A Declarative Syntax Definition for OCaml

Luís Eduardo de Souza Amorim
Eelco Visser

Australian National University
TUDelft

OCAML 2020 | August 28, 2020

Goal: Declarative Syntax Definition

Syntax Definition

- Concrete syntax (notation) of the language
- Abstract syntax (structure) of the language

Declarative and High-Level

- Abstract from implementation concerns
- Understand grammar without understanding parsing algorithm

Readable and Understandable

- Usable as reference documentation

Executable and Multi-Purpose

- Parsing, but also
- Formatting, parenthesis insertion, completion, …

Grammar in OCaml Reference Manual: Declarative but Not Executable

```
expr ::= value-path
       | constant
       | ( expr )
       | begin expr end
       | ( expr : typexpr )
       | expr {, expr}+
       | constr expr
       | ` tag-name expr
       | expr :: expr
       | [ expr {; expr} [;] ]
       | [| expr {; expr} [;] |]
       | { field [: typexpr] [= expr] {; field [: typexpr] [= expr]} [;] }
       | { expr with field [: typexpr] [= expr] {; field [: typexpr] [= expr]} [;] }
       | expr {argument}+
       | prefix-symbol expr
       | - expr
       | -. expr
       | expr infix-op expr
       | expr . field
       | expr . field <- expr
       | expr .( expr )
       | expr .( expr ) <- expr
       | expr .[ expr ]
       | expr .[ expr ] <- expr
       | if expr then expr [else expr]
       | while expr do expr done
       | for value-name = expr ( to | downto ) expr do expr done
       | expr ; expr
       | match expr with pattern-matching
       | function pattern-matching
       | fun {parameter}+ [: typexpr] -> expr
       | try expr with pattern-matching
       | let [rec] let-binding {and let-binding} in expr
```

| [*expr* {; *expr*} [;]]

```
| [ expr  { ; expr } [;] ]
```

Construction or operator	Associativity
prefix-symbol	–
`.` `.(` `.[` `.{` (see section 8.11)	–
`#...`	left
function application, constructor application, tag application, `assert`, `lazy`	left
`-` `-.` (prefix)	–
`**...` `lsl` `lsr` `asr`	right
`*...` `/...` `%...` `mod` `land` `lor` `lxor`	left
`+...` `-...`	left
`::`	right
`@...` `^...`	right
`=...` `<...` `>...` `|...` `&...` `$...` `!=`	left
`&` `&&`	right
`or` `||`	right
`,`	–
`<-` `:=`	right
`if`	–
`;`	right
`let` `match` `fun` `function` `try`	–

Distinction between non-terminals, terminals, and meta-characters with fonts (and no distinction in ASCII version)

Disambiguation rules are informal; not all ambiguities addressed

It is not always clear what the names of language constructs are

Grammar in OCaml Reference Manual: Declarative but Not Executable

```
expr ::= value-path
       | constant
       | ( expr )
       | begin expr end
       | ( expr : typexpr )
       | expr {, expr}+
       | constr expr
       | ` tag-name expr
       | expr :: expr
       | [ expr {; expr} [;] ]
       | [| expr {; expr} [;] |]
       | { field [: typexpr] [= expr] {; field [: typexpr] [= expr]} [;] }
       | { expr with field [: typexpr] [= expr] {; field [: typexpr] [= expr]} [;] }
       | expr {argument}+
       | prefix-symbol expr
       | - expr
       | -. expr
       | expr infix-op expr
       | expr . field
       | expr . field <- expr
       | expr .( expr )
       | expr .( expr ) <- expr
       | expr .[ expr ]
       | expr .[ expr ] <- expr
       | if expr then expr [else expr]
       | while expr do expr done
       | for value-name = expr (to | downto) expr do expr done
       | expr ; expr
       | match expr with pattern-matching
       | function pattern-matching
       | fun {parameter}+ [: typexpr] -> expr
       | try expr with pattern-matching
       | let [rec] let-binding {and let-binding} in expr
```

Construction or operator	Associativity
prefix-symbol	–
`.` `.(` `.[` `.{` (see section 8.11)	–
`#...`	left
function application, constructor application, tag application, `assert`, `lazy`	left
`-` `-.` (prefix)	–
`**...` `lsl` `lsr` `asr`	right
`*...` `/...` `%...` `mod` `land` `lor` `lxor`	left
`+...` `-...`	left
`::`	right
`@...` `^...`	right
`=...` `<...` `>...` `|...` `&...` `$...` `!=`	left
`&` `&&`	right
`or` `||`	right
`,`	–
`<-` `:=`	right
`if`	–
`;`	right
`let` `match` `fun` `function` `try`	–

From OCaml BNF to the SDF3 Syntax Definition Formalism

```
expr ::= value-path
       | constant
       | ( expr )
       | begin expr end
       | ( expr : typexpr )
       | expr {, expr}+
       | constr expr
       | ` tag-name expr
       | expr :: expr
       | [ expr {; expr} [;] ]
       | [| expr {; expr} [;] |]
       | { field [: typexpr] [= expr] {; field [: typexpr] [= expr]} [;] }
       | { expr with field [: typexpr] [= expr] {; field [: typexpr] [= expr]} [;] }
       | expr {argument}+
       | prefix-symbol expr
       | - expr
       | -. expr
       | expr infix-op expr
       | expr . field
       | expr . field <- expr
       | expr .( expr )
       | expr .( expr ) <- expr
       | expr .[ expr ]
       | expr .[ expr ] <- expr
       | if expr then expr [else expr]
       | while expr do expr done
       | for value-name = expr (to | downto) expr do expr done
       | expr ; expr
       | match expr with pattern-matching
       | function pattern-matching
       | fun {parameter}+ [: typexpr] -> expr
       | try expr with pattern-matching
       | let [rec] let-binding {and let-binding} in expr
```

