Chapter 7: Lists

이 장에서는 FunLang에 리스트를 추가한다. 리스트는 가변 길이의 동종(homogeneous) 컬렉션으로, 함수형 프로그래밍의 핵심 자료구조다.

개요

리스트는 같은 타입의 값들을 순서대로 저장하는 컬렉션이다:

- **빈 리스트**: `[]`
- **리스트 리터럴**: `[1, 2, 3]`
- **Cons 연산자**: `h :: t` (요소를 리스트 앞에 추가)

리스트는 재귀적 구조를 가진다. 리스트는 빈 리스트이거나, head 요소와 tail 리스트의 조합이다.

```
[1, 2, 3]  =  1 :: [2, 3]
           =  1 :: 2 :: [3]
           =  1 :: 2 :: 3 :: []
```

AST 확장

Expr 타입에 리스트 표현식 추가

```
// FunLang/Ast.fs

type Expr =
    // ... 기존 케이스들 ...

    // Phase 2 (v3.0): Lists
    | EmptyList                       // 빈 리스트: []
    | List of Expr list               // 리스트 리터럴: [e1, e2, ...]
    | Cons of Expr * Expr             // Cons 연산자: h :: t
```

케이스	구문	설명
EmptyList	[]	빈 리스트
List	[e1, e2, ...]	리스트 리터럴
Cons	h :: t	head를 tail 앞에 추가

Value 타입에 ListValue 추가

```
/// Value type for evaluation results
and Value =
    | IntValue of int
    | BoolValue of bool
    | FunctionValue of param: string * body: Expr * closure: Env
    | StringValue of string
    | TupleValue of Value list
    | ListValue of Value list  // 리스트 값
```

`ListValue`는 `Value list`를 감싸는 래퍼다. F#의 네이티브 리스트를 사용하여 구현한다.

Lexer 확장

Lexer.fsl에 새 토큰을 추가한다.

```
// FunLang/Lexer.fsl

rule tokenize = parse
    // ... 기존 규칙들 ...

    // Phase 2 (v3.0): Cons 연산자 (다중 문자, 단일 문자보다 먼저)
    | "::"          { CONS }

    // Phase 2 (v3.0): 리스트 괄호
    | '['           { LBRACKET }
    | ']'           { RBRACKET }
```

중요: :: 규칙은 단일 문자 : 규칙이 있다면 그보다 먼저 와야 한다. 현재 FunLang에는 : 토큰이 없으므로 순서는 중요하지 않지만, 다중 문자 연산자를 먼저 배치하는 것이 좋은 습관이다.

토큰	패턴	용도
LBRACKET	[	리스트 시작
RBRACKET	]	리스트 끝
CONS	::	Cons 연산자

Parser 확장

토큰 선언

```
// FunLang/Parser.fsy

// Phase 2 (v3.0): List tokens
%token LBRACKET RBRACKET CONS
```

우선순위 선언

Cons 연산자는 **우결합(right-associative)** 이다.

```
// 우선순위 선언 (낮은 것부터 높은 것 순)
%left OR
%left AND
%nonassoc EQUALS LT GT LE GE NE
%right CONS    // 우결합: 1 :: 2 :: [] = 1 :: (2 :: [])
```

왜 우결합인가?

`1 :: 2 :: 3 :: []`는 다음과 같이 파싱되어야 한다:

`1 :: (2 :: (3 :: []))`

좌결합이라면 `((1 :: 2) :: 3) :: []`가 되어 타입 에러가 발생한다 (정수에 정수를 cons할 수 없음).

Cons의 우선순위 위치:

Cons는 비교 연산자보다 낮고, 산술 연산자보다 높다:

`1 + 2 :: [3]  →  (1 + 2) :: [3]  →  [3, 3]`

문법 규칙

```
Expr:
    // ... 기존 규칙들 ...

    // Phase 2 (v3.0): Cons 연산자 (비교보다 낮은 우선순위)
    | Expr CONS Expr                { Cons($1, $3) }

Atom:
    // ... 기존 규칙들 ...

    // Phase 2 (v3.0): 리스트 리터럴
    | LBRACKET RBRACKET                      { EmptyList }
    | LBRACKET Expr RBRACKET                 { List([$2]) }
    | LBRACKET Expr COMMA ExprList RBRACKET  { List($2 :: $4) }
```

