Interactive Programming

Type `IO`

Le type `IO a` représente une action qui produit une valeur de type `a` avec des effets secondaires possibles.

```
main :: IO ()
main = do
    putStrLn "Entrez une ligne de texte:"
    input <- getLine
    putStrLn ("Vous avez entré: " ++ input)
```

Basic I/O Functions

```
return :: a -> IO a
(>>=)  :: IO a -> (a -> IO b) -> IO b
(>>)   :: IO a -> IO b -> IO b

getChar :: IO Char
getLine :: IO String
putChar :: Char -> IO ()
putStr  :: String -> IO ()
putStrLn :: String -> IO ()
```

Do Blocks

La notation `do` facilite l'écriture de séquences d'actions I/O.

```
getLine :: IO String
getLine = do c <- getChar
    if c == '\n' then return ""
    else do line <- getLine
        return (c:line)
```

- Les actions sont exécutées dans l'ordre
- `<-` capture le résultat d'une action I/O
- `let` définit des variables locales sans I/O

Control Flow Actions

```
-- Exécute une liste d'actions
sequence_ :: [IO a] -> IO ()

-- Applique une fonction I/O à une liste
mapM_ :: (a -> IO b) -> [a] -> IO ()

-- Version avec arguments inversés
forM_ :: [a] -> (a -> IO b) -> IO ()

-- Exécution conditionnelle
when :: Bool -> IO () -> IO ()
unless :: Bool -> IO () -> IO ()
```

Files

```
import System.IO

-- Écriture
writeFile "example.txt" "Hello, World!"

-- Lecture
contents <- readFile "example.txt"

-- Gestion des lignes
let linesToWrite = ["Line 1", "Line 2"]
writeFile "file.txt" (unlines linesToWrite)
fileLines = lines contents
```

Monads and more

Functors

`fmap` applique une fonction à une valeur encapsulée dans un contexte.

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
    (<$) :: a -> f b -> f a
```

Lois :
1. Identité : `fmap id = id`
2. Composition : `fmap f (fmap g x) == fmap (f . g) x`

Opérateur `<$` : remplace toutes les valeurs dans un contexte.

```
> "abc" <$ Just 123
Just "abc"
```

Applicative

Permet d'appliquer un nombre illimité d'arguments dans un foncteur.

```
class Functor f => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

Exemple avec Maybe :

```
> pure (+) <*> Just 1 <*> Just 2
Just 3
> pure (+) <*> Nothing <*> Just 2
Nothing
```

Monads

Les monades permettent de chaîner des opérations dépendantes.

```
class Applicative m => Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

Lois :
1. Left identity : `return a >>= f == f a`
2. Right identity : `m >>= return == m`
3. Associativity : `(m >>= f) >>= g == m >>= (\x -> f x >>= g)`

Do Notation :

```
result :: Maybe Double
result = do
    a <- safeDivide 10 2
    b <- safeDivide a 0
    return b
```

Lazy Evaluation

Reduction Strategies

```
square (3 + 4)
= (. * .) (3 + 4)
   |___|_____^
= (. * .) 7
   |___|__^
= 49
```

Innermost reduction : réduit les expressions internes en premier.

```
square (3 + 4)
= square 7
= 7 * 7
= 49
```

Outermost reduction : réduit les expressions externes en premier.

```
square (3 + 4)
= (3 + 4) * (3 + 4)
= 7 * (3 + 4)
= 7 * 7
= 49
```

L'outermost reduction évite les évaluations inutiles et permet de travailler avec des structures infinies.

Infinite Lists

```
ones :: [Int]
ones = 1 : ones

head ones = head (1 : ones)
          = 1
```

Strict Evaluation

Force l'évaluation immédiate avec `seq` ou `$!`.

```
(!$) :: (a -> b) -> a -> b
f $! x = x `seq` f x

> const 42 $ undefined
42
> const 42 $! undefined
*** Exception: undefined
```

Syntax Analysis

Concrete vs Abstract Syntax

$\langle stmt \rangle$ ::= ...
| 'while' '(' $\langle expr \rangle$ ')' $\langle body \rangle$

$\langle body \rangle$ ::= ';'
| $\langle stmt \rangle$ ';'
| '{' $\langle stmts \rangle$ '}'

$\langle stmts \rangle$ ::= $\langle stmt \rangle$ ';'
| $\langle stmt \rangle$ ';' $\langle stmts \rangle$

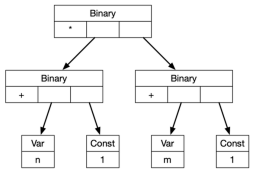
$\langle expr \rangle$ ::= ...

Syntaxe concrète : représentation textuelle du code (lexèmes, grammaire, ponctuation).

Syntaxe abstraite : structure logique représentée par des AST.

```
data Expr = Const Int
          | Var String
          | Binary Char Expr Expr
```

Abstract Syntax Tree (AST)



Context-Free Grammars

Une grammaire $G = (V, \Sigma, R, S)$ avec :
- V : variables (non-terminaux)
- Σ : symboles terminaux
- R : règles de production
- S : symbole de départ

Backus-Naur Form (BNF)

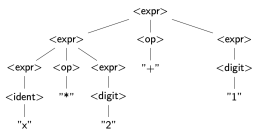
Construct	Notation	Example
Non-terminal symbol	<...>	<keywords>
Terminal symbol	"..."	"while"
Alternation	|	<expr> | <stmt>
Production rule	::=	<bool> ::= "true" | "false"

$\langle expr \rangle$ ::= $\langle expr \rangle$ $\langle op \rangle$ $\langle expr \rangle$ | $\langle digit \rangle$ | $\langle ident \rangle$
$\langle digit \rangle$ ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
$\langle ident \rangle$ ::= 'a' | 'b' | 'c' | 'd' | 'e' | ... | 'z'
$\langle op \rangle$ ::= '+' | '-' | '*' | '/'

Derivation

$$
\begin{aligned}
&< expr > \Rightarrow < expr > < op > < expr > \\
&\Rightarrow < expr > < op > < digit > \\
&\Rightarrow < expr > < op > 1 \\
&\Rightarrow < expr > + 1 \\
&\Rightarrow < expr > < op > < expr > + 1 \\
&\Rightarrow < expr > < op > < digit > + 1 \\
&\Rightarrow < expr > < op > 2 + 1 \\
&\Rightarrow < expr > * 2 + 1 \\
&\Rightarrow < ident > * 2 + 1 \\
&\Rightarrow x * 2 + 1
\end{aligned}
$$

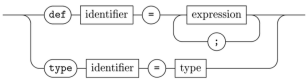
Parse Tree



EBNF Extensions

- Options : `[ ... ]` (optionnel)
- Répétitions : `{ ... }` (zéro ou plusieurs fois)
- Groupements : `( ... )`

Syntax Diagram



Precedence and Associativity

Précédence : ordre d'évaluation des opérateurs.
`3 + 4 * 5 = 3 + (4 * 5)  // * avant +`

Associativité :
- Left-associative : `5 - 3 - 2 = (5 - 3) - 2`
- Right-associative : `2 ^ 3 ^ 2 = 2 ^ (3 ^ 2)`