Державний вищий навчальний заклад

«Прикарпатський національний університет імені Василя Стефаника»

Фізико-технічний факультет

Кафедра комп’ютерної інженерії та електроніки

Самостійн робота

з курсу “ Системне програмування”

на тему:

«Генератор парсерів CUP (Java)»

Виконав студент

групи КІ-31

Савчук Андрій

Івано-Франківськ

Зміст

I. Про Cup Версія 0.10

Версія 0.10 CUP додає багато нових змін та функцій порівняно з попередніми випусками версії 0.9. Ці зміни намагаються зробити CUP більше схожим на свого попередника YACC. Як результат, старі технічні характеристики 0,9 парсера для CUP не сумісні, і для написання нових специфікацій необхідно буде прочитати додаток C до нового посібника. Нова версія, однак, дає користувачеві більше потужностей та можливостей, що полегшує запис специфікацій аналізатора.

II. Про Cup Версія 0.11

У версії 0.11 команда TUM намагається продовжити історію успіху CUP 0.10, починаючи з впровадження загальних типів даних для нетермінальних символів, а також модернізації інтерфейсу користувача із зручною структурою плагінів ANT.

1. Вступ та приклад

Ця інструкція описує основні операції та використання конструктора користувацьких парсерів на базі Java (tm) (коротко CUP). CUP - це система генерування парсерів LALR з простих специфікацій. Система виконує ту саму роль, що і широко використовувана програма YACC [1], і насправді пропонує більшість можливостей YACC. Однак CUP написаний на Java, використовує специфікації, включаючи вбудований код Java, і виробляє парсери, реалізовані в Java.

Хоча ця інструкція охоплює всі аспекти системи CUP, вона порівняно коротка, і передбачає, що користувач хоч трохи знає про LR-синтаксис. Знання роботи YACC також дуже допомагають зрозуміти, як працюють специфікації CUP. Ряд підручників з побудови компілятора (наприклад, [2,3]) висвітлюють цей матеріал та обговорюють систему YACC (яка досить схожа на цю) як конкретний приклад.

Використання CUP передбачає створення простої специфікації на основі граматики, для якої потрібен аналізатор, а також побудова сканера, здатного розбивати символи на значущі лексеми (наприклад, ключові слова, цифри та спеціальні символи).

Як простий приклад розглянемо систему оцінювання простих арифметичних виразів за цілими числами. Ця система буде читати вирази зі стандартного вводу (кожне закінчується крапкою з комою), оцінювати їх та друкувати результат на стандартному виведенні. Граматика для введення в таку систему виглядає наступним чином:

```
expr_list ::= expr_list expr_part | expr_part
expr_part ::= expr ';'
expr      ::= expr '+' expr | expr '-' expr | expr '*' expr
            | expr '/' expr | expr '%' expr | '(' expr ')'
            | '-' expr | number
```

Щоб задати аналізатор на основі цієї граматики, наш перший крок - визначити та назвати набір символів терміналу, який з’явиться на вході, та набір нетермінальних символів. У цьому випадку нетерміналами є:

expr_list, expr_part and expr .

Для назв терміналів ми можемо вибрати:

SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD, NUMBER, LPAREN, and RPAREN

Досвідчений користувач відзначить проблему з вищезазначеною граматикою. Це неоднозначно. Неоднозначна граматика - це граматика, яка при певному введенні може скоротити частини введення двома різними способами, наприклад, дати дві різні відповіді. Візьмемо, наприклад, вищезгадану граматику з урахуванням наступного вводу:

3 + 4 * 6

Граматика може або оцінити 3 + 4, а потім помножити сім на шість, або вона може оцінити 4 * 6, а потім додати три. Старіші версії CUP змушували користувача писати однозначні граматики, але тепер існує конструкція, що дозволяє користувачеві визначати пріоритети та асоціативи для терміналів. Це означає, що вищевказану неоднозначну граматику можна використовувати після уточнення пріоритетів та асоціативностей. Пізніше є пояснення. На основі цих найменувань я можу побудувати невелику специфікацію CUP таким чином:

```
// Специфікація CUP для простого оцінювача виразів (без дій)

import java_cup.runtime.*;

/* Передумови для налаштування та використання сканера */
init with {: scanner.init();            :};
scan with {: return scanner.next_token(); :};

/* Термінали (токени, повернені сканером).*/
terminal          SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
terminal          UMINUS, LPAREN, RPAREN;
terminal Integer  NUMBER;

/* Нетермінали */
non terminal          expr_list, expr_part;
non terminal Integer  expr, term, factor;

/* Прецеденти */
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE, MOD;
precedence left UMINUS;

/* Граматика */
expr_list ::= expr_list expr_part |
              expr_part;
```

```
expr_part ::= expr SEMI;
expr      ::= expr PLUS expr
          | expr MINUS expr
          | expr TIMES expr
          | expr DIVIDE expr
          | expr MOD expr
             | MINUS expr %prec UMINUS
          | LPAREN expr RPAREN
             | NUMBER
             ;
```

Кожну частину синтаксису специфікації я детально розгляну згодом. Однак, тут можна побачити, що специфікація містить чотири основні частини. У першій частині подано попередні та різні декларації, щоб вказати, як формувати аналізатор та подати частини коду виконання. У цьому випадку вказано, що класи *java_cup.runtime* слід імпортувати, надаючи невеликий біт коду ініціалізації та деякий код для виклику сканера для отримання наступного вводу маркера. Друга частина специфікації декларує термінали та нетермінали та пов'язує об'єктні класи з кожним. У цьому випадку термінали оголошуються як без типу, так і типу *Integer*. Зазначений тип терміналу або нетерміналу - це тип значення цих терміналів або нетерміналів. Якщо тип не вказаний, термінал або нетермінал не несе значення. Тут жоден тип не вказує на те, що ці термінали та нетермінали не мають значення. Третя частина визначає пріоритетність та асоціативність терміналів. Остання декларація пріоритету надає своїм терміналам найвищий пріоритет. Заключна частина специфікації містить граматику.

Для створення аналізатора з цієї специфікації я використаю генератор CUP. Якщо ця специфікація зберігалася у файлі parser.cup, то (як мінімум у системі Unix) я можу викликати CUP за допомогою команди типу:

```
java -jar java-cup-11b.jar parser.cup
```

У цьому випадку система створить два вихідних файли Java, що містять частини згенерованого аналізатора: *sym.java* та *parser.java*. Як я й очікував, ці два файли містять декларації для класів *sym* та *parser*. Клас *sym* містить ряд постійних оголошень, по одному для кожного символу терміналу. Зазвичай сканер використовується для позначення символів (наприклад, з кодом, таким як "повернути новий символ (*sym.SEMI*);"). Клас аналізатора реалізує сам парсер.

Наведена вище специфікація, будуючи повний парсер, не виконує жодних смислових дій *&emdash*; це вказуватиме лише на успіх чи невдачу розбору. Щоб обчислити та надрукувати значення кожного виразу, я повинен вбудувати код Java в аналізатор для виконання дій у різних точках. В CUP дії містяться в рядках коду, які оточені роздільниками форми {: і:}. Загалом система записує всі символи в роздільники, але не намагається перевірити, чи містить він дійсний код Java.

Більш повна специфікація CUP для прикладної системи (з діями, вбудованими в різні точки граматики) наведена нижче:

```
// Специфікація CUP для простого оцінювача виразів (без дій)

import java_cup.runtime.*;

/* Передумови для налаштування та використання сканера */
init with {: scanner.init();            :};
scan with {: return scanner.next_token(); :};

/* Термінали (токени, повернені сканером).*/
terminal          SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
terminal          UMINUS, LPAREN, RPAREN;
terminal Integer   NUMBER;

/* Нетермінали */
non terminal           expr_list, expr_part;
```

```
non terminal Integer    expr;

/* Преценденти */
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE, MOD;
precedence left UMINUS;

/* Граматика */
expr_list ::= expr_list expr_part
              |
          expr_part;

expr_part ::= expr:e
              {: System.out.println("= " + e); :}
          SEMI
              ;

expr      ::= expr:e1 PLUS expr:e2
              {: RESULT = new Integer(e1.intValue() + e2.intValue()); :}
              |
          expr:e1 MINUS expr:e2
          {: RESULT = new Integer(e1.intValue() - e2.intValue()); :}
              |
          expr:e1 TIMES expr:e2
              {: RESULT = new Integer(e1.intValue() * e2.intValue()); :}
              |
          expr:e1 DIVIDE expr:e2
              {: RESULT = new Integer(e1.intValue() / e2.intValue()); :}
              |
          expr:e1 MOD expr:e2
              {: RESULT = new Integer(e1.intValue() % e2.intValue()); :}
              |
          NUMBER:n
```

```
    {: RESULT = n; :}
    |
MINUS expr:e
    {: RESULT = new Integer(0 - e.intValue()); :}
    %prec UMINUS
    |
LPAREN expr:e RPAREN
    {: RESULT = e; :}
    ;
```

Тут можна побачити кілька змін. Найголовніше, що код, який слід виконати в різних точках розбору, міститься у рядках коду, розміщених на {: і:}. Крім того, позначки були розміщені на різних символах у правій частині. Для прикладу:

```
expr:e1 PLUS expr:e2
    {: RESULT = new Integer(e1.intValue() + e2.intValue()); :}
```

Перший нетермінальний *expr* був позначений *e1*, а другий *e2*. Значення лівої сторони кожного зародження завжди неявно позначається як RESULT.

