PLP

Lexical Analysis

13 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.

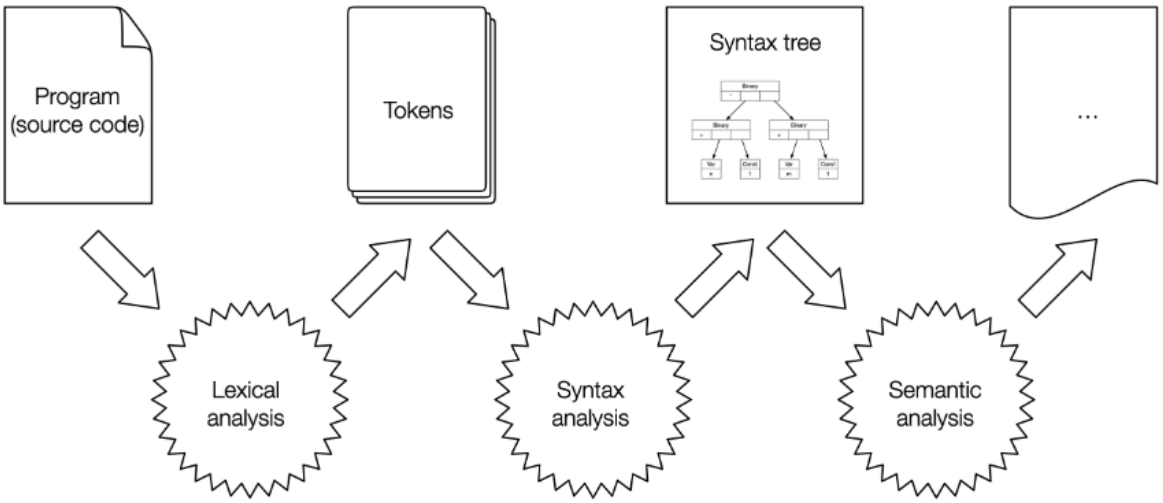

Fig. 2. – Capture des slides du cours – Phases de traduction d'un langage de programmation

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

4 Finite Automata

5 Lexers