SYNTEX
Générateur de Compilateur

Release 2.0

(C) J.F. Le Téno
3614 TEASER - BAL: Big John

"S'il te plait, écris-moi un compilateur"

le Petit Programmeur.

R E M A R Q U E P R E L I M I N A I R E

Ce programme n'est PAS dans le 'domaine public', et aussi génial ou aussi nul qu'il puisse vous paraître, il représente un investissement personnel certain en temps et en sueur. Je n'exige en retour le versement d'aucune licence, mais j'apprécierais grandement un signe de vie - de solidarité, de reconnaissance ?!... - de votre part, si toutefois vous décidez d'utiliser ce logiciel. Sentez-vous donc libre de m'envoyer ce qui vous semble bon (chocolat, coup de fil, matériel inutilisé, carte postale, quelques sous, ...); ceux d'entre vous qui ont déjà programmé comprendront ...
Je me ferais alors un plaisir de mettre les sources à votre disposition ainsi que la version complète, et je serais également heureux de recevoir toute critique, tout conseil ou suggestion sur Syntex 2.0.

L'auteur.

Contenu du Fichier

A l'attention des novices

Ce fichier est à la fois la documentation de Syntex et une introduction à la compilation.

Le chapitre "Exemple" est la meilleur introduction pour une personne totalement néophyte; il montre clairement ce que FAIT Syntex.

Le chapitre "Ecrire une grammaire LL(1)" présente en détail ce qu'est un compilateur, la technique de compilation descendante, la notion d'arbre syntaxique, de production, de terminal, et enfin de grammaire LL(1). les paragraphes e/ et f/ peuvent être sautés en première lecture.

La lecture de ces deux chapitres doit permettre - si ce n'est pas le cas, il faut me le dire - de comprendre précisément ce que fait Syntex, et de l'exploiter dans ses propres projets.
Pour celui - ou celle - qui s'intéresse à la compilation, je recommande très vivement le bouquin de Aho, Sethi, et Ullman "COMPILATEURS Principes, Techniques et Outils", chez InterEditions.

Le chapitre "Présentation" est volontairement "technique"; si vous ne le comprenez pas, lisez donc les deux chapitres mentionnés précédemment.

Les chapitres "Syntaxe BNF" et "Interface utilisateur" présentent respectivement le format des fichiers .Grm, et le rôle des différentes options accessibles en cours de programme.

Présentation

Syntex est un Générateur de Compilateur basé sur un article d'Alix Alix paru dans les numéros 29, 31 et 35 de Pascalissime, et dont il reprend en grande partie les "moteurs".

A partir d'un fichier texte contenant la grammaire du langage pour lequel on veut écrire un compilateur, il génère le corps de l'analyseur syntaxique.

La **grammaire** source doit être écrite avec la **notation BNF** (Backus-Naur Form) dans un fichier texte. Elle doit être de la classe LL(1) pour être compilable par déscente récursive sans rebroussement, avec un seul symbole d'avance - comme celle du Pascal, par exemple -.

Syntex implante alors en (Turbo) **Pascal** le corps d'un analyseur syntaxique prédictif récursif.

!! Attention !! Dans sa version actuelle,

_ Syntex ne vérifie *PAS* que la grammaire est non contextuelle, non ambigüe et LL(1). De plus, il n'effectue aucune correction, ni optimisation du texte source (traitement des récursions à gauche, factorisation à gauche, élimination des productions cycliques ou vides)

_ Seul le squelette de l'analyseur lexical est produit. Il est donc nécessaire de l'écrire.

_ Seul le squelette de la procédure de traitement des erreurs syntaxiques est produit. Il est donc

nécessaire de l'écrire - voir le chapitre Considérations diverses.

_ L'analyse sémantique est à insérer dans le corps de l'analyseur syntaxique.