Кожен символ, що з’являється, представлений під час виконання об'єктом типу *Symbol* на стеці розбору. Мітки посилаються на значення змінної екземпляра в цих об'єктах. У виразі *expr: e1* ПЛЮС *expr: e2, e1 і e2* відносяться до об'єктів типу *Integer*. Ці об'єкти знаходяться у полях значень об’єктів типу *Symbol*, що представляють ті нетермінали на стеці розбору. RESULT також є типом *Integer*, оскільки отриманий нетермінальний *expr* був оголошений як тип *Integer*. Цей об'єкт стає змінною екземпляра значення нового об'єкта Symbol.

Для кожної мітки оголошуються ще дві змінні, доступні користувачеві. Ліві та праві мітки значень передаються до рядка коду, щоб користувач міг з'ясувати, де знаходиться ліва і права сторона кожного терміналу чи нетерміналу у вхідному потоці. Назва цих змінних - назва мітки плюс left або

right. Наприклад, з огляду на праву частину зародження *expr: e1 PLUS expr: e2*, користувач міг не лише отримати доступ до змінних *e1* та *e2*, але й *e1left*, *e1right*, *e2left* та *e2right*. Ці змінні мають тип *int*.

Останнім кроком у створенні робочого аналізатора є створення сканера (також відомого як лексичний аналізатор або просто лексера). Ця програма відповідає за читання окремих символів, видалення таких речей, як пробіл та коментарі, розпізнавання термінальних символів із граматики, що представляє кожна група символів, а потім повернення об'єктів *Symbol*, що представляють ці символи парсеру. Термінали будуть отримані з викликом функції сканера. У прикладі аналізатор викличе *scanner.next_token ()*. Сканер повинен повертати об'єкти типу *java_cup.runtime.Symbol*. Цей тип сильно відрізняється від старих версій файлу *java_cup.runtime.symbol* CUP. Ці об'єкти *Symbol* містять значення змінної примірника типу *Object*, яка повинно бути встановлено лексером. Ця змінна відноситься до значення цього символу, а тип об'єкта, який має значення, повинен бути того ж типу, що оголошений у термінальних та нетермінальних деклараціях. У наведеному вище прикладі, якщо лексер хотів передати токен NUMBER, він повинен створити символ із змінною екземпляра значення, заповненою об'єктом типу *Integer*. Об'єкти *Symbol*, що відповідають терміналам і нетерміналам без значення, мають поле нульового значення.

Код, що міститься в *init* з пунктом специфікації, буде виконаний до того, як буде надісланий запит будь-яким токенам. Кожен токен буде запитаний, використовуючи те, що код знайдений в *scan*. Крім цього, точна форма сканера залежить від користувача; проте слід зауважити, що кожен виклик функції сканера повинен повертати новий екземпляр *java_cup.runtime.Symbol* (або підклас). Ці об'єкти символів позначаються інформацією парсера та висуваються на стек. Станом на CUP 0.10j, повторне використання *Symbol* слід виявити, якщо воно відбудеться; аналізатор видасть Помилка, яка dbvfuf’ виправити сканер.

У наступному розділі буде подано більш детальне та формальне пояснення всіх частин специфікації CUP. У розділі 3 описані варіанти роботи

системи CUP. У розділі 4 розглядаються деталі, як налаштувати аналізатор CUP, в той час як розділ 5 розглядає інтерфейс сканера, доданий у CUP 0.10j. У розділі 6 розглядається відновлення помилок. Нарешті, розділ 7 дає висновоки.

2. Синтаксис специфікації

Тепер, коли ми побачили невеликий приклад, ми представляємо повний опис усіх частин специфікації CUP. Специфікація має чотири розділи із загальною кількістю восьми конкретних частин (однак, більшість із них є необов'язковими). Специфікація складається з:

- Специфікації пакета та імпорту;
- назва класу парсера;
- компоненти коду користувача;
- списки символів (термінальні і нетермінальні);
- декларації пріоритету
- граматика

Кожна з цих частин повинна відображатися у наведеному тут порядку. (Повна граматика мови специфікації наведена в Додатку А.) Деталі кожної частини специфікації описані в підрозділах нижче.

2.1 Специфікації пакета та імпорту

Специфікація починається з додаткової декларації про пакет та імпорт. Вони мають той самий синтаксис і відіграють ту саму роль, що і декларації пакету та імпорту, знайдені в звичайній програмі Java. Декларація пакету має наступну форму:

package name;

де ім'я - ідентифікатор пакету Java, можливо, в декількох частинах, розділених ".". Загалом, CUP використовує лексичні умовні положення Java. Так, наприклад, підтримуються обидва стилі коментарів Java, і ідентифікатори будуються, починаючи з літери, знака долара ($) або підкреслення (_), після

чого можуть супроводжуватися нулями або більше літер, цифр, знаків долара та підкреслення.

import package_name.class_name;

чи

import package_name.*;

Декларація пакета вказує, в якому пакеті будуть знаходитися класи *sym* та *parser*, що генеруються системою. Будь-які декларації про імпорт, які з'являються у специфікації, також з'являться у вихідному файлі класу *parser*, що дозволяє безпосередньо використовувати різні імена цього пакету у коді дії, що надається користувачем. Після пакета та імпорту специфікація може містити назву класу парсера таким чином:

class name;

Якщо вводиться ім'я класу, то це ім'я, яке буде використано для генерованого коду аналізатора. Крім того, клас згенерованих символів матиме вказану назву, а потім "*Sym*".

Якщо ім'я класу не вказано, то клас аналізатора називається "*parser*", а клас символів - "*sym*".

2.2 Компоненти коду користувача

Після цих декларацій є ще одна серія необов’язкових декларацій, які дозволяють включити код користувача як частину згенерованого аналізатора (див. Розділ 4 для повного опису того, як аналізатор використовує цей код). Як частина файлу аналізатора створюється окремий непублічний клас, який містить усі вбудовані дії користувача. Перший розділ декларації коду дії дозволяє включити код до цього класу. У цьому розділі зазвичай розміщуються рутини та змінні для використання коду, вбудованого в граматику (типовим прикладом може бути підпрограми маніпулювання таблицею символів). Ця декларація має форму:

action code {: ... :};

де {: ...:} - рядок коду, вміст якого буде розміщено безпосередньо в *action class* декларації класу.

Після *action code* необов'язкове оголошення коду аналізатора. Ця декларація дозволяє розмістити методи та змінну безпосередньо в створеному класі аналізатора. Хоча це рідше, воно може бути корисним при налаштуванні аналізатора &emdash; можливо, наприклад, включити методи сканування всередині аналізатора та / або змінити підпрограми звітування про помилки за замовчуванням. Ця декларація дуже схожа на декларацію *action code* та приймає форму:

parser code {: ... :};

Знову ж таки, код з рядка коду розміщується безпосередньо у створеному визначенні класу парсера. Далі в специфікації - необов'язкова декларація *init*, яка має форму:

init with {: ... :};

Ця декларація надає код, який буде виконаний аналізатором до того, як запитає перший токен. Зазвичай це використовується для ініціалізації сканера, а також різних таблиць та інших структур даних, які можуть знадобитися для семантичних дій. У цьому випадку код, заданий у рядку коду, утворює тіло *void* методу всередині *parser* класу. Заключний (необов’язковий) розділ коду користувача специфікації вказує, як аналізатор повинен запитувати наступний токен від сканера. Він має форму:

scan with {: ... :};

Як і у випадку з *init*, вміст рядка коду утворює тіло методу в створеному аналізаторі. Однак у цьому випадку метод повертає об’єкт типу *java_cup.runtime.Symbol*. Отже, код, знайдений у *scan with*, повинен повертати таке значення.

На CUP 0.10j код дії, код парсера, код *init* та сканування з розділами можуть з’являтися в будь-якому порядку. Однак, вони повинні передувати спискам символів.

2.3 Списки символів

Після коду, що надається користувачем, надходить перша необхідна частина специфікації: список символів. Ці декларації відповідають за найменування та надання типу для кожного термінального та нетермінального символу, який відображається в граматиці. Як зазначено вище, кожен термінальний та нетермінальний символ представлений під час виконання об'єктом Symbol. У разі терміналів вони повертаються сканером і розміщуються на стеці розбору. Лексеру слід помістити значення терміналу в змінну екземпляра значення. Що стосується нетерміналів, вони замінюють ряд об'єктів Symbol на стеці розбору кожного разу, коли розпізнається права частина певного зародження. Для того щоб сказати аналізатору, які типи об'єктів слід використовувати, для яких символів, термінальних та нетермінальних декларацій використовуються. Вони приймають форми:

terminal classname name1, name2, ...;
non terminal classname name1, name2, ...;
terminal name1, name2, ...;

і

non terminal name1, name2, ...;

Де *classname* може бути іменем декількох частин, розділеним на "." *Classname* представляє тип значення цього терміналу або нетерміналу. Під час доступу до цих значень через мітки користувачі використовують оголошений тип. *Classname* може бути будь-якого типу. Якщо не вказано *classname*, то термінал або нетермінал не має значення. Мітка, що посилається на такий символ, має нульове значення. Станом на CUP 0.10j, ви можете вказати нетерміналам декларацію "nonterminal" (зауважте, немає місця), а також оригінальне написання "non terminal".

