PLP

Lexical Analysis

15 December 2025

Table des matières

1 Design of Programming Languages

Un langage de programmation est défini par sa syntaxe et sa sémantique. La syntaxe décrit la structure des programmes, tandis que la sémantique définit leur signification.

1.1 Syntax

Un langage de programmation est avant tout un **langage** qui est composé de:

- **Alphabet** : Ensemble fini de symboles (caractères)
- **Vocabulaire** : Ensemble fini de mots (lexèmes) construits à partir de l'alphabet
- **Grammaire** : Ensemble de règles définissant la structure des phrases (programmes)

1.2 Specifying syntax through grammar rules

Les grammaires formelles (BNF) sont utilisées pour spécifier la syntaxe des langages de programmation. Si on exemplifie avec une boucle `while`, nous aurions:

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

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

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

⟨*expr*⟩ ::= ...

Fig. 1. – Capture des slides du cours – While loop grammar

1.3 Semantics

La sémantique d'un langage de programmation concerne la signification des constructions syntaxiques. L'analyse sémantique vérifie que les programmes respectent les règles sémantiques du langage, au-delà de la simple syntaxe.

Pour décrire la sémantique formellement, on utilise des **sémantiques opérationnelles** (définissant l'exécution des programmes). Si nous souhaitions représenter la sémantique `if then else`, nous pourrions avoir:

$E-\text{IfTrue}$

$$\frac{\Gamma \vdash e_c \Rightarrow \text{true}}{\Gamma \vdash \text{if } e_c \text{ then } e_t \text{ else } e_e \Rightarrow e_t}$$

$E-\text{IfFalse}$

$$\frac{\Gamma \vdash e_c \Rightarrow \text{false}}{\Gamma \vdash \text{if } e_c \text{ then } e_t \text{ else } e_e \Rightarrow e_e}$$

Ces règles permettent de définir formellement comment un programme `if then else` doit être évalué en fonction de la condition. Nous appelons ça des **règles d'inférence**.

1.4 Typing

Les systèmes de types attribuent des types aux expressions pour garantir la cohérence et prévenir les erreurs. Il existe deux principaux types de systèmes de types :

- **Typage statique** : Les types sont vérifiés à la compilation (ex: Java, C)
- **Typage dynamique** : Les types sont vérifiés à l'exécution (ex: Python, JavaScript)

Haskell utilise un système de types statique avec inférence de types, permettant au compilateur de déduire les types sans annotations explicites. Cela permet de s'assurer que les programmes sont corrects avant l'exécution, réduisant ainsi les erreurs à l'exécution.

1.4.1 Type inference

L'inférence de types permet au compilateur de déduire les types des expressions sans annotations explicites. Par exemple, dans Haskell, si nous écrivons une fonction sans spécifier les types, le compilateur peut les inférer automatiquement.

```
add x y = x + y
```

Ici, le compilateur déduit que `x` et `y` sont de type `Num` (nombre) en fonction de l'opération `+`.

1.5 Typing rules

Les règles de typage définissent comment les types sont attribués aux expressions. Par exemple, pour une expression `if`, nous aurions:

$T-\text{If}$

$$\frac{\Gamma \vdash e_c : \text{Bool}\ \Gamma \vdash e_t : T \Gamma \vdash e_e : T}{\Gamma \vdash \text{if } e_c \text{ then } e_t \text{ else } e_e : T}$$

Grâce à cette règle, nous pouvons vérifier que les branches `then` et `else` ont le même type (T), assurant ainsi la cohérence du programme. On s'assure aussi que la condition est de type `Bool`.

1.6 Runtime system

Le **système d'exécution** (runtime system) est responsable de la gestion de l'exécution des programmes, y compris la gestion de la mémoire, l'évaluation des expressions et la gestion des erreurs. Parmis les tâches déléguées au runtime system, on retrouve:

- Gestion de la mémoire (allocation, garbage collection)
- Accès aux variables et fonctions
- Mechanisme pour passer les arguments et retourner les valeurs
- Interfacer avec le système d'exploitation

1.7 Standard Library

La **bibliothèque standard** fournit un ensemble de fonctions et de types prédéfinis pour faciliter le développement. Elle inclut souvent:

- Algorithmes de base (tri, recherche)
- Structures de données (listes, tableaux, dictionnaires)
- Interactions avec le système (fichiers, entrées/sorties)

1.8 Programming language implementation

L'implémentation d'un langage de programmation peut se faire via un **interpréteur** ou un **compilateur**.