Exemple

On lance Syntex, et on charge le fichier Test.GRM.

```
'='=Egal.
'+'=Plus.
'*'=Fois.
'('=POuverte.
')'=PFermee.
';'=PointVirgule.
DEBUT=nom'='expression';'.
nom=NOM.
expression=terme{'+'terme}.
terme=facteur{'*'facteur}.
facteur=nom|'('expression')'.
<Eof>
```

On sélectionne l'option Generate du menu du même nom,

Syntex génère alors le code suivant, qu'il place dans le fichier Test.PAS:

```
Program test;

type TString80 = String[80];
   TSymbole  = (SymbInconnu,
                SymbEgal,
                SymbPlus,
                SymbFois,
                SymbPOuverte,
                SymbPFermee,
                SymbPointVirgule,
                SymbNOM,
                SymbFin);
var Symbole : TSymbole;

procedure LisSymbole;
begin
end;

procedure AfficheErreur(PErreur : TString80);
begin
end;

procedure Traitenom;
 Begin {Traitenom}
   if Symbole = SymbNOM then
   begin
     {LisSymbole après NOM}
```

```
    LisSymbole;
  end {TraiteNOM}
  else AfficheErreur('NOM attendu');
end; {Traitenom}

procedure Traiteexpression;

 procedure Traiteterme;

  procedure Traitefacteur;
   Begin {Traitefacteur}
    if Symbole = SymbNOM then
    begin
     Traitenom;
    end {NOM}
    else
    if Symbole = SymbPOuverte then
    begin
     {LisSymbole après (}
     LisSymbole;
     Traiteexpression;
     if Symbole = SymbPFermee then
     begin
      {LisSymbole après )}
      LisSymbole;
     end {Traite)}
     else AfficheErreur(') attendu');
    end {POuverte}
    else AfficheErreur('NOM ou POuverte attendu');
   end; {Traitefacteur}

  Begin {Traiteterme}
   Traitefacteur;
   while Symbole=SymbPlus do
   begin
    {LisSymbole après +}
    LisSymbole;
    Traitefacteur;
   end;
  end; {Traiteterme}

 Begin {Traiteexpression}
  Traiteterme;
  while Symbole=SymbFois do
  begin
   {LisSymbole après *}
   LisSymbole;
   Traiteterme;
  end;
 end; {Traiteexpression}

Begin {TraiteDEBUT}
 LisSymbole;
 Traitenom;
 if Symbole = SymbEgal then
 begin
```

```
  {LisSymbole après =}
  LisSymbole;
  Traiteexpression;
  if Symbole = SymbPointVirgule then
  begin
    {LisSymbole après ;}
    LisSymbole;
  end {Traite;}
  else AfficheErreur('; attendu');
 end {Traite=}
 else AfficheErreur('= attendu');
End.
```

Syntaxe BNF - Fichiers .Grm

Les fichiers contenant les descriptions grammaticales ont pour extension .Grm. La syntaxe utilisée pour décrire la grammaire est dérivée de la notation de Backus-Naur (BNF).

Le signe	est utilisé pour
=	séparer la définition de son texte.
|	séparer les alternatives.
[ ]	encadrer les parties de définition qui peuvent être présentes zéro ou une fois.
{ }	encadrer les parties de définition qui peuvent être présentes zéro ou plusieurs fois.
.	termine une définition

(tableau des méta-symboles BNF)

La priorité est laissée aux métasymboles [,], {, et }, sur le symbole |.

Afin de reconnaître les terminaux dans l'arbre syntaxique, Syntex impose les deux règles suivantes:

__ La liste des symboles de ponctuation, ie de tous les terminaux qui ne sont pas alphanumériques, doit être donnée en début du fichier .Grm, en plaçant les symboles entre guillemets.

__ Les terminaux qui ne sont pas des symboles de ponctuation doivent débuter par une majuscule. Ce qui interdit également l'usage des majuscules pour la désignation des noeuds non-terminaux.

L'ordre dans lequel les règles de syntaxes sont présentées dans le fichier est indifférent; Dans le fichier TEST.GRM elles sont présentées par ordre de dérivation; elles peuvent tout aussi bien

être présentées dans le désordre.

A noter qu'une production peut s'étendre sur plusieurs lignes, mais qu'aucun symbole ne peut excéder 80 catactères de longueur.

