 라이브러리 제안 (Haskell) : YAPB (LR 기반)

- 임진택, 김가영, 신승현, 최광훈, 김익순, LR 오토마타 생성 모듈을 공유하고 범용 프로그래밍언어로 명세를 작성하는 파서 생성 도구, 정보과학회논문지(소프트웨어및응용), Vol.47, No.1, pp52-60, 2020년 1월.

LET 언어

Step3. 핵심 문법 구조(Abstract Syntax Tree, AST)

```
type Program = Exp

data Exp =
     Const_Exp   Int
   | Diff_Exp    Exp Exp
   | IsZero_Exp  Exp
   | If_Exp      Exp Exp Exp
   | Var_Exp     Identifier
   | Let_Exp     Identifier Exp Exp

type Identifier = String
```

파싱 예:

"-(44,33)" => Diff_Exp (Const_Exp 44, Const_Exp 33)

"if zero?(1) then 3 else 4"

=> If_Exp (IsZero_Exp (Const_Exp 1))

(Const_Exp 3) (Const_Exp 4)

"let x = 3 in x" => Let_Exp "x" (Const_Exp 3) (Var_Exp "x")

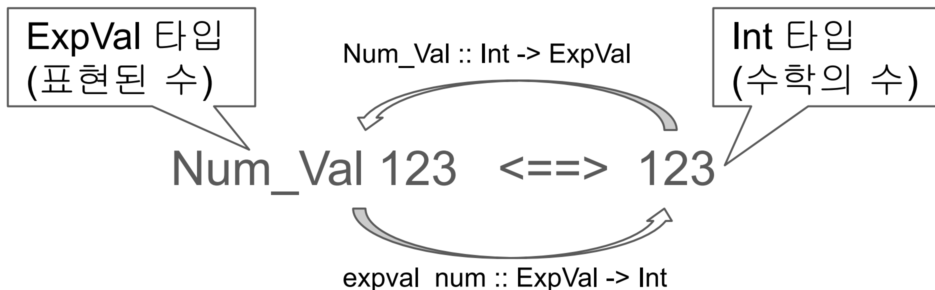
LET 언어

Step4. 값 (Expressed Value)

```
data ExpVal =
    Num_Val Int
  | Bool_Val Bool
```

-(4,1)의 결과 값
 * Num_Val 3

zero? (-(4,1))의 결과 값
 * Bool_Val False



LET 언어

Step5. 실행기 (Interpreter)

- 핵심 문법 구조 => 값
- value_of :: Exp -> Env -> ExpVal

```
value_of
  ( Diff_Exp (Const_Exp 44, Var_Exp "i") )
    (extend_env "i" (Num_Val 1) empty_env)
= Num_Val 43
```

[참고] 환경 Env : 변수 이름 => 값

```
empty_env :: Env
extend_env :: Identifier -> ExpVal -> Env -> Env
apply_env :: Env -> Identifier -> ExpVal
```

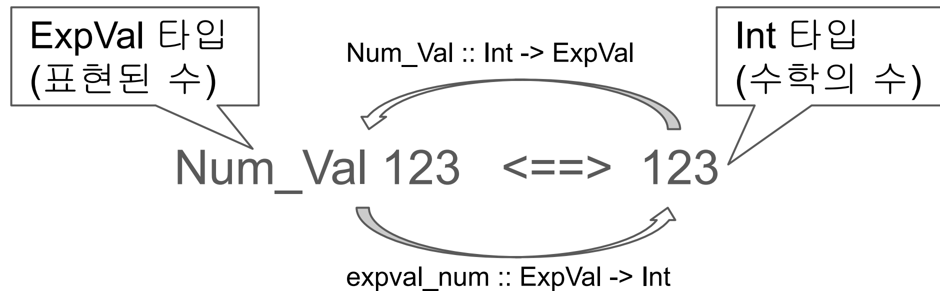
```
{ i => 1, b => true }
extend_env  "i"  (Num_Val 1)
  (extend_env   "b"  (Bool_Val True)
     empty_env)

변수  i 의 값을 위 환경에서 찾기
apply_env ( … ) "i" = Num_Val 1
```

LET 언어

Step5. 실행기 (Interpreter)

- value_of :: Exp -> Env -> ExpVal




```
value_of (Diff_Exp exp1 exp2) env =
  let val1 = value_of exp1 env
      val2 = value_of exp2 env

      num1 = expval_num val1
      num2 = expval_num val2
  in  Num_Val (num1 - num2)
```