- **Interpréteur** : Exécute le code source directement, ligne par ligne (ex: Python, JavaScript)
- **Compilateur** : Traduit le code source en code machine avant l'exécution (ex: C, Rust)

1.9 Translation process

Le processus de traduction d'un langage de programmation comprend plusieurs étapes:

- Chaque phase transforme le programme d'une représentation à une autre.
- Chaque phase prends en entrée la sortie de la phase précédente.

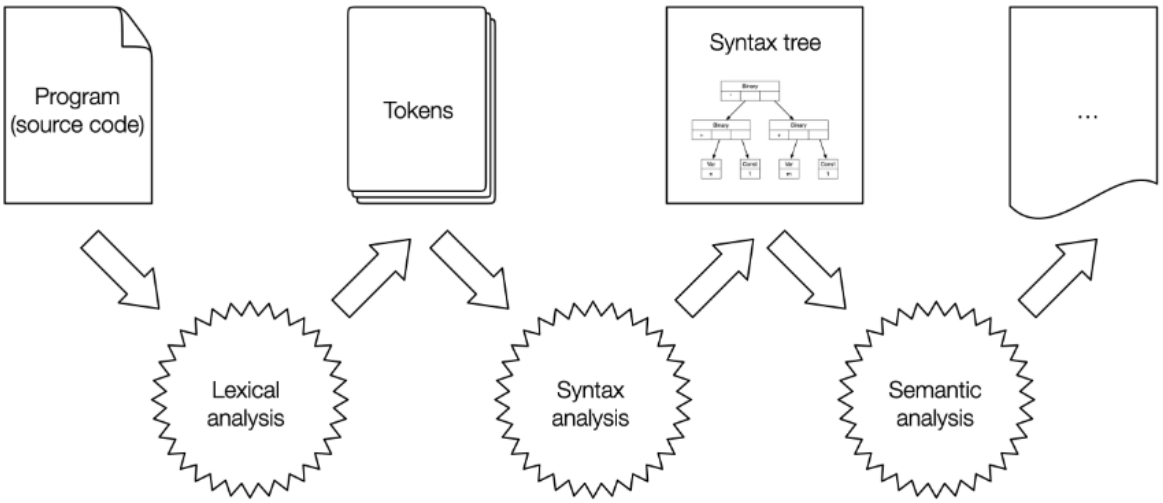

Fig. 2. – Capture des slides du cours – Phases de traduction d'un langage de programmation

2 Lexical Analysis

L'analyse lexicale est la première étape du processus de traduction d'un langage de programmation. Elle consiste à transformer le code source en une séquence de **tokens** (unités lexicales) qui seront utilisées par l'analyse syntaxique.

En lisant le code source, l'analyser lexical peut faire des taches annexes comme:

- Suppression des commentaires et espaces blancs
- Gestion des erreurs lexicales (caractères invalides)

2.1 Tokens

Un **token** est une unité lexicale qui représente une catégorie de lexèmes dans le code source. Chaque token a un **type** et une **valeur**.

Il existe plusieurs catégories de tokens:

- **Identifiers**: noms de variables, fonctions, etc.
- **Keywords**: mots réservés du langage (if, while, return)
- **Separators**: symboles de ponctuation (parenthèses, virgules)
- **Operators**: symboles d'opération (+, -, *, /)
- **Literals**: valeurs constantes (nombres, chaînes de caractères)
- **Comments**: annotations dans le code (souvent ignorées)

Exemple de classification de tokens pour une ligne de code:

- identifier: `x, foo, PI`
- keyword: `if, return, while`
- separator: `(), {}, ;`
- operator: `+, -, =, ==`
- literal: `42, "hello", 3.14`
- comment: `/* Multi-line */ // This is a comment`

