Analyseurs syntaxiques
Leur fonctionnement par l'exemple

Sébastien Jean Robert Doeraene

Année scolaire 2005-2006

Remerciements

Ce travail n'aurait jamais pu voir le jour sans l'aide de certaines personnes. Elles m'ont soit aidé et soutenu, soit relu, soit été une source d'informations importante.

Tout d'abord, je remercie monsieur Paul Goethals, qui a accepté d'être mon promoteur pour ce travail. Ses conseils m'ont été très utiles pour cerner le sujet à étudier, ainsi que pour le rendre accessible à des non-dingues-de-programmation...

Ensuite, je voudrais remercier messieurs Dick Grune, Henri E. Bal, Ceriel J.H. Jacobs et Koen G. Langendoen, pour leur livre *Compilateurs* [GBJL02]. Cet ouvrage a été ma référence en matière de compilation tout au long de la préparation et de la rédaction de ce document.

Troisièmement, je remercie monsieur Laurent Dardenne, membre de l'équipe de rédaction de www.developpez.com, pour ses relectures attentives et ses conseils toujours aussi constructifs.

Enfin, encore un grand merci à messieurs Néguib Serhani et Pierre Caboche, également rédacteurs sur www.developpez.com, qui ont relu mon travail et en ont corrigé les fautes d'orthographe et de formulation.

Table des matières

Chapitre I

Introduction à la compilation

Qu'est-ce que la compilation ? Voilà une question que la plupart des personnes poseront en entendant parler de cette notion. Non, ce n'est pas le fait de *compiler* plusieurs chansons sur un CD ; du moins, ce n'est pas cette compilation-là que nous étudierons ici.

Qui sont ces personnes ? Ce sont celles qui n'ont jamais programmé. En effet, la compilation est un des fondements de la programmation.

De même les développeurs, pour la plupart d'entre eux, ne se sont jamais réellement posé la question de savoir ce qu'étaient les compilateurs, au sens technique du terme.

Nous allons donc voir, au travers de ce document, ce que sont réellement les compilateurs, et comment ils fonctionnent. Nous nous attarderons plus particulièrement sur une étape de leur fonctionnement qui fait peur à beaucoup de débutants en la matière.

I.1 Qu'est-ce qu'un compilateur ?

Un compilateur est tout d'abord un programme. C'est un programme qui satisfait les quelques propriétés suivantes.

Premièrement, un compilateur reçoit un *code source* en entrée, et produit du *code objet* ou *code exécutable* en sortie. Il indique également les erreurs éventuelles survenues lors de la compilation.

La compilation en elle-même est la *transformation*, ou *conversion* du code source en code objet. Cela a une influence beaucoup plus large que la programmation. Le principe initial de la compilation n'est pas de produire un programme exécutable à partir de son source, c'est *de transformer un fichier écrit dans un format en un autre fichier utilisant un autre format mais ayant une* sémantique *identique*.

Voici quelques opérations courantes qui, contre toute apparence, sont effectivement liées à la compilation :

- transformer un document XML en un document d'un autre type (ex. : (X)HTML) au moyen d'une feuille de style XSL ;
- transformer la chaîne de caractères d'une expression mathématique en son résultat ;
- transformer un texte écrit dans une langue en ce texte traduit dans une autre langue.

Cependant, pour faciliter la compréhension de ce tutoriel, nous nous limiterons au sens le plus commun, à savoir la transformation du texte source d'un programme en code exécutable.

Le code source est un fichier texte. Il est habituellement codé avec la norme ANSI, bien que les compilateurs récents commencent à supporter les autres normes.

Le code exécutable est un fichier binaire, respectant le format des fichiers exécutables pour une architecture et un OS donnés. Pour un environnement Windows, il s'agit du format PE. Pour de plus amples informations sur ce format, consultez le tutoriel d'Olivier Lance [LAN05].

Pour certains types de langages, que l'on appelle les langages semi-compilés, le fichier en sortie n'est pas du code objet. Il s'agit alors de *pseudo-code*. C'est une version analysée mais non fonctionnelle du code source. Cela a l'avantage de pouvoir être exécuté sur plusieurs plates-formes, tout en étant plus rapide qu'un langage interprété. C'est le cas, pour exemple, du langage **Java**. Dans ce cas, le pseudo-code ainsi généré sera lu et exécuté par une *machine virtuelle*.

En ce qui concerne les erreurs, il en existe traditionnellement de quatre types :

Les conseils : Ils donnent une information au programmeur pour probablement améliorer l'efficacité de son programme, mais n'arrêtent pas le processus de compilation ;

Les avertissements : Ils avertissent le programmeur d'une erreur probable lors de l'exécution, mais n'arrêtent pas le processus de compilation ;

Les erreurs : Elles indiquent au programmeur une faute dans le code source, faute qui doit être corrigée avant de pouvoir produire un code objet correct (mais la compilation continue) ;

Les erreurs fatales : Elles stoppent immédiatement la compilation, et empêchent le compilateur de produire le code objet.

I.2 Langages source, cible et d'implémentation

Lors de l'étude d'un compilateur, on parle régulièrement de langage source, de langage cible, et de langage d'implémentation.

Le langage source est le langage dans lequel est écrit le code source. Il peut s'agir, comme nous l'avons signalé au début de ce document, aussi bien de XML ou d'une expression mathématique que d'un langage de programmation. Dans nos exemples, nous parlerons essentiellement du langage Pascal ; les mêmes raisonnements peuvent être appliqués à n'importe quel type de langage source.

Le langage cible est le langage dans lequel est écrit le code objet (ou exécutable). Nous nous limiterons dans notre cas à une représentation XML de la structure du code source.

Le langage d'implémentation est le langage de programmation avec lequel est créé le compilateur lui-même. Ici, il s'agira de Delphi, version 2005, édition Architecte.

I.3 Les différentes parties d'un compilateur

Tout compilateur est composé de deux parties : la partie avant et la partie arrière.

La partie avant se charge en premier lieu d'analyser le texte source, suite linéaire de caractères, pour le transformer en une représentation en arbre. Ceci permet de clarifier la structure du

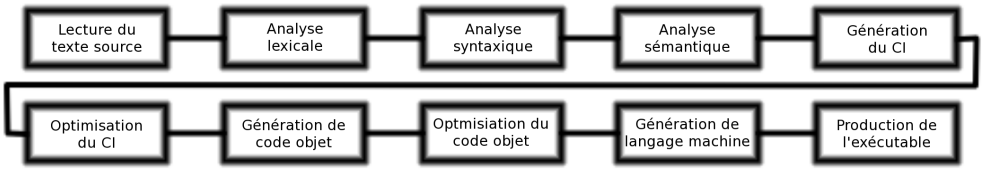

FIG. 1.1 – Modules d'un compilateur

code, conformément aux spécifications du langage source. Dans un second temps, elle réécrit le comportement du programme dans un *Langage Intermédiaire* (abrégé en LI), plus simple que le langage source, et donc plus facile à convertir ensuite en du code objet. Il en ressort du **Code Intermédiare**, abrégé en CI.

La partie arrière reçoit le code intermédiaire généré par la partie avant, et le décline dans le langage cible.

Chacune de ces deux parties est elle-même composée de plusieurs *modules*. Nous allons voir leurs rôles respectifs dans les grandes lignes.

La figure 1.1 représente les connexions entre les différents modules.

I.3.1 Partie avant

Comme nous l'avons vu plus haut, la partie avant reçoit en entrée la suite linéaire des caractères du code source, et a pour responsabilité d'en extraire la structure, afin de la réécrire dans un langage plus simple, le langage intermédiaire.

Pour faire ceci, le code source passe au travers de cinq modules différents.

I.3.1.1 Module de lecture du texte source

Ce module lit le fichier source, au moyen des APIs du système d'exploitation utilisé pour la compilation, et donne au module suivant la suite des caractères qui le composent.

En Delphi, on peut se servir simplement de la méthode **LoadFromFile** de la classe **TStrings**.

I.3.1.2 Module d'analyse lexicale

Ce deuxième module rassemble des suites de caractères, données par le module de lecture du texte source, en une suite, tout autant linéaire, de lexèmes. Un lexème est une entité composée de plusieurs caractères, qui a une signification dans le langage source. Il s'agit par exemple d'identificateurs, d'opérateurs, ou de mots-clefs. Les lexèmes peuvent être séparés par des espaces ; et dans certains cas, ils doivent l'être.

Observons l'instruction Pascal suivante :

```
if Entier > 0 then
  ShowMessage('Le nombre est positif');
```


On peut y trouver 10 lexèmes, que l'on peut répartir en cinq catégories :

Identificateurs : `Entier` et `ShowMessage` ;

Mots-clefs : `if` et `then` ;

Opérateurs : plus grand (`>`), parenthèses `()` et point-virgule (`;`) ;

Nombres entiers : `0` ;

Chaînes de caractères : `'Le nombre est positif'`.

Il ne faut pas confondre les *constantes* de types nombre entier ou chaîne de caractères, comme `0` et `'Le nombre est positif'`, avec les *variables* comme `Entier`. Dans le cas des constantes, c'est à l'analyseur lexical de déterminer le type et la valeur, à l'aide d'une classe de lexèmes différente selon le type. Dans le cas des variables, c'est à l'analyseur sémantique de découvrir le type, puisqu'elles sont toutes regroupées sous la classe de lexèmes `Identificateur`.

Vous pourrez trouver une introduction à l'analyse lexicale dans l'article sur les Lexers (autre nom des analyseurs lexicaux) d'Olivier Lance [LAN04].

Bien que nous n'étudierons pas l'analyse lexicale, nous aurons besoin d'un analyseur, dans la mesure où nous travaillerons sur des lexèmes dans l'analyse syntaxique. Vous trouverez une brève explication de l'analyseur lexical utilisé dans l'annexe A.

I.3.1.3 Module d'analyse syntaxique

"Le module d'analyse syntaxique restructure le flot de lexèmes, donnés de façon linéaire par le module précédent, en un arbre abstrait. Chaque feuille de l'arbre correspond à un lexème..."

Le module d'analyse syntaxique restructure le flot de lexèmes, donnés de façon linéaire par le module précédent, en un arbre abstrait. Chaque lexème est placé sur une feuille de l'arbre. Il fournit cet arbre abstrait non décoré au module d'analyse sémantique.

I.3.1.4 Module d'analyse sémantique

L'analyse sémantique collecte des informations dans l'arbre abstrait non décoré pour le *décorer* des résultats obtenus. Elle forme ainsi l'arbre abstrait décoré, abrégé en arbre abstrait.