뺄셈의 왼쪽 피연산자 exp1을 계산하고,
그 다음 오른쪽 피연산자 exp2를 계산

val1, val2 :: ExpVal

ExpVal 숫자에서 수학 숫자(Int)를 꺼내고

num1, num2 :: Int

수학 숫자(Int)를 ExpVal 숫자로 표현하고
Num_Val (num1 - num2) :: ExpVal

LET 언어

Step5. 실행기 (Interpreter)

- value_of :: Exp -> Env -> ExpVal

ExpVal 타입
(표현된 부울)

Bool_Val :: Bool -> ExpVal

Int 타입
(수학의 부울)

Bool_Val True <==> True

expval_bool :: ExpVal -> Bool

```
value_of (IsZero_Exp exp) env =
  let val1 = value_of exp env in
    let num1 = expval_num val1 in
      if num1 == 0
      then Bool_Val True
      else Bool_Val False
```

테스트 대상 피연산자 식 exp 계산
val1 :: ExpVal

ExpVal 숫자에서 '수학' 숫자(Int)를 꺼내고
num1 :: Int

그 숫자 num1가 0인지 검사

검사 결과를 ExpVal 부울로 표현
Bool_Val :: Bool -> ExpVal

LET 언어

Step6. 실행기 구현 및 테스트

```
ghci> run "11"
Const_Exp 11
11
ghci> run "-(44,33)"
Diff_Exp (Const_Exp 44) (Const_Exp 33)
11
ghci> run "zero?(0)"
IsZero_Exp (Const_Exp 0)
True
ghci> run "if zero?(1) then 3 else 4"
If_Exp (IsZero_Exp (Const_Exp 1)) (Const_Exp 3) (Const_Exp 4)
4
ghci> run "let x = 3 in x"
Let_Exp "x" (Const_Exp 3) (Var_Exp "x")
3
ghci> :q
Leaving GHCi.
```

학생들에게 실행기를 제외한 나머지 코드들(파서, 핵심 문법 구조 선언, 환경 선언 -런타임 라이브러리)를 제공하고,

테스트 케이스를 모두 통과하는 실행기 코드를 구현하는데 집중하도록 진행

LET 언어

Step6. 실행기 구현 및 테스트

```
value_of (Diff_Exp exp1 exp2) env =
```

학생들이 할 일
여기 코드를 작성하기

학생들에게 실행기를 제외한 나머지 코드들(파서, 핵심 문법 구조 선언, 환경 선언 -런타임 라이브러리)를 제공하고,

테스트 케이스를 모두 통과하는 실행기 코드를 구현하는데 집중하도록 진행

LET 언어

Step6. 실행기 (Interpreter) 테스트

- $ stack test ch3:test:letlang-test

- 모든 TC를 통과하면,

  LET 언어의 의미를 구현한 실행기

```
letlang
  ./app/letlang/examples/positive_const.let
  ./app/letlang/examples/negative_const.let
  ./app/letlang/examples/simple_arith_1.let
  ./app/letlang/examples/simple_arith_var_1.let
  ./app/letlang/examples/nested_arith_left.let
  ./app/letlang/examples/nested_arith_right.let
  ./app/letlang/examples/test_var_1.let
  ./app/letlang/examples/test_var_2.let
  ./app/letlang/examples/test_var_3.let
  ./app/letlang/examples/test_unbound_var_1.let
  ./app/letlang/examples/test_unbound_var_2.let
  ./app/letlang/examples/if_true.let
  ./app/letlang/examples/if_false.let
  ./app/letlang/examples/no_bool_to_diff_1.let
  ./app/letlang/examples/no_bool_to_diff_2.let
  ./app/letlang/examples/no_int_to_if.let
  ./app/letlang/examples/if_eval_test_true.let
  ./app/letlang/examples/if_eval_test_false.let
  ./app/letlang/examples/if_eval_test_true_2.let
  ./app/letlang/examples/if_eval_test_false_2.let
  ./app/letlang/examples/simple_let_1.let
  ./app/letlang/examples/eval_let_body.let
  ./app/letlang/examples/eval_let_rhs.let
  ./app/letlang/examples/simple_nested_let.let
  ./app/letlang/examples/check_shadowing_in_body.let
  ./app/letlang/examples/check_shadowing_in_rhs.let

Finished in 0.4150 seconds
26 examples, 0 failures
```

