Д.І. Нагірняк
Херсонський державний університет

Використання ANTLR при реалізації інтерпретаторів

Теренс Парр пише [7; 10; 11], що він займається написанням компіляторів та трансляторів з 1984 року. Звичайно перші з них він розробляв „вручну” методом рекурсивного спуску. В кінці кінців йому знадобився засіб для автоматизації даної роботи. Але, як говорить Теренс Парр, він терпіти не міг один із популярних на той час засіб під назвою YACC (та подібні до нього) саме тому, що вони генерують незрозумілу таблицю цілих чисел замість зрозумілого для людини коду. Отже, LR-базована технологія була значно складнішою для розуміння, ніж інтуїтивно зрозуміла LL-базована технологія рекурсивного спуску. Він робить навіть більше, ніж розробнику було потрібно, коли він не використовував додаткових засобів. За кілька років Теренс Парр розширив свій інструмент такими засобами як обхід дерева.

Популярність ANTLR обумовлена такими фактами, які задовольняють фундаментальним вимогам. Програмісти хочуть використовувати такі інструменти, які:

- надають зрозумілий для них механізм,
- є достатньо потужними для розв’язування потрібних задач,
- є гнучкими,
- автоматизують трудоємкі задачі,
- генерують вихід, котрий легко інтегрувати в код програми.

ANTLR має узгоджений синтаксис для опису лексера (Lexer), парсера (Parser) та парсера дерев (TreeParser або TreeWalker).

Символ	Опис
`( . . . )`	підправило
`( . . . ) *`	замикання жодного або більше підправила
`( . . . ) +`	замикання одного або більше підправила
`( . . . ) ?`	необов’язкове правило жодного або одне
`{ . . . }`	семантичні дії
`[ . . . ]`	аргументи правил
`{ . . . } ?`	семантичний предикат
`( . . . ) =>`	синтаксичний предикат
`|`	альтернативний оператор

`..`	оператор діапазону
`~`	Not оператор
`.`	груповий символ (будь-який)
`=`	оператор надання значення
`:`	оператор мітки, початок правила
`;`	кінець правила
`class`	граматичний клас
`extends`	описує базовий клас граматики
`returns`	описує тип параметра, що повертається із функції
`options`	розділ опцій

Пояснити, чому використання ANTLR є дуже корисним, можна лише після того, як програміст вже має деякий досвід написання парсерів. ANTLR є одним із небагатьох мовних інструментів, який дозволяє перетворювати граматичні структури в дерева.

Основою ANTLR є граматичні файли, в яких описуються три частини [6; 7; 8]:

- Lexer – розпізнає та виділяє токени.
- Parser – будує абстрактне синтаксичне дерево.
- TreeParser (TreeWalker) – реалізує обхід дерева.

Правила опису кожного із них дуже подібні.

Після компіляції файлів граматики системою ANTLR отримується декілька вихідних файлів (мовою C++, C# або Java), які реалізують Lexer, Parser та TreeWalker.

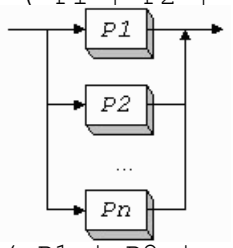
Словник мови складає:

- Пробіли. Пробіли, символи табуляції та символ переходу на новий рядок є роздільниками в мові.
- Коментарі. Можуть бути строковими, які починаються із символів „//”, або блочними. Початок блоку — „/*”, кінець — „*/”.
- Символи. Символьні літерали такі, як і в мові Java. Можуть містити вісімкові символи (наприклад, ‘\377’), символи Юнікода (наприклад, ‘\uFF00’) та різні ескейп послідовності (як в мові Java, C++).
- Рядки. Рядкові літерали – це послідовність символів, що заключені в подвійних лапках.

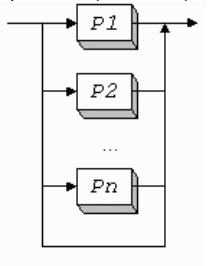
- Означення токенів. Опис токена в розділі лексера.
- Означення правила. Опис правила в розділі парсера.
- Семантичні дії. Заключаються в фігурні дужки.
- Аргументи правил. Заключаються в квадратні дужки. Вони являють собою параметри для правила.
- Символи.

ANTLR підтримує розширену нотацію в формі Бекаса-Наура (BNF) для наступних чотирьох підправил. Нижче зображені синтаксичні діаграми для них.

