Алгоритм рекурсивного спуска: краткое введение

29 мая 2019 г.

1 Грамматика

Определение 1.1. Грамматикой называется упорядоченная четвёрка $\langle A_{\text{Н}}, A_{\text{Т}}, R, S\rangle$, в которой указаны:

1. два непересекающихся алфавита (терминальных $A_{\text{Т}}$ и нетерминальных $A_{\text{Н}}$ символов);
2. множество правил R вида
$$\alpha \to \beta$$
где α и β — строки из алфавита $A_{\text{Н}} \cup A_{\text{Т}}$, причём хотя бы один символ из α — нетерминальный;
3. начальный символ $S \in A_{\text{Н}}$.

В данной теме мы будем рассматривать только бесконтекстные грамматики: то есть, все правила имеют следующий вид:
$$x \to \beta$$
где $x \in A_{\text{Н}}$, то есть слева от стрелки всегда указан ровно один символ, и этот символ — нетерминальный.

Будем говорить, что строка α получается из строки β применением правила $x \to \sigma$, если в строку β входит символ x, и замена одного из его вхождений на σ даёт α.

Например, пусть $A_{\text{Н}} = \{(,)\}$ (символы круглых скобок), а $A_{\text{Т}} = \{X\}$ (большая буква X). И пусть некоторое правило имеет вид

$$X \quad \longrightarrow \quad (X)\ X$$

Тогда в строке $(X)X$ это правило можно применить двумя способами: заменив первое либо второе вхождение X:

Заменяемое вхождение (подчёркнуто)	результат подстановки
$(\underline{X})X$	$((X)X)X$
$(X)\underline{X}$	$(X)(X)X$

Делать замены можно многократно, например, $(X)(X)X$ может быть получена из строки X за два применения данного правила:

$$\underline{X} \Rightarrow (X)\underline{X} \Rightarrow (X)(X)X$$

Пусть теперь заданы два правила (напомним, что через ε мы обозначаем пустую строку):

$$\begin{aligned} X &\longrightarrow (X)\ X \\ X &\longrightarrow \varepsilon \end{aligned}$$

С их помощью теперь мы можем построить строку, содержащую любую правильную скобочную запись. Например, за 9 подстановок мы получим строку ()(())():

$$\underline{X} \Rightarrow (X)\underline{X} \Rightarrow (X)(X)\underline{X} \Rightarrow (X)(X)(X)\underline{X} \Rightarrow$$
$$\Rightarrow (X)(\underline{X})(X) \Rightarrow (X)((\underline{X})X)(X) \Rightarrow (X)(()\underline{X})(X) \Rightarrow (\underline{X})(())(X) \Rightarrow$$
$$\Rightarrow ()(())(\underline{X}) \Rightarrow ()(())()$$

Что более интересно, никакие неправильные скобочные записи с помощью этих правил быть построены не могут.

В итоге мы приходим к определению:

Определение 1.2. Множество строк $\mathcal{L}$ задаётся некоторой грамматикой, если:

1. любая строка из него может быть получена из начального символа путём последовательного применения правил грамматики;
2. никакие другие строки не могут быть получены таким образом.

1.1 Запись грамматик: сокращения

Часто бывает так, что мы имеем более одного правила для нетерминала. В этом случае мы будем использовать скобки и знак вертикальной черты («альтернатива»). Например, грамматику

$$\begin{aligned} N &\longrightarrow 0N \\ N &\longrightarrow 1N \\ N &\longrightarrow 2N \\ N &\longrightarrow 3N \\ N &\longrightarrow \varepsilon \end{aligned}$$

мы могли бы переписать так:

$$N \longrightarrow (0 \mid 1 \mid 2 \mid 3)\ N$$

В общем же случае запись выглядит так:

$$Z \longrightarrow \gamma\ (\alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n)\ \delta$$

Данная запись означает, что нетерминал Z может быть преобразован в строку вида $\gamma\alpha_i\delta$: мы указали общие начальные и конечные части получающейся строки (γ и δ), а для середины перечислили возможные варианты (α_1, α_2,..., α_n).

