Projet du cours *Fondements de l'informatique*

École des Mines de Nancy 2A

Frédéric Sur
sur@loria.fr
www.loria.fr/~sur/enseignement/projetFondInfo/

1 Objectif

Le but du projet est d'écrire un interprète pour un nouveau langage de programmation. Cet interprète (un programme Java) sera basé sur le parcours de l'Arbre de Syntaxe Abstrait (AST) associé à un programme dans ce langage. SABLECC génère automatiquement un *parser* permettant de construire l'AST associé à une grammaire hors contexte. Le travail sera donc limité à écrire la grammaire associée au langage de programmation créé, puis à décrire les actions à effectuer lors du parcours de l'AST.

Notre langage sera assez rudimentaire : le seul type de données est le type entier, les structures de base sont les expressions arithmétiques, le langage dispose d'une pile où évaluer ces expressions, il offre la possibilité de définir des variables, et la seule instruction conditionnelle est le `while`.

2 Évaluation du projet

- Travail par groupe de deux.
- Évaluation sur la base d'un rapport écrit. Pour chacune des étapes qui suivent, il faut décrire les modifications apportées, les expériences menées, et les commentaires pertinents que cela vous inspire. Rédigez au fur et à mesure de la progression.
- Date limite pour rendre le rapport, la grammaire finale et les codes Java que vous avez écrits :

  lundi 19 juin 2006.

3 Présentation simplifiée de SABLECC

3.1 Généralités

Un *parser* est un programme qui effectue une analyse syntaxique : il détermine la structure grammaticale d'une entrée par rapport à une grammaire formelle.

Étant donnée une grammaire, SABLECC génère un *parser* en Java, qui transforme l'entrée en un arbre de syntaxe abstraite (AST). En fait, SABLECC génère d'abord un *lexer* qui effectue une analyse lexicale de l'entrée, de manière à identifier des *tokens* (des unités lexicales qui correspondent aux symboles terminaux de la grammaire). Ensuite le *parser* génère l'AST.

Ici, l'entrée sera un fichier ASCII.

3.2 Description de la grammaire

La grammaire est décrite dans un fichier comportant cinq sections (toutes facultatives).

- Package : endroit où sont créés les *lexer* et *parser*.
- Helpers : des raccourcis.
- Tokens : les terminaux, décrits par une expression rationnelle.

– Ignored Tokens : sont ignorés par le *parser*.
– Productions : les règles de production de la grammaire.

3.3 Classes et méthodes générées

Téléchargez l'exemple de grammaire `arithmetic.sable`, et observez les règles de production.

L'exécution de : `sablecc arithmetic.sable` génère dans le répertoire `arithmetic/` quatre classes : `lexer` (le *lexer*), `parser` (le *parser*), `node` (décrivant l'AST), et `analysis` pour le parcours de l'AST.

Regardez de plus près `node` après avoir importé les classes sous ECLIPSE.

Les *tokens* (terminaux) seront des instances des classes `TBlank`, `TDiv`, *etc* héritant de la classe abstraite `Token`.

Les non-terminaux seront des instances des classes `PExpr`, `PFactor`, `PTerm` héritant de la classe abstraite `Node`.

Chacun de ces non-terminaux est étendu selon les alternatives qu'il autorise dans les règles de production de la grammaire. Regardez `ADivFactor`, `AExprTerm`...

À présent, examinez `analysis`. On s'intéresse au parcours en profondeur de l'AST (qui est représenté à l'aide des classes héritant de `node`), implémenté par `DepthFirstAdapter`. Le parcours se fait de manière récursive, en commençant par la racine de l'arbre. `DepthFirstAdapter` a des méthodes `in`, `out` et `case`. La méthode `case` (exemple : `CaseAExprTerm`) appelle la méthode `in` (dans notre exemple : `inAExprTerm`) puis les méthodes `case` correspondant aux alternatives dans la règle de production (toujours dans notre exemple : regardez dans `node` ce qu'appelle `node.getLPar().apply(this)`, et les lignes suivantes), et enfin appelle la méthode `out` (ici : `outAExprTerm(node)`).

En bref, le parcours de l'AST se fait récursivement, la méthode `case` associée à un noeud étant constituée :

1. d'un appel à la méthode `in` en entrant dans un noeud,
2. d'appels aux méthodes `case` sur les fils
3. finalement d'un appel à la méthode `out` (donc lorsque l'on remonte).

Tout le travail consistera à adapter le parcours de l'arbre, en modifiant éventuellement l'une de ces trois méthodes de manière à effectuer certaines actions.

