

MiniFSharp – A Small Subset of F# Language

1 INTRODUCTION

MiniFSharp is a toy language for the [COSE212](#) course at Korea University. MiniFSharp is inspired by the F# programming language.¹ The name MiniFSharp indicates that it is a minimal subset of the F# language, and it supports the following features:

- **number (non-negative integer) values** (0, 1, 2, 3, ...)
- **boolean values** (true and false)
- **arithmetic operators**: negation (-), addition (+), subtraction (-), multiplication (*), division (/), and modulo (%)
- **logical operators**: conjunction (&&), disjunction (||), and negation (!)
- **comparison operators**: equality (= and <>) and relational (<, >, <=, and >=)
- **conditionals** (if-then-else)
- **lists** (:: and [e1, ..., en] where $n \geq 0$)
- **tuples** (()) and e1, ..., en where $n \geq 2$)
- **options** (None and Some e)
- **immutable variable definitions with patterns** (let)
- **mutually recursive functions with patterns** (let rec)
- **first-class functions with patterns** (->)
- **function applications** (f e)
- **pattern matching** (match-with)

This document is the specification of MiniFSharp. First, Section 2 describes the concrete syntax, and Section 3 describes the abstract syntax with the desugaring rules. Then, Section 4 describes the big-step operational (natural) semantics of MiniFSharp.

2 CONCRETE SYNTAX

The concrete syntax of MiniFSharp is written in a variant of the extended Backus–Naur form (EBNF). The notation <nt> denotes a nonterminal, and "t" denotes a terminal. We use ? to denote an optional element and + (or *) to denote one or more (or zero or more) repetitions of the preceding element. We use **butnot** to denote a set difference to exclude some strings from a producible set of strings. We omit some obvious terminals using the ellipsis (...) notation.

```
// basic elements
<digit>      ::= "0" | "1" | "2" | ... | "9"
<number>     ::= <digit>+
<boolean>    ::= "true" | "false"
<alphabet>   ::= "A" | "B" | "C" | ... | "Z" | "a" | "b" | "c" | ... | "z"
<idstart>    ::= <alphabet> | "_"
<idcont>     ::= <alphabet> | "_" | <digit>
<keyword>    ::= "and" | "elif" | "else" | "false" | "fun" | "if" | "in" | "let"
              | "match" | "rec" | "then" | "true" | "with"
<id>         ::= <idstart> <idcont>* butnot <keyword>
```

¹<https://fsharp.org/>

```

// expressions
<expr> ::= <number> | <boolean> | <id> | "(" <expr> ")"
        // unary and binary operators
        | <uop> <expr> | <expr> <bop> <expr>
        // conditionals
        | "if" <expr> "then" <expr>
        | "elif" <expr> "then" <expr>* "else" <expr>
        // lists
        | "[" "]" | "[" <expr> ";" <expr>]* "]" | <expr> "::" <expr>
        // tuples
        | "(" ")" | <expr> ["," <expr>]+
        // immutable variable definitions with patterns
        | "let" <pattern> "=" <expr> "in" <expr>
        | "let" <id> <pattern>+ "=" <expr> "in" <expr>
        // mutually recursive functions with patterns
        | "let" "rec" <named-fun> ["and" <named-fun>]* "in" <expr>
        // first-class functions
        | "fun" <pattern>+ "->" <expr>
        // function applications
        | <expr> <expr>
        // pattern matching
        | "match" <expr> "with" ["|" <pattern> "->" <expr>]+

// operators
<uop>  ::= "-" | "!"
<bop>  ::= "+" | "-" | "*" | "/" | "%" | "&&" | "||"
        | "=" | "<" | ">" | "<=" | ">=" | ">="

// patterns
<pattern> ::= <number> | <boolean> | <id> | "[" "]"
        | "[" <pattern> ";" <pattern>]* "]" | <pattern> "::" <pattern>
        | "(" ")" | <pattern> ["," <pattern>]+
        | None | "Some" <pattern>

// named functions
<named-fun> ::= <id> <pattern>+ "=" <expr>

