

Chapter 4: Conditionals and Boolean

Chapter 3의 변수 바인딩에 조건문과 Boolean 타입을 추가한다. 이제 FunLang은 여러 타입을 다루는 언어가 된다.

개요

이 chapter에서 추가하는 기능:

- Boolean 리터럴 (true, false)
- If-then-else 표현식 (if condition then expr1 else expr2)
- 비교 연산자 (=, <, <=, >, >=, >=)
- 논리 연산자 (&&, ||)
- 타입 검사 (런타임)
- 단락 평가 (Short-circuit evaluation)

Value 타입 도입

이전까지 evalExpr는 int를 반환했다. Boolean을 추가하면 결과가 정수일 수도, 불린일 수도 있다. 이를 위해 Discriminated Union을 사용한다.

Ast.fs에 Value 타입을 추가한다.

```
module Ast

/// 평가 결과 타입
/// Phase 4: 이중 타입 (int와 bool)
type Value =
    | IntValue of int
    | BoolValue of bool
```

왜 Discriminated Union인가?

방식	문제점
obj	타입 안정성 없음, 캐스팅 필요
제네릭	표현식마다 타입이 다름
DU	가능한 타입을 명시적으로 열거, 패턴 매칭으로 안전하게 처리

AST 확장

Ast.fs에 제어 흐름 노드를 추가한다.

```
type Expr =
    | Number of int
    | Add of Expr * Expr
    | Subtract of Expr * Expr
    | Multiply of Expr * Expr
    | Divide of Expr * Expr
    | Negate of Expr
    // Phase 3: Variables
    | Var of string
    | Let of string * Expr * Expr
    // Phase 4: Control flow
    | Bool of bool // Boolean 리터럴
```

```

| If of Expr * Expr * Expr // if condition then expr1 else expr2
// Phase 4: 비교 연산자 (BoolValue 반환)
| Equal of Expr * Expr // =
| NotEqual of Expr * Expr // <>
| LessThan of Expr * Expr // <
| GreaterThan of Expr * Expr // >
| LessEqual of Expr * Expr // <=
| GreaterEqual of Expr * Expr // >=
// Phase 4: 논리 연산자 (단락 평가)
| And of Expr * Expr // &&
| Or of Expr * Expr // ||

```

AST 예시:

if 5 > 3 then 1 else 2

If(GreaterThan(Number 5, Number 3), Number 1, Number 2)

Lexer 확장

Lexer.fsl에 새 키워드와 연산자를 추가한다.

```

{
open System
open FSharp.Text.Lexing
open Parser

let lexeme (lexbuf: LexBuffer<_>) =
    LexBuffer<_>.LexemeString lexbuf
}

let digit = ['0'-'9']
let whitespace = [' ' '\t']
let newline = ('\n' | '\r' '\n')
let letter = ['a'-'z' 'A'-'Z']
let ident_start = letter | '_'
let ident_char = letter | digit | '_'

rule tokenize = parse
| whitespace+ { tokenize lexbuf }
| newline { tokenize lexbuf }
| digit+ { NUMBER (Int32.Parse(lexeme lexbuf)) }
// 키워드는 반드시 식별자보다 먼저!
| "true" { TRUE }
| "false" { FALSE }
| "if" { IF }
| "then" { THEN }
| "else" { ELSE }
| "let" { LET }
| "in" { IN }
// 식별자
| ident_start ident_char* { IDENT (lexeme lexbuf) }
// 다중 문자 연산자는 단일 문자보다 먼저!
| "<=" { LE }

```

```

| ">="      { GE }
| "<>"      { NE }
| "&&"      { AND }
| "||"      { OR }
// 단일 문자 연산자
| '+'       { PLUS }
| '-'       { MINUS }
| '*'       { STAR }
| '/'       { SLASH }
| '<'       { LT }
| '>'       { GT }
| '('       { LPAREN }
| ')'       { RPAREN }
| '='       { EQUALS }
| eof       { EOF }

```

핵심 포인트: 연산자 매칭 순서

```

// 올바른 순서
| "<="      { LE }
| "<"       { LT }

// 잘못된 순서
| "<"       { LT } // "<=" 입력 시 "<"만 매칭됨
| "<="      { LE } // 도달 불가

```

fslex는 가장 긴 매칭(longest match)을 선호하지만, 같은 길이면 먼저 나온 규칙이 우선이다. 안전을 위해 다중 문자 연산자를 먼저 배치한다.

