Analyse syntaxique

Rappel

L'analyseur syntaxique construit un arbre à partir du flux de lexèmes provenant de l'analyseur lexical. C'est sur cet arbre que se dérouleront les phases suivantes.

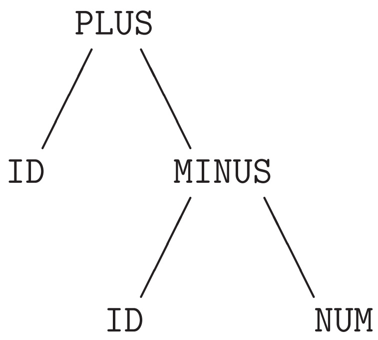
Pour obtenir une représantation plus compacte, il élimine entre autres les parenthèses, qui sont implicites dans la structure de l'arbre.

Entrée: flux de lexèmes provenant de l'analyseur lexical.

Sortie: représentation compacte du programme en forme d'arbre.

```
ID PLUS LPAREN ID MINUS NUM RPAREN
```

$\Longrightarrow$



Survol de cette période:

- Syntaxe concrète et syntaxe abstraite
- Grammaires non-contextuelles
- Le problème d'ambiguïté
- Fonctionnement d'un analyseur par descente récursive

Deux tâches

Le travail d'un analyseur syntaxique comprend deux tâches, l'**analyse** du flux de lexèmes provenant de l'analyseur lexical ainsi que la **synthèse** d'un arbre qui servira de base pour les phases suivantes.

Pour que les productions soient mieux lisibles, on écrit souvent la syntaxe du langage (`"if"`, `"+"`, etc.) au lieu des lexèmes (`IF`, `PLUS`, etc.).

Tâche 1: Analyse

L'analyseur syntaxique reconnaît la structure d'arbre qui est implicite dans les données. Cet **arbre concret** n'est toutefois jamais construit.

Ceci n'est possible que si la grammaire de la **syntaxe concrète** est **non-ambiguë**.

```
Exp  -->  Exp  "+"  Exp
Exp  -->  id
```

Quel est l'arbre de dérivation pour `id "+" id "+" id` ? Il y en a **deux** !

Tâche 2: Synthèse

L'analyseur syntaxique construit un arbre qui suit la structure d'arbre de la syntaxe concrète pour construire un **arbre abstrait**.

Dans ce dernier il n'y a plus de parenthèses (ni d'autres lexèmes superflues). En plus, la grammaire de la **syntaxe abstraite** peut être **ambiguë**.

Ici on choisit une grammaire qui permet une représentation **compacte**. La grammaire ci-dessus serait donc bien utilisable comme grammaire abstraite.

Tâche 1: Analyse

L'analyseur syntaxique détermine la structure d'arbre de l'entrée. Cette fonctionnalité est basée sur la grammaire de la **syntaxe concrète**.

L'arbre de dérivation concret n'est pourtant jamais explicitement construit, c'est immédiatement l'arbre abstrait (voir page 39) qui est conçu.

Une grammaire concrète ne doit pas seulement être non-contextuelle, mais en plus elle doit être **non-ambiguë** (pour pouvoir être réalisée dans un algorithme).

Il existe des techniques pour transformer une grammaire ambiguë en une grammaire non-ambiguë. En théorie ce n'est pas possible pour toutes les grammaires, car il y a **strictement** plus de langages non-contextuels que de langages (avec des grammaires) utilisables dans l'analyse syntaxique. En pratique il est rare qu'on ne puisse pas trouver de grammaire analysable.

Les analyseurs syntaxiques réalisés à la main utilisent habituellement la technique de la **descente récursive** (voir projet), tandis que ceux construit par un **générateur d'analyseurs syntaxiques** comme Yacc ou JavaCup utilisent l'analyse par empilement-réduction (shift-reduce).

Exemple: Une grammaire pour des expressions qui exprime des précédences d'opérateurs.

```
Exp   -->  Term   [ "+"  Exp ]       // [ R ]  : 0 ou 1 fois R
Term  -->  Fact  [ "*"  Term ]
Fact  -->  num   |  "(" Exp ")"
num   -->  "0"   |  "1"
```

Exercice:

• Dérivez `"1 + 0 * ( 0 )"` et esquissez l'arbre de dérivation. Vous verrez que l'arbre est toujours le même, indépendemment de la dérivation. La grammaire est non-ambiguë.