Syntex accepte les commentaires dans les fichiers .Grm, à condition de les placer entre '@'.

Exemple:

```
@ Grammaire de la syntaxe Pascal des réels @
chiffre= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'.
entier = chiffre{chiffre}.
signe  = '+'|'-'.
entier_signe = [signe] entier.
reel = entier['.'entier]['E'entier_signe].
```

Voir également les fichiers GRM fournis en exemple.

Egalement à titre d'illustration, voici la grammaire des fichiers .Grm :
(voir le chapitre /5/f)

```
@ Description de la grammaire des fichiers .Grm @
@ Syntex - Release 2.0                           @
'='= Egal.
'{'= Acc_ouverte.
'}'= Acc_fermée.
'['= Crochet_ouvert.
']'= Crochet_farmé.
'|'= Alternatif.
'%'= Fin_production.
Fichier = ponctuation { ponctuation } production { production }.
 ponctuation = Caractère_ponctuation '=' Nom '%'.
 production = Nom '=' expression '%'.
   expression = terme { '|' terme }.
     terme = facteur { facteur }.
       facteur = '{' expression '}' |
                 '[' expression ']' |
                  Terminal          |
                  Non_terminal.
```

Remarque tout à fait personnelle: quel plaisir de voir que l'on peut écrire automatiquement une partie importante de ce qui a été écrit à la main!

(Note personnelle (encore!): J'y suis particulièrement sensible pour m'être 'coltiné' à la main l'écriture de l'interpréteur de WIZZ, un petit programme de tracé de fonctions mathématiques - en tenant compte du domaine de définition - et de calcul scientifique. C'était un très bon exercice pédagogique pour moi, mais comme toujours dans ce genre d'exercices, les vertues disparaissent rapidement dès qu'on l'a fait une fois. FdNote)

Interface Utilisateur - release 2.0

Elle s'est considérablement améliorée - voir la Version History - grâce à l'utilisation de Turbo Vision.

L'utilisation est maintenant "intuitive" - comme on dit, histoire de rire un bon coup -, et l'aide est accessible en ligne par la touche F1.

Menu =

Etiquette d'identification du programme.

Menu FICHIER

L'objet éditeur utilisé est celui livré avec la version 2.0 de Turbo Vision; c'est donc un éditeur classique, disposant d'un presse papier, et configuré en auto-indentation pour faciliter la lisibilité des descriptions de grammaires.

La seule particularité vient de ce que lorsqu'une fenêtre d'édition d'un fichier .Grm est fermée, les fenêtre de débogage associées éventuellement ouvertes sont fermées également - Syntax Tree, KeyWords List, Ponctuation List -.

Menu EDIT

Ce menu permet l'édition du texte source par le biais d'un presse-papier.

Menu GENERATE

_ Generate génère le code (Turbo) Pascal à partir du source descriptif de la grammaire contenu dans la fenêtre courante; si ce n'est pas une fenêtre d'édition, une erreur est générée.

_ Options permet de choisir le type de présentation du code de sortie: Unit (TP4+) ou Program (TP3+).

_ Make compile le code produit. Pour le moment, n'étant pas sûr que c'est une nécessité, cette option est inactive.

Menu DEBUG

_ Ponctuation ouvre une fenêtre contenant la liste des ponctuation déclarées dans le source descriptif de la grammaire.

_ Key Words ouvre une fenêtre contenant la liste des mots-clef déclarés dans le source descriptif de la grammaire.

_ Syntax Tree ouvre une fenêtre contenant l'arborescence des productions déclarées dans le source descriptif de la grammaire.

Menu OPTIONS

_ Video passe du mode 25 lignes au mode 43 lignes, et réciproquement.

_ Save Desktop sauve l'environnement de travail sur disque. Cette option n'est pas implémenté sur la version non enregistrée.

_ Load Desktop charge l'environnement de travail préalablement sauvegardé sur disque. Elle n'est pas implémenté sur la version non enregistrée.