Назви терміналів і нетерміналів не можуть бути зарезервованими CUP словами; До них відносяться "code", "action", "parser", "terminal", "non", "nonterminal", "init", "scan", "with", "start", "precedence", "left", "right", "nonassoc", "import" та "package".

2.4 Декларації про пріоритети та асоціативність

Третій розділ, який не є обов'язковим, вказує на пріоритети та асоціативність терміналів. Це корисно для розбору неоднозначних граматик, як це зроблено у прикладі вище. Існує три типи декларацій пріоритетності / асоціативності:

precedence left terminal[, terminal...];
precedence right terminal[, terminal...];
precedence nonassoc terminal[, terminal...];

Список, розділений комами, вказує на те, що ці термінали повинні мати асоціативність, вказану на цьому рівні пріоритету, і пріоритет цієї декларації. Порядок пріоритету - від найвищого до нижнього - знизу вгору. Отже, це вказує, що множення і ділення мають вищий пріоритет, ніж додавання і віднімання:

precedence left ADD, SUBTRACT;
precedence left TIMES, DIVIDE;

Пріоритетність вирішує проблеми зменшення порушень. Наприклад, з огляду на вхід до вищевказаного прикладу аналізатора 3 + 4 * 8, аналізатор не знає, чи слід зменшувати 3 + 4 або переміщувати "*" на стек. Однак, оскільки '*' має більший пріоритет, ніж '+', він буде зміщений і множення буде виконано перед додаванням.

CUP присвоює кожному своєму терміналу пріоритет відповідно до цих декларацій. Будь-які термінали, які не зазначені в цій декларації, мають найнижчий пріоритет. CUP також присвоює кожному своєму зародженню пріоритет. Цей пріоритет дорівнює перевазі останнього терміналу в цьому зародженні. Якщо немає терміналів, то це має найнижчий пріоритет. Наприклад, expr :: = *expr TIMES expr* матиме такий саме пріоритет, що і TIMES.

Коли виникає конфлікт зсуву / зменшення, аналізатор визначає, чи має термінал, який слід змістити, більш високий пріоритет. Якщо термінал має більший пріоритет, він зміщується, якщо зародження має більший пріоритет, проводиться зменшення. Якщо вони мають однаковий пріоритет, асоціативність терміналу визначить, що станеться.

Асоціативність присвоюється кожному терміналу, який використовується в деклараціях про пріоритетність / асоціативність. Три асоціативи - *left, right і nonassoc.* Асоціативи також використовуються для вирішення змін / зменшення конфліктів, але лише у випадку рівних пріоритетів. Якщо асоціативність терміналу, що може бути зміщена, залишена, то проводиться зменшення. Це означає, що якщо вхід - це рядок доповнень, наприклад 3 + 4 + 5 + 6 + 7, то аналізатор завжди зменшить їх зліва направо, у цьому випадку, починаючи з 3 + 4. Якщо асоціативність терміналу правильна, він зміщується на стек. Отже, скорочення відбуватимуться справа наліво. Отже, якби ПЛЮС було оголошено асоціативністю right, 6 + 7 було б зменшено спочатку у вищевказаному рядку. Якщо термінал оголошений як nonassoc, то два послідовних випадки, що мають неасоціативні термінали з рівним пріоритетом, генерують помилку. Це корисно для порівняльних операцій. Наприклад, якщо рядок введення дорівнює 6 == 7 == 8 == 9, аналізатор повинен створити помилку. Якщо '==' оголошено як nonassoc, буде створена помилка.

Всі термінали, які не використовуються в деклараціях про пріоритетність / асоціативність, трактуються як найнижчий пріоритет. Якщо результати зсуву / зменшення помилок, що включають два таких термінали, вирішити не вдасться, то про це буде повідомлено.

2.5 Граматика

Заключний розділ декларації CUP містить граматику. Цей розділ необов'язково починається з декларації форми:

start with non-terminal;

Це вказує, який нетермінал є нетерміналом початку або мети для розбору. Якщо стартовий нетермінал явно не оголошений, то буде використаний нетермінал з лівого боку першого входження. В кінці успішного розбору CUP повертає об’єкт типу *java_cup.runtime.Symbol*. Ця *Symbol* значень містить кінцевий результат зменшення.

Сама граматика слідує за необов'язковим початковим оголошенням. Кожне входження в граматиці має нетермінальний лівий бік з символом ":: =", за яким слідує серія нульових або більше дій, термінальних або нетермінальних символів з подальшим необов'язковим контекстуальним пріоритетом призначення та закінчується крапкою з комою (;).

2.5.1 Символьні мітки та місця

Кожен символ праворуч може бути додатково позначений іменем. Імена міток з’являються після назви символу, розділеного двокрапкою (:). Назви мітки повинні бути унікальними і можуть використовуватися в коді дії для позначення значення символу. Поряд з міткою створюються ще дві змінні, які є мітка плюс left та мітка плюс right. Це значення int, які містять праве та ліве розташування того, що охоплює термінал або нетермінал у вхідному файлі. Ці значення повинні бути належним чином ініціалізовані в терміналах лексером. Ліві та праві значення потім поширюються на нетермінали.

Якщо є кілька постановок для одного і того ж нетермінального, вони можуть бути оголошені разом. У цьому випадку постановки починаються з нетермінальних і ":: =". За цим слідує ланцюг правої сторони, кожен з яких розділений смужкою (|). Потім повний набір закінчується крапкою з комою.

Однак, самі цілі числа можуть бути недостатньо хорошими для розміщення символів у вхідному файлі. Ось чому CUP також підтримує більш складну інформацію про місцезнаходження у вигляді об’єктів *Location*. Ці об'єкти потрібно створити компонентом сканера. Вони стають доступними в межах дій, якщо генератор парсера отримує правильний параметр командного рядка. Вони доступні через *??xleft і ??xright* з *??*, будучи міткою символу.

2.5.2 Дії користувача

Дії відображаються в правій частині як рядки коду (наприклад, код Java всередині {: ...:} роздільники). Вони виконуються парсером у тому місці, коли частина зліва від дії була розпізнана. (Зверніть увагу, що сканер поверне маркер за минулу точку дії, оскільки аналізатору потрібен цей додатковий маркер пошуку для розпізнавання.)

2.5.3 Контекстуальний пріоритет

Контекстуальне призначення пріоритету слідувати символам та діям правої частини, пріоритет якого він присвоює. Контекстуальне призначення пріоритету дозволяє присвоювати пріоритет, не заснований на останньому терміналі в ньому. Хороший приклад показаний у наведеній вище специфікації аналізатора вибірки:

```
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE, MOD;
precedence left UMINUS;

expr ::=  MINUS expr:e
        {: RESULT = new Integer(0 - e.intValue()); :}
        %prec UMINUS
```

Тут виробництво оголошено таким, що має перевагу UMINUS. Отже, аналізатор може дати знаку MINUS два різні пріоритети, залежно від того, чи це одинарний мінус чи операція віднімання.

3. Запуск CUP

3.1 Інтерфейс командного рядка

Як було сказано вище, CUP написаний на Java. Щоб викликати його, потрібно використовувати інтерпретатор Java для виклику статичного методу *java_cup.Main ()*, передаючи масив рядків, що містять параметри. Якщо

припустити машину Unix, найпростіший спосіб зробити це, як правило, викликати її безпосередньо з командного рядка з такою командою, як:

java -jar java-cup-11b.jar options inputfile

Після запуску, CUP розраховує знайти специфікаційний файл на стандартному вході та видає два вихідних файли Java як вихід. Крім файлу специфікацій, поведінку CUP також можна змінити, передавши на нього різні параметри. Правові варіанти задокументовані на *Main.java* і включають:

-**package name**

Парсери та символи класів мають бути розміщені у названому пакеті. За замовчуванням в генерований код не вводиться специфікація пакету (звідси класи за замовчуванням для спеціального пакету "без імені").

-**parser** name

Виводить парсер і код дії у файл (і клас) із заданим іменем, а не за замовчуванням "аналізатор".

-**symbols** name

Виводить константу код символу в клас із заданим іменем замість типового значення "sym".

-interface

Виводить код постійного символу як *interface*, а не як *class*.

-nonterms

Помістити константи для нетерміналів у постійний клас символів. Парсеру не потрібні ці константи символів, тому вони зазвичай не виводяться. Однак, це може бути дуже корисно посилатися на ці константи при налагодженні генерованого аналізатора.

