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>+
<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>
```

```
// expressions
<expr> ::= <number> | "true" | "false" | <id> | "(" <expr> ")"
```

[1] https://fsharp.org/

```
          // 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> ::= <num> | <bool> | <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:

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

$$
\begin{array}{lllllllll}
(\texttt{Pattern}) & \text{Patterns} & \Pi \ni \pi ::= n & (\texttt{PNum}) & \mid \texttt{[]} & (\texttt{PNil}) & \mid \texttt{None} & (\texttt{PNone}) \\
 & & \mid b & (\texttt{PBool}) & \mid \pi :: \pi & (\texttt{PCons}) & \mid \texttt{Some}\ \pi & (\texttt{PSome}) \\
 & & \mid x & (\texttt{PId}) & \mid \pi, \ldots, \pi & (\texttt{PTuple}) & &
\end{array}
$$

where

$$
\begin{array}{llllll}
\text{Identifiers} & x \in \mathbb{X} & (\texttt{String}) & \text{Booleans} & \mathbb{B} \ni b ::= \texttt{true} \mid \texttt{false} & (\texttt{Boolean}) \\
\text{Numbers} & n \in \mathbb{Z} & (\texttt{BigInt}) & \text{Named Functions} & \mathbb{F} \ni f ::= x(\pi) = e & (\texttt{NamedFun}) \\
 & & & \text{Match Cases} & \mathbb{K} \ni \kappa ::= \pi \to e & (\texttt{Case})
\end{array}
$$

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

$$
\begin{array}{ll}
\mathcal{D}[\![\ !\ e]\!] & = \texttt{if}\ \mathcal{D}[\![e]\!]\ \texttt{then false else true} \\
\mathcal{D}[\![e_1 - e_2]\!] & = \mathcal{D}[\![e_1]\!] + \mathcal{D}[\![- e_2]\!] \\
\mathcal{D}[\![e_1\ \&\&\ e_2]\!] & = \texttt{if}\ \mathcal{D}[\![e_1]\!]\ \texttt{then}\ \mathcal{D}[\![e_2]\!]\ \texttt{else false} \\
\mathcal{D}[\![e_1\ \texttt{||}\ e_2]\!] & = \texttt{if}\ \mathcal{D}[\![e_1]\!]\ \texttt{then true else}\ \mathcal{D}[\![e_2]\!] \\
\mathcal{D}[\![e_1 <> e_2]\!] & = \mathcal{D}[\![\ !\ (e_1 = e_2)]\!] \\
\mathcal{D}[\![e_1 <= e_2]\!] & = \mathcal{D}[\![(e_1 < e_2)\ \texttt{||}\ (e_1 = e_2)]\!] \\
\mathcal{D}[\![e_1 > e_2]\!] & = \mathcal{D}[\![\ !\ (e_1 <= e_2)]\!] \\
\mathcal{D}[\![e_1 >= e_2]\!] & = \mathcal{D}[\![\ !\ (e_1 < e_2)]\!] \\
\mathcal{D}[\![\texttt{[} e_1 \texttt{;} \ldots \texttt{;} e_n \texttt{]}]\!] & = \mathcal{D}[\![e_1]\!]\ \texttt{::} \ldots \texttt{::}\ \mathcal{D}[\![e_n]\!]\ \texttt{::}\ \texttt{[]}
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{D}[\![\texttt{let}\ x\ \pi_1\ \ldots\ \pi_n = e_1\ \texttt{in}\ e_2]\!] & = \texttt{let}\ x = \texttt{fun}\ \mathcal{D}[\![\pi_1]\!]\ \texttt{->}\ \ldots\ \texttt{fun}\ \mathcal{D}[\![\pi_n]\!]\ \texttt{->}\ \mathcal{D}[\![e_1]\!]\ \texttt{in}\ \mathcal{D}[\![e_2]\!] \\
\mathcal{D}[\![\texttt{fun}\ \pi_0\ \pi_1\ \ldots\ \pi_n\ \texttt{->}\ e]\!] & = \texttt{fun}\ \mathcal{D}[\![\pi_0]\!]\ \texttt{->}\ \texttt{fun}\ \mathcal{D}[\![\pi_1]\!]\ \texttt{->}\ \ldots\ \texttt{fun}\ \mathcal{D}[\![\pi_n]\!]\ \texttt{->}\ \mathcal{D}[\![e]\!] \\
\mathcal{D}[\![x\ \pi_0\ \pi_1\ \ldots\ \pi_n = e]\!] & = x\ \mathcal{D}[\![\pi_0]\!] = \texttt{fun}\ \mathcal{D}[\![\pi_1]\!]\ \texttt{->}\ \ldots\ \texttt{fun}\ \mathcal{D}[\![\pi_n]\!]\ \texttt{->}\ \mathcal{D}[\![e]\!] \\
\mathcal{D}[\![\texttt{[} \pi_1 \texttt{;} \ldots \texttt{;} \pi_n \texttt{]}]\!] & = \mathcal{D}[\![\pi_1]\!]\ \texttt{::} \ldots \texttt{::}\ \mathcal{D}[\![\pi_n]\!]\ \texttt{::}\ \texttt{[]}
\end{array}
$$