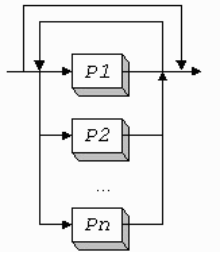
```
( P1 | P2 | ... | Pn )
```



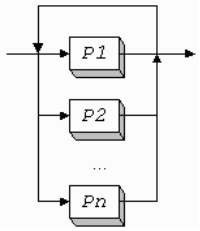

```
( P1 | P2 | ... | Pn )?
```




```
( P1 | P2 | ... | Pn )*
```




```
( P1 | P2 | ... | Pn )+
```



Для поставленої задачі можна повністю автоматизувати процес лексичного аналізу. Можна повністю описати граматику для лексичного аналізатора (на прикладі мови Паскаль).

```
class PascalLexer extends Lexer;
options {
  exportVocab = Pascal;
  caseSensitive = false;
  caseSensitiveLiterals = false;
}
tokens {
  AND   = "and";
  ARRAY       = "array";
  BEGIN       = "begin";
  BOOLEAN   = "boolean";
  CASE  = "case";
  CHAR = "char";
  CONST_TOKEN    = "const";
  DIV           = "div";
  DO            = "do";
  DOWNTO    = "downto";
  ELSE  = "else";
  END   = "end";
  FOR           = "for";
  FUNCTION  = "function";
  IF            = "if";
  IN_TOKEN  = "in";
  INTEGER     = "integer";
  LABEL= "label";
  MOD   = "mod";
  NOT   = "not";
  OR            = "or";
  PROCEDURE       = "procedure";
  PROGRAM   = "program";
  REAL  = "real" ;
  RECORD      = "record";
  REPEAT      = "repeat";
  THEN = "then";
  TO            = "to";
  TYPE  = "type";
  UNTIL= "until";
  VAR           = "var";
  WHILE       = "while";
  WITH = "with";
  STRING      = "string";
}
```

Розділювачі (в тому числі пробіли), коментарі, ідентифікатори, цілі та дійсні числа описуються звичним способом в ANTLR граматиці.

Отже для написання лексичного аналізатора слід, фактично, лише описати токени.

Детальніше розберемо синтаксичний аналіз. Він є дещо складнішим, хоча майже всю роботу на себе тут знову бере ANTLR. Далі будемо використовувати в якості прикладів псевдокод, який схожий на мову програмування C++ щоб не загромаджувати неважливими елементами.

Єдине на чім можна зупинити увагу, це наступні моменти.

Проблема 1:

В стандарті мови Паскаль виклик функції без параметрів має такий вигляд:

functionName;

Отже, на етапі побудови абстрактного синтаксичного дерева можна розрізнити, що це є ідентифікатор (тобто генеруємо вершину з типом IDENT). Ми не можемо генерувати вершину із типом FUNC_CALL (виклик функції) або PROC_CALL (виклик процедури), оскільки ми не знаємо типу ідентифікатора.

Розв'язок:

При знаходженні опису функції заносимо її до таблиці символів в даній області видимості

```
functionDeclaration
   :! func:FUNCTION^ id:identifier other:functionPart
{

      PascalAST funcAST = #(#[FUNCTION], #id, #other);
      Function funcItem = new Function(funcAST);
      if ( !_scopes.Declare(funcItem) )
      throw reportError("Identifier "+funcItem.Text +" already
defined");
}
      body:block
{
      ## = funcAST;
      ((PascalAST)##.getFirstChild()).getLastSibling().setNext
Sibling(#body);
}
         ;
```

Слід зазначити, що на даному етапі частково проводиться семантичний аналіз, оскільки в даному випадку можна визначити, чи є вже функція із заданим іменем в стеці.

Потім при знаходженні ідентифікатора в семантичних діях можна вибрати, що потрібно: ідентифікатор чи виклик процедури/функції.

```
    identifier !
       : i:IDENT
    {
      PascalItem item = _scopes[i.getText()];
      Function func = item as Function;
```

```
    if (func != null && _generateFuncCallForIdent) {
        #identifier = #([FUNC_CALL, "func_call"], #i);
    }
    else
        #identifier = #i;
}
  ;
```

Проблема 2:

Нехай в програмі вхідною мовою (Паскаль) описана функція з іменем func. В тілі цієї функції є оператор присвоювання func := 1. Потрібно в цьому місці згенерувати не виклик функції, а ідентифікатор. Але у зв'язку з тим, що функція з іменем func вже є в таблиці символів, то буде згенеровано саме виклик функції.

