Дополнительные материалы о рекурсивном спуске и LL(1) РБНФ

Коновалов А.В.

17 апреля 2024 г.

Общий взгляд на рекурсивный спуск

Рекурсивный спуск — методика написания синтаксических анализаторов для LL(1)-грамматик в РБНФ на ЯП с поддержкой рекурсии.

Написание парсера выполняется в три этапа:

1. Составление LL(1)-грамматики в РБНФ — творческий этап.
2. Написание кода по правилам грамматики:
   - не восстанавливающийся анализатор — формальный этап,
   - восстанавливающийся при ошибках — тоже творческий.
3. В скелет парсера, полученный на этапе 2, добавляются семантические действия — творческий этап.

Рекомендации по рекурсивному спуску

Возвращаемые значения функций анализа нетерминалов

В псевдокоде, рассмотренном ранее, функции анализа нетерминалов ничего не возвращали.

На практике они могут возвращать атрибуты нетерминалов, например, абстрактное синтаксическое дерево, фрагмент сгенерированного кода или даже, в самых простых парсерах, значение выражения.

Также они могут принимать дополнительные параметры, вроде таблицы символов.

Начало разбора

Корректная программа является результатом раскрытия аксиомы — после прочтения аксиомы должен быть прочитан токен конца программы.

Кроме того, перед разбором аксиомы в переменную `Sym` должен быть помещён первый токен программы.

Поэтому псевдокод разбора рекурсивным спуском выглядит так (`S` — аксиома):

```
main() {
    Sym = NextToken();
    S();
    if (Sym ≠ EOP)
        ReportError();
}
```

Выражение приоритета и ассоциативности в LL(1) РБНФ

Выражение приоритета в LL(1) РБНФ

Приоритет выражается обычным образом — путём введения вспомогательных символов для каждого уровня приоритета. В классической грамматике арифметических выражений:

```
Expr   ::= Term (('+' | '-') Term)*.
Term   ::= Factor (('*' | '/') Factor)*.
Factor ::= Number | '(' Expr ')'.
```

приоритет сложения и вычитания ниже, чем у умножения и деления.

С ассоциативностью интереснее.

Выражение ассоциативности двуместных операций в LL(1) РБНФ

Ранее мы говорили, что левая ассоциативность в грамматике выражается при помощи левой рекурсии, а правая — при помощи правой:

- $X \to Y \mid X \bullet Y$ — левая ассоциативность,
- $X \to Y \mid Y \bullet X$ — правая ассоциативность.

Однако, оба правила не являются LL(1).

В последующих примерах считаем, что функции разбора нетерминалов сразу вычисляют значение выражения.

Левая ассоциативность двуместных операций

Левая ассоциативность выражается при помощи итерации:

```
X ::= Y ('•' Y)*.
```

Обоснование:

```
X() {
    value = Y();
    while (Sym.Tag == '•') {
        Sym = NextToken();
        value = value • Y();
    }
    return value;
}
```

Правая ассоциативность двуместных операций

Правая ассоциативность выражается при помощи правой рекурсии и опции:

```
X ::= Y ('•' X)?.
```

Обоснование:

```
X() {
    value = Y();
    if (Sym.Tag == '•') {
        Sym = NextToken();
        value = value • X();
    }
    return value;
}
```

Грамматика арифметических выражений с возведением в степень

```
Expr   ::= Term (('+' | '-') Term)*.
Term   ::= Factor (('*' | '/') Factor)*.
Factor ::= Base ('^' Factor)?.
Base   ::= Number | '(' Expr ')'.
```

Возведение в степень правоассоциативное и имеет больший приоритет, чем умножение и деление.

Префиксные одноместные операции

Префиксные одноместные операции фактически являются правоассоциативными (например, в C++: «`-*p`» — сначала выполнится разыменование указателя, потом смена знака), поэтому они тоже выражаются через правую рекурсию:

```
X ::= '•' X | Y.
```

Обоснование:

```
X() {
    if (Sym.Tag == '•') {
        Sym = NextToken();
        return •X();
    } else {
        return Y();
    }
}
```

Постфиксные одноместные операции

Постфиксные одноместные операции фактически являются левоассоциативными (например, в C++: «`a[i]++`» — сначала обращение по индексу, затем инкремент), поэтому они тоже выражаются через итерацию:

```
X ::= Y '•'*.
```

Обоснование:

```
X() {
    value = Y();
    while (Sym.Tag == '•') {
        Sym = NextToken();
        value = value•;
    }
    return value;
}
```

Пример: арифметические выражения с унарным минусом и факториалом (1)

Вариант, где префиксные операции имеют более высокий приоритет:

```
Expr    ::= Term (('+' | '-') Term)*.
Term    ::= Factor (('*' | '/') Factor)*.
Factor  ::= Postfix ('^' Factor)?.
Postfix ::= Prefix '!'*.
Prefix  ::= ('+' | '-') Prefix | Primary.
Primary ::= Number | '(' Expr ')'.
```

Нетерминал `Primary` выделен для наглядности, его можно встроить в правило для `Prefix`.