-**expect** number

Під час побудови парсера система може виявити, що сталась неоднозначна ситуація під час виконання. Це називається конфліктом. Взагалі аналізатор може не мати можливості вирішити, чи змістити (прочитати інший символ) чи зменшити (замінити розпізнану праву частину лівою частиною). Це називається

конфліктом зміщення / зменшення. Аналогічно, аналізатор може не мати можливості вирішити між скороченням з двома різними постановками. Це називається конфліктом зменшення / зменшення. Зазвичай, якщо виникає один або кілька цих конфліктів, генерація парсера припиняється. Однак у певних ретельно розглянутих випадках може бути вигідно довільно розірвати такий конфлікт. У цьому випадку CUP використовує конвенцію YACC і вирішує зміщення / зменшення конфліктів шляхом зміщення, а також зменшення / зменшення конфлікти, використовуючи виробництво "найвищого пріоритету" (таке, яке було оголошено першим у специфікації). Щоб ввімкнути автоматичний розрив конфліктів, слід вказати опцію -expect, яка вказує, скільки саме конфліктів очікується. Про конфлікти, вирішені пріоритетами та асоціативами, не повідомляється.

-nowarn

Цей параметр спричиняє заборону всіх попереджувальних повідомлень (на відміну від повідомлень про помилки), вироблених системою.

-nosummary

Зазвичай система друкує підсумок, в якому перераховуються такі речі, як кількість терміналів, нетерміналів, стан розбору тощо, в кінці його запуску. Цей параметр пригнічує цей звіт.

-progress

Цей параметр призводить до того, що система друкує короткі повідомлення, що свідчить про прогрес через різні частини процесу генерації парсера.

-dump_grammar

-dump_states

-dump_tables

-dump

Ці параметри призводять до того, що система створює читаний дамп граматики, побудований стан розбору (часто необхідний для вирішення конфліктів

розбору) та таблиці розбору (рідко потрібні) відповідно. Параметр -dump може використовуватися для отримання всіх цих видів.

-time

Цей параметр додає детальну статистику хронометражу до звичайного підсумків результатів. Це, як правило, представляє великий інтерес лише для самих підтримуючих систему.

-debug

Ця опція створює об'ємну внутрішню налагоджувальну інформацію про систему під час її роботи. Це, як правило, цікавить лише тих, хто підтримує систему.

-nopositions

Цей параметр утримує CUP від генерації коду для розповсюдження лівих та правих значень терміналів до нетерміналів, а потім від нетерміналів до інших терміналів. Якщо ліві та праві значення не будуть використовуватися аналізатором, то це збереже деякі обчислення часу виконання, щоб не генерувати ці розповсюдження позицій. Цей параметр також запобігає генеруванню лівої та правої змінних міток, тому будь-яке посилання на них призведе до помилки.

-locations

Цей параметр змушує CUP генерувати *xleft / xright* для доступу до об'єктів *Location* для початку / кінця символу всередині дій.

-**genericlabels**

Цей параметр іде ще на крок далі від -xmlactions, створюючи дерево повного розбору як дерево XMLElement.

-**noscanner**

CUP 0.10j представив покращену інтеграцію сканера та новий інтерфейс, *java_cup.runtime.Scanner*. За замовчуванням згенерований аналізатор

посилається на цей інтерфейс, а це означає, що ви не можете використовувати ці парсери із часом виконання CUP старше 0,10j. Якщо ваш аналізатор не використовує нові функції інтеграції сканера, ви можете вказати параметр -noscanner для придушення посилань на *java_cup.runtime.Scanner* і дозволити сумісність зі старими програмами. Не багато людей матимуть для цього підстави.

-version

Викликання CUP із прапорцем -version призведе до роздрукування робочої версії CUP та зупинки. Це дозволяє автоматичну перевірку версій CUP для *Makefiles*, встановлювати сценарії та інші програми, які цього можуть вимагати.

3.2 Інтеграція CUP у сценарій ANT

Щоб використовувати cup в ANT-скрипті, вам слід додати наступне визначення завдання у файл build.xml:

```
<taskdef name="cup"
  classname="java_cup.anttask.CUPTask"
  classpathref="cupclasspath"
/>
```

Тепер я готовий використовувати <cup /> для створення власних аналізаторів з ANT. Це може виглядати так:

```
<target name="cup">
<cup srcfile="path/to/cupfile/Parser.cup"
  destdir="path/to/javafiles"
  interface="true"
/>
</target>
```

4. Налаштування аналізатора

Кожен згенерований аналізатор складається з трьох створених класів. Клас *sym* (який можна перейменувати за допомогою параметра -*symbols*) просто містить ряд констант *int*, по одному для кожного терміналу. Також не включені

термінали, якщо вказано параметр *-notterms*. Вихідний файл для класу парсера (який можна перейменувати за допомогою параметра *-parser*) насправді містить два визначення класу, загальнодоступний клас *parser*, який реалізує фактичний аналізатор, та інший непублічний клас (званий CUP$action), який інкапсулює всі користувацькі дії, що містяться в граматиці, а також код з декларації *action code*. На додаток до наданого користувачем коду, цей клас містить один метод: CUP$do_action, який складається з великого оператора перемикання для вибору та виконання різних фрагментів коду дії, що надається користувачем. Загалом, всі імена, що починаються з префіксу CUP $, зарезервовані для внутрішнього використання кодом, створеним CUP.

Клас аналізатора містить фактично генерований аналізатор. Це підклас *java_cup.runtime.lr_parser*, який реалізує загальну таблицю, керовану основою для парсерів LR. Сформований клас парсера містить низку таблиць для використання загальними рамками. Три таблиці надаються:

the production table

надає номер символу лівого боку, який не є кінцевим, разом з довжиною правої частини для кожного випуску граматики,

the action table

вказує, які дії (зсув, зменшення чи помилка) слід вжити для кожного символу пошуку, якщо виникає у кожному стані,

the reduce-goto table

вказує, у який стан перейти пісдя зменшення (під кожним нетермінальним з кожного стану).

Поза таблицями розбору, генерований (або успадкований) код надає низку методів, які можна використовувати для налаштування створеного аналізатора. Деякі з цих методів надаються кодом, який знаходиться в частині специфікації і можуть бути налаштовані безпосередньо таким чином. Інші надаються базовим класом *lr_parser* і можуть бути замінені новими версіями

(через декларацію коду парсера) для налаштування системи. Доступні способи налаштування включають:

public void user_init()

Цей метод викликає аналізатор перед тим, як запитати перший токен від сканера. Тіло цього методу містить код з *init* із пунктом специфікації.

public java_cup.runtime.Symbol scan()

Цей метод інкапсулює сканер і викликається кожного разу, коли парсер потребує нового терміналу. Тіло цього способу забезпечується скануванням із застереженням специфікації, якщо воно є; в іншому випадку він повертає *getScanner (). next_token ().*

public java_cup.runtime.Scanner getScanner()

Повертає сканер за замовчуванням (розділ 5).

public void setScanner(java_cup.runtime.Scanner s)

Встановлює сканер за замовчуванням (розділ 5).

public void report_error(String message, Object info)

Цей метод слід викликати щоразу, коли має бути видано повідомлення про помилку. У реалізації цього методу за замовчуванням перший параметр надає текст повідомлення, який друкується на *System.err*, а другий параметр просто ігнорується. Дуже характерно переосмислити цей метод, щоб забезпечити більш досконалий механізм повідомлення про помилки.

public List expected_token_ids()

Цей метод інформує під час синтаксичного розбору, особливо при обробці проблем розбору та недійсного введення даних про дійсні продовження розбору. Він повертає список цілих констант, які відповідають константам в згенерованому *sym.java*. Ви можете повторно перекласти ідентифікатори до їх рядкових представлень, які там використовується у специфікації граматики для декларування (не) імен терміналів, з *symbol_name_from_id(id)***.** Це зазвичай може використовуватися для реалізації завершення синтаксису або більш детальних повідомлень про помилки.

public String symbol_name_from_id(int id)

Цей метод переводить числові константи, що представляють ідентифікатори символів назад до їх текстуального подання, оголошеного у файлі специфікації *.cup parser.*

public void syntax_error(Symbol cur_token)

Цей метод викликає аналізатор, як тільки виявляється синтаксична помилка (але перед спробою відновлення помилки). У реалізації за замовчуванням він викликає: report_error ("Помилка синтаксису", null) ;

public void unrecovered_syntax_error(Symbol cur_token)

Цей метод викликається аналізатором, якщо він не в змозі відновити синтаксичну помилку. У реалізації за замовчуванням він викликає: report_fatal_error ("Не вдалося відновити та продовжити розбір", null) ;.

protected int error_sync_size()

Цей метод викликає аналізатор для визначення кількості токенів, які він повинен успішно проаналізувати, щоб визнати відновлення помилок успішним. Реалізація за замовчуванням повертає 3. Значення нижче 2 не рекомендуються. Деталі див. У розділі про відновлення помилок.

Сам розбір виконується методом *public Symbol parse ().* Цей метод починається з отримання посилань на кожну таблицю розбору, потім ініціалізує об'єкт *CUP$action* (викликаючи *protected void init_action ()).* Далі він викликає *user_init (),* потім отримує перший токен пошуку з викликом *scan ().* Нарешті, починається розбір. Розбір триває до виклику *done_parsing ()* (це робиться автоматично, наприклад, коли аналізатор приймає). Потім він повертає *Symbol* із змінною екземпляра значення, що містить *REZULT* стартового виробництва, або *null*, якщо немає значення.