Розв'язок:

Для розв'язку проблеми доповнимо семантичну дію в правилі identifier. В тому місці, де перевіряється, чи є даний ідентифікатор функцією поставимо більш строгу умову. Для цього використаємо прапорець з іменем *_generateFuncCallForIdent.*

Цей прапорець завжди дорівнює true (тобто якщо ідентифікатор — функція, то генерувати виклик функції). Але ми змінюємо його в операторі надання значення для лівостороннього значення (змінної). Тому завжди буде згенеровано ідентифікатор, навіть якщо він є функцією.

```
assignmentStatement
{ _generateFuncCallForIdent = false; }
  : variable
{ _generateFuncCallForIdent = true; }
  ASSIGN^ expression
  ;
```

Для синтаксичного аналізу в файлі з граматикою було написано лише цей код на цільовій мові (C++). Все інше знову взяв на себе інструмент ANTLR. Звичайно порівняно із обсягом роботи, який був затрачений на побудову лексичного аналізатора обсяг роботи на побудову синтаксичного аналізатор був значно більшим. Хоча слід зауважити, що на написання парсера „вручну" потрібно було в кілька раз (або, навіть, в кілька порядків раз) більше часу. Крім того тепер буде легко додавати нові синтаксичні конструкції або змінювати існуючі (хоча це є ризиком зміни вимог!). Для цього потрібно лише змінити відповідні синтаксичні правила, згенерувати код та перекомпілювати систему (точніше модуль інтерпретатора системи). В результаті ми отримаємо досить гнучку і розширювану систему.

Для того, щоб наглядно побачити дерево, яке побудувано на даному етапі, була написана функція, котра зберігає інформацію про дерево в XML файлі. Ось його вигляд та відповідна програма на Паскалі:

Програма мовою Паскаль:

```
Program Name;
var
  a, s: Integer;
  r : Integer;
Function f: Integer;
begin
  f := 1
end;
Begin
  f;
End.
```

Абстрактне синтаксичне дерево:

```
- <root text="Program" token="_program_">
  <root text="Name" token="_IDENT_" />
- <root text="var" token="_var_">
- <root text=":" token="_VARDECL_">
- <root text="" token="_IDLIST_">
                <root text="a" token="_IDENT_" />
                <root text="s" token="_IDENT_" />
  </root>
          <root text="Integer" token="_integer_" />
  </root>
- <root text=":" token="_VARDECL_">
- <root text="" token="_IDLIST_">
                <root text="r" token="_IDENT_" />
          </root>
          <root text="Integer" token="_integer_" />
  </root>
</root>
- <root text="" token="_function_">
  <root text="f" token="_IDENT_" />
  <root text="Integer" token="_integer_" />
- <root text="block" token="_BLOCK_">
- <root text=":=" token="_ASSIGN_">
                        <root text="f"
token="_IDENT_" />
                        <root text="1"
token="_NUM_INT_" />
          </root>
</root>
</root>
- <root text="block" token="_BLOCK_">
- <root text="func_call" token="_FUNC_CALL_">
<root text="f" token="_IDENT_" />
          </root>
</root>
</root>
```

Проаналізувавши вимоги, використовуючи об’єктну модель та специфікацію мови Паскаль, можна сказати, що інтерпретація дуже схожа на процес перевірки семантики (тобто вони можуть

використовувати однакові засоби). Проілюструємо це твердження на таблиці.

Операція	**Процес перевірки семантики**	**Процес інтерпре-тації**	**Коментар**
Оператор надання значення	Допустимість типів	+ надання значення	Можна включити перевірку після допустимості типів і в різних процесах виконувати, або ні надання значення. Тобто перевірка допустимості типів буде виконуватись автоматично. Можна реалізувати у внутрішніх класах системи.
Оператор if	Допустимість типів, перевірка блоку then, перевірка блоку else	Допустимість типів, вибір блоку та його виконання.	Допустимість типів перевіряються автоматично. Дії відрізняються, тому їх потрібно описати в граматиці ANTLR.
Оператори циклу	Допустимість типів, перевірка блоку	Допустимість типів; *якщо* потрібно, то *виконання* блоку	Допустимість типів перевіряються автоматично. Дії відрізняються, тому їх потрібно описати в граматиці ANTLR.
Виклик процедури/ функції	Допустимість типів, відповідність фактичних параметрів формальним.	+ виклик процедури/ функції.	Допустимість типів перевіряються автоматично. В залежності від процесу виконуємо виклик або ні. Можна реалізувати у внутрішніх класах системи.