Notre module d'analyse sémantique ne fera rien en réalité. Puisque tout ce que nous voulons dans cette étude de l'analyse syntaxique, c'est récupérer l'arbre abstrait non décoré.

I.3.1.5 Module de génération du code intermédiaire

Ce dernier module de la partie avant produit un code intermédiaire à partir de l'arbre abstrait décoré. C'est une écriture de l'arbre abstrait que pourra comprendre la partie arrière.

Puisque nous ferons office nous-mêmes de partie arrière, afin de vérifier si l'analyseur syntaxique fait bien son travail, notre module de génération du code intermédiaire nous fournira une écriture XML de l'arbre abstrait[1].

[1] Note : dans un vrai compilateur, ce serait une bien mauvaise idée, étant donné que la représentation XML devrait être à nouveau analysée !

I.3.2 Partie arrière

La partie arrière est responsable de la transformation du code intermédiaire, en principe identique quelle que soit la plate-forme d'exploitation, en du code objet exécutable par un système en particulier, par un système en particulier tel que Windows ou Linux.

Les informations ci-après sont présentes uniquement à titre indicatif, puisque nous ne produirons pas du tout de partie arrière. Nous récupérerons directement le code intermédiaire écrit en XML pour visualiser l'arbre abstrait.

I.3.2.1 Module d'optimisation du code intermédiaire

Ce module effectue quelques optimisations directement sur le code intermédiaire. Une optimisation couramment implémentée ici est le pliage des constantes. C'est la compression d'une expression, dont on connaît la valeur de toutes les opérandes à la compilation, en son résultat.

I.3.2.2 Module de génération de code

La génération de code réécrit le code intermédiaire en une liste linéaire d'instructions du système d'exploitation, de façon plus ou moins symbolique.

I.3.2.3 Module d'optimisation du code objet

Ce troisième module de la partie arrière optimise à nouveau le code symbolique que lui donne la génération de code. Certaines de ces optimisations peuvent être placées dans le module d'optimisation du CI, et *vice versa*. Cependant, certaines ont plus leur place dans l'un ou l'autre module.

I.3.2.4 Module de génération de langage machine

Le module de génération de langage machine convertit la liste des instructions symboliques en leur représention binaire supportée par le processeur de la machine cible. Il génère aussi des tables d'adresses, de constantes et de relocalisation.

I.3.2.5 Module de production du code exécutable

Ce dernier module assemble les suites de bits des instructions et les différentes tables, ainsi que d'éventuels prologues et épilogues, en un seul fichier qui respecte le format des fichiers exécutables du système d'exploitation.

I.3.3 Quels modules peuvent signaler des erreurs ?

La question peut se poser : quels sont les modules susceptibles de signaler des erreurs ?

Techniquement, n'importe quel module peut provoquer des erreurs – comme n'importe quel programme – mais seuls les trois modules que l'on peut qualifier d'*analyse* indiquent des erreurs relatives à la compilation :

- l'analyseur lexical signale des erreurs lexicales, c'est-à-dire des caractères non autorisés, comme la présence (hors-chaîne) d'un $ dans un code source Pascal ;
- l'analyseur syntaxique signale des erreurs syntaxiques, c'est-à-dire une malformation dans la structure du code source, par exemple une instruction `MaVariable := ;` dans un code source Pascal ;
- l'analyseur sémantique signale des erreurs sémantiques, comme des incompatibilités de types de variables (affectation d'un `integer` à un `string` en dans un code source Pascal, par exemple).

Si un autre module génère une erreur, il s'agit probablement d'un bogue du compilateur, ou d'un quelconque problème système comme des erreurs d'entrée/sortie (accès aux fichiers) ou un dépassement de la mémoire disponible.

I.4 Architectures des compilateurs

Il existe des compilateurs avec différentes architectures. Malheureusement, il n'existe pas de terminologie exacte définie pour tel ou tel type d'architecture.

Cependant, les grandes différences d'architectures peuvent se placer en deux grandes catégories : la *largeur* du compilateur, et le *choix du module* qui commande.

Nous allons voir ces deux choix.

I.4.1 Largeur d'un compilateur

Une des deux grandes questions est quelles sont les données – quelle est la granularité des données pour être exact – qui transitent entre les différents modules.

Il y a deux choix raisonnables possibles : soit la plus petite quantité significative de données d'un module au suivant (par exemple, un lexème de l'analyse lexicale à l'analyse syntaxique) ; soit le programme dans son intégralité. En l'absence de terminologie pour ces deux types de largeur, nous parlerons respectivement de compilateurs **étroits** et **larges**.

Jusqu'en 1980, l'utilisation de compilateurs larges était impensable, à cause des besoins en mémoire d'un tel type de compilateur. Seuls des compilateurs étroits étaient alors construits. Ainsi, de nombreux outils ont été développés pour faciliter ce type d'architecture. C'est pourquoi les compilateurs étroits sont encore fortement utilisés.

En revanche, les compilateurs larges sont plus intéressants d'un point de vue pédagogique, puisqu'ils sont en vérité construits de telle sorte que le texte du programme passe successivement dans différents modules, chacun lui appliquant certaines transformations. En outre, ils sont aussi plus simples à concevoir, puisqu'ils évitent de se poser des questions telles que le choix du module qui commande, comme nous le verrons dans la section suivante.

Bien que ces deux architectures paraissent diamétralement opposées, il n'est pas impossible de mélanger les deux types. Par exemple, un compilateur pourrait rassembler des modules consécutifs en un seul, qui aurait une entrée et une sortie large, mais qui serait étroit à l'intérieur.

En pratique, avec l'augmentation permanente de la puissance et de la mémoire des machines, les compilateurs larges se développent. Ceci particulièrement avec des paradigmes non impératifs, comme les paradigmes fonctionnel et logique.

Cependant, nous utiliserons le modèle étroit dans nos deux exemples, car nous n'aurons pas besoin des apports supplémentaires d'une architecture large, dans la mesure où nous nous arrêterons à l'analyse syntaxique.

Pour terminer, sachez que, même dans les compilateurs étroits, le premier module – celui de lecture du texte source – est généralement large. En effet, cela permet d'obtenir des informations intéressantes pour la production de messages d'erreurs constructifs, tels que la position des lexèmes dans le texte source.

I.4.2 Quel module commande ?

La seconde grande question d'architecture pose le problème du choix du module qui doit commander les autres.

Comme nous l'avons dit plus haut, cette question ne se pose pas dans le cas des compilateurs larges. En effet, ceux-ci traitent l'information module par module.

En revanche, dans un compilateur étroit, il est essentiel de savoir quel est le module qui tourne en permanence et qui « appelle » les autres.

Cette question étant complexe, et ne faisant pas partie du sujet de cet article, je vous renvoie au livre *Compilateurs* [GBJL02, section 1.4.2] pour plus d'informations à ce propos.

Pour notre part, c'est le module d'analyse syntaxique qui commandera. C'est en effet celui que nous étudions et il sera plus simple de l'étudier s'il possède en permanence le contrôle.

I.5 Grammaires et leurs notations

Les grammaires sont un élément important dans la définition d'un langage de programmation.

Non seulement, elle permettent de définir de façon formelle la syntaxe valide d'un code source, mais elles permettent aussi de faire générer automatiquement des analyseurs syntaxiques pour celles-ci.

D'autre part, l'analyse syntaxique étant étroitement liée (pour ne pas dire fusionnée) aux grammaires, l'utilisation en est largement faite dans ce tutoriel. Il est donc important d'en comprendre la signification et la notation.

Nous allons voir ici la notation **BNF** (*Backus Normal Form*, en français **forme normale de Backus** ; aussi appelée *Backus-Naur Form*, ou **forme de Backus-Naur**), qui est la plus répandue pour les grammaires.

I.5.1 Symboles grammaticaux

L'élément de base de la grammaire est le symbole grammatical. Il en existe de deux types : les terminaux et les non-terminaux.

D'un point de vue de la théorie des arbres, un terminal est une feuille, et un non-terminal est un nœud intérieur, y compris la racine de l'arbre d'analyse.

D'un point de vue de la structure d'un langage, un terminal est un lexème, et un non-terminal est un groupe cohérent sémantiquement de symboles grammaticaux.

Des exemples de terminaux que nous pouvons trouver dans le langage Pascal sont des identificateurs, des mots-clefs, des opérateurs, etc. Des non-terminaux seraient les instructions, les blocs `begin...end`, les déclarations de classes.

Un non-terminal peut être composé de lui-même également ! Et c'est en effet très souvent le cas.

Un terminal est habituellement noté au moyen de la lettre t, avec éventuellement un indice indiquant sa position dans l'entrée, ainsi qu'au moyen des lettres x, y et z. Tandis qu'un non-terminal est noté au moyen d'une lettre majuscule, essentiellement N, A, B, et C. Le terminal symbolisant la fin du fichier source est représenté par le symbole $\dashv$.

Chaque grammaire définit un *symbole de départ*, qui est un non-terminal. Il correspond au non-terminal qui se trouve au-dessus de tous les autres, et à partir duquel on peut produire tout le texte du programme, en entier. Ce symbole de départ étant un non-terminal, il est noté au moyen d'une lettre majuscule ; en général, on utilise la lettre S, pour *Start*.

En Pascal, le symbole de départ pourrait être le non-terminal `Unite` suivant :

```
Unite  →  'unit' Identificateur ';' 'interface' PartieInterface
          'implementation' PartieImplementation 'end' '.' ⊣
```

Souvent, on a besoin de représenter une suite de symboles grammaticaux. On utilise pour cela les lettres grecques minuscules, en particulier α (alpha), β (bêta) et γ (gamma). Une chaîne de symboles grammaticaux vide est notée $\mathcal{E}$ (epsilon).

Voici un exemple de suite de symboles grammaticaux que l'on peut trouver dans un source Pascal :

```
, UnNombre * ( 2 +
```

Comme vous pouvez le remarquer, une suite de symboles grammaticaux ne débute et ne s'arrête pas forcément à un endroit logique du point de vue de la sémantique. La chaîne vue plus haut pourrait par exemple faire partie de l'instruction complète suivante :

```
UneFonction(False, UnNombre * (2 + AutreNombre));
```

I.5.2 Productions et choix

Une production est la recette de fabrication d'un non-terminal. Elle indique les différents symboles grammaticaux qui peuvent former un non-terminal. On note une production de cette façon :

$$N \rightarrow \alpha$$

Cela indique que le non-terminal N peut être formé de la suite de symboles grammaticaux α.