참고

실행기와 타입 검사/유추기의 라인수 (50~364라인)

- 실행기만 구현하도록, 그 외 나머지 코드는 모두 제공

LET (50라인)	PROC (62라인)	LETREC (64라인)	EXPLICIT-REFS (96라인)	IMPLICIT-REFS (85라인)
LETRECCPS (95라인)	EXCEPTIONS (142라인)	THREADS (203라인)	CHECKED 타입검사기(101라인)	INFERRED 타입유추기(112라인)
SIMPLEMODULES (78라인), 타입검사기 (148라인)	OPAQUETYPES (82라인), 타입검사기 (232라인)	PROCMODULES (93라인), 타입검사기 (271라인)	CLASSES (173라인)	TYPEDOO (195라인), 타입검사기(364라인)

LET 언어를 확장하여 PROC 언어 설계

-

(2): “(f 30)” 함수 호출

Step1. 예제+실행 결과

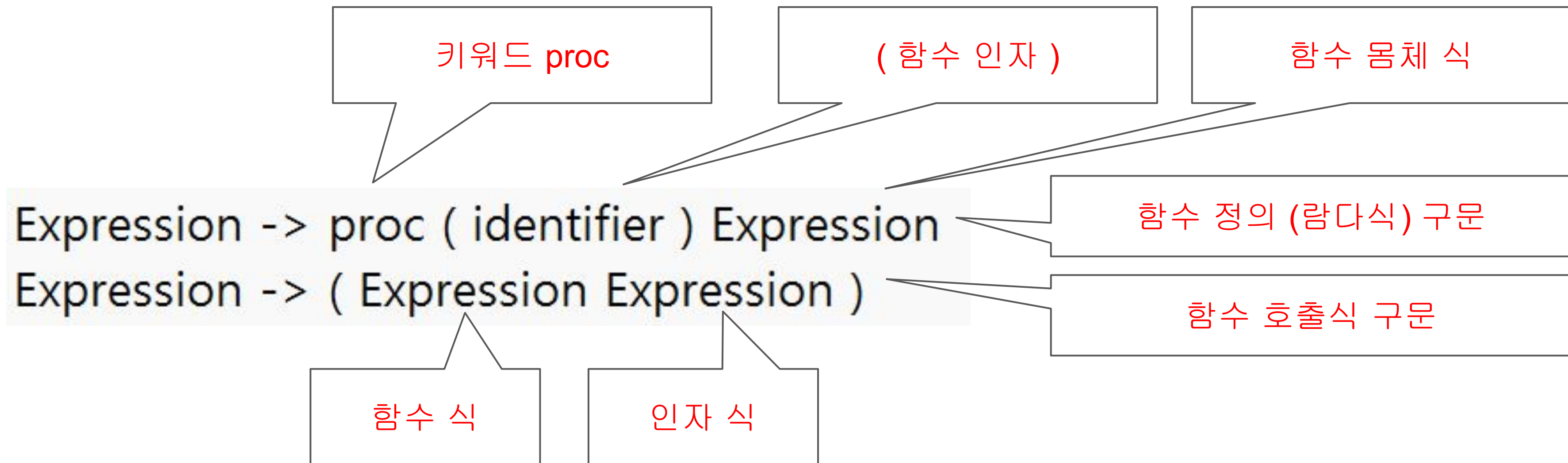
(apply-simple-proc.proc “let f = proc (x) -(x,1) in (f 30)” 29)

(1): “proc(x) -(x,1)” 함수 정의 (람다식)

LET 언어를 확장하여 PROC 언어 설계

Step2 구문 확장(BNF 문법)

- Expression -> . . .



