Notes de cours

GIF 340 : ÉLÉMENTS DE COMPILATION

Chapitre 6

Analyse syntaxique descendante

Ahmed KHOUMSI

PLAN

Analyse descendante
Parsers LL(1)
Grammaires LL(1)
Réalisation d'un parser LL(1)
Exemple simple de réalisation
Génération d'AST à l'aide d'un parser LL(1)
Récupération sur erreur à l'aide d'un parser LL(1)

Introduction à l'analyse descendante

Soit une grammaire G définie par Vt, Vn, S et P, et soit p une phrase à analyser

But : déterminer si p (qui est une séquence de terminaux) appartient à L(G)

Approche : α représente une séquence de terminaux et/ou non-terminaux
- Initialisation de α par le symbole (non-terminal) de départ S
- Lecture séquentielle des terminaux de p et
  Application à α des règles de productions définies dans P

Si après une séquence de dérivation, α devient égale à p (c-à-d. S a été tranformée en p), alors la phrase p est syntaxiquement correcte

Exemple : Vt = { *id* , + , * }, Vn = { *E* , *T* , *F* }, S = *E*
P = { $E \longrightarrow E + T$, $E \longrightarrow T$, $T \longrightarrow T * F$, $T \longrightarrow F$, $F \longrightarrow id$ }
p = *id + id * id*

Voici une séquence de règles permettant de passer de S à p

$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow$ id + $T \Rightarrow id + T * F$
$\Rightarrow id + F * F \Rightarrow id + id * F \Rightarrow id + id + id$

Problème de l'analyse descendante

On aurait pu effectuer une séquence de dérivations qui n'aurait pas abouti à p

Exemple 1 : $E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow$ id + $T \Rightarrow id + F \Rightarrow id + id$

Exemple 2 : $E \Rightarrow T \Rightarrow T * F \Rightarrow F * F \Rightarrow$ id * $F \Rightarrow id * id$

Exemple 3 : $E \Rightarrow T \Rightarrow F \Rightarrow id$

Voici deux approches permettant de résoudre ce problème
- retour en arrière (backtracking) (voir page 5)
- contraintes sur grammaire (pages 6 à 8)

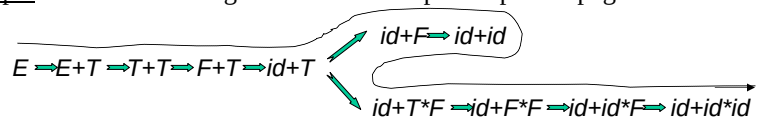
Analyse descendante avec backtracking

Lorsqu'on détecte qu'on a effectué une séquence de dérivations qui ne permet pas d'aboutir à p, alors on commence à effectuer des réductions en parcourant en sens inverse la séquence de dérivation.

On arrête d'effectuer les réductions lorsqu'on peut effectuer une dérivation ne correspondant pas à la réduction la plus récente.

Et ainsi de suite, jusqu'à ce que :
- on obtienne p : dans ce cas, p est syntaxiquement correcte; ou
- on essaye toutes les séquences possibles de dérivations sans obtenir p : dans ce cas, p est syntaxiquement incorrecte.

Exemple : on considère la grammaire G et la phrase p de la page 3



p est syntaxiquement correcte car on arrive à l'obtenir

Cette méthode peut être *très coûteuse*

Analyse descendante avec contraintes sur grammaire

Il s'agit d'imposer des contraintes sur la grammaire G qui assurent que :
- si une phrase p est syntaxiquement correcte
- alors on l'obtient après une seule séquence de dérivations (sans backtracking)

Autrement dit, les contraintes sur G doivent nous garantir que :
- chaque fois qu'il y a un choix entre plusieurs règles de productions
- alors on dispose d'informations qui nous permettent de faire le "bon" choix (c-à-d. le choix qui nous permettra d'aboutir à p si celle-ci est correcte)

Exemple : On utilise un pointeur de lecture *ptr* qui parcourt p de gauche à droite

Information : **connaissance des k terminaux (k ≥ 0) devant *ptr***

c-à-d, on consulte les k prochains terminaux sans avancer le pointeur de lecture.

