Chapter 5: Functions & Abstraction

이 장에서는 FunLang에 함수를 추가한다. 람다 표현식, 함수 호출, 재귀 함수, 클로저를 구현하여 언어를 Turing-complete하게 만든다.

개요

함수형 프로그래밍의 핵심인 일급 함수(first-class functions)를 구현한다: - **람다 표현식**: `fun x -> x + 1` - **함수 호출**: `f 5` - **재귀 함수**: `let rec fact n = ...` - **클로저**: 함수가 정의 시점의 환경을 캡처

AST 확장

Value 타입에 FunctionValue 추가

함수도 값이다. 함수 값은 매개변수, 본문, 그리고 **클로저**(정의 시점의 환경)를 포함한다.

```
// FunLang/Ast.fs

/// Value type for evaluation results
and Value =
    | IntValue of int
    | BoolValue of bool
    | FunctionValue of param: string * body: Expr * closure: Env

/// Environment mapping variable names to values
and Env = Map<string, Value>
```

Value와 Env는 상호 참조하므로 and 키워드로 상호 재귀 타입을 정의한다.

Expr 타입에 함수 표현식 추가

```
type Expr =
    // ... 기존 케이스들 ...

    // Phase 5: Functions
    | Lambda of param: string * body: Expr
    | App of func: Expr * arg: Expr
    | LetRec of name: string * param: string * body: Expr * inExpr: Expr
```

케이스	구문	설명
Lambda	fun x -> body	익명 함수 생성
App	f arg	함수 호출
LetRec	let rec f x = body in expr	재귀 함수 정의

Lexer 확장

새로운 키워드와 연산자를 추가한다.

```
// FunLang/Lexer.fsl
```

```
// Phase 5: Function keywords
| "fun"          { FUN }
| "rec"          { REC }

// Phase 5: Arrow for lambda expressions (multi-char, before single-char operators)
| "->"           { ARROW }
```

중요: -> 규칙은 - 규칙보다 먼저 와야 한다. 그렇지 않으면 -> 가 MINUS GT로 잘못 토큰화된다.

Parser 확장

토큰 선언

```
// FunLang/Parser.fsy

%token FUN REC ARROW
```

문법 규칙

```
Expr:
    // ... 기존 규칙들 ...

    // Phase 5: Function definitions
    | LET REC IDENT IDENT EQUALS Expr IN Expr  { LetRec($3, $4, $6, $8) }
    | FUN IDENT ARROW Expr                     { Lambda($2, $4) }
```

함수 호출과 연산자 우선순위

함수 호출은 juxtaposition(나란히 쓰기)으로 표현된다: `f 5`. 이는 가장 높은 우선순위를 가진다.

문제: `f - 1`은 어떻게 해석해야 하는가? - 뺄셈: f에서 1을 뺀다 - 함수 호출: f에 -1을 전달한다

해결책: Atom 비단말을 도입하여 함수 인자에서 단항 마이너스를 제외한다.

```
// Factor: 단항 마이너스 포함
Factor:
    | MINUS Factor       { Negate($2) }
    | AppExpr            { $1 }

// AppExpr: 함수 호출 (left-associative)
// Atom만 인자로 받음 - 단항 마이너스 제외
AppExpr:
    | AppExpr Atom       { App($1, $2) }
    | Atom               { $1 }

// Atom: 가장 기본적인 표현식 (연산자 없음)
Atom:
    | NUMBER             { Number($1) }
    | IDENT              { Var($1) }
    | TRUE               { Bool(true) }
```

```
    | FALSE              { Bool(false) }
    | LPAREN Expr RPAREN { $2 }
```

이 구조로: `-f - 1` → 뺄셈 (`Subtract(Var "f", Number 1)`) `-f (-1)` → 함수 호출 (`App(Var "f", Negate(Number 1))`) `-f 1 2` → 커링 (`App(App(Var "f", Number 1), Number 2)`)

Evaluator 구현

Lambda: 클로저 생성

람다 표현식을 평가하면 현재 환경을 캡처한 클로저를 생성한다.

```
// FunLang/Eval.fs

| Lambda (param, body) ->
    FunctionValue (param, body, env)
```

핵심: 함수 값은 정의 시점의 환경(env)을 캡처한다. 이것이 클로저의 본질이다.

App: 함수 호출

함수를 호출할 때는 **클로저의 환경**을 확장한다 (호출 시점의 환경이 아님).

```
| App (funcExpr, argExpr) ->
    let funcVal = eval env funcExpr
    match funcVal with
    | FunctionValue (param, body, closureEnv) ->
        let argValue = eval env argExpr
        // 재귀 함수: 자기 자신을 클로저에 추가
        let augmentedClosureEnv =
            match funcExpr with
            | Var name -> Map.add name funcVal closureEnv
            | _ -> closureEnv
        let callEnv = Map.add param argValue augmentedClosureEnv
        eval callEnv body
    | _ -> failwith "Type error: attempted to call non-function"
```

중요 포인트: 1. 인자는 **호출 시점**의 환경에서 평가 2. 본문은 **클로저의 환경**을 확장하여 평가 3. 재귀 함수는 자신을 클로저에 추가하여 자기 참조 가능

LetRec: 재귀 함수 정의

```
| LetRec (name, param, funcBody, inExpr) ->
    let funcVal = FunctionValue (param, funcBody, env)
    let recEnv = Map.add name funcVal env
    eval recEnv inExpr
```

재귀가 작동하는 이유: 1. LetRec은 함수를 환경에 바인딩 2. App에서 변수로 함수를 호출할 때 자기 자신을 클로저에 추가 3. 함수 본문에서 자신의 이름으로 재귀 호출 가능

formatValue: 함수 출력

```
let formatValue (v: Value) : string =
    match v with
    | IntValue n -> string n
    | BoolValue b -> if b then "true" else "false"
    | FunctionValue _ -> "<function>"
```

함수는 <function>으로 표시한다.