LET 언어를 확장하여 PROC 언어 설계

Step3 핵심 문법 구조 정의 확장

-
```
type Program = Exp

data Exp =
    ...
  | Proc_Exp   Identifier Exp
  | Call_Exp   Exp Exp

type Identifier = String
```

cf.
```
Expression -> proc ( identifier ) Expression
Expression -> ( Expression Expression )
```

LET 언어를 확장하여 PROC 언어 설계

Step4 의미 값 정의 확장

-
```
data ExpVal =
    Num_Val {expval_num :: Int}
  | Bool_Val {expval_bool :: Bool}
  | Proc_Val {expval_proc :: Proc}
```

클로저 (Closure) : 코드 + 환경
- 코드 (var + body)
- 환경 (saved_env)

```
-- Procedure values : data structures
data Proc = Procedure {var :: Identifier, body :: Exp, saved_env :: Env}
```

'수학' 프로시저 (Proc)
i.e., *λval. value_of body (extend_env var val saved_env)*

LET 언어를 확장하여 PROC 언어 설계

Step5 실행기 확장

- value_of :: Exp -> Env -> ExpVal
- 함수 정의

```
value_of (Proc_Exp var body) env =
  Proc_Val (procedure var body env)
```

procedure는 클로저를 만드는 함수
(클로저: Procedure var body env)

LET 언어를 확장하여 PROC 언어 설계

Step5 실행기 확장

- value_of :: Exp -> Env -> ExpVal

ExpVal 타입

Proc 타입

Proc_Val (Procedure “x” (Var_Exp “x”) env0)
<==> (Procedure “x” (Var_Exp “x”) env0)

- 함수 호출

```
value_of (Call_Exp rator rand) env =
  let proc = expval_proc (value_of rator env)
      arg  = value_of rand env
  in apply_procedure proc arg
```

함수식 rator를 계산하고,

인자식 rand를 계산한 다음

런타임 라이브러리 apply_procedure로
- 클로저 proc에서 코드(var, body)와 환경 (saved_env)을 꺼내어
- 인자 값(var,arg)으로 확장한 환경에서 몸체 식 body를 계산

LET 언어를 확장하여 PROC 언어

Step6 확장된 실행기의 테스트

- $ stack test ch3:test:proclang-test

*.let : 회귀 테스트 (26개)
*.proc : 새로운 특징 테스트 (6개)

```
proclang
  ./app/proclang/examples/positive_const.let
  ./app/proclang/examples/negative_const.let
  ./app/proclang/examples/simple_arith_1.let
  ./app/proclang/examples/simple_arith_var_1.let
  ./app/proclang/examples/nested_arith_left.let
  ./app/proclang/examples/nested_arith_right.let
  ./app/proclang/examples/test_var_1.let
  ./app/proclang/examples/test_var_2.let
  ./app/proclang/examples/test_var_3.let
  ./app/proclang/examples/test_unbound_var_1.let
  ./app/proclang/examples/test_unbound_var_2.let
  ./app/proclang/examples/if_true.let
  ./app/proclang/examples/if_false.let
  ./app/proclang/examples/no_bool_to_diff_1.let
  ./app/proclang/examples/no_bool_to_diff_2.let
  ./app/proclang/examples/no_int_to_if.let
  ./app/proclang/examples/if_eval_test_true.let
  ./app/proclang/examples/if_eval_test_false.let
  ./app/proclang/examples/if_eval_test_true_2.let
  ./app/proclang/examples/if_eval_test_false_2.let
  ./app/proclang/examples/simple_let_1.let
  ./app/proclang/examples/eval_let_body.let
  ./app/proclang/examples/eval_let_rhs.let
  ./app/proclang/examples/simple_nested_let.let
  ./app/proclang/examples/check_shadowing_in_body.let
  ./app/proclang/examples/check_shadowing_in_rhs.let
  ./app/proclang/examples/apply_proc_in_rator_pos.proc
  ./app/proclang/examples/apply_simple_proc.proc
  ./app/proclang/examples/let_to_proc_1.proc
  ./app/proclang/examples/nested_procs_1.proc
  ./app/proclang/examples/nested_procs_2.proc
  ./app/proclang/examples/y_combinator_1.proc

Finished in 0.4824 seconds
32 examples, 0 failures
```

애자일 방식으로 배우기

(1) 의미를 정의하는 수학 표기법 규칙보다 실제로 동작하는 실행기!