Parser 확장

Parser.fsy에 새 토큰과 문법을 추가한다.

```

%{
open Ast
%}

%token <int> NUMBER
%token <string> IDENT
%token PLUS MINUS STAR SLASH
%token LPAREN RPAREN
%token LET IN EQUALS
%token TRUE FALSE IF THEN ELSE
%token LT GT LE GE NE
%token AND OR
%token EOF

// 우선순위 선언 (낮은 것부터 높은 것 순)
%left OR
%left AND
%nonassoc EQUALS LT GT LE GE NE

%start start

```

```

%type <Ast.Expr> start

%%

start:
| Expr EOF          { $1 }

Expr:
// Let - 가장 낮은 우선순위
| LET IDENT EQUALS Expr IN Expr { Let($2, $4, $6) }
// If-then-else - 낮은 우선순위
| IF Expr THEN Expr ELSE Expr { If($2, $4, $6) }
// 논리 연산자 (%left 선언으로 우선순위 적용)
| Expr OR Expr          { Or($1, $3) }
| Expr AND Expr         { And($1, $3) }
// 비교 연산자 (%nonassoc로 비연관성)
| Expr EQUALS Expr      { Equal($1, $3) }
| Expr LT Expr          { LessThan($1, $3) }
| Expr GT Expr          { GreaterThan($1, $3) }
| Expr LE Expr          { LessEqual($1, $3) }
| Expr GE Expr          { GreaterEqual($1, $3) }
| Expr NE Expr          { NotEqual($1, $3) }
// 산술 연산 (Term/Factor 패턴 유지)
| Expr PLUS Term        { Add($1, $3) }
| Expr MINUS Term       { Subtract($1, $3) }
| Term                  { $1 }

Term:
| Term STAR Factor      { Multiply($1, $3) }
| Term SLASH Factor     { Divide($1, $3) }
| Factor                { $1 }

Factor:
| NUMBER                { Number($1) }
| IDENT                 { Var($1) }
| TRUE                  { Bool(true) }
| FALSE                 { Bool(false) }
| LPAREN Expr RPAREN   { $2 }
| MINUS Factor          { Negate($2) }

```

연산자 우선순위:

낮음 → 높음:

OR < AND < 비교 < +, - < *, / < 단항- < 괄호/리터럴

예: a || b && c < 5 + 1

→ a || (b && ((c < 5) + 1)) -- 잘못!

→ a || (b && (c < (5 + 1))) -- 올바른

%nonassoc 비교 연산자:

1 < 2 < 3 같은 표현을 파싱 에러로 만든다. 왜? - 수학적으로 1 < 2 < 3은 “1 < 2이고 2 < 3”을 의미 - 그러나 대부분 언어에서 (1 < 2) < 3으로 파싱됨 → true < 3 → 타입 에러 - 혼란 방지를 위해 비연관성으로 선언