리스트 리터럴은 `ExprList` 비단말을 재사용한다 (튜플에서 이미 정의됨):

```
// 콤마로 구분된 표현식 목록
ExprList:
    | Expr                      { [$1] }
    | Expr COMMA ExprList       { $1 :: $3 }
```

리스트 파싱 규칙:

입력	매칭 규칙	결과
`[]`	`LBRACKET RBRACKET`	`EmptyList`
`[1]`	`LBRACKET Expr RBRACKET`	`List([Number 1])`
`[1, 2, 3]`	`LBRACKET Expr COMMA ExprList RBRACKET`	`List([Number 1; Number 2; Number 3])`

Evaluator 구현

EmptyList: 빈 리스트 평가

```
// FunLang/Eval.fs

| EmptyList ->
    ListValue []
```

빈 리스트는 빈 `ListValue`로 평가된다.

List: 리스트 리터럴 평가

```
| List exprs ->
    let values = List.map (eval env) exprs
    ListValue values
```

각 요소를 순서대로 평가하고 ListValue로 감싼다.

Cons: Cons 연산자 평가

```
| Cons (headExpr, tailExpr) ->
    let headVal = eval env headExpr
    match eval env tailExpr with
    | ListValue tailVals -> ListValue (headVal :: tailVals)
    | _ -> failwith "Type error: cons (::) requires list as second argument"
```

핵심 포인트:

1. head는 어떤 값이든 가능
2. tail은 반드시 ListValue여야 함
3. F#의 :: 연산자로 실제 prepend 수행

구조적 동등성 (Structural Equality)

리스트는 구조적 동등성을 지원한다:

```
| Equal (left, right) ->
    match eval env left, eval env right with
    // ... 기존 케이스들 ...
    | ListValue l, ListValue r -> BoolValue (l = r)
    | _ -> failwith "Type error: = requires operands of same type"

| NotEqual (left, right) ->
    match eval env left, eval env right with
    // ... 기존 케이스들 ...
    | ListValue l, ListValue r -> BoolValue (l <> r)
    | _ -> failwith "Type error: <> requires operands of same type"
```

F#의 Value 타입이 구조적 동등성을 자동으로 지원하므로, l = r로 중첩된 리스트도 올바르게 비교된다.

formatValue: 리스트 출력

```
let rec formatValue (v: Value) : string =
    match v with
    // ... 기존 케이스들 ...
    | ListValue values ->
        let formattedElements = List.map formatValue values
        sprintf "[%s]" (String.concat ", " formattedElements)
```

출력 형식: [1, 2, 3]

중첩 리스트의 경우 재귀적으로 포맷팅된다: [[1, 2], [3, 4]]

연산자 우선순위 정리

FunLang의 전체 연산자 우선순위 (낮은 것부터 높은 것 순):

우선순위	연산자	결합성	설명
1 (낮음)	||	좌결합	논리 OR
2	&&	좌결합	논리 AND
3	=, <>, <, >, <=, >=	비결합	비교
4	::	**우결합**	Cons
5	+, -	좌결합	덧셈, 뺄셈
6	*, /	좌결합	곱셈, 나눗셈
7	단항 -	-	부정
8 (높음)	함수 호출	좌결합	f x

Examples

빈 리스트

```
$ dotnet run --project FunLang -- --expr "[]"
[]
```

리스트 리터럴

```
$ dotnet run --project FunLang -- --expr "[1, 2, 3]"
[1, 2, 3]

$ dotnet run --project FunLang -- --expr "[true, false]"
[true, false]
```

Cons 연산자

```
$ dotnet run --project FunLang -- --expr "1 :: [2, 3]"
[1, 2, 3]

$ dotnet run --project FunLang -- --expr "1 :: 2 :: 3 :: []"
[1, 2, 3]
```

우결합이므로 `1 :: 2 :: 3 :: []`는 `1 :: (2 :: (3 :: []))`로 파싱된다.