- cf. 작동하는 소프트웨어를 상세한 문서보다 중시하자! (애자일 선언문)

  Working software over comprehensive documentation (https://agilemanifesto.org/)

(2) 테스트케이스로 프로그램 동작을 정의

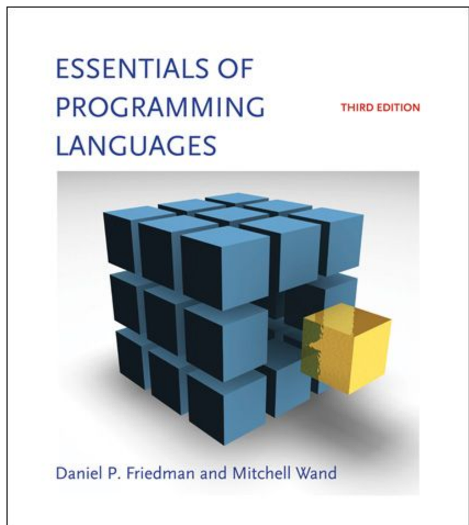
- cf. 테스트 주도 개발 (테스트케이스를 작성하고 이에 맞추어 코드를 개발)

직접 실행해보는 프로그래밍 언어 수업

개인적인 경험담

- 애자일 방식으로 배우기

✔ 구현 언어로 타입 기반 순수 함수형 언어 사용



강의 자료 및 소스 코드 (Haskell)
: https://github.com/kwanghoon/Lecture_EOPL

구현 언어로 타입 기반 순수 함수형 언어 사용

장점1: 조립식 타입 (Algebraic data types)

- 합집합 + 곱집합 + 재귀적 집합을 조립하여 구성
- 프로그래밍 언어 요소를 (수학 집합으로) 표현하는 도구

장점2: 구현 언어의 타입으로 대상 언어를 설명하기 쉬움

- 실행기나 타입 검사기 구조를 타입으로 설명
- 프로그래밍 언어 요소들을 타입으로 구분

구현 언어로 타입 기반 순수 함수형 언어 사용

장점1: 조립식 타입 (Algebraic data types)

- 합집합 + 곱집합 + 재귀적 집합을 조립하여 구성

- 프로그래밍 언어 요소를 (수학 집합으로) 표현하는 도구

- 예) 핵심 문법 구조 (Exp), 의미 값 (ExpVal), 타입 (Type), 컨티뉴에이션 (Cont), 모듈 (TypedModule), 클래스 (Class)

=> 조립식 타입 사례 (https://github.com/kwanghoon/Lecture_EOPL)

구현 언어로 타입 기반 순수 함수형 언어 사용

장점1: 조립식 타입 (Algebraic data types)

- 합집합 + 곱집합 + 재귀적 집합을 조립하여 구성
- 프로그래밍 언어 요소를 (수학 집합으로) 표현하는 도구

구현 언어에 따른 지원 방법

- 타입 기반 함수형 언어 (e.g., ML, Haskell) => 조립식 타입 직접 지원
- 타입 없는 함수형 언어 (e.g., Scheme) => 조립식 타입을 흉내내는 Library
- 객체지향 언어 (e.g., Scala) => 클래스 상속으로 조립식 타입을 흉내냄

구현 언어로 타입 기반 순수 함수형 언어 사용

장점2: 구현 언어의 타입으로 대상 언어를 설명하기 쉬움

- 실행기나 타입 검사기 구조를 타입으로 설명
- 프로그래밍 언어 요소들을 타입으로 구분

실행기 구조를 타입으로 설명 가능

- LET, PROC, LETREC 실행기 : Exp -> Env -> ExpVal

Store passing style 실행기
- Exp를 계산하기 전 Store를 받아서
- Exp를 계산하는 동안 바뀐 Store를 리턴

- EXPLICIT-REF, IMPLICIT-REFS 실행기 : Exp -> Env -> Store -> (ExpVal, Store)
- LETRECCPS 실행기 : Exp -> Env -> Cont -> ExpVal

Exp를 계산하고,
Cont까지 계산한 결과를 리턴

구현 언어로 타입 기반 순수 함수형 언어 사용

장점2: 구현 언어의 타입으로 대상 언어를 설명하기 쉬움

- 실행기나 타입 검사기 구조를 타입으로 설명
- 프로그래밍 언어 요소들을 타입으로 구분

타입 검사기 구조와 Side effect도 타입으로 설명

- 타입 검사기 : Exp -> TyEnv -> Either ErrMsg Type
- 타입 유추기 : Exp -> TyEnv -> Subst -> Either_ ErrMsg (Type, Subst)

type Either_ e a = State Int (Either e a)
- 전역 변수 int 값을 유지하면서
  에러 e 값 또는 정상 a 값을 리턴

- Call_Exp rator rand 식에서 rator와 rand의
  타입으로 ty1, ty2라 유추한 다음
  unify (ty1, ty2 -> *fresh tyvar*)