Menu WINDOWS

Options générales - et classiques - de gestion des fenêtres. Permet également de fermer en une fois toutes les fenêtre ouvertes.

Comment écrire une grammaire LL(1)
----

a/ Survol de la structure d'un compilateur

Un compilateur est un programme qui, **à partir d'un texte source écrit dans un langage donné, produit une traduction**. Il doit donc lire le texte source, vérifier qu'il est correct, et le traduire. Ces fonctions sont réparties entre différents modules.

La lecture du source est effectuée par l'**analyseur lexica**l, dont le rôle est de découper le source en unité lexicales (ou symboles) correspondant au langage utilisé - en mots -.

Vient ensuite l'**analyseur syntaxique**, qui a pour rôle de vérifier que les suites d'unités lexicales produites par l'analyseur lexical - les phrases -sont "grammaticalement" correctes, par rapport à la grammaire du langage utilisé.
C'est généralement l'analyseur syntaxique qui, au fur et à mesure de ses besoins, demande à l'analyseur lexical de lui fournir des unités lexicales.

Puis l'**analyseur sémantique** entre en jeux pour vérifier ces phrases grammaticalement correctes ont un "sens". Il lui revient en général de traiter les aspects contextuels du langage qui ne peuvent être traités par la grammaire qui, comme son nom l'indique, est non contextuelle. Parmi ces aspects contextuels, citons le contrôle de type, l'existence et la visibilité des identificateurs, etc.

Enfin, le dernier module produit la **traduction**. Cette phase peut être traitée de façons bien différentes, avec production d'un code intermédiaire, ou non, etc.
Cette dernière phase, fortement dépendante de la machine déstination, est appellée la partie finale d'un compilateur, par opposition aux phases d'analyse (lexicale, syntaxique, et sémantique) qui sont regroupées sous le terme de partie frontale.

Syntex est donc une machine à fabriquer des analyseurs syntaxiques, ie des vérificateurs de correction grammaticale, à partir d'une description de la grammaire souhaitée.

Il existe bien évidemment plusieurs approches pour pratiquer une analyse syntaxique, de même qu'il existe plusieurs types de grammaires. De plus, certains types de grammaires nécessitent l'utilisation de certaines approches, et réciproquement.

Dans le cas de Syntex, l'analyse syntaxique se fait par descente récursive sur des grammaires de la classe LL(1). Voyons donc tout cela en détail.

b/ L'analyse syntaxique par descente récursive avec un symbole de prévision
---

Les Production - règles de grammaire - contenues dans un fichier .Grm sont en fait des schémas qu'une phrase doit suivre pour être correcte vis à vis de cette grammaire.
Certains éléments de ces schémas sont eux-même décrits dans le fichier .GRM comme des schémas: ce sont des symboles non terminaux. D'autres ne donnent lieu à aucun développement: ce sont les symboles terminaux.

Développer chacun de ces schémas au sein d'une production s'appelle dériver la production.

D'autre part, certaines de ces productions étant récursives, ou imbriquées, il est nécessaire de disposer d'un point d'entrer unique, appelé Axiome.

L'ensemble des productions forme un Arbre Syntaxique (celui que construit Syntex) dont l'axiome est la racine.
Les branches sont constituées des productions, et les feuilles des symboles terminaux.

L'analyse syntaxique descendante consiste à vérifier qu'une phrase est en tout point conforme aux productions; pour cela on vérifie d'abord que l'axiome est vérifié, puis au fur et à mesure que l'on avance dans la phrase, on dérive - développe - les productions qui doivent être mises en jeu, en fonction du symbole de prévision. On "descend" ainsi dans l'arbre syntaxique, de la racine vers les feuilles (en informatique, une structure d'arbre a toujours la tête en bas).

Une analyse syntaxique descendante procède ainsi:

_ Pour un symbole donné, à partir de la position courante dans l'arbre syntaxique, on développe au maximum les productions possibles pour trouver le premier terminal.

