Державний вищий навчальний заклад
«Прикарпатський національний університет імені Василя Стефаника»
Фізико-технічний факультет
Кафедра комп’ютерної інженерії та електроніки

Самостійна робота
з курсу «Системне програмування»
на тему:
«Генератор парсерів Grammatica (C#, Java)»

Виконала:
студентка групи КІ-31
Стамбульська Р.Т.

Івано-Франківськ
2020

ЗМІСТ

ВСТУП

Grammatica - це генератор парсерів C#, Java та Visual Basic (компілятор компілятора). Він читає граматичний файл (у форматі EBNF) і створює добре прокоментований та прочитуваний C# або Java вихідний код для аналізатора. Він підтримує граматики LL(k), автоматичне відновлення помилок, читабельні повідомлення про помилки та чітке розділення між граматикою та вихідним кодом, лексеми описуються як регулярні вирази.

1 ТЕРМІНИ ТА ВИЗНАЧЕННЯ

Grammatica використовує певну термінологію, яка важлива для розуміння. Ці ж терміни також використовуються в цій роботі та документах, які пов'язані з нею. У наведеному нижче списку спробуємо визначити та описати найважливіші з цих термінів:

- **Граматика**

  Визначення мови. Grammatica підтримує лише формальні (або штучні) мови, які не мають контексту, тобто там, де всі двозначності у введенні можна вирішити без знання її значення. Граматика визначається набором токенів і продукцій. Правильно побудована граматика додає мові програмування структуру, яка сприяє полегшенню трансляції початкової програми в коректний об'єктний код і виявленню помилок. Дозволяє мові еволюціонувати та збагачуватись новими продукціями для вирішення нових завдань.

- **Токен**

  Група одного або декількох символів, що складаються з імені токену і необов'язкового атрибута. Ім'я токену – це абстрактний символ, який подає

тип лексичної одиниці, наприклад конкретне ключове слово чи послідовність вхідних слів, яка утворює ідентифікатор. Токен є основним складовим елементом у граматиці. Він зазвичай визначається у вигляді стрічки чи регулярного виразу.

- **Регулярний вираз**

  це рядок, що описує або збігається з множиною рядків, відповідно до набору спеціальних синтаксичних правил. Вони використовуються в багатьох текстових редакторах та допоміжних інструментах для пошуку та зміни тексту на основі заданих шаблонів. Багато мов програмування підтримують регулярні вирази для роботи з рядками. Синтаксис регулярних виразів дещо відрізняється між реалізаціями. Регулярні вирази можна використовувати для визначення токенів у граматиці.

- **Продукції**

  Правило граматики для групування та об’єднання лексем разом. Продукції складається з набору взаємовиключних альтернатив, кожна з яких визначає певну послідовність токенів і продукцій, які повинні бути прочитані, щоб відповідати правилу. Продукції та їх альтернативи визначені в EBNF.

- **EBNF (Extended Backus-Naur Form)**

  Синтаксис для визначення продукцій. Синтаксис EBNF трохи відрізняється від реалізацій. EBNF використовується для визначення продукцій у граматиці.

- **Розбір (парсер)**

  Процес читання та аналізу потоку символів. Аналізатор - це програма, яка може читати вхідні символи та структурувати їх відповідно до граматики.

- **Дерево розбору**

Структура даних сформована під час розбору. Кожен токен та відповідність продукції знаходяться у дереві в ієрархічному порядку, в якому вони були оброблені.

- **Синтаксичне дерево**подібне до дерева розбору. Часто термін абстрактне синтаксичне дерево (AST) використовується як синонім для розбору дерева, хоча абстрактне синтаксичне дерево може бути змінене після розбору.

- **LL граматика.**

Перший L стосується зчитування вводу зліва направо. Другий L стосується розбору символів у постановках зліва направо. Граматика LL, як правило, є рекурсивною та має бути проаналізована згори вниз.

- **LR граматика.**

Літера L стосується зчитування вводу зліва направо, а R стосується розбору символів у постановках справа наліво. LR граматика зазвичай ліво-рекурсивна і повинна бути проаналізована знизу вгору.

2 ВСТАНОВЛЕННЯ GRAMMATICA

2.1 Вимоги

Для запуску Grammatica потрібне наступне програмне забезпечення:

Java Runtime, яке сумісне з JDK 1.5.