À partir de cette connaissance des k terminaux après *ptr*, on décide :

(1) s'il faut avancer *ptr*, ou
(2) s'il faut faire une dérivation

Dans le cas (2), la connaissance des k terminaux après *ptr* nous permet aussi de choisir la dérivation adéquate.

La valeur de k dépend de la grammaire et de la méthode d'analyse

Analyse descendante avec contraintes sur grammaire
Exemple 1

Grammaire nécessitant k = 1

Vt = {*a, b*} Vn = {*A*} S = *A* P = {$A \xrightarrow{1} b\,A$, $A \xrightarrow{2} a$ } et p = *b b a*

La séquence de dérivations à appliquer est la suivante

$A \overset{1}{\Rightarrow} b\,A \overset{1}{\Rightarrow} b\,b\,A \overset{1}{\Rightarrow} b\,b\,a$

- Initialement, *ptr* devant 1er *b*
- Dérivation 1 utilisée car *b* est le prochain terminal devant *ptr*
- On avance *ptr* qui se trouve alors devant le 2nd *b*
- Dérivation 1 utilisée car *b* est le prochain terminal devant *ptr*
- On avance *ptr* qui se trouve alors devant *a*
- Dérivation 2 utilisée car *a* est le prochain terminal devant *ptr*

Analyse descendante avec contraintes sur grammaire
Exemple 2

Grammaire nécessitant k = 2

Vt = {*a*} Vn = {*A*} S = *A* P = {$A \xrightarrow{1} a\,A$, $A \xrightarrow{2} a$ } et p = *a a a*

La séquence de dérivations à appliquer est la suivante

$A \overset{1}{\Rightarrow} a\,A \overset{1}{\Rightarrow} a\,a\,A \overset{2}{\Rightarrow} a\,a\,a$

La dérivation 2 ne doit être appliquée que si le prochain terminal est le dernier symbole de p, c-à-d. il est suivi par un symbole désignant la fin de la phrase
Il faut donc connaître les deux prochains symboles pour décider s'il faut appliquer la règle 2.

Plus préciséement :

- Initialement, *ptr* devant 1er *a*
- Dérivation 1 utilisée car les 2 prochains terminaux devant *ptr* sont *a* et *a*
- On avance *ptr* qui se trouve alors devant le 2nd *a*
- Dérivation 1 utilisée car les 2 prochains terminaux devant *ptr* sont *a* et *a*
- On avance *ptr* qui se trouve alors devant le 3ème *a*
- Dérivation 2 utilisée car les 2 prochains terminaux devant *ptr* sont *a* et "Fin de phrase"

Incompatibilité entre grammaire et méthode d'analyse

Une méthode M et une grammaire G sont incompatibles si, quelque soit k, la connaissance des k prochains terminaux ne permet pas de choisir les bonnes dérivations

Autrement dit, même si on connaît d'avance toute la chaîne à analyser, on ne peut pas choisir les bonnes dérivations

Parsers LL

Il s'agit d'analyseurs descendants

Premier L signifie que la phrase analysée est lue de gauche (Left) à droite

Second L signifie qu'on effectue une séquence de dérivations à gauche

Exemple de séquence de dérivations à gauche

$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow id + T \Rightarrow id + F \Rightarrow id + id$

Premier contre-exemple (dérivations à droite)

$E \Rightarrow E + T \Rightarrow E + F \Rightarrow E + id \Rightarrow T + id \Rightarrow F + id \Rightarrow id + id$

Second contre-exemple (mélange de dérivations à droite et à gauche)

$E \Rightarrow E + T \Rightarrow T + T \Rightarrow T + F \Rightarrow F + F \Rightarrow F + id \Rightarrow id + id$

Parsers LL(k)

k est le nombre des prochains terminaux connus

LL(0) : - sont des parsers simples
- mais ne sont applicables que pour des grammaires trop restrictives

LL(k), avec k > 1 : - sont applicables pour des grammaires assez générales
- mais sont : - difficiles à mettre au point et
- peu performants

LL(1) : - sont un bon compromis entre LL(0) et LL(k>1)
- sont d'un grand intérêt pratique