구현 언어로 타입 기반 순수 함수형 언어 사용

장점2: 구현 언어의 타입으로 대상 언어를 설명하기 쉬움

- 실행기나 타입 검사기 구조를 타입으로 설명
- 프로그래밍 언어 요소들을 타입으로 구분

스레드의 타입

- Store -> SchedState -> (ExpVal, Store)

SchedState (스케쥴러 상태):
현재 스레드 실행의 남은 시간, 실행 준비가 된 스레드들 대기 큐

구현 언어로 타입 기반 순수 함수형 언어 사용

장점2: 구현 언어의 타입으로 대상 언어를 설명하기 쉬움

- 실행기나 타입 검사기 구조를 타입으로 설명
- 프로그래밍 언어 요소들을 타입으로 구분

예1) 구문 vs. 값 :

프로그램 식 구문 123 => Const_Exp 123 :: Exp

프로그램 결과 값 123 => Num_Val 123 :: ExpVal

구현 언어로 타입 기반 순수 함수형 언어 사용

장점2: 구현 언어의 타입으로 대상 언어를 설명하기 쉬움

- 실행기나 타입 검사기 구조를 타입으로 설명
- 프로그래밍 언어 요소들을 타입으로 구분

예) 값 vs. 값 표현:

의도하는 값 => 123 :: Int

실행기 결과 => Num_Val 123 :: ExpVal

e.g., Num_Val 123 + Num_Val 456 => Num_Val (123 + 456)

구현 언어로 타입 기반 순수 함수형 언어 사용

장점2: 구현 언어의 타입으로 대상 언어를 설명하기 쉬움

- 실행기나 타입 검사기 구조를 타입으로 설명
- 프로그래밍 언어 요소들을 타입으로 구분

예) 느슨하게 정의한 실행기 구조 문제: THREADS 프로그래밍 언어의 ExpVal

프로그램 식을 계산해서 나오는 값이 아닌 큐가 포함

(스토어에 큐를 저장하는 이유로 포함됨)

ExpVal = Int + Bool + Proc + List(ExpVal) + Mutex + Queue

논의: 파서 작성하는 문제

새로운 구문을 정의하는 프로젝트의 경우 파서 작성도 필요

- 예) IMPLICIT-REFS 언어에 Immutable 변수(let x = exp in …)와 Mutable 변수(letmut x = exp in …)를 추가하여 명시적으로 선언하도록 확장

=> 파서 작성 (letmut 키워드 도입)

=> AST 확장(let과 letmut을 구분하여 표현)

=> 실행기/타입 검사기 확장 (let과 letmut의 코드 각각 구현)

(필요한 경우 ExpVal을 확장)

논의: 파서 작성하는 문제 (계속)

동일한 구현 언어로 1)파서와 2)실행기를 작성하는 파싱 라이브러리

	EOPL	EOPL in Haskell
파싱 라이브러리	SLLGEN[(1)]	YAPB[(2)]
구현 언어	스킴	하스켈
파싱 방법	LL	LR
문법과 AST	문법과 AST 선언이 동일	문법과 AST 선언 분리

=> 파서 작성을 위해 별도의 언어(예: YACC)를 사용하지 않아도 되는 장점

(1) Daniel Friedman, Mitchel Wand, Essentials of Programming Languages, 3rd Ed., 2007.
(2) 임진택, 김가영, 신승현, 최광훈, 김익순, LR 오토마타 생성 모듈을 공유하고 범용 프로그래밍언어로명세를 작성하는 파서 생성 도구, 정보과학회논문지(소프트웨어및응용), Vol.47, No.1, pp52-60, 2020년 1월.

논의: 파서 작성하는 문제 (계속)

LET 언어의 문법 명세 (SLLGEN in SCHEME)

```
(define the-lexical-spec
  '((whitespace (whitespace) skip)
    (comment ("%" (arbno (not #\newline))) skip)
    (identifier
     (letter (arbno (or letter digit "_" "-" "?")))
     symbol)
    (number (digit (arbno digit)) number)
    (number ("-" digit (arbno digit)) number)
    ))
```

```
(define the-grammar
  '((program (expression) a-program)

    (expression (number) const-exp)
    (expression
      ("-" "(" expression "," expression ")")
      diff-exp)

    (expression
     ("zero?" "(" expression ")")
     zero?-exp)

    (expression
     ("if" expression "then" expression "else" expression)
     if-exp)

    (expression (identifier) var-exp)

    (expression
     ("let" identifier "=" expression "in" expression)
     let-exp)

    ))
```