Крім звичайного аналізатора, система виконання також забезпечує налагоджувальну версію аналізатора. Це працює точно так само, як і звичайний

аналізатор, але друкує повідомлення про налагодження (виклику *public void debug_message (String mess)*, реалізація якого за замовчуванням друкує повідомлення на *System.err*).

Виходячи з цих процедур, виклик аналізатора CUP зазвичай виконується з кодом, таким як:

```
/* створити об'єкт розбору*/
parser parser_obj = new parser();

/* тут відкрити вхідні файли тощо */
Symbol parse_tree = null;

try {
  if (do_debug_parse)
    parse_tree = parser_obj.debug_parse();
  else
    parse_tree = parser_obj.parse();
} catch (Exception e) {
  /* зробити очистку - - можливо перекинути е */
} finally {
      /* виконати закриття*/
}
```

5. Інтерфейс сканера

5.1 Основне управління символами

У CUP 0.10j інтеграція сканерів була покращена відповідно до пропозицій Девіда Макмахона. Ці зміни полегшують включення JLex та інших автоматично генерованих сканерів у парсер CUP.

Щоб використовувати новий код, сканер повинен реалізувати інтерфейс *java_cup.runtime.Scanner*, він визначений як:

```
package java_cup.runtime;
```

```
public interface Scanner {
   public Symbol next_token() throws java.lang.Exception;
}
```

Окрім методів, описаних у розділі 4, клас *java_cup.runtime.lr_parser* має два нові методи доступу, *setScanner ()* та *getScanner ()*. Типовою реалізацією *scan ()* є:

```
 public Symbol scan() throws java.lang.Exception {
   return getScanner().next_token();
 }
```

Згенерований аналізатор також містить конструктор, який бере *Scanner* і викликає *setScanner ()* з ним. У більшості випадків *init with* і *scan with* директиви можуть бути опущені. Ви можете просто створити аналізатор із посиланням на потрібний сканер:

```
    /* create a parsing object */
    parser parser_obj = new parser(new my_scanner());
```

або встановити сканер після створення аналізатора:

```
    /* create a parsing object */
    parser parser_obj = new parser();
    /* set the default scanner */
    parser_obj.setScanner(new my_scanner());
```

Зауважте, що оскільки аналізатор використовує препереглядd, скидання сканера посеред розбору не рекомендується. Якщо ви спробуєте використати програму *scan ()* за замовчуванням без попереднього виклику *setScanner ()*, буде викинуто *NullPointerException*.

Як приклад інтеграції сканера, наступні три рядки на вході генератора лексерів - це все, що потрібно для використання сканера JLex з CUP:

```
%implements java_cup.runtime.Scanner
%function next_token
%type java_cup.runtime.Symbol
```

Очікується, що директива JLex %cup скоротить вищевказані три директиви в наступній версії JLex. Викликати аналізатор зі сканером JLex просто:

parser parser_obj = new parser(new Yylex(some_InputStream_or_Reader));

Зверніть увагу, що ви все ще повинні правильно поводитися з EOF; код JLex для цього є чимось на зразок:

```
%eofval{
  return sym.EOF;
%eofval}
```

де *sym* - це назва класу символів для створеного аналізатора.

Приклад *simple_calc* в дистрибутиві CUP ілюструє використання функцій інтеграції сканера з кодованим вручну сканером.

5.2 Розширене управління символами

CUP v11a пропонує можливість вдосконаленої обробки символів у CUP. Таким чином, можна реалізувати свій власний *SymbolFactory*, що походить від *java_cup.runtime.SymbolFactory*, і CUP керувати власним типом символів. Це вже зроблено заздалегідь визначеним *java_cup.runtime.ComplexSymbolFactory*, який забезпечує підтримку детальної інформації про місцезнаходження у класі символів. *Lexer.jflex CUP* вже використовує нову функцію.

Все, що потрібно зробити - це забезпечити свій аналізатор, створений CUP, новим *SymbolFactory*, виглядає наступним чином:

```
SymbolFactory symbolFactory = new ComplexSymbolFactory();
MyParser parser = new MyParser(new
Lexer(inputfile,symbolFactory),symbolFactory);
```

ComplexSymbolFactory використовується прямо: у своєму лексемі, замість того, щоб безпосередньо створювати *Symbol-Objects*, викликаються методи створення, які називаються *newSymbol (...).*

У більшості випадків потрібно буде використовувати саме *ComplexSymbolFactory*. Можна **зробити це,** додавши зручні методи в тіло свого парсера, як-от наступний для стмволів із доданим значенням *String*:

```
public Symbol symbol(String plainname, int terminalcode, String lexem){
   return symbolFactory.newSymbol(plainname, terminalcode, new
Location(yyline+1, yycolumn +1), new Location(yyline+1,yycolumn+yylength()),
lexem);
 }
```

Об'єкти *Location* надають зручний спосіб управління детальною інформацією про положення символів. Щоб використовувати їх у своїх специфікаціях дій, не забудьте встановити параметр в командному рядку -*locations* під час запуску CUP! Це генерує непрості цілі значення. Потім можна отримати доступ до всієї інформації так:

```
expr     ::= expr:e1 PLUS expr:e2
             {: RESULT = BinExpr.createAdd(e1,e2);          :}
             | NUMBER:n
             {: RESULT = Expr.createConst(nxleft,n,nxright); :}
```

У прикладі я створюю дерево виразів з нашої мови введення. Тут я зберігаю інформацію про місцеположення для підвиразів в *AST*. Це можна побачити в дії для *NUMBER*, де ми отримуємо доступ до *nxleft* та *nxright*. Зауважу, що в цьому прикладі ми зберігаємо лише місця розташування безпосередньо. У зручній реалізації *AST*, ми очікуємо рекурсивну функцію, яка обчислює місця розташування для кожного посередницького вузла на дереві.

5.3 Кешування потоку символів

CUP пропонує додатковий шар між сканером і аналізатором, щоб кешувати фактичну послідовність символів терміналу, надану сканером для можливого подальшого використання.

Зважаючи на те, що клас *Lexer* реалізує інтерфейс *Scanner* CUP, ми можемо захистити його вихід за допомогою *ScannerBuffer* і отримати доступ до токена потоку за допомогою *getBuffered ()*, коли розбір буде закінчений:

```
Scanner scanner = new Lexer();
ScannerBuffer buffer = new ScannerBuffer(lexer);
Parser p = new Parser(buffer,new ComplexSymbolFactory);
```

```
p.parse();
System.out.println(buffer.getBuffered());
```

6. Відновлення помилок

6.1 Символ помилки

Кінцевим важливим аспектом побудови парсерів за допомогою CUP є підтримка відновлення синтаксичних помилок. CUP використовує ті самі механізми відновлення помилок, що і YACC. Зокрема, він підтримує спеціальний символ помилки (позначається просто як *error*). Цей символ відіграє роль спеціального нетермінального, який замість того, щоб визначатись виробництвом, натомість відповідає помилковій послідовності введення.

Символ помилки вступає в дію лише у випадку виявлення синтаксичної помилки. Якщо виявлена синтаксична помилка, то аналізатор намагається замінити деяку частину потоку вхідних токенів *error*, а потім продовжити розбір. Наприклад, у нас можуть бути такі постановки, як:

```
stmt ::= expr SEMI | while_stmt SEMI | if_stmt SEMI | ... |
         error SEMI
         ;
```

Це вказує на те, що якщо жодна з нормальних продукці для *stmt* не може бути узгоджена за допомогою введення, тоді слід оголосити синтаксичну помилку, а відновлення слід пропустити помилкові токени (еквівалентні відповідності та заміни *error*) аж до точки в якій синтаксичний розбір може бути продовжений крапкою з комою. Помилка вважається відновленою за умови, і лише в тому випадку, якщо достатня кількість токенів, що минули символ помилки, можна успішно проаналізувати. (Кількість необхідних лексем визначається методом *error_sync_size ()* аналізатора і за замовчуванням до 3).

Зокрема, аналізатор спочатку шукає найближчий стан до вершини стека розбору, який має вихідний перехід під *error*. Це, як правило, відповідає роботі від виробництв, які представляють більш детальні конструкції (наприклад,

конкретний вид твердження), аж до виробництв, які представляють більш загальні або вкладені конструкції (наприклад, загальне виробництво для всіх тверджень або виробництво, що представляє цілий розділ декларацій) поки ми не потрапимо до місця, де передбачено виробництво відновлення помилок.

Після того, як аналізатор буде розміщений у конфігурації, яка має негайне відновлення помилок (переміщаючи стек до першого такого стану), аналізатор починає пропускати токени, щоб знайти точку, в якій синтаксичний аналіз можна продовжити. Після відкидання кожного токена, аналізатор намагається проаналізувати наперед вхід (не виконуючи жодних вбудованих смислових дій). Якщо аналізатор може успішно проаналізувати необхідну кількість токенів, то введене резервне копіювання до точки відновлення і розбір буде відновлено нормально (виконуючи всі дії). Якщо розбір не може бути продовжений досить далеко, то інший токен відкидається, і аналізатор знову намагається проаналізувати. Якщо кінець введення досягнуто без успішного відновлення (або не було знайдено відповідного стану відновлення помилок у стеці розбору для початку), то відновлення помилок не вдається.

