Chapter 9: Prelude (Standard Library)

이 장에서는 FunLang의 표준 라이브러리(Prelude)를 구현한다. Prelude는 자주 사용되는 리스트 조작 함수와 유틸리티를 제공하며, **FunLang 자체로 작성**되어 언어의 표현력을 증명한다.

개요

Prelude는 다음 기능을 제공한다:

- **고차 함수**: `map`, `filter`, `fold` (리스트 처리의 핵심)
- **리스트 유틸리티**: `length`, `reverse`, `append`
- **리스트 접근**: `hd`, `tl` (head, tail)
- **Combinators**: `id`, `const`, `compose` (함수 조합)

핵심 아이디어: Prelude는 FunLang으로 작성되고, 인터프리터 시작 시 자동으로 로드된다. 이를 **self-hosted standard library**라고 한다.

구현 전략

Prelude 구현은 두 부분으로 나뉜다:

1. **Prelude.fun**: 표준 라이브러리 함수들을 FunLang으로 작성
2. **Prelude.fs**: Prelude.fun을 파싱하고 환경에 로드하는 인프라

Self-Hosting의 장점

Prelude를 FunLang 자체로 작성하는 것은 여러 이점이 있다:

- **언어 능력 증명**: FunLang이 충분히 표현력 있음을 보여줌
- **일관성**: 사용자 코드와 동일한 문법/의미론 사용
- **유지보수성**: 표준 라이브러리를 언어 자체로 확장 가능
- **교육적 가치**: 표준 라이브러리가 "마법"이 아니라 일반 코드임을 보여줌

Prelude.fun: 표준 라이브러리 소스

`Prelude.fun` 파일은 FunLang으로 작성된 표준 라이브러리다. 중첩된 `let ... in` 구조로 함수들을 정의한다.

고차 함수

map: 리스트 변환

```
let rec map f = fun xs ->
    match xs with
    | [] -> []
    | h :: t -> (f h) :: (map f t)
in
```

`map`은 함수를 리스트의 각 요소에 적용하여 새 리스트를 생성한다.

- 빈 리스트는 빈 리스트로 매핑
- 비어있지 않은 리스트는 head에 함수 적용 후 tail을 재귀적으로 매핑

타입: `('a -> 'b) -> 'a list -> 'b list`

filter: 조건으로 필터링

```
let rec filter pred = fun xs ->
    match xs with
    | [] -> []
    | h :: t -> if pred h then h :: (filter pred t) else filter pred t
in
```

filter는 조건을 만족하는 요소만 남긴다.

- 빈 리스트는 빈 리스트 반환
- head가 조건 만족 시 결과에 포함, 아니면 스킵

타입: `('a -> bool) -> 'a list -> 'a list`

fold: 리스트 누적

```
let rec fold f = fun acc -> fun xs ->
    match xs with
    | [] -> acc
    | h :: t -> fold f (f acc h) t
in
```

fold는 리스트를 왼쪽에서 오른쪽으로 순회하며 누적값을 계산한다 (left fold).

- 빈 리스트는 누적값 반환
- head와 누적값을 함수에 전달하여 새 누적값 생성

타입: `('acc -> 'a -> 'acc) -> 'acc -> 'a list -> 'acc`

리스트 유틸리티

length: 리스트 길이

```
let rec length xs =
    match xs with
    | [] -> 0
    | _ :: t -> 1 + (length t)
in
```

재귀적으로 리스트를 순회하며 길이를 센다.

타입: `'a list -> int`

reverse: 리스트 뒤집기

```
let reverse = fun xs ->
    let rec rev_acc acc = fun ys ->
        match ys with
        | [] -> acc
        | h :: t -> rev_acc (h :: acc) t
    in
    rev_acc [] xs
in
```

누적 리스트를 사용하여 효율적으로 뒤집는다 (tail-recursive).

- 내부 함수 rev_acc는 누적 리스트에 요소를 cons하며 순회
- `[1, 2, 3]` → `rev_acc [] [1,2,3]` → `rev_acc [1] [2,3]` → `rev_acc [2,1] [3]` → `[3,2,1]`

타입: `'a list -> 'a list`

append: 리스트 연결

```
let rec append xs = fun ys ->
    match xs with
    | [] -> ys
    | h :: t -> h :: (append t ys)
in
```

두 리스트를 연결한다.