Пример: арифметические выражения с унарным минусом и факториалом (2)

Вариант, где постфиксные операции имеют более высокий приоритет:

```
Expr    ::= Term (('+' | '-') Term)*.
Term    ::= Factor (('*' | '/') Factor)*.
Factor  ::= Prefix ('^' Factor)?.
Prefix  ::= ('+' | '-') Prefix | Postfix.
Postfix ::= Primary '!'*.
Primary ::= Number | '(' Expr ')'.
```

Здесь тоже можно встроить нетерминалы `Postfix` и `Primary` в ущерб наглядности.

Пример: тернарная операция в Python

Тернарная операция в Python правоассоциативна — выражение

```
'плюс' if x > 0 else 'минус' if x < 0 else 'ноль'
```

интерпретируется как

```
'плюс' if x > 0 else ('минус' if x < 0 else 'ноль')
```

При записи правила грамматики кусок `if … else` рассматриваем как знак операции:

```
Cond = Expr ('if' Cond 'else' Cond)?.
```

Комментарии перед функциями разбора нетерминалов

Комментарии перед функциями разбора нетерминалов

Перед функциями разбора нетерминалов нужно обязательно добавлять комментарии с соответствующими правилами грамматики.[1]

```
/* Expr ::= ('+' | '-')? Term (('+' | '-') Term)*. */
Expr() {
    if (Sym == '+' || Sym == '-')
        Sym = NextToken();
    while (Sym == '+' || Sym == '-') {
        Sym = NextToken();
        Term();
    }
}
```

Иначе лабу не зачту.

[1] Чтобы подчеркнуть важность, на это отведён целый слайд.

Советы и грязные хаки

Совет: функция `expect` для анализа терминальных символов

Для невосстанавливающихся парсеров удобно определить функцию `expect`.

Однообразный код разбора терминальных символов можно поместить в функцию `expect(domain)`, которая либо потребляет токен и возвращает его атрибут, либо прерывает разбор с выдачей сообщения об ошибке:

```
expect(domain) {
    if (Sym.Tag == domain) {
        attr = Sym.Attr;
        Sym = NextToken();
        return attr;
    } else {
        throw SyntaxError(Sym.Coords, "Ожидался " + domain
            + ", однако получен " + Sym);
    }
}
```

Совет: функция `expect` в статически типизированном языке

В статически типизированном языке (C++, Java) базовый класс токена не содержит атрибутов, однако, его потомки могут содержать атрибуты, причём разных типов. Функция `expect` должна будет выполнять приведение типа токена.

```
template <class T = Token>
std::shared_ptr<T> expect(Domain domain) {
    if (Sym→Tag == domain) {
        std::shared_ptr<T> symT = std::dynamic_pointer_cast(Sym);
        Sym = NextToken();
        return symT;
    } else {
        using std::to_string;
        throw SyntaxError(Sym→Coords, "Ожидался " + to_string(domain)
             + ", однако получен " + to_string(Sym));
    }
}
```

Совет: раскрыть скобки (1)

Длину правила сокращает группировка нескольких операций одного приоритета:

```
Term ::= Factor (('*' | '/') Factor)*.
```

Однако, при написании кода удобнее скобки раскрыть:

```
Term ::= Factor ('*' Factor | '/' Factor)*.
```

В этом случае код разбора, выполняющий вычисление атрибутов нетерминалов, оказывается лаконичнее.

