LANGAGES, GRAMMAIRES, ANALYSES, COMPILATION
UNE INTRODUCTION
COURS/TD/TP

JACQUES ROUILLARD

Illustration de couverture : voir note 2 page 16

Table des Matières

1 Grammaires et classifications [cours/TD]

1.1 Définition

Pour définir une grammaire, nous allons utiliser des quadruplets. {Vt, Vn, Ax, R}.

- Vt est le vocabulaire terminal, c'est-à-dire la liste des mots qui apparaissent dans le langage. En informatique, cela peut être des mots «figés » comme **if** ou **return**. Ou des mots porteurs de valeur, comme 123 (c'est un mot, un entier portant la valeur 123) ou Toto (c'est un identificateur symbolisant quelque chose). En Français, ce sont les mots du dictionnaire.

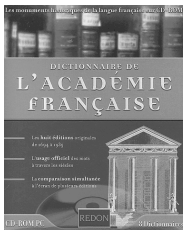


Figure 1 Dictionnaire

- Vn est la liste des concepts du langage, ceux qui se définissent les uns les autres. On dira « non-terminal ». Ainsi en Français, la notion de « groupe nominal » peut être définie comme « *article adjectif substantif*» (le beau garçon) ou « *article substantif* » (la fleur) etc. En C, l'*instruction_if* peut être définie (très sommairement et entre autres) par « **if (** *expression* **)** *instruction* **;** ». En Français, ce sont donc les concepts définis dans une grammaire.

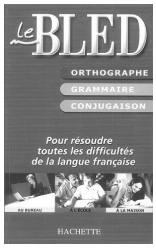


Figure 2 Grammaire

- R est l'ensemble des règles qui définissent les éléments de Vn en fonction des éléments de Vn+Vt. On voit ci-dessus que l'*instruction*_if est définie par des mots réservés, éléments de Vt « (« if », « (» , «) », « ; ») et des éléments de Vn (« *expression* », « *instruction* »). La règle vide est appelée ε.

- Ax est l'axiome, **la** règle particulière par laquelle on entre dans le langage. Par exemple « programme ». En Français, c'est la désignation de l'entrée du texte que l'on va lire.

On voit que le travail de définition est essentiellement dans la rédaction des règles, impliquant la liste de Vn. On voit aussi que le même langage peut être décrit par une infinité de grammaires différentes.
Inversement, la plupart des langages sont infinis (acceptent une infinité de phrases), mais l'infinité de grammaires qui peuvent décrire chaque langage sont, chacune, finie. Cela est vrai en Français comme en informatique : quiconque a des enfants scolarisés voit que l'enfant apprend le Français avec des concepts qui n'existaient pas 10 ans avant. Or c'est (en principe) la même langue.

Les règles peuvent donner à la grammaire des propriétés, suivant (entre autres)

- qu'elle est définie récursivement ou non.
- que la définition d'une règle dépende de son contexte. Par exemple, en C, l'instruction `break` n'est pas acceptée partout.

1.2 Notation

On écrit une règle en décrivant deux parties: la gauche est la partie définie, la droite est la partie définissante. On utilise la flèche "→" en général et le symbole "**::=**" dans la notation

dite BNF que l'on emploie généralement pour les grammaires hors contexte, ou simplement "**:**" dans le formalisme de *yacc* qui est utilisé par la majorité des outils. On dira qu'une règle est **dérivée** quand on remplace sa partie gauche par sa partie droite, et **réduite** quand on remplace la partie droite par sa partie gauche.
Par exemple, soit la règle
`Achille → héros au pied léger` (πόδας ὠκὺς Ἀχιλλεύς)

On la dérive quand on dit "J'ai vu Achille, un héros au pied léger".
On la réduit quand on dit: "Arrive le héros au pied léger, j'ai nommé Achille."

1.3 Classification

Les grammaires sont classées (classification dite de *Chomsky-Schützenberger* [1]), de 0 -très général- à 4 -très restrictif-. Chaque classe est incluse dans la précédente (inclusion stricte). Dans les exemples suivants, les lettres grecques représentent des séquences d'éléments de vocabulaire, terminal, non-terminal ou séquence quelconque des deux. Les minuscules des éléments du vocabulaire terminal (les mots), les majuscules des éléments du vocabulaire non terminal (les noms des règles).

- Type 4 parfois mentionné et d'usage quasi-nul, pour lequel chaque non-terminal est défini comme étant exactement un et un seul terminal.

- Type 3 : les grammaires régulières, que l'on peut reconnaître avec un automate d'états finis (sans pile). Les règles sont ou peuvent se ramener à des formes pour lesquelles la partie gauche est un non-terminal unique, et la partie droite commence ou se termine par un terminal. `A → aB` (ou `A → Ba`) et `A→ a` ; dans le premier cas on la dit linéaire droite, dans le second cas linéaire gauche.
  Exemple, la description de la règle « ENTIER » qui acceptera tous les entiers composés de plusieurs chiffres, où « chiffre » est ici vu comme un élément terminal :
  ```
  ENTIER → chiffre ENTIER
  ENTIER → chiffre
  ```
  L'entier 123 se dérivera ainsi (on met entre parenthèse la portion de la phrase qu'on a substituée, et en gras souligné celle qu'on va dériver) :
  ENTIER (1**ENTIER**) 1(2**ENTIER**) 12(3)

1

Noam Chomsky (USA, 1928-) est un professeur de linguistique qui s'est illustré par ses travaux sur les grammaires, les langages naturels et la psychologie cognitive. Par ailleurs il est connu comme anarcho-libertaire, et émet des critiques politiques virulentes contre les engagements de son pays. C'est un auteur très souvent cité dans beaucoup de domaines.

Marcel-Paul Schützenberger (France, 1920-1996), double doctorat en médecine – généticien, épidémiologiste- et en mathématiques, domaine dans lequel il s'est fait connaître par ses travaux en théorie des langages et des systèmes combinatoires. Par ailleurs il a développé des arguments âprement controversés contre le darwinisme. Il a inspiré le *Dr. Schütz* de "*Et on tuera tous les affreux*" de *Boris Vian*, qui était son ami.

- Type 2 : les grammaires hors contexte, que l'on peut reconnaître avec un automate à pile non déterministe (des entrées peuvent activer plusieurs mouvements de l'automate). Les règles acceptées sont telles que l'élément non terminal est défini de façon univoque : $A \rightarrow \gamma$. Donc ici la partie gauche est toujours un non-terminal unique, la partie droite est quelconque. Exemple, un palindrome formé de 0 et de 1 (éventuellement vide)

  ```
  PALIND →   1 PALIND  1
  PALIND →   0 PALIND  0
  PALIND →   ε
  ```

  Le palindrome 101101 va se dériver ainsi (on met entre parenthèse la portion de la phrase qu'on a substituée, et en gras souligné celle qu'on va dériver) :

  PALIND (1**PALIND**1) 1(0**PALIND**0)1 10(1**PALIND**1)01 101(**ε**)101 101101

- Type 1 : les grammaires contextuelles, reconnaissables par une machine de Turing **l**inéairement bornée (son ruban a une longueur bornée) non déterministe. On s'autorise des définitions dans lesquelles l'élément à définir est « encadré » d'un côté, de l'autre ou des deux par un contexte.$\alpha A \beta \rightarrow \alpha\gamma\beta$. Exemple, une grammaire acceptant N « a » puis N « b » puis N « c », N étant quelconque mais le même dans les trois cas.

  ```
  S → aSBC
  S → abC
  CB → BC
  aB → ab
  bB → bb
  bC → bc
  cC → cc
  ```

  On voit ici que les phrases **abc**, **aabbcc** et **aaabbbccc** peuvent se dériver ainsi (on met entre parenthèse la portion de la phrase qu'on a substituée, et en gras souligné celle qu'on va dériver) :

  ```
  abc       : S (abC) a(bc)
  aabbcc    : S (aSBC) a(abC)BC aab(BC)C aa(bb)CC aab(bc)C aabb(cc)
  aaabbbccc : S (aSBC) a(aSBC)BC aa(abC)BCBC aaabCB(BC)C aaab(BC)BCC
                 aaabB(BC)CC aaa(bb)BCCC aaab(bb)CCC aaabb(bc)CC
                 aaabbb(cc)C aaabbbc(cc)
  ```

- Type 0 : toutes les grammaires dites récursivement énumérables reconnaissables par une machine de Turing ; hélas si la reconnaissance se fait en un temps fini, la non-reconnaissance peut demander un temps infini avant d'être diagnostiquée. On s'autorise des règles de définition dans lesquelles la partie droite (définissante) et la partie gauche (définie) sont des phrases de même nature. $\alpha \rightarrow \beta$. Pas d'exemple simple.

Là-dessus, il existe des classes intermédiaires remarquables, par exemple entre 2 et 3 : les langages reconnaissables par des automates à pile déterministes (pour chaque entrée, un seul mouvement est possible). Les générateurs d'analyseurs ne s'attaquent concrètement qu'aux grammaires de types 2 et 3 et intermédiaires.

- Le type 3 est parfaitement adapté pour faire des grammaires reconnaissant les mots d'un langage : l'exemple donné supra (NOMBRE) l'illustre très bien. On peut décrire ainsi ce qu'est un entier (une succession de chiffres), un identificateur (une lettre suivie d'une succession de chiffres ou de lettres), ou, plus long, un réel (une succession de chiffre, un point décimal, puis une succession de chiffres, puis optionnellement une partie exposant avec un signe optionnel, etc etc.) Ce qu'on ne peut pas faire, c'est décrire des symboles « symétriques » nécessitant des règles

récursives (on ne peut pas exiger que les identificateurs soient des palindromes ni vérifier l'équilibre de parenthèses.) On utilisera l'outil LEX ou son clone FLEX.

- Les types 2 et 2.5 sont bien adaptés à l'analyse des langages informatiques modernes. On peut y décrire des constructions récursives (une boucle peut en contenir d'autres) ou symétriques par paires (parenthèses ouvrantes et fermantes en nombre égal). On ne peut pas y décrire des structures dépendant du contexte, par exemple exiger que dans N paires de parenthèses se trouvent N points (((…))). C'est l'exemple proposé plus haut avec abc comme terminaux, qui est de type 1. On utilisera l'outil YACC ou ses clones BYACC ou BISON.

1.4 Exercices

- X → l'homme qui a vu
  X → l'homme qui a vu X
  TOTO → c'est X l'ours et qui n'a pas eu peur.
  L'axiome est TOTO. Quelles sont les phrases du langage?

- Soit la grammaire :
  A → B y
  B → x
  B → ε
  L'axiome est A, les mots sont x et y. Quelles sont les phrases de ce langage ?

- Même question :
  A → B y
  B → x B
  B → ε

- Même question :
  A → B y B
  B → x B
  B → ε

- Aller sur le site :
  http://cui.unige.ch/db-research/Enseignement/analyseinfo/Ada95/BNFindex.html
  Déterminer si l'on peut mettre une clause ***use*** *paquetage* dans la zone déclarative d'une procédure

```
procedure P is
   use pck;
begin
  null ;
end;
```

- Soit la grammaire:

```
expression → terme '+' terme
expression → terme ‘-' terme
expression → terme '*' terme
expression → terme '/' terme
expression → terme
terme → entier
terme → '(' expression ')'
```

  Accepte-t-elle une expression comme 2+3?
  Accepte-t-elle une expression comme 2+(3+4)?
  Accepte-t-elle une expression comme 2+3+4 ?
  Si non, que faire pour qu’elle l’accepte ?
  Si ou quand oui, sera-t-on content avec l’analyse de 2+3*4, sachant qu’on ne peut faire « quelque chose » pour calculer qu’en fin de chaque ligne, quand les opérandes sont calculés ?
  Si non que faire ?

2 Premier, suivant [cours/TD]

On a souvent besoin de deux ensembles de terminaux associés à chaque non-terminal (autrement dit, de mots associés à chaque règle). Prenons la grammaire (récursive à gauche) où *entier* est un terminal.

```
expr → expr '+' terme
expr → expr '-' terme
expr → terme
terme → terme '*' primaire
terme → terme '/' primaire
terme → primaire
primaire → '(' expr ')'
primaire → entier
```

2.1 Premier

C'est l'ensemble des terminaux (les mots) susceptibles de *commencer* la règle considérée dans toutes les phrases possibles du langage. C'est un ensemble qui va servir dans l'analyse récursive descendante, pour ne pas aller explorer une règle inutilement. Il va aussi nous aider à construire SUIVANT, voir ci-après.

Si une des règles définissant A est
`A → x B`
Alors x est dans PREMIER de A.
Si une règle est
`A→ B y`
Alors PREMIER(A) hérite de PREMIER(B) et de y si B contient un choix vide (ε)
Et ceci transitivement:
`A→ BCDz`
Si B et C et D contiennent tous un choix vide, PREMIER(A) contient PREMIER(B), PREMIER(C), PREMIER(D) et z.

Dans notre grammaire exemple, PREMIER(primaire) est *entier*, plus la parenthèse ouvrante. Le calcul dans le cas général demande un algorithme récursif, car si la règle A commence par un non-terminal B, alors elle hérite des premiers de B. Si la chaîne vide ε est une option de B, alors les premiers du suivant de B dans la règle A sont de la partie, etc.
Ici donc, les premiers de `primaire, terme` et `expr` sont les mêmes, puisque tous les débuts de règles se ramènent in fine à primaire qui n'a pas de chaîne vide.

PREMIER(primaire) = { *entier*, (}
PREMIER(terme) = { *entier*, (}
PREMIER(expr) = { *entier*, (}

2.2 Suivant

SUIVANT(X) où X est une règle, est l'ensemble des mots du langage susceptibles d'apparaître à droite de X dans une dérivation possible de la grammaire (c'est-à-dire dans une phrase du langage). C'est un ensemble qui va nous servir dans l'analyse récursive ascendante, pour marquer les endroits où il faut réduire la règle à sa définition.

1. Donc, si une règle dit X -> α Y β, le contenu de SUIVANT(Y) est augmenté de PREMIER(β), sauf ε. Chercher le premier de β peut impliquer d'aller chercher le suivant du suivant (s'il y a une règle vide dans le jeu). Par exemple:
   X -> α Y Z β avec Z qui contient ε va donner pour SUIVANT(Y) l'union de PREMIER(Z) et de PREMIER(β), sauf ε.
2. Si une règle dit X -> α Y (ou équivalent, α Y β avec ε dans PREMIER(β)), SUIVANT(Y) est augmenté de SUIVANT(X) puisque là où apparaîtra X peut apparaître α Y avec les mêmes suivants.

Ici, SUIVANT(`expr`) contient ‘)’, '+' et '-' (règle 1).
`terme` est devant '*' et '/' (règle 1). `terme` est aussi en fin de règle, SUIVANT(terme) hérite de SUIVANT(expr) (règle 2). Ce qui nous donne: {* / + -) }
De même, primaire hérite de terme.

SUIVANT(expr) = { + -)}
SUIVANT(terme) = { + - * /) }
SUIVANT(primaire) = { + - * /) }

2.3 Exercices

- Soit la grammaire dont l'axiome est A:

  - `A → ε`
  - `A → B A a`
  - `B → b`
  - `B → ε`

  Construire les ensembles PREMIER et SUIVANT des non-terminaux.

- Même question:

  - `A → A B a  C D`
  - `A → a`
  - `B → B b D`
  - `B → ε`
  - `C → c D`
  - `D → d`
  - `D → ε`

- Construire l'ensemble PREMIER de la règle `primary_expression` de la grammaire C qui est en annexe page 69 (la syntaxe utilisée est celle de *yacc*, les deux points ":" remplacent la flèche "→" et les règles se terminent par un point-virgule.

3 Analyse par la droite, par la gauche [cours]

(*Rightmost & Leftmost derivation*)

Nous allons voir un peu plus loin qu'il y a deux façons de parcourir les grammaires: soit on s'attaque aux règles "par la gauche", soit on le fait "par la droite".

Prenons l'exemple de la grammaire très simple quoique récursive à droite et à gauche:

```
S → S + S
S → 1
```

Et analysons la phrase:1+1+1 dans cette grammaire

La stratégie "d'abord à gauche" va donner ce chemin d'analyse:

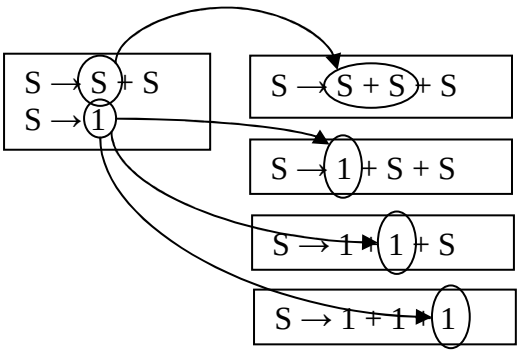


Figure 3 Analyse "d'abord à gauche" (Leftmost)

La stratégie "d'abord à droite" va donner celui-ci:

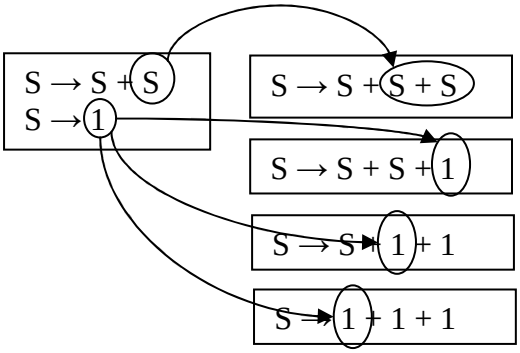


Figure 4 Analyse "d'abord à droite" (Rightmost)

4 Expressions régulières: analyse par automate d'états finis [cours/TD]

Rappelons ici la définition d'un automate fini: c'est un système capable de prendre en entrée une séquence de "tokens" (jetons), par exemple des mots du vocabulaire, et de dire en sortie si oui ou non la séquence est une phrase du langage. Le nombre d'états est fini et il n'y a pas de pile. Comme on l'a vu plus haut, on va pouvoir analyser ainsi des grammaires de type 3 et on ne pourra pas analyser des grammaires demandant à "empiler" de l'information, par exemple on ne peut pas compter les parenthèses.

On distingue les automates déterministes, et les automates non déterministes.

- Un automate déterministe est tel que, pour chaque état de l'automate, la reconnaissance d'un mot peut le faire évoluer **au plus** vers un autre état.
- Un automate non déterministe est tel que, pour chaque état de l'automate, la reconnaissance d'un mot peut le faire évoluer vers zéro, un **ou plusieurs** autres états.

4.1 Notations

Dans la suite, nous allons utiliser la notation classique utilisant la barre verticale pour signifier « ou », l'étoile * pour signifier « zéro fois ou plus », le plus + pour signifier « une fois ou plu » et le point d'interrogation pour « optionnel » ; tout ceci en notation postfixée. Ainsi **(ab|c)+** signifie : répétition au moins une fois de (**ab** ou **c**). Par exemple **ab**, ou **abc**, ou **c**, ou **cab**, ou **cababc**.

4.2 Automates déterministes

Prenons l'exemple de l'automate **a(a|b)b** capable de reconnaître les phrases "abb" et "aab". On le représente très facilement sous la forme graphique:

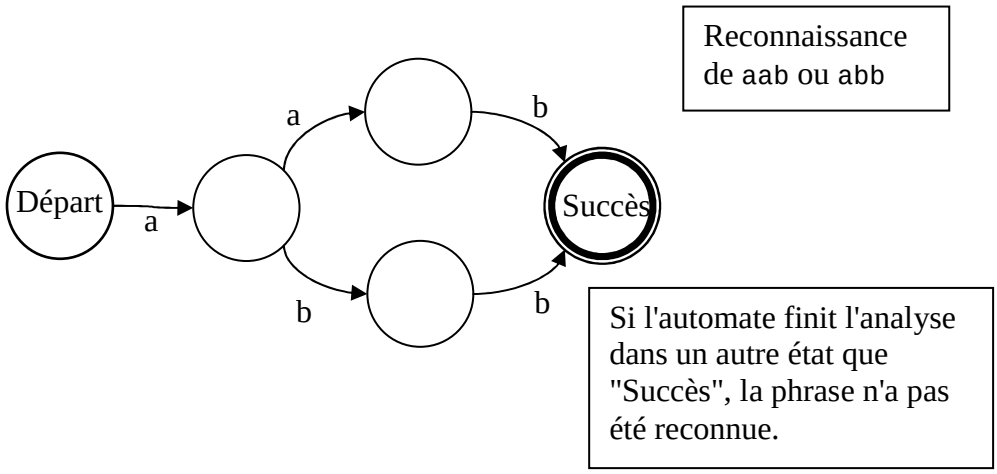


Figure 5 Représentation d'un automate déterministe

Le programmeur voit bien quel parti on peut tirer de la propriété "une transition au plus par mot". Le codage va se terminer en un **`switch`** traitant un état par branche, et décidant du changement d'état associé, unique donc, en fonction de l'entrée..