Evaluator: 타입 검사

Eval.fs를 Value 반환으로 수정한다.

```
module Eval

open Ast

/// 환경: 변수 이름 → Value 매핑
type Env = Map<string, Value>

let emptyEnv : Env = Map.empty

/// 출력용 포매팅
let formatValue (v: Value) : string =
    match v with
    | IntValue n -> string n
    | BoolValue b -> if b then "true" else "false"

/// 환경 내에서 표현식 평가
/// 타입 에러 시 예외 발생
let rec eval (env: Env) (expr: Expr) : Value =
    match expr with
    | Number n -> IntValue n
    | Bool b -> BoolValue b

    | Var name ->
        match Map.tryFind name env with
        | Some value -> value
        | None -> failwithf "Undefined variable: %s" name

    | Let (name, binding, body) ->
        let value = eval env binding
        let extendedEnv = Map.add name value env
        eval extendedEnv body

    // 산술 연산 - 정수 타입 검사
    | Add (left, right) ->
        match eval env left, eval env right with
        | IntValue l, IntValue r -> IntValue (l + r)
        | _ -> failwith "Type error: + requires integer operands"

    | Subtract (left, right) ->
        match eval env left, eval env right with
        | IntValue l, IntValue r -> IntValue (l - r)
        | _ -> failwith "Type error: - requires integer operands"

    | Multiply (left, right) ->
        match eval env left, eval env right with
        | IntValue l, IntValue r -> IntValue (l * r)
        | _ -> failwith "Type error: * requires integer operands"

    | Divide (left, right) ->
        match eval env left, eval env right with
```

```

    | IntValue l, IntValue r -> IntValue (l / r)
    | _ -> failwith "Type error: / requires integer operands"

| Negate e ->
  match eval env e with
  | IntValue n -> IntValue (-n)
  | _ -> failwith "Type error: unary - requires integer operand"

// 비교 연산 - 정수 입력, 불린 출력
| LessThan (left, right) ->
  match eval env left, eval env right with
  | IntValue l, IntValue r -> BoolValue (l < r)
  | _ -> failwith "Type error: < requires integer operands"

| GreaterThan (left, right) ->
  match eval env left, eval env right with
  | IntValue l, IntValue r -> BoolValue (l > r)
  | _ -> failwith "Type error: > requires integer operands"

| LessEqual (left, right) ->
  match eval env left, eval env right with
  | IntValue l, IntValue r -> BoolValue (l <= r)
  | _ -> failwith "Type error: <= requires integer operands"

| GreaterEqual (left, right) ->
  match eval env left, eval env right with
  | IntValue l, IntValue r -> BoolValue (l >= r)
  | _ -> failwith "Type error: >= requires integer operands"

// 동등 비교 - 같은 타입 요구
| Equal (left, right) ->
  match eval env left, eval env right with
  | IntValue l, IntValue r -> BoolValue (l = r)
  | BoolValue l, BoolValue r -> BoolValue (l = r)
  | _ -> failwith "Type error: = requires operands of same type"

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

// 논리 연산 - 단락 평가
| And (left, right) ->
  match eval env left with
  | BoolValue false -> BoolValue false // 단락: 오른쪽 평가 안 함
  | BoolValue true ->
    match eval env right with
    | BoolValue b -> BoolValue b
    | _ -> failwith "Type error: && requires boolean operands"
  | _ -> failwith "Type error: && requires boolean operands"

| Or (left, right) ->

```

```

match eval env left with
| BoolValue true -> BoolValue true // 단락: 오른쪽 평가 안 함
| BoolValue false ->
    match eval env right with
    | BoolValue b -> BoolValue b
    | _ -> failwith "Type error: || requires boolean operands"
| _ -> failwith "Type error: || requires boolean operands"

// If-then-else
| If (condition, thenBranch, elseBranch) ->
    match eval env condition with
    | BoolValue true -> eval env thenBranch
    | BoolValue false -> eval env elseBranch
    | _ -> failwith "Type error: if condition must be boolean"

let evalExpr (expr: Expr) : Value =
    eval emptyEnv expr

```

단락 평가 (Short-circuit Evaluation):

false && (실행 안 됨) → false
 true || (실행 안 됨) → true

단락 평가는 부작용(side effect)이 있을 때 중요하다. 예: $x < 0 \ \&\& \ 10 / x > 1$

타입 검사:

연산자	입력 타입	출력 타입
+, -, *, /	int, int	int
<, >, <=, >=	int, int	bool
=, <>	같은 타입	bool
&&, \ \	bool, bool	bool
if	bool, any, any	any

Program.fs 수정

Value를 문자열로 출력하도록 변경한다.

```

[<EntryPoint>]
let main argv =
    match argv with
    | [| "--expr"; expr |] ->
        try
            let result = expr |> parse |> evalExpr
            printfn "%s" (formatValue result) // int → Value 포매팅
            0
        with ex ->
            eprintfn "Error: %s" ex.Message
            1
    // ... 나머지 옵션

```

Examples

Boolean 리터럴

```
$ dotnet run --project FunLang -- --expr "true"
true

$ dotnet run --project FunLang -- --expr "false"
false

$ dotnet run --project FunLang -- --expr "let b = true in b"
true
```