Щоб використовувати парсери, згенеровані Grammatica, потрібен один із наступних програмних пакетів (залежно від мови, що створюється):

Java Runtime, сумісний з JDK 1.5, для Java-парсерів.

Mono 2.0 (або інший .NET 2.0 сумісний) для парсерів C# або VisualBasic.

2.2 Встановлення

Це програмне забезпечення не вимагає спеціального встановлення. Єдиний необхідний файл - це файл **lib / grammatica-1.6.jar**.

2.3 Запуск з командного рядка

Наступна команда запускає Grammatica і виводить параметри командного рядка:

```
java -jar lib / grammatica-1.6.jar
```

2.4 Запуск від Apache Ant

Grammatica також доступна як завдання Apache Ant. Додайте grammatica-1.6.jar до CLASSPATH та додайте наступні команди, щоб включити завдання Grammatica:

```
<taskdef name = "grammatica"
    classname="net.percederberg.grammatica.ant.GrammaticaTask" />
```

Детальну інформацію про використання <grammatica> можна знайти у посібнику.

2.5 Запуск згенерованих парсерів

Для виконання всіх згенерованих парсерів Grammatica потрібна відповідна бібліотека часу виконання. Бібліотеки часу виконання для різних мов можна знайти у lib підкаталозі:

grammatica-1.6.jar для Java-парсерів
grammatica-1.6.dll для C # або VisualBasic-аналізаторів

3 ОСОБЛИВОСТІ GRAMMATICA

Це короткий опис особливостей, які зараз є у Grammatica.

- Стандартні визначення синтаксису граматики.

Файли граматики визначають лексеми з регулярними виразами, а синтаксис зпростим EBNF. Також можна вказати лексеми як рядки, уникаючи жодних спеціальних символів регулярного виразу.

- Багаторазові граматичні файли.

  У файлах граматики немає вихідного коду аналізатора, що забезпечує більш чітке розділення між граматикою та аналізатором. Це означає, що граматичні файли можуть бути повторно використані іншими програмами.

- Підтримка LL(k) граматики

  Підтримується LL граматика з різноманітною кількістю прогнозованих лексем. Не потрібно зазначати кількість потрібних для перегляду токенів.

- Детальні повідомлення про помилки для згенерованих парсерів

  Повідомлення, створені при виявленні помилки, завжди описують проблему з усіма необхідними деталями. Зокрема, список очікуваних токенів завжди подано.

- Автоматичне виправлення помилок.

  Виправлення помилок – це те, чого завжди намагаються добитись за допомогою згенерованих аналізаторів. Головне, щоб це відбувалось без змін у граматичних файлах. Це робить згенеровані парсери набагато більш зручними для використання.

- Аналіз, за допомогою зворотних зв’язків або синтаксичних дерев.

  Граматику можна проаналізувати за допомогою зворотних зв'язків або шляхом проходу по синтаксичногму дереві. Під час використання зворотних зв’язків можна уникнути створення синтаксичного дерева.

- Генерація парсера для C# та Java

Обидва аналізатори C # та Java можуть бути створені з одного граматичного файлу. Ці аналізатори поводяться однаково і мають однаковий набір функцій, включаючи підтримку регулярних виразів.

- Повна підтримка регулярних виразів та Unicode

  Згенеровані парсери використовують підтримку платформи Java або .NET для Unicode та регулярних виразів для забезпечення оптимальної інтеграції та сумісності.

- Вихідний код

  Створений вихідний код є повністю проситуваним, з правильними відступами та коментуванням.

- Створення парсера часу виконання

  За допомогою бібліотек парсера Grammatica, спеціальний аналізатор може бути створений під час виконання без необхідності генерування вихідного коду. Це можна використовувати для більш простого знаходження та усунення помилок граматики або для інтерактивного створення парсерів.

- Інтеграції з Apache Ant

  Доступні завданняApache Ant, що інтегрують Grammatica зі стандартним інструментом створення для розробників Java.

4 ІНТЕФЕЙС КОМАНДНОГО РЯДКА

Gramatica можна запустити з командного рядка. Доступні параметри командного рядка виводяться під час роботи без будь-яких параметрів, як це видно на прикладі довідки з командного рядка, що виведена з Grammatica:

```
# java -jar grammatica-1.6.jar
 Generates source code for a C#, Java or Visual Basic parser from a grammar file. This program comes with ABSOLUTELY NO WARRANTY;
for details see the LICENSE.txt file.
```

Syntax: Grammatica <grammarfile><action> [<options>]

Actions:

--debug

Debugs the grammar by validating it and printing the internal representation.