4.3 Automates non déterministes

Prenons l'exemple d'un langage à deux mots (a et b) qui accepte tout à condition que ça se termine par "abb", que nous notons donc **(a|b)*abb**. L'exemple est repris de *Aho/Sethi/Ullman* "Compilers " dit "*The Dragon Book*".[2]

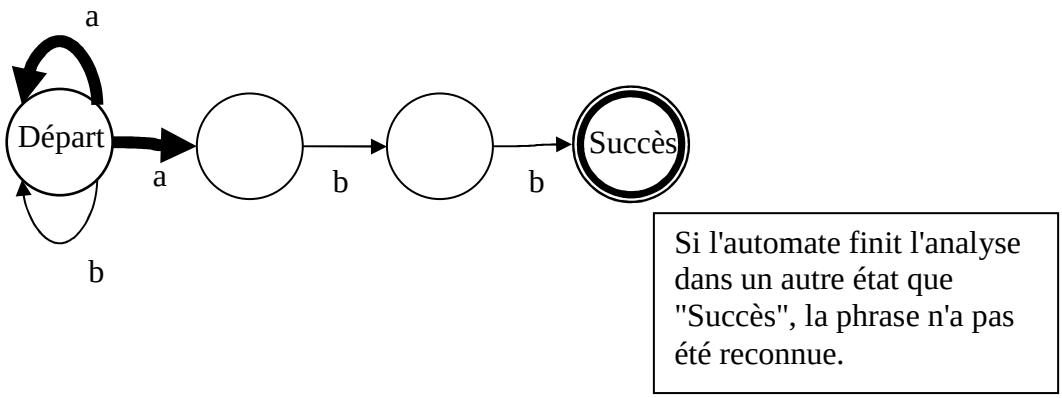


Figure 6 Représentation d'un automate non déterministe

Ici le programmeur voit bien la difficulté: quand "deviner" qu'on va arriver sur les trois derniers mots et que, depuis l'état de départ, il faut prendre le choix "de droite" plutôt que celui "du haut" sur le mot "a" (flèches en gras)? Quand l'automate est simple, un peu de réflexion peut permettre de construire "à la main" un automate déterministe équivalent

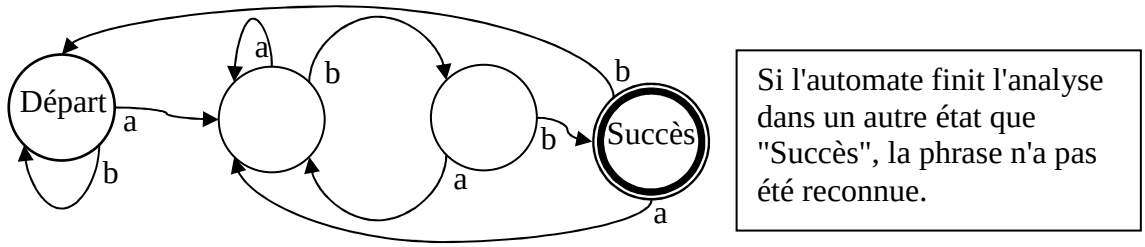
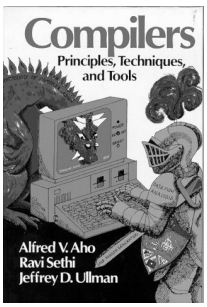


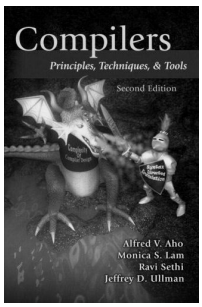
Figure 7 Le même automate rendu déterministe

4.4 Rendre déterministe un automate à états finis non déterministe

Dans l'automate non déterministe, on peut énumérer un ensemble d'états pour chaque place. L'idée est d'associer un état d'un automate déterministe, à chaque possibilité de chemin de

2





The Dragon Book: Ce livre est "la" bible des constructeurs de compilateurs. Il existe en Français chez *Interéditions*.
Les couvertures des différentes éditions (« *the red Dragon Book* », « *the purple Dragon Book* ») représentent toujours le concepteur de compilateur en chevalier face au dragon de la complexité de la compilation.

L'illustration de couverture est de ce fait adaptée à cette courte et simple introduction à la question.

l'automate non-déterministe. Cela peut conduire à une explosion combinatoire, mais en général c'est praticable.
Dans un premier temps, nous allons construire un diagramme "canonique" de notre automate non déterministe à partir de la définition (a|b)*abb en appliquant les règles des paragraphes suivants.
Rappelons que ε est l'élément vide du vocabulaire, autrement dit dans une transition par ε, on passe d'un état à l'autre sans lire de mot. Les règles ci-dessous font en sorte de ne laisser subsister le non déterminisme que sur des transitions ε. Donc, **quand il y a une transition sur un terminal non-vide, elle n'est jamais ambiguë**.

Ces règles de transformations s'appliquent évidemment de façon transitive. Dans chaque schéma, la place marquée de deux traits de taille différente ◉ est l'état final de l'automate construit. Les places marquées avec un trait gras ○ sont les états finaux des automates intermédiaires.

4.4.1 Règles de construction

4.4.1.1 Terminaux

Pour commencer, les deux automates reconnaissant ε et un symbole terminal sont représentés par deux places : entrée et sortie. Le passage de l'une à l'autre se fait sous condition d'avoir le symbole terminal disponible (ce qui est toujours le cas pour ε, et c'est dans ces transitions qu'on va isoler le non-déterminisme).

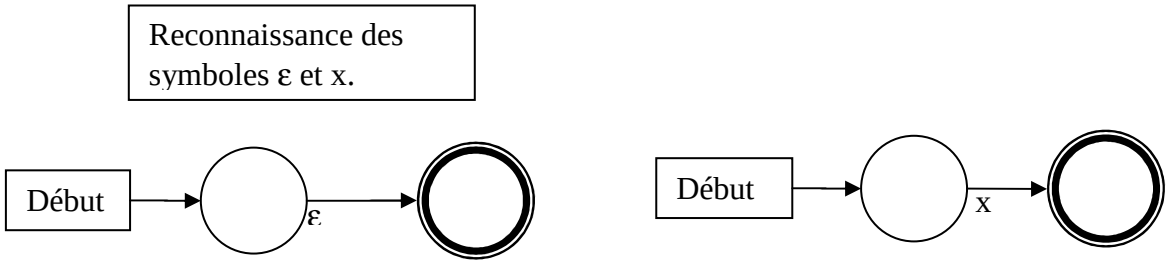


Figure 8 Transformation pour la reconnaissance des symboles terminaux

4.4.1.2 Séquence

La séquence de deux automates ayant chacun une place de départ et une place d'arrivée se fait simplement en déclarant que l'arrivée du premier est le départ du second.

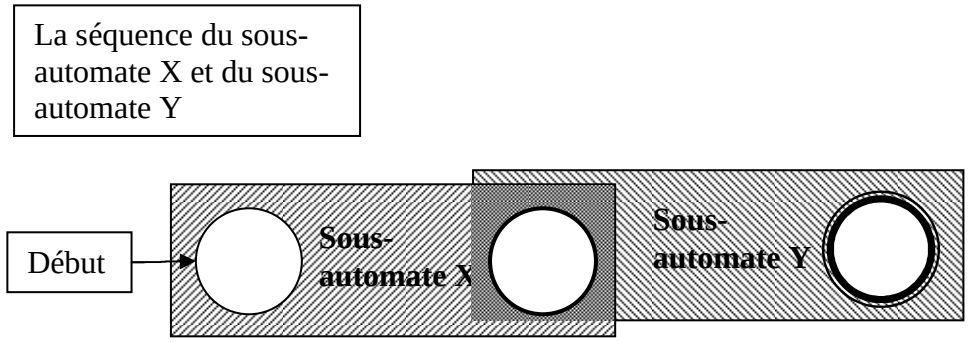


Figure 9 Transformation pour une séquence de deux sous-automates

4.4.1.3 Répétition

La répétition N fois d'un automate, avec N>=0, comme dans **X+** ou **X***, se fait en mettant une boucle entre l'état final et l'état initial de l'automate, et en autorisant un « court-circuit » pour le cas **X*** (N = 0 autorisé).

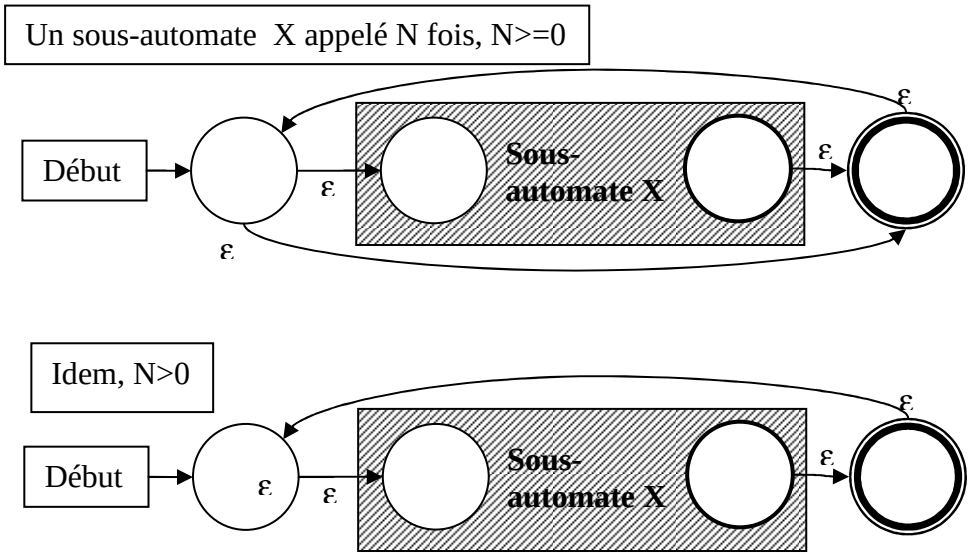


Figure 10 Transformation pour la répétition

4.4.1.4 Choix

Le choix (comme dans **X|Y**) se fait en mettant deux transitions ε vers les deux sous-automates depuis et vers les états d'entrée et de sortie.

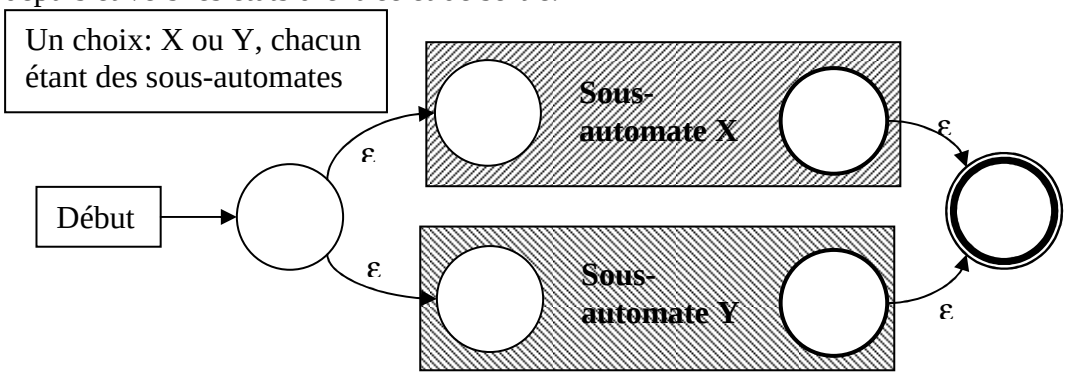


Figure 11 Transformation pour un choix entre deux sous-automates

4.4.1.5 Option

L'option (comme dans **X?**) se fait en utilisant un choix entre A et rien.

.

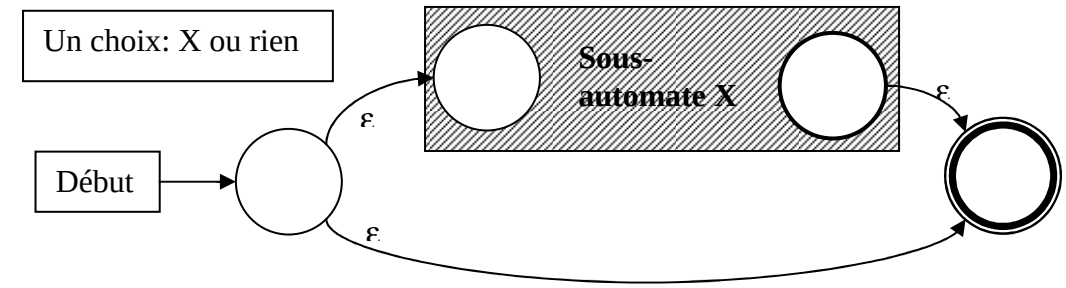


Figure 12 Transformation pour une option

4.4.2 Construction d'un exemple

Ainsi, sur l'exemple supra **`(a|b)*abb`**, qui se lit donc « répétition zéro fois ou plus de **(a** ou **b)** terminé de **`aab`** **»**, le lecteur peut vérifier à titre d'exercice -en décomposant en choix, séquences et répétition selon les méthodes dites supra- qu'une construction systématique de l'automate donnera:

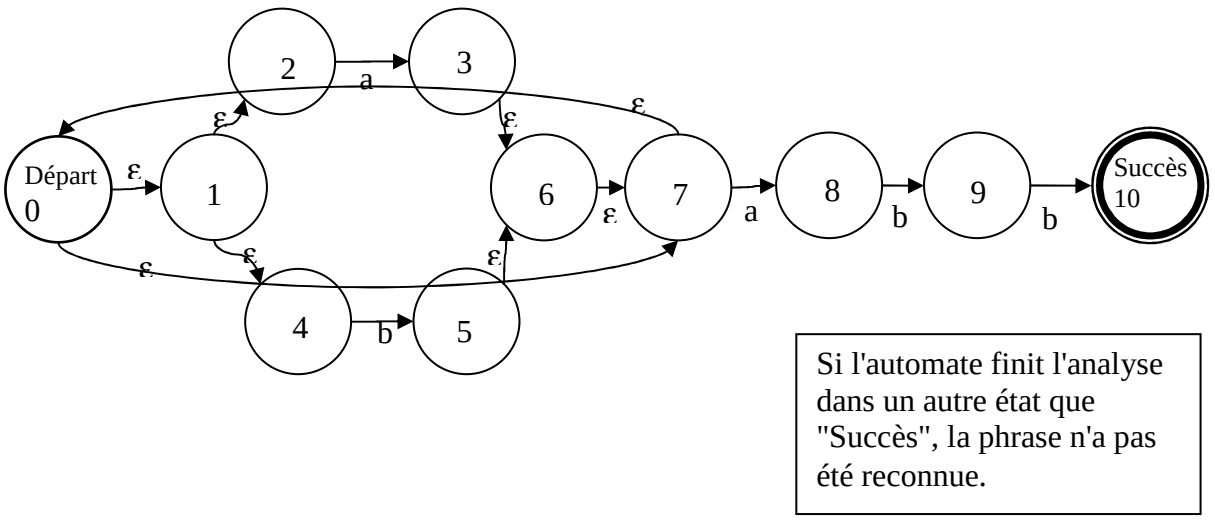


Figure 13 Automate non déterministe pour (a|b)*abb

Calculons maintenant les états d'un automate déterministe équivalent en procédant par inondation (rappelons que le but du jeu est de définir quels sont les ensembles de places atteignables pour chaque séquence possible): on voit que depuis l'état de départ, on peut atteindre les états 0, 1, 2, 4, 7 par des transitions **ε**. Appelons A l'ensemble en question. **A : {0, 1, 2, 4, 7}.** A sera la place de départ de notre futur automate déterministe.

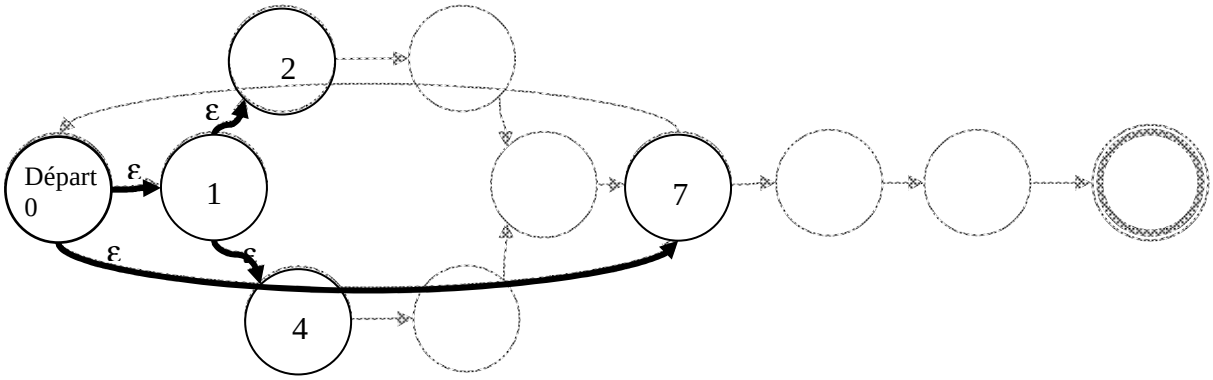


Figure 14 États atteignables depuis le départ par une transition ε

Voyons maintenant l'ensemble des états accessibles par les deux symboles **a** et **b**, et ceci transitivement, en construisant les ensembles intéressants au fur et à mesure.

- Partir de A avec le symbole **a** nous fait partir de 2 ou 7 et tomber directement dans les états 3 et 8,

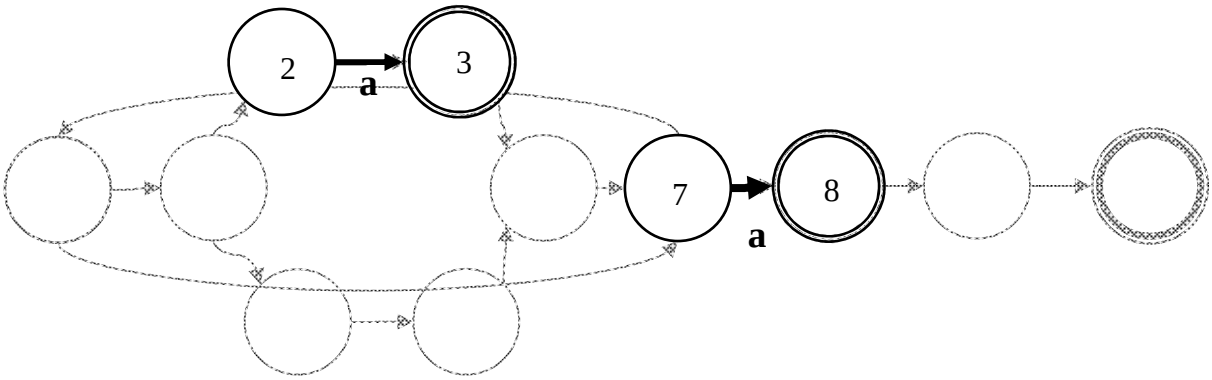


Figure 15 États atteignables depuis A par une transition "a"

qui à leur tour par les transitions **ε** donnent l'ensemble **B: {0, 1, 2, 3, 4, 6, 7, 8}.** Ce sont donc là tous les états atteignables depuis l'état de départ, après la reconnaissance de **a**.

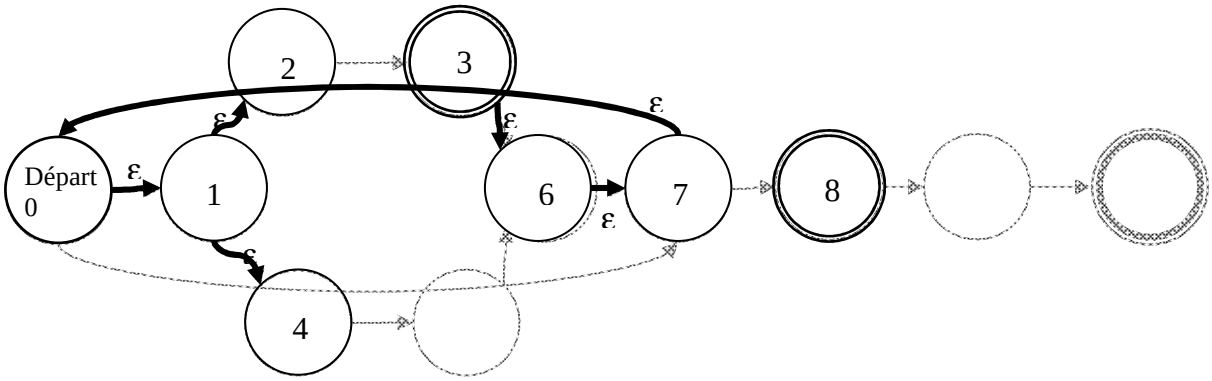


Figure 16 États atteignables depuis ces derniers par une transition ε

Nous pouvons donc continuer cette inondation de façon systématique :

- Partir de A avec le symbole **b** se résume à une seule place accessible directement: 5, qui par les transitions **ε** donnera l'ensemble **C: {0, 1, 2, 4, 5, 6, 7}**
- Partir de B avec le symbole **a** donnera directement {3, 8} donc nous retombons sur B.
- Partir de B avec le symbole **b** donnera directement {5, 9} ce qui nous donne par les transitions **ε** l'ensemble **D: {0, 1, 2, 4, 5, 6, 7, 9}**
- Partir de C avec le symbole **a** donnera directement {3, 8} donc nous retombons sur B
- Partir de C avec le symbole **b** donnera directement {5} donc nous retombons sur C
- Partir de D avec le symbole **a** donnera directement {3, 8} ce qui nous redonne B
- Partir de D avec le symbole **b** donnera directement {5, 10} ce qui, par les transitions ε, nous donne **E: {0, 1, 2, 4, 5, 6, 7, 10}**

On peut maintenant vérifier que partir de E avec **a** ou **b** fait retomber sur des ensembles déjà construits (B et C).
Nous avons donc énuméré les cinq états de notre nouvelle machine: A, B, C, D, E, il n'y en aura pas d'autres. On peut constater qu'ils correspondent chacun à une propriété de l'automate non-déterministe :

- A est l'état de départ
- B est l'état où l'on se trouve quand il y a eu une succession de deux **a**
- C est l'état où l'on se trouve quand la séquence de fin **abb** n'est pas commencée (**b** ou **bb** depuis l'état de départ, ou **bbb** ensuite).
- D est l'état où l'on se trouve quand il y a eu une séquence **ab**
- E est l'état d'acceptation (présence de **abb**), qui indique le succès si c'est aussi l'arrêt de la chaîne d'entrée.

Pour dessiner cette nouvelle machine et ses transitions, il suffit de relire comment et sous quelles conditions nous avons défini les ensembles dans l'énumération ci-dessus.

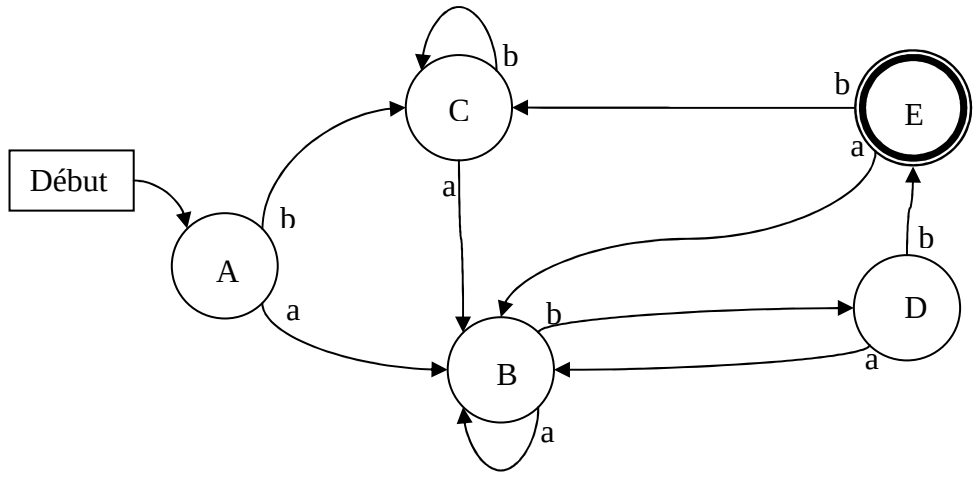


Figure 17 Automate déterministe équivalent construit mécaniquement

Bien entendu, ce processus peut être assez long pour de « vrais » automates, mais il est certain qu'il est borné, puisqu'il n'y a « que » 2^{11} combinaisons possibles des états 0 à 10. Cette explosion combinatoire qui survient sur des questions pathologiques n'est que rarement un problème.

