Compilation

Réalisation d'un compilateur pour le langage RAT

Utilisation de Xtext

1 De la grammaire formelle à la représentation EBNF

1.1 Rappel : la grammaire du langage Rat

1. $PROG' \rightarrow PROG$
2. $PROG \rightarrow FUN\ PROG$
3. $FUN \rightarrow TYPE\ id\ (\ DP\ )\ \{IS\ return\ E\ ;\}$
4. $PROG \rightarrow\ id\ \ BLOC$
5. $BLOC \rightarrow\ \{\ IS\ \}$
6. $IS \rightarrow\ I\ \ IS_1$
7. $IS \rightarrow$
8. $I \rightarrow\ TYPE\ \ id\ =\ E\ ;$
9. $I \rightarrow\ id\ =\ E\ ;$
10. $I \rightarrow\ const\ \ id\ =\ entier\ ;$
11. $I \rightarrow\ print\ \ E;$
12. $I \rightarrow\ if\ \ E\ \ BLOC_1\ \ else\ \ BLOC_2$
13. $I \rightarrow\ while\ \ E\ \ BLOC$
14. $DP \rightarrow$
15. $DP \rightarrow TYPE\ id\ DP$
16. $TYPE \rightarrow\ bool$
17. $TYPE \rightarrow\ int$
18. $TYPE \rightarrow\ rat$
19. $E \rightarrow call\ id\ (\ CP\ )$
20. $CP \rightarrow$
21. $CP \rightarrow E\ CP$
22. $E \rightarrow\ [\ E\ /\ E\ ]$
23. $E \rightarrow\ num\ \ E$
24. $E \rightarrow\ denom\ \ E$
25. $E \rightarrow\ id$
26. $E \rightarrow\ true$
27. $E \rightarrow\ false$
28. $E \rightarrow\ entier$
29. $E \rightarrow\ (\ E\ +\ \ E\ )$
30. $E \rightarrow\ (\ E\ *\ \ E\ )$
31. $E \rightarrow\ (\ E\ =\ \ E\ )$
32. $E \rightarrow\ (\ E\ <\ \ E\ )$

1.2 EBNF (Exetended Backus-Naur Form)

1.2.1 BNF

BNF se base sur un nombre donné de métasymboles, avec lesquels décrire une syntaxe. Ces métasymboles sont combinés à des symboles "terminaux" et "non terminaux ".

Métasymbole	Utilisation	Traduction	Exemple
: := ou :=	Attribution	"est défini comme" ou "contient"	x : := "lapin !"
|	Alternatives	"ou"	x | y | z
< et >		encadre les noms des non terminaux	<SymboleNonTerminal>

1.2.2 EBNF

La notation BNF a ensuite été étendue sous le nom EBNF (Extended BNF). Cette extension permettait l'usage d'une syntaxe proche des expressions régulières pour décrire certains aspects, comme la répétition ou le regroupement de blocs.

Symbole	Signification
x ?	0 ou une occurrence de x
x+	au moins une occurrence de x
x*	0 ou plusieurs occurrences de x
x y	x suivi de y
(x y)	regroupement de la séquence x y
$[x\|y\|z]$	regroupement d'éléments optionnels

1.2.3 Exemples

BNF Grammaire non ambiguë des expressions :

1. $expr \rightarrow term + expr$
2. $expr \rightarrow term$
3. $term \rightarrow term * factor$
4. $term \rightarrow factor$
5. $factor \rightarrow (expr)$
6. $factor \rightarrow const$
7. $const \rightarrow integer$

BNF de la même grammaire :

```
<expr>   ::= <term> "+" <expr>
           |  <term>
<term>   ::= <factor> "*" <term>
           |  <factor>
<factor> ::= "(" <expr> ")"
           |  <const>
<const>  ::= integer
```

EBNF Grammaire des nombres décimaux :

1. $expr \rightarrow Sign\ Digits\ Suite$
2. $Sign \rightarrow -$
3. $Sign \rightarrow \Lambda$
4. $Digits \rightarrow Digit\ Digits$
5. $Digits \rightarrow Digit$
6. $Suite \rightarrow .\ Digits$
7. $Suite \rightarrow \Lambda$
8. $Digit \rightarrow 0$
9. $Digit \rightarrow 1$
10. ...