6.2 Продовження роботи

У разі помилок розбору, CUP пропонує підтримку в наданні змістовних відгуків користувачам. Вважайте, що зазначений аналізатор повинен застосовуватися в IDE, який реєструє помилки розбору та робить пропозиції щодо того, як виправити ці помилки. У цих випадках реалізація коду дії символу помилки може полягати в повідомленні IDE про точне розташування невідповіного вводу, а також переліку символів, за допомогою яких аналізатор може просунутися у поточному стані , досягнувши стану, що приймає. Ці символи кодуються як цілі числа, і їх можна розшифрувати до їх рядкових представлень за допомогою викликів *symbol_name_from_id ()*. Така дія може виглядати так:

```
stmt ::= expr SEMI | while_stmt SEMI | if_stmt SEMI | ... |
          error:e {:
         List expected = expected_token_ids();
```

```
        myIDE.registerParseError(exleft,exright,expected);
    :} SEMI
        ;
```

7. Висновки

Ця інструкція коротко описала систему генерації парсера CUP LALR. CUP призначений для виконання тієї ж ролі, що і добре відома система генераторів парсера YACC, але написана та працює повністю з кодом Java, а не C або C ++.

Використана література

1. S. C. Johnson, "YACC &emdash; Yet Another Compiler Compiler", CS Technical Report #32, Bell Telephone Laboratories, Murray Hill, NJ, 1975.
2. A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing, Reading, MA, 1986.
3. C. Fischer, and R. LeBlanc, *Crafting a Compiler with C*, Benjamin/Cummings Publishing, Redwood City, CA, 1991.

Додаток A. Граматика специфікації файлів CUP (0.10j)

```
java_cup_spec      ::= package_spec import_list class_name code_parts
symbol_list precedence_list start_spec
production_list
package_spec       ::= PACKAGE multipart_id SEMI | empty
import_list        ::= import_list import_spec | empty
import_spec        ::= IMPORT import_id SEMI
class_name         ::= CLASS ID SEMI | empty
code_part          ::= action_code_part | parser_code_part |
```

init_code | scan_code

code_parts ::= code_parts code_part | empty

action_code_part ::= ACTION CODE CODE_STRING opt_semi

parser_code_part ::= PARSER CODE CODE_STRING opt_semi

init_code ::= INIT WITH CODE_STRING opt_semi

scan_code ::= SCAN WITH CODE_STRING opt_semi

symbol_list ::= symbol_list symbol | symbol

symbol ::= TERMINAL type_id declares_term |

NON TERMINAL type_id declares_non_term |

NONTERMINAL type_id declares_non_term |

TERMINAL declares_term |

NON TERMINAL declares_non_term |

NONTERMIANL declared_non_term

term_name_list ::= term_name_list COMMA new_term_id | new_term_id

non_term_name_list ::= non_term_name_list COMMA new_non_term_id |

new_non_term_id

declares_term ::= term_name_list SEMI

declares_non_term ::= non_term_name_list SEMI

precedence_list ::= precedence_l | empty

precedence_l ::= precedence_l preced + preced;

preced ::= PRECEDENCE LEFT terminal_list SEMI

| PRECEDENCE RIGHT terminal_list SEMI

| PRECEDENCE NONASSOC terminal_list SEMI

terminal_list ::= terminal_list COMMA terminal_id | terminal_id

start_spec ::= START WITH nt_id SEMI | empty

production_list ::= production_list production | production

production ::= nt_id COLON_COLON_EQUALS rhs_list SEMI

rhs_list ::= rhs_list BAR rhs | rhs

rhs ::= prod_part_list PERCENT_PREC term_id |

prod_part_list

prod_part_list ::= prod_part_list prod_part | empty

prod_part ::= symbol_id opt_label | CODE_STRING

opt_label ::= COLON label_id | empty

```
multipart_id       ::= multipart_id DOT ID | ID
import_id          ::= multipart_id DOT STAR | multipart_id
type_id            ::= multipart_id
terminal_id        ::= term_id
term_id            ::= symbol_id
new_term_id        ::= ID
new_non_term_id    ::= ID
nt_id              ::= ID
symbol_id          ::= ID
label_id           ::= ID
opt_semi               ::= SEMI | empty
```

Додаток Б. Простий приклад сканера

```
import java_cup.runtime.*;
import sym;

public class scanner {
  /* одинарний знак пошуку */
  protected static int next_char;
  // since cup v11 we use SymbolFactories rather than Symbols
  private SymbolFactory sf = new DefaultSymbolFactory();

  /* розширене введення одним символом */
  protected static void advance()
    throws java.io.IOException
    { next_char = System.in.read(); }

  /* ініціалізувати сканер */
  public static void init()
    throws java.io.IOException
    { advance(); }
```

```
/* розпізнати та повернути наступний виконаний токен */
public static Symbol next_token()
  throws java.io.IOException
  {
    for (;;)
      switch (next_char)
            {
              case '0': case '1': case '2': case '3': case '4':
              case '5': case '6': case '7': case '8': case '9':
                /* проаналізувати ціле десяткове число */
                int i_val = 0;
                do {
                  i_val = i_val * 10 + (next_char - '0');
                  advance();
                } while (next_char >= '0' && next_char <= '9');
              return sf.newSymbol("NUMBER",sym.NUMBER, new Integer(i_val));

              case ';': advance(); return sf.newSymbol("SEMI",sym.SEMI);
              case '+': advance(); return sf.newSymbol("PLUS",sym.PLUS);
              case '-': advance(); return sf.newSymbol("MINUS",sym.MINUS);
              case '*': advance(); return sf.newSymbol("TIMES",sym.TIMES);
              case '/': advance(); return sf.newSymbol("DIVIDE",sym.DIVIDE);
              case '%': advance(); return sf.newSymbol("MOD",sym.MOD);
              case '(': advance(); return sf.newSymbol("LPAREN",sym.LPAREN);
              case ')': advance(); return sf.newSymbol("RPAREN",sym.RPAREN);

              case -1: return return sf.newSymbol("EOF",sym.EOF);

              default:
                /* у цьому простому сканері ми просто ігноруємо все інше */
                advance();
              break;
```

```
        }
    }
};
```

Додаток В. Різниця між CUP 0.9 і CUP 0.10

Зміни є серйозними, тобто немає сумісності з попередніми версіями. Зміни складаються з:

- Інший лексичний інтерфейс,
- Нові термінальні / нетермінальні декларації,
- Різні посилання на мітки,
- Інший спосіб RESULT,
- Тепер парсер повертає значення,
- Декларації пріоритету терміналу
- Правило контекстного призначення пріоритету

Лексичний інтерфейс

CUP тепер взаємодіє з лексером зовсім по-іншому. У попередніх випусках новий клас використовувався для кожного окремого типу терміналів. Однак у цьому випуску використовується лише один клас: клас *Symbol*. Клас *Symbol* має три змінні екземпляра, які є важливими для аналізатора при передачі інформації з лексеру. Перший - це змінна інстанція значення. Ця змінна містить значення цього терміналу. Це тип, оголошений як термінальний тип у файлі специфікації аналізатора. Другі два - це змінні екземпляри *left* та *right*. Вони повинні бути заповнені значенням *int*, де у вхідному файлі, відповідно до символів, був знайдений цей термінал.

Термінальні / нетермінальні декларації

Термінальні та нетермінальні декларації тепер можуть бути оголошені двома різними способами для позначення значень терміналів або нетерміналів. Попередні декларації форми:

terminal *classname terminal* [, *terminal* ...];

все ще працюють. Однак ім'я класу вказує тип значення терміналу або нетерміналу і не вказує тип об'єкта, розміщеного на стеці розбору. Декларація, наприклад:

terminal *terminal* [, *terminal* ...];
вказує, що термінали у списку не мають значення.

Посилання на мітки

Посилання на мітки не відносяться до об'єкта в стеці розбору, як у старому CUP, а до значення змінної екземпляра значення символу, що представляє цей термінал або нетермінал. Отже, посилання на термінальні та нетермінальні значення є прямими, на відміну від старих CUP, де мітки посилаються на об'єкти, що містять значення терміналу або нетерміналу.

Значення RESULT

Змінна RESULT посилається безпосередньо на значення нетерміналу, до якого правило зводить, а не на об'єкт у стеці розбору. Отже, RESULT - це той самий тип, який не є терміналом, до якого він зменшується, як заявлено в нетермінальній декларації. Знову ж таки, посилання є прямим, а не тим, що міститиме дані.

Повернення значень

Виклик для parse () або debug_parse () повертає *Symbol*. Цей *Symbol* є запуском нетермінальним, тому змінна екземпляр значення містить остаточне призначення RESULT.