4 Le projet

Vous travaillerez sous ECLIPSE, en important les classes générées par SABLECC à partir de la grammaire. L'analyse lexicale / syntaxique sera effectuée sur un fichier texte que l'on pourra modifier sous ECLIPSE. On pourra passer le nom de ce fichier en argument (onglet `Run` puis choix `Run...`).

Étape 1 (langages réguliers)

Dans cet exercice, on utilisera uniquement le *Lexer* produit par SABLECC de manière à reconnaître différentes expressions rationnelles :

1. les mots de la forme `abQCQab` (où `QCQ` est un mot formé de lettres quelconques, éventuellement vide) ;
2. les mots de la forme `baQCQ(ba)+` ;
3. les mots correspondant à des *noms de variables* (formés d'une lettre, puis d'un certain nombre de lettres ou de chiffres)
4. les commentaires délimités par les balises `/*` et `*/`. On supposera qu'un commentaire ne comprend pas `*/`. Que se passe-t-il si on ne fait pas cette hypothèse, dans le cas concret de l'analyse lexicale d'un fichier correspondant à un programme ?

La page 32 de la thèse d'Étienne Gagnon donne la syntaxe des expressions régulières pour définir les *tokens*.

Le fichier `Rationnel.java` montre comment utiliser le *lexer* produit par SABLECC. Complétez-le en un programme de reconnaissance des mots précédents. Faites des expériences, et en particulier interrogez-vous sur les règles de priorités utilisées pour reconnaître les *tokens*.

Étape 2 (transformation des expressions arithmétiques infixées en expressions postfixées)

Le fichier `arithmetic.sable` correspond à la grammaire des expressions arithmétiques infixées (exemple : `3+5*(4-1)`).

Générez le *parser* à l'aide de SABLECC et importez les fichiers dans un projet Java sous ECLIPSE.

Le fichier `Analyse.java` comprend une classe instanciant le parser, et l'interface avec une classe `Interprete` (dans `Interprete.java`) étendant `DepthFirstAdapter` de manière à afficher la forme postfixée de l'expression entrée (avec l'exemple : `3 5 4 1 - * +`).

Expérimentez, tracez l'AST correspondant à une expression et suivez le parcours implémenté. Les priorités usuelles sont-elles respectées ?

Remarque : ces fichiers sont adaptés de la thèse d'Étienne Gagnon, pp. 25-30.

Étape 3 (évaluation)

Modifiez `Interprete.java` pour permettre d'évaluer une expression arithmétique. On remarquera le lien entre la forme postfixée et l'évaluation de l'expression sur une pile, que l'on pourra implémenter par un tableau d'entiers.

Étape 4 (affichage de la pile)

Modifiez `Interprete.java` pour afficher l'état de la pile à la sortie.

Indication : la méthode `public void outStart(Start node)` est appelée à la sortie du parcours de l'AST.

Étape 5 (évaluations successives)

Modifiez la grammaire pour évaluer plusieurs expressions successives (chacune terminant par un `;`), en laissant le résultat de chaque évaluation dans la pile.

Indication : ajoutez un *token* « point-virgule » et modifiez les règles de production.

On doit pouvoir évaluer :

```
3*(4+2*(3-1));
1+4*19/3+54;
```

Étape 6 (commentaires)

Modifiez la grammaire pour intégrer des commentaires sous la forme `/* commentaires */` à ne pas prendre en compte.

Étape 7 (déclaration de variables)

Modifiez la grammaire pour permettre la déclaration/initialisation de variables par `set nom_var expression;` qui peuvent être utilisées dans une expression. Les noms des variables auront plusieurs caractères.

Indication sur les règles de production : un *programme* sera une collection de *lignes de commandes*, et une ligne de commande sera soit une *déclaration*, soit une *expression*.

On pourra simuler une zone mémoire où sont stockées les valeurs des variables par un (ou plusieurs...) tableau(x)... Il faut bien sûr modifier `Interprete.java` de manière à mettre à jour la mémoire à chaque appel de `set`.

À ce stade, on doit pouvoir interpréter le programme suivant :

```
set titi 3; /* initialisation de titi */
(2+titi)*5;
set toto 1+(3*titi); /* initialisation de toto */
toto+3;
```

et obtenir sur la pile : `25 13`.

Étape 8 (affectation de variables)

Modifiez `Interprete.java` de manière à permettre de changer la valeur de variables existantes par `set`. On pourrait aussi utiliser un autre mot-clé.

Il est donc possible d'interpréter le programme :

```
set x 3;          /* initialisation de x */
x;
set x (2+x)*5;    /* changement de la valeur de x */
x-1;
```

qui doit laisser sur la pile : `3 24`.