Dans les productions, on note les terminaux fixes (comme les mots-clefs ou les opérateurs) tels quels, entourés de guillemets simples. Et on note les terminaux variables (comme les identificateurs et les nombres) par leur nom de classe.

Par exemple, une production possible en Pascal pour l'instruction d'affection serait :

```
Affectation  →  Identificateur ':=' Expression
```

Un non-terminal a souvent plusieurs façons d'être implémenté. Chacune des façons d'implémenter un non-terminal N est appelée un choix de N. Chaque choix est une suite de symboles grammaticaux et est donc noté au moyen d'une lettre grecque minuscule.

Pour écrire l'ensemble des productions de N en une seule fois, on utilise la notation suivante :

$$N \rightarrow \alpha|\beta|\gamma|...$$

Où α, β, γ sont les différents choix de N, séparés d'un caractère pipe (|).

Voici par exemple un ensemble de productions de l'instruction `if...then...else` :

```
IfThenElse  →  'if' ExpressionBool 'then' Instruction Else
      Else  →  'else' Instruction | ℰ
```

Certains non-terminaux peuvent aussi avoir un choix vide. Dans ce cas, on n'oubliera pas d'ajouter $\mathcal{E}$ comme dernier choix de N.

$$N \rightarrow \alpha|...|\mathcal{E}$$

Ainsi, en Pascal par exemple, étant donné qu'une instruction peut être vide, on écrira :

```
Instruction  →  Affectation | IfThenElse | ... | ℰ
```

I.5.3 Propriétés

Les grammaires sont dotées de propriétés. Ces propriétés sont très importantes pour leur étude. Vous verrez que ce tutoriel fait souvent appel aux propriétés des grammaires.

Un non-terminal N est **récursif à gauche** si, à partir du syntagme N (c'est-à-dire un sous-arbre de dérivation qui correspond au non-terminal N), on peut produire un autre syntagme qui commence par N. Voici une forme de récursivité gauche (directe) :

$$N \rightarrow N\alpha|\beta$$

Il existe aussi la récursivité indirecte, lorsqu'un syntagme A produit un syntagme commençant par B, qui produit lui-même un syntagme commençant par A. On peut envisager une récursivité plus longue encore.

L'exemple type de récursion directe est le non-terminal représentant une somme de nombres :

```
Somme  →  Somme '+' Nombre | Nombre
```

Toute grammaire qui contient au moins un non-terminal récursif à gauche est elle-même dite récursive à gauche. La **récursivité à droite** existe aussi mais a moins d'importance.

Un non-terminal N est **nullifiable** si, à partir du syntagme N, on peut produire un syntagme vide ($\mathcal{E}$).

Le contenu de la partie **interface** d'une unité Pascal en est un bon exemple :

```
PartieInterface  →  ClauseUses ClauseTypes ClauseConsts DefProcs
     ClauseUses  →  'uses' Identificateur Identificateurs ';' | ℰ
    ClauseTypes  →  'type' DefType DefTypes | ℰ
   ClauseConsts  →  'const' DefConst DefConsts | ℰ
       DefProcs  →  DefProc DefProcs | ℰ
```

Étant donné que chaque composante de `PartieInterface` peut être une chaîne de symboles grammaticaux vides ($\mathcal{E}$), la `PartieInterface` peut elle-même être une chaîne de symboles grammaticaux vides, et est donc *nullifiable*.

Un non-terminal N est **inutile** s'il ne peut dériver aucune chaîne de terminaux. Cela peut arriver si, de quelque façon qu'on dérive le syntagme N, on retombe inévitablement sur un syntagme contenant N (de façon directe ou indirecte). En voici un exemple :

$$N \rightarrow \alpha N \beta$$

Une grammaire est dite **ambiguë** lorsque deux arbres de dérivation différents produisent la même suite de lexèmes. Ce type de grammaires doit être absolument écarté, car il empêche toute réussite de création d'un analyseur syntaxique.

I.5.4 À propos des diagrammes de Conway

Habituellement, dans la documentation des langages, ce n'est pas la notation BNF qui est utilisée. Elle n'est pas assez lisible pour les humains que nous sommes.

On utilise dans ce type de documents les diagrammes de Conway, qui sont une représentation graphique des grammaires, plutôt que textuelle.

Une excellente source d'informations à propos des diagrammes de Conway est un document sur le lexique et la syntaxe de Michel Beaudoin-Lafon [BELA98, section 3].

Chapitre II

Introduction à l'analyse syntaxique

Nous en arrivons au sujet principal de ce tutoriel, à savoir l'analyse syntaxique proprement dite. Ainsi que nous l'avons vu, il s'agit d'un des modules de la partie avant du compilateur.

Son rôle est de déterminer la structure en arbre, que cache la suite linéaire de lexèmes fournie par l'analyseur lexical, et ceci en correspondance avec la grammaire du langage à compiler.

J'ai choisi d'étudier ce module car j'estime que c'est celui qui me semble le plus délicat à implémenter pour un néophyte de la compilation. Sans un minimum de connaissances théoriques, le risque est d'être rapidement bloqué par la manière d'arranger les lexèmes en arbre.

Mais pourquoi un arbre ? Tout simplement parce que c'est de cette façon qu'est décrite un langage. Les terminaux des grammaires forment les *feuilles* de l'arbre, tandis que les non-terminaux en forment les nœuds intérieurs, le nœud racine étant le symbole de départ de la grammaire.

Il y a deux types de méthodes pour l'analyse syntaxique : l'analyse *déterministe, de gauche à droite, et descendante* et l'analyse *déterministe, de gauche à droite, et ascendante*.

Le terme *de gauche à droite* signifie que l'analyseur avance séquentiellement, dans l'ordre de gauche à droite du texte du programme, ou plus exactement des lexèmes.

Le *déterminisme* signifie qu'aucune recherche n'est nécessaire. Le traitement de chaque lexème amène l'analyseur un pas plus loin vers la construction de l'arbre syntaxique.

Les termes *descendant* et *ascendant* seront expliqués plus loin.

L'arbre syntaxique qui résulte du module d'analyse syntaxique représente la structure du code source à compiler, et satisfait les points suivant :

- Les nœuds feuilles sont étiquetés par des terminaux, et les nœuds internes par des non-terminaux ;
- Le nœud racine est étiqueté par le symbole de départ de la grammaire ;
- Les fils d'un nœud interne étiqueté N correspondent aux membres d'un des choix de N, dans le même ordre que dans le choix ;
- Les terminaux étiquetant les nœuds feuilles correspondent à la suite de lexèmes, dans le même ordre que dans l'entrée.

Vous trouverez une représentation d'un arbre abstrait sur la figure 2.1.

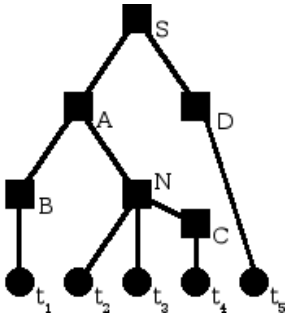

FIG. 2.1 – Arbre abstrait ou arbre syntaxique

II.1 Déterminisme de l'analyse

Premièrement, l'importance du déterminisme dans l'analyse syntaxique est marquée par le fait que le temps d'exécution d'une analyse déterministe est une fonction linéaire de la taille du texte du programme. Des techniques non déterministes existent, mais ne sont pas (encore) utilisées, car elles sont trop lentes. Nous n'en parlerons donc pas.

Cependant, une analyse déterministe n'est pas capable de traiter toutes les grammaires. La plupart des grammaires trouvées dans des manuels de langages ne conduisent pas directement à une méthode déterministe. Heureusement, il existe des techniques qui permettent de rendre adaptée à une méthode déterministe une grammaire qui ne l'est pas. Nous supposerons donc ici que toutes les grammaires utilisées peuvent être analysées de façon déterministe. Pour plus de renseignements sur les techniques de transformations, reportez-vous au livre *Compilateurs* [GBJL02, sections 2.2.4.3 et 2.2.5.7].

L'autre avantage du déterminisme est qu'une grammaire qui peut être analysée de façon déterministe est non ambiguë. C'est une propriété très importante pour un langage de programmation. Si un langage ne l'était pas, il pourrait exister des codes sources qui peuvent avoir deux significations différentes. Permettre une analyse déterministe et être non ambigu ne sont pas tout à fait les mêmes notions. La première implique la seconde mais la réciproque n'est pas vraie. Toutefois, demander le déterminisme est en pratique le meilleur test de non ambiguïté que nous possédons.

II.2 Descendant ou ascendant ?

L'on fait une distinction importante entre deux catégories d'analyseurs, selon l'ordre dans lequel ils construisent l'arbre d'analyse.

Un analyseur descendant construit l'arbre d'analyse en préordre (ou préfixe), alors qu'un analyseur ascendant le fait en postordre (ou postfixe, voire suffixe). On trouvera sur le Web de nombreuses définitions de ces termes, par exemple, le tutoriel sur les arbres de Romuald PERROT [PER06, chapitre VI].

La méthode en préordre commence par construire le nœud racine, puis « rentre à l'intérieur » pour construire ses fils ; la méthode en postordre assemble les premiers lexèmes rencontrés en

petites portions de l'arbre, puis remonte pour construire le reste à partir des portions déjà construites.

Nous verrons le principe général de ces deux types d'analyseurs. En revanche, nous n'étudierons en détail que l'analyse descendante dans le cadre de ce travail de fin d'études, et laisserons donc de côté les techniques de l'analyse ascendante. Ce choix a été fait essentiellement à cause des limitations imposées sur la longueur des travaux.

Ainsi, pour la méthode descendante, nous étudierons ensuite en détail le fonctionnement de son algorithme, au moyen d'un exemple de grammaire.

Chapitre III

Analyse descendante

Nous commençons par l'analyse descendante, et ce pour la bonne et simple raison que c'est la première à avoir été inventée... En effet, c'est la seule des deux méthodes qu'il est possible d'appliquer *à la main*, en programmant son analyseur de la première à la dernière ligne de code.

Rappelons au passage que c'est la seule méthode que nous étudierons en détail dans ce travail.

III.1 Principe général

Dans un analyseur descendant, on connaît le nœud courant et le premier lexème en entrée (voire plusieurs des premiers).

Au départ le nœud courant est le symbole de départ de la grammaire, noté S, et le lexème en entrée est le tout premier lexème du source, t_1.