- 첫 번째 리스트가 빈 리스트면 두 번째 리스트 반환
- 그렇지 않으면 첫 번째 리스트의 head를 cons하고 tail과 두 번째 리스트를 재귀적으로 연결

타입: `'a list -> 'a list -> 'a list`

리스트 접근

hd: 리스트의 첫 요소

```
let hd = fun xs ->
    match xs with
    | h :: _ -> h
in
```

리스트의 head를 반환한다. 빈 리스트에 대해서는 매칭 실패 에러가 발생한다.

타입: `'a list -> 'a`

tl: 리스트의 나머지

```
let tl = fun xs ->
    match xs with
    | _ :: t -> t
in
```

리스트의 tail을 반환한다. 빈 리스트에 대해서는 매칭 실패 에러가 발생한다.

타입: `'a list -> 'a list`

Combinators

id: 항등 함수

```
let id = fun x -> x
in
```

입력을 그대로 반환한다.

타입: `'a -> 'a`

const: 상수 함수

```
let const = fun x -> fun y -> x
in
```

두 개의 인자를 받아 첫 번째 인자를 반환한다. 두 번째 인자는 무시된다.

타입: `'a -> 'b -> 'a`

compose: 함수 합성

```
let compose = fun f -> fun g -> fun x -> f (g x)
in
```

두 함수를 합성한다. g를 먼저 적용하고 그 결과에 f를 적용한다.

타입: `('b -> 'c) -> ('a -> 'b) -> ('a -> 'c)`

구조

Prelude.fun은 중첩된 `let ... in` 구조를 사용한다:

```
let map = ... in
let filter = ... in
let fold = ... in
...
let compose = ... in
0
```

마지막 표현식 0은 더미 값이다. Prelude 로딩 시 이 값은 버려지고 환경만 추출된다.

Prelude.fs: 로딩 인프라

FunLang/Prelude.fs 모듈은 Prelude.fun을 파싱하고 초기 환경에 로드한다.

evalToEnv: 환경 추출

evalToEnv는 중첩된 `let ... in` 구조를 순회하며 바인딩을 환경에 누적한다.

```fsharp
let rec private evalToEnv (env: Env) (expr: Expr) : Env =
    match expr with
    | Let (name, binding, body) ->
        let value = eval env binding
        let extendedEnv = Map.add name value env
        evalToEnv extendedEnv body
    | LetRec (name, param, funcBody, inExpr) ->
        let funcVal = FunctionValue (param, funcBody, env)
        let extendedEnv = Map.add name funcVal env
        evalToEnv extendedEnv inExpr
    | _ ->
        // Base case: return accumulated environment (final expr is discarded)
        env
```

동작 방식:

1. Let 또는 LetRec 표현식을 만나면 바인딩을 평가
2. 환경을 확장하고 body를 재귀적으로 처리
3. 최종 표현식(예: 0)을 만나면 누적된 환경을 반환

중요: 최종 표현식은 버려진다. 목표는 바인딩만 수집하는 것이다.

loadPrelude: 파일 로딩

loadPrelude는 Prelude.fun을 읽고 파싱하여 초기 환경을 생성한다.

```
let loadPrelude () : Env =
    let preludePath = "Prelude.fun"
    if File.Exists preludePath then
        try
            let source = File.ReadAllText preludePath
            let ast = parse source
            evalToEnv emptyEnv ast
        with ex ->
            eprintfn "Warning: Failed to load Prelude.fun: %s" ex.Message
            emptyEnv
    else
        eprintfn "Warning: Prelude.fun not found, starting with empty environment"
        emptyEnv
```

Graceful Degradation: 파일이 없거나 파싱 실패 시 빈 환경을 반환한다. 인터프리터는 Prelude 없이도 동작한다.