Étape 9 (print)

Ajoutez un `print var;` au langage permettant d'afficher la valeur d'une variable `var`.

Étape 10 (while)

Ajoutez une commande `while` au langage. La syntaxe sera `while expr programme;` où `programme` est exécuté tant que l'évaluation de `expr` est non nulle.

Attention à ne pas implémenter un `do programme while expr...`

De manière à permettre l'indentation des programmes, ajoutez la tabulation à la liste des *tokens* ignorés (cherchez le code ASCII correspondant).

Étape 11 (test : factorielle)

Ecrivez un programme (dans notre nouveau langage bien sûr) qui calcule la factorielle d'un entier.

Étape 12 (if)

Comment simuler le comportement du test conditionnel `if then else` dans notre langage, uniquement à l'aide des instructions `while` et `set` ?

Plus précisément, traduisez dans notre langage une séquence comme :

`if (x non nul) then {set y 1;} else {set y 2;}`

où `x` est une variable a valeur entière.

Étape 13 (opérateurs booléens)

De même, en identifiant les entiers non nuls au booléen `true` et l'entier `0` à `false`, comment simuler les opérateurs booléens `and`, `or`, `not` ?

Étape 14 (test : suite de Syracuse)

Soit u_0 un entier. La suite de Syracuse est définie par récurrence : si u_n est pair alors $u_{n+1} = u_n/2$, et si u_n est impair alors $u_{n+1} = 3u_n + 1$. On conjecture que pour toute valeur de u_0 il existe n tel que $u_n = 1$.

Ecrivez un programme qui affiche les termes successifs obtenus avant d'arriver à 1 pour un certain entier, et le nombre d'itérations nécessaires.

Étape 15 (comparaison d'entiers)

Ajoutez un test entre entiers > tel que l'expression `expr1 > expr2` ait pour valeur `1` si le test est vrai et `0` sinon.

Attention à la priorité des opérateurs lorsque l'on modifie la grammaire...

Étape 16 (test : Syracuse, le retour)

Modifiez le programme calculant la suite de Syracuse pour afficher le plus grand entier obtenu lors des itérations.

Étape 17 (vérification de bonne construction)

Ajoutez des vérifications dans `Interprete.java` pour ne pas chercher à évaluer une expression qui contiendrait par exemple une variable non initialisée.

Quelles autres vérifications seraient souhaitables lors de l'interprétation ?

Si vous avez fait (correctement) tout ce qui précède, vous pouvez réfléchir aux exercices suivants. Ils sont plutôt dans un ordre de difficulté croissante ; vous n'êtes pas obligé de les traiter dans cet ordre.

Étape 18 (améliorations diverses - *bonus*)

Ajoutez au langage des instructions `input`, `if`, `for`...

Pour `if` et `for`, il y a deux manières fondamentalement différentes de les implémenter. Lesquelles ?

Étape 19 (type booléen - *bonus*)

Introduisez un type booléen, et les opérateurs associés. Modifiez la syntaxe des instructions conditionnelles en conséquence.

Étape 20 (type décimal - *bonus*)

Idem avec un type décimal. Vous pouvez ajouter quelques fonctions mathématiques de base.

Étape 21 (portée des variables - *bonus*)

En utilisant des délimiteurs de blocs d'instructions (`{...}`), ajoutez au langage la notion de variable locale.

Étape 22 (fonctions - *bonus*)

Voyez-vous comment implémenter des fonctions dans le langage ? (il est peut-être plus simple de ne pas autoriser la récursivité dans un premier temps...)

Faites-le ; la déclaration d'une fonction sera de la forme :

```
def nom_fonc nom_var1 ... nom_varn programme;
```

Comment gérer la récursivité ou les appels croisés de fonctions ?

5 Références complémentaires

[1] Le cours de *Fondements de l'informatique* de J.-Y. Marion.

[2] `http://sablecc.org/`
⟶ Le site officiel de SABLECC, avec des exemples de grammaires. Préférez SABLECC 2 à SABLECC 3.

[3] *SableCC, an object-oriented compiler framework*, Étienne Gagnon, Master Thesis, McGill University, Montreal, 1998.
⟶ Sert *de facto* de documentation... (disponible sur le site de SABLECC)

[4] *Modern Compiler Implentation in Java*, Andrew A. Appel, Cambridge University Press, 2nd edition, 2002.
⟶ Un livre de cours, avec des exemples SABLECC.

[5] `http://www.brainycreatures.org/compiler/sablecc.asp`
⟶ Un tutoriel sur l'écriture d'un compilateur avec SABLECC. On y trouve des informations intéressantes sur les *lexer* et *parser* générés.