--tokenize <file>

Debugs the grammar by using it to tokenize the specified file. No code has to be generated for this.

--parse <file>

Debugs the grammar by using it to parse the specified file. No code has to be generated for this.

--profile <file>

Profiles the grammar by using it to parse the specified file and printing a statistic summary.

--csoutput <dir>

Creates a C# parser for the grammar (in source code).

The specified directory will be used as output directory for the source code files.

--javaoutput <dir>

Creates a Java parser for the grammar (in source code).

The specified directory will be used as the base output directory for the source code files.

C# Output Options:

--csnamespace <package>

Sets the C# namespace to use in generated source code files. By default no namespace declaration is included.

--csclassname <name>

Sets the C# class name prefix to use in generated source code files. By default the grammar file name is used.

--cspublic

Sets public access for all C# types generated. By default type access is internal.

Java Output Options:

--javapackage <package>

Sets the Java package to use in generated source code files. By default no package declaration is included.

--javaclassname <name>

Sets the Java class name prefix to use in generated source code files. By default the grammar file name is used.

--javapublic

Sets public access for all Java types. By default type access is package local.

Visual Basic Output Options:

--vbnamespace <package>

Sets the namespace to use in generated source code files.

By default no namespace declaration is included.

--vbclassname <name>

Sets the class name prefix to use in generated source code files. By default the grammar file name is used.

--vbpublic

Sets public access for all types generated. By default type access is internal.

Error: Missing grammar file and/or action

Упорядкування параметрів командного рядка є важливим. Першим аргументом завжди повинен бути граматичний файл. Другий аргумент завжди повинен бути одним із доступних дій із додатковим параметром файлу чи каталогу. Конкретні параметри виводу генераторів коду є необов'язковими, вибираючи значення за замовчуванням, якщо вони не вказані.

Дозволяється використовуватиGrammatica для розбору та токенізації файлів, навіть якщо не було створено жодного аналізатора. Ця функція може бути дуже корисною для тестування та налагодження граматик, як показано на прикладі нижче.

Командний рядок використання Grammatica для розбору файлу тестових даних:

 # java -jar grammatica-1.6.jar test.grammar --parse test.data

5 ІНТЕРФЕЙС APACHE ANT

Grammatica може бути запущена під Apache Ant. Щоб Apache знайшов завдання Grammatica необхідне просте оголошення завдання. Приклад такого оголошення завдання показаний у прикладі нижче.

Декларація завдання Ant, необхідна для надання grammatica завдання:

```
<taskdef resource = "ant-grammatica.properties"
        classpath = "grammatica-1.6.jar" />
```

5.1 Елемент <Grammatica>

Елемент<Grammatica> дозволяє наступні атрибути:

- grammar

Файли граматики для використання. Цей атрибут необхідний.

- failonerror

Припиняє процес компілювання, якщо виявлена помилка граматики. Значення за замовчуванням - "true"

Граматична обробка визначається в одному або декількох елементах, таких як **<csharp>**,**<java>**або**<validation>**. Принаймні один такий елемент повинен бути присутнім, але допускається також декілька елементів.

5.2 Елемент <csharp>

Підрозділ **<csharp>** контролює генерацію вихідного коду C#. Він дозволяє наступні атрибути:

- dir

Вихідний каталог кінцевого коду. Цей атрибут є необхідним.

- namespace

Простір імен для використання.Використовується для оголошення області дії, яка містить набір пов'язаних об'єктів. Простір імен можна використовувати для організації елементів коду і для створення глобально унікальних типів. Якщо його немає, то декларації простору імен не створюватимуться (за замовчуванням).

- prefix

Клас префікс. За замовчуванням для цього встановлено базове ім'я файлу граматики.

- public

Прапор доступу для публічного типу. Якщо встановлено значення true, всі типи (класи та інтерфейси) матимуть загальнодоступний доступ. Інакше вони матимуть внутрішній доступ. За замовчуванням "false".

5.3 Елемент <java>

< iava > управляє генерацією вихідного коду Java. Він дозволяє наступні атрибути:

- dir

Вихідний каталог кінцевого коду. Цей атрибут є необхідним.