Parsers LL(1)

Propriété avec l'approche LL(1) :
À chaque fois qu'on effectue une dérivation $A \longrightarrow \alpha$, alors il existe une séquence de dérivations qui permettent de transformer α en un préfixe de la partie non encore traitée de la phrase à analyser
(la partie non encore traitée est en fait la partie devant *ptr*)

Exemple : on considère la grammaire de la page 3 (les préfixes sont soulignés)

Partie de p non traitée	Règle $A \longrightarrow \alpha$ appliquée
<u>*id + id * id*</u>	$E \longrightarrow E + T$
<u>*id* + *id* * *id*</u>	$E \longrightarrow T$
<u>*id* + *id* * *id*</u>	$T \longrightarrow F$
<u>*id* + *id* * *id*</u>	$F \longrightarrow id$
id + *id* * *id*	shift de *id*
+ *id* * *id*	shift de +
<u>*id* * *id*</u>	$T \longrightarrow T * F$
<u>*id* * *id*</u>	$T \longrightarrow F$
<u>*id* * *id*</u>	$F \longrightarrow id$
id * *id*	shift de *id*
* *id*	shift de *
<u>*id*</u>	$F \longrightarrow id$

Grammaires LL(1)

Grammaire LL(1) est une grammaire pour laquelle parser LL(1) est applicable

Grammaire LL(1) doit respecter deux contraintes qui peuvent être définies à l'aide de deux concepts PREMIER et SUIVANT

Pour définir ces deux concepts, nous utiliserons les notations suivantes :
- une lettre minuscule désigne un terminal
- une lettre majuscule désigne un non-terminal
- une lettre grecque désigne une séquence de terminaux et/ou non-terminaux
- S représente le symbole de départ d'une grammaire
- $\alpha \overset{*}{\Rightarrow} \beta$ signifie qu'il existe (au moins) une séquence de dérivations qui permet de transformer α en β

Exemples : Vt = {*a*, *b*} Vn = {*A*} S = *A* P = {$A \rightarrow A\,a$, $A \rightarrow b$ } et

$A \overset{*}{\Rightarrow} b$ (règle 2 une fois)
$A \overset{*}{\Rightarrow} Aa$ (règle 1 une fois)
$A \overset{*}{\Rightarrow} AAa$ (règle 1 deux fois)
$A \overset{*}{\Rightarrow} AAAa$ (règle 1 trois fois)
$A \overset{*}{\Rightarrow} ba$ (règle 1 suivie de règle 2)

Concept PREMIER

Définition formelle : PREMIER(α) = {$b \mid \alpha \overset{*}{\Rightarrow} b\ldots$ }

Définition intuitive : PREMIER(α) contient tout terminal qui est le début d'une chaîne dérivable (à gauche) à partir de α

Définition opérationnelle :
- PREMIER(*a*) = {*a*}
- PREMIER(ε) = {ε}
- Si $A \rightarrow b\,\alpha$ alors $b \in$ PREMIER(*A*)
- Si $A \rightarrow B\,\alpha$ alors : PREMIER(*B*) $\subseteq$ PREMIER(*A*)
  Si $B \overset{*}{\Rightarrow} \varepsilon$ alors PREMIER(α) $\subseteq$ PREMIER(*A*)
- Si $\alpha = \beta 1\ \beta 2 \ldots \beta n$ alors :
  - PREMIER($\beta 1$) $\subseteq$ PREMIER(α)
  - Si $\beta 1 \overset{*}{\Rightarrow} \varepsilon$ alors : PREMIER($\beta 2$) $\subseteq$ PREMIER(α)
    …
  - Si $\beta 1, \beta 2, \ldots, \beta i \overset{*}{\Rightarrow} \varepsilon$ alors : PREMIER($\beta i+1$) $\subseteq$ PREMIER(α)
- Si $\beta 1, \beta 2, \ldots, \beta n \overset{*}{\Rightarrow} \varepsilon$ alors : $\varepsilon \in$ PREMIER(α)