논의: 파서 작성하는 문제 (계속)

LET 언어의 문법 명세 (YAPB in HASKELL)

```
lexerSpec :: LexerSpec Token IO ()
lexerSpec = LexerSpec
  {
    endOfToken    = END_OF_TOKEN,
    lexerSpecList =
      [ ("[ \t\n]" , skip),

        ("[0-9]+"  , mkFn INTEGER_NUMBER),

        ("\\-"     , mkFn SUB),
        ("\\("     , mkFn OPEN_PAREN),
        ("\\)"     , mkFn CLOSE_PAREN),
        ("\\,"     , mkFn COMMA),

        ("zero\\?" , mkFn ISZERO),

        ("if"      , mkFn IF),
        ("then"    , mkFn THEN),
        ("else"    , mkFn ELSE),

        ("let"     , mkFn LET),
        ("in"      , mkFn IN),
        ("\\="     , mkFn EQ),

        ("[a-zA-Z][a-zA-Z0-9]*"    , mkFn IDENTIFIER)
      ]
  }
```

```
parserSpec :: ParserSpec Token Exp IO ()
parserSpec = ParserSpec
  {
    startSymbol = "Expression'",

    tokenPrecAssoc = [],

    parserSpecList =
    [
      rule "Expression' -> Expression" (\rhs -> return $ get rhs 1),

      rule "Expression -> integer_number"
        (\rhs -> return $ Const_Exp (read (getText rhs 1) :: Int)),

      rule "Expression -> - integer_number"
        (\rhs -> return $ Const_Exp (-(read (getText rhs 2) :: Int))),

      rule "Expression -> - ( Expression , Expression )"
        (\rhs -> return $ Diff_Exp (get rhs 3) (get rhs 5)),

      rule "Expression -> zero? ( Expression )"
        (\rhs -> return $ IsZero_Exp (get rhs 3)),

      rule "Expression -> if Expression then Expression else Expression"
        (\rhs -> return $ If_Exp (get rhs 2) (get rhs 4) (get rhs 6)),

      rule "Expression -> identifier" (\rhs -> return $ Var_Exp (getText rhs 1)),

      rule "Expression -> let identifier = Expression in Expression"
        (\rhs -> return $ Let_Exp (getText rhs 2) (get rhs 4) (get rhs 6))
    ],
```

논의: 프로그래밍 언어 공부의 틀

프로그래밍 언어 설계 및 구현 프로젝트 중심의 방법

새로 연구되는 프로그래밍 언어의 프로젝트를 기존 15가지 언어 중 하나를 확장하여 구성

논의: 구현 언어로 왜 하스켈인가?

구현 언어, 하스켈 사용

- 조립식 타입(algebraic data type), 타입, 패턴 매칭 기반 간결한 함수 정의만 사용
- 실행기나 타입 검사기를 다른 PL로 쉽게 재작성할 수 있도록 작성
- 지연 계산(laziness)이나 타입 클래스 등 하스켈 의존적인 특징을 전혀 사용 안함

왜 하스켈?

- 수학에 가까운 프로그래밍 언어(전역 변수가 없어서) cf. 수학으로 의미 정의
- 그 결과, 실행기나 타입 검사기 구조가 타입에 분명하게 반영되어 설명하기 쉬움

논의: 실행기 기반 수업 방식이 항상 옳은가?

실행기 기반 방식과 수학 기반 의미 정의 방식의 적절한 균형 필요

- 간단한 PL: 실행기 구현을 활용 => 수학 표기법 의미 정의를 설명
- 복잡한 PL: 수학 표기법 의미 정의를 활용 => 실행기 구현을 설명

수학 표기법 보다 더 간결한 구문의 구현 언어가 존재할까?

결국, 수학이 가장 간결한 구현 언어!

맺음말

실행기로 배우는 프로그래밍 언어 수업 경험담

- 애자일 방식으로 배우기
- 구현 언어로 타입 기반 순수 함수형 언어의 장점

=> 프로그래밍 언어를 배우는 틀로 활용 가능성

[발표 자료]

EOPL in Haskell 강의 자료 및 소스 코드

- https://github.com/kwanghoon/Lecture_EOPL