- package

Пакет Java для використання. Якщо його немає, буде використано пакет за замовчуванням (тобто відсутність декларації пакета). Зауважте, що необхідні підкаталоги пакетів будуть створені під **dir** каталогом.

- prefix

Клас названий prefix. За замовчуванням, для нього встановлено базове ім'я граматичного файлу.

- public

Прапор доступу для публічного типу. Якщо встановлено значення true, всі типи (класи та інтерфейси) матимуть загальнодоступний доступ. Інакше вони матимуть захищений пакетом доступ. За замовчуванням "false".

5.4 Елемент <VisualBasic>

Піделемент **<VisualBasic>** управляє генерацією кодуVisual Basic.NET. Він дозволяє наступні атрибути:

- dir

Вихідний каталог кінцевого коду. Цей атрибут є необхідним

- namespace

Простір імен для використання. Якщо його немає, то декларації простору імен не створюватимуться (за замовчуванням).

- prefix

Клас названий prefix. За замовчуванням, для нього встановлено базове ім'я граматичного файлу.

- public

Прапор доступу для публічного типу. Якщо встановлено значення true, всі типи (класи та інтерфейси) матимуть загальнодоступний доступ. Інакше вони матимуть внутрішній доступ. За замовчуванням "false".

5.5 Елемент <validation>

Елемент **<validation>** дозволяє проводити різні тести та перевірки граматичних та/або файлів даних. Він дозволяє наступні атрибути:

- type

Тип перевірки. Цей атрибут необхідний і його потрібно встановити на "налагодження", "токенізація", "розбір" або "профіль".

- inputfile

Файл введення даних для перевірки. Цей атрибут необхідний для всіх, крім типу "налагодження" (який ігнорує цей атрибут).

- quiet

Прапор виходу. Якщо встановлено значення "true" не буде виведено дерево розбору. За замовчуванням "false".

5.6 Приклад використання

Простий приклад використання завдання Grammatica Ant показаний на прикладі нижче.

Частка простого Ant-коду для використання Grammatica створить Java-аналізатор для **simple.grammar**.

```
<taskdef resource = "ant-grammatica.properties"
  classpath = "grammatica-1.6.jar" />
<grammatica grammar = "simple.grammar">
<java dir = "src"
  package = "com.test.simple"
  public = "true" />
</grammatica>
```

6 СТРУКТУРА ГРАМАТИКИ ТА ДЕКЛАРАЦІЙ

Grammatica використовує власний граматичний формат, який, як правило, зберігається у файлах із розширенням **.grammar**. Файли поділяються на три частини - заголовок, токени та продукції. Частина заголовка використовується для

декларацій про граматику, таких як тип граматики. Приклад нижче демонструє структури файлів.

Загальна структура файлу граматики:

```
% header%
GRAMMARTYPE = "LL"
...
% to%
...
% productions%
...
```

Граматичні файли також підтримують коментарі в будь-якому місці файлу. Синтаксис коментарів схожий на коментарі в C# та Java (а також як і у багатьох інших популярних мовах). Два приклади коментарів показано нижче.

Два типи коментарів, доступні у граматичних файлах:

```
/ * A comment spanning
  multiple lines */

// A comment reaching to end of line
```

Заголовна частина граматичного файлу містить визначення параметрів граматики. Це імена-значення пар, при цьому найчастіше це ім'я є великим регістром. Для того, щоб Grammatica обробляв граматичний файл, принаймні необхідно визначити параметр **GRAMMARTYPE**. Нижче наведено список стандартних назв та значень параметрів:

- AUTHOR

Ім'я автора граматики. Використовується Grammatica для створення @author тегів у виході Java.

- CASESENSITIVE

Прапор граматики з урахуванням регістру. Якщо встановлено значення "ні" або "помилково", Grammatica генерує нечутливий до регістру аналізатор, інакше (за замовчуванням) парсери залежать від регістру.

- COPYRIGHT

Граматична заява про авторські права. Використовується Grammatica для коментарів до створених файлів.

- DATE

Дата граматичної версії. Не використовується Grammatica.

- DESCRIPTION

Містить опис граматики. Не використовується Grammatica.