EBNF de la même grammaire :

```
<expr> := '-'? <digit>+ ('.' <digit>+)?
<digit> := '0' | '1' | '2' | '3' | '4'
         | '5' | '6' | '7' | '8' | '9'
```

1.3 Exercice

Donner une représentation EBNF pour la grammaire du langage RAT.

2 Analyse syntaxique et génération de l'arbre abstrait

1. Installer Xtext, s'il n'est pas déjà installer dans votre Eclipse
2. Créer un nouveau projet xtext
   - Project name : rat
   - Language name : rat.Rat
   - extensions : rat

2.1 Xtext et arbre abstrait

Pour écrire la grammaire de notre langage, il faut ouvrir le premier projet parmi ceux qui ont été créés, puis ouvrir le fichier ayant l'extension .xtext (`Rat.xtext`). On constante que le fichier s'ouvre avec une grammaire exemple.

Analyse du fichier créer

- La première ligne :
  `grammar rat.Rat with org.eclipse.xtext.common.Terminals`
  représente la déclaration de notre grammaire nommé `Rat` et le fait qu'elle utilise la grammaire initiale offerte par xtext (`org.eclipse.xtext.common.Terminals`). Cette grammaire déclare pour nous un ensemble d'éléments importants comme les chaînes de caractères (représentées par le terminal ID), les entiers (représentés par le terminal INT), les espaces, les tabulations, les retours chariot,. . ..

— La seconde ligne :
`generate rat "http://www.Rat.rat"`
est importante pour xtext (ne pas la supprimer !) mais nous n'entrerons pas les détails de ce qu'elle représente.
— Puis il y a une ébauche de la façon dont les grammaires doivent être écrites.

```
Model:
   greetings+=Greeting*;

Greeting:
   'Hello' name=ID '!';
```

Cette grammaire permet décrire zéro ou une liste de phrases du genre
`Hello world! Hello Georges!"`
Dans cette grammaire `Model` et `Greeting` représentent des non terminaux. `Model` représente l'axiome de la grammaire puisqu'il s'agit du tout premier terminal, la racine. Si l'on ignore pour l'instant les variable (`greetings` et `name`), on voit que `Model` se dérive en `Greeting*` (zéro ou plusieurs fois le non terminal `Greeting*`) et `Greeting` se dérive en `'Hello' ID '!'`.

La grammaire formelle équivalente est donc :

1. Model → Greeting Model
2. Model → Λ
3. Greeting → 'Hello' id '!'

où une expression régulière est associée au terminal `id`.

L'EBNF équivalent :

```
<Model> := <Greeting>*
<Greeting>:='Hello' <ID> '!'
...
```

Fichiers générés et accesseurs Lorsque Xtext analyse une grammaire, il génère entre autre objets (lexer, parser,...), des classes Java (interfaces et implémentations) correspondant à chaque non terminal. A l'intérieur de ses classes nous disposons de getters et setters correspondant à la structure des règles de production de la grammaire et aux variables.

Soit r une règle de production :
— Si r s'écrit `X:y=ID`
alors Xtext génère une classe `X` contenant un getter et un setter pour la variable `y` de type String (cf terminal ID). De même si on remplace ID par un non terminal quelconque (ex :Z), alors `y` sera de type Z.
— Si r s'écrit `X:(y=Z)?`
alors cette règle signifie que X se dérive zéro ou une fois en Z. Xtext génère une classe X exactement comme au point précédent.
— Si r s'écrit `X:(y+=Z)*`
alors cette règle signifie que X se dérive zéro ou plusieurs fois en Z. Le type de la variable y est une liste de Z. Xtext va générer une classe X contenant le getter de y qui représente une liste de Z.
— Si r s'écrit `X:(y+=Z)+`
comme le point précédent, sauf qu'X se dérive au moins une fois en Z.