Такая запись удобна, но, поскольку скобки и вертикальные чёрточки могут быть и терминальными символами, надо быть внимательным. Чтобы исключить ошибки, мы будем брать терминальные символы в кавычки и/или выделять их шрифтом. Например, грамматику для правильных скобочных записей из прошлого раздела мы будем записывать так:

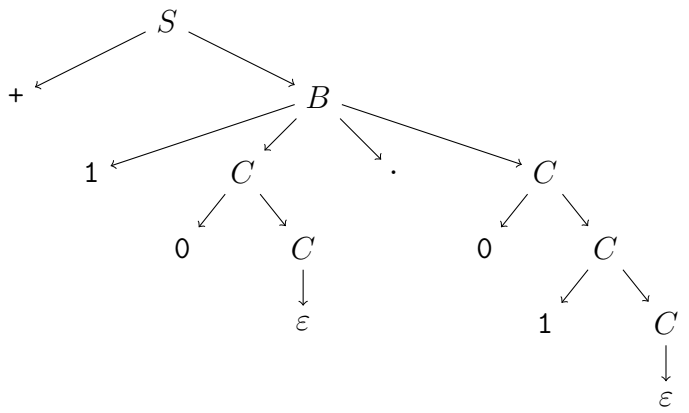
$$X \longrightarrow (\texttt{'('}X\texttt{')'}X) \mid \varepsilon$$

2 Терминалы и нетерминалы в реальных строках

Терминальные символы увидеть просто, а вот что соответствует в реальных строках нетерминальным символам? Рассмотрим грамматику:

$$\begin{aligned} S &\longrightarrow \left(\texttt{'+'} \mid \texttt{'-'}\right) B \\ B &\longrightarrow \texttt{'1'}C\texttt{'.'}C \\ C &\longrightarrow (\,(0 \mid 1)\; C) \mid \varepsilon \end{aligned}$$

Рассмотрим строку `+10.01`, данная строка может быть порождена данной грамматикой, что следует из дерева разбора для неё:



Терминальным символам в этом дереве разбора соответствуют листья, а вот нетерминальным символам соответствует сразу несколько листьев: скажем, нетерминалу B соответствует почти весь текст (кроме первого символа `+`), а у нетерминала C есть пять подстрок, соответствующих ему:

`+1`$\underbrace{\texttt{0}}_{\text{№1}}$`.01, +10`$\underbrace{\texttt{}}_{\text{№2}}$`.01, +10.`$\underbrace{\texttt{01}}_{\text{№3}}$`, +10.0`$\underbrace{\texttt{1}}_{\text{№4}}$`, +10.01`$\underbrace{\texttt{}}_{\text{№5}}$

Заметьте, что варианты прочтения №2 и №5 пусты — данные нетерминалы в дереве разбора раскрылись в пустую подстроку.

3 Разбор текста

Если даны исходный нетерминальный символ и последовательность применения правил, мы легко получим итоговую строку. Обратная же задача — задача *разбора текста* — состоит в угадывании того, какие правила нужно применить, чтобы получить данную строку.

То есть, если мы по каждому символу строки можем указать, какие узлы дерева разбора его содержат, то мы решаем задачу разбора строки.

Если дан символ в строке, Проверить, что в строке `s` в позиции `n` находится терминальный символ `c` просто — достаточно сравнить реальный символ и ожидаемый:

```
if s.[n] = c then "Подходящий символ"
             else "Неправильный символ";;
```

Давайте научимся сопоставлять участкам строки нетерминальные символы.

Напишем для каждого нетерминального символа функцию, предназначенную для разбора соответствующего нетерминала: функцию, умеющую отвечать на вопрос: «где заканчивается нетерминальный символ, если он начинается в данной позиции?». При необходимости мы можем научить эту функцию в том числе и строить дерево разбора.

Такая функция берёт на вход целое число — некоторую позицию в строке, и возвращает другое целое число — первую позицию *после* разбираемого ею нетерминального символа. В случае, если есть несколько способов прочесть нетерминал, функция ведёт себя «жадно» и выбирает самое длинное прочтение.

```
parse_n: int -> int
```

Помимо новой позиции, функция может возвращать что-то ещё, например, построенное синтаксическое дерево.

```
parse_n: int -> tree * int
```

Если функция вызвана в позиции, начиная с которой данной нетерминальный символ прочесть нельзя, поведение её может быть сколь угодно странным.