Au moyen de ces deux informations, on détermine le bon choix α de S, mais on n'avance pas dans l'entrée. L'analyseur se retrouve alors avec le non-terminal N du début du choix α déterminé, et le lexème t_1 en entrée.

On répète le processus jusqu'à ce que le premier symbole grammatical du choix déterminé soit un terminal. Ce terminal reconnaît le lexème t_1 en entrée. Ce n'est pas un hasard : on a déterminé les choix α, α', etc. des non-terminaux S, N, etc. de sorte que cela arrive. Nous verrons comment on peut le faire.

On peut donc avancer dans l'entrée et on se retrouve avec la partie gauche de l'arbre construite et en entrée le deuxième lexème t_2.

En continuant le processus, la construction de l'arbre est poursuivie en sélectionnant les bons choix de sorte que les lexèmes t_i soient tous reconnus (sauf s'il y a erreur syntaxique bien sûr).

Vous pouvez voir sur la figure 3.1 une situation dans laquelle l'analyseur a déjà construit les nœuds des non-terminaux S, A et B, a avancé sur le lexème t_1, puis a construit le nœud du non-terminal N, et a finalement avancé sur les lexèmes t_2 et t_3. Il doit maintenant continuer le remplissage du nœud du non-terminal N, avec en entrée le lexème t_4.

Dans la figure suivante, les carrés symbolisent les noeuds de l'arbre. Les noirs sont déjà construits, tandis que les blancs ne le sont pas encore , bien que leur existence soit connue.

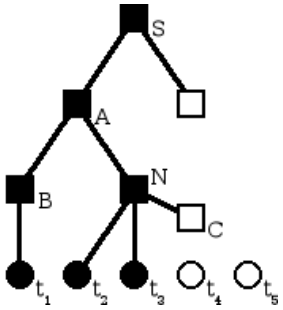

FIG. 3.1 – Exemple de situation d'un analyseur descendant

Dans cette figure, les carrés symbolisent les nœuds de l'arbre. Les noirs sont déjà construits, tandis que les blancs ne le sont pas encore, bien que leur existence soit connue. D'autre part, les cercles symbolisent les lexèmes, les noirs étant ceux sur lesquels l'analyseur a déjà avancé, et les blancs étant ceux qui restent dans l'entrée.

III.2 L'algorithme détaillé

À présent que nous avons vu comment fonctionnait l'algorithme dans sa plus grande généralité, nous allons pouvoir entrer en profondeur dans ses techniques les plus obscures...

Il existe deux grands types d'analyseurs descendants, les analyseurs non-prédictifs et les analyseurs prédictifs dits *LL(1)*. LL signifie que l'on travaille de gauche (*Left*) à droite, avec une dérivation gauche (*Leftmost*). Le (1) indique que nous travaillons avec un lexème d'avance. C'est une appellation un peu théorique et peu explicite... Retenez simplement LL(1).

Les analyseurs LL(1) sont encore divisés en deux sous-groupes : les analyseurs prédictifs *par descente récursive* et les analyseurs prédictifs *non-récursifs*.

En revanche, les analyseurs non-prédictifs sont tous *par descente récursive*.

Ces analyseurs – non-prédictifs et par descente récursive – sont les analyseurs syntaxiques que l'on peut écrire à la main. Nous commencerons par étudier ce type d'analyseur : il est simple à comprendre et à mettre en œuvre.

Ensuite, nous étudierons les analyseurs prédictifs par descente récursive, qui sont toujours écrits à la main, mais qui nécessitent d'avoir des informations qui doivent être calculées par programme.

Enfin, nous nous approcherons un peu plus de la modernité avec les techniques avancées d'analyse syntaxique que sont celles des analyseurs prédictifs non-récursifs engendrés de A à Z par des générateurs.

Comme nous l'avons signalé plus haut, dans la petite introduction au principe de l'analyse descendante, on connaît le nœud courant (et donc le non-terminal associé) et un lexème dans l'entrée. À partir de ces deux informations, nous allons devoir déterminer quel choix α de ce nœud étiqueté du non-terminal N (par la suite, pour la simplicité, nous écrirons le nœud N, bien que ce soit un abus de langage) sélectionner pour qu'une dérivation du syntagme α puisse

```
                Entree  →  Expression ⊣
            Expression  →  Terme ResteExpression
                 Terme  →  Nombre | ExpressionParenthesee
ExpressionParenthesee   →  '(' Expression ')'
       ResteExpression  →  '+' Expression | ℰ
```

FIG. 3.2 – Exemple de grammaire pour l'analyse descendante

commencer par le terminal t en entrée.

Les trois types d'analyse descendante que nous allons étudier diffèrent justement essentiellement par la façon de déterminer quel est le bon choix.

La descente récursive est légèrement naïve : elle teste chaque choix de N jusqu'à ce que l'un d'eux fonctionne. Les méthodes LL(1) sont plus intelligentes : elles prédisent le bon choix directement. La méthode non-récursive est encore plus efficace car elle utilise des tables et non des tests.

III.2.1 Analyse par descente récursive

Nous venons de dire que la descente récursive était naïve. Alors... Soyons naïfs pour découvrir comment elle fonctionne.

La manière la plus simpliste de trouver quel est le bon choix du nœud N à sélectionner, est tout simplement de les tester l'un après l'autre, et de prendre le premier qui fonctionne.

Pour cela, l'analyseur est composé d'une fonction pour chaque non-terminal de la grammaire. Pour rendre tout cela moins abstrait et ainsi améliorer la compréhension, nous allons utiliser une grammaire exemple relativement simple, que vous pouvez voir sur la figure 3.2[1].

Ainsi, notre analyseur possèdera cinq fonctions essentielles, une pour chacun des cinq non-terminaux de cette grammaire. Une fonction supplémentaire est ajoutée pour analyser un terminal (dont la classe est passée en paramètre). Chaque fonction renvoie une valeur booléenne indiquant si on a pu dériver le syntagme N jusqu'à obtenir le lexème t en entrée.

Pour arriver à faire cela, elle teste si le premier choix α du non-terminal N peut commencer par t. Si le premier symbole grammatical de α est un terminal, alors c'est trivial : le choix est bon si ce terminal est t. Si c'est un non-terminal, alors α est le bon choix si, en *descendant*[2] à la fonction correspondante à ce non-terminal, on reçoit une valeur de retour positive.

Si le choix α n'était pas le bon, alors on teste le deuxième choix β, et ainsi de suite jusqu'à ce que tous les choix aient été épuisés. Dans ce cas, la fonction renvoie **False**, pour indiquer à la fonction appelante que le choix du non-terminal N n'était pas bon.

Si on a trouvé un choix α correct, alors on admet qu'il est *complètement* correct, et ce pour assurer le déterminisme tant recherché. Cela signifie qu'on exige que la suite de ce choix soit présente. Si ce n'est pas le cas, il y a erreur syntaxique.

Le fichier source **DescenteRecursive.pas** (voir annexe B) montre le code d'un analyseur pour la grammaire de la figure 3.2. Pour analyser une chaîne d'entrée, il faut la passer en pa-

[1] Cet exemple a été repris à partir du livre *Compilateurs* [GBJL02, figure 2.57]
[2] D'où le nom d'analyse par *descente* récursive

```
        AnalyseSyntaxique('(i + i) + i + (i)')
t = (   ↪ Entree
t = (      ↪ Expression
t = (         ↪ Terme
t = (            ↪ Lexeme(lexNombre) → renvoie False
t = (         ↪ Terme
t = (            ↪ ExpressionParenthesee
t = (               ↪ Lexeme('(') → renvoie True
t = 1            ↪ ExpressionParenthesee
t = 1               ↪ Expression
t = 1                  ↪ Terme
t = 1                     ↪ Lexeme(lexNombre) → renvoie True
t = +                  ↪ Terme → renvoie True
t = +   etc.
```

FIG. 3.3 – Début du graphe de contrôle du flux de l'analyse de `(1 + 2) + 3 + (4)` ⊣

ramètre à la fonction **AnalyseSyntaxique**, et on reçoit en valeur de retour un objet de type **TNoeudEntree** qui est la racine de l'arbre syntaxique analysé. Le même fonctionnement sera utilisé pour tous les types d'analyseurs syntaxiques que nous donnerons en exemple.

Nous allons maintenant examiner comment cet analyseur peut analyser une entrée. Voici l'exemple d'entrée (valide) que nous allons utiliser :

```
(1 + 2) + 3 + (4) ⊣
```

Je vous conseille fortement de suivre le déroulement de l'explication avec le source de l'unité **DescenteRecursive.pas** (voir annexe B) à côté de vous, et éventuellement de vous représenter un graphe de contrôle du flux entre les différentes fonctions. Vous trouverez la partie explicitée de ce graphe à la figure 3.3.

Un appel est donc fait de l'extérieur à la fonction **AnalyseSyntaxique**, comme ceci :

```
ArbreSyntaxique := AnalyseSyntaxique('(1 + 2) + 3 + (4)');
```

Le code de la routine **AnalyseSyntaxique** commence par faire démarrer l'analyseur lexical avec cette entrée[3]. Ensuite, il exige de reconnaître le non-terminal `Entree`, qui est le symbole de départ de la grammaire.

Le programme rentre dans la fonction **Entree**, qui elle-même, appelle immédiatement la fonction **Expression**, puis **Terme**, qui finalement appelle **Lexeme**.

Cette dernière ne reconnaît pas le premier lexème `(` comme étant un nombre. Elle renvoie donc **False** à **Terme**, qui teste alors **ExpressionParenthesee**.

Cette fois, la fonction **Lexeme**, appelée par **ExpressionParenthesee**, reconnaît correctement le lexème `(`. L'analyseur syntaxique construit alors le nœud correspondant à ce lexème puis demande à l'analyseur lexical de passer au lexème suivant.

De retour dans la fonction **Terme**, avec un code de retour positif, on entre dans le bloc `if...then`.

[3] Pour rappel, des informations sur l'analyseur lexical utilisé pourront être trouvées en annexe A.

On commence alors à reconnaître un non-terminal `Expression`. À la différence que, cette fois-ci, on l'exige, ce qui signifie que si l'appel à la fonction **Expression** renvoie **False**, c'est qu'il y a erreur syntaxique.

De la même manière que **ExpressionParenthesee** avait reconnu le lexème identificateur, la fonction courante **Expression** reconnaît un nouveau lexème nombre, au travers d'un non-terminal `Terme`.

