ANALYSE SYNTAXIQUE

DEFINITIONS

Objectif : Reconnaître les phrases appartenant à la syntaxe du langage. Son entrée est le flot de *tokens* et sa sortie un arbre de syntaxe abstraite ie. passer de `fun x -> ( x + 1 )` à `Fun("x", App(App(Op +, Var "x") 1))`.

En particulier, l'analyse syntaxique doit détecter les erreurs de syntaxe et :

- les localiser précisément
- les identifier
- éventuellement reprendre l'analyse pour découvrir de nouvelles erreurs

On va utiliser :

- une **grammaire non contextuelle** pour décrire la syntaxe
- un **automate à pile** pour la reconnaître

C'est l'analogie des regexp/automates finis utilisés dans l'analyse lexicale.

Définition : Une grammaire non contextuelle est un quadruplet *(N, T, S, P)* où :

- N est un ensemble fini de **symboles non terminax**
- T est un ensemble fini de **symboles terminaux**
- $S \in N$ le symbole de départ, dit **axiome**
- $R \subseteq N \times (N \cup T)^\star$ est un ensemble fini de **règles de production**

Exemple : N = {E}, T = {+, *, (,) , int}, S = E, R = {(E, E+E), (E, E*E), (E, (E)), (E, int)}

En pratique, on note les règles sous la forme :

```
E ->  E + E
    | E * E
    | ( E )
    | int
```

Les *terminaux* de la grammaire seront les tokens produits par l'analyse lexicale. `int` dégien ici le token correspondant à une constante entière.

Définition : Un mot $u \in (N \cup T)^\star$ se **dérive** en un mot $v \in (N \cup T)^\star$, et on note $u \to v$ s'il existe une décomposition

$$u = u_1 X u_2$$

avec $X \in N, X \to \beta \in R$ et

$$v = u_1 \beta u_2$$

Une suite $w_1 \to w_2 \to ... \to w_n$ est une dérivation. On parle de **dérivation gauche** (resp. **droite**) si le non-terminal réduit est systématiquement le plus à gauche (resp. le plus à droite). On note $\to^\star$ la cloture transitive et réflexive de $\to$.

Exemple : $E \times (\underbrace{E}_{X}) \to E \times (\underbrace{E + E}_{\beta})$

On a, en particulier mais pas uniquement : $E \to^\star int * (int + int)$

Définition : Le **langage** défini par une grammaire non contextuelle $G = (N, T, S, R)$ est l'ensemble des mots de $T^\star$ dérivés de l'axiome, i.e.

$$L(G) = \{w \in T^\star \mid S \to^\star w\}$$

Définition : À toute dérivation $S \to^\star w$, on peut associer un **arbre de dérivation**, dont les nœuds sont étiquetés ainsi :

- la racine est S
- les feuilles forment le mot w dans l'ordre infixe
- tout nœud interne X est un non-terminal dont les fils sont étiquetés par $\beta \in (N \cup T)^\star$ avec $X \to \beta$ une règle de dérivation

L'arbre de dérivation "capture" tous les chemins représentant la même dérivation (dérivation gauche, droite ou ni l'un ni l'autre).

Définition : Une grammaire est ambiguë si un mot admet plusieurs arbres de dérivation.

Exemple : En reprenant la grammaire, ambiguë, des exemples précedents, on peut construire une autre grammaire, non ambiguë, qui définit le même langage :

```
E ->  E + T
    | T
T ->  T * F
    | F
F -> (E)
    | int
```

Cette grammaire traduit la priorité de la multiplication sur l'addition et le choix d'une associativité à gauche pour ces deux opérations.

Problème : La détermination du caractère ambiguë d'une grammaire est un problème *non décidable*. C'est-à-dire qu'un programme ne peut pas le faire en général.

Solution : On va utiliser des **critères décidables suffisants** pour garantir qu'uen grammaire est non ambiguë, et pour lesquels on sait en outre décider l'appartenance au langage efficacement (via un automate à pile déterministe). Les classes de grammaires définies par ces critères sont $LR(0)$, $SLR(1)$, $LALR(1)$, $LR(1)$, $LL(1)$, etc.

