PLP

Syntax Analysis

25 November 2025

Table des matières

1 Syntax Analysis

L'analyse syntaxique est une étape cruciale dans le processus de compilation, où le code source est transformé en une structure hiérarchique appelée arbre syntaxique abstrait (AST). Cette étape permet de vérifier la conformité du code aux règles grammaticales du langage de programmation utilisé.

Un programme qui est syntaxiquement incorrect possède donc des erreurs de syntaxe.

1.1 Syntax

La syntaxe d'un langage de programmation définit la structure et les règles qui régissent la formation des instructions et des expressions. Elle spécifie comment les différents éléments du langage, tels que les mots-clés, les opérateurs, les identificateurs et les littéraux, peuvent être combinés pour former des programmes valides.

1.1.1 Concrete syntax

La syntaxe concrète fait référence à la représentation textuelle réelle du code source, telle qu'elle est écrite par les programmeurs. Elle inclut les règles de grammaire, la ponctuation, les espaces blancs et les commentaires.

La spécification formelle de la syntaxe inclus deux types de règles :

- Les règles lexicales, qui définissent les unités de base du langage, appelées lexèmes (mots-clés, identificateurs, opérateurs, etc.).
- Les règles syntaxiques, qui définissent comment les lexèmes peuvent être combinés pour former des structures plus complexes, telles que les expressions, les instructions et les blocs de code.

1.1.1.1 Example

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

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

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

⟨*expr*⟩ ::= ...

Fig. 1. – Capture des slides du cours – Exemple de syntaxe concrète

1.1.2 Abstract syntax

La syntaxe abstraite, en revanche, se concentre sur la structure logique et hiérarchique du code, indépendamment de sa représentation textuelle. Elle est souvent représentée sous forme d'arbres syntaxiques abstraits (AST), où chaque nœud de l'arbre représente une construction syntaxique spécifique.

1.1.2.1 Example

```haskell
type Env = [(String, Int)]
data Expr = Const Int
          | Var String
          | Binary Char Expr Expr

eval :: Expr -> Env -> Int
```

1.2 Abstract Syntax Tree (AST)

Un arbre syntaxique abstrait (AST) est une représentation arborescente de la structure syntaxique d'un programme. Chaque nœud de l'arbre correspond à une construction syntaxique spécifique, telle qu'une expression, une instruction ou une déclaration.

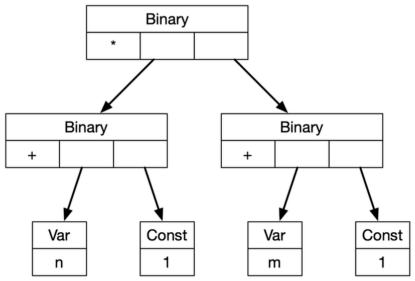

Fig. 2. – Capture des slides du cours – Exemple d'arbre syntaxique abstrait (AST)

1.3 Syntactic sugar

Le sucre syntaxique (syntactic sugar) fait référence à des constructions syntaxiques qui rendent le code plus lisible ou plus facile à écrire, sans ajouter de nouvelles fonctionnalités au langage. Le sucre syntaxique est souvent utilisé pour simplifier des expressions complexes ou pour introduire des raccourcis syntaxiques.

1.3.1 Désucrage

Le désucrage (desugaring) est le processus de transformation du sucre syntaxique en des constructions syntaxiques plus fondamentales. Cela permet de simplifier l'analyse syntaxique et la génération de code en réduisant la variété des constructions syntaxiques à traiter.

1.3.1.1 Example

```haskell
f x y = x * y

-- Désucrage
f = \x y -> x * y

-- Désucrage
f = \x -> \y -> x * y
```

1.4 Sytax Errors

Les erreurs de syntaxe se produisent lorsque le code source ne respecte pas les règles grammaticales du langage de programmation. Ces erreurs peuvent inclure des fautes de frappe, des parenthèses non appariées, des mots-clés mal utilisés ou des structures incorrectes.