2.2 Lexemes

Un **lexème** est une séquence de caractères dans le code source qui correspond à un token spécifique. Par exemple, dans l'expression `x = 42;`, les lexèmes sont:

- `x` (identifier)
- `=` (operator)
- `42` (literal)
- `;` (separator)

2.3 Lexical grammar

La grammaire lexicale définit les règles pour reconnaître les lexèmes dans le code source. Elle est souvent spécifiée à l'aide d'expressions régulières. Par exemple:

- Identifiers: `[a-zA-Z_][a-zA-Z0-9_]*`
- Keywords: `if|else|while|return`
- Separators: `\(|\)|\{|\}|;|,`
- Operators: `\+|\-|\*|\/|=|==`
- Literals: `\d+|"\w*"`

2.4 Lexical errors

Les erreurs lexicales se produisent lorsque le lexer rencontre une séquence de caractères qui ne correspond à aucun lexème valide. Par exemple:

- Caractères non reconnus: `@, #, $`
- Identificateurs mal formés: `1variable, var-name!`

Ces erreurs doivent être signalées pour que le programme puisse être corrigé avant l'analyse syntaxique.

3 Regular Expressions

Les expressions régulières (regex) sont une approche algébrique pour décrire un langage. Elles sont souvent utilisées pour spécifier des patterns de texte, comme les lexèmes dans l'analyse lexicale.

3.1 Basic operations

Les opérations de base des expressions régulières incluent:

1. **Alternation** (|) : Permet de choisir entre plusieurs options. Ex: `a|b` correspond à `a` ou `b`.
2. **Concatenation** : Combine des séquences de caractères. Ex: `ab` correspond à `a` suivi de `b`.
3. **Closure** (*) : Permet de répéter une séquence zéro ou plusieurs fois. Ex: `a*` correspond à une séquence de `a` répétée zéro ou plusieurs fois.

Les parenthèses peuvent être utilisées pour grouper des expressions et contrôler l'ordre des opérations. Elles ont les priorités les plus élevées.

3.2 Finite and posive closure

Une **fermeture finie** (Kleene star) permet de répéter une séquence zéro ou plusieurs fois, tandis qu'une **fermeture positive** (Kleene plus) permet de répéter une séquence une ou plusieurs fois.

- Finite closure (*): `a*` correspond à ``, `a`, `aa`, `aaa`, ...
- Positive closure (+): `a+` correspond à `a`, `aa`, `aaa`, ...

3.2.1 Examples

Regex	Matches
abc	the set of strings with the single member {*abc*}
a|*b*|*c*	the set of independent characters {*a*, *b*, *c*}
*a**	the infinite set {ϵ, *a*, *aa*, *aaa*, *aaaa*, ...}
*ab**	the infinite set {*a*, *ab*, *abb*, *abbb*, ...}
a(*bc*|*d*)**e*	the set of strings including {*ae*, *abce*, *abcde*, *ade*, ...}.

Fig. 3. – Capture des slides du cours – Examples of finite and positive closure

3.3 Notational shortcuts

Quelques raccourcis notatifs couramment utilisés dans les expressions régulières incluent:

- `r?` : Correspond à zéro ou une occurrence de `r` (équivalent à `r|ε`).
- `[a-z]` : Correspond à n'importe quelle lettre minuscule de `a` à `z`.
- `*` : Correspond à zéro ou plusieurs occurrences de l'élément précédent.

3.4 Lexical analysis with regex

Les expressions régulières sont souvent utilisées pour définir les règles de l'analyse lexicale. Chaque type de token peut être spécifié à l'aide d'une expression régulière. Par exemple:

Token	**Lexeme**	**Regex**
Keyword	where	where
Identifier	A_123	[a-zA-Z_][a-zA-Z0-9_]*
Integer	123	[+-]?[0-9]+

4 Finite Automata

Les finite automata (automates finis) sont des modèles mathématiques utilisés pour reconnaître des patterns de langages. Ils sont similaires aux machines à états finis vu et discuté au cours d'ARO.