L'analyse se poursuit ainsi jusqu'à ce que le flux soit rendu à la fonction **Entree**, avec pour lexème t en entrée le lexème $\dashv$. Elle va alors reconnaître complètement le non-terminal `Entree`, et comme c'est le symbole de départ de la grammaire, l'analyse est terminée.

III.2.1.1 Limitations de l'analyse par descente récursive

Nous avons pu voir que la construction d'un analyseur par descente récursive est relativement simple. Mais tout n'est pas si rose.

En effet, il existe de nombreuses grammaires qui posent problème avec les analyseurs par descente récursive.

D'abord, vous aurez remarqué dans le graphe de contrôle du flux de la figure 3.3 qu'il faut plusieurs appels avant d'avancer sur un lexème. Cela représente une perte de temps qui peut être dérangeante. De plus, les tests qui renvoient la valeur **False** sont en quelque sorte un retour en arrière sur les lexèmes, ce qui signifie que l'analyse n'est pas réellement déterministe.

Ensuite, dans la majorité des cas de grammaires usuelles, il est impossible de créer un analyseur correct. Par exemple, toute grammaire récursive à gauche produit inéluctablement un bouclage sans fin sur la procédure de traitement du non-terminal responsable.

Finalement, le traitement des erreurs est quasiment nul. Tout ce que nous sommes en mesure de faire, c'est de signaler qu'on s'attendait à voir un *non-terminal* particulier et qu'on a trouvé un *lexème* incorrect, avant de devoir purement et simplement abondonner la compilation. Ce n'est réellement pas pratique pour l'utilisateur.

III.2.2 Analyse LL(1)

En examinant le comportement de l'analyseur par descente récursive vu précédemment, le lecteur attentif aura remarqué que pour chaque fonction, le résultat ne dépend que d'un seul facteur : le lexème en entrée.

En effet, pour une fonction donnée et un lexème donné, c'est toujours la même suite d'appels récursifs qui est exécutée. Par exemple, l'appel de la fonction **Expression** avec un lexème `Nombre` en entrée renverra toujours **True**, mais avec le lexème `+` elle renverra toujours **False**.

On peut exploiter cela par ce que l'on appelle le *précalcul*. Il s'agit de déterminer *avant* la compilation le résultat de la fonction pour chaque paire non-terminal/lexème en entrée. Plus exactement, on le fait lors de la production du code du compilateur.

Une fois que l'on connaît ces résultats, on peut les exploiter dans les fonctions de non-terminaux, pour qu'elles ne doivent plus essayer chaque choix mais *prédire* immédiatement le bon. Ce serait évidemment un gain de temps considérable, et surtout cela rendrait réellement déterministe notre analyseur.

Le problème, c'est que pour obtenir ces résultats, il faut effectuer des quantités astronomiques de calculs. Bien entendu ce n'est pas le cas pour notre petite grammaire exemple, mais la quantité de calculs à effectuer est fonction exponentielle de la complexité de la grammaire !

Y a-t-il alors une solution ? Oui. Il s'agit de faire calculer ces résultats par un autre programme, que l'on appelle communément *générateur de compilateur*.

Voyons maintenant quelles sont exactement les informations dont nous avons besoin, avant de nous demander comment les obtenir.

III.2.2.1 Informations requises pour une analyse LL(1)

Nous devons donc savoir comment déterminer quel choix devra être choisi plutôt qu'un autre, sur seule base du premier terminal en entrée.

Comment le savoir ? Tout simplement en connaissant l'ensemble des terminaux par lesquels peut commencer un choix donné. En effet, en connaissant cela, il nous suffit de choisir ce choix si le lexème en entrée appartient à cet ensemble. Nous appellerons désormais cet ensemble l'*ensemble des premiers* d'un choix α, et le noterons $\textsc{prem}(\alpha)$.

Cela paraît suffisant... Mais ne l'est malheureusement pas.

En effet, que se passe-t-il si un choix est nullifiable ? Il peut alors ne commencer par aucun terminal. Comment pourrions-nous alors déterminer qu'il faut sélectionner ce choix ?

On se base sur la considération suivante : il faut choisir un choix nullifiable si le lexème en entrée est un lexème qui peut *suivre directement* ce choix, cela revient à dire ceux qui peuvent suivre directement le non-terminal concerné. Cela nécessite de calculer une nouvelle information : l'ensemble des lexèmes qui peuvent suivre un non-terminal donné. Nous appellerons cet ensemble l'*ensemble des suivants* d'un non-terminal N, et le noterons $\textsc{suiv}(N)$.

Restera à savoir si oui ou non un choix donné est nullifiable. Cette donnée peut être obtenue facilement en étendant la notion d'ensemble des premiers, pour lui faire comprendre la chaîne de symboles grammaticaux vide $\mathcal{E}$ si le choix correspondant est nullifiable.

Nous allons maintenant voir comment on peut calculer le contenu de ces deux ensembles.

III.2.2.2 Ensemble des premiers PREM

Voyons tout d'abord comment calculer les $\textsc{prem}(\alpha)$.

Pour chaque choix, s'il commence par un terminal t, c'est facile : $\textsc{prem}(\alpha)$ est le singleton $\{t\}$. Si c'est un non-terminal N, alors le choix α peut commencer par n'importe quel terminal par lequel N peut commencer. Cela nécessite donc d'obtenir un nouveau type d'information : l'ensemble des premiers d'un non-terminal N, que l'on notera bien entendu $\textsc{prem}(N)$.

Afin de travailler de façon plus générique et plus abstraite – et donc plus simple –, nous adopterons également un ensemble des premiers pour les terminaux. Ainsi, il existera aussi l'ensemble des premiers d'un terminal t – noté $\textsc{prem}(t)$ pour changer – qui sera équivalent au singleton $\{t\}$.

Ainsi, $\textsc{prem}(\alpha)$ est tout simplement égal à l'ensemble des premiers de son premier symbole grammatical.

Nous connaissons donc les ensembles des premiers des terminaux, et nous savons que pour

calculer ceux des choix, nous avons encore besoin de ceux des non-terminaux. Voyons donc comment les calculer.

Pour un non-terminal N de la forme :

$$N \rightarrow \alpha|\beta|\gamma$$

PREM(N) est naturellement l'union des ensembles des terminaux par lesquels peuvent commencer les choix α, β et γ de N. Nous avons donc besoin des PREM des non-terminaux pour calculer ceux des choix, mais aussi ceux des choix pour calculer les PREM des non-terminaux.

Cela peut sembler être un cercle vicieux. En fait, beaucoup d'algorithmes dans le domaine de la compilation paraissent être insolvables à cause de cela. Pourtant, ils sont bel et bien réalisables. On appelle ce type d'algorithme des *algorithmes de fermeture transitive*, ou plus simplement *algorithmes de fermeture*.

Ce type d'algorithme possède trois parties : la définition des données, l'initialisation, et les règles de déduction. La définition des données peut être assimilée à la déclaration **var** d'une routine Pascal. Durant l'initialisation, on définit les données de départ. Et les règles de déduction sont les directives qui indiquent comment faire progresser l'ensemble des données vers la solution finale.

Les algorithmes de fermetures ont besoin de données initiales, à partir desquelles calculer des informations supplémentaires. Mais que sont donc nos données initiales, dans ce cas ? Il s'agit des ensembles des premiers des terminaux. Nous savons en effet que pour tout terminal t, PREM(t) = $\{t\}$.

Vous trouverez en figure 3.4 l'algorithme de fermeture qui calcule les ensembles PREM d'une grammaire. Appliqué sur la grammaire de la figure 3.2 (page 19), les ensembles PREM calculés seront tels que montrés sur la figure 3.5.

Vous aurez remarqué qu'une quatrième variante de l'ensemble des PREM a été ajoutée. Il s'agit des PREM des *fins de choix*. Nous aurons en effet besoin de ces informations pour le calcul de l'ensemble des suivants.

III.2.2.3 Ensemble des suivants SUIV

Le calcul de l'ensemble des suivants d'un non-terminal est plus simple. Par ailleurs, on ne doit pas étendre la notion aux terminaux et choix/fins de choix dans ce cas. Entrons donc directement dans le vif du sujet.

Pour calculer les SUIV, nous aurons à nouveau recours à un algorithme de fermeture. Vous trouverez cet algorithme sur la figure 3.6, et le résultat de son application à la grammaire de la figure 3.2 (page 19) à la figure 3.7.

Nous disposons maintenant de toutes les informations nécessaires à la conception d'un analyseur LL(1). Toutefois, nous nous devons d'abord d'étudier certains cas qui rendent impossible l'écriture d'un tel analyseur. Ce sont ce qu'on appelle des *conflits LL(1)*.

Définition des données

1. Un ensemble de lexèmes nommé PREM(t) (resp. PREM(N), PREM(α)) pour chaque terminal (resp. non-terminal, choix) de la grammaire G ;
2. Un ensemble de lexèmes nommé PREM(α) pour chaque fin de choix de G ; nous appelons fin de choix une suite de symboles grammaticaux α – longue de zéro à plusieurs symboles – telle qu'il existe au moins un choix de la forme $A\alpha$ dans la grammaire G.

Initialisation

1. $\forall\ t \in G_T$: PREM(t) = $\{t\}$;
2. $\forall\ N \in G_N$: PREM(N) = $\emptyset$;
3. $\forall\ \alpha$ choix ou fin de choix non vide de G : PREM(α) = $\emptyset$;
4. $\forall\ \alpha$ choix ou fin de choix vide de G : PREM(α) = $\{\mathcal{E}\}$.

Règles de déduction

1. Pour chaque production $N \rightarrow \alpha$ dans G, PREM(N) doit contenir tous les lexèmes de PREM(α) ;
2. Pour chaque choix ou fin de choix α de la forme $N\beta$, PREM(α) doit contenir tous les lexèmes de PREM(N), excepté $\mathcal{E}$;
3. Pour chaque choix ou fin de choix α de la forme $N\beta$ telle que PREM(N) contient $\mathcal{E}$, PREM(α) doit contenir tous les lexèmes de PREM(β).

FIG. 3.4 – Algorithme de fermeture pour le calcul des PREM d'une grammaire G

Élement de grammaire	PREM
`lexNombre`	{ `lexNombre` }
`'+'`	{ `'+'` }
`'('`	{ `'('` }
`')'`	{ `')'` }
⊣	{ ⊣ }
`Entree`	{ `lexNombre '('` }
`Expression`	{ `lexNombre '('` }
`Terme`	{ `lexNombre '('` }
`ExpressionParenthesee`	{ `'('` }
`ResteExpression`	{ `'+'` $\mathcal{E}$ }

FIG. 3.5 – Ensembles PREM finaux pour la grammaire de la figure 3.2

Définition des données

1. Un ensemble de lexèmes nommé SUIV(N) pour chaque non-terminal de la grammaire G ;
2. Les ensembles PREM de la grammaire G.