Пріоритет

CUP тепер має пріоритетні термінали. новий розділ декларування, що виникає між термінальним і нетермінальним оголошеннями та граматикою, визначає пріоритет і асоціативність правил. Декларації мають форму:

precedence {left| right | nonassoc} *terminal*[, *terminal* ...];
...

Терміналам присвоюється пріоритет, коли термінали в одному рядку мають рівні пріоритети, а декларації пріоритету далі в списку декларацій пріоритету мають більший пріоритет. *Left, right* та *nonassoc* задають асоціативність цих терміналів. *Left* асоціативність відповідає зменшенню конфлікту, *right* - переходу на конфлікт, а *nonassoc* - помилці на конфлікт. Отже, тепер можуть використовуватися неоднозначні граматики.

Контекстуальні пріоритети

Нарешті, новий CUP додає контекстуальний пріоритет. Може бути оголошений таким чином:

lhs ::= *{right hand side list of terminals, non-terminals and actions}*
%prec *{terminal}*;

Додаток Г. Недоліки

У цій версії CUP важко для семантичних фраз дій отримати доступ до методу *report_error* та інших подібних методів та об'єктів, визначених у директиві коду парсера.

Це відбувається тому, що таблиці розбору знаходяться в одному об'єкті (належить до аналізатора класу або будь-якій назві, визначеній директивою - *parser*), а семантичні дії - в іншому об'єкті (класу *CUP$actions*).

Однак є спосіб це зробити, хоча це трохи некоректно. Об'єкт дії має приватне заключне поле з назвою парсер, яке вказує на об'єкт розбору. Таким чином, до змінних методів і змінних екземплярів аналізатора можна отримати доступ як до смислових дій:

```
parser.report_error(message,info);
x = parser.mydata;
```