- GRAMMARTYPE

Визначає тип граматики. Дійсні значення "LL" або "LR", але в даний час Grammatica підтримує лише граматики LL.

- LICENSE

Граматична заява про ліцензію. Використовується Grammatica для коментарів до створених файлів.

- VERSION

Номер граматичної версії. Використовується Grammatica для створення @version тегів у виході Java.

7 ТОКЕНИ

Визначення токенів у граматичному файлі складаються з імені токену та зразка токену. Ім'я токена повинно складатися з символів з набору [a-zA-Z0-9_] і не може суперечити жодному іншому імені токену, а також жодному з назв продукцій.

Шаблони токенів можна вказати як рядок, у подвійних лапках ("), або як регулярний вираз, між спеціальними роздільниками (<< і >>). Синтаксис регулярного виразу значною мірою підтримується JDK 1.4, як задокументовано в Java API до **java.util.regexp.Pattern.class.**

Нижче можна побачити варіанти для двох визначень токенів. Перший для дослівних простих стрічок, а інший для регулярних виразів.

```
STRING_TOKEN = "Value"
REGEXP_TOKEN = <<. >>
```

Також можна встановити прапор ігнорування або прапор помилки на визначені токени. Прапор ігнорування використовується для сигналу про те, що токен слід відкинути після читання, тоді як прапор помилки використовується для викликання помилки синтаксичного аналізу кожного разу, коли токен знайдено. Прапор помилки також дозволяє додати певне повідомлення про помилку до помилки розбору, викинутої при виникненні.

Два приклади визначення лексеми з прапорами ігнорування та помилки:

```
WHITESPACE  = << [\ t \ n \ r] + >>% ignore%
UNKNOWN_CHAR = <<. >> error unexpected token%
```

8 ПРОДУКЦІЇ

Продукції граматики визначено в EBNF. Потрібно зауважити, що у граматичних файлах використовується власний "діалект" EBNF, пояснений нижче.

Кожне визначення продукції складається з назви продукції та набору альтернатив продукції. Кожна альтернатива, в свою чергу, містить посилання на

токени або інші продукції. Також приймається самопосилання. Простий приклад визначення продукції показаний на прикладі нижче.

Ця продукція містить дві альтернативи, що ілюструють усі дійсні способи посилання на токен або на продукцію.

```
Prod = "TokenString" OtherProd
     | TOKEN_NAME OtherProd;
```

У наведеному вище прикладі "=" використовується для відділення назви продукції від визначення, а не від стандартного EBNF ": =". Також визначення має закінчуватися знаком ";" характер. Назва продукції дотримується тих же обмежень, що і назва токенів, тобто вона може містити лише символи з набору [a-zA-Z0-9_].

Граматичні файли також дозволяють деякі конструкції, які не входять до стандартних BNF або EBNF. Зокрема, це включає в себе дужки з символами "{", "}", "[" та "]". Звичайно, дозволено також звичайне групування з "(" і ")".

Приклад прийнятих форм групування та повторень. Символи "?", "*" та "+" дозволені після продукцій, лексем або звичайних дужок.

"{...}" є короткою формою для "(...) *", а"[...]" є короткою формою для "(...)?".

```
Prod = "1"?
     | "2" *
     | "3" +
     | {"4"}
     | ["5"];
```

Діалект EBNF, що використовується в Grammatica, наразі не дає **null** (тобто порожніх) альтернатив продукцій. Натомість, той самий результат можна

отримати, зробивши всі посилання на продукції необов’язковими з [...] конструкцією або подібним.

9 ОБРОБЛЕННЯ НЕОДНОЗНАЧНОСТЕЙ

Граматичні неоднозначності - це пункти в граматиці, де суперечать численні альтернативи. Оскільки парсер є детермінованим, слід вибрати єдину альтернативу, тому граматичні двозначності не допускаються. Дивлячись на кілька лексем попереду, аналізатор може вирішити деякі граматичні двозначності. LL-аналізатор в Grammatica підтримує перегляд на багато токенів вперед.

Проста граматична неоднозначність. Ця неоднозначність спричиняє проблеми для парсерів з одним токеном перегляду вперед, але Grammatica може впоратися з цим, вдавшись до двох токенів, що дивляться вперед.

```
Prod = "a" "b"
     | "a" "c" ;
```

На жаль, більшість неоднозначностей не так легко вирішити автоматично, як на прикладі вище. Наприклад, нескінченна кількість токенів, що дивляться вперед, може бути недостатньою, якщо зіткнення може складатися з нескінченним числом лексем. У цих випадках граматику потрібно переписати, щоб усунути неоднозначність. На прикладі нижче продемонстрована така неоднозначність.