Initialisation

1. Calculer les ensembles des PREM au moyen de l'algorithme de la figure 3.4 ;
2. $\forall\ N \in G_N$: SUIV(N) $= \emptyset$.

Règles de déduction

1. Pour chaque production $M \rightarrow \alpha N \beta$, SUIV($N$) doit contenir tous les lexèmes de PREM(β), excepté $\mathcal{E}$;
2. Pour chaque production $M \rightarrow \alpha N \beta$ telle que $\mathcal{E} \in$ PREM(β), SUIV(N) doit contenir tous les lexèmes de SUIV(M).

FIG. 3.6 – Algorithme de fermeture pour le calcul des SUIV d'une grammaire G

Non-terminal	SUIV
`Entree`	`{}`
`Expression`	`{ ')' ⊣ }`
`Terme`	`{ '+' ')' ⊣ }`
`ExpressionParenthesee`	`{ '+' ')' ⊣ }`
`ResteExpression`	`{ ')' ⊣ }`

FIG. 3.7 – Ensembles SUIV finaux pour la grammaire de la figure 3.2

III.2.2.4 Conflits LL(1)

Nous avons vu comment on pouvait décider quel choix devait être sélectionné. Il faut prendre celui qui contient dans son ensemble PREM le lexème t en entrée, ou $\mathcal{E}$ si le lexème t est dans l'ensemble SUIV du non-terminal en cours.

Mais que se passe-t-il si plusieurs choix remplissent les conditions ? Dans ce cas il y a *conflit LL(1)*.

Il y aura conflit LL(1) si les choix α et β d'un non-terminal N sont tels que :

- $\text{PREM}(\alpha) \cap \text{PREM}(\beta) \neq \emptyset$ (conflit PREM-PREM) ;
- $\mathcal{E} \in \text{PREM}(\alpha)$ et $\text{PREM}(\beta) \cap \text{SUIV}(N) \neq \emptyset$ (conflit PREM-SUIV) ;
- $\mathcal{E} \in \text{PREM}(\alpha)$ et $\mathcal{E} \in \text{PREM}(\beta)$ (conflit SUIV-SUIV).

Une grammaire qui ne contient aucun de tels conflits est une grammaire LL(1). On ne peut écrire d'analyseurs LL(1) que pour les grammaires qui sont LL(1).

L'ennui, c'est que la plupart des grammaires ne sont pas LL(1). Il convient, pour créer un analyseur syntaxique LL(1), de supprimer ces conflits en modifiant la grammaire du langage.

Cela dépassant le cadre de ce travail, je vous renvoie une fois de plus au livre *Compilateurs* [GBJL02, section 2.2.4.3].

III.2.2.5 Analyse prédictive par descente récursive

Voilà, nous disposons enfin de toutes les informations nécessaires à la construction d'un analyseur LL(1). Afin de vous permettre d'avancer progressivement, et donc de vous faciliter la compréhension, nous étudierons d'abord l'analyse prédictive par descente récursive. Elle est aussi proche de l'analyse non-prédictive tant par son fonctionnement que par son nom.

En fait, elle fonctionne exactement de la même façon, si ce n'est qu'on n'effectue plus un test de type `if` sur le résultat des fonctions récursives, mais bien une instruction de type `case of` sur le lexème en entrée.

Du point de vue de l'exécution, c'est évidemment beaucoup plus rapide. Et du point de vue de la maintenance et de la compréhension du code, c'est également beaucoup plus clair. On aura donc toujours avantage à utiliser une méthode LL(1) par rapport à la méthode entièrement manuelle.

Vous trouverez le code d'un analyseur prédictif par descente récursive pour la grammaire de la figure 3.2 (page 19) dans le fichier **Predictif.pas**.

Ainsi que vous pouvez le constater, la structure générale est totalement conservée : on utilise toujours une routine par non-terminal et une générique pour les terminaux. La différence, c'est qu'elles ne renvoient plus un booléen témoin de leur succès ou de leur échec. En effet, on sait, dès lors qu'on entre dans une routine de non-terminal (ou même d'un terminal) que ce non-terminal est le bon – sauf s'il y a erreur syntaxique.

D'autre part, cela nous permet d'apporter une amélioration notoire. Puisque l'on sait qu'on étudie le bon non-terminal, on peut déjà construire le nœud correspondant, et le transmettre aux routines appelées récursivement en tant que parent du nœud qu'elles devront construire. Cela permet non seulement de simplifier la libération propre des objets en cas d'erreur syntaxique (remarquez que les blocs `try`...`except`...`end` sont plus élégants) ; mais cela permet aussi de

	lexNombre	'+'	'('	')'	⊣
Entree	Expression		Expression		
Expression	Terme ResteExpr∼		Terme ResteExpr∼		
Terme	Identificateur		ExpressionParenth∼		
ExpressionParenth∼			'(' Expression ')'		
ResteExpr∼		'+' Expression		$\mathcal{E}$	$\mathcal{E}$

FIG. 3.8 – Table de transition LL(1) pour la grammaire de la figure 3.2

posséder directement des informations de contexte qui pourront être utiles à une analyse sémantique combinée à l'analyse syntaxique. Un avantage indéniable pour les compilateurs étroits.

Finalement, on peut maintenant utiliser la valeur de retour des fonctions pour une information bien plus intéressante qu'un code de réussite, à savoir le nœud construit. Cela permet de simplifier également l'écriture des blocs de choix.

Vous voyez donc que l'analyse prédictive par descente récursive ne possède que des atouts par rapport à l'analyse non-prédictive. Aussi, je vous déconseille de jamais construire un analyseur non-prédictif professionnel.

III.2.2.6 L'automate à pile LL(1) : l'analyse prédictive non-récursive

Courage... Il ne nous en reste plus qu'un ! L'analyseur prédictif non-récursif.

Quelles différences avec l'analyseur récursif ? D'abord, c'est que ce petit dernier range tous les résultats calculés précédemment dans une table constante, plutôt que de les exploiter au travers d'un `case of`. Cette table est à deux dimensions : l'une indexée par des non-terminaux, l'autre par des lexèmes. L'autre différence majeure réside dans le fait que nous n'avons plus une routine par non-terminal, mais bien une seule routine concentrant tout l'algorithme en elle-même.

Afin de mieux vous repérer, vous trouverez en figure 3.8[4] ce que l'on appelle la *table de transition* de la grammaire de la figure 3.2 (page 19).

Cette table indique, pour chaque non-terminal actif et terminal en entrée, le choix du non-terminal à sélectionner. Lorsqu'une case est vide, cela signifie qu'il y a erreur syntaxique.

On peut se représenter les non-terminaux comme les *états* d'un automate d'états, et les terminaux en entrée comme les *événements extérieurs* de cet automate. Par exemple, lorsqu'on est dans l'état `Entree`, et qu'un terminal `'('` se présente dans l'entrée, on passe dans l'état `Expression`.

Cependant, ce n'est pas aussi évident qu'un automate d'états fini. En effet, lorsqu'on se trouve dans l'état `ResteExpression` et qu'on reçoit un lexème `'+'` en entrée, on transite vers une suite d'états ! Mieux encore, si l'on reçoit un lexème `')'`, on va à une chaîne vide d'états...

C'est pourquoi nous aurons besoin de ce que l'on appelle un *automate à pile*. Un tel automate est constitué d'une pile d'états. Lors d'un cycle de l'automate, on dépile le sommet de la pile (qui est un état) et, selon son type et le contenu de la table de transition, on empile les composantes du choix sélectionné *dans l'ordre inverse* (pour les dépiler ensuite dans le bon ordre). Lorsqu'on termine un choix, on effectue une sorte de mouvement-$\mathcal{E}$ pour remonter au non-terminal parent.

[4] Les noms longs des non-terminaux ont été tronqués pour faire tenir le tableau dans la largeur de la page.

Tout cela doit vous sembler bien abstrait et peu compréhensible... Ne vous en faites pas, je suis de votre avis. Aussi je vous propose de découvrir sans plus tarder le code de cet automate, que nous allons expliquer ensuite. Vous pourrez trouver ce code dans le fichier **AutomateAPile.pas** (voir annexe B).

La routine maîtresse de ce source est bien entendu la routine **AnalyseSyntaxique**. Nous la reproduisons sur la figure 3.9. Voyons son comportement pas à pas.

Tout d'abord, à l'instar de nos deux premiers analyseurs syntaxiques, celui-ci démarre l'analyse lexicale (ligne 122).

Ensuite, on crée la pile prédictive (ligne 126). On transmet en paramètre du constructeur le symbole de départ de la grammaire : il sera empilé originellement.

De la ligne 128 à la ligne 156, on trouve la boucle principale de l'algorithme. Cette boucle remplace en réalité les appels récursifs des analyseurs précédents. Elle ne s'arrête que sur erreur syntaxique ou lorsque la pile prédictive est vide. Cette dernière situation signifie que l'entrée est complètement analysée.

L'algorithme répétitif effectue la tâche suivante : il dépile le sommet de la pile prédictive et, selon son type, effectue une des trois actions suivantes.

Si le sommet de la pile est un code de retour au parent (ligne 133), alors on redéfinit le nœud courant comme étant le parent de l'ancien. Il s'agit du mouvement-$\mathcal{E}$ dont nous avions parlé plus haut. D'autres implémentations de ce mouvement peuvent être faites. Par exemple, si l'on code de manière à ce qu'un nœud de non-terminal puisse indiquer lorsque son choix est complet, on peut se baser sur cette information pour remonter au parent.

Si le sommet de la pile est un terminal (ligne 137), il faut simplement reconnaître ce terminal et l'ajouter aux fils du nœud courant.

Si le sommet de la pile est un non-terminal (ligne 145), on effectue un mouvement de prédiction. Voici les différentes instructions qui sont effectuées.

