Learning to program with F#

Jon Sporring

September 8, 2016

Part V

Appendix

Appendix D

F♭

Minimal F# used in Part I

Listing D.1: F♭, a subset of F#

```
(*Special characters*)
codePoint = ?Any unicode codepoint?;
Lu = ?Upper case letters?;
Ll = ?Lower case letters?;
Lt = ?Digraphic letters, with first part uppercase?;
Lm = ?Modifier letters?;
Lo = ?Gender ordinal indicators?;
Nl = ?Letterlike numeric characters?;
Pc = ?Low lines?;
Mn = ?Nonspacing combining marks?;
Mc = ?Spacing combining marks?;
Cf = ?Soft Hyphens?;

(*Whitespace*)
whitespace = " " {" "};
newline = "\n" | "\r" "\n";

(*Comments*)
blockComment = "(*" {codePoint} "*)";
lineComment = "//" {codePoint - newline} newline;

(*Literal digits*)
dDigit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
bDigit = "0" | "1";
oDigit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7";
xDigit =
  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
  | "A" | "B" | "C" | "D" | "E" | "F" | "a" | "b" | "c" | "d" | "e" | "f";

(*Literal integers*)
dInt = dDigit {dDigit};
bitInt = "0" ("b" | "B") bDigit {bDigit};
octInt = "0" ("o" | "O") oDigit {oDigit};
hexInt = "0" ("x" | "X") xDigit {xDigit};
xInt = bitInt | octInt | hexInt;

int = dInt | xInt;
```

```
sbyte = (dInt | xInt) "y";
byte = (dInt | xInt) "uy";
int32 = (dInt | xInt) ["l"];
uint32 = (dInt | xInt) ("u" | "ul");

(*Literal floats*)
float = dFloat | sFloat;
dFloat = dInt "." {dDigit};
sFloat = (dInt | dFloat) ("e" | "E" ) ["+" | "-"] dInt;
ieee64 = float | xInt "LF";

(*Literal chars*)
char = "'" codePoint | escapeChar "'";
escapeChar =
  "\" ("b" | "n" | "r" | "t" | "\" | '"' | "'" | "a" | "f" | "v")
  | "\u" xDigit xDigit xDigit xDigit
  | "\U" xDigit xDigit xDigit xDigit xDigit xDigit xDigit xDigit
  | "\" dDigit dDigit dDigit;

(*Literal strings*)
string = '"' { stringChar }  '"';
stringChar = char - '"';
verbatimString = '@"' {char - ('"' | '\"' )| '""'} '"';

(*Operators*)
infixOrPrefixOp = "+" | "-" | "+." | "-." | "%" | "&" | "&&";
prefixOp = infixOrPrefixOp | "~" {"~"} | "!" {opChar} - "!=";
infixOp =
  {"."} (
    infixOrPrefixOp
    | "-" {opChar}
    | "+" {opChar}
    | "||"
    | "<" {opChar}
    | ">" {opChar}
    | "="
    | " |" {opChar}
    | "&" {opChar}
    | "^" {opChar}
    | "*" {opChar}
    | "/" {opChar}
    | "%" {opChar}
    | "!=" )
  | ":=" | "::" | "$" | "?";
opChar =
  "!" | "%" | "&" | "*" | "+" | "-" | ". " | "/"
  | "<" | "=" | ">" | "@" | "^" | "|" | "~";

(*Expressions*)
expr =
  const (*a const value*)
  | "(" expr ")" (*block*)
  | longIdentOrOp (*identifier or operator*)
  | expr "." longIdentOrOp (*dot lookup expression, no space around "."*)
  | expr expr (*application*)
  | expr infixOp expr (*infix application*)
  | prefixOp expr (*prefix application*)
  | expr ".[" expr "]" (*index lookup, no space before "."*)
  | expr ".[" sliceRange "]" (*index lookup, no space before "."*)
```

```
  | expr "<-" expr (*assingment*)
  | exprTuple (*tuple*)
  | "[" (exprSeq | rangeExpr) "]" (*list*)
  | "[|" (exprSeq | rangeExpr) "|]" (*array*)
  | expr ":" type (*type annotation*)
  | expr ";" expr (*sequence of expressions*)
  | "let" valueDefn "in" expr (*binding a value or variable*)
  | "let" ["rec"] functionDefn "in" expr (*binding a function or operator*)
  | "fun" argumentPats "->" expr (*anonymous function*)
  | "if" expr "then" expr {"elif" expr "then" expr} ["else" expr] (*conditional*)
  | "while" expr "do" expr ["done"] (*while*)
  | "for" ident "=" expr "to" expr "do" expr ["done"] (* simple for expression *)
  | "try" expr "with"  ["|"] rules (*exception*)
  | "try" expr "finally" expr; (*exception with cleanup*)
exprTuple = expr | expr "," exprTuple;
exprSeq =  expr | expr ";" exprSeq;