En parcourant cet automate « à la main », on voit qu'il est déterministe (chaque état a au plus une sortie pour un terminal particulier). Il est facile de constater que l'arrêt sur l'état final ne peut se faire que si la chaîne d'entrée se termine par **abb**, ce qui était bien la question.

On voit aussi que cet automate construit « automatiquement » possède un état de plus que celui que nous avons construit « à la main » page 16. D'une part les techniques habituelles d'optimisation peuvent encore s'appliquer, d'autre part il n'est pas forcément inutile d'avoir un état « départ » par lequel on ne repasse plus, ce qui est le cas ici.

4.5 Exercices :

- Faire une grammaire qui accepte des mots symétriques composés de N x, avec un seul y au milieu : **y** ou **xyx** ou **xxyxx** ou **xxxyxxx** mais pas **xxyxxx**

- Reprendre l'exemple donné supra dans les grammaires de type 1, et dériver pour obtenir **aaaabbbbcccc**.

- L'expression **b*(a|b) ?a** ne se recopie pas directement en automate déterministe. Utiliser la méthode du §4.4.2 pour le rendre déterministe.

- Même question avec : **1***(**1**|**01**)*

5 Grammaires hors contexte: analyse descendante LL [cours/TP]

5.1 Définition

LL est l'acronyme de ***L****eft to Right,* ***L****eftmost derivation,* ce qui signifie qu'on analyse le texte de gauche à droite, en choisissant toujours la dérivation la plus à gauche (voir chap.3 page 13).
L'analyse descendante consiste à parcourir la grammaire de gauche à droite en prenant des décisions locales basées sur le terminal (ou les terminaux) que l'on voit. Quand on tombe sur une sous-règle dans l'analyse d'une règle, le contexte est empilé.
La grammaire que nous allons maintenant utiliser ci-après est de la forme :

```
axiome → expr ';'

expr → terme '+' terme
expr → terme '-' terme
expr → terme

terme → primaire '*' primaire
terme → primaire '/' primaire
terme → primaire

primaire → '(' expr ')'
primaire → entier

entier → chiffre entier
entier → chiffre

chiffre → '0'
chiffre → '1'
etc…
chiffre → '9'
```

La règle `expr` étant appelée de façon récursive, il est de bon goût d'entrer dans la grammaire par une autre règle qui, elle, n'est pas appelée. Cela donnera l'opportunité de faire quelque chose de spécifique à l'entrée et à la sortie de l'analyse. Le point-virgule terminal permet d'obliger l'interpréteur à faire quelque chose avant de tomber sur la fin de fichier.

Ici la règle chiffre est décomposée en 10 sous-règles, en fait on aura avantage à en faire une primitive (voir plus bas).

Figure 18 Grammaire d'expression

Cette grammaire accepte les expressions de la forme 2+3 ou 3*(4+5)

Le lecteur attentif remarquera que cette grammaire n'accepte *pas* les cascades d'opérateurs de même priorité comme 2+3+4 ou 3-2+5 (alors qu'on peut écrire 2+3*4)

Ceci peut se régler en rendant les règles récursives à gauche.

```
expr → expr  '+' terme
expr → expr  '-' terme
expr → terme
```

```
terme → terme  '*' primaire
terme → terme  '/' primaire
terme → primaire
```

Néanmoins nous ne le ferons pas dans un premier temps, car l'analyse LL1 que nous allons voir ci-après ne permet pas la récursivité à gauche. La récursivité à droite serait possible et accepterait les mêmes phrases syntaxiques, mais au moment de générer les actions aurait l'inconvénient de factoriser de façon contre-intuitive : 2-3+4 deviendrait 2 – (3+4) au lieu de l'espéré (2-3) +4

Rassurons-nous, un contournement très simple permet de remplacer la récursivité par une boucle explicite, et on pourrait y arriver.

5.2 Écrire un interpréteur LL1 en C

LL1 est l'acronyme de ***L****eft to Right,* ***L****eftmost derivation* ***1*** *token*, ce qui signifie que c'est une analyse LL (voir §5.1 ci-dessus) et en ayant un seul mot d'avance pour décider de la branche explorée. C'est le cas habituel, analyser avec deux ou plus jetons d'avance conduit à une explosion combinatoire pour les "vrais" langages.

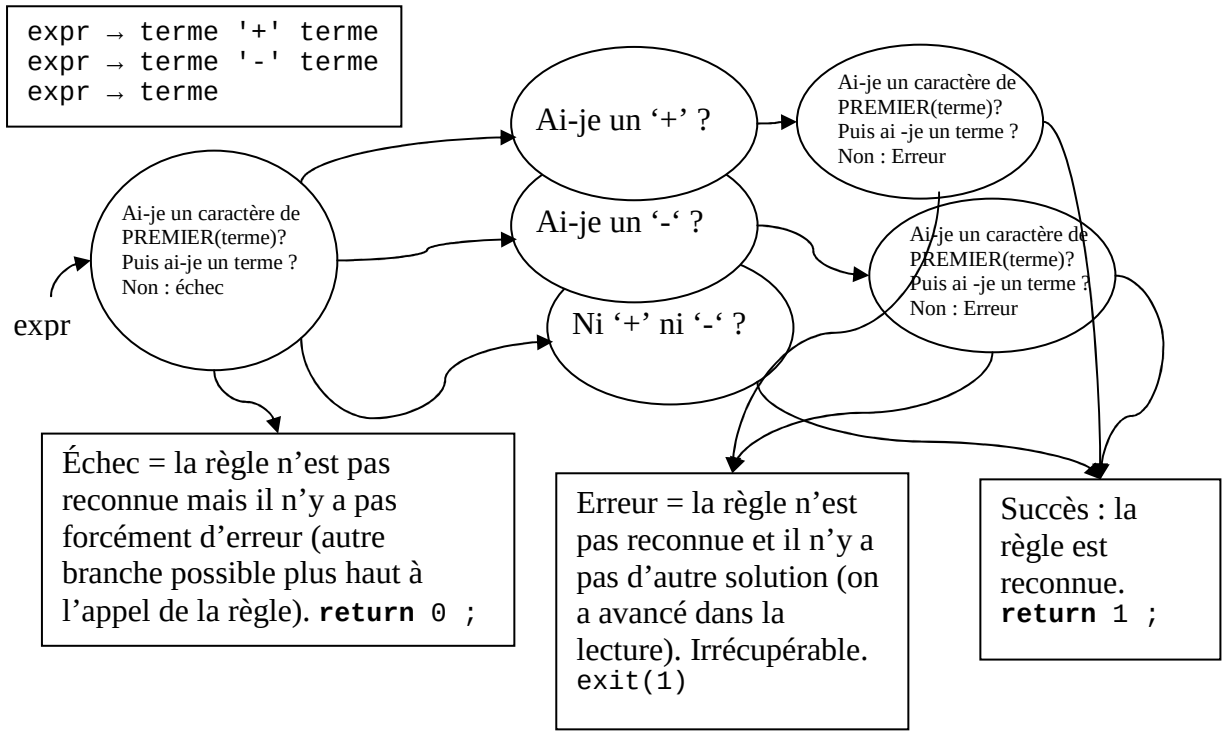


Figure 19 L'analyse descendante LL1 d'une règle

Comme dit supra, il faut empiler les contextes quand une règle en appelle une autre. Une façon simple de le faire est…de ne pas le faire et d'utiliser pour décrire l'algorithme d'analyse un langage autorisant la récursivité (Ada, Pascal, C mais pas Fortran ou Basic), à qui on va laisser le soin d'empiler l'information lui-même quand on appelle un sous-programme. Nous allons ici reproduire quasi-mécaniquement la grammaire en un programme C en appliquant quelques méthodes simples :

- Chaque symbole terminal (ici ce sera le caractère) est reconnu par une fonction qui rend 1 ou 0 selon qu'elle le voit comme terminal courant. De plus, si c'est oui, une lecture est faite et le terminal courant change, on avance. Cette méthode est « marche ou crève », les choix faits ne peuvent pas être remis en question (le LL de LL1) et ils doivent être faits avec un seul terminal (le 1 de LL1). Pour cela, on peut être obligé de transformer légèrement la grammaire, de façon que tous les choix de branche puissent être faits avec un terminal « décisif » en première position. Voir l'exemple ci-après.
- Chaque symbole non-terminal devient une fonction C qui rend un booléen (entier 0 ou 1 selon la convention C). Les séquences de terminaux ou non-terminaux sont simplement des cascades de « if ». Les choix (barre verticale dans la BNF, ou

multiples définitions du même non-terminal) deviennent des cascades de « else if ». Ce qui fait qu'*in fine*, la règle racine (l'axiome) rendra 1 ou 0 suivant que la syntaxe soumise a été correcte ou pas.

- Chaque fonction (terminal ou non) rend donc 1 ou 0 selon qu'elle a reconnu ce qu'on lui demande.

Accepter un terminal, ici un char:

```
char char_courant ;
/* global, initialisé dans main() */
            …
int tok(char quoi) {
  if (char_courant==quoi) {
    char_courant = char_suivant() ;
/* à écrire avec getchar ou scanf*/
/* on lit le caractère suivant */
   return 1 ;
   } else return 0 ;
}
```

Autres fonctions utiles :

On sera vite amené à faire plusieurs fonctions « tok », s'intéressant aux étendues (lettre, chiffres) : par exemple `int tok_chiffre()` qui rend 1 si le terminal courant est un chiffre, et qui dans ce cas passe au terminal suivant..

On sera aussi amené à écrire `char_suivant()` de façon à gérer les blancs et retour-chariots.

Accepter A puis B :

```
int A_puis_B () {
  if (A()) {
    if (B()) {
      return 1;
     }
    else return 0;
       /*(ou exit(1), si A a fait
          lire un caractère */
  }
 else return 0 ;
}
```

Accepter A ou sinon B :

```
int A_ou_sinon_B () {
  if (A()) {
    return 1;
    }
  else if (B()) {
    return 1;
    }
  else return 0 ;
}
```

Accepter A optionnel :

```
int A_opt () {
   A();
   return 1 ;
}
```

Accepter A zéro fois ou plus :

```
int A_opt () {
  while (A());
  return 1 ;
}
```

(Exercice: écrire "accepter A une fois ou plus".)

Figure 20 Transformation de BNF en C

La fonction main se bornera à initialiser le caractère courant, à appeler l'axiome et à faire un message en cas d'erreur.

```
int main()
{
   char_courant = char_suivant() ;
   if (axiome()) printf("Ok\n") ;
     else printf("Ko\n");
}
```

Ainsi, à titre d'exemple, la règle

```
expr → terme '+' terme
expr → terme '-' terme
expr → terme
```

deviendra, après la factorisation nécessaire (car chaque règle doit décider de la branche qu'elle prend avec un seul terminal d'avance), ce qui impose que les choix entre les terminaux soient au début des règles :

```
expr → terme fin_expr
fin_expr → '+' terme
fin_expr → '-' terme
fin_expr →
```

Le code C peut copier fidèlement la transformation (première colonne), ou rassembler tout le code en une seule fonction (deuxième colonne).

```
int fin_expr() }
  if (tok('+')) {
    if (terme()) {
     return 1;
     }
    else printf("erreur\n");
         exit(1);
  }
  else if (tok('-')) {
    if (terme()) {
     return 1;
     }
    else printf("erreur\n");
         exit(1);
  }
 else return 1;
}

int expr() {
  if (terme()) {
    if (fin_expr())
      return 1;
    else return 0;
  else return 0;
}
```

```
int expr() {
  if (terme()) {
    if (tok('+')) {
      if (terme()) {
       return 1;
       }
      else printf("erreur\n");
           exit(1);
    }
    else if (tok('-')) {
      if (terme()) {
       return 1;
       }
      else printf("erreur\n") ;
           exit(1);
    }
    else return 1;
  }
 else return 0 ;
}
```

Figure 21 Exemple de transformation BNF en C

La principale source d'erreurs dans cette méthode est l'oubli d'un return dans une des multiples branches.

Les actions :
Pour que notre calculette calcule, nous allons ajouter des actions dans notre programme. Il nous faudra une pile de calcul avec les fonctions élémentaires

```
void pousser(int)
int valeur()
void tirer()
```

- Chaque fois qu'il y a une opération à faire, on la fait quand 1/on sait quel opérateur est concerné et 2/ les valeurs des opérandes ont été préalablement calculées. C'est là pour '+' : la branche où l'on est dit qu'il faut faire une addition, et on est passé par les deux

règles « terme » qui ont chacune fait en sorte que la valeur correspondante soit en haut de pile, qui contient donc en son sommet les deux opérandes. Il faut donc écrire à cet endroit quelque chose comme :
`x=valeur() ; tirer() ; y=valeur() ; tirer() ; pousser(y+x) ;`
(attention pour les opérateur asymétriques, la première valeur est la dernière tirée)

- Quand on lit un entier, il faut le pousser sur la pile. Ceci est fait dans la règle `primaire()`après l'appel `d'entier()`. L'entier est calculé dans la règle `entier()`, par exemple en déclarant un entier global tmp, initialisé à 0 dans primaire, et mis à jour à chaque appel récursif de `chiffre()` :
`tmp = 10*entier + (char_courant-'0' ;.`
- À la fin de l'analyse, il faut afficher le résultat : imprimer la valeur du haut de la pile. Ceci est fait à la fin de la règle de plus haut niveau, l'axiome.

5.3 Faire facilement un générateur d'interpréteurs LL1

Le côté « automatique » de l'écriture des analyseurs LL1 à partir de la grammaire peut être mis à profit avec un jeu de macros très simples, que l'on instanciera avec la grammaire sous une forme adaptée, et qui engendreront le programme C qui analysera la grammaire. Toujours avec le même exemple, voici :

```
#include <stdio.h>
#define AND2(A,B) ((A)?(B):0)
#define AND3(A,B,C) ((A)?((B)?(C):0):0)
#define OR2(A,B) ((A)?1:(B))
#define OR3(A,B,C) ((A)?1:((B)?1:(C)))
#define RIEN (1)
#define DEF(regle, definition) int regle() { return definition; }

/* les marcos AND et OR appellent les symboles qu'on leur passe, et rendent
un ou zéro suivant qu'ils sont tous vrais en séquence (AND) ou si l'un est
vrai (OR). La macro DEF définit une nouvelle règle. */

#define GET_ET_SUITE(CHAR,SUITE) (\
                         get(CHAR)?\
                          ((SUITE)?1:(printf("erreur\n"),exit(1),0))\
                          :0)

/* cette macro gère le fait que, si un caractère est lu et reconnu, le
retour en arrière est impossible. Donc il ne faut pas rendre 0 (échec de la
branche courante, mais possibilité de succès à l'étage au dessus) mais
finir l'analyse avec une erreur. Ensuite les deux fonctions de service
permettant de lire un caractère ou un entier au clavier*/
char char_courant;
int get(char c) {
  if (c==char_courant)
    {scanf("%c",&char_courant);
     return 1;
    } else return 0;
}
int get_entier() {
  if ((char_courant>='0')&&(char_courant<='9'))
   {scanf("%c",&char_courant);return 1;}
    else return 0;
}
```

```
/***********************************************
    Ici la définition de la grammaire
***********************************************/
DEF(expression,\
    AND2( terme(), \
      OR3( \
         GET_ET_SUITE ('+', terme() ),
         GET_ET_SUITE ('-', terme() ),\
         RIEN\
       ) ) )
```

```
expr → terme '+' terme
expr → terme '-' terme
expr → terme
```

```
DEF(terme, \
    AND2( prim(),\
      OR3( \
         GET_ET_SUITE ('*', prim() ), \
         GET_ET_SUITE ('/', prim() ),\
         RIEN\
       ) ) )
```

```
terme → primaire '*' primaire
terme → primaire '/' primaire
terme → primaire
```

```
DEF(prim, \
    OR2( \
          GET_ET_SUITE ('(', \
                        AND2(expression(), get(')'))), \
          get_entier()\
       ) )
```

```
primaire → '(' expr ')'
primaire → entier
```

```
/* et le main pour tester tout ça. */

int main()
{
 scanf("%c",&char_courant);
 if (expression()) printf("ok\n"); else printf("ko\n");
}
```

On remarquera ici l'utilisation de la macro GET_ET_SUITE dont l'effet est de rendre 0 si le caractère fourni n'est pas reconnu, et de faire une erreur s'il est reconnu, mais pas la suite ; ce qui est irrécupérable. Dans cette macro, voir l'opérateur *virgule* dans l'expression (`printf("erreur\n"),exit(1),0))` qui exécute le printf et le exit, avec le 0 pour satisfaire au typage. Rappelons que *virgule* est un opérateur qui évalue ses opérandes, et rend la valeur du deuxième.

Mettre des actions dans ce petit générateur demanderait d'utiliser massivement cet opérateur virgule, en mettant comme premier argument la fonction agissante et en second argument la fonction analysante, mais bien que cela soit possible on sortirait des limites du raisonnable : déboguer des macros est un véritable casse-tête.

6 Grammaires hors contexte: analyse ascendante LR [cours]

LR est l'acronyme de ***L****eft to Right,* ***R****ightmost derivation*, ce qui signifie qu'on analyse le texte de gauche à droite, en choisissant toujours la dérivation la plus à droite (voir chap.3 page 13). On dira LR**1** si on utilise pour la décision un seul jeton d'avance (cas habituel, analyser avec 2 jetons d'avance ou plus conduit à une explosion combinatoire). Nous allons voir ici une catégorie d'analyse appelée **S**LR1, où le S veut dire ***S****imple*. Les outils automatiques comme *yacc* sont capables d'analyser selon l'algorithme **LA**LR1, où LA veut dire ***L****ook-****A****head (voir la note3 page 34*). La complication induite n'est pas extraordinaire mais sort du cadre d'une introduction.

6.1 Définition

L'analyse ascendante consiste à considérer le texte à analyser, et à y reconnaître les parties droites des règles de la grammaire, qu'on remplace par leur partie gauche. Et ceci transitivement, jusqu'à ce qu'on ait reconnu la règle axiome.

L'analyse ascendante sera illustrée ici par un système à pile dans laquelle nous allons mettre tous les symboles terminaux ou non-terminaux, par un codage quelconque (on assigne un entier à chaque). Attention, l'exemple est seulement illustratif **du résultat de** l'automate et, même si le commentaire peut en donner l'illusion, ce n'est pas l'algorithme qui permet d'obtenir ce résultat!

Une case contient le terminal (ici le caractère) courant, celui qu'«il faut caser ». À chaque étape, on examine le sommet de la pile et on regarde si la règle qui s'y trouve s'accommode du terminal « à caser » dans une partie droite d'une des règles de la grammaire.

- Si oui, (c'est dans le schéma ce qui se passe quand on met le '+', puisqu'il y a une partie droite de règle qui commence par « `terme '+'` ») on le met sur la pile en espérant que la suite conviendra. Cette décision n'est pas du tout triviale, en regardant la figure ci-contre on pourra voir que des choix sont faits qui ne sont pas « locaux » à une règle, par exemple la décision de ne pas réduire le tout premier « chiffre » en « entier ».
- Si non, on cherche à réduire la pile en appliquant les règles (c'est ce qu'on fait un peu partout, par exemple en remplaçant `primaire` par `terme` ou `terme '+' terme` par `expr`.
- Si on ne peut pas réduire, on met le terminal sur la pile (c'est ce qu'on fait quand on met le « 4 » dans la pile) en espérant que des réductions vont fonctionner.

Quand l'axiome lui-même est réduit, l'analyse est un succès.

Cette forme d'analyse se prête très mal à l'écriture directe d'un programme par un programmeur humain ; en regardant de près l'exemple ci-contre, on verra que même la décision qui paraît « intellectuellement » facile à prendre (faire une réduction, par exemple), demande, soit une exploration systématique de toute la grammaire, impraticable, soit la compilation de la grammaire en tables d'états parcourues par une machine. C'est ce que font les analyseurs comme *yacc* ou *bison*. C'est aussi ce que nous allons expérimenter au §6.2.

Analyse ascendante de 13+4

Un terminal d'entrée d'avance (ici caractère du programme). En gras quand on procède à une lecture. EOT est le symbole « end of text ».