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

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

4 SEMANTICS

We use the following notations in the semantics:

$$
\begin{array}{llllllll}
\text{Value} & \mathbb{V} \ni v ::= n & (\texttt{NumV}) & \mid [v, \dots, v] & (\texttt{ListV}) & \mid \langle \lambda\pi.e, \sigma\rangle & (\texttt{CloV}) \\
 & \quad\mid b & (\texttt{BoolV}) & \mid (v, \dots, v) & (\texttt{TupleV}) & & \\
\text{Environment} & \sigma \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{V} & (\texttt{Env}) & & & &
\end{array}
$$

The big-step operational (natural) semantics of MiniFSharp is defined as follows:

$$\boxed{\sigma \vdash e \Rightarrow v}$$

$$
\texttt{Num}\ \frac{}{\sigma \vdash n \Rightarrow n}
\qquad
\texttt{Bool}\ \frac{}{\sigma \vdash b \Rightarrow b}
\qquad
\texttt{Id}\ \frac{x \in \text{Domain}(\sigma)}{\sigma \vdash x \Rightarrow \sigma(x)}
$$

$$
\texttt{Neg}\ \frac{\sigma \vdash e \Rightarrow n}{\sigma \vdash \texttt{-}\ e \Rightarrow -n_2}
\qquad
\texttt{Add}\ \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 \texttt{ + } e_2 \Rightarrow n_1 + n_2}
\qquad
\texttt{Mul}\ \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 \texttt{ * } e_2 \Rightarrow n_1 \times n_2}
$$

$$
\texttt{Div}\ \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2 \qquad n_2 \neq 0}{\sigma \vdash e_1 \texttt{ / } e_2 \Rightarrow n_1 \div n_2}
\qquad
\texttt{Mod}\ \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2 \qquad n_2 \neq 0}{\sigma \vdash e_1 \texttt{ \% } e_2 \Rightarrow n_1 \,\%\, n_2}
$$

$$
\texttt{Eq}\ \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_1 \texttt{ = } e_2 \Rightarrow \mathsf{eq}(v_1, v_2)}
\qquad
\texttt{Lt}\ \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 \texttt{ < } e_2 \Rightarrow n_1 < n_2}
$$

$$
\texttt{If}_T\ \frac{\sigma \vdash e_0 \Rightarrow \texttt{true} \qquad \sigma \vdash e_1 \Rightarrow v_1}{\sigma \vdash \texttt{if}\ e_0\ e_1\ e_2 \Rightarrow v_1}
\qquad
\texttt{If}_F\ \frac{\sigma \vdash e_0 \Rightarrow \texttt{false} \qquad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash \texttt{if}\ e_0\ e_1\ e_2 \Rightarrow v_2}
$$

$$
\texttt{Nil}\ \frac{}{\sigma \vdash \texttt{[]} \Rightarrow \texttt{[]}}
\qquad
\texttt{Cons}\ \frac{\sigma \vdash e_1 \Rightarrow v \qquad \sigma \vdash e_2 \Rightarrow [v_1, \dots, v_n]}{\sigma \vdash e_1 :: e_2 \Rightarrow [v, v_1, \dots, v_n]}
\qquad
\texttt{None}\ \frac{}{\sigma \vdash \mathsf{None} \Rightarrow \mathsf{None}}
$$

$$
\texttt{Some}\ \frac{\sigma \vdash e \Rightarrow v}{\sigma \vdash \mathsf{Some}\ e \Rightarrow \mathsf{Some}\ v}
\qquad
\texttt{Tuple}\ \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \dots \qquad \sigma \vdash e_n \Rightarrow v_n}{\sigma \vdash (e_1, \dots, e_n) \Rightarrow (v_1, \dots, v_n)}
$$

