4. Синтактичен анализ отгоре-надолу. Рекурсивно спускане. Построяване на синтактичен анализатор чрез рекурсивно спускане

Цел на упражнението

Упражнението представя алгоритъма на рекурсивното спускане за реализация на синтактичен анализ отгоре-надолу. Показано е как се съставят синтактични функции за разпознаване на символи от формална граматика. Демонстрира се как се прилага рекурсивното спускане за разработване на синтактичен анализатор за учебния програмен език STUDENT.

4.1. Методи за синтактичен анализ

Задачата на синтактичния анализ е разпознаване на изречение на входния език. Тази задача се свежда до отговор на въпроса принадлежи ли даден низ *s* на език *L(G)*, т.е. $s \in L(G)$, където низът s е входната програма, а езикът L е множество от низове, дефинирано с формална граматика G. Синтактичният анализ представлява процес на построяване на синтактично дърво, което представя заместване на символи в съответствие с правилата на граматиката.

Съществуват два общи метода за синтактичен анализ:

- Синтактичен анализ отгоре-надолу (или още низходящ синтактичен анализ)

При този метод разпознаването на изречението започва от стартовия символ на граматиката. Правилата се прилагат отляво надясно, т.е. нетерминалният символ в лявата страна на правило се замества със символ или низ от символи, намиращи се в дясната страна на правилото. Прилагането на правилата продължава до получаване на низ от терминални символи. Ако полученият низ съвпада с разпознавания низ, то входната програма е валидно изречение на езика. Синтактичното дърво се строи от корена към листата.

- Синтактичен анализ отдолу-нагоре (или още възходящ синтактичен анализ)

Разпознаването на изречението започва от разпознавания низ. Правилата се прилагат отдясно наляво, т.е. подниз от разпознавания низ, съответстващ на дясната страна на някое от правилата, се замества с нетерминалния символ в лявата страна на правилото. Ако чрез прилагане на правилата се получи редуциране на входния низ до единствен нетерминален символ, който е стартов символ в граматиката, то разпознавания низ (т.е. входната програма) представлява валидно изречение на езика. При този метод синтактичното дърво се строи от листата към корена.

Низходящият синтактичен анализ е по-лесен за реализация, но работи с ограничено подмножество на контекстно-свободни граматики. Възходящият синтактичен анализ работи с по-широк клас от контекстно-свободни граматики, но е по-труден за реализация, защото изисква използване на стек.

В учебния компилатор е реализиран низходящ синтактичен анализ. За целта е използван най-простият алгоритъм за низходящ синтактичен анализ, наречен рекурсивно спускане.

4.2. LL(1) граматика

Граматиката на езика STUDENT е от тип LL(1). LL(1) граматиките са подмножество на контекстно-свободните граматики (т.е. граматиките от тип 2 по класификацията на Хомски). Тези граматики позволяват реализация на низходящ синтактичен анализ с рекурсивно спускане.

Абревиатурата LL(1) има следния смисъл:

- L (left) - прочитане на входната програма (изречение) от ляво на дясно
- L (left) - прилагане на правилата на граматиката от ляво на дясно
- 1 – в правила с две или повече разклонения изборът на клон се изпълнява чрез прочитане на една лексема (символ) напред.

Всяко правило в LL(1) граматиката отговаря на следните изисквания:

- Изискване 1 се отнася за всички правила с алтернативни клонове, например

A : A1 | A2 | ... | An.

За такова правило трябва да е изпълнено условието

first(Ai) ∩ first(Aj) = ∅, за всички i != j

, където first(Ai) и first(Aj) са съответно множества от терминални символи, с които е допустимо да започват нетерминалните символи Ai и Aj.

- Изискване 2 се отнася за правила, генериращи празен низ, т.е.