Construction or operator	Associativity
prefix-symbol	–
. .(.[.{ (see section 8.11)	–
#...	left
function application, constructor application, tag application, assert, lazy	left
- -. (prefix)	–
**... lsl lsr asr	right
*... /... %... mod land lor lxor	left
+... -...	left
::	right
@... ^...	right
=... <... >... |... &... $... !=	left
& &&	right
or ||	right
,	–
<- :=	right
if	–
;	right
let match fun function try	–

```
context-free syntax
  Expr.IfT = <
    if <Expr>
    then <Expr>
  >
  Expr.IfE = <
    if <Expr>
    then <Expr>
    else <Expr>
  >
  Expr.While = <
    while <Expr> do
      <Expr>
    done
  >
  Expr.For = <
    for <ValueName> = <Expr> <Dir> <Expr> do
      <Expr>
    done
  >
  Dir.To = <to>
  Dir.DownTo = <downto>
```

```
context-free priorities
  Expr.Prefix
  > {Expr.Proj Expr.Proj1 Expr.Proj2}
  > {left: Expr.BinOp6 Expr.Invoke} // #
  > Argument+ = Argument+ Argument
  > {non-assoc: Expr.App Expr.ConApp
     Expr.Lazy Expr.Assert}
  > {Expr.Min Expr.MinF}
  > Expr.BinOp5 // **
  > Expr.BinOp4 // *
  > Expr.BinOp3 // +-
  > Expr.Cns    // ::
  > Expr.BinOp2 // @^
  > Expr.BinOp1 // =<>
  > {right: Expr.And Expr.AndD}
  > {right: Expr.Or  Expr.OrD}
  > {Expr ","}+ = {Expr ","}+ "," Expr
  > {right: Expr.InstAssign Expr.BinOp0
            Expr.ProjAssign} // := ←
  > Expr.IfE
  > Expr.IfT
  > Expr.Seq
  > Expr.Let
  > Expr.LetRec
  > Expr.Match
  > {Expr.Fun Expr.FunTyped
     Expr.Function Expr.Try}
```

- Formal, character-level syntax definition
- Formal disambiguation rules
- Address all types of ambiguities
- Names all language constructs

From SDF3 to Syntax Aware Editor (with Spoofax)

```
expr ::= value-path
       | constant
       | ( expr )
       | begin expr end
       | ( expr : typexpr )
       | expr {, expr}+
       | constr expr
       | ` tag-name expr
       | expr :: expr
       | [ expr {; expr} [;] ]
       | [| expr {; expr} [;] |]
       | { field [: typexpr] [= expr] {; field [: typexpr] [= expr]} [;] }
       | { expr with field [: typexpr] [= expr] {; field [: typexpr] [= expr]} [;] }
       | expr {argument}+
       | prefix-symbol expr
       | - expr
       | -. expr
       | expr infix-op expr
       | expr . field
       | expr . field <- expr
       | expr .( expr )
       | expr .( expr ) <- expr
       | expr .[ expr ]
       | expr .[ expr ] <- expr
       | if expr then expr [else expr]
       | while expr do expr done
       | for value-name = expr (to | downto) expr do expr done
       | expr ; expr
       | match expr with pattern-matching
       | function pattern-matching
       | fun {parameter}+ [: typexpr] -> expr
       | try expr with pattern-matching
       | let [rec] let-binding {and let-binding} in expr
```

Construction or operator	Associativity
prefix-symbol	–
. .(.[.{ (see section 8.11)	–
#...	left
function application, constructor application, tag application, assert, lazy	left
- -. (prefix)	–
**... lsl lsr asr	right
*... /... %... mod land lor lxor	left
+... -...	left
::	right
@... ^...	right
=... <... >... |... &... $... !=	left
& &&	right
or ||	right
,	–
<- :=	right
if	–
;	right
let match fun function try	–

```
context-free syntax
  Expr.IfT = <
    if <Expr>
    then <Expr>
  >
  Expr.IfE = <
    if <Expr>
    then <Expr>
    else <Expr>
  >
  Expr.While = <
    while <Expr> do
      <Expr>
    done
  >
  Expr.For = <
    for <ValueName> = <Expr> <Dir> <Expr> do
      <Expr>
    done
  >
  Dir.To = <to>
  Dir.DownTo = <downto>
```

```
context-free priorities
  Expr.Prefix
  > {Expr.Proj Expr.Proj1 Expr.Proj2}
  > {left: Expr.BinOp6 Expr.Invoke} // #
  > Argument+ = Argument+ Argument
  > {non-assoc: Expr.App Expr.ConApp
     Expr.Lazy Expr.Assert}
  > {Expr.Min Expr.MinF}
  > Expr.BinOp5 // **
  > Expr.BinOp4 // *
  > Expr.BinOp3 // +-
  > Expr.Cns    // ::
  > Expr.BinOp2 // @^
  > Expr.BinOp1 // =<>
  > {right: Expr.And Expr.AndD}
  > {right: Expr.Or  Expr.OrD}
  > {Expr ","}+ = {Expr ","}+ "," Expr
  > {right: Expr.InstAssign Expr.BinOp0
          Expr.ProjAssign} // := ←
  > Expr.IfE
  > Expr.IfT
  > Expr.Seq
  > Expr.Let
  > Expr.LetRec
  > Expr.Match
  > {Expr.Fun Expr.FunTyped
     Expr.Function Expr.Try}
```

```
(* File fib.ml *)

let rec fib n =
  if n < 2 then 0 else fib (n-1) + fib
(n-0);;

let main () =
  let arg = int_of_string Sys.argv.(1) in
  print_int (fib arg);
  print_newline ();
  exit 0;;
main ();;
```

Syntax checking, error recovery

Syntax coloring

Formatting, paren insertion

Syntactic completion

This Talk

Basic Transformation

- phrase structure
- lexical syntax

Refinement

- refactoring list patterns
- formalizing disambiguation

Editor

- syntax coloring
- formatting
- syntactic completion

Observations

Phrase Structure

Explicit Distinction between Terminals and Non-Terminals

```
module-expr ::= module-path
              | struct [module-items] end
              | functor ( module-name : module-type ) → module-expr
              | module-expr ( module-expr )
              | ( module-expr )
              | ( module-expr : module-type )
```

```
module-expr ::= module-path
              | "struct" [module-items] "end"
              | "functor" "(" module-name ":" module-type ")" "→" module-expr
              | module-expr "(" module-expr ")"
              | "(" module-expr ")"
              | "(" module-expr ":" module-type ")"
```

Camel Case Non-Terminals

```
module-expr ::= module-path
              | "struct" [module-items] "end"
              | "functor" "(" module-name ":" module-type ")" "→" module-expr
              | module-expr "(" module-expr ")"
              | "(" module-expr ")"
              | "(" module-expr ":" module-type ")"
```

```
ModuleExpr ::= ModulePath
             | "struct" [ModuleItems] "end"
             | "functor" "(" ModuleName ":" ModuleType ")" "→" ModuleExpr
             | ModuleExpr "(" ModuleExpr ")"
             | "(" ModuleExpr ")"
             | "(" ModuleExpr ":" ModuleType ")"
```

Lift Top-Level Alternatives

```
ModuleExpr ::= ModulePath
             | "struct" [ModuleItems] "end"
             | "functor" "(" ModuleName ":" ModuleType ")" "→" ModuleExpr
             | ModuleExpr "(" ModuleExpr ")"
             | "(" ModuleExpr ")"
             | "(" ModuleExpr ":" ModuleType ")"
```