• Comment pourrait-on réaliser la grammaire en Java? (voir aussi projet)

Tâche 2: Synthèse

L'analyseur syntaxique construit l'**arbre abstrait** à en fonction de l'entrée.

Comme sa structure est déjà connue de l'analyse (voir page 38), la grammaire abstraite peut être ambiguë. Pour réduire la taille de l'arbre, on renonce aux parenthèses et d'autres lexèmes (par exemple?).

Exemple:

```
Exp   -->  Exp  "+"  Exp  |  Exp  "*" Exp  |  num
num   -->  "0"  |  "1"
```

ou encore

```
Exp   -->  Exp  Op  Exp  |  num
Op    -->  "+"  |  "*"
num   -->  "0"  |  "1"
```

Exercice:

- Esquissez les arbres résultants de l'application de chacune des grammaires à `"1 + 0 * ( 0 )"` en gardant la structure de l'arbre concret.
- Quels arguments pour ou contre chacune d'elles trouvez-vous?
- Montrez que les deux grammaires sont ambiguës.

Ambiguïtés

Une grammaire est dite **ambiguë** s'il existe un mot dans les langages qu'elle produit avec au moins deux arbres de dérivation différents. (**Attention:** des différentes dérivations sont bien possibles! Voir exercice, page 38.)

Nous discutons quelques sources d'ambiguïté et d'autres problèmes typiques, ainsi que des solutions (voir page 41):

Récursion à gauche:

```
Exp   -->  Exp  "+"  Term
```

La fonction `exp()` s'appelle soi-même sans avancer. Boucle infinie.

Problème du "dangling `else`":

```
Cond  -->  "if"  Exp  "then"  Stm  "else"  Stm
Cond  -->  "if"  Exp  "then"  Stm
```

Quand la fonction `cond()` rencontre un `"if"` (représenté par le lexème `IF`), elle ne peut pas savoir si un `"else"` va suivre.

A quel `"if"` appartient le `"else"` dans `"if a then if b then c else d"` ?

Précédence:

```
Exp   -->  Exp  "+"  Exp  |  Exp "*" Exp  |  "(" Exp ")"  |  "2"
```

Dans cette grammaire, `"+"` et `"*"` ont la même précédence. Pour l'illustrer, construisez deux arbres différentes pour `"2 + 2 * 2"`.

Elimination des ambiguïtés

Récursion à gauche:

On utilise une récursion à droite en introduisant un nouveau symbole non-terminal `Exp'`.

```
Exp   -->  Term Exp'
Exp'  -->  "+"  Term  Exp'  |  eps    // eps : mot vide
```

ou encore

```
Exp   -->  Term { "+" Term }
```

Comment est-ce que cela change la structure de l'arbre pour `"1 + 1 + 1"`?

Problème du "dangling `else`":

Factorisation à gauche, avec un nouveau symbole non-terminal `Else`.

```
Cond  -->  "if"  Exp  "then"  Stm  Else
Else  -->  "else"  Stm  |  eps
```

ou encore

```
Cond  -->  "if"  Exp  "then"  Stm  [ "else" Stm ]
```

Qu'est-ce que cela implique pour `"if a then if b then c else d"` ?

Précédence:

La hiérarchie des symboles non-terminaux reflète une hiérarchie des opérateurs, ce qui crée les précédences.

```
Exp    -->  Term   Exp’              Exp’   -->  "+"  Exp    |  eps
Term   -->  Factor  Term’            Term’  -->  "*"  Term   |  eps
Factor  -->  "2"  |  "(" Exp ")"
```

ou encore

```
Exp    -->  Term    { "+"  Term }
Term   -->  Factor  { "*"  Factor }
Factor  -->  "2"  |  "(" Exp ")
```

Exercice: Esquissez l’arbre pour `"2 + 2 * 2"`.