If-Then-Else

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

$ dotnet run --project FunLang -- --expr "if false then 1 else 2"
2

$ dotnet run --project FunLang -- --expr "if 5 > 3 then 10 else 20"
10

$ dotnet run --project FunLang -- --expr "if 2 + 3 > 4 then 10 else 20"
10
```

중첩 If

```
$ dotnet run --project FunLang -- --expr "if 5 > 3 then if 2 < 4 then 100 else 50 else 0"
100
```

비교 연산자

```
$ dotnet run --project FunLang -- --expr "if 3 < 5 then 1 else 0"
1

$ dotnet run --project FunLang -- --expr "if 5 > 3 then 1 else 0"
1

$ dotnet run --project FunLang -- --expr "if 3 <= 3 then 1 else 0"
1

$ dotnet run --project FunLang -- --expr "if 5 >= 5 then 1 else 0"
1

$ dotnet run --project FunLang -- --expr "if 5 = 5 then 1 else 0"
1
```



```
$ dotnet run --project FunLang -- --expr "if 5 <> 3 then 1 else 0"
1
```

논리 연산자

```
$ dotnet run --project FunLang -- --expr "if true && true then 1 else 0"
1
$ dotnet run --project FunLang -- --expr "if true && false then 1 else 0"
0
$ dotnet run --project FunLang -- --expr "if false || true then 1 else 0"
1
$ dotnet run --project FunLang -- --expr "if false || false then 1 else 0"
0
```

Let과 조합

```
$ dotnet run --project FunLang -- --expr "let x = 10 in if x > 5 then x else 0"
10
$ dotnet run --project FunLang -- --expr "let x = 10 in let y = 20 in if x = 10 && y = 20 then 1 else 0"
1
```

타입 에러

```
$ dotnet run --project FunLang -- --expr "if 1 then 2 else 3"
Error: Type error: if condition must be boolean
$ dotnet run --project FunLang -- --expr "true + 1"
Error: Type error: + requires integer operands
$ dotnet run --project FunLang -- --expr "1 && 2"
Error: Type error: && requires boolean operands
$ dotnet run --project FunLang -- --expr "true < false"
Error: Type error: < requires integer operands
```

디버깅: 토큰 및 AST

```
$ dotnet run --project FunLang -- --emit-tokens --expr "if true then 1 else 2"
IF TRUE THEN NUMBER(1) ELSE NUMBER(2) EOF
$ dotnet run --project FunLang -- --emit-ast --expr "if true then 1 else 2"
If (Bool true, Number 1, Number 2)
```

```
$ dotnet run --project FunLang -- --emit-tokens --expr "5 > 3 && 2 < 4"
NUMBER(5) GT NUMBER(3) AND NUMBER(2) LT NUMBER(4) EOF
```

테스트

fslit (CLI 테스트)

tests/control/ 디렉토리에 20개 테스트:

```
$ make -C tests control
...
PASS: all 20 tests passed
```

Expecto (단위 테스트)

```
$ dotnet run --project FunLang.Tests
[16:21:05 INF] EXPECTO? Running tests...
[16:21:05 INF] EXPECTO! 93 tests run - 93 passed, 0 failed. Success!
```

요약

파일	변경 사항
Ast.fs	Value 타입, Bool, If, 비교/논리 연산자 노드
Lexer.fsl	TRUE, FALSE, IF, THEN, ELSE, 비교/논리 토큰
Parser.fsy	우선순위 선언, if-then-else 문법, 비교/논리 규칙
Eval.fs	Value 반환, 타입 검사, 단락 평가
Program.fs	formatValue로 출력

핵심 개념

개념	설명
Value Type	평가 결과를 표현하는 discriminated union
Type Check	런타임에 피연산자 타입 검증
Short-circuit	&&, 에서 첫 피연산자로 결과 결정 시 두 번째 평가 생략
Precedence	연산자 우선순위로 파싱 모호성 해결

다음 Chapter

Chapter 5에서는 함수 정의와 호출을 추가한다.

관련 문서

- [adapt-tests-for-value-type-evolution](#) — evalExpr가 int→Value로 변경될 때 테스트 호환성 유지
- [write-fsyacc-parser](#) — fsyacc 우선순위 선언
- [fsyacc-operator-precedence-methods](#) — 연산자 우선순위 처리 방법
- [testing-strategies](#) — 테스트 전략 가이드