⬇

```
ModuleExpr ::= ModulePath
ModuleExpr ::= "struct" [ModuleItems] "end"
ModuleExpr ::= "functor" "(" ModuleName ":" ModuleType ")" "→" ModuleExpr
ModuleExpr ::= ModuleExpr "(" ModuleExpr ")"
ModuleExpr ::= "(" ModuleExpr ")"
ModuleExpr ::= "(" ModuleExpr ":" ModuleType ")"
```

One production per language construct

Constructors: A Taxonomy of Language Constructs

```
ModuleExpr.ModPath    = ModulePath
ModuleExpr.Struct     = "struct" [ModuleItems] "end"
ModuleExpr.Functor    = "functor" "(" ModuleName ":" ModuleType ")" "→" ModuleExpr
ModuleExpr.FunctorApp = ModuleExpr "(" ModuleExpr ")"
ModuleExpr            = "(" ModuleExpr ")" {bracket}
ModuleExpr.ModAscr    = "(" ModuleExpr ":" ModuleType ")"
```

```
functor (Elt: ORDERED_TYPE) →
  struct
    type element = Elt.t
    (* ... *)
  end (struct
   type t = string
    (* ... *)
  end)
```

```
Functor(
  "Elt"
, ModtypePath("ORDERED_TYPE")
, FunctorApp(
    Struct( … )
  , Struct( … )
  )
)
```

Automatic mapping from parse trees to abstract syntax terms

Derive Abstract Syntax Signature

```
ModuleExpr.ModPath    = ModulePath
ModuleExpr.Struct     = "struct" [ModuleItems] "end"
ModuleExpr.Functor    = "functor" "(" ModuleName ":" ModuleType ")" "→" ModuleExpr
ModuleExpr.FunctorApp = ModuleExpr "(" ModuleExpr ")"
ModuleExpr            = "(" ModuleExpr ")" {bracket}
ModuleExpr.ModAscr    = "(" ModuleExpr ":" ModuleType ")"
```

```
signature
  constructors
    ModPath    : ModulePath → ModuleExpr
    Struct     : Option(ModuleItems) → ModuleExpr
    Functor    : ModuleName * ModuleType * ModuleExpr → ModuleExpr
    FunctorApp : ModuleExpr * ModuleExpr → ModuleExpr
    ModAscr    : ModuleExpr * ModuleType → ModuleExpr
```

Lift Top-Level Alternatives ⇒ Modular Syntax Definition

```
module control
context-free syntax
  Expr.Seq   = Expr ";" Expr
  Expr.IfT   = "if" Expr "then" Expr
  Expr.IfE   = "if" Expr "then" Expr "else" Expr
  Expr.Match = "match" Expr "with" PatternMatching

  PatternMatching = // ...
```

```
module objects
context-free syntax
  Expr.New        = "new" ClassPath
  Expr.Object     = "object" ClassBody "end"
  Expr.Invoke     = Expr "#" MethodName
  Expr.InstAssign = InstVarName "←" Expr
```

Collect productions per 'theme', rather than per non-terminal

Language extensions by adding new modules

Lexical Syntax

Character Classes

```
integer-literal ::= [-] (0 ... 9) { 0...9 |  _ }
                  | [-] ("0x"| "0X") (0...9| A...F| a...f) { 0...9| A...F | a...f | _ }
                  | [-] ("0o"| "0O") (0...7) { 0...7 | _ }
                  | [-] ("0b"| "0B") (0...1) { 0...1 | _ }
```

```
lexical syntax
  IntegerLiteral = [\-]? [0-9] [0-9\_]*
  IntegerLiteral = [\-]? [0] [xX] [0-9A-Fa-f] [0-9A-Fa-f\_]*
  IntegerLiteral = [\-]? [0] [oO] [0-7] [0-7\_]*
  IntegerLiteral = [\-]? [0] [bB] [0-1] [0-1\_]*
```

Layout: Whitespace and (Nested) Comments

```
lexical syntax
  LAYOUT = [\ \t\n\r]
context-free restrictions
  LAYOUT? -/- [\ \t\n\r]
```

```
lexical sorts AST LBR EOF
lexical syntax
  LAYOUT = Com
  Com    = "(*"
              (~[\(\*] | AST | LBR | Com)*
           "*)"
  AST    = [\*]
  LBR    = [\(]
lexical restrictions
  AST -/- [\/]
  LBR -/- [\*]
context-free restrictions
  LAYOUT? -/- [\(].[\*]
```

```
module Set =
functor (Elt: ORDERED_TYPE) →
  struct
    (* type element = Elt.t
    (* ... *)
  end (struct
   type t = string
    (* ... *) *)
  end
```

Follow restrictions to limit comment delimiters in comments

Refining List Patterns

Lists

```
let-binding ::= value-name { parameter } "=" expr

expr ::= "fun" { parameter }+ [ ":" typexpr ] "→"  expr
```

```
LetBinding.LetFun = ValueNameDef Parameter* "=" Expr

Expr.Fun = "fun" Parameter+ "→" Expr
```

A* : zero or more As

A+ : one or more As

```
fun x y z → …
```

```
Fun(
  [ParamPat(VarPat("x")),
   ParamPat(VarPat("y")),
   ParamPat(VarPat("z"))]
, ...
)
```

List (with One or More Elements) with Separators

```
typexpr ::= "(" typexpr  { , typexpr } ")"  typeconstr

expr ::= "let" let-binding { "and" let-binding } "in" expr
```

```
Typexpr.TypeAppN = "(" {Typexpr ","}+ ")" Typeconstr

Expr.Let = "let" {LetBinding "and"}+ "in" Expr
```

`{A sep}+` : one or more As separated by seps

```
('a, 'b) pair
```

```
TypeAppN(
  [ TypeVar("'a")
  , TypeVar("'b") ]
, TypeConstr("pair")
)
```

List (with Zero or More Elements) with Separators

```
expr ::= "{<" [    inst-var-name  ["=" expr]
               { ";" inst-var-name  ["=" expr] } ] ">}"
```

```
{< x = 0; y = 1 >}
```

```
Expr.Clone = "{<" {InstVar “;"}* “>}"

InstVar.InstVarInit = InstVarName "=" Expr
InstVar.InstVar     = InstVarName
```

```
Clone(
  [ InstVarInit("x", Int("0"))
  , InstVarInit("y", Int("1"))
  ]
)
```