4.1 String recognition

Un automate fini peut être utilisé pour reconnaître si une chaîne de caractères appartient à un langage défini. Pour ce faire, 3 étapes principales sont nécessaires:

1. On commance à l'état initial.
2. Dans chaque étape, nous faisons une des deux choses:
   - Suivre une transition à un autre état
   - Lire le caractère courant et suivre la transition correspondante
3. Quadn tous les les caractères ont été lus, si l'état courant est un état d'acceptation, la chaîne est reconnue.
   - Si l'état est bon, alors la chaîne lue est acceptée.
   - Sinon, elle est rejetée.

4.1.1 Example

Voici un exemple d'automate fini qui reconnaît la chaîne décrite par la regex: `(a|b)ba*b`

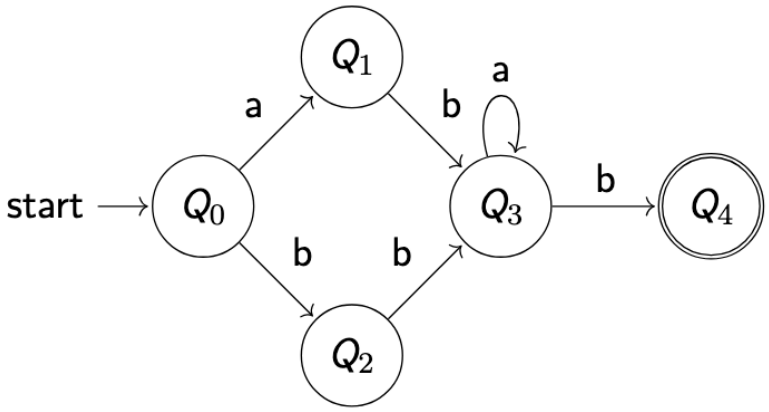

Fig. 4. – Capture des slides du cours – String recognition with finite automata

4.2 Types of finite automata

Il existe deux types principaux d'automates finis:

- **Deterministic Finite Automata (DFA)** : Chaque état a au plus une transition pour chaque symbole de l'alphabet.
- **Nondeterministic Finite Automata (NFA)** : Un état peut avoir plusieurs transitions pour le même symbole, ou des transitions ε (sans lire de symbole).

> **i Info**
>
> Chaque DFA peut-être converti en un NFA équivalent, cependant l'inverse n'est pas vrai.

4.2.1 Converting NFA to DFA

La conversion d'un NFA en un DFA peut être réalisée à l'aide de l'algorithme:

1. On définit $Q' = \emptyset$
2. On ajoute l'état intial q_0 from Q to Q'
3. Pour chaque état q dans Q'
   - Trouve tous les états accessibles à partir de q pour chaque symbole de l'alphabet
     - Si un nouvel état est trouvé, l'ajouter à Q'

4. L'état final `F'` sera l'ensemble des états qui ont `F` dedans.

5 Lexers

Un **lexer** (ou analyseur lexical) est un composant logiciel qui effectue l'analyse lexicale. Il lit le code source et produit une séquence de tokens en utilisant des expressions régulières et des automates finis.

Les lexers sont considérés comme générateur de stream de tokens avec état parce que l'état courant du lexer est sauvé.

> **i Info**
>
> Le token final est souvent un token spécial appelé `EOF` (End Of File) pour indiquer la fin du flux de tokens.

5.1 Single-pass process

Les lexers fonctionnent généralement en un seul passage (single-pass), lisant le code source une seule fois pour produire les tokens. Cela permet une analyse rapide et efficace. Les étapes sont les suivantes:

1. Lire les caractères du code source un par un.
2. Grouper les caractères en lexèmes en fonction des règles définies par les expressions régulières.
3. Catégoriser chaque lexème en un token.

5.2 Ambiguity and priority

Dans le cas ou plusieurs règles correspondent à un même lexème, des règles de priorité sont appliquées pour déterminer quel token doit être généré. De manière générale nous utilisons la règle du **longest match** (le plus long correspondance) et la règle de priorité des tokens (certains tokens ont une priorité plus élevée que d'autres).

- `Keyword` = int
- `Identifier` = `[a-zA-Z_][0-9a-zA-Z_]*`