Совет: раскрыть скобки (2) — пример

```
/* Term ::= Factor ('*' Factor | '/' Factor)*. */
Term() {
    value = Factor();
    while (Sym.Tag == '*' || Sym.Tag == '/') {
        if (Sym.Tag == '*') {
            expect('*');
            value = value * Factor();
        } else {
            expect('/');
            value = value / Factor();
        }
    }
    return value;
}
```

Грязный хак: арифметические выражения с присваиваниями

Задача: написать программу-калькулятор, которая допускает арифметические выражения и присваивания значений переменным:

```
2 + 2 * 2
6
x = 5 + 2
7
x * 3 + 1
22
```

Первая попытка — точная, но неудачная

Если считать, что «`VAR '='`» — необязательная часть оператора, то мы получим грамматику, которую невозможно разобрать рекурсивным спуском:

```
Statement ::= (VAR '=')? Expr.
Expr      ::= ('+' | '-')? Term ('+' Term | '-' Term)*.
Term      ::= Factor ('*' Factor | '/' Factor)*.
Factor    ::= NUMBER | VAR | '(' Expr ')'.
```

По первому токену-переменной невозможно принять решение о разборе «`(VAR '=')?`» — см. «`x = 5 + 2`» и «`x * 3 + 1`».

Первая попытка — точная, но неудачная

Если считать, что «`VAR '='`» — необязательная часть оператора, то мы получим грамматику, которую невозможно разобрать рекурсивным спуском:

```
Statement ::= (VAR '=')? Expr.
Expr      ::= ('+' | '-')? Term ('+' Term | '-' Term)*.
Term      ::= Factor ('*' Factor | '/' Factor)*.
Factor    ::= NUMBER | VAR | '(' Expr ')'.
```

По первому токену-переменной невозможно принять решение о разборе «`(VAR '=')?`» — см. «`x = 5 + 2`» и «`x * 3 + 1`».

В данном случае можно разбирать грамматику как LL(2) — заглядывать на два токена вперёд. Но этот подход не обобщается на массивы произвольной глубины, например,

```
a[1][2][3] = 7
a[1][2][3] + 1
```

Грязный хак Страуструпа

Однако, в книге *Б. Страуструп Язык программирования C++. Специальное издание. Пер. с англ. — М.: ООО «Бином-Пресс», 2005 г. — 1104 с.: ил.* для такого калькулятора был написан парсер рекурсивным спуском.

Как Страуструпу это удалось?

Грязный хак Страуструпа

Однако, в книге *Б. Страуструп Язык программирования C++. Специальное издание. Пер. с англ. — М.: ООО «Бином-Пресс», 2005 г. — 1104 с.: ил.* для такого калькулятора был написан парсер рекурсивным спуском.

Как Страуструпу это удалось?

Он сделал присваивание операцией, причём с наивысшим приоритетом!

«Хакнутая» грамматика (1)

```
Expr   ::= ('+' | '-')? Term ('+' Term | '-' Term)*.
Term   ::= Factor ('*' Factor | '/' Factor)*.
Factor ::= NUMBER | VAR ('=' Expr)? | '(' Expr ')'.
```

Однозначная ли это грамматика?

«Хакнутая» грамматика (1)

```
Expr   ::= ('+' | '-')? Term ('+' Term | '-' Term)*.
Term   ::= Factor ('*' Factor | '/' Factor)*.
Factor ::= NUMBER | VAR ('=' Expr)? | '(' Expr ')'.
```

Однозначная ли это грамматика?

Нет. Выражение «`x = 5 + 2`» можно проинтерпретировать двумя способами:

- и как «`x = (5 + 2)`»,
- и как «`(x = 5) + 2`».

«Хакнутая» грамматика (2)

```
Expr   ::= ('+' | '-')? Term ('+' Term | '-' Term)*.
Term   ::= Factor ('*' Factor | '/' Factor)*.
Factor ::= NUMBER | VAR ('=' Expr)? | '(' Expr ')'.
```

Однако, по ней можно написать работающий парсер методом рекурсивного спуска, который будет правильно понимать выражения из постановки задачи.

Парсер выражение «`x = 5 + 2`» будет понимать как «`x = (5 + 2)`» — знак «`=`» будет «съедать» выражение до конца.

«Хакнутая» грамматика (3)

```
Expr   ::= ('+' | '-')? Term ('+' Term | '-' Term)*.
Term   ::= Factor ('*' Factor | '/' Factor)*.
Factor ::= NUMBER | VAR ('=' Expr)? | '(' Expr ')'.
```

Заметим, что в этой грамматике также окажутся допустимы выражения вида

```
2 + 3 * x = 4 - 5 * (6 / y = 7 - 1)
```

Поскольку присваивание съедает остаток выражения, оно проинтерпретируется так:

```
2 + 3 * x = (4 - 5 * (6 / y = (7 - 1)))
```