ANALYSE ASCENDANTE

Idée : On lit l'entrée de la gauche vers la droite et on cherche à reconnaître des membres droits de productions pour construire l'arbre de dérivation de bas en haut.

L'analyse manipule une pile qui est un mot de $(T \cup N)^\star$ À chaque isntant, deux actions sont possibles :

- **lecture** (*shift*) : on lit un terminal de l'entrée et on l'empile.
- **réduction** (*reduce*) : on reconnaît en sommet de pile le membre droit β d'une production $X \to \beta$ et on remplace β par X en sommet de pile.

Initialement, la pile est vide. Lorsqu'il n'y a plus d'action possible, l'entrée est **reconnue** si elle a été entièrement lue et si la pile est réduite à S.

Comment prendre la décision lecture/réduction? Et se servant d'un automate fini et en examinant les k premiers caractères de l'entrée : c'est l'analyse $LR(k)$ (Knuth). (LR signifie " **L**eft to right scanning, **R**ightmost derivative").

En pratique, $k = 1$.

Fonctionnement :

La pile est de la forme $s_0 x_1 s_1 ....x_n s_n$ où s_i état de l'automate et $x_i \in T \cup N$ comme auparavant.

Soit a le premier token de l'entrée. Une table indexée par s_n et a nous indique l'action à effectuer :

- Si c'est un usccès on un échec, on s'arrête
- Si c'est une lecture, on empile a et l'état s résultant de la transition $s_n \xrightarrow{a} s$
- Si c'est une réduction $X \to \alpha$ avec α de longueur p, alors on doit trouver α en sommet de pile :

  $$s_0 x_1 s_1 ...x_{n-p} s_{n-p} | \alpha_1 s_{n-p+1} ...\alpha_p s_n$$

  on dépile alors α et on empile Xs, où s l'état résultant de la transition $s_{n-p} \xrightarrow{X} s$ dans l'automate, i.e.

  $$s_0 x_1 s_1 ...x_{n-p} s_{n-p} X s$$

En pratique, on ne travaille pas directement avec l'automate mais avec deux tables :

- une table d'**actions** ayant pour lignes les états et pour colonnes les terminaux; la case `action(s,a)` indique :
  - *shift* s' pour une lecture et un nouvel état s'
  - *reduce* $X \to \alpha$ pour une réduction
  - un succès
  - un échec
- Une table de **déplacements** ayant pour lignes les états et colonnes les non-terminaux; la case `goto(s, X)` indique l'état résultant d'une réduction de X.

On ajoute un caractère spécial, #, désignant la fin de l'entrée. On peut le voir comme l'ajout d'un nouveau non-terminal S (devenant l'axiome).

L'analyse ascendante est puissante mals le calcul des tables est complexe. Ce travail est automatisé par des outils : `yacc`, `menhir`, `bison`, `cup`, `ocamlyacc`, ...

L'OUTIL MENHIR

Menhir est un outil qui transforme une grammaire en un analyseur OCaml. Il est basé sur une analyse $LR(1)$. Chaque production de la grammaire est accompagnée d'une **action sémantique** *i.e.* du code OCaml construisant une valeur sémantique (typiquement un arbre de syntaxe abstraite). Menhir s'utilise conjointement avec un analyseur lexical (typiquement `ocamllex`).

Un fichier Menhir porte le suffixe `.mly`. Comme avec `ocamllex`, on a droit à un prélude et un postlude en OCaml.

```
%{
  ... (* code OCaml arbitraire *)
%}
... (* déclaration des tokens *)
... (* déclaration des prédécences, associativités *)
... (* déclaration des points d'entrée *)
%%
non-terminal-1 :
| production { action }
| production { action }
;
```

```
non-terminal-2 :
| production { action }
...
%%
... (* code OCaml arbitraire *)
```

Exemple :

```
%token PLUS LPAR RPAR EOF
%token <int> INT
%start <int> phrase
%%

phrase :
  e = expression; EOF { e }
;

expression :
  | e1 = expression; PLUS; e2 = expression {e1 + e2}
  | LPAR; e = expression; RPAR { e }
  | i = INT { i }
;
```