REPL 통합

Repl.fs의 startRepl 함수는 Prelude를 로드하여 초기 환경으로 사용한다.

```
let startRepl () : int =
    printfn "FunLang REPL"
    printfn "Type '#quit' or Ctrl+D to quit."
    printfn ""
    let initialEnv = Prelude.loadPrelude()
    replLoop initialEnv
    0
```

REPL 시작 시 Prelude의 모든 함수가 사용 가능하다.

CLI 통합

Program.fs도 --expr과 파일 실행 모드에서 Prelude를 로드한다.

```
// Load prelude for evaluation modes
let initialEnv = Prelude.loadPrelude()

// --expr only
elif results.Contains Expr then
    let expr = results.GetResult Expr
    try
        let result = eval initialEnv (parse expr)
        printfn "%s" (formatValue result)
        0
    with ex ->
        eprintfn "Error: %s" ex.Message
        1
```

CLI에서도 Prelude 함수를 바로 사용할 수 있다.

Examples

map: 리스트 변환

```
$ dotnet run --project FunLang -- -e 'map (fun x -> x * 2) [1, 2, 3]'
[2, 4, 6]

$ dotnet run --project FunLang -- -e 'let double = fun x -> x * 2 in map double [1, 2, 3, 4, 5]'
[2, 4, 6, 8, 10]
```

map은 리스트의 각 요소에 함수를 적용한다.

filter: 조건 필터링

```
$ dotnet run --project FunLang -- -e 'filter (fun x -> x > 1) [1, 2, 3]'
[2, 3]

$ dotnet run --project FunLang -- -e 'filter (fun x -> x > 0) [1, -2, 3, -4, 5]'
[1, 3, 5]
```

filter는 조건을 만족하는 요소만 남긴다.

fold: 리스트 누적

```
$ dotnet run --project FunLang -- -e 'fold (fun a -> fun b -> a + b) 0 [1, 2, 3]'
6

$ dotnet run --project FunLang -- -e 'fold (fun a -> fun b -> a * b) 1 [1, 2, 3, 4]'
24
```

fold는 리스트를 순회하며 값을 누적한다. 첫 번째 예시는 합계(sum), 두 번째는 곱(product).

length: 리스트 길이

```
$ dotnet run --project FunLang -- -e 'length [1, 2, 3]'
3

$ dotnet run --project FunLang -- -e 'length []'
0
```

reverse: 리스트 뒤집기

```
$ dotnet run --project FunLang -- -e 'reverse [1, 2, 3]'
[3, 2, 1]

$ dotnet run --project FunLang -- -e 'reverse [[1, 2], [3, 4]]'
[[3, 4], [1, 2]]
```

append: 리스트 연결

```
$ dotnet run --project FunLang -- -e 'append [1, 2] [3, 4]'
[1, 2, 3, 4]

$ dotnet run --project FunLang -- -e 'append [] [1, 2, 3]'
[1, 2, 3]
```

hd와 tl: 리스트 분해

```
$ dotnet run --project FunLang -- -e 'hd [1, 2, 3]'
1

$ dotnet run --project FunLang -- -e 'tl [1, 2, 3]'
[2, 3]
```

빈 리스트에 대해서는 매칭 실패 에러가 발생한다:

```
$ dotnet run --project FunLang -- -e 'hd []'
Error: Match failure: no pattern matched
```

Combinators

```
$ dotnet run --project FunLang -- -e 'id 42'
42

$ dotnet run --project FunLang -- -e 'const 1 2'
1

$ dotnet run --project FunLang -- -e 'let f = fun x -> x * 2 in let g = fun x -> x + 1 in (compose f g) 5'
12
```

`compose f g`는 `f (g x)`를 의미한다. `(compose f g) 5` = `f (g 5)` = `f 6` = 12.

함수 조합

고차 함수를 조합하여 복잡한 연산을 표현할 수 있다:

```
$ dotnet run --project FunLang -- -e 'map (fun x -> x * 2) (filter (fun x -> x > 1) [1, 2, 3, 4])'
[4, 6, 8]

$ dotnet run --project FunLang -- -e 'fold (fun a -> fun b -> a + b) 0 (map (fun x -> x * x) [1, 2, 3, 4])'
30
```

두 번째 예시는 `[1, 2, 3, 4]`를 제곱하고(`[1, 4, 9, 16]`) 합계를 계산한다(`1 + 4 + 9 + 16 = 30`).