Il peut y avoir différentes catégories d'erreurs de syntaxe :

- Utilisation incorrecte de la ponctuation.
- Parenthèses ou accolades non appariées.
- Constructions mal formées.
- Et pleins d'autres.

2 Context-free grammars

Une grammaire sans contexte (context-free grammar, CFG) est un ensemble de règles qui définissent la structure syntaxique d'un langage de programmation. Une CFG est composée des 4 éléments suivants $G = (V, \Sigma, R, S)$

- V : un ensemble fini de variables (ou non-terminaux) qui représentent les différentes constructions syntaxiques du langage.
- Σ : un ensemble fini de symboles terminaux qui représentent les unités de base du langage (mots-clés, opérateurs, identificateurs, etc.).
- R : un ensemble de règles de production qui définissent comment les variables peuvent être remplacées par des combinaisons de variables et de symboles terminaux.
- S : une variable spéciale appelée symbole de départ, qui représente la construction syntaxique la plus élevée du langage (généralement le programme entier).

2.1 Backus-Naur Form (BNF)

La forme de Backus-Naur (Backus-Naur Form, BNF) est une notation utilisée pour décrire les grammaires sans contexte. Elle utilise des règles de production pour spécifier comment les différentes constructions syntaxiques du langage peuvent être formées.

2.1.1 Strcuture

Une règle de production en BNF a la forme suivante :

```
<symbol> ::= <expression>
```

où `<symbol>` est une variable (non-terminal) et `<expression>` est une combinaison de variables et de symboles terminaux.

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

Fig. 3. – Capture des slides du cours – Exemple de grammaire en BNF

⟨*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*⟩ ::= '+' | '-' | '*' | '/'

Fig. 4. – Capture des slides du cours – Exemple de grammaire en BNF

2.2 Derivation

La dérivation est le processus de génération d'une chaîne de symboles terminaux à partir du symbole de départ en appliquant les règles de production de la grammaire. Chaque application d'une règle de production remplace une variable par l'expression correspondante.

2.2.1 Example

On derive une une expression arithmétique $x * 2 + 1$ en appliquant les règles de production de la grammaire définie précédemment.

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

Fig. 5. – Capture des slides du cours – Exemple de dérivation

2.3 Parse tree

Un arbre de dérivation (parse tree) est une représentation arborescente de la dérivation d'une chaîne de symboles terminaux à partir du symbole de départ. Chaque nœud de l'arbre correspond à une application d'une règle de production, et les feuilles de l'arbre représentent les symboles terminaux de la chaîne dérivée.

2.3.1 Example

Pareil que pour l'exemple de dérivation précédent, on peut représenter la dérivation de l'expression arithmétique $x * 2 + 1$ sous forme d'un arbre de dérivation.

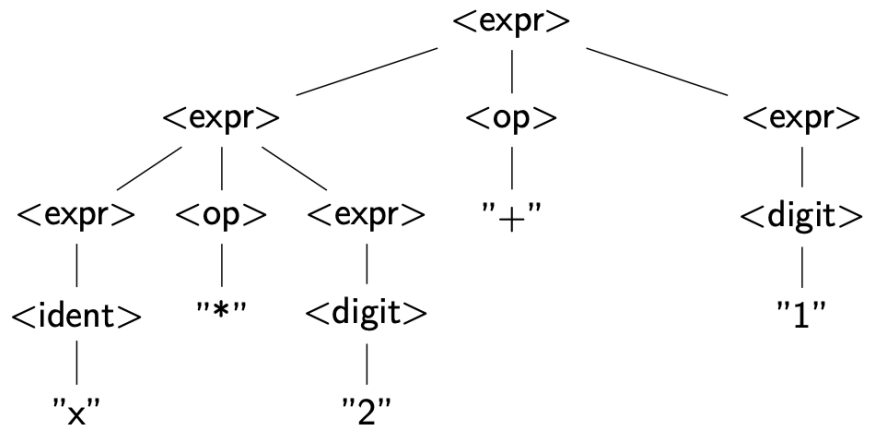

Fig. 6. – Capture des slides du cours – Exemple d'arbre de dérivation

2.4 Extended Backus-Naur Form (EBNF)

La forme de Backus-Naur étendue (Extended Backus-Naur Form, EBNF) est une extension de la BNF qui introduit des notations supplémentaires pour simplifier la description des grammaires sans contexte. L'EBNF utilise des opérateurs pour représenter des constructions syntaxiques plus complexes, telles que les répétitions et les options.