On compile le fichier avec `% menhir -v arith.mly`. On obtient du code OCaml pur dans `arith.ml(i)`, qui contient notamment :

- la déclaration de : `ocaml type token = RPAR | PLUS | LPAR | INT of int | EOF`
- pour chaque non-terminal déclaré avec `%start`, une fonction du type

```
ocaml val phrase :  (Lexing.lexbuf -> token) -> Lexing.lexbuf ->
int
```

Cette fonction prend en argument un analyseur lexical, du type celui produit par `ocamllex`. Donc il faut fournir à `menhir` l'analyseur lexical, lui-même constuit à partir des déclarations de tokens de `menhir`.

Quand on combine `ocamllex` et `menhir` :

- `lexer.mll` fait référence aux tokens définis dans `parser.mly`. `ocaml { open Parser }`
- l'analyseur lexical et l'analyseur syntaxique sont combinés ainsi : `ocaml let c = open_in file in let lb = Lexing.from_channel c in let e = Parser.phrase Lexer.token lb in ...`

Lorsque la grammaire n'est pas $LR(1)$, Menhir présente les **conflits** à l'utilisateur :

- le fichier `.automaton` contient une description de l'automate $LR(1)$; les conflits y sont mentionnés.
- le fichier `.conflicts` contient, le cas échéant, une explication de chaque conflit, sous la forme d'une séquence de tokens conduisant à deux arbres de dérivation.

Une manière de résoudre les conflits est d'indiquer à Menhir comment choisir entre lecture et réduction. Pour cela, on peut donner des **priorités** (*prédécences*) aux tokenset productions, ainsi que desrègles d'**associativité**.

Par défaut, la priorité d'une production est celle de *son token le plus à droite.*

Si la priorité de la production est supérieure à celle du token à lire, alors la réduction est favorisée. Sinon, la lecture est favorisée. En cas d'égalité, l'**associativité** est consultée : un token associatif à gauche favorise la réduction et un token associatif à drotie favorise la lecture.

Dans notre **exemple**, on ajoute :

```
%left PLUS
```

Pour associer des priorités aux tokens, on utilise la convention suivante :

- l'ordre de déclaration des associativités fixe les priorités (les *premiers* ont les priorités *les plus faibles*)
- plusieurs tokens peuvent apparaître sur la même ligne, ayant ainsi une même priorité.

Exemple :

```
%left PLUS MINUS
%left TIMES DIV
```

Autre problème possible : `IF a THEN IF b THEN c ELSE d` -> ambigu Pour associer le `ELSE` au `THEN` le plus proche, il faut privilégier la lecture :

```
%nonassoc THEN
%nonassoc ELSE
```

Menhir offre de nombreux avantages par rapport aux outisl traditionnels :

- non-terminaux paramétrés par des (non-)terminaux. En particulier facilités pour écrire des regexp, des listes avec séparateur, etc.
- explication des conflits
- mode interactif
- analyse $LR(1)$, là où la plupart des outils n'offrent que du $LALR(1)$ (cf. plus bas).

Pour que les phases suivantes de l'analyse (typiquement le typage) puissent **localiser** les messages d'erreur, il convient de conserver une information de lcoalisation dans l'arbre de syntaxe abstraite. Menhir fournit cette information dans `$startpos` et `$endpos`, deux valeurs du type `Lexing.position`, transmise par l'analyseur lexical.

/!\ `ocamllex` ne maintient automatiquement que la position absolue dans le fichier. Pour avoir les numéros de ligne et colonne à jour, il faut appeler `Lexing.new_line` pour chaque retour chariot.

Une façon de conserver l'information de localisation dans l'arbre de syntaxe abstraite est la suivante :

```
type expression =
  { desc : desc;
    loc : Lexing.position * Lexing.position};;

and desc =
  | Econst of int
  | Eplus of expression * expression
  | Eneg of expression
  | ...
```

On a ainsi :

```
expression:
| d = desc { {desc = d; loc = $startpos, $endpos} }
;

desc :
| i = INT {Econst i}
| e1 = expression; PLUS; e2 = expression {Eplus (e1, e2)}
| ...
;
```

Comme dans le cas d'`ocamllex`, il faut s'assurer de l'application de `menhir` avant le calcul des dépendances.

