Chapter 2: Arithmetic Expressions

Chapter 1에서 구축한 파이프라인에 사칙연산을 추가하여 실제로 계산하는 인터프리터를 만든다.

개요

이 chapter에서 추가하는 기능:

- 사칙연산 (+, -, *, /)
- 연산자 우선순위 (*, /가 +, -보다 먼저)
- 괄호로 우선순위 변경
- 단항 마이너스 (-5, --5, -(1+2))
- Evaluator 컴포넌트

AST 확장

Ast.fs에 산술 연산 노드를 추가한다.

```
module Ast

/// Expression AST for arithmetic operations
/// Phase 2: Arithmetic expressions with precedence
type Expr =
    | Number of int
    | Add of Expr * Expr
    | Subtract of Expr * Expr
    | Multiply of Expr * Expr
    | Divide of Expr * Expr
    | Negate of Expr  // Unary minus
```

설계 포인트: - 각 연산자마다 별도 케이스 (타입 안전성) - Negate는 단항 연산자 (표현식 하나만 받음) - 재귀 구조 (Expr * Expr)로 중첩 표현식 지원

Parser: Expr/Term/Factor 패턴

연산자 우선순위를 처리하는 핵심 패턴이다.

```
// Parser.fsy

%{
open Ast
%}

%token <int> NUMBER
%token PLUS MINUS STAR SLASH
%token LPAREN RPAREN
%token EOF

%start start
%type <Ast.Expr> start

%%
```

```
// Expr/Term/Factor 패턴으로 우선순위 인코딩
// 낮은 우선순위 = 높은 문법 레벨

start:
    | Expr EOF           { $1 }

Expr:                               // + - (낮은 우선순위)
    | Expr PLUS Term     { Add($1, $3) }
    | Expr MINUS Term    { Subtract($1, $3) }
    | Term               { $1 }

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

Factor:                             // 숫자, 괄호, 단항 (가장 높은 우선순위)
    | NUMBER             { Number($1) }
    | LPAREN Expr RPAREN { $2 }
    | MINUS Factor       { Negate($2) }
```

왜 이 패턴을 사용하나?

FsYacc의 `%left`/`%right` 선언에는 알려진 버그가 있다. Expr/Term/Factor 패턴은 문법 구조 자체로 우선순위를 표현하므로 버그 영향이 없다.

파싱 예시:

`2 + 3 * 4`:

```
Expr
├─── Expr → Term → Factor → 2
├─── PLUS
└─── Term
     ├─── Term → Factor → 3
     ├─── STAR
     └─── Factor → 4
```

결과: `Add(Number 2, Multiply(Number 3, Number 4))`

Lexer 확장

`Lexer.fsl`에 연산자 토큰을 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')

rule tokenize = parse
    | whitespace+   { tokenize lexbuf }
    | newline       { tokenize lexbuf }
    | digit+        { NUMBER (Int32.Parse(lexeme lexbuf)) }
    | '+'           { PLUS }
    | '-'           { MINUS }
    | '*'           { STAR }
    | '/'           { SLASH }
    | '('           { LPAREN }
    | ')'           { RPAREN }
    | eof           { EOF }
```

핵심 포인트: - MINUS는 단일 토큰 (단항/이항 구분은 파서가 함) - 괄호 (,)도 토큰으로 정의

Evaluator

새로운 파일 Eval.fs를 추가한다. 패턴 매칭으로 AST를 평가한다.

```
module Eval

open Ast

/// Evaluate an expression to an integer result
let rec eval (expr: Expr) : int =
    match expr with
    | Number n -> n
    | Add (left, right) -> eval left + eval right
    | Subtract (left, right) -> eval left - eval right
    | Multiply (left, right) -> eval left * eval right
    | Divide (left, right) -> eval left / eval right
    | Negate e -> -(eval e)
```

빌드 순서 업데이트 (FunLang.fsproj):

```
<Compile Include="Ast.fs" />
<!-- Parser, Lexer 생성 -->
<Compile Include="Parser.fsi" />
<Compile Include="Parser.fs" />
<Compile Include="Lexer.fs" />
<Compile Include="Eval.fs" />      <!-- NEW -->
<Compile Include="Program.fs" />
```

Program.fs

CLI 인터페이스를 추가한다.

```
open System
open FSharp.Text.Lexing
open Ast
open Eval
```

```
let parse (input: string) : Expr =
    let lexbuf = LexBuffer<char>.FromString input
    Parser.start Lexer.tokenize lexbuf

[<EntryPoint>]
let main argv =
    match argv with
    | [| "--expr"; expr |] ->
        try
            let result = expr |> parse |> eval
            printfn "%d" result
            0
        with ex ->
            eprintfn "Error: %s" ex.Message
            1
    | _ ->
        eprintfn "Usage: funlang --expr <expression>"
        1
```

Examples

기본 연산

```
$ dotnet run --project FunLang -- --expr "2 + 3"
5

$ dotnet run --project FunLang -- --expr "10 - 4"
6

$ dotnet run --project FunLang -- --expr "3 * 4"
12

$ dotnet run --project FunLang -- --expr "20 / 4"
5
```

연산자 우선순위

```
$ dotnet run --project FunLang -- --expr "2 + 3 * 4"
14

$ dotnet run --project FunLang -- --expr "10 - 6 / 2"
7
```

괄호

```
$ dotnet run --project FunLang -- --expr "(2 + 3) * 4"
20
```

```
$ dotnet run --project FunLang -- --expr "((2 + 3) * (4 - 1))"
15
```

단항 마이너스

```
$ dotnet run --project FunLang -- --expr "-5"
-5

$ dotnet run --project FunLang -- --expr "-5 + 3"
-2

$ dotnet run --project FunLang -- --expr "--5"
5

$ dotnet run --project FunLang -- --expr "-(2 + 3)"
-5
```

좌결합성

```
$ dotnet run --project FunLang -- --expr "2 - 3 - 4"
-5

$ dotnet run --project FunLang -- --expr "24 / 4 / 2"
3
```

테스트

별도의 테스트 프로젝트 FunLang.Tests에서 Expecto로 테스트한다.

```
$ dotnet run --project FunLang.Tests
[13:25:00 INF] EXPECTO? Running tests...
[13:25:01 INF] EXPECTO! 19 tests run – 19 passed, 0 failed. Success!
```

요약

파일	변경 사항
Ast.fs	Add, Subtract, Multiply, Divide, Negate 추가
Parser.fsy	Expr/Term/Factor 문법 패턴
Lexer.fsl	+, -, *, /, (,) 토큰
Eval.fs	**NEW** - 재귀 평가 함수
Program.fs	–expr CLI 인터페이스
FunLang.fsproj	Eval.fs 빌드 순서 추가

다음 Chapter

Chapter 3에서는 변수 바인딩 (`let x = 5`)과 스코프 (`let x = 1 in x + 1`)를 추가한다.

관련 문서

- fsyacc-precedence-without-declarations — Expr/Term/Factor 패턴 상세
- fsyacc-operator-precedence-methods — 우선순위 처리 방법 비교
- implement-unary-minus — 단항 마이너스 구현
- write-fsyacc-parser — fsyacc 파서 작성
- setup-expecto-test-project — Expecto 테스트 설정