Chapter 8: Pattern Matching

이 장에서는 FunLang에 패턴 매칭을 추가한다. match 표현식을 사용하여 값의 구조를 검사하고 분해할 수 있다.

개요

패턴 매칭은 함수형 프로그래밍의 핵심 기능이다. 조건문보다 강력하며, 값을 검사하는 동시에 내부 구조를 분해하여 변수에 바인딩할 수 있다.

FunLang의 패턴 매칭:

- **Match 표현식**: `match e with | p1 -> e1 | p2 -> e2`
- **패턴 종류**: 상수, 변수, 와일드카드, cons, 빈 리스트, 튜플
- **First-match 의미**: 패턴을 위에서 아래로 순서대로 시도
- **비소진 매칭 에러**: 어떤 패턴도 매칭되지 않으면 런타임 에러

패턴 매칭을 사용하면 리스트 처리와 재귀 함수를 우아하게 작성할 수 있다.

AST 확장

Match 표현식과 패턴 타입

```
// FunLang/Ast.fs

type Expr =
    // ... 기존 케이스들 ...

    // Phase 3 (v3.0): Pattern Matching
    | Match of scrutinee: Expr * clauses: MatchClause list

/// Pattern for destructuring bindings
and Pattern =
    | VarPat of string           // 변수 패턴: x
    | TuplePat of Pattern list   // 튜플 패턴: (p1, p2, ...)
    | WildcardPat                // 와일드카드 패턴: _
    // Phase 3 (v3.0): New pattern types for match expressions
    | ConsPat of Pattern * Pattern     // Cons 패턴: h :: t
    | EmptyListPat                     // 빈 리스트 패턴: []
    | ConstPat of Constant             // 상수 패턴: 1, true, false

/// Match clause: pattern -> expression
and MatchClause = Pattern * Expr

/// Constant values for patterns
and Constant =
    | IntConst of int
    | BoolConst of bool
```

타입	설명
`Match`	검사할 값(scrutinee)과 매칭 절 리스트
`MatchClause`	패턴과 결과 표현식의 쌍
`Constant`	패턴에서 사용할 상수 값 (정수, 불린)

패턴 종류

패턴	구문	매칭 조건	바인딩
VarPat	x	모든 값	x에 값 바인딩
WildcardPat	_	모든 값	바인딩 없음
ConstPat	1, true	상수와 동일한 값	바인딩 없음
EmptyListPat	[]	빈 리스트	바인딩 없음
ConsPat	h :: t	비어있지 않은 리스트	h에 첫 요소, t에 나머지
TuplePat	(x, y)	같은 길이의 튜플	각 요소를 변수에 바인딩

Lexer 확장

Lexer.fsl에 패턴 매칭 토큰을 추가한다.

```
// FunLang/Lexer.fsl

// Phase 3 (v3.0): Pattern matching keywords
| "match"       { MATCH }
| "with"        { WITH }

// Phase 3 (v3.0): Pipe for match clauses
| '|'           { PIPE }
```

토큰	용도
MATCH	match 표현식 시작
WITH	scrutinee와 패턴 절 구분
PIPE	각 패턴 절 앞에 위치 (| 문자)

Parser 확장

토큰 선언

```
// FunLang/Parser.fsy

// Phase 3 (v3.0): Pattern matching tokens
%token MATCH WITH PIPE
```

Match 표현식 문법

Match 표현식은 가장 낮은 우선순위를 가진다 (let, if보다도 낮음).

```
Expr:
    // Phase 3 (v3.0): Match expression - lowest precedence
    | MATCH Expr WITH MatchClauses   { Match($2, $4) }
    // ... 기존 규칙들 ...
```

구문 구조:

```
match <scrutinee> with
| <pattern1> -> <expr1>
```

```
| <pattern2> -> <expr2>
...
```

Match 절 문법

```
// Phase 3 (v3.0): Match clauses (non-empty, requires leading PIPE)
MatchClauses:
    | PIPE Pattern ARROW Expr                { [($2, $4)] }
    | PIPE Pattern ARROW Expr MatchClauses   { ($2, $4) :: $5 }
```

핵심 포인트:

- 각 절은 반드시 |로 시작
- 첫 번째 절도 |로 시작 (일관성)
- 재귀적 정의로 여러 절 지원

패턴 문법

```
Pattern:
    | LPAREN PatternList RPAREN   { TuplePat($2) }
    | IDENT                       { VarPat($1) }
    | UNDERSCORE                  { WildcardPat }
    // Phase 3 (v3.0): Extended patterns for match expressions
    | NUMBER                      { ConstPat(IntConst($1)) }
    | TRUE                        { ConstPat(BoolConst(true)) }
    | FALSE                       { ConstPat(BoolConst(false)) }
    | LBRACKET RBRACKET           { EmptyListPat }
    | Pattern CONS Pattern        { ConsPat($1, $3) }
```

Cons 패턴의 우선순위:

Cons 패턴은 이미 선언된 %right CONS를 재사용한다:

```
// 우선순위 선언
%right CONS    // Right-associative
```

예시:

h :: t :: rest → ConsPat(h, ConsPat(t, rest))

Evaluator 구현

matchPattern: 패턴 매칭 함수

matchPattern은 패턴과 값을 비교하여 바인딩을 생성한다.

```
// FunLang/Eval.fs

/// Match a pattern against a value, returning bindings if successful
let rec matchPattern (pat: Pattern) (value: Value) : (string * Value) list option =
    match pat, value with
    | VarPat name, v -> Some [(name, v)]
    | WildcardPat, _ -> Some []
```

```
    | TuplePat pats, TupleValue vals ->
        if List.length pats <> List.length vals then
            None  // Arity mismatch
        else
            let bindings = List.map2 matchPattern pats vals
            if List.forall Option.isSome bindings then
                Some (List.collect Option.get bindings)
            else
                None
    // Constant patterns
    | ConstPat (IntConst n), IntValue m ->
        if n = m then Some [] else None
    | ConstPat (BoolConst b1), BoolValue b2 ->
        if b1 = b2 then Some [] else None
    // Empty list pattern
    | EmptyListPat, ListValue [] -> Some []
    // Cons pattern - matches non-empty list
    | ConsPat (headPat, tailPat), ListValue (h :: t) ->
        match matchPattern headPat h with
        | Some headBindings ->
            match matchPattern tailPat (ListValue t) with
            | Some tailBindings -> Some (headBindings @ tailBindings)
            | None -> None
        | None -> None
    | _ -> None  // Type mismatch
```

반환 값:

- Some [바인딩 리스트]: 매칭 성공
- None: 매칭 실패

매칭 규칙:

패턴	값	결과	바인딩
VarPat "x"	모든 값	성공	[("x", value)]
WildcardPat	모든 값	성공	[]
ConstPat (IntConst 1)	IntValue 1	성공	[]
ConstPat (IntConst 1)	IntValue 2	실패	-
EmptyListPat	ListValue []	성공	[]
ConsPat (VarPat "h", VarPat "t")	ListValue [1; 2; 3]	성공	[("h", IntValue 1); ("t", ListValue [2; 3])]
TuplePat [VarPat "x"; VarPat "y"]	TupleValue [IntValue 1; IntValue 2]	성공	[("x", IntValue 1); ("y", IntValue 2)]

evalMatchClauses: Match 절 평가

```
/// Evaluate match clauses sequentially, returning first match
and evalMatchClauses (env: Env) (scrutinee: Value) (clauses: MatchClause list) : Value =
    match clauses with
```

```
    | [] -> failwith "Match failure: no pattern matched"
    | (pattern, resultExpr) :: rest ->
        match matchPattern pattern scrutinee with
        | Some bindings ->
            let extendedEnv = List.fold (fun e (n, v) -> Map.add n v e) env bindings
            eval extendedEnv resultExpr
        | None ->
            evalMatchClauses env scrutinee rest
```

작동 방식:

1. 절 리스트를 순서대로 순회
2. 각 패턴을 matchPattern으로 시도
3. 매칭되면:
   - 바인딩을 환경에 추가
   - 결과 표현식을 확장된 환경에서 평가
4. 매칭 실패하면 다음 절 시도
5. 모든 절이 실패하면 런타임 에러

Match 표현식 평가

```
// Phase 3 (v3.0): Pattern Matching
| Match (scrutinee, clauses) ->
    let value = eval env scrutinee
    evalMatchClauses env value clauses
```

1. scrutinee를 평가하여 값 얻기
2. evalMatchClauses로 첫 번째 매칭되는 절 찾기
3. 해당 절의 결과 반환

패턴 매칭 활용 예시

상수 패턴

```
match x with
| 0 -> "zero"
| 1 -> "one"
| _ -> "other"
```

숫자를 문자열로 변환한다. 와일드카드 _는 모든 값을 매칭한다.