Граматична неоднозначність та неоднозначність продукції:

```
OldProd = "a"* "b"
        | "a"* "c" ;

NewProd = "a"* ProdTail ;
```

```
ProdTail = "b"
    | "c" ;
```

Оскільки кількість суперечливих лексем потенційно нескінченна, ця неоднозначність не може бути вирішена за допомогою аналізатора перегляду вперед. Натомість продукція має бути розділена на дві частини, як це проілюстровано черз**NewProd** та **ProdTail**.

Нерозв'язні неоднозначності також можуть виникати через петлі в граматиці, внаслідок чого певна послідовність токенів може повторюватись нескінченно. Деякі неоднозначності також спричинені необов'язковим посиланням, що суперечить наступному посиланню всередині альтернативи продукції. Всі ці типи неоднозначностей виявляються Grammatica та повідомляються користувачеві.

10 ВІДНОВЛЕННЯ ПІСЛЯ ПОМИЛОК

Grammatica підтримує автоматичне відновлення як розбору, так і помилок аналізу. Також вживаються заходи щодо зменшення кількості помилок залежності, тобто помилок, викликаних раніше повідомленими помилками. Однак, автоматичне відновлення не є ідеальним, тому в деяких випадках помилки опускаються або включаються, хоча вони не повинні були бути.

Неочікувані помилки символів (тобто лексичні помилки) обробляються шляхом повідомлення про помилку та ігнорування символу. Аналізатор також увійде в режим відновлення, а це означає, що про подальші помилки не буде повідомлено, поки три токени не будуть прочитані правильно. На практиці це означає, що повідомлятиметься лише про перший символ у помилковій послідовності, хоча всі ігноруються. На малюнку нижче це зображено для граматичного файлу.

$@#!** – це помилкова послідовність символів для граматичного файлу. Якщо ця послідовність зустрічається у граматичному файлі (без коментарів), повідомлятиметься лише про символ "**$". Решта символів також помилкові, але повідомлення про помилки для них будуть придушені.

Неочікувані помилки токену (тобто синтаксичні помилки) обробляються, повідомляючи про помилку та ігноруючи токени, поки не буде знайдений дійсний токен. Аналізатор також переходить у режим відновлення помилок, як пояснено вище. На прикладі нижче показано поведінку відновлення помилки аналізатора. Помилковий знак у продукції граматики:

```
Prod = "one" <<.>> "two" ;
```

Маркер регулярного виразу <<. >> не заборонений у продукціях, і він спричинить помилку. Аналізатор відновиться, ігноруючи всі лексеми до символу ';'.

Про помилки аналізу повідомляється лише до тих пір, поки аналізатор не зіткнувся з лексичною чи синтаксичною помилкою. Насправді, після такої помилки, додаткові звернення до аналізатора взагалі не здійснюються. Усі помилки аналізатора, що виникають під час розбору, повідомляються, але при аналізі наявного дерева розбору деякі фільтрації намагаються уникнути помилок залежності.

Через відновлення помилок слід бути обережним при впровадженні аналізатора. Методи зворотного виклику будуть застосовані, навіть якщо попередні зворотні виклики, можливо, не відбулися. Зокрема, це означає, що дочірні вузли можуть бути відсутніми або впорядкованими, а значення вузлів можуть бути відсутніми, що зробить нульові помилки вказівника більш імовірними. Тому в базовому класі Analyzer були створені допоміжні методи, які виконують необхідну перевірку нуля, і настійно рекомендується їх використовувати.