- Options: `[ ... ]` indique que l'élément à l'intérieur des crochets est optionnel.
- Répétitions: `{ ... }` indique que l'élément à l'intérieur des accolades peut apparaître zéro ou plusieurs fois.
- Groupements: `( ... )` permet de grouper des éléments pour appliquer des opérateurs à l'ensemble du groupe.

2.5 Syntax diagram

Un diagramme syntaxique (syntax diagram) est une représentation graphique de la grammaire d'un langage de programmation. Il utilise des formes géométriques pour représenter les différentes constructions syntaxiques et les relations entre elles.

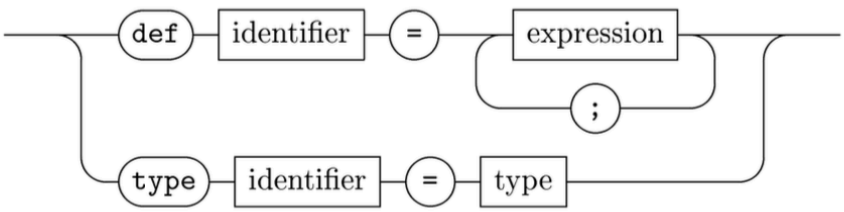

Fig. 7. – Capture des slides du cours – Exemple de diagramme syntaxique

3 Ambiguity

Une grammaire est dite ambiguë si une même chaîne de symboles terminaux peut être dérivée de plusieurs façons différentes, conduisant à des arbres de dérivation distincts. L'ambiguïté peut poser des problèmes lors de l'analyse syntaxique, car elle rend difficile la détermination de la structure correcte du code source.

3.1 The dangling else problem

Le problème du « dangling else » est un exemple classique d'ambiguïté dans les grammaires de langages de programmation. Il se produit lorsqu'une instruction « else » peut être associée à plusieurs instructions « if » imbriquées, ce qui conduit à des interprétations différentes du code source.

La plus part des langages de programmation résolvent ce problème en associant chaque « else » à l'instruction « if » la plus proche qui n'a pas encore d'« else ».

4 Predecence and associativity

Un parser qui evaluerait des expressions doit établir l'ordre dans lequel des opérations doivent être effectuées. Cela est déterminé par deux concepts clés : la précédence et l'associativité.

Sans une gestion appropriée de la précédence et de l'associativité, un parser pourrait interpréter une expression de manière incorrecte, conduisant à des résultats inattendus.

4.1 Precedence

La précédence détermine l'ordre dans lequel les opérateurs sont évalués dans une expression. Les opérateurs avec une précédence plus élevée sont évalués avant ceux avec une précédence plus basse.

4.1.1 Example

Dans l'expression `3 + 4 * 5`, l'opérateur `*` a une précédence plus élevée que l'opérateur `+`, donc l'expression est évaluée comme `3 + (4 * 5)`.

4.2 Associativity

L'associativité détermine l'ordre dans lequel les opérateurs de même précédence sont évalués. Les opérateurs peuvent être associatifs à gauche (left-associative) ou à droite (right-associative).

4.2.1 Example

Dans l'expression `5 - 3 - 2`, l'opérateur `-` est associatif à gauche, donc l'expression est évaluée comme `(5 - 3) - 2`.

4.2.2 Types of associativity

- Left-associative: Les opérateurs sont évalués de gauche à droite.
- Right-associative: Les opérateurs sont évalués de droite à gauche.

Dans la majorité des langages de programmation, les opérateurs arithmétiques tels que `+`, `-`, `*`, et `/` sont left-associative, tandis que l'opérateur d'exponentiation (par exemple, `^`) est souvent right-associative.