1. On commence par consulter la table de transition pour déterminer le choix à sélectionner, en fonction du non-terminal récupéré en pile et du lexème en entrée (ligne 146).
2. On crée ensuite le non-terminal indiqué en pile et on signale son choix (ligne 147). Remarquez au passage que nous utilisons désormais un type générique pour les non-terminaux. Il serait en effet impensable d'utiliser ici un `case of` pour déterminer son type. Si l'on construit un analyseur étroit, ce peut être gênant. On recourra alors à des tables de correspondance entre un non-terminal et sa classe et on utilisera un constructeur polymorphique.
3. La ligne 149 constitue une action un peu barbare : on assigne à **Result** le premier non-terminal rencontré. Cela marche effectivement, mais c'est peu joli. Une programmation propre requerrait ici un code plus lourd et donc plus lent. On préfère ainsi conserver cette impureté informatique...
4. L'instruction suivante relie le nouveau nœud au nœud courant, qui est son parent (ligne 151).
5. Après cela, on définit le nouveau nœud comme le nœud courant (ligne 153).
6. Finalement, on empile les différents symboles du choix sélectionné sur la pile (ligne 154). Notez que les procédures **EmpileChoix*i*** empilent d'abord un code de retour au parent, qui est la fin du choix, puis tous les symboles *dans l'ordre inverse* sur la pile. Et ce pour les dépiler plus tard dans le bon ordre.

```
function AnalyseSyntaxique(Entree : string) : TNoeudEntree;
var Courant, Nouveau : TNoeudNonTerminal;
    NouveauLexeme : TNoeudLexeme;
    Symbole : Char;
    Choix : integer;
begin
  DemarrerAnalyseurLexical(Entree);
  Result := nil;
  try
    Courant := nil;
    Pile := TPilePredictive.Create(ntEntree);
    try
      while not Pile.Empty do
      begin
        Symbole := Pile.Pop;

        if Symbole = codeRetourneAuParent then
        begin // le choix courant est terminé
          Courant := Courant.Parent;
        end else
        if Symbole < ntEntree then
        begin // la prédiction est un terminal -> reconnaissance
          if CurLex.Classe <> Symbole then
            ErreurSyntaxique;
          NouveauLexeme := TNoeudLexeme.Create;
          NouveauLexeme.Lexeme := CurLex;
          Courant.AjouteFils(NouveauLexeme);
          LexemeSuivant;
        end else
        begin // la prédiction est un non-terminal -> prédiction
          Choix := TableTransition[Symbole, CurLex.Classe];
          Nouveau := TNoeudNonTerminal.Create(
            Courant, Symbole, ChoixDuNonTerminal[Symbole](Choix));
          if not Assigned(Result) then
            Result := Nouveau;
          if Assigned(Courant) then
            Courant.AjouteFils(Nouveau);
          Courant := Nouveau;
          EmpileChoix[Choix];
        end;
      end;
    finally
      Pile.Free;
    end;
  except
    if Assigned(Result) then
      Result.Free;
    raise;
  end;
end;
```

FIG. 3.9 – Algorithme principal de l'analyseur prédictif non-récursif

Notez enfin la clause `except` (ligne 160) qui doit libérer les nœuds construits en cas d'erreur. La seule libération du nœud **Result** suffit à libérer tous les nœuds construits. En effet, dans cet algorithme, le nœud **Result** est le premier construit et dès qu'un nouveau nœud est créé, il est directement relié à son parent et son parent à lui. Ainsi, comme chaque parent détruit les enfants qui lui sont reliés à sa libération, une seule libération entraîne toutes les autres. Rien de plus simple...

III.3 Pour aller plus loin...

Ici s'achève notre tour d'horizon des analyseurs syntaxiques descendant. Bien entendu la notion n'a pas été étudiée jusqu'au bout. On peut encore et toujours approffondir le sujet, autant qu'on le désire.

Les lecteurs dont la soif de connaissance n'aura pas été apaisée pourront se référer au livre *Compilateurs* [GBJL02], comme toujours.

Par exemple, ils y trouveront des informations complémentaires (mais non complètes, étant donné que ce sujet est toujours à l'étude) sur la gestion efficace des erreurs. Ils y apprendront différentes techniques permettant de remettre sur ses rails un analyseur qui a déraillé suite à une erreur syntaxique.

D'autre part, ils trouveront également dans cet ouvrage de référence une introduction à l'utilisation du programme **LLgen**, qui est le générateur d'analyseurs syntaxiques le plus connu. Ce logiciel reçoit une définition formelle d'une grammaire et engendre complètement le texte d'un programme en C, qui, compilé, donnera un analyseur syntaxique complet pour cette grammaire.

Chapitre IV

Analyse ascendante

Ainsi que nous l'avons indiqué plusieurs fois, nous n'étudierons pas en détail l'analyse ascendante. Nous nous contenterons d'en donner le principe général. Ceci pour rester dans les limites acceptables d'un travail de rhétorique.

IV.1 Principe général

Dans un analyseur ascendant, le paradigme est tout autre. On construit les sous-arbres syntaxiques qui englobent les premiers lexèmes de l'entrée.

Au départ, on ne connaît aucun nœud à construire. On ne connaît que le premier lexème t_1 en entrée.

On le reconnaît comme étant un nœud feuille (puisque c'est un terminal) et on construit celui-ci. On avance ensuite sur le lexème t_2, dont on construit aussi le nœud feuille correspondant.

À partir de là, si les nœuds n_1 et n_2 correspondant aux lexèmes t_1 et t_2 sont les nœuds fils d'un non-terminal N, alors le nœud correspondant n_3 est créé puis relié à ses fils n_1 et n_2. Par la suite, ce nœud n_3 pourra lui-même être fils d'un autre non-terminal.

Si ce n'est pas la cas, on continue d'avancer sur les lexèmes t_i jusqu'à ce que les n derniers lexèmes soient les composantes d'un choix d'un non-terminal N, auquel cas on procède à la création du nœud correspondant que l'on relie à ses fils.

On continue ainsi jusqu'à ce que toutes les composantes d'un choix du symbole de départ S de la grammaire soient créées. On construit alors le nœud racine r correspondant au symbole de départ S, et on le relie à ses fils. L'analyse est alors terminée et l'arbre est construit.

La figure 4.1 montre une situation dans laquelle un analyseur ascendant a avancé sur les lexèmes t_1 à t_4 de l'entrée, tout en construisant leurs nœuds correspondants n_1 à n_4, puis, ayant reconnu les nœuds t_2, t_3 et t_4 comme les trois composantes d'un non-terminal N, a construit le nœud n_5 correspondant et l'a relié à ses trois fils, et finalement a avancé sur le lexème t_5 et a construit son nœud correspondant n_6.

Les carrés noirs représentent les nœuds de l'arbre déjà construits. Les cercles symbolisent les lexèmes, en noir ceux déjà lus et reconnus et en blanc ceux qui restent dans l'entrée.

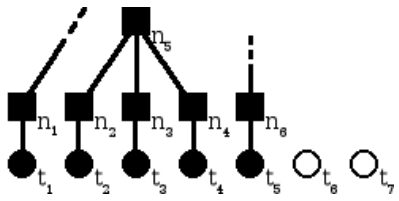

FIG. 4.1 – Exemple de situation d'un analyseur ascendant

On remarque que, contrairement à la figure 3.1 (page 18), on ne trouve pas ici de carré blanc, représentant les nœuds non construits mais dont on connaît l'existence. En effet, dans l'analyse ascendante, on ne connaît l'existence d'un nœud qu'au tout dernier moment, et on le construit alors immédiatement.

IV.2 Vous cherchez plus d'informations ?

Si vous êtes avide de plus de détails sur la méthode ascendante, je ne puis que trop peu vous conseiller de vous reporter au livre *Compilateurs* [GBJL02, section 2.2.5]. Sachez aussi que ce document fera certainement plus tard l'objet d'une extension sur la page Web suivante : http://sjrd.developpez.com/algorithmique/compilation/analyseurs-syntaxiques/

Chapitre V

Projet exemple

Nous avons donc étudié les analyseurs syntaxiques sous beaucoup d'angles. Cependant, nous n'avons encore étudié que des petites parties isolées de leur fonctionnement.

C'est pourquoi j'ai développé un projet exemple. C'est un ensemble de packages et de programmes Delphi qui permet d'analyser un code source *Extra Simple Pascal* et d'en donner la structure au format XML.

Le langage que j'ai appelé *Extra Simple Pascal* est une simplification à l'extrême du langage Pascal. Les deux seuls types de variables autorisés sont `integer` et `string`. Les seules opérations utilisables sont `+ - * /` et le parenthésage. Enfin, il n'existe que l'affectation, l'appel de routine et l'impression à l'écran comme instructions. Le fichier **Exemple de source.txt** de l'archive indiquée ci-après présente un exemple de code source valide de ce langage.

Vous pourrez trouver sur la figure 5.1 la grammaire de ce mini-langage. Bien entendu, celle-ci a été écrite de façon à être LL(1).

Vous pourrez télécharger les sources complètes *via* ce lien :
`ftp://ftp-developpez.com/sjrd/tutoriels/compilation/analyseurs-syntaxiques/projet.zip`

Si votre configuration informatique vous empêche d'utiliser le FTP, utilisez le miroir HTTP suivant :
`http://sjrd.ftp-developpez.com/tutoriels/compilation/analyseurs-syntaxiques/projet.zip`

Ces sources ont été développées avec Delphi 2005 édition Architecte, en Delphi Win32. Normalement, elle devraient être compilables directement avec n'importe quelle édition de Delphi 2005 et Delphi 2006, et devraient également se compiler sans problèmes avec les versions antérieures.