rangeExpr = expr ".." expr [".." expr];
sliceRange =
  expr
  | expr ".." (*no space between expr and ".."*)
  | ".." expr (*no space between expr and ".."*)
  | expr ".." expr (*no space between expr and ".."*)
  | "*";

(*Constants*)
const =
  byte
  | sbyte
  | int32
  | uint32
  | int
  | ieee64
  | char
  | string
  | verbatimString
  | "false"
  | "true"
  | "()";

(*Identifiers*)
ident = (letter | "_") {letter | dDigit | specialChar};
letter = Lu | Ll | Lt | Lm | Lo | Nl; (*e.g. "A", "B" ... and "a", "b", ...*)
specialChar = Pc | Mn | Mc | Cf; (*e.g., "_"*)

longIdent = ident | ident "." longIdent; (*no space around "."*)
longIdentOrOp = [longIdent "."] identOrOp; (*no space around "."*)
identOrOp =
  ident
  | "(" infixOp | prefixOp ")"
  | "(*)";

(*Keywords*)
identKeyword =
  "abstract" | "and" | "as" | "assert" | "base" | "begin" | "class" | "default"
  | "delegate" | "do" | "done" | "downcast" | "downto" | "elif" | "else" | "end"
  | "exception" | "extern" | "false" | "finally" | "for" | "fun" | "function"
  | "global" | "if" | "in" | "inherit" | "inline" | "interface" | "internal"
  | "lazy" | "let" | "match" | "member" | "module" | "mutable"
  | "namespace" | "new" | "null" | "of" | "open" | "or" | "override" | "private"
```

```
  | "public" | "rec" | "return" | "sig" | "static" | "struct" | "then" | "to"
  | "true" | "try" | "type" | "upcast" | "use" | "val" | "void" | "when"
  | "while" | "with" | "yield";

reservedIdentKeyword =
  "atomic" | "break" | "checked" | "component" | "const" | "constraint"
  | "constructor" | "continue" | "eager" | "fixed" | "fori" | "functor"
  | "include" "measure" | "method" | "mixin" | "object" | "parallel"
  | "params" | "process" | "protected" | "pure" | "recursive" | "sealed"
  | "tailcall" | "trait" | "virtual" | "volatile";

reservedIdentFormats = ident ( "!" | "#");

(*Symbolic Keywords*)
symbolicKeyword =
  "let!" | "use!" | "do!" | "yield!" | "return!" | "|" | "->" | "<-" | "." | ":"
  | "(" | ")" | "[" | "]" | "[<" | ">]" | "[|" | "|]" | "{" | "}" | "'" | "#"
  | ":?>" | ":?" | ":>" | ".." | "::" | ":=" | ";;" | ";" | "=" | "_" | "?"
  | "??" | "(*)" | "<@" | "@>" | "<@@" | "@@>";

reservedSymbolicSequence =  "~" | "'";

(*Types*)
type =
  longIdent (*named such as "int"*)
  | "(" type ")" (*paranthesized*)
  | type "->" type (*function*)
  | typeTuple (*tuple*)
  | "'" ident (*variable, no space after "'"*)
  | type longIdent (*named such as "int list"*)
  | type "[" typeArray "]"; (*array, no spaces*)
typeTuple = type | type "*" typeTuple;
typeArray = "," | "," typeArray;

(*Value definition*)
valueDefn = ["mutable"] pat "=" expr;

(*Patterns*)
pat =
  const (*constant*)
  | "_" (*wildcard*)
  | ident (*named*)
  | pat "::" pat (*construction*)
  | pat ":" type (*type constraint*)
  | "(" pat ")" (*paranthesized*)
  | patTuple (*tuple*)
  | patList (*list*)
  | patArray (*array*)
  | ":?" type; (*dynamic type test*)
patTuple = pat | pat "," patTuple;
patList = "[" [patSeq] "]";
patArray = "[|" [patSeq] "|]";
patSeq = pat | pat ";" patSeq;

(*Function definition*)
functionDefn = identOrOp argumentPats [":" type] "=" expr;
argumentPats = pat | pat argumentPats;

(*Rules*)
```

```
rules = rule | rule "|" rules;
rule = pat ["when" expr] "->" expr;

(*script-file*)
moduleElems = moduleElem | moduleElem moduleElems;
moduleElem =
  "let" valueDefn "in" expr (*binding a value or variable*)
  | "let" ["rec"] functionDefn "in" expr (*binding a function or operator*)
  | "exception" ident of typeTuple (*exception definition*)
  | "open" longIdent (*import declaration*)
  | "#" ident string; (*compiler directive, no space after "#"*)
```

1

[1] Todo: **I don't think we need `type="'"ident` nor `moduleelm = "#"ident string`**