Отже, як для процесу перевірки семантичної правильності, так і для процесу інтерпретації будемо використовувати ті ж засоби (звичайно, модифіковані для таких потреб).

Наведемо приклад реалізації оператора надання значення SetValue(RValue val) класу LValue.

```
virtual void LValue.SetValue(RValue val)
{
```

```
    // Якщо, приведення типів недопустиме, то виникне виняткова ситуація (Exception)
      _type.Convert(val.Type);
    // Якщо не процес інтерпретації, то більше нічого не виконуємо
      if ( !Running)
            return;
      _value = val.Value;
      _type = val.Type;
    }
```

Аналогічно реалізуються інші операції. Спочатку проводимо саму перевірку, потім виконуємо операції, якщо це є процес інтерпретації, а не перевірки семантики.

Особливим способом опрацьовуються оператори циклу та умовний оператор. Це пов'язано з тим, що на етапі перевірки семантичної правильності програми потрібно завжди виконувати (точніше імітувати виконання) кожну операцію кожної конструкції оператора для того, щоб перевірити всі операції. А на етапі інтерпретації потрібно вибрати один із них і виконати тільки його. Тому таку різницю не можна реалізувати у внутрішніх класах інтерпретатора, оскільки в них виконуються локальні (атомарні) дії. Для розв'язування цієї проблеми необхідно реалізувати розподіл на етапи в самому файлі граматики.

Покажемо, як це було зроблено на прикладі оператора if.

```
ifStatement
{
  RValue cond ;
  bool val;
}
: #(IF cond=expression {val = cond.GetBoolean();}
  statement[val || !_scopes.Running] (statement[!val || !_scopes.Running])?)
;
```

Як бачимо, така реалізація досить проста. Довелось лише ввести в правило statement додатковий параметр, який вказує на те, чи потрібно виконувати дану операцію (або блок операцій) чи ні. Слід зазначити, що операція виконується завжди (незалежно від значення cond) у тому випадку, якщо перевіряється семантика (!_scopes.Running).

ЛІТЕРАТУРА

1. Abelson H. and Sussman G.J. Structure and Interpretation of Computer programs, MIT Press, Cambridge, Mass, 1985.
2. Knuth, D.E., 1971. Topdown Synatax Analisis, Acta informatica, vol. 1, pp. 79-110. Technical Report 39, Bell Laboratories, NJ.
3. Loudon, K.C., 1997. Compiler Construction: Principles and Practice, PWS Publishing Ltd.
4. Stroustrup, B. 1991. The C++ Programming Language. Second Edition. Reading, MA: Addison-Wesley