3.1 Пример

Напомним, если рассмотреть дерево из прошлого раздела, нетерминал C в дереве разбора присутствует в пяти местах, соответственно, соответствует ему пять подстрок. Функция `parse_c` в соответствии с соглашением должна по началу каждой такой подстроки находить её конец:

№	Подстрока	интервал её индексов	результат вызова `parse_c`
1	`+10.01`	$[2, 3)$	`parse_c 2 = 3`
2	`+10.01`	$[3, 3)$	`parse_c 3 = 3`
3	`+10.01`	$[4, 6)$	`parse_c 4 = 6`
4	`+10.01`	$[5, 6)$	`parse_c 5 = 6`
5	`+10.01`	$[6, 6)$	`parse_c 6 = 6`

3.2 Как реализовать разбор нетерминала

Начнём с функции `parse_c`. Рассмотрим все правила грамматики, преобразующие нетерминал C:

$$C \longrightarrow (\,(0 \mid 1)\; C) \mid \varepsilon$$

Рассмотрим эти правила: нетерминал C — это:

1. либо символ 0 или 1, за которым снова идёт нетерминал C;
2. либо пустая строка.

Значит, при разборе нетерминала надо разобрать эти два случая:

1. либо в данной позиции находится 0 или 1 — тогда надо вернуть результат рекурсивного вызова C со следующей позиции;
2. либо в данной позиции что-то другое — тогда надо вернуться из вызова сразу.

Из этого сразу получается требуемый код (здесь `str` — это строка, которую мы разбираем, она известна из контекста):

```
let rec parse_c pos =
    if str.[pos] = '0' || str.[pos] = '1' then parse_c (pos + 1)
    else pos;;
```

3.3 Реализация разборов остальных нетерминалов

Теперь рассмотрим функцию `parse_b`:

$$B \longrightarrow {'1'}C{'.'}C$$

Надо убедиться, что первый символ — это 1, затем идёт нетерминал C, затем точка, а затем — ещё раз C. Как мы уже знаем, проверка терминалов 1 и точки может быть сделана прямо, а разбор нетерминала C выполнит вызов функции `parse_c`:

```
let rec parse_b pos =
    if str.[pos] <> '1' then failwith "ожидалось 1";
    let pos1 = parse_c (pos + 1) in
    if str.[pos1] <> '.' then failwith "ожидалась точка";
    parse_c (pos1 + 1);;
```

Обратите внимание на работу с позициями строки: раз начальный символ 1 находится в позиции `pos`, то первый нетерминал C начинается в позиции `pos+1`, точка находится в первой позиции после C (эту позицию возвращает `parse_c`), и второй нетерминал C — в позиции сразу за точкой.

Нам осталась грамматика и функция для S:

$$S \longrightarrow \left('+' \,|\, '-'\right) B$$

И осталось написать функцию для него:

```
let rec parse_s pos =
    if str.[pos] = '+' || str.[pos] = '-' then parse_b (pos + 1)
    else failwith "Ожидался символ '+' или '-'"
```

3.4 Реализация разбора в целом

Нам осталось организовать разбор — соединить все функции вместе и добавить к ним «обвязку»: сформировать контекст (строка `str` где-то должна быть определена), сделать правильный вызов разбора начального нетерминала (в нашем примере это S) и правильно проанализировать результат. В данном разделе мы напишем функцию, проверяющую, является ли строка на входе построенной по указанной грамматике.

Мы сделаем это с помощью *взаимной рекурсии* — особой формы рекурсии, где несколько функций вызывают друг друга в хаотичном порядке. Такие функции должны быть определены рядом с помощью ключевого слова `and`:

```
let rec is_odd n =
    if n > 0 then is_even (n-1)
    else false
and is_even n =
    if n > 0 then is_odd (n-1)
    else true;;
```

Приведённые выше два определения взаимно-рекурсивных функций позволяют определить чётность числа: убедитесь, что `is_odd 4 = false` и `is_even 2 = true`.

Итак, определим главную функцию `parse` для «разбора вообще», эта функция будет содержать в себе определения всех функций для разбора нетерминалов и проверки результата работы:

```
let parse s =
    let str = s ^ ";" in

    let rec parse_c pos = ... in
    and parse_b pos = ... in
    and parse_s pos = ... in

    parse_s pos = String.length str - 1;;
```

Обратим внимание на несколько важных особенностей функции `parse`:

1. Переменная `str` — которую используют все функции, разбирающие отдельные нетерминалы — это входная строка, к которой добавлен знак точки с запятой в конце. Данный дополнительный символ служит ограничителем для разбора.