La première grammaire correspond à un style de réalisation **fonctionnel**, tandis que la deuxième correspond à un style **impératif**:

```
Tree exp() {   // retourne l’arbre abstrait
  Tree t = term();
  if (c == PLUS) {
    next(); Tree e = exp();
    return new Exp(t, PLUS, e); }
  return t;
}

Tree exp() {
  Tree t = term();
  while (c == PLUS) {
    next(); Tree e = term();
    t = new Exp(t, PLUS, e); }
  return t;
}
```

Fonctionnement

Une fonction pour chaque non-terminal analyse le flux d'entrée et produit un arbre abstrait.

La classe de l'analyseur syntaxique est une sous-classe de l'analyseur lexical. Comme ça la fonction principale du compilateur va seulement définir un seul objet de la classe de l'analyseur syntaxique. Pas d'objet propre à l'analyseur lexicale.

Elle utilise la fontion `next()` pour avancer d'un lexème à l'autre. Le lexème est stocké dans `symbol`, la représentation dans `chars`, et la position dans `pos`.

```
import scanner.*;

class Parser extends Scanner {

        // methode constructeur
        public Parser(InputStream in) {
                super(in);
                next();       // on lit deja le premier lexeme
        }                     // (1 symbole "lookahead")

        // objet construisant les arbres
        TreeFactory make = new TreeFactory();

        // fonctions d'analyse et de construction
        ...
}
```

Les arbres

Nous allons travailler avec une seule sur-classe d'arbres, `Tree`. Mais il est aussi possible d'en avoir plusieurs. Ou bien une hiérarchie de (sous-)classes d'arbre. Voir aussi exercice 4.

Une sous-classe pour chaque non-terminal de la grammaire abstraite

```
abstract class Tree {
  int pos;
}
class Prog extends Tree {
  Tree[] defs; Tree[] procs;
}
class Process extends Tree {
  String id; Tree block;
}
...
```

Nous utilisons une "**tree factory**", c-à-d. au lieu d'utiliser les constructeurs directement dans l'analyseur lexicale, nous créons un objet `TreeFactory make`. Ce concept du génie logiciel

- sépare la construction de la représentation des arbres, ce qui peut être utile lors de modifications de la représentation;
- évite de jongler avec des constructeurs dans le code de l'analyseur syntaxique;
- rend le travail en groupes de programmeurs plus facile (un groupe pour la représentation, un autre pour la construction);
- peut aider à éviter et mieux détecter des erreurs, par du code mieux structuré.

Les fonctions

Une fonction pour chaque non-terminal de la syntaxe concrète.

Les fonctions retournent un arbre ou une liste d'arbres; la sous-classe retournée n'est pas spécifiée dans le type de retour.

Processus

Un processus commence par le mot clé `proc` suivi par un identificateur et un bloc.

Dans la grammaire concrète: `Proc` $\longrightarrow$ `"proc" id Block`

Dans le programme:

```
// Proc  -->  "proc" id Block
public Tree proc() {
  if (symbol == PROC) {
      next();
      if (symbol == ID) {
          String id = chars;     // memoriser le nom du processus
          next();                // avancer sur le bloc
          Tree block = block();  // construire l'arbre du bloc
          return make.Process(pos, id, block);
      } else error("process id expected");
   } else error("process expected");
}
```

Enchaînement de trois sous-mots. Traitement de gauche à droite. Voir aussi page 33.

`Block` est un non-terminal $\Rightarrow$ il existe une fonction `Tree block()` qui produit un arbre abstrait.

Blocs

Un bloc commence par une accolade gauche suivie d'une suite de commandes et se termine par une accolade droite.

Dans la grammaire concrète: `Block` $\rightarrow$ `"{" Stms "}"`

```
// Block  -->  "{" Stms "}"
public Tree block() {
        if (symbol == LBRACE) {
            next();
            Tree[] stms = stms();
               if (symbol == RBRACE) {
                   next();
                   return make.Block(pos, stms);
               } else error("right brace expected");
        } else error("left brace expected");
}
```

Enchaînement de trois sous-mots. Traitement de gauche à droite. Voir aussi page 33.

`Stms` est un non-terminal $\Rightarrow$ il existe une fonction `Tree[] stms()` qui produit un arbre abstrait.