Concept SUIVANT

Définition formelle : SUIVANT(A) = {b | S $\xrightarrow{*}$ $\alpha A \gamma$ avec $b \in$ PREMIER(γ) }

Définition intuitive : SUIVANT(A) contient tout terminal qui est le début d'une chaîne qui suit le non-terminal A dans une chaîne dérivable (à gauche) à partir de S

Définition opérationnelle : où $ désigne la fin de la phrase

- Si A est le symbole de départ, alors $ $\in$ SUIVANT(A)
- Si $A \longrightarrow \ldots B\ X \ldots$ alors PREMIER(X)\{ε} $\subseteq$ SUIVANT(B)
  (cas particulier du cas ci-dessus :
  Si $A \longrightarrow \ldots B\ X1\ X2 \ldots Xn\ X \ldots$ alors :
  Si $X1, X2, \ldots, Xn \xrightarrow{*} \varepsilon$ alors :
  PREMIER(Xi)\{ε} $\subseteq$ SUIVANT(B) pour i=1 à n
  PREMIER(X)\{ε} $\subseteq$ SUIVANT(B))
- Si $A \longrightarrow \ldots B$ alors SUIVANT(A) $\subseteq$ SUIVANT(B)
- Si $A \longrightarrow \ldots B\ X1\ X2 \ldots Xn$ alors :
  Si $X1, X2, \ldots, Xn \xrightarrow{*} \varepsilon$ alors : SUIVANT(A) $\subseteq$ SUIVANT(B)

Concepts PREMIER et SUIVANT : Exemple

Soit la grammaire G définie par Vt = { id , + , * ,) , (}, Vn = { E , T , F }, S = E
P = { $E \longrightarrow E + T$, $E \longrightarrow T$, $T \longrightarrow T * F$, $T \longrightarrow F$, $F \longrightarrow id$, $F \longrightarrow (E)$ }

Intuitivement, cette grammaire décrit la syntaxe des expressions arithmétiques constituées d'opérandes (id), des parenthèses ouvrante et fermante, et des opérateurs + et *.

PREMIER(F) = {id, $($ }
PREMIER(T) = PREMIER(F)
PREMIER(E) = PREMIER(T) = PREMIER(F)

SUIVANT(E) = {+, $)$, $ }
SUIVANT(T) = SUIVANT(E) $\cup${* } = {+, $)$, *, $ }
SUIVANT(F) = SUIVANT(T) = {+, $)$, *, $ }

Notons que dans des cas très simples, les ensembles PREMIER et SUIVANT peuvent être déterminés intuitivement. Les définitions opérationnelles des pages 14 et 15 sont surtout utiles pour des exemples non triviaux.

Première contrainte des grammaires LL(1)

Pour que la connaissance du prochain terminal de la phrase à analyser soit suffisante pour choisir la règle de dérivation à appliquer, il faudrait que,

- pour toutes les règles ayant une même partie gauche :

$A \longrightarrow \alpha 1$
$A \longrightarrow \alpha 2$
….
$A \longrightarrow \alpha n$

- on ait : $\text{PREMIER}(\alpha 1) \cap \text{PREMIER}(\alpha 2) \cap \ldots \cap \text{PREMIER}(\alpha n) = \varnothing$

Si t = prochain terminal, alors choisir règle $A \longrightarrow \alpha i$ telle que $t \in \text{PREMIER}(\alpha i)$
Si une telle règle n'existe pas, alors on effectue une opération shift (càd avance ptr)

Intuitivement : Deux chaînes αi et αj ne peuvent pas être dérivées en des phrases commençant par un même symbole terminal. Sinon, si ce dernier est le prochain terminal de la phrase à analyser, on ne peut pas décider s'il faut appliquer $A \longrightarrow \alpha i$ ou $A \longrightarrow \alpha j$

Conséquence : Au plus une des chaînes $\alpha 1, \alpha 2, \ldots, \alpha n$ peut être dérivée en une chaîne vide. Sinon on aurait eu : $\{\varepsilon\} \subseteq \text{PREMIER}(\alpha i) \cap \text{PREMIER}(\alpha j)$

Seconde contrainte des grammaires LL(1)

Comme pour la première contrainte, on considère toutes les règles ayant une même partie gauche :