5. Terence Parr, An Introduction To ANTLR http://www.cs.usfca.edu/~parrt/course/652/lectures/antlr.html
6. Terence Parr, and Russell Quong, LL and LR Translator Need k http://www.antlr.org/article/needlook.html
7. Terence Parr, ANTLR Reference Manual http://www.antlr.org/doc/index.html
8. Terence Parr, Exactly 1800 Words on Languages and Parsing http://www.antlr.org/article/langparse.html
9. Terence Parr,ANTLR Forum Home Page http://www.jguru.com/forums/ANTLR
10. Terence Parr,What is ANTLR http://www.antlr.org/about.html
11. Terence Parr,Why use ANTLR http://www.antlr.org/why.html
12. Terry, P.D., 1997. Compilers and Compiler Generators: an Introduction with C++ International Thomson Computer Press.
13. Watt, D.A., 1997. An extended attribute grammar for Pascal, Report number 11, Department of Computing, University of Glasgow.
14. Ахо, Альфред, В., Сети, Рави, Ульман, Джеффри, Д. Компиляторы: принципы, технологии и инструменты.: Пер. с англ. – М.: Издательский дом «Вильямс», 2001. – 768 с.: ил. – Парал. тит. англ.
15. Бадд Т. Объектно-ориентированное программирование в действии./Пер. с англ. – СПб.: Питер, 1997. – 464 с., ил.
16. Бальцук Н.Б., Буняев М.М., Матросів В.Л. Деякі можливості використання електронно-обчислювальної техніки в навчальному процесі. – М.: Прометей, 1989. - 135 с.
17. Буч Г. Объектно-ориентированное проектирование с примерами применения /Пер. с англ. – М.: Конкорд, 1992. – 519 с., ил.
18. Васюкова Н.Д., Тюляева В.В. Практикум по основам программирования. Язык Паскаль: Учеб. пособие для учащихся сред. спец. учеб. заведений. – М.: Высш. Шк., 1991. – 160 с.: ил.
19. Гуннерсон Э. Введение в C#. Библиотека программиста. – СПб: Питер, 2001. – 304 с.: ил.
20. Жалдак М.И. Система подготовки учителя к использованию информационной технологии в учебном процессе. – Дисс. на соискание уч. ст. доктора пед. наук. – М, НИИСИМО АПН СССР,1989. – 48 с.
21. Дейтл, П.Дж. Дейтел, Т.Р. Нието, и другие. Как программировать на XML / Х.М. Пер. с англ. – М.: ЗАО "Издательство БИНОМ", 2001. – 944 с., ил.
22. Керниган Б. Ритчи Д. Язык программирования Си./Пер. с англ., 3-е изд., испр. – СПб.: "Невский Диалект", 2001. – 352 с., ил.
23. Кнут Д. Исскуство программирования. Т.1 Основные алгоритмы. — 3-е изд. – М.: Издательский дом „Вильямс”, 2000
24. Кнут Д. О переводе (трансляции) языков слева направо// Сб. «Языки и автомати» — М.: Мир, 1975. — С. 9-42.

25. Ларман, Крег. Применение UML и шаблонов проектирования.: Пер.с англ.: Уч. пос. –М.: Издательский дом «Вильямс», 2001. – 496 с.: ил. – Парал. тит. англ.
26. Мархель И.И., Овакімян Ю.О. Комплексний підхід до використання технічних засобів навчання: Навч.-метод. посібник. – М.: Вищ. шк., 1987. – 175 с.: іл.
27. Машбиц Е.И. Психолого-педагогические проблемы компьютеризации обучения. – М.: Педагогика, 1988. – 191 с.
28. Новиков Ф.А. Дискретная математика для программистов. – СПб: Питер, 2001. – 304 с.: ил.
29. Ройс Уокер. Управление проектами по созданию програмного обеспечения. Унифицировнный подход / Пер. с англ. – М.: ЛОРИ, 2002. – 426 с.
30. Ройс Уокер. Управление проектами по созданию програмного обеспечения. Унифицированній поход / Пер. с англ. – М.: ЛОРИ, 2002. – 426 с.
31. Себеста, Роберт, У. Основные концепции языков программирования, 5-е изд.: Пер. с англ. – М.: Издательский дом «Вильямс», 2001. – 672 с.: ил. – Парал. тит. англ.
32. Соммервил, Иан. Инженерия програмного обеспечения, 6-е издание / Пер. с англ. – М.: Издательский дом "Вильямс", 2002. – 624 с.
33. Співаковський О.В., Львов М.С. Кравцов Г.М., Крекнін В.А., Зайцева Т.В., Кушнір Н.О., Кот С.М. Педагогічні технології та педагогічно-орієнтовані програмні системи: предметно-орієнтований підхід // Комп′ютер у школі та сім′ї. – 2002.- №2(20). – С. 17-22.
34. Фридман. А. Основы объектно-ориентированного программирования на языке С++. – М.: «Горячая линия - Телеком», 1999. – 208с.
35. Хантер, Робин. Основные концепции компиляторов.: Пер. с англ. — М.: Издательский дом «Вильямс», 2002. — 256 с.: ил. Парал. тит. англ.
36. Хомский Н. Три модели для описания языка// Кибернетический сборник. — М.: ИЛ, 1961. — Вып. 2. – С. 237-266
37. Шлеер С., С.Мэллор Объектно-ориентированный анализ: Моделирование мира в состояниях. – К.: Диалектика, 1993. – 240 с.
38. Шлеер. С. Объектно-ориентированный анализ: Моделирование мира в состояниях. – К.: Диалектика, 1993. – 240 с.
39. Wirth, N. January 1983. Program Development by Stepwise Refinement. Communications of the ACM vol.26(1)
40. Dijkstra, E. January 1993. American Programmer vol.6(1)