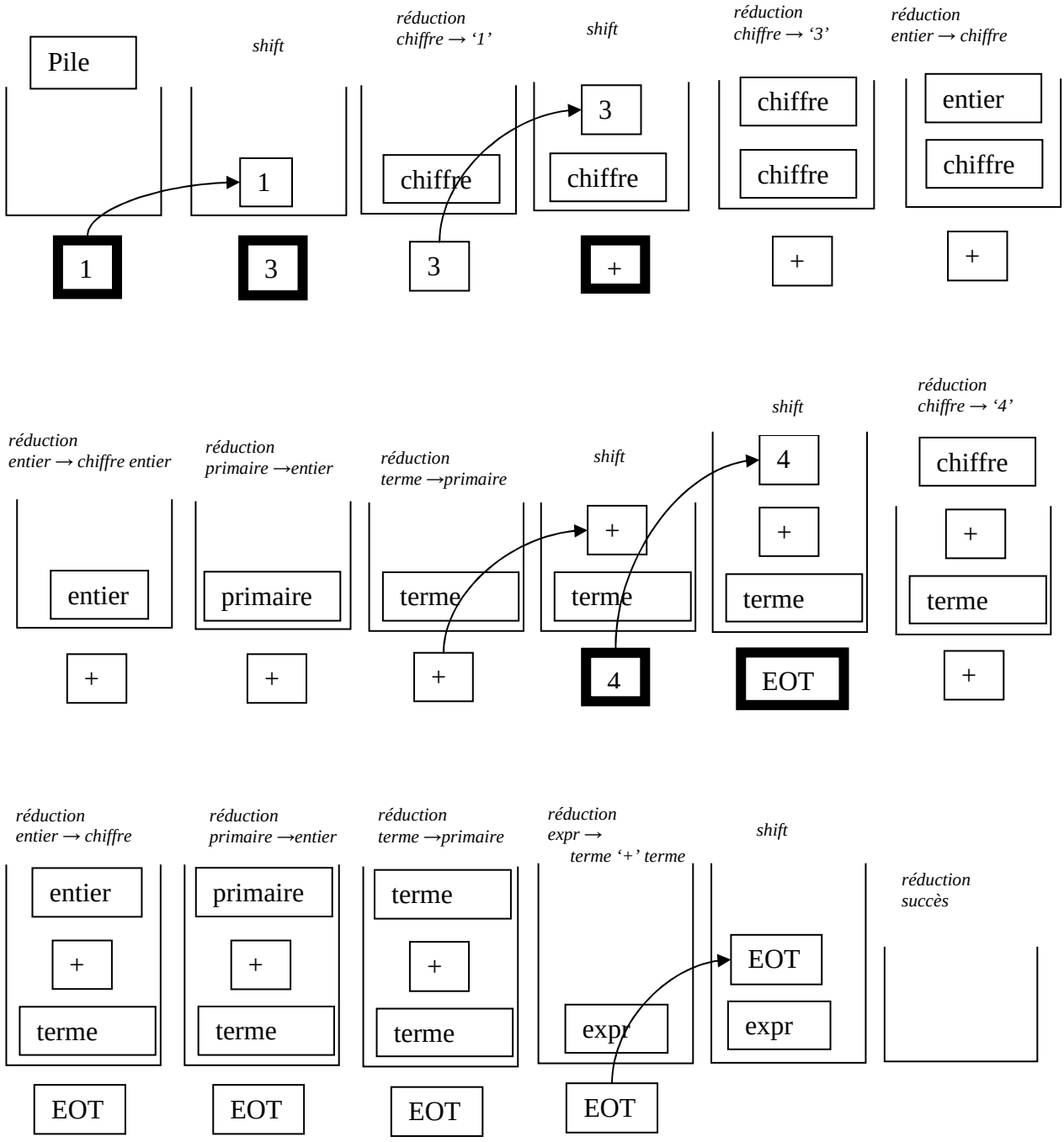


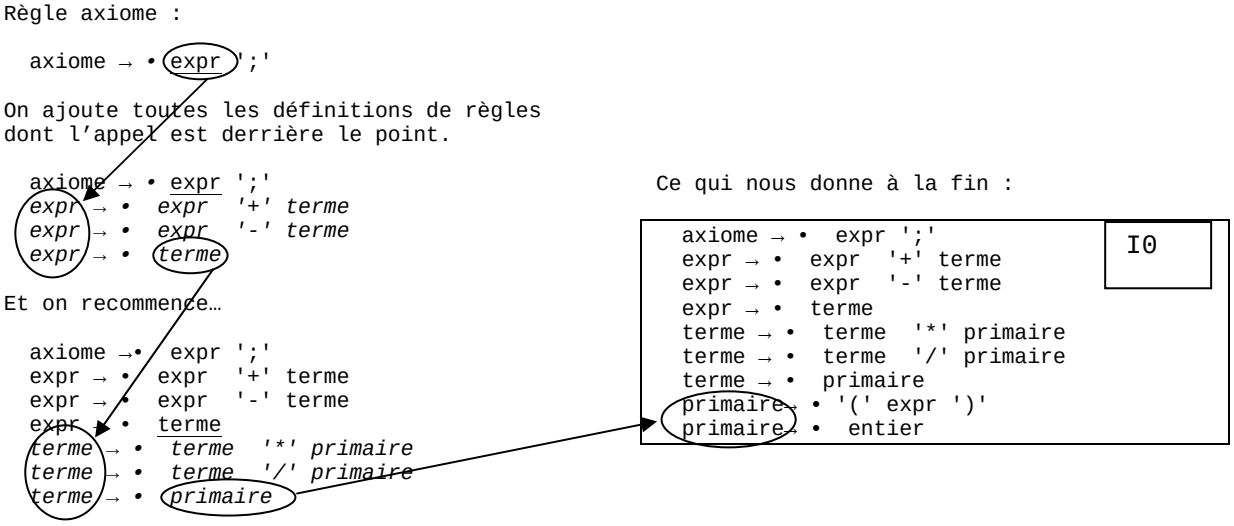
Figure 22 Analyse ascendante

6.2 Algorithme d'analyse SLR 1

Comme dit supra, SLR1 est une méthode d'analyse où le S veut dire ***S****imple*. Nous allons voir que ce n'est pas si simple que cela.
Pour illustrer une des forces des analyses LR, nous allons légèrement modifier notre grammaire pour qu'elle soit récursive à gauche (et accepte donc des cascades d'opérateurs de même priorité comme 2+3+4-5). Nous allons aussi considérer que entier est un terminal, et numéroter les règles.

```
1. axiome → expr ';'
2. expr → expr  '+' terme
3. expr → expr  '-' terme
4. expr → terme
5. terme → terme  '*' primaire
6. terme → terme  '/' primaire
7. terme → primaire
8. primaire → '(' expr ')'
9. primaire → entier
```

Le jeu consiste à, construire des ensembles de règles représentant les positions "possibles" de l'analyse après réception d'une chaîne de mots donnée. Ainsi, nous commençons au début par la règle axiome, et nous mettre un point (•) en tête de la règle. Ce point représente "l'endroit où l'on en est".
Puis, de façon transitive, nous recopions les règles qui sont derrière un point comme appartenant au même ensemble. Ci-dessous, les règles soulignées sont celles que l'on va recopier dans l'ensemble courant, qui sont en italique dessous. Quand la construction de l'ensemble est terminée, on l'encadre ; celui-ci est baptisé.



Passe 1

Sur cet ensemble, voyons les items qui sont sensibles à l'arrivée des terminaux (`+ - * / ( ) ;` et `entier`) et des non terminaux (`axiome, expr, terme, primaire`) : comme terminaux, il n'y a que la parenthèse ouvrante et l'entier qui permettent d'avancer, avancer par

`entier` nous donne une règle unique qui reste toute seule dans son ensemble puisqu'elle « débouche » ainsi sur la fin de la règle. Nous l'ajoutons cet ensemble à la liste des ensembles intéressants : c'est I1. Avancer par la parenthèse ouvrante va nous donner
`primaire→ '(' • expr ')' –` Le point est devant un non-terminal, donc en recommençant la mécanique ci-dessus nous construisant un ensemble que nous ajoutons à la liste des ensembles intéressants; c'est I2:

```
I0 -> I1 par entier
I2 -> I1 par entier (voir ci-dessous)
I7 -> I1 par entier (voir ci-dessous)
I8 -> I1 par entier (voir ci-dessous)
I9 -> I1 par entier (voir ci-dessous)
I10 -> I1 par entier (voir ci-dessous)
primaire → entier •          I1
```

```
I0 -> I2 par '('
I2 -> I2 par '(' (voir ci-dessous)
I9 -> I2 par '(' (voir ci-dessous)
I10 -> I2 par '(' (voir ci-dessous)
primaire → '(' •  expr ')'
expr → •  expr  '+' terme
expr → •  expr  '-' terme          I2
expr → •  terme
terme→ •  terme '*' primaire
terme→ •  terme '/' primaire
terme→ •  primaire
primaire → • '(' expr ')'
primaire → •  entier
```

Pour ce qui est des non terminaux, `axiome` n'étant jamais appelé ne donnera jamais rien ; `expr` fait avancer trois règles. Ce qui va nous donner l'ensemble I3, lequel ne sera pas augmenté, aucun point n'étant devant un non-terminal. De même, `terme` fait avancer trois règles, ce qui nous donne I4. `primaire` fait avancer une règle, ce qui nous donne I5.

```
I0 -> I3 par expr
axiome → expr •  ';'
expr →  expr •  '+' terme          I3
expr →  expr •  '-' terme
```

```
I0 -> I4 par terme
I2 -> I4 par terme (voir ci-dessous)
expr → terme •
terme → terme •  '*' primaire      I4
terme → terme • '/' primaire
```

```
I0 -> I5 par primaire
I2 -> I5 par primaire (voir ci-dessous)
I7 -> I5 par primaire (voir ci-dessous)
I8 -> I5 par primaire (voir ci-dessous)
terme → primaire •          I5
```

Passe 2 :

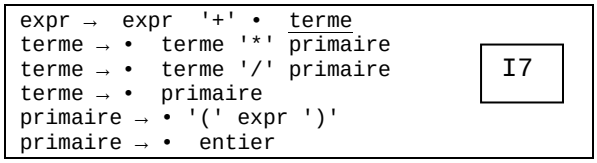
Recommençons toute l'opération. Cette fois-ci nous acceptons les terminaux {'+', '-', ';') dans I3, ('*', '/',) dans I4, '(' dans I2, et comme non-terminal `expr, terme` et `primaire` (dans I2). `expr` va nous donner l'ensemble I6. `terme` nous fait retomber sur I4 et `primaire` sur I5. I2 reboucle sur lui-même par '(' et va sur I1 par `entier`. Nous rajoutons cela en italique dans les commentaires au dessus des ensembles déjà construits. .
Par '+', nous faisons avancer

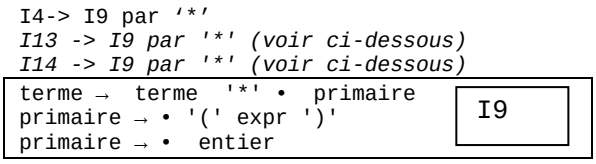
```
expr →  expr  '+' •  terme
```

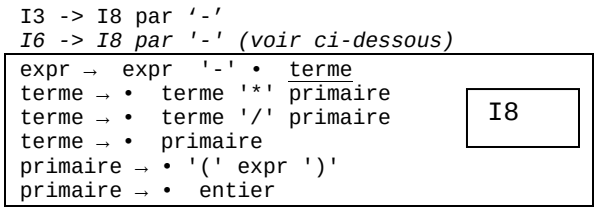
Ceci place donc le point devant 'terme ' et nous amène à construire I7. De même I8, I9, I10, I11 et I12 que l'on construit en cherchant les règles qui avancent par '–', '*', '/' et ';'.

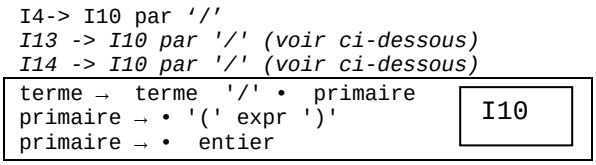
```
I2 -> I6 par expr
primaire → '('  expr • ')'
expr → expr •  '+' terme           I6
expr → expr •   '-' terme
```

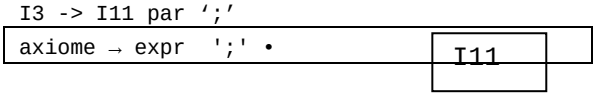
```
I3 -> I7 par '+'
I6 -> I7 par '+' (voir ci-dessous)
```





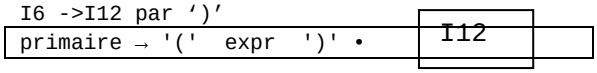




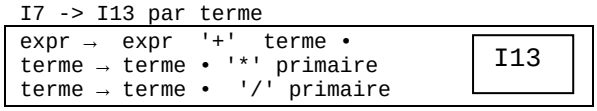


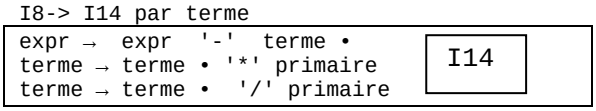
Passe 3

Recommençons encore. Dans ces nouveaux ensembles, la sensibilité aux terminaux est sur '+', '-', '(', `entier` et ')'. Cela nous fait retomber sur les ensembles I7, I8, I1, I2 pour les quatre premiers, nous mettons à jour les commentaires en italiques au dessus des ensembles concernés. Et cela nous amène à construire I12 pour ')'.

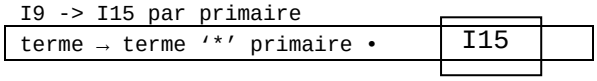


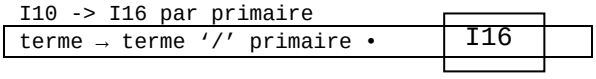
La sensibilité aux non-terminaux concerne `terme` et `primaire`. Nous retombons sur I4 et I5 (mise à jour des dépendances en italique au dessus des ensembles concernés), mais aussi nous avons les nouveaux ensembles:





De même, en regardant les règles sensibles à `primaire`, nous vérifions que I7 et I8 font retomber sur I1, nous mettons à jour ses dépendances en italique. Puis nous obtenons :





Le point n'est devant aucun non-terminal, les ensembles ne sont pas augmentés.

Passe 4

Nous constatons que de I13 et I14 nous allons vers les ensembles déjà décrits I9 et I10. Nous mettons à jour les commentaires en italique. Et c'est fini !

Faisons le diagramme de transitions de cet automate. Il suffit de recopier les noms de nos ensembles et d'illustrer avec des flèches les conditions de transition qui se trouvent au dessus de leur définition. Pour éviter d'avoir des flèches dans tous les sens, celles qui vont de droite à gauche sont juste nommées, et certains renvois sont regroupés.

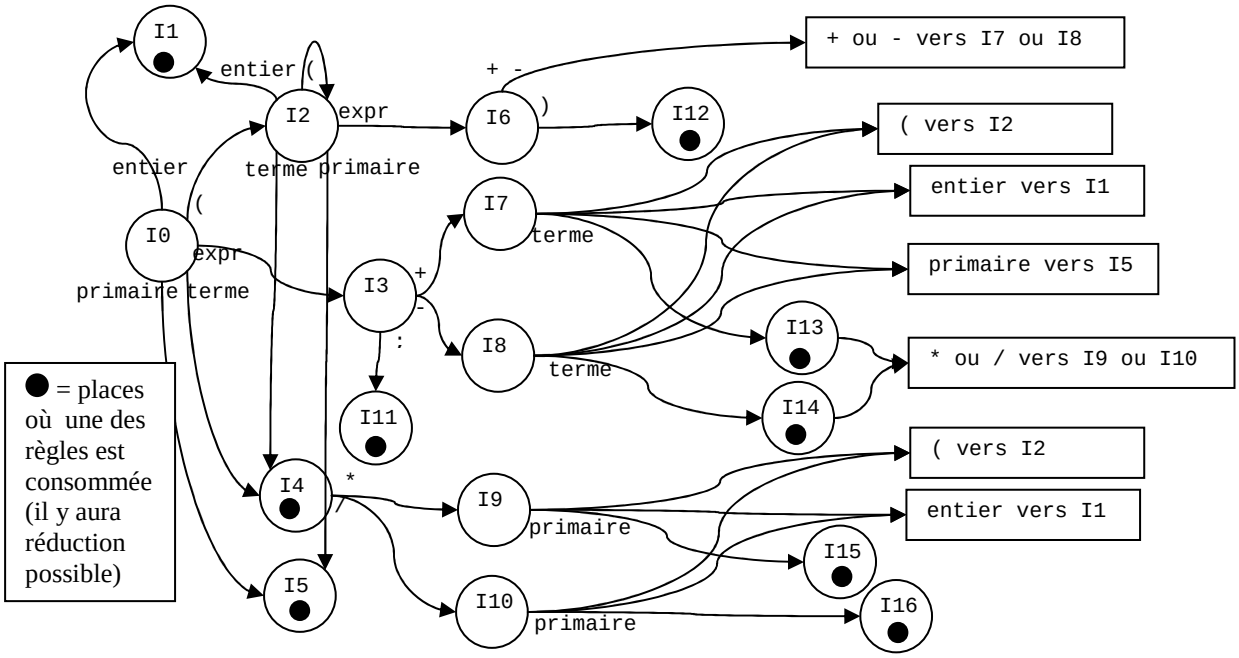


Figure 23 Diagramme de transitions

Construction de la table : verticalement les états, horizontalement les terminaux et non-terminaux. C'est la table des trois actions que peut faire l'automate à pile.

1. gX (goto). signifie « aller à l'état X ». Quand on va d'un IX à un IY par un non-terminal N, on pose l'action gY dans la colonne N.
2. dX (décaler, *shift* en anglais) signifie : « pousser le terminal, avancer au suivant et aller à l'état X ». Quand on va d'un IX à un IY par un terminal T, on pose l'action dY dans la colonne T.
3. Rn signifie: "réduire par la règle numéro n". Quand un IX est consommé (le point est à la fin de la règle), on pose l'action R dans toutes les colonnes de SUIVANT(X) [3]. Pour notre grammaire, les calculs des SUIVANTS ont été fait §2.2.
   Réduire l'axiome revient à accepter la phrase. Il peut advenir des conflits: un conflit entre décalage et réduction (shift/reduce), ou entre deux réductions (reduce/reduce).
   Le *shift/reduce* indique que l'analyseur ne sait pas décider entre avancer ou réduire.
   Cela se produit avec ce genre de règle (c'est ainsi en C):

   ```
   instruction_if →   if (expression) instruction
   instruction_if →   if (expression) instruction else instruction ;
   ```

   L'analyse de la séquence: "`if (x) if (y) instr; else instr;`" sera ambigüe: le **`else`**, d'après les règles, peut appartenir au premier ou au deuxième if.

   Quand l'analyseur tombe dessus, il peut décider de réduire (ramener le **`if`** intérieur à une instruction, et le **`else`** appartient alors au **if** extérieur); ou d'avancer (le **`else`** sera réduit plus tard avec le **`if`** intérieur, et le **`if`** extérieur n'en aura pas. Dans ces cas là,

[3] C'est là la différence avec la méthode d'analyse LALR utilisée par *yacc* et *bison*: dans cette dernière, on poserait l'action seulement dans les colonnes des symboles qui peuvent suivre l'état courant étant donné l'état actuel de l'analyse. C'est le calcul de l'ensemble "Look-Ahead" –le LA de LALR- qui remplacerait celui de Suivant.

l'option par défaut courante est d'avancer. Ce gag est évité dans les langages qui ont un "**end if**".

Attention: il n'y a pas de rapport entre les numéros de règles et les numéros des ensembles! Rappelons donc notre grammaire avec ses numéros de règles.

```
1. axiome → expr ';'
2. expr → expr '+' terme
3. expr → expr '-' terme
4. expr → terme
5. terme → terme '*' primaire
6. terme → terme '/' primaire
7. terme → primaire
8. primaire → '(' expr ')'
9. primaire → entier
```

Dans cette grammaire, en appliquant les règles vues en §2.2

- SUIVANT(axiome) est EOT ($)
- SUIVANT(expr) est { ‘;’ ‘)’ ‘+’ ‘-‘ }
- SUIVANT de terme est { ‘;’ ‘)’ ‘+’ ‘-‘ ‘*’ ‘/’ }
- SUIVANT de primaire est { ‘;’ ‘)’ ‘+’ ‘-‘ ‘*’ ‘/’}

	+	-	*	/	(	)	;	entier	EOT	expr	terme	primaire
0					d2			d1		g3	g4	g5
1	R9	R9	R9	R9		R9	R9					
2					d2			d1		g6	g4	g5
3	d7	d8					d11					
4	R4	R4	d9	d10		R4	R4					
5	R7	R7	R7	R7		R7	R7					
6	d7	d8				d12						
7						d2		d1			g13	g5
8						d2		d1			g14	g5
9					d2			d1				g15
10					d2			d1				g16
11									Accept			
12	R8	R8	R8	R8		R8	R8					
13	R2	R2	I9	I10		R2	R2					
14	R3	R3	I9	I10		R3	R3					
15	R5	R5	R5	R5		R5	R5					
16	R6	R6	R6	R6		R6	R6					

Figure 24 Tables d'analyse

L'automate va partir de l'état 0. En fonction des mots reçus, il va faire ce qui est indiqué dans la case correspondante. Si la case est vide, c'est une erreur.

- S'il tombe sur un goto (gX): il change simplement la ligne considérée, et pousse le symbole d'entrée puis le numéro de l'état où il se trouve alors.
- S'il tombe sur un décalage (shift), il pousse le mot sur la pile, et va sur la ligne considérée; il pousse aussi le numéro de l'état.
- S'il tombe sur une réduction, il considère la règle dont le numéro est donné, et enlève la séquence qui se trouve en haut de la pile qui doit correspondre à sa partie droite; et repousse le non-terminal en question dans la chaîne d'entrée. En principe il y a correspondance exacte, sinon c'est une erreur de construction de la table! Puis il se remet dans l'état dont le numéro est au sommet de pile.

Voyons sur quelques lignes de l'automate que l'on va exécuter complètement ensuite (Figure 26).