2.2 Exercice

1. Modifier le fichier `Rat.xtext` pour que la grammaire soit celle de Rat et que l'arbre abstrait généré ne contienne que les informations nécessaires et ait une forme facilement exploitable (cf cours).
2. Générer les classes : clic droit sur `Rat.xtext` → `Run As` → `Generate Xtext Artifacts`
3. Le fichier `AnalyseSyntaxique.java` contient un programme permettant de charger un fichier, de réaliser son analyse syntaxique et d'afficher les éventuelles erreurs. Importer le fichier dans le projet `rat`, répertoire `src`. Tester le programme principal sur plusieurs fichiers : certains syntaxiquement corrects, d'autres faux.
4. Importer le package attribut (disponible sous Moodle) dans le projet `rat`, répertoire `src`.

3 Visiteurs et analyse sémantique

Lorsque l'analyse syntaxique du fichier est réalisée sans erreur, Xtext nous renvoie l'arbre abstrait (un pointeur sur la racine de l'arbre). Pour réaliser les traitements sémantiques souhaités nous devons parcourir cet arbre.

Pour réaliser le parcours de l'arbre, Xtext fourni un visiteur (`RatSwitch.java`).

3.1 Rappel : le patron de conception visiteur

`http://sourcemaking.com/design_patterns/visitor`

3.2 Gestion des identifiants

Écrire un visiteur (extension de la classe `RatSwitch.java`) qui vérifie la bonne utilisation des identificateurs.

Procéder par étapes :

1. Propager la TDS jusqu'aux déclaration de fonctions et ajouter les identificateurs de fonctions dans la TDS
2. Tester en affichant la TDS
3. Tester en déclarant deux fonctions de même nom → erreur
4. Propager la TDS jusqu'aux paramètres de fonctions et ajouter les identificateurs des paramètres dans la TDS
5. Tester en affichant la TDS
6. Tester en déclarant deux paramètres de même nom pour la même fonction → erreur
7. Tester en déclarant deux paramètres de même nom dans deux fonctions différents → OK
8. Propager la TDS jusqu'aux instructions et ajouter les variables et constantes dans la TDS
9. Tester en affichant la TDS
10. Tester en déclarant deux variables de même nom dans le même bloc → erreur
11. Tester en déclarant deux variables de même nom dans deux blocs différents → OK
12. Tester en affectant une variable déclarée → OK
13. Tester en affectant une variable non déclarée → erreur
14. Tester en modifiant une constante → erreur
15. Tester en modifiant une fonction → erreur
16. Propager la TDS jusqu'aux expressions
17. Tester en utilisant un identifiant déclaré dans un bloc parent → OK
18. Tester en utilisant un identifiant non déclaré → erreur
19. Tester en passant un identificateur de fonction en paramètre d'une opération → erreur
20. Tester en passant un identificateur de variables ou constante en paramètre d'un `call` → erreur

3.3 Typage

Compléter le visiteur précédent pour qu'il vérifie le bon typage du programme Rat.

Comme pour le visiteur de la TDS, procéder par étapes et tester au fur et à mesure.

3.4 Placement mémoire

Compléter le visiteur précédent pour qu'il calcule le placement mémoire des variables. Les informations de placement sont ajoutées à la TDS.

Comme pour le visiteur de la TDS, procéder par étapes et tester au fur et à mesure.

3.5 Génération du code

Compléter le visiteur précédent pour qu'il génère le code TAM à partir d'un code RAT. Vérifier le code généré à l'aide de la machine virtuelle itam.

Comme pour le visiteur de la TDS, procéder par étapes et tester au fur et à mesure.