클로저의 이해

클로저는 함수가 **정의 시점의 환경**을 기억하는 것이다.

```
let x = 10 in
let f = fun y -> x + y in
let x = 100 in
f 5
```

결과는 15이다 (115가 아님). f가 정의될 때 x = 10이었고, 나중에 x = 100으로 섀도잉되어도 f의 클로저에는 여전히 x = 10이 저장되어 있다.

클로저 vs 동적 스코프

스코핑	설명	위 예제 결과
렉시컬 스코프 (클로저)	정의 시점 환경 사용	15
동적 스코프	호출 시점 환경 사용	115

FunLang은 렉시컬 스코프를 사용한다 (대부분의 현대 언어와 동일).

커링 (Currying)

다중 매개변수 함수는 커링으로 구현한다:

```
let add = fun x -> fun y -> x + y in
add 3 4
```

이것은 다음과 같이 동작한다: 1. add 3 → fun y -> 3 + y (클로저에 x = 3 저장) 2. (add 3) 4 → 3 + 4 → 7

Examples

기본 람다

```
$ dotnet run --project FunLang -- --expr "fun x -> x"
<function>

$ dotnet run --project FunLang -- --expr "fun x -> x + 1"
<function>
```

함수 바인딩과 호출

```
$ dotnet run --project FunLang -- --expr "let f = fun x -> x + 1 in f 5"
6

$ dotnet run --project FunLang -- --expr "let double = fun x -> x * 2 in double 7"
14
```

커링과 부분 적용

```
$ dotnet run --project FunLang -- --expr "let add = fun x -> fun y -> x + y in add 3 4"
7

$ dotnet run --project FunLang -- --expr "let add = fun x -> fun y -> x + y in let add5 = add 5 in add5 10"
15
```

중첩 함수 호출

```
$ dotnet run --project FunLang -- --expr "let f = fun x -> x + 1 in f (f (f 0))"
3
```

재귀 함수: Factorial

```
$ dotnet run --project FunLang -- --expr "let rec fact n = if n <= 1 then 1 else n * fact (n - 1) in fact 5"
120
```

재귀 함수: Fibonacci

```
$ dotnet run --project FunLang -- --expr "let rec fib n = if n <= 1 then n else fib (n - 1) + fib (n - 2) in
8

$ dotnet run --project FunLang -- --expr "let rec fib n = if n <= 1 then n else fib (n - 1) + fib (n - 2) in
55
```

클로저: 기본

```
$ dotnet run --project FunLang -- --expr "let x = 10 in let f = fun y -> x + y in f 5"
15
```

클로저: 중첩 바인딩

```
$ dotnet run --project FunLang -- --expr "let a = 1 in let b = 2 in let f = fun x -> a + b + x in f 3"
6
```

클로저: makeAdder 패턴

```
$ dotnet run --project FunLang -- --expr "let makeAdder = fun x -> fun y -> x + y in let add5 = makeAdder 5
8
```

클로저: 값 캡처 (섀도잉 불변)

```
$ dotnet run --project FunLang -- --expr "let x = 1 in let f = fun y -> x + y in let x = 100  n f 5"
6
```

f는 정의 시점의 x = 1을 캡처했으므로 나중에 x = 100으로 섀도잉되어도 영향받지 않는다.

함수 호출 vs 뺄셈

```
# 뺄셈: f - 1
$ dotnet run --project FunLang -- --expr "let f = 10 in f - 1"
9

# 함수 호출: f (-1)
$ dotnet run --project FunLang -- --expr "let f = fun x -> x * 2 in f (-1)"
-2
```

AST 확인

```
$ dotnet run --project FunLang -- --emit-ast --expr "fun x -> x + 1"
Lambda ("x", Add (Var "x", Number 1))

$ dotnet run --project FunLang -- --emit-ast --expr "f 5"
App (Var "f", Number 5)

$ dotnet run --project FunLang -- --emit-ast --expr "let rec f x = x in f 1"
LetRec ("f", "x", Var "x", App (Var "f", Number 1))
```

토큰 확인

```
$ dotnet run --project FunLang -- --emit-tokens --expr "fun x -> x + 1"
FUN IDENT(x) ARROW IDENT(x) PLUS NUMBER(1) EOF
```

타입 에러

```
# 함수가 아닌 값을 호출
$ dotnet run --project FunLang -- --expr "5 3"
Type error: attempted to call non-function
```

```
$ dotnet run --project FunLang -- --expr "true 1"
Type error: attempted to call non-function
```

정리

이 장에서 구현한 내용:

기능	구문	예시
람다 표현식	`fun param -> body`	`fun x -> x + 1`
함수 호출	`func arg`	`f 5`
재귀 함수	`let rec name param = body in expr`	`let rec fact n = ... in fact 5`
클로저	자동 (정의 시점 환경 캡처)	`let x = 10 in fun y -> x + y`
커링	중첩 람다	`fun x -> fun y -> x + y`

FunLang은 이제 Turing-complete 언어이다. 재귀 함수를 통해 모든 계산 가능한 함수를 표현할 수 있다.

테스트

```
# fslit 테스트 (13개 함수 테스트 포함)
make -C tests

# Expecto 단위 테스트
dotnet run --project FunLang.Tests
```

전체 소스 코드는 FunLang/ 디렉토리를 참조한다.

관련 문서

- resolve-function-application-vs-subtraction - 함수 호출과 뺄셈 문법 충돌 해결
- implement-unary-minus - 단항 마이너스 구현
- adapt-tests-for-value-type-evolution - Value 타입 변경 시 테스트 적응