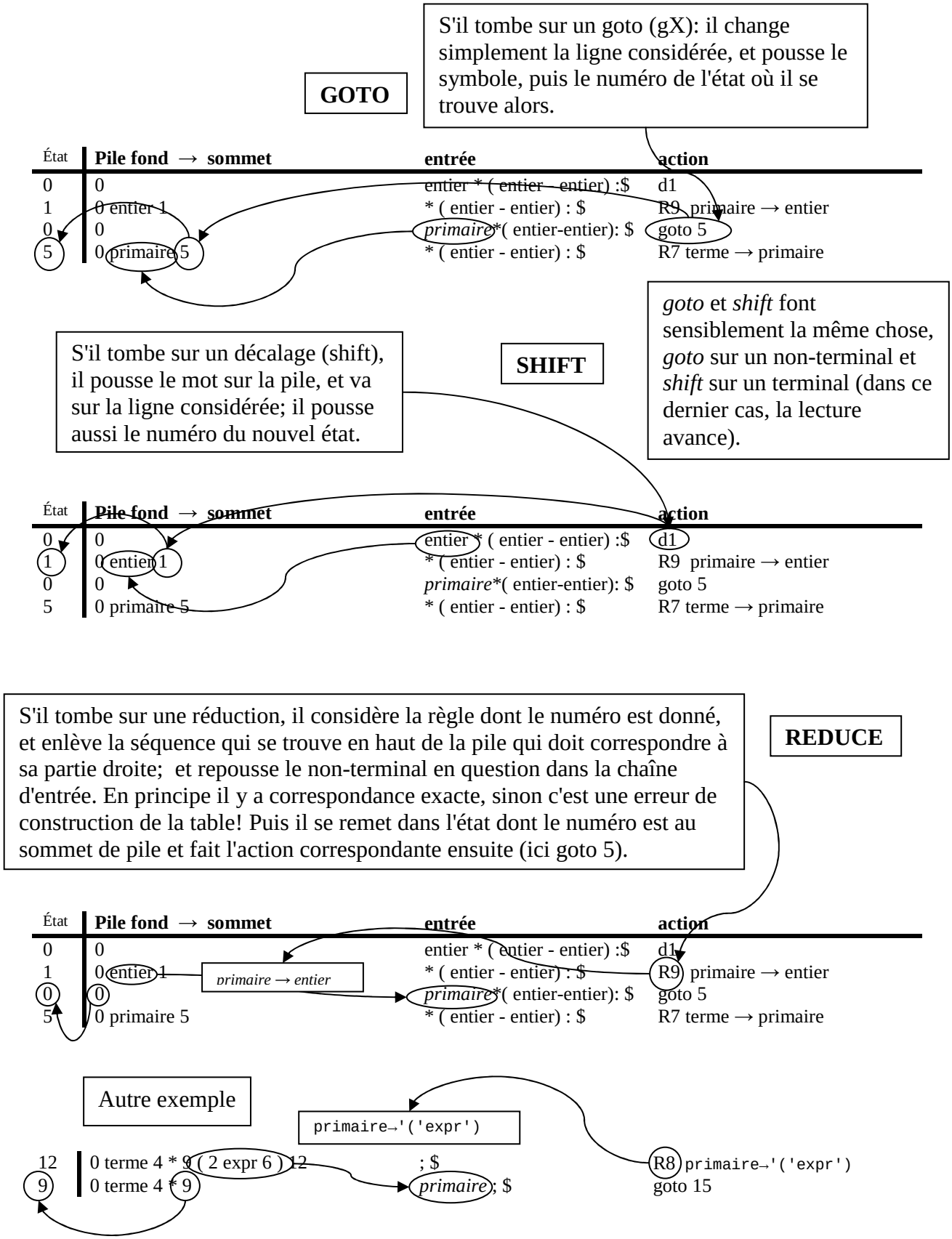


Figure 25 Exemples d'exécution de l'automate

Voyons maintenant l'analyse de "`2* (3 - 4) ;`" qui va arriver à l'analyseur sous la forme de "**`entier * ( entier - entier) ;`**

État	**Pile fond → sommet**	**entrée**	**action**
0	0	entier * (entier - entier) :$	d1
1	0 entier 1	* (entier - entier) : $	R9 primaire → entier
0	0	*primaire**(entier-entier): $	goto 5
5	0 primaire 5	* (entier - entier) : $	R7 terme → primaire
0	0	*terme* * (entier - entier) : $	goto 4
4	0 terme 4	* (entier - entier) : $	d9
9	0 terme 4 * 9	entier - entier) : $	d2
2	0 terme 4 * 9 (2	entier - entier) ; $	d1
1	0 terme 4 * 9 (2 entier 1	- entier) ; $	R9 `primaire → entier`
2	0 terme 4 * 9 (2	*primaire* - entier) ; $	goto 5
5	0 terme 4 * 9 (2 primaire 5	- entier) ; $	R7 `terme → primaire`
2	0 terme 4 * 9 (2	*terme* - entier) ; $	goto 4
4	0 terme 4 * 9 (2 terme 4	- entier) ; $	R4 `expr → terme`
2	0 terme 4 * 9 (2	*expr* - entier) ; $	goto 6
6	0 terme 4 * 9 (2 expr 6	- entier) ; $	d8
8	0 terme 4 * 9 (2 expr 6 - 8	entier) ; $	d1
1	0 terme 4 * 9 (2 expr 6 – 8 entier 1	) ; $	R9 `primaire → entier`
8	0 terme 4 * 9 (2 expr 6 – 8	*primaire*) ; $	goto 5
5	0 terme 4 * 9 (2 expr 6 – 8 primaire 5	) ; $	R7 terme → primaire
8	0 terme 4 * 9 (2 expr 6 – 8	*terme*) ; $	goto 14
14	0 terme 4 * 9 (2 expr 6 – 8 terme 14	) ; $	R3 `expr→expr '-' terme`
2	0 terme 4 * 9 (2	*expr*) ; $	goto 6
6	0 terme 4 * 9 (2 expr 6	) ; $	d12
12	0 terme 4 * 9 (2 expr 6) 12	; $	R8 `primaire→'('expr')`
9	0 terme 4 * 9	*primaire* ; $	goto 15
15	0 terme 4 * 9 primaire 15	; $	R5 `terme→terme'*'primaire`
0	0	*terme* ; $	goto 4
4	0 terme 4	; $	R4 `expr → terme`
0	0	*expr* ; $	goto 3
3	0 expr 3	; $	d11
11	0 expr 3 ; 11	$	ACCEPT

Figure 26 Fonctionnement de l'automate

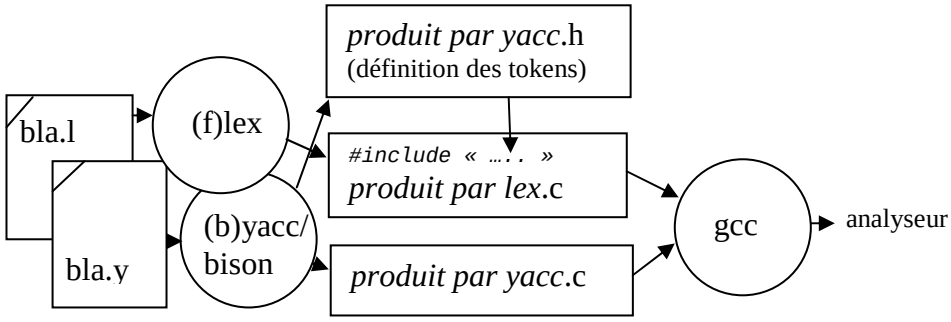
(Rappel de la table de transitions ci-dessous)

	+	-	*	/	(	)	;	entier	EOT	expr	terme	primaire
0					d2			d1		g3	g4	g5
1	R9	R9	R9	R9		R9	R9					
2					d2			d1		g6	g4	g5
3	d7	d8					d11					
4	R4	R4	d9	d10		R4	R4					
5	R7	R7	R7	R7		R7	R7					
6	d7	d8				d12						
7						d2		d1			g13	g5
8						d2		d1			g14	g5
9					d2			d1				g15
10					d2			d1				g16
11									Accept			
12	R8	R8	R8	R8		R8	R8					
13	R2	R2	I9	I10		R2	R2					
14	R3	R3	I9	I10		R3	R3					
15	R5	R5	R5	R5		R5	R5					
16	R6	R6	R6	R6		R6	R6					

6.3 Lex/flex et yacc/bison/byacc

Ce sont les outils classiques les plus répandus pour construire des analyseurs. Le premier analyse des expressions régulières et utilise un algorithme du genre de celui décrit au § 4.4, et le second est LALR qui est une forme proche de la méthode SLR décrire au § 6.2. Voir la note page 34 pour une explication sommaire de la différence entre les deux algorithmes.

Les deux outils ont un format commun et un peu baroque à première vue ; le fichier comprend trois zones délimitées par des symboles %{ , %} et %%. Les deux outils sont faits pour être utilisés ensemble, même si le couplage est assez léger et facile à couper.



Les noms des fichiers produits dépendent de la combinaison d'outils utilisés(lex/flex – yacc/byacc/bison) et du système d'exploitation

Figure 27 Génération de C par Lex/yacc

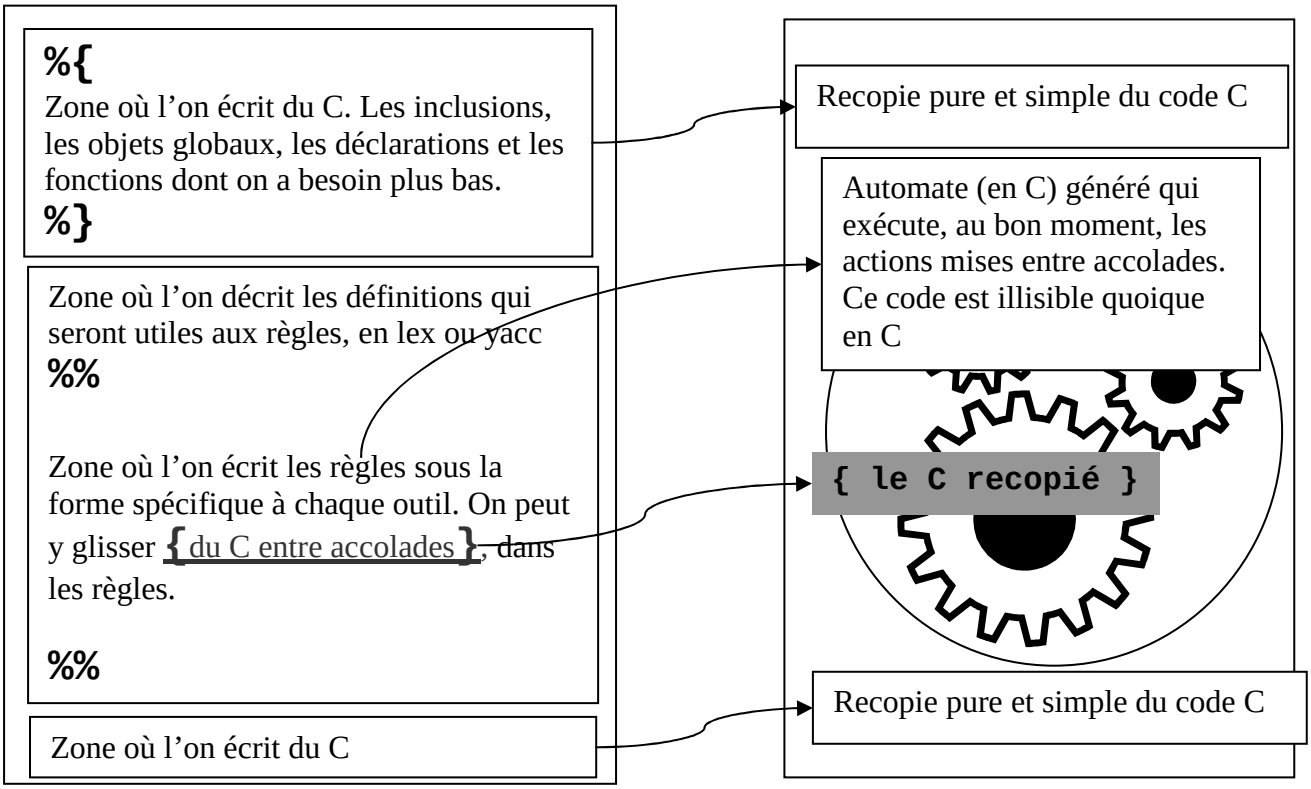


Figure 28 Structure des fichiers lex/yacc et leur transformation en C

On voit sur le schéma ci-dessus que le résultat final de la transformation est un fichier C qui est compilé par un compilateur ; c'est à ce moment là seulement que les erreurs de C seront indiquées, aussi triviales soient-elles.

6.4 LEX

Le travail de LEX est de fournir une fonction **`int`** `yylex()` , qui, à chaque appel (fait par YACC, dans notre cas), rendra un entier codant l'élément lexical reconnu (un mot clé, un nombre, etc..). Ce codage est arbitraire et à la discrétion du programmeur ; néanmoins une « tradition » solide et compréhensible dit que, si l'élément lexical est un caractère unique (comme '+', '-', ' ;', '(') le code sera sa valeur ASCII ; et ceci est utilisable par la syntaxe de YACC. Et si l'élément a plusieurs caractères, son code sera supérieur à 256 pour ne pas interférer avec la convention des caractères uniques. YACC fournit dans un fichier .h une liste de #define codant de cette façon les éléments nommés –les tokens- dans le fichier .y. C'est pourquoi le code C associé à la reconnaissance de '+' finira par …**`return`** `'+' ;}` car en C, le caractère est un entier, qui est sa valeur ASCII. Et le code associé à la reconnaissance de « >= » finira par …**`return`** `GE ;}` où GE (pour Greater or Equal, c'est un nom arbitraire) est une macro définie par exemple à 258 dans le fichier généré par YACC.

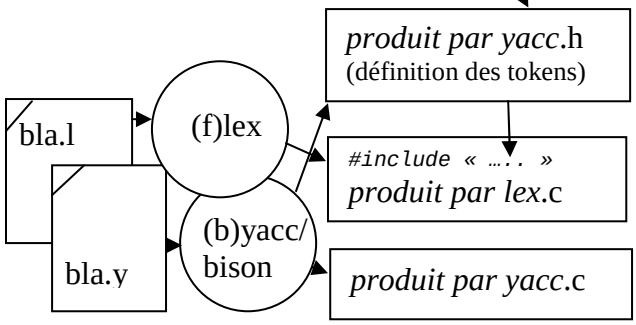


Figure 29 Interaction Lex/Yacc

Les règles utilisent quelques conventions très simples :

- ? : le point d'interrogation est l'option : la chose qui est en préfixe est optionnelle.
- * : Le * est « zéro fois ou plus ». La chose en préfixe peut être répétée, éventuellement absente.
- + : Le + est « une fois ou plus ». La chose en préfixe peut être répétée, mais pas absente.
- […] : entre crochets, un choix de caractères. [abx] veut dire a ou b ou x.
- - : une étendue de caractères : [a-z] veut dire toute lettre minuscule, donc [A-Za-z] veut dire lettre..
- () : on peut parenthéser des symboles avant de leur appliquer les opérateurs ?,*,+.
- ^ : On peut exclure un caractère : ^x veut dire tout sauf x.
- . : le point tout seul signifie qu'on prend tout ; en tant que dernière règle, cela signifie que si aucun token n'a été reconnu, on renonce à faire une erreur et on en laisse la charge à l'étage du dessus, ce qui est souvent une option raisonnable.
- Si on a besoin d'un de ces meta-symboles dans la grammaire, on le préfixe par un \.

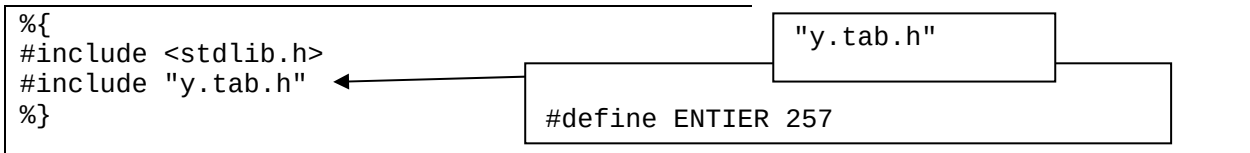
On peut dans LEX définir des macros. Cela se fait dans la zone de définitions, juste avant le %% et après %}. On les déclare très simplement, et on les appelle en les mettant entre accolades :

```
lettre            [A-Za-z]
chiffre           [0-9]
lettre_ou_chiffre {lettre}|{chiffre}
```

Enfin, souvent ce n'est pas tout de reconnaître le numéro du token, on a besoin de sa valeur. Pour cela, un **`char`** `* yytext` pointe toujours sur la chaîne de caractères du dernier token reconnu. À la charge du programmeur que de le convertir en entier, si c'est le cas. Par exemple en écrivant `val = atoi(yytext)` ; où val sera une variable partagée avec le fichier YACC.
Ainsi, on pourra avoir les règles (la règle en gras, puis l'action). :

```
"+" { return '+';}

[0-9]+ {val_entier = atoi(yytext) ; return ENTIER;}

[0-9]+\.[0-9]+([Ee][+-]?[0-9]+)? {sscanf(yytext,"%f",&val_reel); return REEL;}

[ \t] {/* on ne retourne rien sur les espaces et tabs, l'automate continue*/}

\n {num_ligne++; /* pourra server dans les messages d'erreur */}

. {/* on accepte le caractère courant et on ne fait pas d'erreur */}
```

Le petit fichier LEX de la grammaire ressemblera donc, (avant l'introduction des actions), à :




```
%{
#include <stdlib.h>
#include "y.tab.h"
%}

%%
"+" {return '+' ;}
"-" {return '-' ;}
"*" {return '*' ;}
"/" {return '/' ;}
"(" {return '(' ;}
")" {return ')' ;}
";" {return ';' ;}
[0-9]+ {return ENTIER;}
[ \t] {/* on ignore blancs et tabs*/}
[\n] { num_ligne ++;}
. {}
%%
```

Figure 30 Actions en Lex

La seule action consistera à mettre à jour la valeur de l'entier dans une variable globale (déclarée dans le fichier *yacc*, et ici en **extern**. Attention à la syntaxe: il faut des doubles quotes " pour la définition des mots reconnus; rien à voir avec les simples quotes ' encadrant les caractères, qui sont simplement une façon d'écrire un entier en C.

```
%{
...
extern int value;
%}

%%
...
[0-9]+ {value = atoi(yytext) ; return ENTIER;}
...
%%
```

Figure 31 Passage de la valeur d'un token

6.5 YACC

Le travail de *yacc* est de fournir une fonction `int yyparse()` qui rend 0 si l'analyse s'est bien passée, et 1 sinon. Tout le reste de l'activité de l'analyseur ou du compilateur est faite dans les actions émises au fil de l'eau dans la grammaire. L'automate appelle la fonction `yylex()` quand il en a l'usage, cette fonction doit exister. Si on utilise LEX, c'est de là qu'elle viendra ;

Les règles sont données dans une variante de la BNF. Le nom de la règle est suivi de deux points, puis de sa définition, puis d'un point-virgule.

```
regle : définition ;
```

La définition est simplement la liste des terminaux ou non-terminaux dans l'ordre espéré ; Un seul méta-opérateur est possible, le « ou » symbolisé par une barre verticale |. Sa priorité est la plus faible, et on ne peut pas parenthéser. Donc la barre verticale s'applique toujours à des choix qui regardent la règle définie, pas à des sous-cas.

```
regle : A B C | D E F ;
```

Ici, c'est *regle* qui a deux options ABC ou DEF, on ne parle pas d'un choix local entre C et D.

Un élément du vocabulaire terminal doit être symbolisé,

- soit par un caractère entre apostrophe (comme '+'),
- soit par un nom qui doit être déclaré dans une ligne %token qui a sa place dans les définitions (juste avant le %%), comme dans nos exemples : ENTIER.

Si on choisit la notation '+', on compte sur le fait que la fonction `yylex()` rendra la valeur ASCII de '+' sur le terminal en question (qui est sans doute "+" dans le fichier LEX, du moins il faut l'espérer.) Si on choisit la notation nommée (ENTIER), *yacc* fera un fichier .h contenant des `#define` adaptés que pourra récupérer LEX . Par exemple :
`#define ENTIER 257`.

Là-dessus, *yacc* appelle une fonction yyerror(char *) quand il y a une erreur, il faut la fournir. Et il faut aussi une fonction `yywrap(),` vide ici, qui sert à changer le fichier ouvert quand on lit une grammaire qui change de fichier, par exemple en C quand on revient d'un `#include`: l'automate appelle yywrap pour donner l'opportunité de fermer le fichier inclus et de revenir au fichier incluant.

Par exemple, notre grammaire d'expression sera :

```
%{
ici du C
%}


%TOKEN ENTIER
%START debut
%%

debut: expr ';'

expr : expr '+' terme | expr '-' terme | terme ;

terme  :  terme '*' primaire | terme '/' primaire | primaire ;

primaire : ENTIER | '(' expr ')' ;

%%


int main ()
{
  if (yyparse()==0) printf("ok\n") ; else  printf( "ko\n")
}

int yyerror(char * s)
{
  printf( "erreur %s\n",s);
}
int yywrap() {
}
```

Figure 32 Grammaire d'expression en Yacc

La ligne %TOKEN est celle qui définit les valeurs numériques des symboles terminaux nommés. La ligne START définit l'axiome. Ici l'axiome ne fait que renvoyer sur expr, c'est parce qu'il est toujours intéressant d'avoir une entrée et une sortie de la grammaire qui ne soient pas appelées par la grammaire; cela permet de définir des zones d'initialisation et de finalisation.

7 Faire un interpréteur en Lex/Yacc [TP]

Un interpréteur est un analyseur qui prend des actions au fil de l'eau de l'analyse. Par exemple, les *shells* d'Unix sont basés sur des interpréteurs; quand on tape une commande, elle est exécutée sur le champ.

Il s'agit donc de farcir notre description (LEX et YACC) avec des actions (voir figure page 38).

Pour mettre des actions là-dedans, nous allons supposer que nous avons une pile. Il y a trois endroits clés: là où l'on pousse un entier littéral dans la pile (dans primaire), là où l'on affiche le résultat (en sortant de `expr`) et là où l'on fait les calculs (sur chaque opérateur).

```
debut: expr {printf(%d\d, sommet_pile()); tirer_pile();} ';'

expr : expr '+' terme {
                        int x,y;
                        x=sommet_pile(); tirer_pile();
                        y=sommet_pile(); tirer_pile();
                        pousser_pile(y+x);
                      }
…pareil pour les autres opérateurs. Attention à ce que l'opérande de gauche
est le dernier tiré (y).

primaire : ENTIER {pousser_pile(value);}  | '(' expr ')' ;
```

Figure 33 LEs actions dans la grammaire d'expression de Yacc

La structure générale d'un interpréteur ressemble donc à ceci:

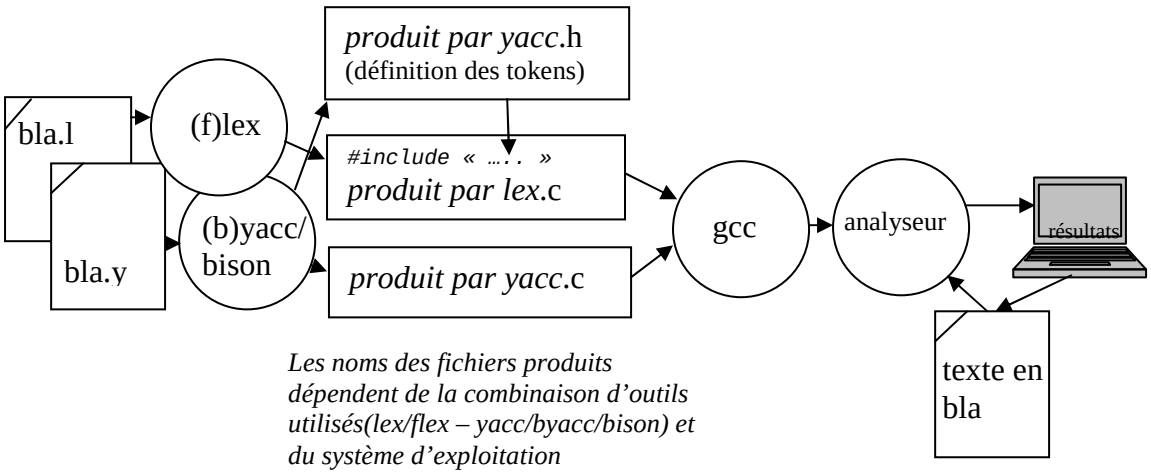


Figure 34 Structure d'un interpréteur basé sur Lex/Yacc

Le *makefile* correspondant est:

```
bla: y.tab.o lex.yy.o Makefile

gcc -g y.tab.o lex.yy.o -o bla

lex.yy.o: lex.yy.c y.tab.h Makefile
	gcc -c -g lex.yy.c

y.tab.o: y.tab.c y.tab.h  Makefile
	gcc -c -g y.tab.c

y.tab.h y.tab.c: bla.y Makefile
	yacc -v -d bla.y

lex.yy.c: bla.l Makefile
	flex bla.l
```

Pour BISON, remplacer y.tab.c, y.tab.h, y.tab.o par bla.tab.c, bla.tab.h, bla.tab.o (bla est le nom du fichier qui contient la grammaire.