Les cinq projets inclus dans le groupe de projet de ce .zip sont les suivants :

AnalyseurCommon.bpl : contient la définition des classes d'analyse et l'analyseur lexical ;

AnalyseurDescendant.bpl : contient une unité commune aux trois types d'analyse descendante, ainsi qu'une unité pour chacun de ces trois analyseurs ;

ProjetTest.exe : permet de tester l'analyse lexicale et les trois types d'analyse descendante implémentés dans les deux précédents packages ;

Generateurs.bpl : fournit de nombreuses classes pour le chargement des grammaires, leur ges-

```
          Source  →  Routines CorpsRoutine '.' ⊣
        Routines  →  Routine Routines | ℰ
         Routine  →  Procedure | Fonction
       Procedure  →  'procedure' Identificateur Parametres ';'
                     CorpsRoutine ';'
        Fonction  →  'function' Identificateur Parametres TypeVariable ';'
                     CorpsRoutine ';'
      Parametres  →  '(' DefVariable SuiteParametres ')' | ℰ
  SuiteParametres →  ';' DefVariable SuiteParametres | ℰ
     DefVariable  →  Identificateur TypeVariable
    TypeVariable  →  ':' Identificateur
    CorpsRoutine  →  DefVariables Bloc
    DefVariables  →  'var' DefVariable ';' SuiteDefVariables | ℰ
SuiteDefVariables →  DefVariable ';' SuiteDefVariables | ℰ
            Bloc  →  'begin' Instruction SuiteInstructions 'end'
SuiteInstructions →  ';' Instruction SuiteInstructions | ℰ
     Instruction  →  Bloc | Print | IdentInstr | ℰ
           Print  →  'print' Expression
      IdentInstr  →  Identificateur SuiteIdentInstr
  SuiteIdentInstr →  Affectation | Augmentation | Diminution | Params
     Affectation  →  ':=' Expression
    Augmentation  →  '+=' Expression
     Diminution   →  '-=' Expression
          Params  →  '(' Expression SuiteParams ')' | ℰ
     SuiteParams  →  ',' Expression SuiteParams | ℰ
      Expression  →  Terme SuiteExpression
  SuiteExpression →  '+' Expression | '-' Expression | ℰ
           Terme  →  Facteur SuiteTerme
      SuiteTerme  →  '*' Terme | '/' Terme | ℰ
         Facteur  →  Identificateur Params | Nombre | Chaine
                  |  '(' Expression ')'
```

FIG. 5.1 – Grammaire du mini-langage *Extra Simple Pascal*

tion, et surtout les calculs des PREMet SUIV, des tables de transition, etc. ;

Generateur.exe : fournit une interface avec l'utilisateur pour utiliser les classes du package précédent.

Trois grammaires sont fournies avec les sources (fichiers .gra). L'une représente la grammaire exemple de la figure 3.2 (page 19). L'autre représente la grammaire du langage *Extra Simple Pascal*. La troisième est une grammaire qui provoque les trois types de conflits LL(1) étudiés en section III.2.2.4.

Vous pouvez ouvrir ces fichiers dans le bloc-notes de Windows pour lire leur contenu, mais surtout les charger avec le programme **Generateur.exe** pour récupérer plus d'informations sur elles que vous ne pouvez l'imaginer...

Ceci dit, ce qui est le plus intéressant par rapport à ce que nous avons étudié tout au long de ce document, ce sont bien les trois unités responsables de l'analyse syntaxique selon les trois méthodes vues, ainsi que l'unité qui définit les classes d'analyse.

Je ne vous en dis pas plus ici et vous laisse le soin de découvrir ces sources, muni du bagage de connaissances que vous avez rassemblées lors de la lecture de ce travail.

Chapitre VI

Conclusion

Nous voici donc arrivés au terme de ce travail sur les analyseurs syntaxiques.

Nous avons commencé par introduire la compilation au sens général, et le fonctionnement global d'un compilateur. Nous avons également vu la notation BNF pour les grammaires.

Ensuite, nous avons commencé à nous concentrer sur un module des compilateurs : l'analyse syntaxique. Ce module qui a la responsabilité d'organiser les lexèmes du texte source en un arbre abstrait.

Nous avons après cela étudié en détails les différentes techniques d'analyse syntaxique descendante. Rappelons que nous n'avons pas couvert le sujet à 100 % : nous nous sommes concentrés sur l'essentiel et n'avons pas approché les techniques avancées que sont celles de la récupération d'erreurs par exemple.

Nous avons également vu le principe général de l'analyse ascendante, mais ne nous sommes pas intéressés à son comportement exact. Encore une fois, ce choix a été imposé par les limites de longueur des travaux de rhétorique, qui ont déjà été largement dépassées pour ce travail. Sachez toutefois que ce travail sera plus que probablement étendu dans le futur, et disponible sur la page Web suivante :
http://sjrd.developpez.com/algorithmique/compilation/analyseurs-syntaxiques/

Finalement, vous avez été invité à examiner un projet d'analyse syntaxique fonctionnel, utilisant les trois techniques d'analyse descendante étudiées.

Vous trouverez encore en annexe A des informations sur le fonctionnement de l'analyseur lexical qu'ont utilisé nos codes exemples.

L'annexe B rappelle les différents fichiers sources exemples que vous avez été invité à consulter lors de l'étude de ce travail.

Pour la dernière fois, je renvoie encore les lecteurs non rassasiés au livre *Compilateurs* [GBJL02], qui est sans aucun doute *la* référence en matière de compilation.

J'espère que ce document vous aura permis de comprendre les techniques essentielles de l'analyse syntaxique, et que vous l'aurez apprécié.

Annexe A

Analyseur lexical utilisé

L'analyseur lexical que nous avons utilisé dans nos exemples fonctionne de façon très simple, vu de l'extérieur. Il n'est que virtuel, et n'est pas réellement implémenté. Il est juste là pour pouvoir comprendre son intégration dans l'analyseur syntaxique.

Voici donc la partie `interface` de l'unité virtuelle **AnalyseurLexical**, à laquelle nous faisons référence dans les analyseurs syntaxiques exemples.

```
unit AnalyseurLexical;

interface

type
  TLexeme = record
    Lig, Col : integer;
    Classe : Char;
    Repr : string;
  end;

procedure DemarrerAnalyseurLexical(Entree : string);
procedure LexemeSuivant;

var
  CurLex : TLexeme;
```

Le type **TLexeme** définit les informations que contient chaque lexème. Ses deux propriétés essentielles sont **Classe** et **Repr**, puisque ce sont les deux seules qui sont strictement nécessaires du point de vue de l'analyse. On peut même exagérer, en supprimant la propriété **Repr**, puisque, pour l'analyse syntaxique, seule la propriété **Classe** importe. Les propriétés **Lig** et **Col** sont présentes uniquement afin de pouvoir fournir des messages d'erreurs plus sophistiqués.

La procédure **DemarrerAnalyseurLexical** doit être appelée en premier. Elle reçoit l'entrée à analyser et analyse le premier lexème. Elle stocke les informations sur ce lexème dans la variable **CurLex**.

Ensuite, pour avancer sur un lexème et analyser le suivant, il suffit d'appeler la procédure

LexemeSuivant. À nouveau, les données sur le lexème analysé sont stockées dans la variable **CurLex**.

Il est inutile et hors-sujet d'expliquer ici l'implémentation de cet analyseur. Si vous désirez plus d'informations sur l'analyse lexicale, je vous renvoie au tutoriel d'Olivier Lance [LAN04] ainsi qu'au livre *Compilateurs* [GBJL02, section 2.1]. Vous pouvez aussi vous intéresser au code de l'analyseur lexical du projet de test.

Annexe B

Référentiel des fichiers sources

Ce document est accompagné de quelques fichiers sources, qui servent à illustrer les explications données. Tous ces fichiers sources peuvent être trouvés dans le dossier FTP suivant :
ftp://ftp-developpez.com/sjrd/tutoriels/compilation/analyseurs-syntaxiques/

Si vous n'avez pas la possibilité d'utiliser le FTP sur votre ordinateur, utilisez le miroir HTTP suivant :
http://sjrd.ftp-developpez.com/tutoriels/compilation/analyseurs-syntaxiques/

Vous pouvez télécharger chacune de ces sources séparément, selon vos intérêts, ou vous pouvez télécharger directement l'ensemble des sources proposées au moyen de ce fichier .zip :
ftp://ftp-developpez.com/sjrd/tutoriels/compilation/analyseurs-syntaxiques/sources.zip (FTP)
http://sjrd.ftp-developpez.com/tutoriels/compilation/analyseurs-syntaxiques/sources.zip (HTTP)

DescenteRecursive.pas *19*

Predictif.pas *26*

AutomateAPile.pas *28*

Glossaire

fermeture (algorithme de) 23, **23**
Famille d'algorithmes caractérisés par le départ d'une petite quantité de données et allant en progressant, en agrandissant ces données à partir des données déjà calculées, et ce jusqu'à ce que plus aucune nouvelle donnée ne soit trouvée.

lexème **6**, 11, 14
La plus petite entité significative de caractères dans un langage source.

non-terminal **11**
Utilisé dans la terminologie des grammaires ; est un groupe cohérent sémantiquement de symboles grammaticaux.

premiers (ensemble des) PREM **22**
Ensemble des lexèmes par lesquels peut commencer un non-terminal, un terminal, un choix ou une fin de choix donné. Si l'élément grammatical en question est nullifiable, cet ensemble contient $\mathcal{E}$.

programme 4, 9
Dans ce document, nous entendons par « programme » un code compilé exécutable, autrement dit un fichier .exe. Je précise cela car on peut prendre le terme dans le sens de code source également.

sémantique 4
La sémantique d'un fichier est sa *signification*, peu importe la façon dont elle est décrite. En programmation, la sémantique d'un programme est son comportement – comportement écrit au moyen d'un langage de programmation et traduit en code exécutable par un compilateur.

suivants (ensemble des) SUIV **22**
Ensemble des lexèmes qui peuvent suivre directement un non-terminal donné.

symbole grammatical **10**
Élement de base de la définition d'une grammaire ; ses deux types sont les terminaux et les non-terminaux.

syntagme **12**, 13, 18
Sous-arbre d'un arbre abstrait dont les feuilles peuvent être des non-terminaux. L'action de *dériver* un syntagme signifie remplacer l'un des non-terminaux qui composent ses feuilles par un nouveau nœud dont les fils représentent un choix de ce non-terminal. Une dérivation d'un syntagme est le résultat de l'application répétée zéro ou plusieurs fois de cette action.

terminal **11**, 17
Utilisé dans la terminologie des grammaires, signifie un lexème.

Bibliographie

[BELA98] Michel Beaudouin-Lafon, *Lexique et syntaxe, site Web personnel, 1998*
http://www.lri.fr/~mbl/ENS/DEUG/cours/3-lexique-syntaxe.html

[GBJL02] Grune, Bal, Jacobs et Langendoen, *Compilateurs, Dunod, 2002*
http://www.eyrolles.com/Informatique/Livre/9782100058877/

[LAN04] Oliver Lance, *Lexers 1/2 : Théorie, www.developpez.com, 2004*
http://olance.developpez.com/articles/delphi/lexers-theorie/

[LAN05] Olivier Lance, *Tutoriels sur le format PE, www.developpez.com, 2005*
http://olance.developpez.com/articles/windows/pe-iczelion/

[PER06] Romuald Perrot, *Introduction aux arbres, www.developpez.com, 2006*
http://rperrot.developpez.com/articles/algo/structures/arbres/

Table des figures

Index