_ Si le symbole et le terminal correspondent, on lis le symbole suivant dans la phrase, sinon, on parcourt les éventuelles alternatives de la production pour comparer le symbole avec les terminaux des alternatives, jusqu'à trouver une correspondance, ou jusqu'à épuisement des terminaux possibles à partir de cette position dans l'arbre syntaxique.

Si aucune correspondance n'est trouvée entre le symbole et les terminaux possibles, deux stratégies sont envisageables:

_ soit on rebrousse chemin jusqu'à l'embranchement précédent, pour essayer d'éventuelles alternatives,

_ soit on déclare qu'il y a erreur.

Syntex implémente la deuxième alternative, qui correspond à un analyseur prédictif sans rebroussement avec un symbole de prévision. Ce choix implique que la grammaire ait des propriétés particulières, que nous détaillerons plus loin.

Exemple d'analyse syntaxique descendante avec la grammaire Test.Grm:

Soit la phrase à analyser "bidon = essai + (ratio * longueur);"
^ pointe sur le début du symbole courant fourni par l'analyseur lexical.

```
-------------------------------------------------------------------------##
0 ##
"bidon = essai + (ratio * longueur);"
DEBUT

-------------------------------------------------------------------------##
Etape ##
"bidon = essai + (ratio * longueur);"
^
DEBUT
\
 nom'='expression';'
  \
   NOM
    \
     bidon

-------------------------------------------------------------------------##
Etape ##
"bidon = essai + (ratio * longueur);"
      ^
DEBUT
|
nom   '='  expression';'
|       \
NOM      \
|         \
bidon      =

-------------------------------------------------------------------------##
Etape ##
"bidon = essai + (ratio * longueur);"
       ^
DEBUT
|
nom   '='  expression';'
|      |    \
NOM    |     terme {'+'terme}
|      |      \
bidon  =       facteur {'*'facteur}
               \
                nom
                 \
                  essai
```

```
-------------------------------------------------------------------------##
Etape ##
"bidon = essai + (ratio * longueur);"
            ^
DEBUT
|
nom   '='  expression';'
|      |   |
|      |   terme               { '+'  terme }
|      |   |                       \
bidon  =   facteur {'*'facteur}     \
          |                         +
          nom
          |
          essai
```

```
-------------------------------------------------------------------------##
Etape ##
"bidon = essai + (ratio * longueur);"
              ^
DEBUT
|
nom   '='  expression';'
|      |   |
|      |   terme               { '+'  terme }
|      |   |                       |    \
bidon  =   facteur {'*'facteur}    |     facteur {'*'facteur}
          |                       +      \
          nom                             '(' expression ')'
          |                                 \
          essai                              (
```

```
-------------------------------------------------------------------------##
Etape ##
"bidon = essai + (ratio * longueur);"
              ^
DEBUT
|
nom   '='  expression';'
|      |   |
|      |   terme               { '+' terme }
|      |   |                       |  |
bidon  =   facteur {'*'facteur}    |  facteur {'*'facteur}
          |                       +  |
          nom                        '(' expression ')'
          |                          |    \
```

```
          essai                       (     terme {'+' terme}
                                             \
                                              facteur {'*' facteur}
                                               \
                                                nom
                                                 \
                                                  ratio
```