$A \longrightarrow \alpha 1$
$A \longrightarrow \alpha 2$
….
$A \longrightarrow \alpha n$

S'il existe un αi pour lequel on peut dériver ε (c-à-d. si $\varepsilon \in \text{PREMIER}(\alpha i)$), alors : $(\text{PREMIER}(A) \backslash \text{PREMIER}(\alpha i)) \cap \text{SUIVANT}(A) = \varnothing$

Exemple : $S \longrightarrow A\ a$ $\quad \text{PREMIER}(A) = \{\varepsilon, a\}$
$A \longrightarrow a$ $\quad \text{PREMIER}(S) = \{\varepsilon, a\}$
$A \longrightarrow \varepsilon$ $\quad \text{SUIVANT}(A) = \{a\}$

- Contrainte 1 est respectée : $\text{PREMIER}(a) \cap \text{PREMIER}(\varepsilon) = \varnothing$
- Contrainte 2 non respectée : intuitivement, le problème qui peut se produire est le suivant :
  - on sait que le prochain terminal est a
  - on applique d'abord la règle $S \longrightarrow A\ a$
  - ensuite, on ne sait pas s'il faut effectuer $A \longrightarrow a$ ou $A \longrightarrow \varepsilon$ car les deux règles engendrent une chaîne commençant par a

Ambiguïté et récursivité à gauche

Ambiguïté : Une grammaire ambiguë ne peut pas être LL(1)

Démonstration intuitive : Pour une grammaire LL(1), à chaque étape on connaît la règle à appliquer. On ne peut donc pas avoir plusieurs arbres de dérivations pour une même phrase.

Il n'existe pas de méthode automatique pour supprimer l'ambiguïté.

Récursivité à gauche : Une grammaire possédant une règle récursive à gauche ne peut pas être LL(1)

Exemple : $A \longrightarrow A\ a$
$A \longrightarrow \varepsilon$

PREMIER(A) = { a, ε } et SUIVANT(A) = {a }
La contrainte 2 n'est donc pas respectée

Une méthode pour supprimer la récursivité à gauche a été proposée dans le chapitre 5, page 12

Règles définies à l'aide de métasymboles

Soit une grammaire G définie par Vt, Vn, S et P.

Toutes les règles de P de la forme $X \longrightarrow \ldots$ peuvent être représentées par une seule règle, que nous appellerons RX, à l'aide des métasymboles **|** **[ ]** **{ }**

a **|** b représente le choix entre a et b
[a **]** signifie que a est optionnel
{ a **}** signifie que a est répété un nombre de fois ≥ 0

Exemple d'utilisation de |

$X \longrightarrow \alpha 1$
$X \longrightarrow \alpha 2$
peuvent être remplacées par $X \longrightarrow \alpha 1 \mid \alpha 2$

Exemple d'utilisation de [ ]

$X \longrightarrow \alpha 1\ \alpha 2$
$X \longrightarrow \alpha 2$
peuvent être remplacées par $X \longrightarrow [\ \alpha 1\ ]\ \alpha 2$

Exemple d'utilisation de { }

$X \longrightarrow \alpha 1$
$X \longrightarrow X\ \alpha 2$
peuvent être remplacées par $X \longrightarrow \alpha 1\ \{\ \alpha 2\ \}$

Principe de la descente récursive

Pour chaque non-terminal *X*, on définit une procédure P*X* dont le corps est déterminé par la partie droite de la règle R*X*

P*X* est construit comme suit, à partir de la partie droite de R*X* :
- Un terminal t est traduit par :
  - la vérification que le terminal lu est égal au terminal attendu t
  - une lecture du prochain terminal (effectuée par le scanner)
- Un non-terminal *Y* est traduit par appel de la procédure P*Y*
- Une séquence est traduit par une séquence dans P*X*
- Une alternative simple **[ ]** (dans partie droite de R*X*) est traduite par **if**
- Une alternative multiple (… | … | …) est traduite :
  - soit par **switch**
  - soit par une succession de **if-then-else**
- Une répétition est traduite par **while**

Principe de la descente récursive (suite)

Nous avons vu que :
- le parser a éventuellement le choix entre plusieurs règles de dérivation
- la seule connaissance du prochain symbole (à shifter) doit permettre au parser de faire le bon choix. Ceci est possible dans le cas des grammaires LL(1), où les deux contraintes sont respectées.