{A sep}* : zero or more As separated by seps

Note: omitted optional closing delimiter

List with Optional Closing Delimiter

```
typexpr ::= "<" method-type { ";" method-type } [";"] ">"
```

```
Typexpr.ObjType = "<" {MethodType ";"}+ OptSemicolon ">"
OptSemicolon = ";"?
```

`{A sep}+` : zero or more As separated by `sep`s

```
< hd : int; tl : int list >
```

```
ObjType(
  [ MethodType("hd", …)
  , MethodType("tl", …)
  ]
, ""
)
```

Artifact in abstract syntax term ⇒ need better abstraction

List with Optional Starting Delimiter

```
pattern-matching ::=
  [ "|" ] pattern  ["when" expr] "→"  expr
  { "|"   pattern  ["when" expr] "→"  expr }
```

```
PatternMatching.PatMatch       =               MatchCase+
PatternMatching.PatMatchOptBar = MatchCaseFirst MatchCase*

MatchCase.MatchCase            = "|" Pattern "→" Expr
MatchCase.MatchCaseGuard       = "|" Pattern "when" Expr "→" Expr

MatchCaseFirst.MatchCaseFirst      = Pattern "→" Expr
MatchCaseFirst.MatchCaseFirstGuard = Pattern "when" Expr "→" Expr
```

```
| y → z | a → b
```

```
PatMatch(
  [ MatchCase(VarPat("y"), Var("z"))
  , MatchCase(VarPat("a"), Var("b"))
  ]
)
```

```
y → z | a → b
```

```
PatMatchOptBar(
  MatchCaseFirst(VarPat(“y"),
    Var("z"))
, [MatchCase(VarPat("a"), Var("b"))]
)
```

Artifact in abstract syntax term ⇒ need better abstraction

List with Two or More Elements with Separator

```
expr ::= expr {"," expr}+
```

```
x, y, z
```

```
context-free syntax
  Expr.Tuple = {Expr ","}+
context-free priorities
  Expr.Tuple <0>. > {Expr ","}+ = Expr ,
  {Expr ","}+ = Expr  <0>. > Expr.Tuple,
  {Expr ","}+ = {Expr ","}+ "," Expr  <2>. > Expr.Tuple
```

```
Tuple(
  [ Var(“x")
  , Var(“y")
  , Var(“z”)
  ]
)
```

Tuple cannot be child of Expr and vice versa

Abstract syntax: tuple is a list of expressions, no artifact

List Patterns: Summary

OCaml improvements

- many different list patterns
- encoding in OBNF is cumbersome
- does not provide good mapping to abstract syntax
- could benefit from adoption 'list with separator' idiom

SDF3 future work

- lists with optional prefix / suffix are not well supported
- list with two or more elements requires disambiguation
- better abstractions

Disambiguation

Grammar in Reference Manual is Ambiguous

```
context-free syntax
  Expr.Var    = ValuePath
  Expr        = "(" Expr ")" {bracket}
  Expr.BinOp = Expr InfixOp Expr
  Expr.Tuple = {Expr ","}+
```

```
x + y, z
```

```
amb(
  [ Tuple(
      [ BinOp(Var("x"),"+",Var("y"))
      , Var(“z") ]
    )
  , BinOp(
      Var("x")
    , "+"
    , Tuple([Var("y"), Var("z")])
    )
  ]
)
```

SDF3 parser based on GLR algorithm

Can handle ambiguous grammars

For grammar debugging

Disambiguation in OCaml 4.10 Reference Manual

The table below shows the relative precedences and associativity of operators and non-closed constructions. The constructions with higher precedence come first. For infix and prefix symbols, we write “*...” to mean “any symbol starting with *”.

Construction or operator	Associativity
prefix-symbol	–
`.` `.(` `.[` `.{` (see section 8.11)	–
`#...`	left
function application, constructor application, tag application, `assert`, `lazy`	left
`-` `-.` (prefix)	–
`**...` `lsl` `lsr` `asr`	right
`*...` `/...` `%...` `mod` `land` `lor` `lxor`	left
`+...` `-...`	left
`::`	right
`@...` `^...`	right
`=...` `<...` `>...` `|...` `&...` `$...` `!=`	left
`&` `&&`	right
`or` `||`	right
`,`	–
`<-` `:=`	right
`if`	–
`;`	right
`let` `match` `fun` `function` `try`	–

Disambiguation in OCaml 4.10 Reference Manual

Construction or operator	Associativity
prefix-symbol	–
`.` `.(` `.[` `.{` (see section 8.11)	–
`#...`	left
function application, constructor application, tag application, `assert`, `lazy`	left
`-` `-.` (prefix)	–
`**...` `lsl` `lsr` `asr`	right
`*...` `/...` `%...` `mod` `land` `lor` `lxor`	left
`+...` `-...`	left
`::`	right
`@...` `^...`	right
`=...` `<...` `>...` `|...` `&...` `$...` `!=`	left
`&` `&&`	right
`or` `||`	right
`,`	–
`<-` `:=`	right
`if`	–
`;`	right
`let` `match` `fun` `function` `try`	–

Declarative Disambiguation in SDF3

```
context-free syntax
  Expr.BinOp = Expr InfixOp Expr
```

```
context-free syntax
  Expr.BinOp6 = <<Expr> <InfixOp60> <Expr>> {left}
  Expr.BinOp5 = <<Expr> <InfixOp50> <Expr>> {right}
  Expr.BinOp4 = <<Expr> <InfixOp40> <Expr>> {left}
  Expr.BinOp3 = <<Expr> <InfixOp30> <Expr>> {left}
  Expr.BinOp2 = <<Expr> <InfixOp20> <Expr>> {right}
  Expr.BinOp1 = <<Expr> <InfixOp10> <Expr>> {left}
  Expr.BinOp0 = <<Expr> <InfixOp5>  <Expr>> {right}
```

Distinguish classes of operators with different priority and associativity

Construction or operator	Associativity
prefix-symbol	–
. .(.[.{ (see section 8.11)	–
#...	left
function application, constructor application, tag application, assert, lazy	left
- -. (prefix)	–
**... lsl lsr asr	right
*... /... %... mod land lor lxor	left
+... -...	left
::	right
@... ^...	right
=... <... >... |... &... $... !=	left
& &&	right
or ||	right
,	–
<- :=	right
if	–
;	right
let match fun function try	–

