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 ()
readFile :: FilePath -> IO String
```

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

Modules et Modules Importants

```
-- Export (optionnel)
module MonModule (maFonction) where
```

`System.IO` – Opérations fichiers avancées

`System.Environment` – getArgs, getProgName

`Control.Monad` – when, unless, forM, *mapM*

`Data.Char` – toUpper, isDigit, isSpace, etc.

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"
```

Lazy Evaluation

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
```

Lexical Analysis

Design of Programming Languages

Un langage de programmation est défini par sa **syntaxe** et sa **sémantique**.

- **Syntaxe** : structure des programmes (alphabet, vocabulaire, grammaire)
- **Sémantique** : signification des constructions syntaxiques

Système de types :

- **Typage statique** : types vérifiés à la compilation (Java, C, Haskell)
- **Typage dynamique** : types vérifiés à l'exécution (Python, JavaScript)
- **Type inference** : le compilateur déduit les types automatiquement.

Tokens and Lexemes

Token : unité lexicale avec type et valeur.

Catégories de tokens :

- **Identifiers** : `x, foo, PI`
- **Keywords** : `if, return, while`
- **Separators** : `(), {}, ;`
- **Operators** : `+, -, =, ==`
- **Literals** : `42, "hello", 3.14`
- **Comments** : `// This is a comment`

Lexème : séquence de caractères correspondant à un token.

```
x = 42;
↓
x (identifier), = (operator), 42
(literal), ; (separator)
```

Regular Expressions

Expressions régulières pour spécifier les patterns de lexèmes.

Opérations de base :

- **Alternation** (|) : `a|b` → `a` ou `b`
- **Concatenation** : `ab` → `a` suivi de `b`
- **Closure** (*) : `a*` → zéro ou plusieurs `a`

Fermetures :

- Finite closure (*) : `a*` → `ε, a, aa, aaa, ...`
- Positive closure (+) : `a+` → `a, aa, aaa, ...`

Raccourcis :

- `r?` : zéro ou une occurrence (équivalent à `r|ε`)
- `[a-z]` : caractère entre a et z
- `[0-9]+` : un ou plusieurs chiffres

Exemples pour tokens :

- Identifier : `[a-zA-Z_][a-zA-Z0-9_]*`
- Integer : `[+-]?[0-9]+`
- Keyword : `where|if|else|return`

Finite Automata

Modèles mathématiques pour reconnaître des patterns.

String recognition :

1. Commencer à l'état initial
2. Lire caractère par caractère, suivre les transitions
3. OK -> Si état final après lecture complète

Types d'automates :

- **DFA** (Deterministic) : une seule transition par symbole
- **NFA** (Nondeterministic) : plusieurs transitions possibles, transitions ε

Conversion NFA → DFA :

1. $Q' = \emptyset$
2. Ajouter état initial à Q'
3. Pour chaque état dans Q', trouver tous les états accessibles
4. États finaux F' = ensemble des états contenant F

Lexers

Lexer : composant effectuant l'analyse lexicale, génère un stream de tokens.

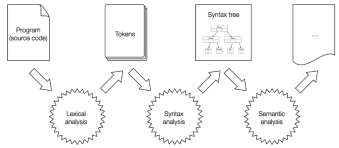
Single-pass process :

1. Lire caractères un par un
2. Grouper en lexèmes selon regex
3. Catégoriser en tokens
4. Token final : `EOF` (End Of File)

Résolution d'ambiguïté :

- **Longest match** : choisir la correspondance la plus longue

Exemple : `int` → keyword (priorité) plutôt qu'identifier



Syntax Analysis

Concrete vs Abstract Syntax

⟨*stmt*⟩ ::= ...
| `'while'` `'('` ⟨*expr*⟩ `')'` ⟨*body*⟩

⟨*body*⟩ ::= `';'`
| ⟨*stmt*⟩ `';'`
| `'{'` ⟨*stmts*⟩ `'}'`

⟨*stmts*⟩ ::= ⟨*stmt*⟩ `';'`
| ⟨*stmt*⟩ `';'` ⟨*stmts*⟩

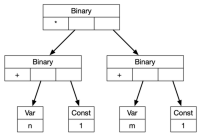
⟨*expr*⟩ ::= ...

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)



Backus-Naur Form (BNF)

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

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

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)`