Les choix à faire se traduisent dans P*X* par :
- l'alternative simple
- l'alternative multiple
- la répétition

Examinons un peu plus en détail ces 3 cas.

Alternative simple

[α] se traduit par :

```
if  (condition) then
{
   /* instructions traduisant α */
}
```

Soit t le prochain symbole terminal de la phrase à analyser

La condition du if est alors : $t \in$ PREMIER(α)

Alternative multiple à l'aide du switch

(α1 | α2 | ... | αn) se traduit par :

```
switch  (expression) {
   case 1 : /* instructions traduisant α1 */
     break;
   case 2 : /* instructions traduisant α2 */
     break;
   ….
   case n : /* instructions traduisant αn */
     break;
}
```

Soit t le prochain symbole terminal de la phrase à analyser

L'expression du switch est alors la fonction f(t, α1, α2, … , αn) qui retourne :
i tel que $t \in$ PREMIER(αi)

Alternative multiple à l'aide d'une séquence de if-then-else

(α1 | α2 | ... | αn) peut aussi se traduire traduire par :

```
if (condition 1) then
{
    /* instructions traduisant α1 */
}
else if (condition 2) then
{
    /* instructions traduisant α2 */
}
….
else if (condition n) then
{
    /* instructions traduisant αn */
}
```

Soit t le prochain symbole terminal de la phrase à analyser

condition i est alors : t $\in$ PREMIER(αi)

Répétition

{ α } se traduit par :

```
while (condition) do
{
   /* instructions traduisant α */
}
```

Soit t le prochain symbole terminal de la phrase à analyser

La condition du while est alors : t $\in$ PREMIER(α)

Tranformation de la grammaire pour respecter les deux contraintes (exemple)

On considère la grammaire de la page 16, où les règles de productions de P peuvent être regroupées en les trois (méta) règles suivantes :

$E \longrightarrow T \mid E + T$ (1)

$T \longrightarrow F \mid T * F$ (2)

$F \longrightarrow id \mid (E)$ (3)

Il y autant de méta-règles que de symboles non-terminaux

Cette grammaire n'est pas LL(1) car la première contrainte n'est pas respectée.

En effet, dans la règle (1) les ensembles PREMIER des deux alternatives sont égaux et non vides. En effet : PREMIER(T) = {*id* , *(* }

PREMIER($E + T$) = PREMIER(E) = {*id* , *(* }

On peut aussi montrer que la 1ère contrainte n'est pas respectée dans la règle (2)

Ce problème peut être résolu par factorisation, en remplaçant $\alpha\ \beta\ \delta \mid \alpha\ \gamma\ \delta$ par $\alpha\ (\ \beta \mid \gamma\ )\ \delta$

Intuitivement, ceci revient à retarder le plus possible l'alternative

Dans cet exemple, on utilisera aussi la commutativité de + et * pour remplacer

$E + T$ par $T + E$

$T * F$ par $F * T$

Tranformation de la grammaire pour respecter les deux contraintes (exemple) (suite)

On obtient alors :

$E \longrightarrow T\ (\ \varepsilon \mid + E\ )$ (1)

$T \longrightarrow F\ (\ \varepsilon \mid * T\ )$ (2)

$F \longrightarrow id \mid (E)$ (3)

Que l'on peut aussi représenter comme suit :

$E \longrightarrow T\ [\ + E\ ]$ (1)

$T \longrightarrow F\ [\ * T\ ]$ (2)

$F \longrightarrow id \mid (E)$ (3)

Les ensembles PREMIER et SUIVANT sont :

PREMIER(F) = PREMIER(T) = PREMIER(E) = {*id*, *(* }

SUIVANT(E) = { *),* $ } SUIVANT(T) = {+, *)* } SUIVANT(F) = {+, *),* *, $ }

La première contrainte est clairement respectée pour les règles (1) et (2).