Додаток Д. Розбір C

```
import java.io.*;
import java.util.*;
import java_cup.runtime.*;
import java_cup.runtime.XMLElement.*;
```

```
import javax.xml.stream.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;


parser code {:
   public void syntax_error(Symbol cur_token){
        System.err.println("Syntax error at "+cur_token);
   }
   public static void newScope(){
        typenames.push(new HashSet<String>());
   }
   public static void deleteScope(){
        typenames.pop();
   }
   public static boolean lookupType(String name){
        for (HashSet<String> scope: typenames)
          if (scope.contains(name)) return true;
        return false;
   }
   public static void addType(String name){
        typenames.peek().add(name);
   }
   public static LinkedList<HashSet<String>> typenames = new LinkedList<HashSet<String>>();
   public Parser(Lexer lex, ComplexSymbolFactory sf) {
        super(lex,sf);
   }

   public static void main(String args[]) {
        try {
          ComplexSymbolFactory csf = new ComplexSymbolFactory();
          // create a buffering scanner wrapper
          ScannerBuffer lexer = new ScannerBuffer(new Lexer(new BufferedReader(new
FileReader(args[0])),csf));
          // start parsing
          Parser p = new Parser(lexer,csf);
          System.out.println("Parser runs: ");
          newScope();
          XMLElement e = (XMLElement)p.parse().value;
          // create XML output file
          XMLOutputFactory outFactory = XMLOutputFactory.newInstance();
          XMLStreamWriter sw = outFactory.createXMLStreamWriter(new
FileOutputStream(args[1]));
          // dump XML output to the file
          XMLElement.dump(lexer,sw,e); //,"expr","stmt");
          // transform the parse tree into an AST and a rendered HTML version
          Transformer transformer = TransformerFactory.newInstance()
                .newTransformer(new StreamSource(new File("tree.xsl")));
          Source text = new StreamSource(new File(args[1]));
          transformer.transform(text, new StreamResult(new File("output.html")));

          System.out.println("Parsing finished!");
```

```
            } catch (Exception e) {
               e.printStackTrace();
            }
   }
:};

terminal IDENTIFIER, CONSTANT, STRING_LITERAL, SIZEOF, PTR_OP, INC_OP,
DEC_OP, LEFT_OP, RIGHT_OP, LE_OP, GE_OP,
   AND_OP, OR_OP, MUL_ASSIGN, DIV_ASSIGN, MOD_ASSIGN, ADD_ASSIGN,
SUB_ASSIGN, LEFT_ASSIGN, RIGHT_ASSIGN,
   XOR_ASSIGN, OR_ASSIGN, TYPE_NAME, TYPEDEF, EXTERN, STATIC, AUTO,
REGISTER, CHAR, SHORT, INT, LONG, SIGNED,
   UNSIGNED, FLOAT, DOUBLE, CONST, VOLATILE, VOID, STRUCT, UNION, ENUM,
ELLIPSIS, CASE, DEFAULT, IF, ELSE,
   SWITCH, WHILE, DO, FOR, GOTO, CONTINUE, BREAK, RETURN, SEMI, CURLYL,
CURLYR, COMMA, COLON, ASSIGN, PARAL,
   PARAR, SQUAREDL, SQUAREDR, POINT, ADRESS, NOT, TILDE, AND_ASSIGN,
EQ_OP, NE_OP, MINUS, PLUS, MUL, DIVIDE,
   MODULUS, LESS, GREATER, XOR, OR, COND;

non terminal translation_unit, primary_expression, postfix_expression, expression,
assignment_expression;
non terminal unary_operator, type_name, cast_expression, multiplicative_expression,
additive_expression;
non terminal shift_expression, equality_expression,  and_expression,  exclusive_or_expression;
non terminal logical_and_expression,  logical_or_expression,  conditional_expression,
constant_expression;
non terminal declaration_specifiers, init_declarator_list, storage_class_specifier, type_specifier,
type_qualifier;
non terminal init_declarator, declarator, struct_or_union_specifier, struct_declaration_list,
struct_declaration;
non terminal initializer,  specifier_qualifier_list,  struct_declarator_list,  struct_declarator,
enum_specifier;
non terminal enumerator_list,  enumerator,  pointer,  direct_declarator,  parameter_type_list,
identifier_list;
non terminal type_qualifier_list,  parameter_declaration,  abstract_declarator,
direct_abstract_declarator;
non terminal initializer_list,  statement,  labeled_statement,  compound_statement,
selection_statement;
non terminal jump_statement, statement_list, expression_statement, external_declaration,
function_definition;
non terminal assignment_operator, parameter_list, unary_expression, iteration_statement,
declaration_list;
non terminal relational_expression,  inclusive_or_expression, declaration;

precedence nonassoc ELSE;

start with translation_unit;

primary_expression ::= IDENTIFIER:ident
                     | CONSTANT:constant
                     | STRING_LITERAL:stringliteral
                     | PARAL expression:e PARAR
                     ;
```

```
postfix_expression ::= primary_expression:pe
                     | postfix_expression:pe SQUAREDL expression:index SQUAREDR
                     | postfix_expression:pe PARAL PARAR
                     | postfix_expression:pe PARAL expression:e PARAR
                     | postfix_expression:pe POINT IDENTIFIER:id
                     | postfix_expression:pe PTR_OP IDENTIFIER:id
                     | postfix_expression:pe INC_OP:op
                     | postfix_expression:pe DEC_OP:op
                     ;

unary_expression ::= postfix_expression:pe
                   | INC_OP:op unary_expression:ue
                   | DEC_OP:op unary_expression:ue
                   | unary_operator:uo cast_expression:ce
                   | SIZEOF unary_expression:ue
                   | SIZEOF PARAL type_name:tn PARAR
                   ;

unary_operator ::=ADRESS
          | MUL:op
          | PLUS:op
          | MINUS:op
          | TILDE
          | NOT:op
          ;

cast_expression
          ::=unary_expression:ue
          | PARAL type_name:tn PARAR cast_expression:ce
          ;

multiplicative_expression ::= cast_expression:ce
                            | multiplicative_expression:me MUL:op cast_expression:ce
                            | multiplicative_expression:me DIVIDE:op cast_expression:ce
                            | multiplicative_expression:me MODULUS:op cast_expression:ce
                            ;

additive_expression ::= multiplicative_expression:me
                      | additive_expression:ae PLUS:op multiplicative_expression:me
                      | additive_expression:ae MINUS:op multiplicative_expression:me
                      ;

shift_expression ::= additive_expression:ae
                   | shift_expression:se LEFT_OP additive_expression:ae
                   | shift_expression:se RIGHT_OP additive_expression:ae
                   ;

relational_expression ::= shift_expression:se
                        | relational_expression:re LESS:op shift_expression:se
                        | relational_expression:re GREATER:op shift_expression:se
                        | relational_expression:re LE_OP:op shift_expression:se
                        | relational_expression:re GE_OP:op shift_expression:se
                        ;
```

```
equality_expression ::= relational_expression:re
                        | equality_expression:ee EQ_OP:op relational_expression:re
                        | equality_expression:ee NE_OP:op relational_expression:re
                        ;

and_expression ::= equality_expression:ee
                | and_expression:ae ADRESS equality_expression:ee
                ;

exclusive_or_expression     ::= and_expression:ae
                            | exclusive_or_expression:eoe XOR and_expression:ae
                            ;

inclusive_or_expression     ::= exclusive_or_expression:eoe
                            | inclusive_or_expression:ioe OR exclusive_or_expression:eoe
                            ;

logical_and_expression ::= inclusive_or_expression:ioe
                         | logical_and_expression:lae AND_OP:op inclusive_or_expression:ioe
                         ;

logical_or_expression ::= logical_and_expression:lae
                        | logical_or_expression:loe OR_OP:op logical_and_expression:lae
                        ;

conditional_expression ::= logical_or_expression:loe
                         | logical_or_expression:loe COND expression:e COLON
conditional_expression:ce
                         ;

assignment_expression ::= conditional_expression:ce
                        | unary_expression:ue assignment_operator:aop assignment_expression:ae
                        ;

assignment_operator ::=ASSIGN
            | MUL_ASSIGN
            | DIV_ASSIGN
            | MOD_ASSIGN
            | ADD_ASSIGN
            | SUB_ASSIGN
            | LEFT_ASSIGN
            | RIGHT_ASSIGN
            | AND_ASSIGN
            | XOR_ASSIGN
            | OR_ASSIGN
            ;

expression ::= assignment_expression:ae
             | expression:e COMMA assignment_expression:ae
             ;

constant_expression ::=conditional_expression:ce
            ;
```

```
declaration ::=declaration_specifiers:ds SEMI
   | declaration_specifiers:ds init_declarator_list:idl {:
            if (ds.toString().indexOf(">typedef<")>0) {
               for (XMLElement e: ((XMLElement)idl).selectById("identifier"))
                     Parser.addType(((Terminal)e).value().toString());
            }
            :} SEMI
      ;

declaration_specifiers ::=storage_class_specifier:scc
      | storage_class_specifier:scc declaration_specifiers:ds
            | type_specifier:ts
            | type_specifier:ts declaration_specifiers:ds
            | type_qualifier:tq
            | type_qualifier:tq declaration_specifiers:ds
            ;

init_declarator_list ::=init_declarator:id
            | init_declarator_list:idl COMMA init_declarator:id
            ;

init_declarator ::=declarator:d
            | declarator:d ASSIGN initializer:i
            ;

storage_class_specifier ::=  TYPEDEF:id
            | EXTERN:id
            | STATIC:id
            | AUTO:id
            | REGISTER:id
            ;

type_specifier ::=VOID:type
            | CHAR:type
            | SHORT:type
            | INT:type
            | LONG:type
            | FLOAT:type
            | DOUBLE:type
            | SIGNED:type
            | UNSIGNED:type
            | struct_or_union_specifier:su
            | enum_specifier:es
            | TYPE_NAME:type
            ;

struct_or_union_specifier ::= STRUCT:s IDENTIFIER:id CURLYL struct_declaration_list:sdl
CURLYR
            |  STRUCT:s CURLYL struct_declaration_list:sdl CURLYR
            |  STRUCT:s IDENTIFIER:id
            |  UNION:u IDENTIFIER:id CURLYL struct_declaration_list:sdl CURLYR
            |  UNION:u CURLYL struct_declaration_list:sdl CURLYR
            |  UNION:u IDENTIFIER:id
```

```
        ;

struct_declaration_list         ::=struct_declaration:s
        | struct_declaration_list:sl struct_declaration:s
        ;

struct_declaration ::=specifier_qualifier_list:sq struct_declarator_list:sd SEMI
        ;

specifier_qualifier_list ::=type_specifier:ts specifier_qualifier_list:sq
        | type_specifier:ts
        | type_qualifier:tq specifier_qualifier_list:sq
        | type_qualifier:tq
        ;

struct_declarator_list ::=struct_declarator:s
        | struct_declarator_list:sl COMMA struct_declarator:s
        ;

struct_declarator ::=declarator:d
        | COLON constant_expression:ce
        | declarator:d COLON constant_expression:ce
        ;

enum_specifier ::=ENUM CURLYL enumerator_list:el CURLYR
        | ENUM IDENTIFIER:id CURLYL enumerator_list:el CURLYR
        | ENUM IDENTIFIER:id
        ;

enumerator_list     ::=enumerator:e
        | enumerator_list:el COMMA enumerator:e
        ;

enumerator ::=IDENTIFIER:id
        | IDENTIFIER:id ASSIGN constant_expression:ce
        ;

type_qualifier ::=CONST:id
        | VOLATILE:id
        ;

declarator ::=pointer:p direct_declarator:direct
        | direct_declarator:direct
        ;

direct_declarator ::=IDENTIFIER:identifier
        | PARAL declarator:d PARAR
        | direct_declarator:dd SQUAREDL constant_expression:ce SQUAREDR
        | direct_declarator:dd SQUAREDL SQUAREDR
        | direct_declarator:dd PARAL parameter_type_list:ptl PARAR
        | direct_declarator:dd PARAL identifier_list:il PARAR
        | direct_declarator:dd PARAL PARAR
        ;
```

```
pointer   ::=MUL:id
          | MUL:id type_qualifier_list:tql
          | MUL:id pointer:p
          | MUL:id type_qualifier_list:tql pointer:p
          ;

type_qualifier_list ::=type_qualifier:tq
          | type_qualifier_list:tql type_qualifier:tq
          ;

parameter_type_list ::=parameter_list:pl
          | parameter_list:pl COMMA ELLIPSIS:id
          ;

parameter_list ::=parameter_declaration:pd
          | parameter_list:pl COMMA parameter_declaration:pd
          ;

parameter_declaration ::=declaration_specifiers:ds declarator:d
          | declaration_specifiers:ds abstract_declarator:ad
          | declaration_specifiers:ds
          ;

identifier_list ::=IDENTIFIER:id
          | identifier_list:idl COMMA IDENTIFIER:id
          ;

type_name ::=specifier_qualifier_list:sl
          | specifier_qualifier_list:sl abstract_declarator:ad
          ;

abstract_declarator ::=pointer:p
          | direct_abstract_declarator:dad
          | pointer:p direct_abstract_declarator:d
          ;

direct_abstract_declarator ::=PARAL:id abstract_declarator:ad PARAR
          | SQUAREDL:id SQUAREDR
          | SQUAREDL:id constant_expression:ce SQUAREDR
          | direct_abstract_declarator:dad SQUAREDL:id SQUAREDR
          | direct_abstract_declarator:dad SQUAREDL:id constant_expression:ce SQUAREDR
          | PARAL:id PARAR
          | PARAL:id parameter_type_list:ptl PARAR
          | direct_abstract_declarator:dad PARAL:id PARAR
          | direct_abstract_declarator:dad PARAL:id parameter_type_list:ptl PARAR
          ;

initializer ::=assignment_expression:ae
          | CURLYL initializer_list:il CURLYR
          | CURLYL initializer_list:il COMMA CURLYR
          ;

initializer_list ::=initializer:i
          | initializer_list:il COMMA initializer:i
```

```
        ;

statement ::=labeled_statement:ls
        | {: Parser.newScope(); :}  compound_statement:cs {: Parser.deleteScope(); :}
        | expression_statement:es
        | selection_statement:ss
        | iteration_statement:is
        | jump_statement:js
        ;

labeled_statement ::=IDENTIFIER:id COLON statement:s
        | CASE constant_expression:ce COLON statement:s
        | DEFAULT COLON statement:s
        ;

compound_statement ::=CURLYL CURLYR
    | CURLYL statement_list:sl CURLYR
        | CURLYL declaration_list:dl CURLYR
        | CURLYL declaration_list:dl statement_list:sl CURLYR
        ;

declaration_list ::=declaration:d
        | declaration_list:dl declaration:d
        ;

statement_list ::=statement:s
        | statement_list:sl statement:s
        ;

expression_statement ::=SEMI
        | expression:e SEMI
        ;

selection_statement ::=IF PARAL expression:e PARAR statement:s
        | IF PARAL expression:e PARAR statement:s1 ELSE statement:s2
        | SWITCH PARAL expression:e PARAR statement:s
        ;

iteration_statement ::=WHILE PARAL expression:e PARAR statement:s
        | DO statement:s WHILE PARAL expression:e PARAR SEMI
        | FOR PARAL expression_statement:es1 expression_statement:es2 PARAR statement:s
        | FOR PARAL expression_statement:es1 expression_statement:es2 expression:e PARAR
statement:s
        ;

jump_statement ::=GOTO IDENTIFIER:id SEMI
        | CONTINUE SEMI
        | BREAK SEMI
        | RETURN SEMI
        | RETURN expression:e SEMI
        ;

translation_unit ::=external_declaration:ed
        | translation_unit:tu external_declaration:ed
```

```
        ;

external_declaration ::=function_definition:fd
        | declaration:d
        ;

function_definition ::=declaration_specifiers:ds declarator:d declaration_list:dl
                                                        {: Parser.newScope(); :}
  compound_statement:cs                         {: Parser.deleteScope(); :}
        | declaration_specifiers:ds declarator:d        {: Parser.newScope(); :}
          compound_statement:cs                 {: Parser.deleteScope(); :}
        | declarator:d declaration_list:dl          {: Parser.newScope(); :}
          compound_statement:cs                 {: Parser.deleteScope(); :}
        | declarator:d                      {: Parser.newScope(); :}
          compound_statement:cs                 {: Parser.deleteScope(); :}
```