Si on utilise `dune`, on indique la présence d'un fichier `menhir` :

```
(ocamllex (modules lexer))
(menhir (flags --explain --dump) (modules parser))
(executable (name minilang))
...
```

CONSTRUCTION DE L'AUTOMATE ET DES TABLES

Définition (NULL) : Soit $\alpha \in (T \cup N)^\star$. NULL($\alpha$) est vrai *ssi* on peut dériver ϵ à partir de α, i.e.

$$\alpha \to^\star \epsilon$$

Définition (FIRST) : Soit $\alpha \in (T \cup N)^\star$. FIRST($\alpha$) est l'ensemble de tous les premiers terminaux des mots dérivés de α, i.e.

$$\{a \in T \mid \exists w.\ \alpha \to^\star aw\}$$

Définition (FOLLOW) : Soit $X \in N$. FOLLOW(X) est l'ensemble de tous les terminaux qui peuvent apparaître après X dans une dérivation, i.e.

$$\{a \in T \mid \exists u, w, S.\ S \to^\star uXaw\}$$

Pour calculer NULL(α), il suffit de déterminer NULL(X) pour tout $X \in N$.

NULL(X) est vrai *ssi* :

- il existe une production $X \to \epsilon$
- ou il existe une production $X \to Y_1 ... Y_m$ où NULL(Y_i) pour tout i

Problème : ensemble d'équations mutuellement récursives.

Dit autrement, si $N = \{X_1, ..., X_n\}$ et $\vec{V} = (NULL(X_1), ..., NULL(X_n))$, on cherche la **plus petite solution** d'une équation de la forme (équation de point fixe)

$$\vec{V} = F(\vec{V})$$

Théorème (Tarski): Soit A un ensemble fini muni d'une relation d'ordre $\leq$ et d'un plus petit élément ϵ. Toute fonction $f : A \to A$ croissante admet un plus petit point fixe.

Dans notre cas, $A = BOOL \times ... \times BOOL$ avec $BOOL = \{false, true\}$. On peut munir $BOOL$ de l'ordre $false \leq truetrue$ et A de l'ordre point-à-point :

$$(x_1, ..., x_n) \leq (y_1, ... y_n) \iff \forall i.\ x_i \leq y_i$$

Le théorème s'applique alors en prenant $\epsilon = (false, .., false)$ car la fonction calculant $NULL(X)$ à partir des $NULL(X_i)$ est croissante.

On part de $NULL(X_i) = false, ..., NULL(X_n) = false$ et on applique les équatiosn jusqu'à ce que les valeurs ne changent plus.

De la même façon : les équations définissant $FIRST$ sont mutuellement récursives.

$$FIRST(X) = \bigcup_{X \to \beta} FIRST(\beta)$$

et on a :

- $FIRST(\epsilon) = \emptyset$
- $FIRST(a\beta) = \{a\}$
- $FIRST(X\beta) = FIRST(X)$, si $\neg NULL(X)$
- $FIRST(X\beta) = FIRST(X) \cup FIRST(\beta)$, si $NULL(X)$

De même, on procède par calcul de point fixe sur le produit cartésien

$$A = \mathcal{P}(T) \times \cdots \times \mathcal{P}(T)$$

muni, point-à-point, de l'ordre $\subseteq$ avec $\epsilon = (, \dots,)$

Enfin, on a encore des équations mutuellement récursives pour $FOLLOW$:

$$FOLLOW(X) = \bigcup_{Y \to \alpha X \beta} FIRST(\beta) \ \cup \bigcup_{Y \to \alpha X \beta, NULL(\beta)} FOLLOW(Y)$$

Là encore, on procède par calcul de point fixe. En pratique, on ajoute # dans les suivants du symbole de départ.

Automate LR

Pour l'instant, on suppose $k = 0$. De plus, on commence par construire un automate **asynchrone**, i.e. pourvu de **ϵ-transitions**. Les **états** sont des items de la forme $[X \to \alpha \bullet \beta]$ où $X \to \alpha\beta$ est une production de la grammaire. L'intuition est "Je cherche à reconnaître X, j'ai déjà lu α et je dois encore lire β".