La première contrainte est aussi respectée pour la règle (3) car on a : PREMIER(*id*) $\cap$ PREMIER(*(E)*) = $\varnothing$

La seconde contrainte est respectée par les trois règles car aucune des règles ne produit ε.

Écriture des procédures de traitement d'un symble terminal (exemple)

Deux fonctions sont utilisées : lexical et terminal

Fonction lexical()

Elle effectue l'analyse lexicale (voir chapitre 2).
Chaque fois qu'elle est appelée, elle retourne le prochain symbole terminal de la phrase à analyser, qui devient alors le symbole terminal courant.

Si Unilex est le type des terminaux, alors le prototype en langage C de la fonction est :

```
Unilex lexical()
{
    /* reconnaissance d'un symbole
       terminal lors de l'analyse lexicale */
}
```

Elle effectue une tâche similaire à celle de la fonction yylex() générée par LEX

Écriture des procédures de traitement d'un symble terminal (exemple) (suite)

Fonction terminal()

Elle vérifie s'il y a égalité entre :
- le terminal attendu (déterminé par l'analyseur syntaxique à partir de la règle de production en cours)
- le terminal courant (précédemment retourné par l'analyseur lexical)

Si l'égalité est satisfaite alors :
on effectue une opération shift en appelant une autre fois lexical()
Sinon un traitement d'erreur est effectué

Le prototype en langage C de la fonction est :

```
void terminal(Unilex attendu)
{
    if (courant == attendu)
        courant = lexical();
    else erreur();
}
```

où courant est une variable globale de type Unilex

Écriture de la procédure de traitement du symble non-terminal *F* (exemple)

La règle utilisée est $F \longrightarrow id \mid (E)$

Nous ne ferons pas appel à une fonction PREMIER, nous utiliserons le fait que
PREMIER(*id*) = { *id* } et PREMIER(*(E)*) = {*(* }

Le prototype en langage C de la fonction f() est :

```
void  f( )
{
    switch (courant) {
       case  id : terminal(id);
         break;
       case '(' : terminal( '(' );
                  e();
                  terminal( ')' );
         break;
       otherwise : erreur();
    }
}
```


Écriture de la procédure de traitement du symble non-terminal *T* (exemple)

La règle utilisée est $T \longrightarrow F [* T]$

Nous ne ferons pas appel à une fonction PREMIER, nous utiliserons le fait que
PREMIER(* *T*) = {* }

Le prototype en langage C de la fonction t() est :

```
void  t( )
{
    f();
    if (courant == '*') {
       terminal('*');
       t();
    }
}
```

Écriture de la procédure de traitement du symble non-terminal *E* (exemple)

La règle utilisée est $E \longrightarrow T\ [\ +\ E\ ]$

Nous ne ferons pas appel à une fonction PREMIER, nous utiliserons le fait que
PREMIER(+ *E*) = {+ }

Le prototype en langage C de la fonction e() est :

```
void  e( )
{
    t();
    if (courant == '+') {
        terminal('+');
        e();
    }
}
```


Récapitulation des procédures du parser
Programme principal

```
Unilex  courant;    /* variable globale contenant le terminal reconnu */
Unilex lexical()  {  /* corps de la procédure qui retourne terminal reconnu */  }
void  terminal()   { /* corps de la procédure qui retourne terminal reconnu
                        seulement s'il n'y a pas d'erreur syntaxique détectée */  }
void  f()    {  /* corps de la procédure qui traite le non-terminal F */     }
void  t()    {/* corps de la procédure qui traite le non-terminal T */   }
void  e()    {/* corps de la procédure qui traite le non-terminal E */   }
main()
{
   courant = lexical();  /* premier terminal de la phrase analysée
   e();   /* car E est le symbole de départ */
   printf("expression syntaxiquement correcte \n");
}
```

Construction d'AST avec la descente récursive

Nous considérons le même exemple et nous allons montrer comment un AST peut être construit à l'aide la méthode de la descente récursive

Feuilles

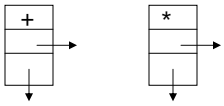
Elles sont associées aux identificateurs