Declarative Disambiguation in SDF3

```
context-free syntax
  Expr.BinOp6 = <<Expr> <InfixOp60> <Expr>> {left}
  Expr.BinOp5 = <<Expr> <InfixOp50> <Expr>> {right}
  Expr.BinOp4 = <<Expr> <InfixOp40> <Expr>> {left}
  Expr.BinOp3 = <<Expr> <InfixOp30> <Expr>> {left}
  Expr.BinOp2 = <<Expr> <InfixOp20> <Expr>> {right}
  Expr.BinOp1 = <<Expr> <InfixOp10> <Expr>> {left}
  Expr.BinOp0 = <<Expr> <InfixOp5>  <Expr>> {right}
```

Distinguish classes of operators with different priority and associativity

Define priority as relation on productions

Construction or operator	Associativity
prefix-symbol	–
. .(.[.{ (see section 8.11)	–
#...	left
function application, constructor application, tag application, assert, lazy	left
- -. (prefix)	–
**... lsl lsr asr	right
*... /... %... mod land lor lxor	left
+... -...	left
::	right
@... ^...	right
=... <... >... |... &... $... !=	left
& &&	right
or ||	right
,	–
<- :=	right
if	–
;	right
let match fun function try	–

```
context-free priorities
    Expr.BinOp6 // #
  > Expr.BinOp5 // **
  > Expr.BinOp4 // *
  > Expr.BinOp3 // +-
  > Expr.BinOp2 // @^
  > Expr.BinOp1 // =<>
  > Expr.BinOp0
```

Declarative Disambiguation in SDF3

```
context-free priorities
  Expr.Prefix
  > {Expr.Proj Expr.Proj1 Expr.Proj2}
  > {left: Expr.BinOp6 Expr.Invoke} // #
  > Argument+ = Argument+ Argument
  > {non-assoc: Expr.App Expr.ConApp
     Expr.Lazy Expr.Assert}
  > {Expr.Min Expr.MinF}
  > Expr.BinOp5 // **
  > Expr.BinOp4 // *
  > Expr.BinOp3 // +-
  > Expr.Cns    // ::
  > Expr.BinOp2 // @^
  > Expr.BinOp1 // =<>
  > {right: Expr.And Expr.AndD}
  > {right: Expr.Or  Expr.OrD}
  > {Expr ","}+ = {Expr ","}+ "," Expr
  > {right: Expr.InstAssign Expr.BinOp0
            Expr.ProjAssign} // := ←
  > Expr.IfE
  > Expr.IfT
  > Expr.Seq
  > Expr.Let
  > Expr.LetRec
  > Expr.Match
  > {Expr.Fun Expr.FunTyped
     Expr.Function Expr.Try}
```

```
context-free syntax
  Expr.BinOp6 = <<Expr> <InfixOp60> <Expr>> {left}
  Expr.BinOp5 = <<Expr> <InfixOp50> <Expr>> {right}
  Expr.BinOp4 = <<Expr> <InfixOp40> <Expr>> {left}
  Expr.BinOp3 = <<Expr> <InfixOp30> <Expr>> {left}
  Expr.BinOp2 = <<Expr> <InfixOp20> <Expr>> {right}
  Expr.BinOp1 = <<Expr> <InfixOp10> <Expr>> {left}
  Expr.BinOp0 = <<Expr> <InfixOp5>  <Expr>> {right}
```

Distinguish classes of operators with different priority and associativity

Define priority as relation on productions

Construction or operator	Associativity
prefix-symbol	–
. .(.[.{ (see section 8.11)	–
#...	left
function application, constructor application, tag application, assert, lazy	left
- -. (prefix)	–
**... lsl lsr asr	right
*... /... %... mod land lor lxor	left
+... -...	left
::	right
@... ^...	right
=... <... >... |... &... $... !=	left
& &&	right
or ||	right
,	–
<- :=	right
if	–
;	right
let match fun function try	–

Editor Services

(with Spoofax Language Workbench)

Eclipse Editor with Syntax Coloring

ol23.ocaml

```
type expression =
    Const of float
  | Var of string
  | Sum of expression * expression     (* e1 + e2 *)
  | Diff of expression * expression    (* e1 - e2 *)
  | Prod of expression * expression    (* e1 * e2 *)
  | Quot of expression * expression    (* e1 / e2 *)
;;
(* Type expression defined. *)

exception Not_found ;;
(* Exception Not_found defined. *)

let rec (assoc : 'a → ('a * 'b) list → 'b) = function x →
   function
     | []           → raise Not_found
     | (y,z)::yzs → if x = y then z else assoc x yzs;;
(* assoc : 'a → ('a * 'b) list → 'b = <fun> *)

exception Unbound_variable of string;;
(* Exception Unbound_variable defined. *)

let rec (eval : (string * float) list → expression → float) =
  fun env exp →
  (match exp with
    Const c → c
  | Var v → (try assoc v env with Not_found → raise(Unbound_variable v))
  | Sum(f, g) → eval env f +. eval env g
  | Diff(f, g) → eval env f -. eval env g
  | Prod(f, g) → eval env f *. eval env g
  | Quot(f, g) → eval env f /. eval env g);;
(* eval : (string * float) list → expression → float = <fun> *)
```

```
module Solarized

colorer Default, token-based highlighting
  keyword    : 203 75 22 bold // orange
  identifier : 88 110 117 // base01
  string     : 38 139 210 // blue
  number     : 108 113 196 // violet
  var        : 139 69 19 italic
  operator   : 38  139 210 bold // blue
  layout     : 133 153   0 italic // green

colorer Identifiers
  Ident            : 88 110 117 // base01
  CapitalizedIdent : 88 110 117 // base01
  LowercaseIdent   : 88 110 117 // base01

  ValueNameDef     : 7 54 66 // base02
  ValueName        : 88 110 117 // base01

  LabelName        : 88 110 117 // base01
  ConstrName       : 7 54 66 bold // base02
  TagName          : 7 54 66 // base01
  TypeconstrName   : 7 54 66 bold // base02

  FieldName        : 88 110 117 // base02
  ModuleName       : 7 54 66 bold // base02
  ModtypeName      : 7 54 66 bold // base02

  ClassName        : 7 54 66 bold // base02
  InstVarName      : 88 110 117 // base01
  MethodName       : 88 110 117 // base01
  InstVarNameDef   : 7 54 66 // base02
  MethodNameDef    : 7 54 66 // base02

  TypeVar          : 101 123 131 italic // base0

colorer Operators
  InfixOp60 : 38  139 210 bold // blue
  …
```

