Chapter 3: Variables and Binding

Chapter 2의 계산기에 변수 바인딩을 추가하여 표현식 내에서 값을 재사용할 수 있게 한다.

개요

이 chapter에서 추가하는 기능:

- 변수 참조 (x, foo, _bar)
- Let 바인딩 (`let x = 5 in x + 1`)
- 중첩 스코프 (`let x = 1 in let y = 2 in x + y`)
- 섀도잉 (`let x = 1 in let x = 2 in x`)
- 환경 기반 평가 (Environment-passing evaluation)

AST 확장

`Ast.fs`에 변수 관련 노드를 추가한다.

```
module Ast

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           // 변수 참조
    | Let of string * Expr * Expr  // let name = expr1 in expr2
```

설계 포인트: - Var: 변수 이름만 저장 (값은 평가 시 환경에서 조회) - Let: 이름, 바인딩 표현식, 본문 표현식 세 부분

예시:

```
let x = 5 in x + 1

Let("x", Number 5, Add(Var "x", Number 1))
```

Lexer 확장

`Lexer.fsl`에 키워드와 식별자 규칙을 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!
    | "let"         { LET }
    | "in"          { IN }
    // 식별자: 문자 또는 _로 시작
    | ident_start ident_char* { IDENT (lexeme lexbuf) }
    | '+'           { PLUS }
    | '-'           { MINUS }
    | '*'           { STAR }
    | '/'           { SLASH }
    | '('           { LPAREN }
    | ')'           { RPAREN }
    | '='           { EQUALS }
    | eof           { EOF }
```

핵심 포인트: 키워드 우선순위

```
// 올바른 순서
| "let"         { LET }
| "in"          { IN }
| ident_start ident_char* { IDENT (lexeme lexbuf) }

// 잘못된 순서 (let이 IDENT로 인식됨)
| ident_start ident_char* { IDENT (lexeme lexbuf) }
| "let"         { LET }  // 절대 매칭되지 않음!
```

fslex는 규칙 순서대로 매칭을 시도한다. `"let"`과 `ident_start ident_char*` 둘 다 “let” 문자열에 매칭되므로, 키워드 규칙이 먼저 와야 한다.

Parser 확장

Parser.fsy에 새 토큰과 문법 규칙을 추가한다.

```
%{
open Ast
%}

%token <int> NUMBER
%token <string> IDENT
%token PLUS MINUS STAR SLASH
%token LPAREN RPAREN
%token LET IN EQUALS
%token EOF

%start start
```

```
%type <Ast.Expr> start

%%

start:
    | Expr EOF           { $1 }

Expr:
    // Let 표현식 - 가장 낮은 우선순위
    | LET IDENT EQUALS Expr IN Expr  { Let($2, $4, $6) }
    | 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) }
    | LPAREN Expr RPAREN { $2 }
    | MINUS Factor       { Negate($2) }
```

Let의 우선순위:

Let이 Expr 레벨에 있으므로 가장 낮은 우선순위를 갖는다:

```
let x = 2 + 3 in x * 4
→ Let("x", Add(2, 3), Multiply(Var "x", 4))
```

IDENT의 위치:

Factor에서 IDENT를 처리하므로 변수는 괄호 없이 직접 사용 가능:

```
x + 1      → Add(Var "x", Number 1)
x * y      → Multiply(Var "x", Var "y")
```

Evaluator: 환경 기반 평가

변수를 지원하려면 "환경(Environment)"이 필요하다. 환경은 변수 이름에서 값으로의 매핑이다.

Eval.fs를 수정한다.

```
module Eval

open Ast

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

/// 빈 환경 (최상위 평가용)
let emptyEnv : Env = Map.empty

/// 환경 내에서 표현식 평가
```

```fsharp
/// 정의되지 않은 변수는 예외 발생
let rec eval (env: Env) (expr: Expr) : int =
    match expr with
    | Number n -> n

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

    | Let (name, binding, body) ->
        // 1. 현재 환경에서 바인딩 평가
        let value = eval env binding
        // 2. 환경 확장 (새 바인딩 추가)
        let extendedEnv = Map.add name value env
        // 3. 확장된 환경에서 본문 평가
        eval extendedEnv body

    | Add (left, right) ->
        eval env left + eval env right

    | Subtract (left, right) ->
        eval env left - eval env right

    | Multiply (left, right) ->
        eval env left * eval env right

    | Divide (left, right) ->
        eval env left / eval env right

    | Negate e ->
        -(eval env e)

/// 편의 함수: 빈 환경에서 평가
let evalExpr (expr: Expr) : int =
    eval emptyEnv expr
```

핵심 개념:

1. **환경 전파**: 모든 재귀 호출에 env를 전달
2. **환경 확장**: Map.add로 새 바인딩 추가 (불변 맵이므로 원본 유지)
3. **스코프**: Let 본문에서만 확장된 환경 사용

Let 평가 흐름:

```
let x = 2 + 3 in x * 4
1. env = {}
2. eval {} (2 + 3) = 5
3. extendedEnv = {x: 5}
4. eval {x: 5} (x * 4)
   - eval {x: 5} x = 5
   - eval {x: 5} 4 = 4
   - 5 * 4 = 20
결과: 20
```

스코프 규칙

중첩 스코프

내부 Let은 외부 변수에 접근 가능:

`let x = 5 in let y = x + 1 in y`

평가:

```
1. env = {}
2. {x: 5}에서 x + 1 = 6
3. {x: 5, y: 6}에서 y = 6
결과: 6
```

섀도잉 (Shadowing)

같은 이름으로 재바인딩하면 내부에서 새 값 사용:

`let x = 1 in let x = 2 in x`

평가:

```
1. env = {}
2. {x: 1}
3. {x: 2}  (Map.add로 덮어씀)
4. x = 2
결과: 2
```

섀도잉 후 외부 스코프:

`let x = 1 in (let x = 2 in x) + x`

평가:

```
1. env = {}
2. env1 = {x: 1}
3. (let x = 2 in x) 평가:
   - env2 = {x: 2}
   - 결과: 2
4. x (env1에서) = 1
5. 2 + 1 = 3
결과: 3
```

불변 맵 덕분에 내부 스코프가 외부에 영향을 주지 않는다.

Program.fs

evalExpr로 호출을 변경한다.

```
[<EntryPoint>]
let main argv =
    match argv with
    | [| "--expr"; expr |] ->
        try
            let result = expr |> parse |> evalExpr  // eval → evalExpr
            printfn "%d" result
            0
        with ex ->
            eprintfn "Error: %s" ex.Message
```

```
            1
    // ... 나머지 옵션
```

Examples

기본 Let 바인딩

```
$ dotnet run --project FunLang -- --expr "let x = 5 in x"
5

$ dotnet run --project FunLang -- --expr "let x = 2 + 3 in x"
5

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

$ dotnet run --project FunLang -- --expr "let x = 10 in x * 2 + 5"
25
```

변수 참조

```
$ dotnet run --project FunLang -- --expr "let x = 3 in x * 4"
12

$ dotnet run --project FunLang -- --expr "let x = 2 in x + x"
4

$ dotnet run --project FunLang -- --expr "let x = 5 in x * x"
25
```

중첩 Let

```
$ dotnet run --project FunLang -- --expr "let x = 1 in let y = 2 in x + y"
3

$ dotnet run --project FunLang -- --expr "let x = 5 in let y = x + 1 in y"
6

$ dotnet run --project FunLang -- --expr "let a = 1 in let b = 2 in let c = 3 in a + b + c"
6
```

섀도잉

```
$ dotnet run --project FunLang -- --expr "let x = 1 in let x = 2 in x"
2

$ dotnet run --project FunLang -- --expr "let x = 1 in (let x = 2 in x) + x"
```

```
3

$ dotnet run --project FunLang -- --expr "let x = 1 in (let y = x + 1 in y) + x"
3
```

정의되지 않은 변수

```
$ dotnet run --project FunLang -- --expr "x"
Error: Undefined variable: x

$ dotnet run --project FunLang -- --expr "y + 1"
Error: Undefined variable: y
```

디버깅: 토큰 및 AST 출력

```
$ dotnet run --project FunLang -- --emit-tokens --expr "let x = 5 in x"
LET
IDENT(x)
EQUALS
NUMBER(5)
IN
IDENT(x)
EOF

$ dotnet run --project FunLang -- --emit-ast --expr "let x = 5 in x"
Let(x, Number(5), Var(x))

$ dotnet run --project FunLang -- --emit-ast --expr "let x = 1 in let y = 2 in x + y"
Let(x, Number(1), Let(y, Number(2), Add(Var(x), Var(y))))
```

테스트

fslit (CLI 테스트)

tests/variables/ 디렉토리에 12개 테스트:

```
$ make -C tests variables
...
PASS: all 12 tests passed
```

Expecto (단위 테스트)

```
$ dotnet run --project FunLang.Tests
[13:25:00 INF] EXPECTO? Running tests...
[13:25:01 INF] EXPECTO! 58 tests run – 58 passed, 0 failed. Success!
```

요약

파일	변경 사항
`Ast.fs`	`Var`, `Let` 노드 추가
`Lexer.fsl`	`LET`, `IN`, `EQUALS`, `IDENT` 토큰, 키워드 우선 규칙
`Parser.fsy`	`Let` 문법, `IDENT` → `Var` 변환
`Eval.fs`	`Env` 타입, 환경 기반 `eval`, `evalExpr` 래퍼
`Program.fs`	`evalExpr` 사용

핵심 개념

개념	설명
Environment	변수 이름 → 값 매핑 (`Map<string, int>`)
Lexical Scope	코드 구조가 변수 가시성 결정
Shadowing	같은 이름 재바인딩 시 내부 값 우선
Immutable Map	스코프 간 격리 보장

다음 Chapter

Chapter 4에서는 조건문 (`if-then-else`)과 Boolean 타입을 추가한다.

관련 문서

- write-fslex-lexer — fslex 렉서 작성, 키워드 우선 규칙
- write-fsyacc-parser — fsyacc 파서 작성
- write-fscheck-property-tests — FsCheck로 변수 속성 검증
- testing-strategies — 테스트 전략 가이드