11 МЕТОДИ

public int getId()

Повертає ідентифікатор токену (шаблону). Це значення встановлюється в якості унікального ідентифікатора при створенні шаблону токенів для спрощення наступної ідентифікації.

public java.lang.String getName()

Повертає ім’я вершини токену.

public java.lang.String getImage()

Повертає зображення токену. Зображення токену складається з вхідних символів, підходять для формування цього токену.

public int getStartLine()

Номер стрічки першого символу в зображенні токена.

public int getStartColumn()

Номер колонки першого символу в зображенні токена.

public int getEndLine()

Номер стрічки останнього символу в зображенні токена.

public int getEndColumn()

Номер колонки останнього символу в зображенні токена.

public Token getPreviousToken()

Повертає токен **previous**. Попередній токен може бути токеном, який був проігнорований в синтаксичному аналізі. Потрібно звернути увагу, якщо функція списку токенів не була використана в токенізаторі, то цей метод завжди буде повертати null. За замовчуванням, функція списку токенів не використовується.

public Token getNextToken()

Повертає наступний токен. Наступний токен може бути токеном, який був проігнорований в синтаксичному аналізі.

public java.lang.String toString()

Повертає детальне стрічкове уявлення про цей токен.

public java.lang.String toShortString()

Повертаєкоротке стрічкове уявлення про цей токен. Стрічка буде містити тільки зображення токену і , можливо, ім’я шаблону токену.

publicvoid reset (java.io.Reader input , Analyzer analyzer)

Скидає цей парсер для використання іншим вхідним потоком. Відповідні токенізатор та аналізатор також будуть скинуті. Цей метод очистить всі внутрішні стани і журнал помилок в синтаксичному аналізаторі. Зазвичай, він викликається для повторного використання пари синтаксичного аналізу і токенізатора з декількома вхідними потоками, що дозволяє запобігти затрат на повторний аналіз граматичних структур.

public Node parse()
throws ParserCreationException,
ParserLogException

Аналізує потік токенів та повертає дерево синтаксичного аналізу. Цей метод викликає **prepare 0**, якщо він ще не викликався. Також викличе метод **reset 0**, щоб переконатись, що тільки метод **Tokenizer.reset()**повинен бути явно викликаний для повторного використання синтаксичного аналізатора для декількох вхідних потоків. У випадку появлення помилки синтаксичного аналізу, аналізатор спробує відновити і скинути всі помилки, що були виявлені в виключеннях журналу синтаксичного аналізу.

public int getId()

Повертає шаблон продукції. Це значення встановлюється в якості унікального ідентифікатору при створенні шаблону для спрощення наступної ідентифікації.

publicstatic final int UNEXPECTED_CHARACTER

Помилка неочікуваного символу. Ця помилка використовується коли був прочитаний символ, який не відповідав дозволеному набору символів в даній позиції.

publicstatic final int UNSUPPORTED_ESCAPE_CHARACTER

Помилка використовується, коли в шаблоні є конструкція escape-символу, але вона не підтримується в цій реалізації.

string PerCederberg.Grammatica.Runtime.Token.GetImage()

Повертає зображення токену. Зображення складається з вхідних символів, що співпадають для формування цього токену.

Token PerCederberg.Grammatica.Runtime.Token.Next

Якщо в токенізаторі використовується функція списку токенів, то всі знайдені токени будуть об’єднанні у список. Наступний токен може бути токеном, який був проігнорований під час синтаксичного аналізу через те, що був встановлений прапор ігнорування. Якщо наступного токену немає чи функція списку токенів не використовувалась в токенізаторі (за замовчуванням), то наступний токен завжди буде дорівнювати нулю.

12 ПРИКЛАДИ

Ці приклади походять з **test/src/grammar** та **test/src/java** каталогів. У цих прикладах вихідний код було спрощено, щоб було легше читати. Можна звернутись до оригінального вихідного коду у вищенаведених каталогах для повних компільованих версій. Також потрібно зауважити, що відповідні приклади C# знаходяться у каталозі **test/src/csharp**.

У нижче наведено повний та робочий приклад граматичного файлу для простої арифметичної мови.

Граматика для простої арифметичної мови.

```
%header%

GRAMMARTYPE = "LL"

DESCRIPTION = "A grammar for a simple arithmetic language."

AUTHOR      = "Per Cederberg, <per at percederberg dot net>"
VERSION     = "1.0"
DATE    = "10 June 2003"

LICENSE     = "Permission is granted to copy this document verbatim in any
  medium, provided that this copyright notice is left intact."

COPYRIGHT = "Copyright (c) 2003 Per Cederberg. All rights reserved."


%tokens%

ADD                 = "+"
SUB                 = "-"
MUL                 = "*"
DIV                 = "/"
LEFT_PAREN              = "("
RIGHT_PAREN             = ")"
NUMBER              = <<[0-9]+>>
IDENTIFIER          = <<[az]>>
WHITESPACE              = <<[ \t\n\r]+>> %ignore%

%productions%

Expression = Term [ExpressionTail] ;

ExpressionTail = "+" Expression
```

```
  | "-" Expression ;
Term = Factor [TermTail] ;

TermTail = "*" Term
  | "/" Term ;

Factor = Atom
  | "(" Expression ")" ;

Atom = NUMBER
  | IDENTIFIER ;
```