중첩 리스트

```
$ dotnet run --project FunLang -- --expr "[[1, 2], [3, 4]]"
[[1, 2], [3, 4]]
```

리스트 동등성

```
$ dotnet run --project FunLang -- --expr "[1, 2] = [1, 2]"
true

$ dotnet run --project FunLang -- --expr "[1, 2] <> [1, 2, 3]"
```

```
true

$ dotnet run --project FunLang -- --expr "[1] <> []"
true
```

튜플과 조합

```
$ dotnet run --project FunLang -- --expr "[(1, 2), (3, 4)]"
[(1, 2), (3, 4)]
```

산술과 Cons 우선순위

```
$ dotnet run --project FunLang -- --expr "1 + 2 :: [3]"
[3, 3]
```

`1 + 2`가 먼저 계산되어 `3 :: [3]` = `[3, 3]`이 된다.

조건문과 조합

```
$ dotnet run --project FunLang -- --expr "if true then [1, 2] else []"
[1, 2]

$ dotnet run --project FunLang -- --expr "[1] = [1] && true"
true
```

Let 바인딩과 조합

```
$ dotnet run --project FunLang -- --expr "let xs = [1, 2, 3] in xs"
[1, 2, 3]

$ dotnet run --project FunLang -- --expr "let x = 1 in x :: [2, 3]"
[1, 2, 3]
```

함수와 조합

```
$ dotnet run --project FunLang -- --expr "let f = fun x -> x :: [] in f 42"
[42]
```

AST 확인

```
$ dotnet run --project FunLang -- --emit-ast --expr "[]"
EmptyList

$ dotnet run --project FunLang -- --emit-ast --expr "[1, 2, 3]"
```

```
List [Number 1; Number 2; Number 3]

$ dotnet run --project FunLang -- --emit-ast --expr "1 :: [2, 3]"
Cons (Number 1, List [Number 2; Number 3])

$ dotnet run --project FunLang -- --emit-ast --expr "1 :: 2 :: []"
Cons (Number 1, Cons (Number 2, EmptyList))
```

마지막 예시에서 우결합이 명확히 드러난다: `Cons (Number 1, Cons (Number 2, EmptyList))`.

토큰 확인

```
$ dotnet run --project FunLang -- --emit-tokens --expr "[1, 2, 3]"
LBRACKET NUMBER(1) COMMA NUMBER(2) COMMA NUMBER(3) RBRACKET EOF

$ dotnet run --project FunLang -- --emit-tokens --expr "1 :: [2, 3]"
NUMBER(1) CONS LBRACKET NUMBER(2) COMMA NUMBER(3) RBRACKET EOF
```

타입 에러

Cons의 두 번째 인자가 리스트가 아닐 때

```
$ dotnet run --project FunLang -- --expr "1 :: 2"
Error: Type error: cons (::) requires list as second argument
```

다른 타입의 리스트 비교

```
$ dotnet run --project FunLang -- --expr "[1, 2] = true"
Error: Type error: = requires operands of same type
```

정리

이 장에서 구현한 내용:

기능	구문	예시
빈 리스트	`[]`	`[]`
리스트 리터럴	`[e1, e2, ...]`	`[1, 2, 3]`
Cons 연산자	`h :: t`	`1 :: [2, 3]`
리스트 동등성	`=`, `<>`	`[1, 2] = [1, 2]`
중첩 리스트	-	`[[1], [2, 3]]`

파일별 변경 사항:

파일	변경 사항
`Ast.fs`	`EmptyList`, `List`, `Cons` 케이스, `ListValue`
`Lexer.fsl`	`LBRACKET`, `RBRACKET`, `CONS` 토큰

파일	변경 사항
`Parser.fsy`	`%right CONS` 선언, 리스트 리터럴 규칙
`Eval.fs`	리스트 평가, Cons 평가, 구조적 동등성, `formatValue`

테스트

```
# fslit 테스트
make -C tests

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

관련 문서

- fsyacc-operator-precedence-methods - 연산자 우선순위 처리 방법 (`%right`로 우결합 선언)
- fsyacc-precedence-without-declarations - 문법 계층으로 우선순위 인코딩
- write-fsyacc-parser - fsyacc 기본 문법
- adapt-tests-for-value-type-evolution - Value 타입 확장 시 테스트 적응