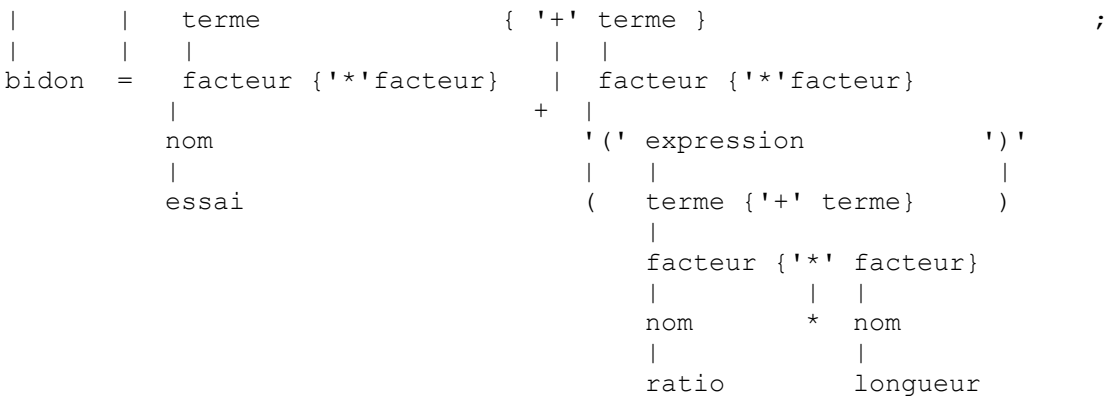
```
-----------------------------------------------------------------------##
Etape ##
"bidon = essai + (ratio * longueur);"
                     ^
DEBUT
|
nom   '='  expression';'
|      |   |
|      |   terme               { '+' terme }
|      |   |                      |  |
bidon  =   facteur {'*'facteur}   |  facteur {'*'facteur}
          |                     +  |
          nom                      '(' expression ')'
          |                        |   |
          essai                    (   terme {'+' terme}
                                       |
                                       facteur {'*' facteur}
                                       |          \
                                       nom         *
                                       |
                                       ratio
```

```
-----------------------------------------------------------------------##
Etape ##
"bidon = essai + (ratio * longueur);"
                     ^
DEBUT
|
nom   '='  expression ';'
|      |   |
|      |   terme               { '+' terme }
|      |   |                      |  |
bidon  =   facteur {'*'facteur}   |  facteur {'*'facteur}
```

```
          |                       +  |
          nom                         '(' expression ')'
          |                           |   |
          essai                       (   terme {'+' terme}
                                          |
                                          facteur {'*' facteur}
                                          |         |   \
                                          nom       *    nom
                                          |              \
                                          ratio           longueur
```

```
-----------------------------------------------------------------------##
Etape ##
"bidon = essai + (ratio * longueur);"
                                 ^
DEBUT
|
nom   '='  expression ';'
|      |   |
|      |   terme               { '+' terme }
|      |   |                     |  |
bidon  =   facteur {'*'facteur}   |  facteur {'*'facteur}
          |                       +  |
          nom                         '(' expression           ')'
          |                           |   |                       \
          essai                       (   terme {'+' terme}       )
                                          |
                                          facteur {'*' facteur}
                                          |         |  |
                                          nom       *  nom
                                          |             |
                                          ratio         longueur
```

```
-----------------------------------------------------------------------##
Etape ##
"bidon = essai + (ratio * longueur);"
                                 ^
DEBUT
|
nom   '='  expression                                              ';'
|      |   |                                                        \
```




```
## Fin ##
-----------------------------------------------------------------------
```

Remarquons que le symbole de prévision n'avance que lorsqu'un terminal concordant avec l'un des terminaux possibles de la production considérée est rencontré.
Dans cet exemple, le symbole rencontré a toujours concordé avec le terminal attendu, la phrase est donc correcte.

Ce parcours de l'arbre syntaxique peut être implémenté sous la forme de procédures récursives; la grammaire est alors traduite de façon implicite - et "visible" - dans la structure de ces procédures; c'est l'un des grands intérêts de ces grammaires, dites LL(1)!

c/ Grammaire LL(1)

Il est possible de construire automatiquement un analyseur syntaxique par descente recursive sans rebroussement et avec un seul symbole d'avance à partir d'une grammaire appartenant à la classe LL(1) - c'est ce que fait Syntex.
Un tel analyseur syntaxique s'appelle analyseur prédictif récursif, que nous nous contenterons d'appeler par son petit nom - "analyseur prédictif".

LL(1) signifie "Left to right scanning - Leftmost derivation" (ie. parcours du source de Gauche à droite - dérivation à Gauche); le "1" indique que chaque étape ne nécessite que la connaîssance du symbole qui suit pour décider de la suite - d'où le qualificatif de "prédictif" -.