Щоб створити аналізатор для вказаної граматики, спочатку потрібно створити вихідний код аналізатора. Якщо граматика вище була збережена у файлі **arithmetic.gramma**, це можна виконати за допомогою наступної команди:

```
# java -jar lib/grammatica-1.6.jar arithmetic.grammar --javaoutput test
```

Це створить файли **ArithmeticAnalyzer.java**, **ArithmeticConstants.java**, **ArithmeticParser.java**, і **ArithmeticTokenizer.java** у тестовому підкаталозі. Ці файли містять вихідний код для аналізатора. Для виклику аналізатора метод на малюнку нижче можна вставити в інший клас Java.

Вихідний код методу, що аналізує арифметичну рядок:

```
private Node parseArithmetic(String input) {
    Parser parser = null;

    parser = new ArithmeticParser(new StringReader(input));
    return parser.parse();
}
```

Що стосується арифметичної мови, цікаво також проаналізувати та оцінити зміст рядка. Це можна зробити, підкласифікуючи клас **ArithmeticAnalyzer** та перевантаживши методи зворотного виклику для відповідних продукцій. Невеликий приклад цього можна побачити нижче.

Частковий (і неповний) вихідний код для аналізатора, що обчислює результат арифметичного виразу:

```
class ArithmeticCalculator extends ArithmeticAnalyzer {

    protected Node exitNumber(Token node) {
    node.addValue(new Integer(node.getImage()));
    return node;
    }

    protected Node exitExpression(Production node) {
    ArrayList values = getChildValues(node);
    Integer     value1;
    Integer     value2;
    String      op;
    int     result;

  if (values.size() == 1) {
    result = ((Integer) values.get(0)).intValue();
    } else {
    value1 = (Integer) values.get(0);
    value2 = (Integer) values.get(2);
    op = (String) values.get(1);
    result = operate(op, value1, value2);
    }
    node.addValue(new Integer(result));
    return node;
    }
```

```
  protected Node exitExpressionTail(Production node) {
  node.addValues(getChildValues(node));
  return node;
  }

  protected Node exitTerm(Production node) {
  ArrayList values = getChildValues(node);
  Integer       value1;
  Integer       value2;
  String        op;
  int     result;

if (values.size() == 1) {
  result = ((Integer) values.get(0)).intValue();
  } else {
  value1 = (Integer) values.get(0);
  value2 = (Integer) values.get(2);
  op = (String) values.get(1);
  result = operate(op, value1, value2);
  }
  node.addValue(new Integer(result));
  return node;
  }

  protected Node exitTermTail(Production node) {
  node.addValues(getChildValues(node));
  return node;
  }

  protected Node exitFactor(Production node) throws ParseException {
  int result;

if (node.getChildCount() == 1) {
   result = getIntValue(getChildAt(node, 0), 0);
```

```
} else {
result = getIntValue(getChildAt(node, 1), 0);
}
node.addValue(new Integer(result));
return node;
}

protected Node exitAtom(Production node) {
node.addValues(getChildValues(node));
return node;
}
```

ВИСНОВКИ

Повний опис Grammatica є на веб-сайті[1]. Grammatica: проста у використанні, добре задокументована, має велику кількість функцій і довідковогоматеріалу, але не так багато прикладів.

Типова граматика Grammatica поділена на три розділи: заголовок, токени та продукції. Легкий код, майже такий ж, як і в ANTLR. Вона також базується на подібному розширеному BNF, хоча формат дещо інший.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Grammatica [Електронний ресурс]. – Режим доступу: https://grammatica.percederberg.

2. Grammatica [Електронний ресурс]. – Режим доступу: https://github.com/cederberg/grammatica

3. Parsing in C# [Електронний ресурс]. – Режим доступу: https://dzone.com/articles/parsing-in-c-all-the-tools-and-libraries-you-can-u-1