```

The precedence and associativity of operators are defined as follows:

| Description | Operator | Precedence | Associativity |
|----------------------|--------------|------------|---------------|
| Unary | -, ! | 11 | right |
| Function Application | . | 10 | left |
| Multiplicative | *, /, % | 9 | left |
| Additive | +, - | 8 | |
| List Construction | :: | 7 | right |
| Relational | <, <=, >, >= | 6 | left |
| Equality | =, <> | 5 | |
| Logical Conjunction | && | 4 | |
| Logical Disjunction | | 3 | |
| Function Definition | -> | 2 | |
| Let Binding | let | 1 | |

3 ABSTRACT SYNTAX

The abstract syntax of MiniFSharp is defined as follows:

| | | | | | |
|--------|-------------|--------------------------|---------|--------------------------------------|----------|
| (Expr) | Expressions | $\mathbb{E} \ni e ::= n$ | (ENum) | [] | (ENil) |
| | | b | (EBool) | $e :: e$ | (ECons) |
| | | x | (EId) | e, \dots, e | (ETuple) |
| | | $- e$ | (ENeg) | None | (ENone) |
| | | $e + e$ | (EAdd) | Some e | (ESome) |
| | | $e * e$ | (EMul) | let $\pi = e$ in e | (ELet) |
| | | e / e | (EDiv) | let rec $f \dots f$ in e | (ERec) |
| | | $e \% e$ | (EMod) | $\lambda \pi. e$ | (EFun) |
| | | $e = e$ | (EEq) | $e e$ | (EApp) |
| | | $e < e$ | (ELt) | match e with $\kappa \dots \kappa$ | (EMatch) |
| | | if $e e e$ | (EIf) | | |

| | | | | | | | |
|-----------|----------|---------------------|---------|-------------------|----------|------------|---------|
| (Pattern) | Patterns | $\Pi \ni \pi ::= n$ | (PNum) | [] | (PNil) | None | (PNone) |
| | | b | (PBool) | $\pi :: \pi$ | (PCons) | Some π | (PSome) |
| | | x | (PId) | π, \dots, π | (PTuple) | | |

where

| | | | | | |
|-------------|--------------------|----------|-----------------|--|------------|
| Identifiers | $x \in \mathbb{X}$ | (String) | Booleans | $\mathbb{B} \ni b ::= \text{true} \mid \text{false}$ | (Boolean) |
| Numbers | $n \in \mathbb{Z}$ | (BigInt) | Named Functions | $\mathbb{F} \ni f ::= x(\pi) = e$ | (NamedFun) |
| | | | Match Cases | $\mathbb{K} \ni \kappa ::= \pi \rightarrow e$ | (Case) |

The semantics of the remaining cases are defined with the following desugaring rules:

| | |
|--|---|
| $\mathcal{D}[\text{! } e]$ | $= \text{if } \mathcal{D}[e] \text{ then false else true}$ |
| $\mathcal{D}[e_1 - e_2]$ | $= \mathcal{D}[e_1] + \mathcal{D}[- e_2]$ |
| $\mathcal{D}[e_1 \&\& e_2]$ | $= \text{if } \mathcal{D}[e_1] \text{ then } \mathcal{D}[e_2] \text{ else false}$ |
| $\mathcal{D}[e_1 \mid\mid e_2]$ | $= \text{if } \mathcal{D}[e_1] \text{ then true else } \mathcal{D}[e_2]$ |
| $\mathcal{D}[e_1 <> e_2]$ | $= \mathcal{D}[\text{! } (e_1 = e_2)]$ |
| $\mathcal{D}[e_1 <= e_2]$ | $= \mathcal{D}[(e_1 < e_2) \mid\mid (e_1 = e_2)]$ |
| $\mathcal{D}[e_1 > e_2]$ | $= \mathcal{D}[\text{! } (e_1 <= e_2)]$ |
| $\mathcal{D}[e_1 >= e_2]$ | $= \mathcal{D}[\text{! } (e_1 < e_2)]$ |
| $\mathcal{D}[[e_1; \dots; e_n]]$ | $= \mathcal{D}[e_1] :: \dots :: \mathcal{D}[e_n] :: []$ |
| $\mathcal{D}[\text{let } x \pi_1 \dots \pi_n = e_1 \text{ in } e_2]$ | $= \text{let } x = \text{fun } \mathcal{D}[\pi_1] \rightarrow \dots \text{fun } \mathcal{D}[\pi_n] \rightarrow \mathcal{D}[e_1] \text{ in } \mathcal{D}[e_2]$ |
| $\mathcal{D}[\text{fun } \pi_0 \pi_1 \dots \pi_n \rightarrow e]$ | $= \text{fun } \mathcal{D}[\pi_0] \rightarrow \text{fun } \mathcal{D}[\pi_1] \rightarrow \dots \text{fun } \mathcal{D}[\pi_n] \rightarrow \mathcal{D}[e]$ |
| $\mathcal{D}[x \pi_0 \pi_1 \dots \pi_n = e]$ | $= x \mathcal{D}[\pi_0] = \text{fun } \mathcal{D}[\pi_1] \rightarrow \dots \text{fun } \mathcal{D}[\pi_n] \rightarrow \mathcal{D}[e]$ |
| $\mathcal{D}[[\pi_1; \dots; \pi_n]]$ | $= \mathcal{D}[\pi_1] :: \dots :: \mathcal{D}[\pi_n] :: []$ |