En présence d'une production A, Syntex part à la recherche des terminaux correspondants à cette expression après dérivation, afin de déterminer la prochaine production à invoquer. Il parcourt également chaque alternative, si toutefois il y en a. Il produit ainsi une liste de tous les premiers terminaux accessibles après dérivation de la production A.

Pourquoi les premiers seulement?

Parce que c'est une propriété des grammaires LL(1), de toujours permettre de décider de la production suivante à utiliser à la "vue" de l'unique symbole de pré-vision produit à la demande de l'analyseur syntaxique par l'analyseur lexical.

d/ Premier jet

Il existe une définition de la classe LL(1) qui permet de vérifier si une grammaire donnée est bien LL(1).

Une grammaire donnée est de la classe LL(1) si et seulement si, chaque fois qu'elle contient une production de la forme A = B | C,

1. Pour tout terminal a, B et C ne peuvent se dériver tous les deux en a; ie. "partant de deux endroits différents, on doit aboutir à deux destinations différentes".
En effet, comme l'on décide du traitement à effectuer en fonction de la nature de la destination atteinte, on ne saurait plus s'il faut faire un traitement B ou un traitement C, il y a ambiguïté.

2. B et C ne doivent pas tous les deux aboutir à la chaîne vide; c'est à dire que "l'un au moins des deux chemins doit aboutir quelque part".
En effet, si B et C aboutissaient toutes les deux à la chaîne vide, l'ambiguïté sur celle à choisir pour le traitement réapparaitrait.

3. Si C aboutit à la chaîne vide, B ne doit pas pouvoir aboutir à un terminal que l'on peut trouver après A; ie on interdit des constructions du genre "if then then write('toto')" où l'on aurait la même chaîne pour un identificateur et pour un mot clé, car l'analyseur syntaxique aurait le risque de "brancher" sur la mauvaise production, et de refuser une construction qui est pourtant correcte.

En pratique on ne vérifie pas systématiquement que la définition est respectée, mais on procède en rédigeant un premier jet pour sa grammaire, puis on effectue les corrections.
Cette deuxième méthode ne garantit pas que le résultat soit dans la classe LL(1), car elle ne vérifie pas que la grammaire n'est ni ambigüe ni contextuelle, mais elle fonctionne dans la majorité des cas.

Inspirez-vous des fichiers .Grm fournis en exemples pour voir le style à adopter dans votre prose personnelle.

e/ Corrections

On peut effectuer deux types de corrections: des corrections sur la grammaire, et des corrections sur le code produit.

Afin que la grammaire soit effectivement adaptée à la construction automatique d'un analyseur prédictif, ses productions doivent éviter certaines constructions, dont voici la liste.

__ La Récursion à Gauche:

Une production est récursive à gauche lorsqu'elle est de la forme A = Aa, où 'A' est le nom de la production, et 'a' une chaîne quelconque située à droite de la production - la "suite de la production".
On constate que la production A commence par s'appeler elle-même.

Comment le problème survient-il?

L'analyseur syntaxique appelle l'analyseur lexical, et reçoit en réponse une unité lexicale. Supposons alors que pour traiter cette unité lexicale, il faille faire appel à la production A. A appelle à son tour la production A, qui est son premier terme, et cela sans passer à l'unité lexicale suivante, puisque seul l'appel d'un élément terminal de la grammaire provoque la demande d'une

nouvelle unité lexicale.
L'analyseur syntaxique boucle alors récursivement indéfiniment sur la production A. Problème.

Eliminer une récursivité immédiate à gauche revient à transformer A = Aa|b en A = bA' et A' = [aA']. Il arrive parfois que la récursivité à gauche n'apparaisse qu'après avoir dérivé une production au moins deux fois; ce dernier cas doit être traité globalement - nous n'en dirons pas plus.

__ La non Factorisation à Gauche:

Certaines productions peuvent être de la forme A = BC | BD, où 'B','C', et 'D' sont des non terminaux.

Au cours de son fonctionnement l'anayseur syntaxique est amené à traiter la production A, ayant à sa disposition une unité lexicale relevant de la production B.
Le problème vient alors de ce qu'il lui est impossible de décider "à la seule vue de l'unité lexicale" - le "1" de "LL(1)", quelle alternative choisir.