A : ε

За такова правило трябва да е изпълнено условието

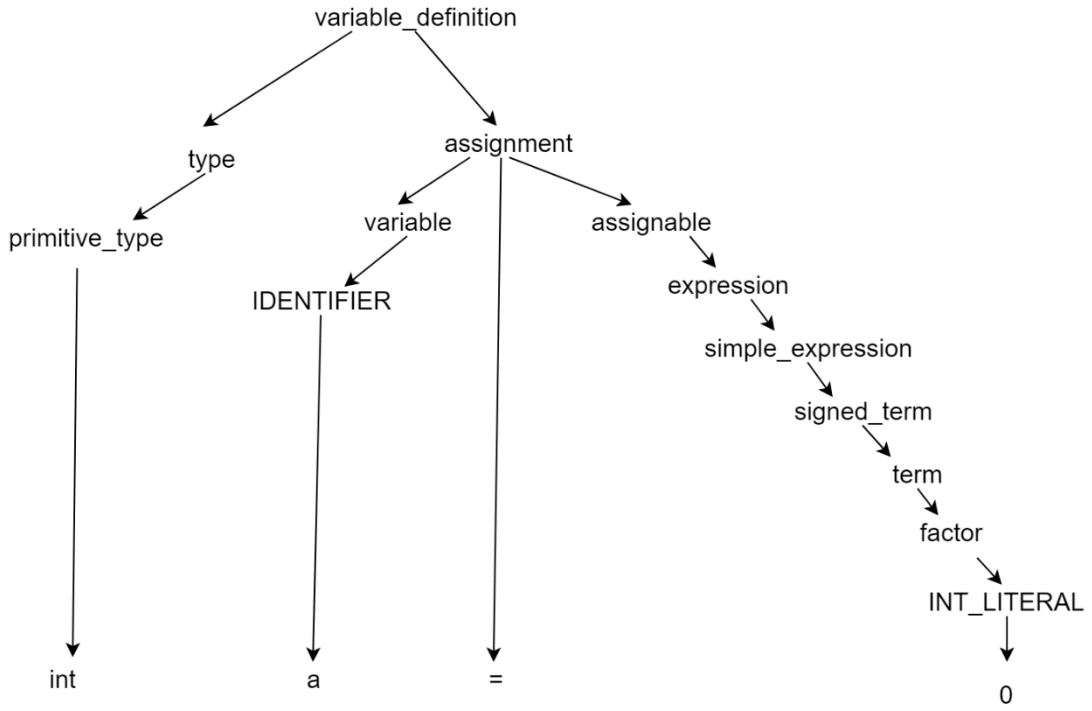
first(A) ∩ follow(A) = ∅

, където follow(A) е множество от терминални символи, които могат да следват непосредствено нетерминалния символ A в изречението.

4.3. Рекурсивно спускане

При низходящия синтактичен анализ синтактичното дърво се построява отгоре надолу, като се започва от корена на дървото, обозначен със стартовия символ в граматиката, а следващите върхове съответстват на синтаксиса на анализираното изречение. Граматичните правила се прилагат отляво надясно. Построяването на дървото приключва, когато в листата се получи разпознаваното изречение.

Фиг. 1 показва синтактичното дърво за нетерминалния символ variable_definition от граматиката на STUDENT при разпознаване на низ int a = 0.



Фиг. 1. Синтактично дърво за variable_definition

За реализация на низходящия синтактичен анализ е използван алгоритъм на рекурсивното спускане (нар. още LL(1) – анализ). Синтактичният анализатор се състои от отделни функции. В общия случай на всеки нетерминален символ от граматиката съответства една функция, отговорна за разпознаване на този нетерминален символ. Действието на всяка функция се състои в разпознаване на подниз, започващ с текущата прочетена лексема, съответстващ на дясната страна на правилото, дефиниращо нетерминалния символ. Дадена функция може да извиква други функции или самата себе си. След като функцията разпознае подниза, свързан с реализирания нетерминален символ, тя завършва работата си и предава управлението на извикващата функция. Текуща става следващата лексема от входния поток. Ако функцията не разпознае нетерминалния символ, се извежда съобщение за грешка.

4.4. Програмна структура на учебния синтактичен анализатор

Входните данни за синтактичния анализатор представляват поток от лексеми, получени от лексическия анализатор. При успешно разпознаване на изречението анализаторът извежда абстрактно синтактично дърво на входната програма. В противен случай се извежда съобщение за синтактична грешка и работата на анализатора приключва.