On choisit une structure de feuille constituée d'un seul champ | *id* |

Nœuds

Ils sont associés aux opérateurs + et *

On choisit une structure de nœud constituée de 3 champs



1er champ : opérateur

2ème et 3ème champs : pointeurs sur feuille ou sur nœud

Fonctions de création de feuille et de nœud

Création de feuille

p = CreerFeuille(Unilex id)

- Alloue de la mémoire pour stocker un identificateur
- Dépose l'identificateur id dans la zone allouée
- Retourne un pointeur p sur la zone allouée

Création de noeud

p = CréeNœud(Op, p1, p2)

- Alloue de la mémoire pour les trois champs
- Dépose Op dans le premier champ
- Dépose p1 dans le second champ
- Dépose p2 dans le troisième champ
- Retourne un pointeur sur la zone allouée

Actions associées aux règles de productions

Dans le but de créer l'AST lors de l'analyse syntaxique, on associe une action éventuelle à chaque règle

Chaque procédure e(), t() et f() devient une fonction qui retourne un pointeur Pe, Pt ou Pf

Ceci peut être schématisé comme suit :

Non-terminal *E*

Règle	Action
$E \longrightarrow T$	Pe = Pt
$E \longrightarrow T+E$	Pe = CreerNoeud('+', Pt, Pe)

Non-terminal *T*

Règle	Action
$T \longrightarrow F$	Pt = Pf
$T \longrightarrow F*T$	Pt = CreerNoeud('*', Pf, Pt)

Non-terminal *F*

Règle	Action
$F \longrightarrow id$	Pf = CreerFeuille(id)
$F \longrightarrow (E)$	Pf = Pe

Fonction de traitement du symble non-terminal *F* avec création d'AST

Le prototype en langage C de la fonction f() devient :

```
Nœud * f( )
{
   Nœud *ptr;
   switch (courant) {
      case  id : terminal(id);
                 ptr = CreerFeuille(id);
        break;
      case '(' : terminal( '(' );
                 ptr = e();
                 terminal( ')' );
        break;
      otherwise : erreur();
   }
   return(ptr);
}
```

Ce qui a été ajouté par rapport à la dernière version est encadré en pointillés

Fonction de traitement du symble non-terminal *T* avec création d'AST

Le prototype en langage C de la fonction t() devient :

```
Nœud * t( )
{
   Nœud *ptr1, *ptr2
   ptr1 = f();
   if (courant == '*') {
      terminal('*');
      ptr2 =  t();
      ptr1 = CreerNoeud('*', ptr1, ptr2);
   }
   return(ptr1);
}
```


Fonction de traitement du symble non-terminal *E* avec création d'AST

Le prototype en langage C de la fonction e() devient :

```
Nœud * e( )
{
   Nœud *ptr1, *ptr2;
   ptr1 = t();
   if (courant == '+') {
      terminal('+');
      ptr2 = e();
      ptr1 = CreerNoeud('+', ptr1, ptr2);
   }
   return(ptr1);
}
```

Programme principal du parser avec création d'AST

Le prototype du programme principal devient :

```
main()
{
   courant = lexical();  /* premier terminal de la phrase analysée */
   root_ast = e();   /* root_ast est un pointeur sur la racine de l'AST */
   /* Divers traitements, par exemple parcourt de l'AST */
}
```


Traitement des erreurs avec la descente récursive
Modification de terminal()

Les plus grandes modifications par rapport à la version précédente se retrouvent dans la fonction terminal()
Ceci était prévisible car terminal() est conçu pour ne retourner un terminal que dans le cas où il n'y a pas d'erreur

Première modification de terminal()

Lorsqu'une erreur est détectée, la procédure terminal() doit envoyer un message signalant le type d'erreur
Le prototype en langage C de terminal() devient donc :

```
void  terminal(Unilex attendu, char *nom_terminal)
{
    if (courant == attendu)
          courant = lexical();
    else erreur(nom_terminal);
}
```

La procédure erreur(nom_terminal) consiste, par exemple, à :
- envoyer un message : "Terminal" nom_terminal "attendu mais pas reçu"
- éventuellement tout quitter