Figure 35 Makefile Interpréteur

Ici on suppose que y.tab.c et y.tab.h sont produits par l'analyseur du .y, et lex.yy.c par l'analyseur du .l.

Notes :

Remarquer que le Makefile est sensible sur ses propres modifications, en effet quand on modifie le Makefile on a souvent envie de tout recompiler ensuite.
L'option –v de yacc/bison provoque l'écriture d'un fichier décrivant l'automate (y.output)
L'option –g de gcc compile en mode débug.

8 Transformer l'interpréteur en compilateur [TP]

Un compilateur produit un fichier dit "objet" destiné à être exécuté par un processeur (compilateur C) ou par une pseudo-machine (Java). C'est cette dernière approche que nous allons implémenter.

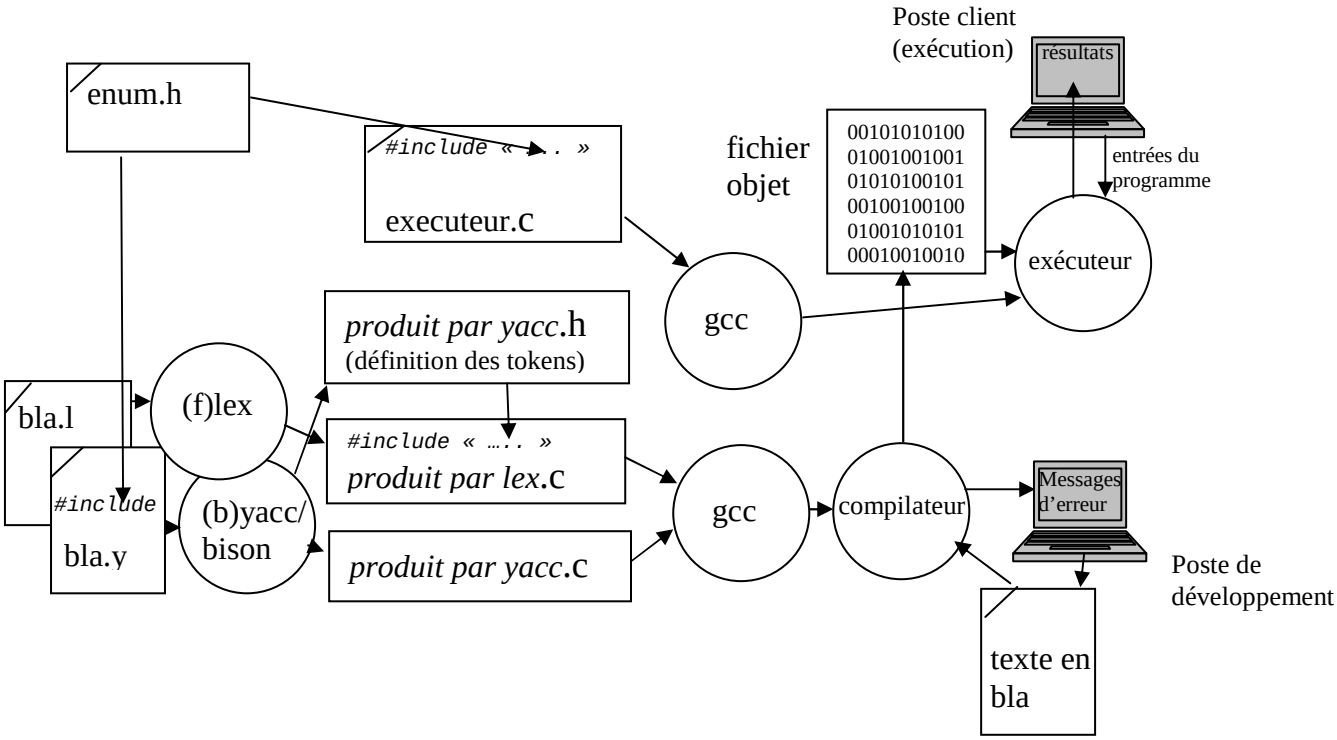


Figure 36 Structure d'un compilateur basé sur Lex/Yacc

Le Makefile :

```
tout: executeur bla Makefile

executeur: executeur.c enum.h Makefile
	gcc executeur.c –o executeur

bla: y.tab.o lex.yy.o Makefile
	gcc -g y.tab.o lex.yy.o -o bla

lex.yy.o: lex.yy.c y.tab.h Makefile
	gcc -c -g lex.yy.c

y.tab.o: y.tab.c y.tab.h  Makefile
	gcc -c -g y.tab.c

y.tab.h y.tab.c: bla.y enum.h Makefile
	yacc -v -d bla.y

lex.yy.c: bla.l Makefile
	flex bla.l
```

Pour BISON, remplacer y.tab.c, y.tab.h, y.tab.o par bla.tab.c, bla.tab.h, bla.tab.o (bla est le nom du fichier qui contient la grammaire.

Figure 37 Makefile compilateur

Notes :
Le fichier Makefile a deux cibles terminales : *bla* et *executeur*. On crée une cible fictive « tout » qu'on met en première ligne, qui appelle les deux autres.
Le fichier « enum.h » est le lien entre compilateur et exécuteur, il contient la convention de codage des opérations élémentaires.

8.1 Structure des fichiers

8.1.1 enum.h

Le fichier `enum.h` contient simplement la définition d'un *enum*. Dans sa première version, il ressemblera à :

```
enum {pousser, plus,  moins, multiplier, diviser, afficher, fin} ;
```

8.1.2 Le fichier objet

Le fichier objet sera une succession d'entiers, chacun correspondant en général à un des codes du type énuméré. Ainsi le « programme »

```
2+3 ;.
```

…sera stocké dans le fichier objet par la séquence symbolique :

```
pousser
2
pousser
3
plus
afficher
fin
```

…c'est-à-dire par la séquence réelle : 0 2 0 3 1 5 6
On voit que certaines « instructions » sont autosuffisantes (plus), certaines autres demandent un argument (pousser).
Ceci est fait par le compilateur qui n'a plus de pile de calcul. Les différentes actions de l'interpréteur sont remplacées par l'écriture dans le fichier objet. Ainsi la séquence
`x=TOP ; PULL ; y=TOP ; PULL ; PUSH(x+y)`
sera simplement remplacée par « `ecrire(plus); »`
De même, l'instruction PUSH(valeur) qui apparaît dans primaire, deviendra :
`« ecrire(pousser) ; ecrire(valeur) ; »`
La gestion des variables demandera quelques items spécifiques supplémentaires dans `enum.h`

8.1.3 L'exécuteur

L'exécuteur est un programme très simple dont la structure est la suivante :

```
inclure enum.h
définir la pile
ouvrir le fichier objet
boucle infinie :
      lire l'entier suivant
      switch (la valeur)
        case pousser: lire l'entier suivant; le pousser sur la pile ;break;
        case plus: x=TOP ; PULL ; y=TOP ; PULL ; PUSH(x+y) ; break;
        etc…
        case fin: fermer fichier; exit(0);
      fin switch
fin boucle
```

Modifications nécessaires

Entrées :
Il ne sert à rien d'avoir un langage compilé qui n'a pas d'entrées ! On ajoutera le symbole « ? » dans le fichier de *lex*, et dans primaire on mettra une quatrième branche avec un item spécifique dans le enum.h :

```
'?'  {ecrire(entree);}
```

La branche correspondante de l'exécuteur sera:

```
case entree : { printf(":"); scanf("%d" ,&tmp) ; PUSH(tmp) ; break;}
```

Ceci permettra d'écrire des expressions comme (2+?)*4 ; où le programme s'arrête pour demander la valeur qu'il faut mettre à la place de « ? ».

Structure du fichier objet :
Dans un premier temps, pour pouvoir inspecter à l'éditeur le fichier objet, on écrira par *fprintf* et on lira par *fscanf*.
Pour pouvoir introduire des instructions conditionnelles et de boucle, il est nécessaire que chaque code dans le fichier objet occupe la même place pour pouvoir calculer les branchements, autrement dit on écrira ensuite par *fwrite* et on lira par *fread*, ce qui rendra le fichier objet illisible à l'éditeur de texte (entiers codés sur un mot de 32 bit).

Mise en mémoire :
Pour gérer les boucles (le même code étant exécuté plusieurs fois) il faudra que l'exécuteur charge le fichier en mémoire d'un coup, et lise ensuite les codes dans un tableau d'entiers *ad-hoc*.
Pour mettre le contenu d'un fichier (binaire) en mémoire, ici un tableau d'entiers:

Déclarer un **`int * p`**

- Mesurer la taille du fichier. Pour cela une seule méthode à peu près portable, bien que cela ne soit pas garanti par la norme :
  - Ouvrir le fichier : **`fopen(…)`**
  - Aller à la fin (fonction **`fseek(…)`**)
  - Lire où on est (**`ftell(…)`** rend le nombre d'octets depuis le début)

    - Revenir au début du fichier (**`fseek(…)`**)
- Faire le **`malloc()`** sur p avec la taille mesurée.
- Charger la zone mémoire allouée avec le contenu du fichier. Une seule instruction : **`fread(…).`** Prendre garde que l'on a une taille en octets, et que **`fread`** demande le nombre d'objets et la taille de chaque objet, ici l'entier a en principe une taille de 4 octets. Utiliser **`sizeof(int).`**
- Penser à fermer le fichier : **`fclose(…).`**

Complications de la grammaire d'expressions :
Pour les tests nous aurons besoin de comparateurs. Il faudra ajouter un niveau à notre grammaire d'expressions (elle en a trois : *expr*, *terme*, *primaire*). Ce niveau sera « au dessus » car les comparateurs ont la priorité la plus faible. La grammaire aura finalement quatre niveaux :
`expr, comp, terme, primaire`

`comp` prendra la place de l'ancienne `expr`, et `expr` ressemblera à :

expr : comp EQ comp
| *comp NE comp*
| *etc.*

Noter la convention classique EQ pour equal, NE pour not equal, GT et LT pour greater/lesser than, GE et LE pour greater/lesser or equal. Cela nous fera 6 tokens de plus, à reconnaître dans le fichier lex sous les symboles habituels = != > < >= <=, ainsi que 6 opérateurs à mettre dans enum.h. Attention, cette règle ne doit pas être récursive à gauche, car A = B = C produirait des résultats non intuitifs, cela serait lu comme (A=B) = C là où on s'attendrait à lire (A=B) and (B=C) .

L'exécuteur utilisera la convention C : 0=faux, 1=vrai.

8.2 Notion d'instructions

Nous voulons faire un langage dont les programmes seront, au début, une simple séquence d'instructions. Deux de ces instructions existent déjà :

- L'affichage d'une valeur : 2 ; imprime « 2 » , X+4 ; imprime la valeur.
- L'affectation d'une variable : A=3 ;

À terme nous en aurons plus, comme les instructions if, de boucle, etc. Le programme sera donc une règle écrite ainsi :

```
programme : sequence '.' ;
```

La séquence d'instruction est définie:

```
sequence : sequence  instruction | ;
```

Et l'instruction est une règle dont deux branches sont celles déjà écrites :

```
instruction : affectation | affichage | etc etc ;
affectation : LETTRE '=' expression ';' ;
affichage : expression ';' ;
```

8.2.1 Test: l'Instruction if

On veut pouvoir écrire

```
if expression then sequence endif ;
if expression then sequence else sequence endif ;
```

On se force à mettre un **endif** pour éviter la question de conflit shift/reduce évoquée au §6.2.

Pour éviter des ambigüités dans la grammaire, il faudra factoriser ces deux règles en définissant un debut_if:

```
debut_if : IF expression THEN sequence ;
```

La règle if aura ensuite deux branches appelant le même préfixe, l'une se terminant par ENDIF et l'autre avec la clause ELSE. On voit que pour implémenter cela, nous avons besoin d'une instruction de branchement si le test est faux (donc s'il y a 0 sur la pile), et d'une instruction de branchement inconditionnel, puisque si l'on a exécut la branche THEN, il ne faut jamais faire la branche ELSE.

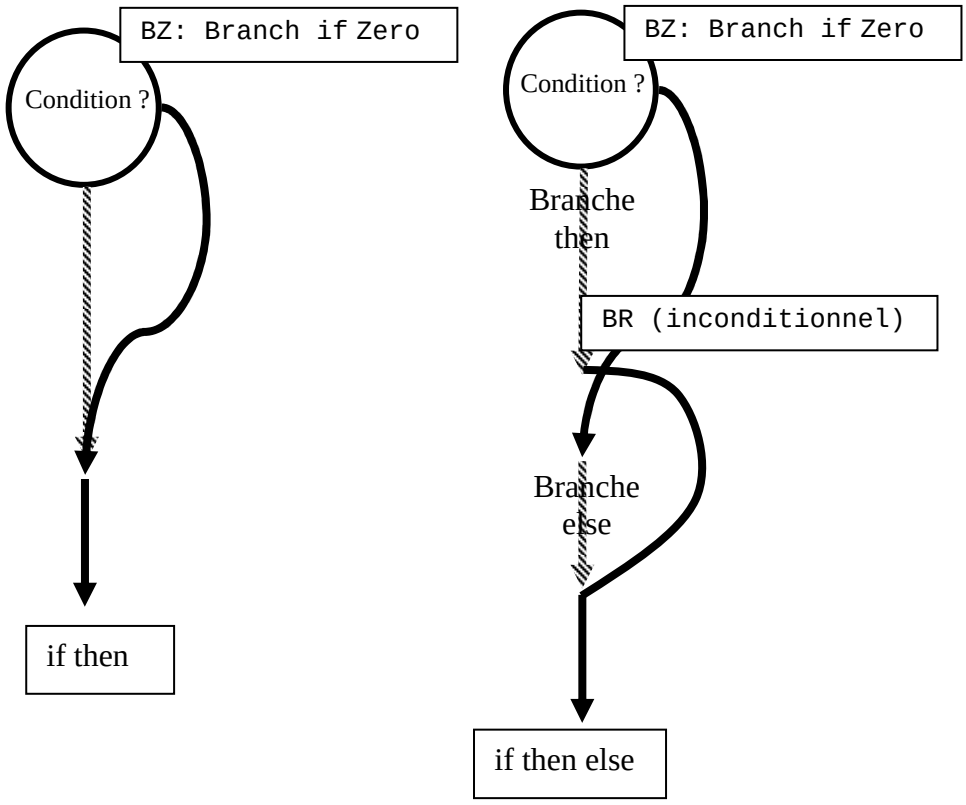


Figure 38 Exécution d'un if/then et d'un if/then/else

Le tableau suivant représente sur plusieurs colonnes et en vis-à-vis le code source qui est lu par le compilateur, un bref commentaire, l'adresse où est déposé le code assembleur compilé, ce code, et enfin un exemple possible des actions nécessaires pour cette production.

Programme	Commentaires	Adresse	Code exécuté en machine	Création de code et jeux de pile
if	Pas d'action			
(condition)	Après évaluation, le sommet de la pile de calcul contient 0 ou 1. Pas d'action spécifique.		(ici le code nécessaire à l'évaluation de la condition)	Ici la simple exécution de l'analyse de l'expression produit le code nécessaire
then	Branchement si le sommet de la pile de calcul vaut 0.		BZ	Écrire BZ
		A1 :	XX puis A2	Pousser A1 (pile) Écrire (n'importe quoi) pour réserver la case mémoire : on ne connaît pas encore la valeur de A2.
(séquence d'instructions contenant éventuellement d'autres « if »)			(ici le code nécessaire à l'exécution de la séquence d'instructions)	
end if ;		A2 :		Tirer le sommet de pile : on trouve A1. Aller en A1 et y écrire A2 qu'on connaît maintenant. Revenir en A2.

Figure 39 Détail des actions à faire pendant la compilation d'un if/else

Le tableau suivant est de même nature que le précédent :

Programme	Commentaires	Adresse	Code produit	Création de code
if	Pas d'action			
(condition)	Après l'évaluation, le sommet de la pile de calcul contient 0 ou 1		(ici le code nécessaire à l'évaluation de la condition)	Ici la simple exécution de l'analyse de l'expression produit le code nécessaire.
then	Branchement si le sommet de la pile de calcul vaut 0.		BZ	Écrire BZ
		A1 :	XX puis A4	Pousser A1 sur la pile. Écrire (n'importe quoi): on ne connaît pas encore la valeur de A4.
(séquence d'instructions contenant éventuellement d'autres « if »)			(ici le code nécessaire à l'exécution de la séquence d'instructions)	
else	Branchement inconditionnel (si on a exécuté la branche then, on ne fait jamais la branche else)	A2 :	BR	Tirer le sommet de pile : on trouve A1. Aller en A1 et y écrire A4 qu'on connaît maintenant (c'est A2+2). Revenir en A2. Écrire BR,
		A3 :	XX puis A5	Pousser A3 (pile). puis écrire n'importe quoi pour réserver la place.
(séquence d'instructions contenant éventuellement d'autres « if »)		A4 :	(ici le code nécessaire à l'exécution de la séquence d'instructions)	
end if ;		A5 :		Tirer le sommet de pile : on trouve A3. Aller en A3 et y écrire A5 qu'on connaît maintenant. Revenir.

Figure 40 Détail des actions à faire pendant la compilation d'un if/then/else

On remarque sur les exemples précédents qu'en faisant un peu attention, le code produit à la fin du « if » (c'est-à-dire sur le « end if ») est le même qu'il y ait une branche *else* ou pas. Cette propriété est utile quand on veut implémenter le fait que la branche *else* est optionnelle. On ne détaillera pas ici le cas de la branche « elsif » qui est de même nature.

8.2.2 Boucle: l'instruction loop

On se simplifie la vie en ne faisant que la boucle infinie, et une instruction exit.

loop : LOOP sequence ENDLOOP ;

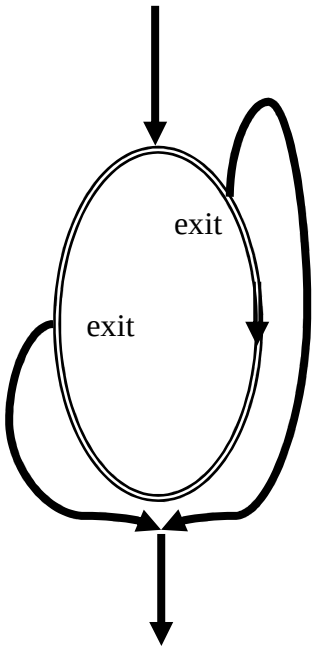
L’instruction exit est à ajouter dans les instructions à côté de l’affectation, etc.

```
instruction_exit: EXIT ";" ;
```

Comme elle peut être mise dans un if, cela permet de faire toutes les boucles possibles.

L'examen de la situation sur le schéma ci-dessous montre que nous n'avons besoin que de l'instruction de branchement inconditionnel.

Figure 41 Exécution d'une boucle



Programme	Commentaires	Adresse	Code produit	Création de code
loop		A1		Pousser A1 dans la pile Puis pousser un marqueur (-1 par ex.)
(séquence d'instructions contenant éventuellement d'autres « loop »)			(ici le code nécessaire à l'exécution de la séquence d'instructions)	
exit	Branchement vers la fin de la boucle		BR	Écrire BR
		A2 :	XX puis A4	Pousser A2 sur la pile. Écrire n'importe quoi: on ne connaît pas encore la valeur de A4.
(séquence d'instructions)			(ici le code nécessaire à l'exécution de la séquence d'instructions)	
exit	Branchement vers la fin de la boucle		BR	Écrire BR
		A3 :	XX puis A4	Pousser A3 (pile). Écrire n'importe quoi pour réserver la place.
(séquence d'instructions)			(ici le code nécessaire à l'exécution de la séquence d'instructions)	
end loop ;		A4 - 2 :	BR A1	Tirer le sommet de pile : on trouve A3. Aller en A3 et y écrire A4 qu'on connaît maintenant (c'est l'adresse courante +2). Recommencer (ici une fois seulement avec A2 où on écrira aussi A4) jusqu'à ce qu'on trouve le marqueur (-1). Tirer le marqueur, puis A1 qui est en dessous. Revenir et écrire le saut à A1 (BR A1)
(suite du programme)		A4 :	(ici le code pour l'exécution de la suite)	

Figure 42 Détail des actions à faire pendant la compilation d'une boucle

8.2.3 Exemples pour tester

Le premier programme "réel": On peut maintenant exécuter un petit algorithme très simple, par exemple l'extraction de tous les diviseurs d'un entier. Avec la syntaxe décrite ci-dessus, cela donnera:

```
X=?;
N=X/2;
loop
  if N==0 then exit; end if;
  if X/N*N==X then N; end if;
  N=N-1;
end loop;
.
```

On peut raffiner en n'imprimant que les diviseurs premiers, cela permet de tester les boucles et if imbriqués.