Функционалността на синтактичния анализатор е реализирана в програмния клас ParserImpl (файл ParserImpl.java). Променливата currentToken съдържа поредната прочетена лексема.

4.4.1. Обработване на терминални символи

Обработване на терминален символ в кода на синтактичния анализатор става в метод accept:

```
    protected void accept(TokenType tokenType) {
        if (currentToken.getTokenType() != tokenType) {
            throw new SyntaxException("Token doesn't match! Expected " + tokenType + ", Got " + currentToken.getTokenType(), currentToken);
        }
        currentToken = lexer.nextToken();
    }
```

Този метод сравнява текущо прочетената с очакваната лексема според правилото за разпознавания нетерминален символ. Ако лексемите не съвпадат, се генерира изключение, което извежда съобщение за синтактична грешка. При съвпадение на прочетената и очакваната лексема анализът продължава с прочитане на следващата лексема от потока.

4.4.2. Обработване на нетерминални символи

В учебния синтактичен анализатор е реализиран алгоритъм на рекурсивното спускане. Програмната структура на синтактичния анализатор следва изискванията на рекурсивното спускане. В общия случай на нетерминален символ от граматиката на STUDENT съответства метод на класа, който е отговорен за разпознаването на символи. Например:

- Нетерминалният символ program, дефиниран в правило 1 на граматиката, се разпознава от метода entryRule()

- Нетерминалният символ program_body, дефиниран в правило 2, се разпознава от метода programBody()

- Нетерминалният символ main_function, дефиниран в правило 3, се разпознава от метода mainFunction()

и т.н.

Има някои изключения, при които разпознаването на повече нетерминални символи е реализирано в един метод. Например методът type() разпознава нетерминалните символи type, primitive_type и array_type (правило 6).

4.4.3. Разпознаване на синтактични грешки

Синтактичният анализатор разпознава повече грешки в сравнение с лексическия анализатор. Примери за синтактични грешки са:

- Незатворена скоба
- Невалиден оператор

Обработването на синтактични грешки се реализира чрез генериране на изключение SyntaxException.

4.5. Примери за реализация на синтактични функции

Пример 1. Нека разгледаме едно примерно правило. В него са използвани фиктивни символи, като с главни букви са означени нетерминални символи, а с малки букви са означени терминални символи.

A : abB

Синтактичната функция за разпознаване на нетерминала A изглежда например по следния начин:

```
void A()
{
  accept(a);
  accept(b);
  B();
}
```

Терминалните символи a и b се обработват с функция accept(), а за нетерминалния символ B се извиква съответна функция, която е отговорна за разпознаването му.

Пример 2. Дадено е следното правило:

A : a(bB)?

Специалният символ '?' означава, че част от правилото може да се приложи нито веднъж или само веднъж. Приемаме, че текущо прочетената лексема от входния поток се съхранява в променлива currentToken. Синтактичната функция за разпознаване на нетерминала A изглежда по следния начин:

```
void A()
{
  accept(a);
  if (currentToken == b)
  {
    accept(b);
    B();
  }
        }
```

Пример 3. Дадено е следното правило:

A : a(bB)*

Специалният символ '*' означава, че част от правилото може да се прилага нула, един или множество пъти. Синтактичната функция за разпознаване на нетерминала A изглежда по следния начин:

```
void A()
{
 accept(a);
 while (currentToken == b)
 {
  accept(b);
  B();
 }
}
```

Пример 4. Дадено е следното правило:

A : abB | cC

Това правило се състои от два алтернативни клона. Това означава, че нетерминалният символ A може да се замести с низ “abB” или с низ “cC”. Изборът на клон от правило се извършва чрез поглеждане един символ напред във входния поток, т.е. проверява се следващата лексема. Ако следващата лексема е символ a, то ще се приложи първият клон и нетерминалът A ще бъде заместен с “abB”. В противен случай ще се приложи вторият клон и нетерминалът A ще бъде заместен с “cC”.

Синтактичната функция за разпознаване на A в този случай има следния вид:

```
void A()
{
 if (currentToken == a)
 {
  accept(a);
  accept(b);
  B();
 }
 else
 {
  accept(c);
  C();
 }
}
```

Пример 5. Дадено е следното правило:

A : abB | d(aA)* | e(eE)?