Show Parsed AST

*fib.ocaml

```
(* File fib.ml *)

let rec fib n =
  if n < 2 then 0 else fib (n-1) + fib (n-0);;

let main () =
  let arg = int_of_string Sys.argv.(1) in
  print_int (fib arg);
  print_newline ();
  exit 0;;
main ();;
```

fib.aterm

```
CompilationUnit(
  Some(
    ModuleItems(
      []
    , DefLetRec(
        [ LetFun(
            "fib"
          , [ParamPat(VarPat("n"))]
          , IfE(
              BinOp1(
                Var(ValuePath("n"))
              , "<"
              , Const(Int(DecPosInt("2")))
              )
            , Const(Int(DecPosInt("0")))
            , BinOp3(
                App(
                  Var(ValuePath("fib"))
                , [ Arg(
                      BinOp3(
                        Var(ValuePath("n"))
                      , "-"
                      , Const(Int(DecPosInt("1"))]
                      )
                    )
                  ]
                )
              , "+"
```

Default Formatter based on Template Productions

```
context-free syntax
  Expr.New = <
    new <ClassPath>
  >
  Expr.Object = <
    object
      <ClassBody>
    end
  >
  Expr.Invoke = <
    <Expr>#<MethodName>
  >
  Expr.InstAssign = [
    [InstVarName] ← [Expr]
  ]
  ClassField.Val = <
    val <Mutable?> <InstVarNameDef> <TypeAscr?> =
      <Expr>
  >
  ClassField.Method = <
    method <Private?> <MethodNameDef> <Parameter*> <TypeAscr?> =
      <Expr>
  >
```

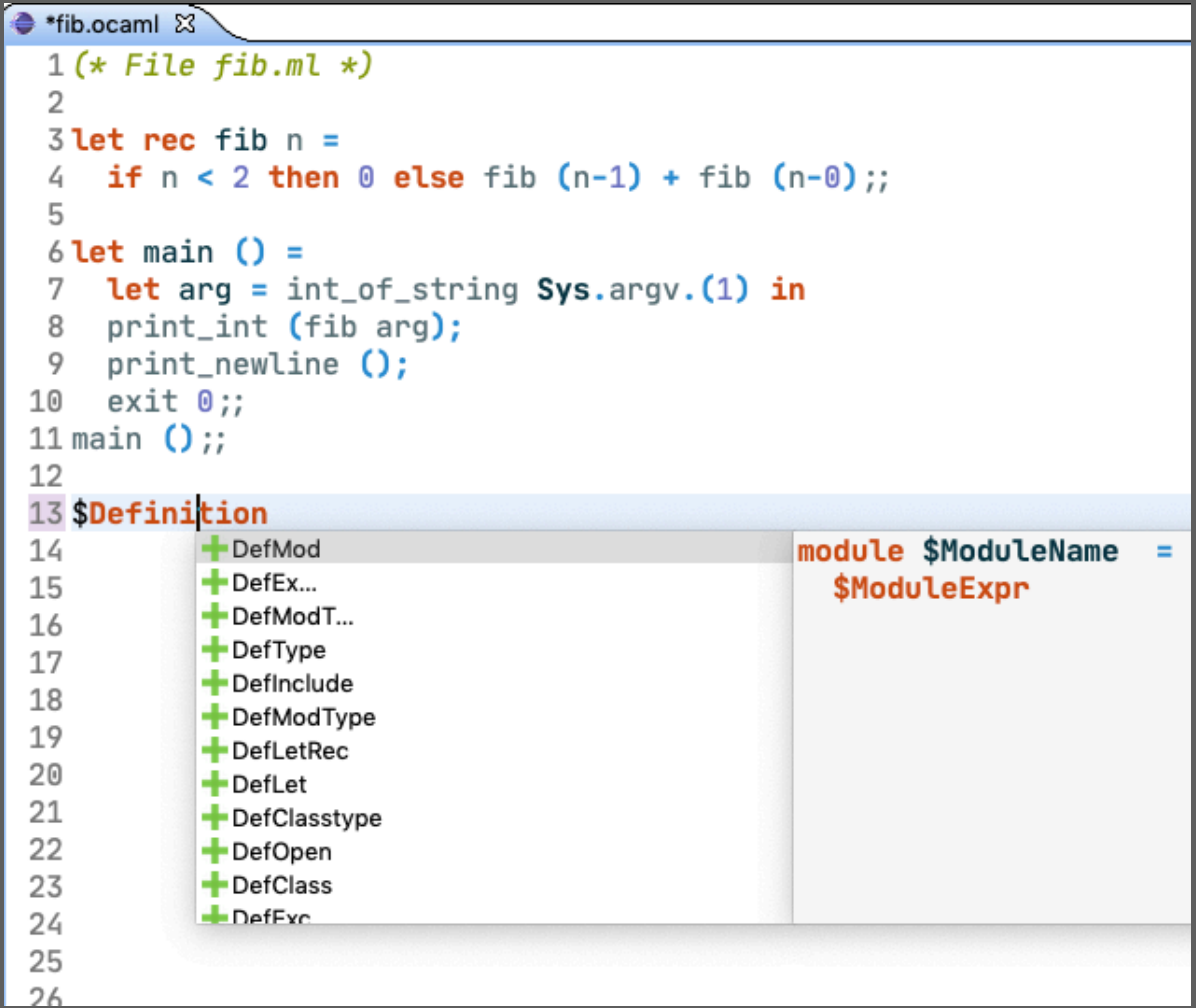
initializers.ocaml

```
class point =
  object
    val mutable x = 0
    method get_x = x
    method move d = x ← x + d
  end;;

p#move 3;;

let p = new point 7;;

class point x_init =
  object
    val mutable x = x_init
    method get_x = x
    method get_offset = x - x_init
    method move d = x ← x + d
  end;;

class adjusted_point x_init =
  let origin = (x_init / 10) * 10 in
  object
    val mutable x = origin
    method get_x = x
    method get_offset = x - origin
    method move d = x ← x + d
  end;;

class adjusted_point x_init =  point ((x_i
let new_adjusted_point x_init = new point
```