$$\mathcal{D} \left[\begin{array}{l} \text{if } e \text{ then } e' \\ \text{elif } e_1 \text{ then } e'_1 \\ \dots \\ \text{elif } e_n \text{ then } e'_n \\ \text{else } e'' \end{array} \right] = \text{if } \mathcal{D}[e] \text{ then } \mathcal{D}[e'] \text{ else } \mathcal{D} \left[\begin{array}{l} \text{if } e_1 \text{ then } e'_1 \\ \dots \\ \text{elif } e_n \text{ then } e'_n \\ \text{else } e'' \end{array} \right]$$

The omitted cases recursively apply the desugaring rule to sub-expressions or sub-patterns.

with the following auxiliary function eq for checking the equality of two values:

$$\boxed{\text{eq} : \mathbb{V} \times \mathbb{V} \rightarrow \mathbb{B}}$$

$$\begin{aligned} \text{eq}(n, n) &= \text{true} \\ \text{eq}(b, b) &= \text{true} \\ \text{eq}([v_1, \dots, v_n], [v'_1, \dots, v'_n]) &= \text{eq}(v_1, v'_1) \wedge \dots \wedge \text{eq}(v_n, v'_n) \\ \text{eq}((v_1, \dots, v_n), (v'_1, \dots, v'_n)) &= \text{eq}(v_1, v'_1) \wedge \dots \wedge \text{eq}(v_n, v'_n) \\ \text{eq}(\text{None}, \text{None}) &= \text{true} \\ \text{eq}(\text{Some } v, \text{Some } v') &= \text{eq}(v, v') \end{aligned}$$

Note that eq is a total function and should return false for the cases not listed above. On the other hand, the following auxiliary function extend is a partial function (defined with \rightarrow) for extending an environment with a pattern and a value only if the pattern matches the value.

$$\boxed{\text{extend} : ((\mathbb{X} \xrightarrow{\text{fin}} \mathbb{V}) \times \Pi \times \mathbb{V}) \rightarrow (\mathbb{X} \xrightarrow{\text{fin}} \mathbb{V})}$$

$$\begin{aligned} \text{extend}(\sigma, n, n) &= \sigma \\ \text{extend}(\sigma, b, b) &= \sigma \\ \text{extend}(\sigma, x, v) &= \sigma[x \mapsto v] \\ \text{extend}(\sigma, \text{None}, \text{None}) &= \sigma \\ \text{extend}(\sigma, \text{Some } \pi, \text{Some } v) &= \text{extend}(\sigma, \pi, v) \\ \text{extend}(\sigma, [], []) &= \sigma \\ \text{extend}(\sigma, \pi_1 :: \pi_2, [v_1, \dots, v_n]) &= \text{extend}(\text{extend}(\sigma, \pi_1, v_1), \pi_2, [v_2, \dots, v_n]) \\ \text{extend}(\sigma, (\pi_1, \dots, \pi_n), (v_1, \dots, v_n)) &= \sigma_n \end{aligned}$$

where $\sigma_i = \text{extend}(\sigma_{i-1}, \pi_i, v_i)$ for $1 \leq i \leq n$ and $\sigma_0 = \sigma$.