리스트 패턴: 재귀적 합계

```
let rec sum xs =
    match xs with
    | [] -> 0
    | h :: t -> h + sum t
```

작동 방식:

- []: 빈 리스트면 0 반환

- `h :: t`: 비어있지 않으면 첫 요소 + 나머지 합계

예시 평가:

```
sum [1, 2, 3]
→ match [1, 2, 3] with ...
→ 1 + sum [2, 3]
→ 1 + (2 + sum [3])
→ 1 + (2 + (3 + sum []))
→ 1 + (2 + (3 + 0))
→ 6
```

튜플 패턴

```
match pair with
| (x, y) -> x + y
```

튜플을 분해하여 각 요소를 변수에 바인딩한다.

중첩 패턴

```
match xs with
| (h1 :: h2 :: t) -> h1 + h2
| _ -> 0
```

최소 두 개의 요소가 있는 리스트에서 첫 두 요소를 추출한다.

Examples

상수 매칭

```
$ dotnet run --project FunLang -- -e 'match 1 with | 1 -> "one" | _ -> "other"'
"one"

$ dotnet run --project FunLang -- -e 'match 2 with | 1 -> "one" | _ -> "other"'
"other"

$ dotnet run --project FunLang -- -e 'match true with | true -> 1 | false -> 0'
1
```

리스트 패턴: Head와 Tail

```
$ dotnet run --project FunLang -- -e 'match [1, 2, 3] with | [] -> 0 | h :: t -> h'
1

$ dotnet run --project FunLang -- -e 'match [] with | [] -> 0 | h :: t -> h'
0

$ dotnet run --project FunLang -- -e 'match [1, 2, 3] with | h :: t -> t'
[2, 3]
```

첫 번째 예시는 리스트의 첫 요소(head)를 반환한다. 세 번째 예시는 나머지(tail)를 반환한다.

튜플 패턴

```
$ dotnet run --project FunLang -- -e 'match (1, 2) with | (x, y) -> x + y'
3

$ dotnet run --project FunLang -- -e 'match (5, 10) with | (a, b) -> a * b'
50
```

재귀 함수: 리스트 합계

```
$ dotnet run --project FunLang -- -e 'let rec sum xs = match xs with | [] -> 0 | h :: t -> h + sum t in sum
15
```

평가 과정:

```
sum [1, 2, 3, 4, 5]
→ 1 + sum [2, 3, 4, 5]
→ 1 + 2 + sum [3, 4, 5]
→ 1 + 2 + 3 + sum [4, 5]
→ 1 + 2 + 3 + 4 + sum [5]
→ 1 + 2 + 3 + 4 + 5 + sum []
→ 1 + 2 + 3 + 4 + 5 + 0
→ 15
```

재귀 함수: 리스트 길이

```
$ dotnet run --project FunLang -- -e 'let rec length xs = match xs with | [] -> 0 | h :: t -> 1 + length t i
3
```

재귀 함수: 리스트 뒤집기

```
$ dotnet run --project FunLang -- -e 'let rec rev xs = match xs with | [] -> [] | h :: t -> rev t + [h] in r
[3, 2, 1]
```

참고: 이 구현은 + 연산자가 리스트 연결을 지원한다고 가정한다. 실제 FunLang에서는 문자열 연결만 지원하므로, 더 복잡한 구현이 필요할 수 있다.

중첩 패턴

```
$ dotnet run --project FunLang -- -e 'match [1, 2, 3] with | h1 :: h2 :: t -> h1 + h2 | _ -> 0'
3

$ dotnet run --project FunLang -- -e 'match [1] with | h1 :: h2 :: t -> h1 + h2 | _ -> 0'
0
```

첫 번째 예시는 최소 두 요소가 있으므로 첫 두 요소를 더한다. 두 번째는 요소가 하나뿐이므로 와일드카드 패턴이 매칭된다.

불린 패턴

```
$ dotnet run --project FunLang -- -e 'match true && false with | true -> "yes" | false -> "no"'
"no"

$ dotnet run --project FunLang -- -e 'match 5 > 3 with | true -> "greater" | false -> "not greater"'
"greater"
```

복잡한 예시: 필터링

```
$ dotnet run --project FunLang -- -e 'let rec filter f xs = match xs with | [] -> [] | h :: t -> if f h then
[2, 4]
```

참고: 이 예시는 설명을 위한 것이며, 실제 FunLang 구현에서 모든 기능이 지원되는지 확인이 필요할 수 있다.