REPL에서 사용

REPL을 시작하면 Prelude가 자동으로 로드되어 모든 함수를 사용할 수 있다:

```
$ dotnet run --project FunLang
FunLang REPL
Type '#quit' or Ctrl+D to quit.

funlang> map (fun x -> x * 2) [1, 2, 3]
[2, 4, 6]
funlang> length [1, 2, 3, 4, 5]
5
funlang> reverse [1, 2, 3]
[3, 2, 1]
```

구현 세부 사항

evalToEnv의 필요성

일반적인 eval은 표현식의 **값**을 반환한다. 그러나 Prelude는 **바인딩 집합**을 추출해야 한다.

evalToEnv는 다음과 같이 동작한다:

```
let map = ... in                → env' = env + {map = <func>}
let filter = ... in             → env'' = env' + {filter = <func>}
...
0                               → env''를 반환 (0은 버림)
```

각 let을 만날 때마다 환경을 확장하고, 최종적으로 모든 바인딩을 포함한 환경을 반환한다.

클로저와 재귀

let rec 함수는 클로저를 생성할 때 **자기 자신을 참조할 수 있어야** 한다.

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

FunctionValue의 클로저 환경은 현재 환경(env)이다. 그러나 extendedEnv에 함수 자신을 추가하므로, 함수 본문 평가 시 재귀 호출이 가능하다.

파일 경로

loadPrelude는 현재 작업 디렉토리에서 Prelude.fun을 찾는다. 인터프리터는 프로젝트 루트에서 실행되므로 파일이 올바르게 로드된다.

에러 처리

Prelude 로딩 실패는 치명적이지 않다. 경고만 출력하고 빈 환경을 반환한다. 이는 다음과 같은 상황에 유용하다:

- 개발 중 Prelude.fun이 아직 없을 때
- Prelude에 문법 에러가 있을 때
- 테스트 환경에서 Prelude 없이 실행할 때

정리

이 장에서 구현한 내용:

기능	파일	설명
표준 라이브러리 소스	`Prelude.fun`	FunLang으로 작성된 11개 함수
환경 추출	`Prelude.fs::evalToEnv`	중첩 let 바인딩을 환경으로 변환
자동 로딩	`Prelude.fs::loadPrelude`	시작 시 Prelude.fun 파싱 및 로드
REPL 통합	`Repl.fs::startRepl`	Prelude 환경으로 REPL 시작
CLI 통합	`Program.fs::main`	평가 모드에서 Prelude 사용

Self-Hosted Standard Library의 의미:

Prelude를 FunLang 자체로 작성함으로써, 언어가 자기 자신의 표준 라이브러리를 표현할 수 있을 만큼 강력하다는 것을 증명했다. 이는 Turing-complete 언어의 핵심 특징이다.

제공되는 함수들:

카테고리	함수	설명
고차 함수	`map`, `filter`, `fold`	리스트 변환, 필터링, 누적
리스트 유틸리티	`length`, `reverse`, `append`	길이, 뒤집기, 연결
리스트 접근	`hd`, `tl`	Head, tail
Combinators	`id`, `const`, `compose`	항등, 상수, 합성

테스트

```
# fslit 통합 테스트
make -C tests

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

Prelude 함수는 `tests/prelude/` 디렉토리의 fslit 테스트로 검증된다.

소스 참조

전체 소스 코드는 다음 위치에서 확인할 수 있다:

- **Prelude.fun**: 표준 라이브러리 소스 (프로젝트 루트)
- **FunLang/Prelude.fs**: 로딩 인프라 (`evalToEnv`, `loadPrelude`)
- **FunLang/Repl.fs**: REPL 통합 (`startRepl`)
- **FunLang/Program.fs**: CLI 통합 (main 함수)

관련 문서

- write-fsharp-repl-loop - REPL 구현 패턴
- setup-argu-cli - Argu CLI 설정
- testing-strategies - 테스트 전략 (fslit, Expecto)