Factoriser à gauche consiste à remplacer A = BC|BD par A = BA' et A' = C|D.

__ Les cycles:

Une grammaire contient un cycle si une production A peut se trouver dérivée en A, après éventuellement plusieurs étapes.

__ Les productions vides:

Une production vide est de la forme A = .

Concernant le code produit, la souplesse - relative - de Syntex par rapport à la définition stricte des grammaires LL(1) permet de générer du code pour des grammaires "presque" LL(1), quitte à faire quelques modifications par la suite sur le code lui-même - l'option Make du menu Generate permet pour cela une compilation (purement gratuite) pour vérifier que le code produit "passe".

f/ Considérations diverses

Lorsque Syntex cherche les premiers terminaux d'une expression accessibles par dérivation, si ces terminaux présentent des alternatives inclues dans un métasymbole, seul le premier terminal de la première alternative est pris en compte.
Notons que cette limitation n'est pas imposée par les grammaires LL(1) ni par la méthode d'analyse prédictive récursive. Cette difficulté est facile à contourner en retouchant légèrement le code produit par Syntex.

Syntex est lui-même un compilateur, qui traduit le texte contenu dans le fichier .Grm en un programme TPascal placé dans un fichier .Pas.
L'option "Generate" regroupe en fait deux sous-phases qui étaient distinctes dans la version 1.x: "MakeTree", qui correspond ainsi à la fonction d'analyse lexicale et syntaxique du texte source, et "GenerateCode" qui regroupe, elle, les fonctions d'analyse sémantique et de production de code.

Bien que Syntex accepte l'imbrication des métasymboles [] et {} sur une seule production, la phase de génération de code à partir de l'arbre syntaxique ne les traite pas. Il est alors nécessaire de passer par des sous-productions traitant ces imbrications.

Syntex n'est pas - pour le moment (?) - un générateur d'analyseur lexical, et seul le corps de cette procédure est généré automatiquement. Cependant, et bien qu'une structure procédurale à base d'appels récursifs soit un peut de la grosse artillerie - on passe plus traditionnellement par des automates construits à partir d'expressions régulières -, il est tout à fait possible d'en passer par là pour certaines parties d'analyse lexicale; la grammaire Pascal des nombres réels donnée au chapitre /3 en est un exemple.

Version History

22/09/93:
_ Saisie du programme brut.

23/09/93:
_ Corrections mineures pour l'adapter à TP4+.
_ C'est la version 1.0.

02/10/93:
_ Implémentation d'un contrôle rudimentaire des E/S.
_ Conventions sur les extensions des fichiers source et destination (resp. .Grm et .Pas).
_ Modification des options de débuggage <List> et <Ponctuation>.
_ Support des chaînes de commentaires dans les fichiers de description de grammaires.
_ C'est la version 1.1.

.. - 06/94:
_ Ré-écriture de l'interface utilisateur à l'aide de Turbo Vision.
_ Ajout de la possibilité de générer des Units à la place des Programs et de compiler le code produit par une grammaire "presque" LL(1).
_ Visualisation souple de l'arbre syntaxique.
_ C'est la version 2.0.

Enregistrement

VOUS AVEZ LE DROIT D'UTILISER SYNTEX 2.0 tel qu'il est livré, ie. sans payer ou envoyer quoi que ce soit.

CEPENDANT, cette version est légèrement bridée: la sauvegarde et la restauration du bureau ne sont pas opérationnelles. Pour obtenir une version comportant ces options activées, je vous demande de m'envoyer un petit quelque chose (Carte postale, coup de fil, chocolat, un modem externe - je n'en ai pas -, quelques sous - autant de fois 30FF que vous le souhaitez -, ou toute autre chose sympathique ...).

L'AUTEUR peut être contacté sur : ou par courrier à l'adresse suivante:

3615 TEASER @BIG JOHN

J.François LE TENO
19, Rue Docteur Bordier
38000 GRENOBLE - FRANCE