AST 확인

```
$ dotnet run --project FunLang -- --emit-ast -e 'match x with | 1 -> "one" | _ -> "other"'
Match (Var "x", [(ConstPat (IntConst 1), String "one"); (WildcardPat, String "other")])

$ dotnet run --project FunLang -- --emit-ast -e 'match xs with | [] -> 0 | h :: t -> h'
Match (Var "xs", [(EmptyListPat, Number 0); (ConsPat (VarPat "h", VarPat "t"), Var "h")])
```

런타임 에러

비소진 매칭 (Non-exhaustive Match)

모든 패턴이 실패하면 런타임 에러가 발생한다.

```
$ dotnet run --project FunLang -- -e 'match 2 with | 1 -> "one"'
Error: Match failure: no pattern matched
```

이 예시는 2를 매칭하지만, 패턴은 1만 처리한다.

해결책: 와일드카드 패턴으로 모든 경우를 처리한다.

```
$ dotnet run --project FunLang -- -e 'match 2 with | 1 -> "one" | _ -> "other"'
"other"
```

타입 불일치

패턴과 값의 타입이 맞지 않으면 매칭 실패한다.

```
$ dotnet run --project FunLang -- -e 'match 1 with | [] -> 0'
Error: Match failure: no pattern matched
```

정수 1은 빈 리스트 패턴 []와 매칭되지 않는다.

패턴 매칭 vs If-Then-Else

기능	If-Then-Else	Pattern Matching
구조 분해	불가능	가능 (`h :: t`, `(x, y)`)
여러 조건	중첩 필요	여러 절로 간결하게
가독성	복잡한 조건에서 낮음	높음
비소진 검사	없음	런타임 에러

If-Then-Else로 리스트 합계:

```
let rec sum xs =
    if xs = [] then
        0
    else
        // head와 tail 추출이 어려움!
        ???
```

리스트를 분해하는 연산이 없으면 if-then-else로 구현하기 어렵다.

Pattern Matching으로 리스트 합계:

```
let rec sum xs =
    match xs with
    | [] -> 0
    | h :: t -> h + sum t
```

패턴 매칭은 구조 분해를 내장하므로 간결하다.

정리

이 장에서 구현한 내용:

기능	구문	예시
Match 표현식	`match e with | p1 -> e1 | p2 -> e2`	`match x with | 1 -> "one" | _ -> "other"`
상수 패턴	`1`, `true`, `false`	`| 0 -> "zero"`
변수 패턴	`x`	`| n -> n + 1`
와일드카드 패턴	`_`	`| _ -> "default"`
빈 리스트 패턴	`[]`	`| [] -> 0`
Cons 패턴	`h :: t`	`| h :: t -> h + sum t`
튜플 패턴	`(x, y)`	`| (a, b) -> a + b`

파일별 변경 사항:

파일	변경 사항
`Ast.fs`	`Match` 케이스, `Pattern` 타입 확장 (`ConsPat`, `EmptyListPat`, `ConstPat`), `MatchClause`, `Constant`

파일	변경 사항
Lexer.fsl	MATCH, WITH, PIPE 토큰
Parser.fsy	Match 표현식 규칙, MatchClauses 비단말, Pattern 확장
Eval.fs	matchPattern 함수 확장, evalMatchClauses 헬퍼 함수

핵심 개념:

- **First-match 의미**: 패턴은 위에서 아래로 순서대로 시도된다.
- **구조 분해**: 패턴은 값을 검사하면서 동시에 내부 구조를 변수에 바인딩한다.
- **비소진 매칭**: 모든 패턴이 실패하면 런타임 에러가 발생한다.

패턴 매칭은 함수형 프로그래밍의 표현력을 크게 향상시키며, 특히 재귀적 자료구조(리스트)를 처리할 때 필수적이다.

테스트

```
# fslit 테스트
make -C tests

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

관련 문서

- write-fsyacc-parser - fsyacc 기본 문법
- fsyacc-operator-precedence-methods - 연산자 우선순위 처리 (%right CONS 재사용)

소스 참조

전체 소스 코드는 FunLang 디렉토리를 참조한다:

- FunLang/Ast.fs - Match, Pattern, MatchClause, Constant 타입
- FunLang/Lexer.fsl - MATCH, WITH, PIPE 토큰
- FunLang/Parser.fsy - Match 표현식 및 패턴 문법
- FunLang/Eval.fs - matchPattern, evalMatchClauses 구현