Това правило се състои от множество клони, поради което е подходящо да бъде реализирано чрез многовариантен избор. Синтактичната функция за разпознаване на нетерминалния символ A може да изглежда по следния начин:

```
void A()
{
 switch (currentToken)
 {
  case a:
   accept(a);
   accept(b);
   B();
   break;
  case d:
   accept(d);
   while (currentToken == a)
   {
    accept(a);
    A();
   }
   break;
  case e:
   accept(e);
   if (currentToken == e)
   {
    accept(e);
    E();
   }
 }
  break;
```

```
    default:
      SyntaxException("Token doesn't match!");
}
```

4.6. Примери за синтактични функции в STUDENT

Примери в предната подточка демонстрират съставянето на синтактичните функции в методите на учебния синтактичен анализатор. Разликата е, че се използват символи от граматиката на STUDENT. По-долу са показани базовите имплементации на някои от методите. В кода на някои от методите има добавки, свързани с построяването на абстрактно синтактично дърво, които са обяснени в следващото упражнение.

Пример 1. Метод, разпознаващ стартовия нетерминален символ program

Стартовият символ program е дефиниран в правило 1:

```
(1) program              : PROGRAM LBRACKET program_body  RBRACKET
```

Правилото включва три терминални символа (PROGRAM, LBRACKET и RBRACKET) и един нетерминален символ (program_body). Методът entryRule, който разпознава program, изглежда по следния начин:

```
public AST entryRule() {
   accept(TokenType.PROGRAM);
   accept(TokenType.LBRACKET);
   programBody();
   accept(TokenType.RBRACKET);
   return currentNode;
}
```

Пример 2. Метод, разпознаващ оператор while (правило 20)

```
(20) while_statement       : WHILE LPAREN expression RPAREN block
```

```
void whileStatement() {
   accept(TokenType.WHILE);
   accept(TokenType.LPAREN);
   expression();
   accept(TokenType.RPAREN);
   block();
}
```

В правила, които съдържат множество клонове, избор на клон се реализира чрез поглеждане една лексема напред.

Пример 3. Метод, разпознаващ оператор (правило 10)

(10) statement : simple_statement SEMICOLON | compound_statement

Например за оператор (правило statement) се проверява дали следващата лексема е стартов символ от прост оператор (правило simple_statement). В този случай се избира прост оператор (simple_statement), в противен случай – съставен оператор (compound_statement).

```
void statement() {
if (TokenType.isSimpleStatementTerminal(currentToken.getTokenType())){
      simpleStatement();
      accept(TokenType.SEMICOLON);
   } else {
      compoundStatement();
   }
}
```

Контролни въпроси:

1. Какво означава LL(1)?

2. Може ли да се реализира рекурсивно спускане за правила с лява рекурсия, т.е. правила от вида:

A -> Ab

, където A е нетерминален символ, b е терминален символ? Обосновете отговора.

3. Защо се налага извикването на функция в езика STUDENT да започва с допълнителен символ (в случая символ ‘@’), предшестващ идентификатора (т.е. името на функцията)?

Упътване: Разгледайте правило 15 от грматиката на STUDENT, което дефинира нетерминалния символ function_call. Припомнете си с какъв символ може да започва валиден идентификатор в езика.

Задачи:

1. Да се състави синтактична функция за разпознаване на нетерминалния символ B чрез рекурсивно спускане. Главните букви означават нетерминален символ, а малките терминален.

а) B : abB

б) B : a(bB)*

в) B : a(bB)?

г) B : aA | bB | c(aC)*

2. Да се довършат телата на методите на класа ParserImpl във файла ParserImpl.java на учебния синтактичен анализатор, маркирани с коментар /* ToDo handle symbol */.

3. Да се тества работата на синтактичния анализатор с входната програма за изчисляване на числата на Фибоначи (файла Fib.txt).

4. Да се тества работата на синтактичния анализатор със следните входни програми:

а) Програма, която извежда сумата на целите числа от 1 до 10

б) Програма, която извежда произведение на две цели числа, подадени като параметри на функция

в) Програма, която изчислява функция факториел за число, подадено като параметър на функцията