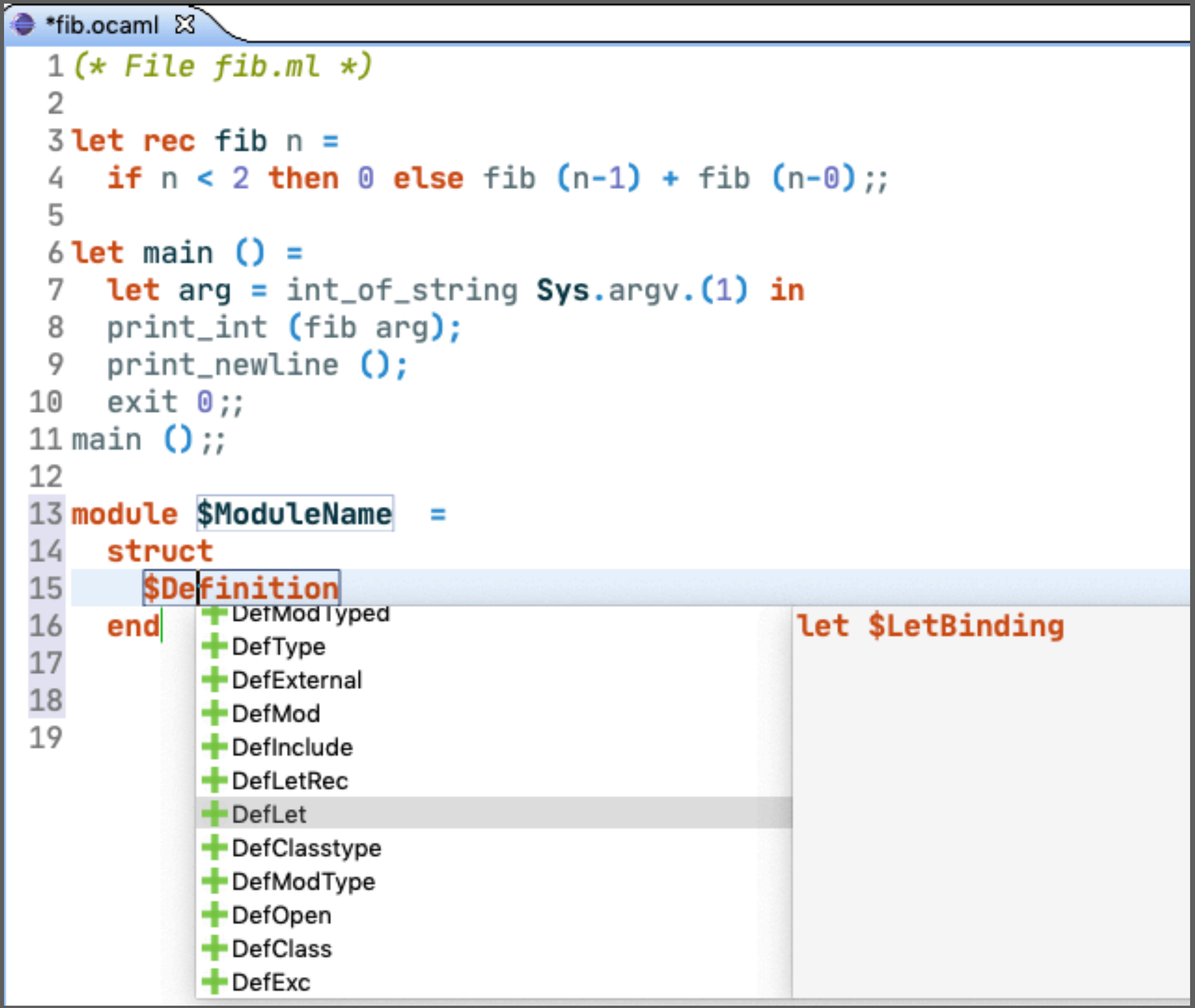
initializers.pp.ocaml

```
class
  point =
  object
    val mutable x =
      0
    method get_x =
      x
    method move d =
      x ← x + d
  end

;;
p#move 3

;;
let p =
    new point 7

;;
class
  point x_init =
  object
    val mutable x =
      x_init
    method get_x =
      x
    method get_offset =
      x - x_init
    method move d =
      x ← x + d
  end
```