Les **transitions** sont étiquetés par $T \cup N$ et sont les suivantes :

- $[Y \to \alpha \bullet a\beta] \xrightarrow{a} [Y \to \alpha a \bullet \beta]$
- $[Y \to \alpha \bullet X\beta] \xrightarrow{X} [Y \to \alpha X \bullet \beta]$
- $[Y \to \alpha \bullet X\beta] \xrightarrow{\epsilon} [X \to \bullet\gamma]$, pour toute production $X \to \gamma$

Déterminisation

Pour cela, on regroupe les états reliés par des ϵ-transitions. Les états de l'automate déterministe sont dons des ensembles d'items.

Par construction, chaqué état s est **saturé** par la propriété :

- si $Y \to \alpha \bullet X\beta \in s$
- et si $X \to \gamma$ est une production
- alors $X \to \bullet\gamma \in s$

Construction des tables

On construit la table `action` :

- action(s, $\#$) = succès si $[S \to E \bullet \#] \in s$
- action(s, a) = shift s' si on a une transition $s \xrightarrow{a} s'$
- action(s, a) = reduce $X \to \beta$ si $[X \to \beta\bullet] \in s$
- échec dans les autres cas

ainsi que la table `goto` :

- goto(s, X) = s' ssi on a une transition $s \xrightarrow{X} s'$

La table $LR(0)$ peut contenir deux types de conflits :

- un conflit **lecture/réduction** (*shift/reduce*), si dans un état s on peut effectuer une lecture mais aussi une réduction.
- un conflit **réduction/réduction** (*reduce/reduce*), si dans un état s, deux réductions différentes sont possibles.

Définition : Une grammaire est dite $LR(0)$ si la table ainsi construite ne contient pas de conflit.

On peut résoudre un conflit *shift/reduce* de deux façons :

- On favorise la **lecture**, traduisant une associativité à droite
- On favorise la **réduction**, traduisant une associativité à gauche

La construction $LR(0)$ engendre très facilement des conflits. On va donc chercher à limiter les réductions. Une idée simple consiste à poser :

action(s, a) = reduce $X \to \beta$ ssi

$$[X \to \beta \bullet] \in s \text{ et } a \in FOLLOW(X)$$

Définition : Une grammaire est dite $SLR(1)$ (Simple LR) si la table ainsi construite ne contient pas de conflit.

En pratique : $SLR(1)$ est trop restrictive et beaucoup de langages causeraient des conflits.

On introduit une classe de grammaire encore plus large, $LR(1)$, au prix de tables encore plus grandes. Dans cette nouvelle analyse, les *items* ont la forme :

$$[X \to \alpha \bullet \beta, a]$$

Donc la signification est “Je cherche à reconnaître X, j'ai déjà lu α, je dois encore lire β puis vérifier que le terminal suivant est a”.

Les transitions sont alors :

$$[Y \to \alpha \bullet a\beta, b] \xrightarrow{a} [Y \to \alpha a \bullet \beta, b]$$

$$[Y \to \alpha \bullet X\beta, b] \xrightarrow{X} [Y \to \alpha X \bullet \beta, b]$$

$$[Y \to \alpha \bullet a\beta, b] \xrightarrow{\epsilon} [Y \to \bullet\gamma, c], \text{ pout tout } c \in FIRST(\beta b)$$

L'état initial est celui qui contient $[S \to \bullet\alpha, \#]$

Comme précédemment, on peut déterminer l'automate et construire la table correspondante. On introduit une action de réduction pour (s, a) seulement lorsque s contient un item de la forme $[X \to \alpha\bullet, a]$.

Définition : Une grammaire est dite $LR(1)$ si la table ainsi construite ne contient pas de conflit.

La construction $LR(1)$ pouvant être très coûteuse, il existe des approximations, comme $LALR(1)$ (*LookAhead LR*), notamment utilisée par la famille `yacc` (cf. *Dragon's book*, 4.7).