$$
\texttt{Let}\ \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \mathsf{extend}(\sigma, \pi, v_1) \vdash e_2 \Rightarrow v_2}{\sigma \vdash \texttt{let}\ \pi \texttt{ = } e_1\ \texttt{in}\ e_2 \Rightarrow v_2}
\qquad
\texttt{Fun}\ \frac{}{\sigma \vdash \lambda\pi.e \Rightarrow \langle \lambda\pi.e, \sigma\rangle}
$$

$$
\texttt{Rec}\ \frac{\sigma' = \sigma[x_1 \mapsto \langle \lambda\pi_1.e_1, \sigma'\rangle, \dots x_n \mapsto \langle \lambda\pi_n.e_n, \sigma'\rangle,] \qquad \sigma' \vdash e \Rightarrow v}{\sigma \vdash \texttt{let rec}\ x_1(\pi_1) \texttt{ = } e_1\ \dots x_n(\pi_n) \texttt{ = } e_n\ \texttt{in}\ e \Rightarrow v}
$$

$$
\texttt{App}\ \frac{\sigma \vdash e_0 \Rightarrow \langle \lambda\pi_2.e_2, \sigma_2\rangle \qquad \sigma \vdash e_1 \Rightarrow v_1 \qquad \mathsf{extend}(\sigma_2, \pi_2, v_1) \vdash e_2 \Rightarrow v}{\sigma \vdash e_0 e_1 \Rightarrow v}
$$

$$
\texttt{Match}\ \frac{\sigma \vdash e \Rightarrow v \qquad i = \min\{j \mid (\sigma, \pi_j, v) \in \mathsf{extend}\} \qquad \mathsf{extend}(\sigma, \pi_i, v) \vdash e_i \Rightarrow v_i}{\sigma \vdash \texttt{match}\ e\ \texttt{with}\ \kappa\ \dots\ \kappa\pi_1 \rightarrow e_1 \dots \pi_n \rightarrow e_n \Rightarrow v_i}
$$

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

$$\boxed{\mathsf{eq} : \mathbb{V} \times \mathbb{V} \to \mathbb{B}}$$

$$\begin{array}{ll}
\mathsf{eq}(n, n) & = \mathtt{true} \\
\mathsf{eq}(b, b) & = \mathtt{true} \\
\mathsf{eq}(\mathtt{[}v_1, \ldots, v_n\mathtt{]}, \mathtt{[}v'_1, \ldots, v'_n\mathtt{]}) & = \mathsf{eq}(v_1, v'_1) \wedge \ldots \wedge \mathsf{eq}(v_n, v'_n) \\
\mathsf{eq}(\mathtt{(}v_1, \ldots, v_n\mathtt{)}, \mathtt{(}v'_1, \ldots, v'_n\mathtt{)}) & = \mathsf{eq}(v_1, v'_1) \wedge \ldots \wedge \mathsf{eq}(v_n, v'_n) \\
\mathsf{eq}(\mathsf{None}, \mathsf{None}) & = \mathtt{true} \\
\mathsf{eq}(\mathsf{Some}\ v, \mathsf{Some}\ v') & = \mathsf{eq}(v, v')
\end{array}$$

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 $\rightharpoonup$) for extending an environment with a pattern and a value only if the pattern matches the value.

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

$$\begin{array}{ll}
\mathsf{extend}(\sigma, n, n) & = \sigma \\
\mathsf{extend}(\sigma, b, b) & = \sigma \\
\mathsf{extend}(\sigma, x, v) & = \sigma[x \mapsto v] \\
\mathsf{extend}(\sigma, \mathsf{None}, \mathsf{None}) & = \sigma \\
\mathsf{extend}(\sigma, \mathsf{Some}\ \pi, \mathsf{Some}\ v) & = \mathsf{extend}(\sigma, \pi, v) \\
\mathsf{extend}(\sigma, \mathtt{[]}, \mathtt{[]}) & = \sigma \\
\mathsf{extend}(\sigma, \pi_1 :: \pi_2, \mathtt{[}v_1, \ldots, v_n\mathtt{]}) & = \mathsf{extend}(\mathsf{extend}(\sigma, \pi_1, v_1), \pi_2, \mathtt{[}v_2, \ldots, v_n\mathtt{]}) \\
\mathsf{extend}(\sigma, \mathtt{(}\pi_1, \ldots, \pi_n\mathtt{)}, \mathtt{(}v_1, \ldots, v_n\mathtt{)}) & = \sigma_n
\end{array}$$

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