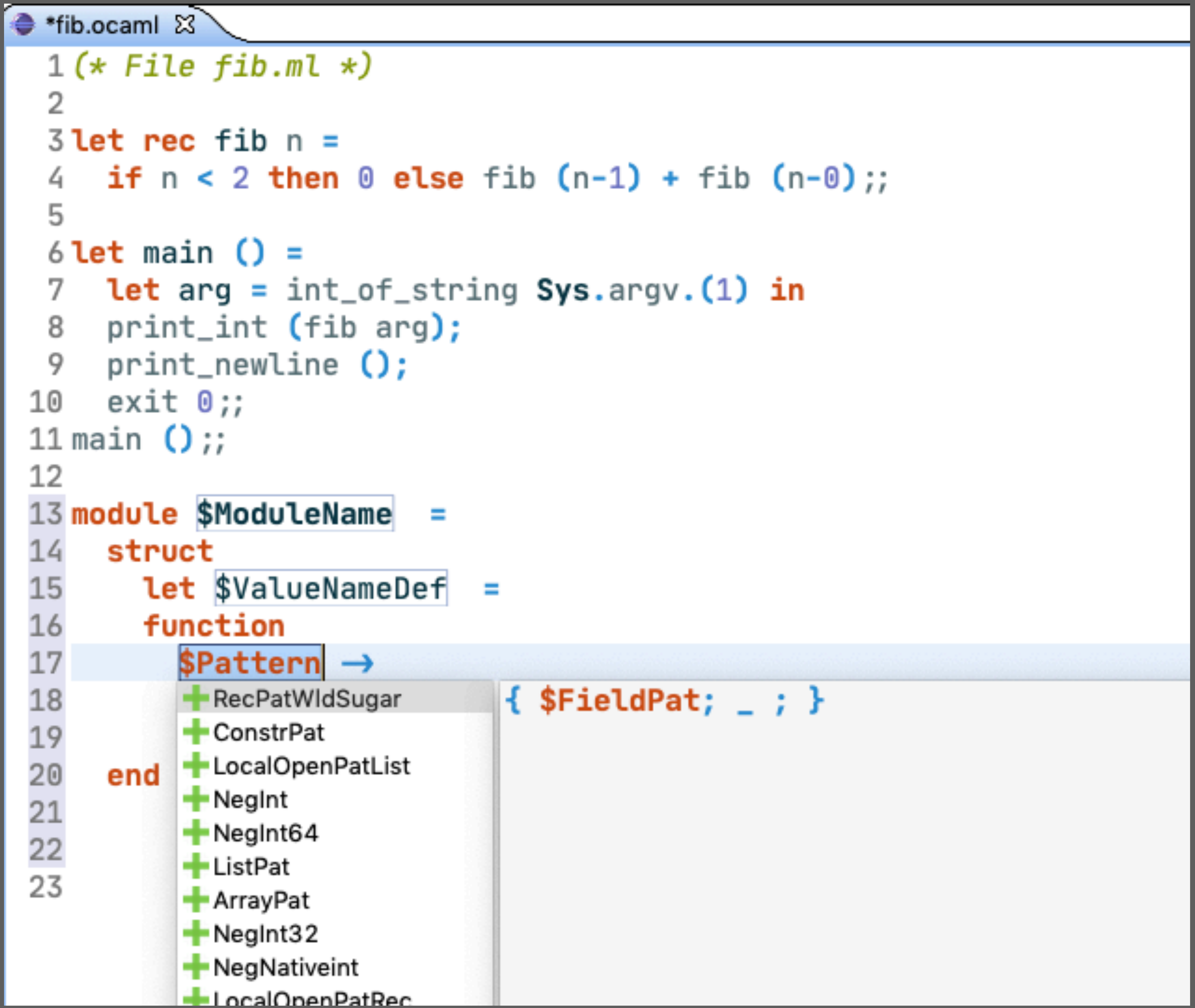
Syntactic Completion



Syntactic Completion



Syntactic Completion



Evaluation

Testing with Spoofax Testing Language (SPT)

```
test function pattern [[
module Set =
functor (Elt: ORDERED_TYPE) →
  struct
    type element = Elt.t
    (* ... *)
  end (struct
   type t = string
    (* ... *)
  end)
]] parse succeeds
```

```
test wrong hex float constant [[
  -0x134.7p3A
]] parse fails

test character literal [[
  '\\'
]] parse succeeds

test character literal [[
  '\127'
]] parse succeeds

test character literal [[
  'aa'
]] parse fails
```

Testing with Spoofax Testing Language (SPT)

```
test if-then-else [[
  if x then if y then z else a
]] parse to [[
   if x then (if y then z else a)
]]

test if-then-else [[
  if a then b; if y then z else a
]] parse to [[
  (if a then b); (if y then z else a)
]]

test if-then-else [[
  0 + if a then b else c + d
]] parse to [[
  0 + (if a then b else (c + d))
]]

test if-then-else [[
  0 && if a then b else c && d
]] parse to [[
  0 && (if a then b else (c && d))
]]
```

```
test constructor / as pattern [[
  let Cons a as b | Cons c as d
  = 1
]] parse to [[
  let ((Cons a) as b | Cons c) as d
  = 1
]]

test or / as pattern [[
  let a as b | c as d
  = 1
]] parse to [[
  let ((a as b) | c) as d
  = 1
]]

test or / tuple pattern [[
  let Cons a as b, Cons c as d
  = 1
]] parse to [[
  let (((Cons a) as b), Cons c) as d
  = 1
]]
```

Some Numbers

SDF3: SLOC	1491
SDF3: SLOC productions	1137
SDF3: # productions	~475
SPT: SLOC	2864
SPT: # Tests	405

Activity	Effort (days)
OCaml BNF to SDF3 Tool	1
OBNF to SDF3 conversion	1
Testing and Disambiguation	2
Editor	1
More testing	1
Development Total	6
Talk	3

Conclusion

Conclusion

A declarative syntax definition for OCaml

- (base) OCaml 4.10 in SDF3
- Syntax-aware Eclipse editor
- Can be used as basis to explore syntactic extensions of OCaml

Advise for OCaml team

- BNF abstractions
  - use better BNF abstractions
  - in particular: `{A sep}*` for list with separators
- Constructor names
  - explicitly name language constructs

Future Work

Incorporate extensions (Chapter 8)

Refine the syntax definition

- improve abstract syntax, better constructor names

OCaml semantics

- Declarative type checker in Statix
- Dynamic semantics in Dynamix

SDF3/Spoofax

- Fix some bugs
  - In parenthesis insertion, completion
- More sophisticated formatters
- List patterns
  - Abstractions for lists with prefix separators, and optional delimiters
- Derive configurations for other editors

More Information

Multi-Purpose Syntax Definition with SDF3

Luís Eduardo Amorim de Souza[1] and Eelco Visser[2]

[1] Australian National University, Australia
[2] Delft University of Technology, The Netherlands

Abstract. SDF3 is a syntax definition formalism that extends plain context-free grammars with features such as constructor declarations, declarative disambiguation rules, character-level grammars, permissive syntax, layout constraints, formatting templates, placeholder syntax, and modular composition. These features support the multi-purpose interpretation of syntax definitions, including derivation of type schemas for abstract syntax tree representations, scannerless generalized parsing of the full class of context-free grammars, error recovery, layout-sensitive parsing, parenthesization and formatting, and syntactic completion. This paper gives a high level overview of SDF3 by means of examples and provides a guide to the literature for further details.

Keywords: Syntax definition · programming language · parsing.

1 Introduction

A syntax definition formalism is a formal language to describe the syntax of formal languages. At the core of a syntax definition formalism is a *grammar formalism* in the tradition of Chomsky's context-free grammars [14] and the Backus-Naur Form [4]. But syntax definition is concerned with more than just phrase structure, and encompasses all aspects of the syntax of languages.

In this paper, we give an overview of the syntax definition formalism SDF3 and its tool ecosystem that supports the multi-purpose interpretation of syntax definitions. The paper does not present any new technical contributions, but it is the first paper to give a (high-level) overview of all aspects of SDF3 and serves as a guide to the literature. SDF3 is the third generation in the SDF family of syntax definition formalisms, which were developed in the context of the ASF+SDF [5], Stratego/XT [10], and Spoofax [38] language workbenches.

The first SDF [23] supported modular composition of syntax definition, a direct correspondence between concrete and abstract syntax, and parsing with the full class of context-free grammars enabled by the Generalized-LR (GLR) parsing algorithm [56,44]. Its programming environment, as part of the ASF+SDF MetaEnvironment [40], focused on live development of syntax definitions through

To appear in: F. S. de Boer and A. Cerone (Eds.). *Software Engineering and Formal Methods (SEFM 2020)*, LNCS, Springer, 2020.

Search docs

Docs » Syntax Definition with SDF3

http://metaborg.org

Syntax Definition with SDF3

The definition of a textual (programming) language starts with its syntax. A grammar describes the well-formed sentences of a language. When written in the grammar language of a parser generator, such a grammar does not just provide such a description as documentation, but serves to generate an implementation of a parser that recognizes sentences in the language and constructs a parse tree or abstract syntax tree for each valid text in the language. **SDF3** is a *syntax definition formalism* that goes much further than the typical grammar languages. It covers all syntactic concerns of language definitions, including the following features: support for the full class of context-free grammars by means of generalized LR parsing; integration of lexical and context-free syntax through scannerless parsing; safe and complete disambiguation using priority and associativity declarations; an automatic mapping from parse trees to abstract syntax trees through integrated constructor declarations; automatic generation of formatters based on template productions; and syntactic completion proposals in editors.

Table of Contents

Previous Next