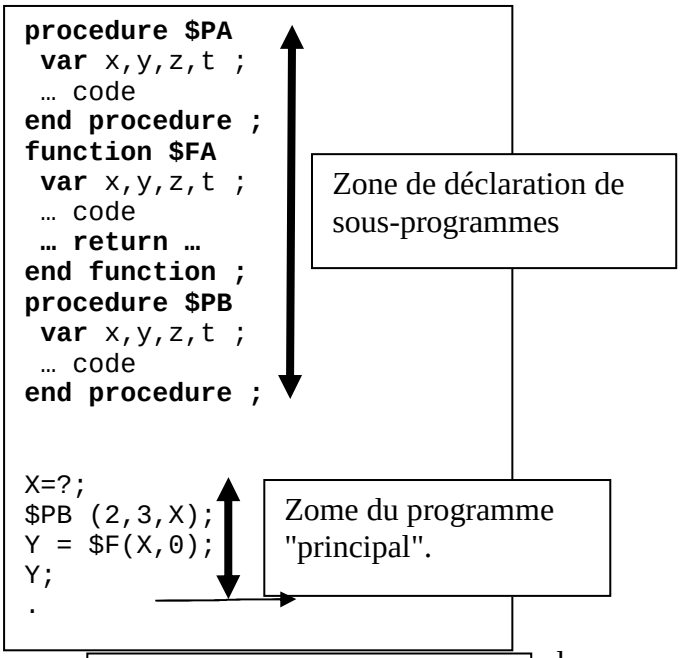
```
X=?;
N=X/2;
loop
  if N==0 then exit; end if;
  if (X/N)*N==X then
    Z=N/2+1;
    loop
      if Z==1 then N; exit; end if;
      if (N/Z)*Z=N then exit; end if;
      Z=Z-1;
    end loop;
  end if;
  N=N-1;
end loop;
.
```

8.3 Les sous-programmes

- On implémentera les sous-programmes récursifs : la procédure et la fonction. Tant qu'on n'a pas de module « table des symboles » on se limitera à des noms d'une lettre avec un préfixe.
- On préfixera les noms de procédure par $P pour éviter quelques ambiguïtés dans la grammaire : dans *lex* : **`$P[A-Za-z0-9] {return NOM_PROCEDURE ;}`**
- L'appel de fonction sera un primaire de plus. Toujours pour éviter une ambiguïté (avec la référence à une variable) on utilisera préfixera les noms de fonction avec F (**`$F[A-Za-z0-9]`**). Exemple : **`X=3+$FA(arg1) ;`**
- On décrètera que les majuscules sont réservées aux variables globales, et les minuscules aux variables locales (modifier le lexique).

La structure d'un programme ressemblera maintenant à ceci:

- Une zone déclarative, où l'on déclare les sous-programmes dans l'ordre qui va bien (il faut déclarer avant d'appeler, ce qui permet la récursivité simple mais pas la récursivité croisée).
- Une zone d'exécution, qui correspond au main de C. C'est là que l'exécution commencera en appelant –ou pas- les sous-programmes définis dessus.



8.3.1 La déclaration

Les sous-programmes seront déclarés avant leur appel.

On remaniera le concept de programme de façon qu'on déclare tous les sous-programmes avant la séquence de calcul principale. **`return`** devient une instruction (comme l'était **`exit`**). Cela implique de « sauter » les sous-programmes au lancement du programme et donc de commencer la génération de code par un BR (adresse du début de la séquence) avec un aller-retour comme d'habitude pour « patcher » l'adresse cible. Cette fois pas besoin de manipulations de pile, les adresses du BR XXX étant connues (0 et 1).

```
procedure $PA
 var x,y,z,t ;
 … code
end procedure ;
```

```
function $FA
 var x,y,z,t ;
 … code
 … return …
end function ;
```

On utilisera dans notre langage une vieille facilité des premières implémentations de C : on ne distingue pas les arguments du sous-programme des variables locales (car en « interne » ils ont de toute façon le même statut.)

Donc dans l'exemple ci-dessus, si **$**PA est appelée avec deux arguments comme dans **$**PA(3,4), ceux-ci seront déposés en x et y. On ne vérifie donc pas que le nombre d'argument passé est correct, ce qui serait possible évidemment avec un peu plus de temps.

```
liste_noms : LETTREMIN | LETTREMIN ',' liste_noms ;
var_locales : VAR liste_noms ';'  | ;
decl_procedure : PROCEDURE NOM_PROCEDURE
                          var_locales
                          sequence
                        END PROCEDURE ';'
decl_fonction : (idem)
decl_sous_prog: decl_procedure  | decl_ fonction ;
liste_decl_sous_progr : decla_sous_prog liste_decl_sous_progrs | ;
programme : liste_decl_sous_progr sequence '.'  ;
```

Figure 43 Syntaxe pour la déclaration des sous-programmes

8.3.2 L'appel

l'appel de procédure est une *instruction* de plus. L'appel de fonction est un *primaire* de plus.
Les arguments seront passés par valeur (comme en C).

```
liste_param : comp | comp ',' liste_param ;
parametres : liste_param | ;
appel_procedure : NOM_PROCEDURE '(' parametres')' ;
appel_fonction : NOM_FONCTION '(' parametres')' ;
```

Figure 44 Syntaxe pour l'appel de sous-programmes

8.3.3 La génération de code

Il faudra deux tableaux int[128] dans le fichier YACC pour stocker les adresses des procédures et des fonctions. Chaque fois qu'on passe sur une déclaration de sous-programme, on sauve dans ces tableaux l'adresse du début de la génération de code du sous-programme.

Il faut aussi un tableau de service int [128] dans le fichier YACC pour les variables locales des sous-programmes. Comme on ne peut travailler que sur une déclaration de sous-programme à la fois, le même tableau peut servir pour toutes. Ce tableau contiendra la position de chaque argument dans la liste : en effet l'exécuteur ne connaîtra que cela, qui servira d'offset dans la pile.

Donc si on a : **`procedure`** `$PP` **`var`** `x,y,z,t …` x est le premier, y le second, etc. et TabArg['x'] contiendra 0, TabArg['y'] contiendra 1, etc.

Si ensuite dans le code de la procédure, on a une référence à y, le code envoyé à l'analyseur sera, selon qu'on est dans une expression ou dans une affectation:
`ref_locale 1`
ou
`affecter_local`
(**`ref_locale`** et **`affecter_locale`** étant des actions de enum.h).

Le but du jeu est que l'exécuteur travaille avec une pile construite ainsi :

Une variable locale « base_pile » dont la valeur est l'adresse de la première variable locale.

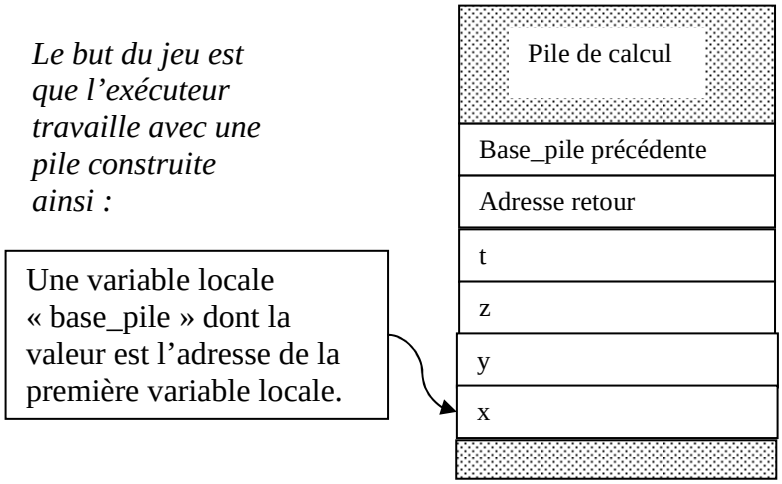


Figure 45 Pile telle qu'on veut la construire pour l'exécution du sous-programme

L'exécuteur ira alors chercher la valeur située à base_pile+1 dans le cas de ref_locale, et affectera base_pile+1 dans le cas de affecter_locale (le 1 étant dans ce cas tiré de la pile).

Génération de code pour l'appel de sous-programme :

Exemple : **`$PA(2,X+3) ;`**

Remarquer que, jusqu'à la parenthèse fermante, l'évaluation des expressions conduira naturellement à ce que la pile contienne X+3 au sommet, et 2 en dessous, sans faire d'action spécifique. Sur la parenthèse fermante : pousser l'adresse de retour (celle où l'on se trouve+1). Pousser le nombre d'arguments passés qu'on aura comptés dans la règle liste_params. Écrire un branchement (BR) à l'adresse trouvée dans le tableau des adresses de procédures.

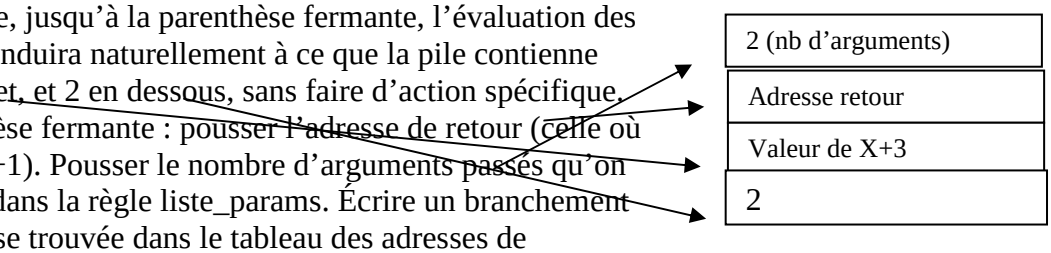


Figure 46 Pille telle qu'elle est juste avant le branchemen,t au sous-programme.

Génération de code du sous-programme :

Exemple : **`procedure`** `$PA` **`var`** `x,y,z,t ;…`

Remarquer que, dans le cas de l'appel du paragraphe précédent, x et y ont déjà une place dans la pile, contenant 2 et la valeur de X+3. Il faut donc réserver seulement z et t qui sont les variables locales de notre procédure. Pour cela, il faut une action spécifique (enum.h) qui dise à l'exécuteur combien il y a de variables en tout pour qu'il puisse faire la soustraction d'avec le nombre passé dans la pile. On les comptera dans la règle liste_noms, on poussera le nombre sur le point-virgule, ou 0 s'il n'y a rien (action à mettre dans la règle appelante). Bref, ici l'action « nbargs 4 » qui marque le début du sous-programme sera passée à l'exécuteur, et son effet sera de

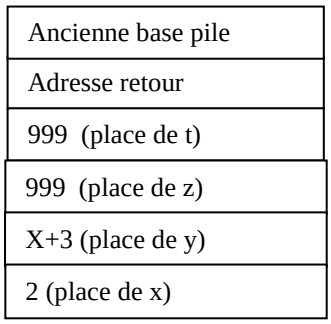


Figure 47 Pile telle qu'elle doit être juste après le branchement au sous-programme

- Sauver l'adresse de retour dans une variable locale.
- Calculer l'adresse du premier argument poussé, ici le 2 (c'est la position du sommet de pile moins le nombre d'arguments poussés (déposé sur le sommet de pile) moins un (la place occupée par ce nombre).
- TIRER le nombre d'arguments passés.
- POUSSER (n'importe quoi, il s'agit juste de réserver la place) 4 – 2 fois. 4 étant le nombre de variables locales passé dans l'action nbargs, 2 étant trouvé sur le sommet de pile à l'item précédent et étant le nombre d'arguments déjà poussés.
- POUSSER l'adresse de retour précédemment sauvée.
- POUSSER l'ancienne base des variables (variable *base_pile*)
- Mettre dans *base_pile* l'adresse du premier argument poussé.

8.3.4 L'instruction de fin de sous-programme

Deux actions RTNP et RTNF vont provoquer la mise à jour de la pile.

- Dans le cas de la procédure (RTNP) :
  - Sauver le sommet de pile dans une variable (l'ancienne base). Tirer.
  - Sauver aussi l'adresse de retour.
  - Positionner le sommet de pile d'après la valeur de « base_pile ». ». Cela dépile d'un coup.
  - Restaurer « base_pile » avec la valeur sauvée dans le premier item.
  - Faire un BR à l'adresse de retour sauvée.

- Dans le cas de la fonction (RTNF), le sommet de pile contient la dernière expression évaluée, celle qui est après le mot-clé **`return`**. Il faudra que cette valeur se retrouve encore en sommet de pile une fois celle-ci « dégonflée ».
    - Sauver la valeur en question dans une variable locale. Tirer.
    - Sauver le sommet de pile dans une variable (l'ancienne base). Tirer.
    - Sauver aussi l'adresse de retour.
    - Positionner le sommet de pile d'après la valeur de « base_pile ». Cela dépile d'un coup.
    - Restaurer « *base_pile* » avec la valeur sauvée dans le deuxième item.
    - POUSSER la valeur sauvée dans le premier item.
    - Faire un BR à l'adresse de retour sauvée.

8.3.5 Les références aux variables locales

Chaque référence **à droite** d'une affectation (r-value) donnera le code
`Ref_locale N` où N est l'offset de la variable locale par rapport à *base_pile*.

Chaque référence **à gauche** d'une affectation (l-value) donnera

- Comme dans l'affectation de variables globales, on pousse la valeur correspondante (ici l'offset par rapport à la base)
- À la fin de l'affectation, une instruction **`affecter_locale`** sur le modèle de l'affectation globale sauf qu'au lieu de taper dans un tableau global, on tape dans la pile avec l'offset donné. Noter qu'il y a maintenant deux règles d'affectation (de variable globale ou locale) correspondant à deux instructions.

8.3.6 Des programmes de test:

Factorielle ($F_n = N * F_{n-1}$)

```
function $FA
var x;
  if x == 1 then return 1;
  else return x * $FA(x-1);
  end if;
end function;

X=?;

$FA (X);
.
```

Fibonnacci ($F_n = F_{n-1} + F_{n-2}$; doublement récursive, très coûteuse en temps et en pile, on pourra comparer le temps d'exécution avec la même fonction écrite en C).

```
function $FI
  var x;
  if x<=2 then return 1;
  else return $FI(x-1) + $FI(x-2);
  end if;
end function;

X=?;
$FI (X);
.
```

8.4 Complications

- Facile:
  - En utilisant une table des symboles, autoriser des variables de longueur quelconque. Pour les noms de procédures et les fonctions, il faut soit avoir plusieurs tables de symboles, soit ajouter dans la table un moyen de flécher les noms (variable, procédure, fonction).
  - Obliger à déclarer les variables globales qu'on utilise.
  - Distinguer les arguments passés des variables locales dans les procédures.
- Difficile:
  - Typage : autoriser d'autres types que les entiers dans les expressions (réels, chaînes). Pour cela,
    - soit on marque les valeurs par leur type avec d'éventuelles erreurs de type à l'exécution (chaque fois qu'on dit à l'exécuteur de pousser une valeur, on lui dit ensuite de pousser à côté l'indication sur son type ; chaque fois qu'on dépile, on dépile aussi le type et on choisit l'opération en fonction : facile, lent, pas propre),
    - soit on augmente sévèrement la grammaire d'expression pour distinguer toutes les combinaisons pertinentes entier/ réel, entier/entier, etc.. (très grosse augmentation, il faut faire des règles *expr_entiere expr_reelle terme_entier, terme_reel,* et toutes les combinaisons avec tous les opérateurs en distinguant par exemple *plus_entier_entier, plus_entier_reel* etc etc…),
    - soit on gère dans l'analyseur une pile "de calcul" des types manipulés avec la gestion d'erreurs afférente : quand on dit à l'exécuteur de pousser une valeur que l'on *caste* toujours sur entier, on pousse dans *l'analyseur* le type de cette valeur, et on maintient dans l'analyseur la pile **des types** des valeurs calculées. Quand on dit à l'exécuteur de faire une addition, on sélectionne dans l'analyseur un des quatre opérateurs possibles *plus_entier_entier, plus_entier_reel*, etc., on le passe à l'exécuteur qui le fera après le *cast* des opérandes.
    - Pour les chaînes de caractères, la solution de facilité mais inefficace est de tout réduire à des caractères uniques et un opérateur de concaténation. L'exécuteur poussera dans la pile **l'adresse** d'une chaîne comprenant la valeur courante de la chaîne éventuellement réduite à un seul caractère. La solution propre est de pousser dans la pile les caractères, leur nombre et enfin –donc en sommet de pile- un indicateur (marquant le fait que c'est une chaîne) Les opérateurs devront bien sûr tenir compte de cette convention. On peut aussi éviter le nombre en mettant la convention C (zéro terminal).
    - Attention à ce que les erreurs deviennent potentiellement nombreuses (par exemple addition d'une chaîne de caractères avec un réel)
  - Édition de liens : il s'agit là de faire plusieurs « modules » s'appelant l'un l'autre, et un programme supplémentaire capable de résoudre les adresses manquantes après la compilation mais avant l'exécution. Pour cela il faudra ajouter dans la syntaxe la notion de déclaration de sous-programme, et une convention permettant de mettre dans le fichier intermédiaire les adresses des cases à compléter avec les noms symboliques qui les baptisent.

9 Table des symboles [TP]

Tous les langages de programmation font l'association entre des noms (chaînes de caractères) et des objets du langage : variables, sous-programmes, etc.

La question est 1/ de gérer le stockage des noms et 2/ d'associer efficacement un nom et une valeur. Le module « table des symboles » a très peu de primitives : une primitive d'association qui « colle » une valeur à un nom ; deux primitives d'interrogation, pour savoir si le nom existe dans la table (par exemple pour voir si une variable a été déclarée) ; et une primitive qui rend la valeur associée. Plus une primitive pour initialiser.

```
void init() ;
void associer (char * nom, un_type valeur) ;
int existe(char * nom) ;
un_type valeur(char * nom) ;
```

Le problème à gérer, c'est que l'espace des noms est infini (on ne peut pas indicer un tableau par un nom), alors même que le nombre de noms réellement utilisés est fini, voire petit. Il faut donc des structures de données permettant de rechercher par association.

9.1 La liste chaînée

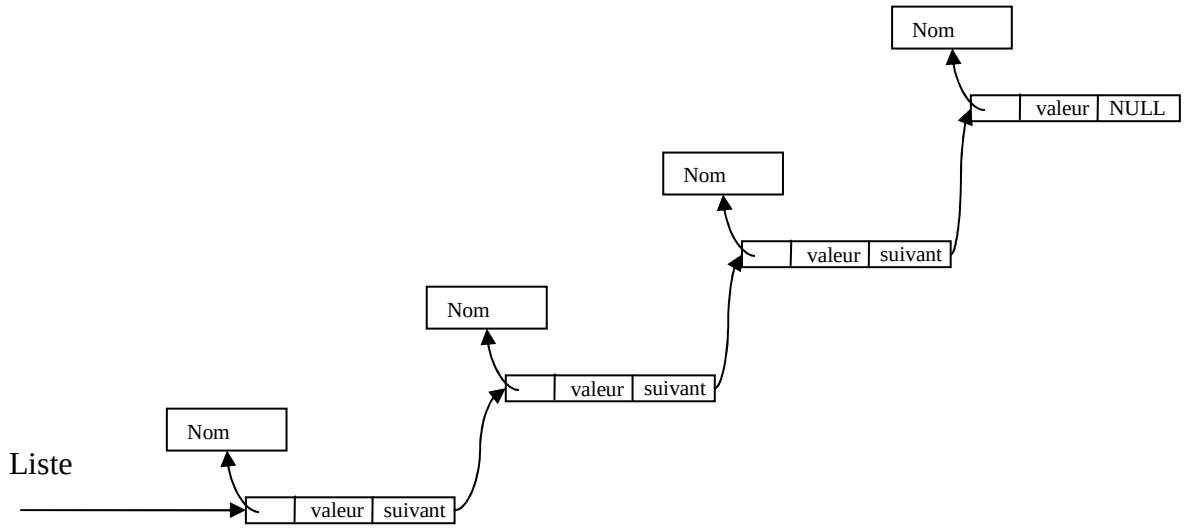


Figure 48 Table des symboles par liste chaînée

La solution par liste chaînée est la plus simple, malheureusement elle ne marche pas. Ou plutôt, elle marche de façon si inefficace que c'est une solution qui n'est bonne que pour faire des essais.

9.2 L'arbre

Faire un arbre peut sembler séduisant. On procède par dichotomie, et chaque cellule feuille de l'arbre pointe vers la valeur intéressante. Les recherches se font avec des algorithmes efficaces, de la même façon qu'on cherche un mot dans un dictionnaire.

Hélas cette solution a quelques inconvénients qui la rendent difficilement praticable : les arbres doivent être équilibrés pour que la procédure de recherche soit efficace. Or il est très fréquent que dans les « vrais » programmes, les noms ne soient pas distribués régulièrement, par exemple toutes les variables peuvent être préfixées de la même façon. Cela conduit à des arbres tellement bancals que, ultimement, ils peuvent se ramener très peu de branches ou à des troncs très longs, ce qui leur donne les inconvénients de la liste simplement chaînée. Évidemment, la théorie des arbres donne les outils pour rééquilibrer l'arbre au fur et à mesure, mais cela coûte du temps et des ressources. Et il se trouve que la solution suivante a presque tous les avantages de l'arbre sans en avoir les inconvénients.

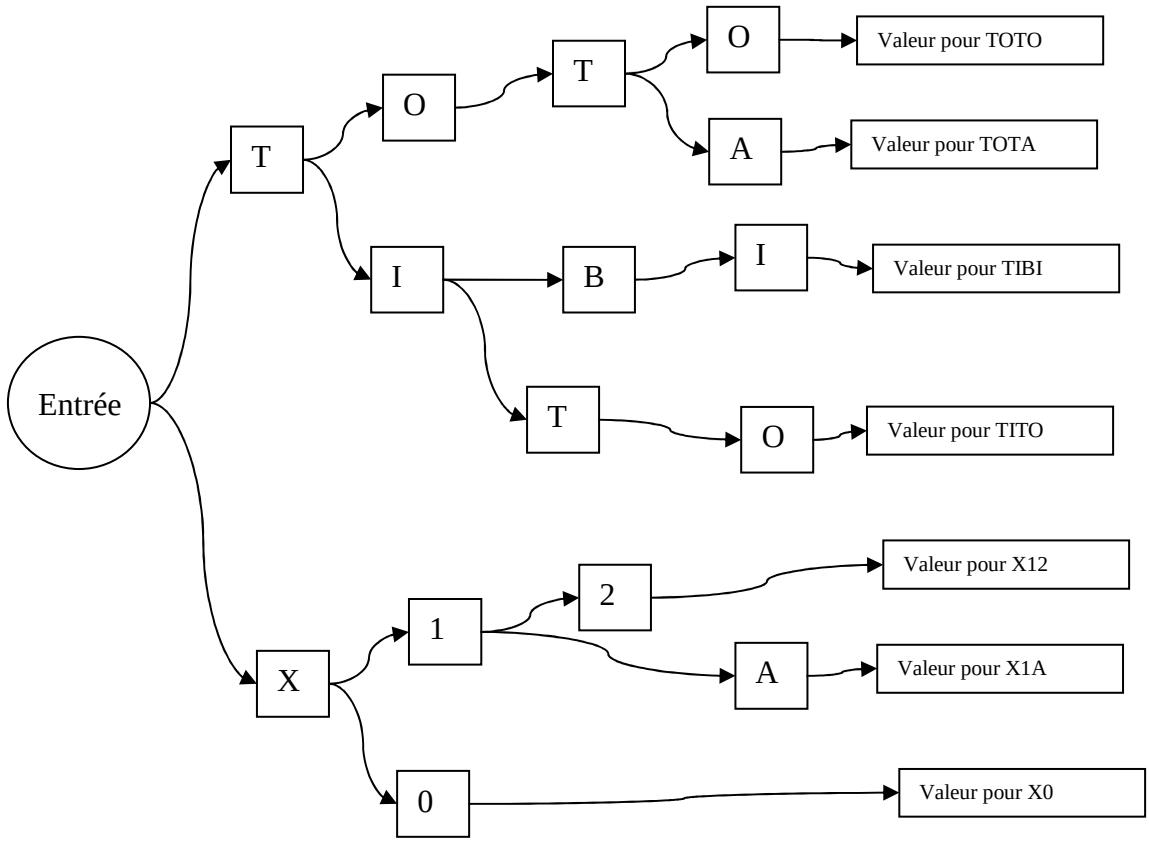


Figure 49 Table des symboles par arbre N-aire

9.3 La table de listes

L'idée est de faire N listes simplement chaînées, ce qui va diviser le temps de parcours par N. Ce nombre peut être grand : 1000 ou 10000 n'est pas rare, cela ne coûte « que » un ou dix k mots... Le tout est de trouver le moyen d'associer un entier compris entre 0 et N qui soit calculable rapidement et évidemment toujours le même, à partir du nom. Cela s'appelle fonction de hachage (*hash function*). Si N est petit, une solution simple est d'additionner les

codes ASCI de tous les caractères. Cela conduit à une dispersion assez mauvaise, par exemple les permutations de lettres donnent le même résultat. Une assez bonne solution consiste à faire, pour chaque lettre du nom, une addition avec un nombre premier, une multiplication par un autre et le modulo par N. La littérature est pleine de « bonnes » fonctions de hachage, et aussi hélas de situations perverses qui mettent en défaut les meilleures fonctions. Insistons sur le fait que la fonction doit avoir deux qualités: la dispersion statistique homogène, et la rapidité. Cela élimine (en C) les fonctions de hachage qui ont besoin de l'accès aux bits des caractères pour faire des permutations circulaires. Par contre on peut appeler des fonctions préexistantes qui sont probablement codées en assembleur, si elles sont disponibles (SHA-1 ou MD5).

Un exemple de fonction probablement suffisante pour une table de symboles et une centaine de listes, utilisant deux nombres premiers autour de 100 :

```
int hash(char * nom) {
  int result = 0;
  while (*nom++) result =  (( result + *nom + 113 ) *  89) % 100;
  return result;
}
```

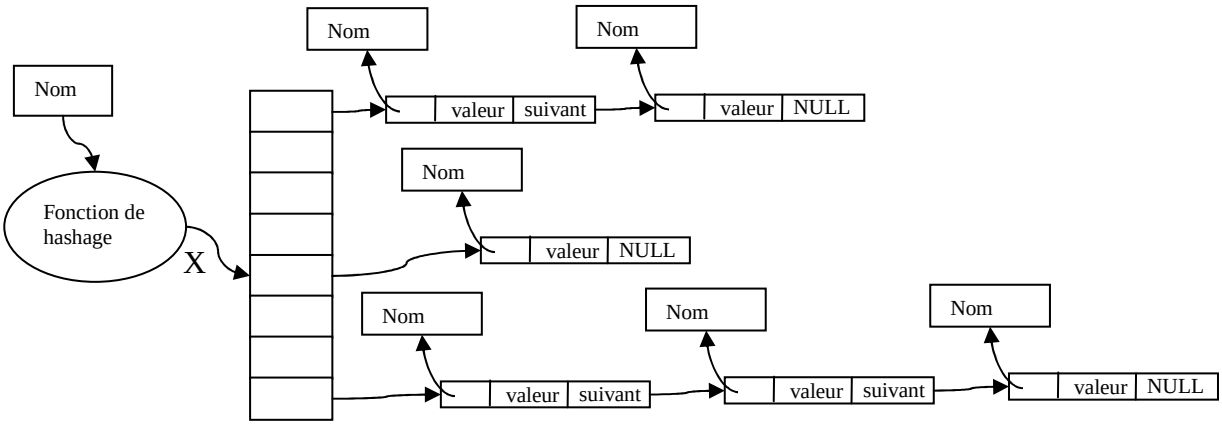


Figure 50 Table des symboles par fonction de hachage et listes

Au bilan, avec N judicieusement choisi, on peut arriver à une situation où la majorité des listes sont de longueur 1, ce qui ramène le temps moyen de la recherche au calcul du hachage, un indexage de tableau et une comparaison de chaînes.

10 Génération de code pour le processeur développé en VHDL [TP]

Pour rester dans le cadre du temps imparti, on se limitera au code nécessaire au petit filtre numérique déjà utilisé en exemple :

- On fait une entrée
- On additionne la valeur avec un registre interne
- On divise ce registre par deux
- On sort la valeur de ce registre
- On revient au début

Ceci est un filtre passe-bas si les valeurs entrées et sorties correspondent à des niveaux de signal. Le programme correspondant sera (avec variantes selon votre syntaxe exacte):

```
Y=0 ;
loop
  X= ?;
  Y=(Y+X)/2;
  Y ;
endloop ;
```

Si l'instruction « loop » n'est pas encore implémentée, il est inutile de perdre du temps là-dessus, on fera d'autorité le code d'un JUMP approprié à la fin de la génération.

Réalisation :

On va modifier l'exécuteur, de façon qu'au lieu de faire les opérations demandées, il produise un fichier binaire à destination du processeur. On pourrait aussi créer directement ce code depuis le fichier YACC, mais on perdrait le travail d'analyse déjà fait. Par exemple, le code « entrer » qui auparavant débouchait sur un *scanf* va maintenant écrire dans un fichier *rom_file.txt* : « 0000000100000110 » ce qui signifie : faire une entrée, et déposer le résultat dans R6 (6 est arbitraire).

Annex

La seule vraie difficulté est la gestion de la pile : en effet elle n'arrive plus « gratuitement » par la grâce d'une macro, il faut la gérer explicitement. Les macros « POUSSER TIRER etc. » vont maintenant écrire du code machine. On utilisera R1 comme index de pile. Les variables locales « x, y, etc. » deviendront des cases mémoire ou des registres de la banque de registres. Il faudra aussi allouer en mémoire les 128 cases de variables possibles.

La division (qui n'existe pas dans notre femto-processeur) sera « arrondie » à la puissance de deux la plus proche et traduite en décalages à droite. Dans un premier temps on peut même n'implémenter que la division par 2.

```
0000000100100101  x00 Load Imm R5 with x0100
0001000000000000  x01 x0100 (arbitrary value)
0000000100000110  x02 inp R6
1000000101101101  x03 add @R5 R6 -> @R5
1000101100001101  x04 div 2 : shr @R5
0000000100011101  x05 outp @R5
0000000100100000  x06 ldim R0  (goto 02)
0000000000000010  x07 valeur 02
```

Si votre processeur en VHDL est vacillant, vous pouvez utiliser ou vous inspirer de http://rouillard.org/femto-controleur.vhd [4]qui va avec http://rouillard.org/rom_file1.txt qui contient, grosso modo, ce que doit produire votre générateur de code (il faudra sans doute modifier le chemin absolu du fichier ROM dans le VHDL.) Le code opération du contrôleur se trouve dans http://rouillard.org/VHDL-cours-seq.mht

[4] Dans ce fichier VHDL qui contient plein d'entités et d'architectures, celle vue en cours se trouvera dans votre bibliothèque sous le nom de MCU(STRUCT1) et le test associé est TEST_ALL1(STRUCT).

11 Annexes

Voici les grammaires *lex* et *yacc* permettant d'analyser ANSI-C. Attention, elles ne comprennent pas les instructions de préprocesseur (inclusions, macros, etc.) car le préprocesseur C est un autre langage!

11.1 Lex pour C

```
D                       [0-9]
L                       [a-zA-Z_]
H                       [a-fA-F0-9]
E                       [Ee][+-]?{D}+
FS                      (f|F|l|L)
IS                      (u|U|l|L)*

%{
#include <stdio.h>
#include "y.tab.h"
void count();
%}

%%
"/*"                    { comment(); }
"auto"                  { count(); return(AUTO); }
"break"                 { count(); return(BREAK); }
"case"                  { count(); return(CASE); }
"char"                  { count(); return(CHAR); }
"const"                 { count(); return(CONST); }
"continue"              { count(); return(CONTINUE); }
"default"               { count(); return(DEFAULT); }
"do"                    { count(); return(DO); }
"double"                { count(); return(DOUBLE); }
"else"                  { count(); return(ELSE); }
"enum"                  { count(); return(ENUM); }
"extern"                { count(); return(EXTERN); }
"float"                 { count(); return(FLOAT); }
"for"                   { count(); return(FOR); }
"goto"                  { count(); return(GOTO); }
"if"                    { count(); return(IF); }
"int"                   { count(); return(INT); }
"long"                  { count(); return(LONG); }
"register"              { count(); return(REGISTER); }
"return"                { count(); return(RETURN); }
"short"                 { count(); return(SHORT); }
"signed"                { count(); return(SIGNED); }
"sizeof"                { count(); return(SIZEOF); }
"static"                { count(); return(STATIC); }
"struct"                { count(); return(STRUCT); }
"switch"                { count(); return(SWITCH); }
"typedef"               { count(); return(TYPEDEF); }
"union"                 { count(); return(UNION); }
"unsigned"              { count(); return(UNSIGNED); }
"void"                  { count(); return(VOID); }
"volatile"              { count(); return(VOLATILE); }
"while"                 { count(); return(WHILE); }
{L}({L}|{D})*           { count();return(IDENTIFIER);}
0[xX]{H}+{IS}?          { count(); return(CONSTANT); }
0{D}+{IS}?              { count(); return(CONSTANT); }
{D}+{IS}?               { count(); return(CONSTANT); }
L?'(\\.|[^\\'])+'       { count(); return(CONSTANT); }
{D}+{E}{FS}?            { count(); return(CONSTANT); }
{D}*"."{D}+({E})?{FS}?{ count(); return(CONSTANT); }
{D}+"."{D}*({E})?{FS}?{ count(); return(CONSTANT); }
L?\"(\\.|[^\\"])*\"     { count(); return(STRING_LITERAL); }
"..."                   { count(); return(ELLIPSIS); }
">>="                   { count(); return(RIGHT_ASSIGN); }
"<<="                   { count(); return(LEFT_ASSIGN); }
"+="                    { count(); return(ADD_ASSIGN); }
"-="                    { count(); return(SUB_ASSIGN); }
"*="                    { count(); return(MUL_ASSIGN); }
"/="                    { count(); return(DIV_ASSIGN); }
"%="                    { count(); return(MOD_ASSIGN); }
```

```
"&="                    { count(); return(AND_ASSIGN); }
"^="                    { count(); return(XOR_ASSIGN); }
"|="                    { count(); return(OR_ASSIGN); }
">>"                    { count(); return(RIGHT_OP); }
"<<"                    { count(); return(LEFT_OP); }
"++"                    { count(); return(INC_OP); }
"--"                    { count(); return(DEC_OP); }
"->"                    { count(); return(PTR_OP); }
"&&"                    { count(); return(AND_OP); }
"||"                    { count(); return(OR_OP); }
"<="                    { count(); return(LE_OP); }
">="                    { count(); return(GE_OP); }
"=="                    { count(); return(EQ_OP); }
"!="                    { count(); return(NE_OP); }
";"                     { count(); return(';'); }
("{"|"<%")              { count(); return('{'); }
("}"|"%>")              { count(); return('}'); }
","                     { count(); return(','); }
":"                     { count(); return(':'); }
"="                     { count(); return('='); }
"("                     { count(); return('('); }
")"                     { count(); return(')'); }
("["|"<:")              { count(); return('['); }
("]"|":>")              { count(); return(']'); }
"."                     { count(); return('.'); }
"&"                     { count(); return('&'); }
"!"                     { count(); return('!'); }
"~"                     { count(); return('~'); }
"-"                     { count(); return('-'); }
"+"                     { count(); return('+'); }
"*"                     { count(); return('*'); }
"/"                     { count(); return('/'); }
"%"                     { count(); return('%'); }
"<"                     { count(); return('<'); }
">"                     { count(); return('>'); }
"^"                     { count(); return('^'); }
"|"                     { count(); return('|'); }
"?"                     { count(); return('?'); }
[ \t\v\n\f]             { count(); }
.                       { /* ignore bad characters */ }
%%

yywrap()
{
        return(1);
}

comment() /* consomme les commentaires */
{
        char c, c1;
loop:
        while ((c = input()) != '*' && c != 0)
                putchar(c);
        if ((c1 = input()) != '/' && c != 0)
        {
                unput(c1);
                goto loop;
        }
        if (c != 0)
                putchar(c1);
}

int column = 0;
void count()  /* compte les colonnes pour donner la position exacte de l'erreur */
{
        int i;
        for (i = 0; yytext[i] != '\0'; i++)
                if (yytext[i] == '\n')
                        column = 0;
                else if (yytext[i] == '\t')
                        column += 8 - (column % 8);
                else
                        column++;
        ECHO;
}
```

11.2 Yacc pour C

Cette grammaire produira un conflit *shif/reduce* pour la branche *else* optionnelle du l'instruction *if* (voir à ce sujet page 34, l'item 3 de la passe 4). La résolution du conflit est heureusement par défaut celle qui est espérée pour C.

```
%token IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN TYPE_NAME

%token TYPEDEF EXTERN STATIC AUTO REGISTER
%token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST VOLATILE VOID
%token STRUCT UNION ENUM ELLIPSIS

%token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN

%start translation_unit
%%

primary_expression
        : IDENTIFIER
        | CONSTANT
        | STRING_LITERAL
        | '(' expression ')'
        ;

postfix_expression
        : primary_expression
        | postfix_expression '[' expression ']'
        | postfix_expression '(' ')'
        | postfix_expression '(' argument_expression_list ')'
        | postfix_expression '.' IDENTIFIER
        | postfix_expression PTR_OP IDENTIFIER
        | postfix_expression INC_OP
        | postfix_expression DEC_OP
        ;

argument_expression_list
        : assignment_expression
        | argument_expression_list ',' assignment_expression
        ;

unary_expression
        : postfix_expression
        | INC_OP unary_expression
        | DEC_OP unary_expression
        | unary_operator cast_expression
        | SIZEOF unary_expression
        | SIZEOF '(' type_name ')'
        ;

unary_operator
        : '&'
        | '*'
        | '+'
        | '-'
        | '~'
        | '!'
        ;

cast_expression
        : unary_expression
        | '(' type_name ')' cast_expression
        ;

multiplicative_expression
        : cast_expression
        | multiplicative_expression '*' cast_expression
        | multiplicative_expression '/' cast_expression
        | multiplicative_expression '%' cast_expression
        ;

additive_expression
```

```
        : multiplicative_expression
        | additive_expression '+' multiplicative_expression
        | additive_expression '-' multiplicative_expression
        ;

shift_expression
        : additive_expression
        | shift_expression LEFT_OP additive_expression
        | shift_expression RIGHT_OP additive_expression
        ;

relational_expression
        : shift_expression
        | relational_expression '<' shift_expression
        | relational_expression '>' shift_expression
        | relational_expression LE_OP shift_expression
        | relational_expression GE_OP shift_expression
        ;

equality_expression
        : relational_expression
        | equality_expression EQ_OP relational_expression
        | equality_expression NE_OP relational_expression
        ;

and_expression
        : equality_expression
        | and_expression '&' equality_expression
        ;

exclusive_or_expression
        : and_expression
        | exclusive_or_expression '^' and_expression
        ;

inclusive_or_expression
        : exclusive_or_expression
        | inclusive_or_expression '|' exclusive_or_expression
        ;

logical_and_expression
        : inclusive_or_expression
        | logical_and_expression AND_OP inclusive_or_expression
        ;

logical_or_expression
        : logical_and_expression
        | logical_or_expression OR_OP logical_and_expression
        ;

conditional_expression
        : logical_or_expression
        | logical_or_expression '?' expression ':' conditional_expression
        ;

assignment_expression
        : conditional_expression
        | unary_expression assignment_operator assignment_expression
        ;

assignment_operator
        : '='
        | MUL_ASSIGN
        | DIV_ASSIGN
        | MOD_ASSIGN
        | ADD_ASSIGN
        | SUB_ASSIGN
        | LEFT_ASSIGN
        | RIGHT_ASSIGN
        | AND_ASSIGN
        | XOR_ASSIGN
        | OR_ASSIGN
        ;

expression
        : assignment_expression
        | expression ',' assignment_expression
```

```
        ;

constant_expression
        : conditional_expression
        ;

declaration
        : declaration_specifiers ';'
        | declaration_specifiers init_declarator_list ';'
        ;

declaration_specifiers
        : storage_class_specifier
        | storage_class_specifier declaration_specifiers
        | type_specifier
        | type_specifier declaration_specifiers
        | type_qualifier
        | type_qualifier declaration_specifiers
        ;

init_declarator_list
        : init_declarator
        | init_declarator_list ',' init_declarator
        ;

init_declarator
        : declarator
        | declarator '=' initializer
        ;

storage_class_specifier
        : TYPEDEF
        | EXTERN
        | STATIC
        | AUTO
        | REGISTER
        ;

type_specifier
        : VOID
        | CHAR
        | SHORT
        | INT
        | LONG
        | FLOAT
        | DOUBLE
        | SIGNED
        | UNSIGNED
        | struct_or_union_specifier
        | enum_specifier
        | TYPE_NAME
        ;

struct_or_union_specifier
        : struct_or_union IDENTIFIER '{' struct_declaration_list '}'
        | struct_or_union '{' struct_declaration_list '}'
        | struct_or_union IDENTIFIER
        ;

struct_or_union
        : STRUCT
        | UNION
        ;

struct_declaration_list
        : struct_declaration
        | struct_declaration_list struct_declaration
        ;

struct_declaration
        : specifier_qualifier_list struct_declarator_list ';'
        ;

specifier_qualifier_list
        : type_specifier specifier_qualifier_list
        | type_specifier
        | type_qualifier specifier_qualifier_list
```

```
        | type_qualifier
        ;

struct_declarator_list
        : struct_declarator
        | struct_declarator_list ',' struct_declarator
        ;

struct_declarator
        : declarator
        | ':' constant_expression
        | declarator ':' constant_expression
        ;

enum_specifier
        : ENUM '{' enumerator_list '}'
        | ENUM IDENTIFIER '{' enumerator_list '}'
        | ENUM IDENTIFIER
        ;

enumerator_list
        : enumerator
        | enumerator_list ',' enumerator
        ;

enumerator
        : IDENTIFIER
        | IDENTIFIER '=' constant_expression
        ;

type_qualifier
        : CONST
        | VOLATILE
        ;

declarator
        : pointer direct_declarator
        | direct_declarator
        ;

direct_declarator
        : IDENTIFIER
        | '(' declarator ')'
        | direct_declarator '[' constant_expression ']'
        | direct_declarator '[' ']'
        | direct_declarator '(' parameter_type_list ')'
        | direct_declarator '(' identifier_list ')'
        | direct_declarator '(' ')'
        ;

pointer
        : '*'
        | '*' type_qualifier_list
        | '*' pointer
        | '*' type_qualifier_list pointer
        ;

type_qualifier_list
        : type_qualifier
        | type_qualifier_list type_qualifier
        ;


parameter_type_list
        : parameter_list
        | parameter_list ',' ELLIPSIS
        ;

parameter_list
        : parameter_declaration
        | parameter_list ',' parameter_declaration
        ;

parameter_declaration
        : declaration_specifiers declarator
        | declaration_specifiers abstract_declarator
        | declaration_specifiers
```

```
        ;

identifier_list
        : IDENTIFIER
        | identifier_list ',' IDENTIFIER
        ;

type_name
        : specifier_qualifier_list
        | specifier_qualifier_list abstract_declarator
        ;

abstract_declarator
        : pointer
        | direct_abstract_declarator
        | pointer direct_abstract_declarator
        ;

direct_abstract_declarator
        : '(' abstract_declarator ')'
        | '[' ']'
        | '[' constant_expression ']'
        | direct_abstract_declarator '[' ']'
        | direct_abstract_declarator '[' constant_expression ']'
        | '(' ')'
        | '(' parameter_type_list ')'
        | direct_abstract_declarator '(' ')'
        | direct_abstract_declarator '(' parameter_type_list ')'
        ;

initializer
        : assignment_expression
        | '{' initializer_list '}'
        | '{' initializer_list ',' '}'
        ;

initializer_list
        : initializer
        | initializer_list ',' initializer
        ;

statement
        : labeled_statement
        | compound_statement
        | expression_statement
        | selection_statement
        | iteration_statement
        | jump_statement
        ;

labeled_statement
        : IDENTIFIER ':' statement
        | CASE constant_expression ':' statement
        | DEFAULT ':' statement
        ;

compound_statement
        : '{' '}'
        | '{' statement_list '}'
        | '{' declaration_list '}'
        | '{' declaration_list statement_list '}'
        ;

declaration_list
        : declaration
        | declaration_list declaration
        ;

statement_list
        : statement
        | statement_list statement
        ;

expression_statement
        : ';'
        | expression ';'
        ;
```

```
selection_statement
        : IF '(' expression ')' statement
        | IF '(' expression ')' statement ELSE statement
        | SWITCH '(' expression ')' statement
        ;

iteration_statement
        : WHILE '(' expression ')' statement
        | DO statement WHILE '(' expression ')' ';'
        | FOR '(' expression_statement expression_statement ')' statement
        | FOR '(' expression_statement expression_statement expression ')' statement
        ;

jump_statement
        : GOTO IDENTIFIER ';'
        | CONTINUE ';'
        | BREAK ';'
        | RETURN ';'
        | RETURN expression ';'
        ;

translation_unit
        : external_declaration
        | translation_unit external_declaration
        ;

external_declaration
        : function_definition
        | declaration
        ;

function_definition
        : declaration_specifiers declarator declaration_list compound_statement
        | declaration_specifiers declarator compound_statement
        | declarator declaration_list compound_statement
        | declarator compound_statement
        ;

%%
#include <stdio.h>

extern char yytext[];
extern int column;

yyerror(s)
char *s;
{
        fflush(stdout);
        printf("\n%*s\n%*s\n", column, "^", column, s);
}
```

12 Index

T

V

Y

13 Table des figures