В самом деле, если разбор дошёл до конца строки, но в строке нет никакого ограничителя (в рассматриваемом примере для второго нетерминала C такого ограничителя нет), мы будем двигаться вперёд, пока не обратимся к символу за концом строки — и в результате не получим ошибку.

Можно решать эту проблему добавлением во все функции для нетерминалов проверок — но более простым решением является добавление в конец строки символа, заведомо отсутствующего в грамматике. Тогда рекурсивный спуск, дойдя до этого символа, неизбежно остановится: проследите по функциям, что это будет именно так.

2. Мы должны убедиться, что строка действительно полностью разобрана: вполне возможно, во входной строке есть ошибочные символы, из-за которых разбор остановится раньше. Для этого мы вызовем функцию для начального нетерминала S и убедимся, что она остановилась на последнем символе (на точке с запятой).

3. Если же на вход программе дадут строку, не содержащую каких-то требуемых грамматикой символов (например, `"+10"`), то разбор, наткнувшись на неподходящий символ (в крайнем случае — на ограничивающую строку точку с запятой), закончит свою работу с исключением.

4 Разбор леворекурсивной грамматики

4.1 Ассоциативность операций и рекурсивность грамматики

Определение 4.1. Операция $\triangle$ — *ассоциативная*, если для любых значений a, b и c выполнено $(a \triangle b) \triangle c = a \triangle (b \triangle c)$.

Нетрудно видеть, что для ассоциативной операции порядок расстановки скобок не важен. Например, операция сложения для натуральных чисел — ассоциативна: $(2+3)+5 = 2+(3+5)$, поэтому мы обычно записываем эту сумму как $2 + 3 + 5$.

Однако, не все операции ассоциативны: например, $(2 - 3) - 5 \neq 2 - (3 - 5)$, и расстановку скобок для правильного вычисления результата знать необходимо. Если же скобок не расставлено, может быть определён способ чтения выражения «по умолчанию». Например, для вычитания скобки расставляются слева направо:

$$2 - 3 - 5 - 7 = ((2 - 3) - 5) - 7$$

А вот для возведения в степень скобки расставляют справа налево:

$$2^{3^{5^7}} = 2^{\left(3^{\left(5^7\right)}\right)}$$

Определение 4.2. Если дана неассоциативная операция $\triangle$, и выражение $a \triangle b \triangle c$ читается по умолчанию как $(a \triangle b) \triangle c$, такая операция называется *левоассоциативной*. Если же выражение читается как $a \triangle (b \triangle c)$, то тогда такая операция — правоассоциативная.

Грамматика легко позволяет задавать различную ассоциативность операций: например, левоассоциативное вычитание можно записать так:

$$X \longrightarrow X - T \mid T$$

а правоассоциативное возведение в степень — так:

$$X \longrightarrow T\hat{}X \mid T$$

Соответственно, грамматика для левоассоциативной операции естественным образом называется леворекурсивной (поскольку «рекурсивный» нетерминал указывается слева от операции), а для правоассоциативной операции — праворекурсивной.

Для завершённости рассмотрения вопроса заметим, что в программировании нет в точном смысле ассоциативных операций. Вычисление — это некоторый процесс, он может прерваться в произвольный момент (например, в силу переполнения или деления на 0). Поэтому даже ассоциативные операции часто имеют некоторый предпочтительный порядок вычислений. Например, плюс или умножение часто считают левоассоциативными.

4.2 Разбор леворекурсивной грамматики

Пусть дана некоторая леворекурсивная грамматика:

$$X \longrightarrow X - T \mid T$$

К сожалению, алгоритм рекурсивного спуска в чистом виде не даёт способа разбора такой грамматики. Ведь если мы напишем по этой грамматике функцию разбора, то она никогда не закончит свою работу.

```
let rec parse_x pos =
    let i = parse_x pos in (* вечный рекурсивный вызов *)
    match s.[pos] with
    ...
```

Поэтому требуется придумать какой-нибудь трюк. Ниже приводится один из таких трюков.

Что нам требуется? Рассмотрим, какого вида строчки мы хотели бы разбирать:

$$\begin{aligned} X \longrightarrow\ & T \\ \mid\ & T - T \\ \mid\ & T - T - T \\ \mid\ & \ldots \end{aligned}$$

Здесь мы видим перечисление нескольких нетерминалов T, разделённых знаком минуса. То есть, мы должны разбирать нетерминалы T, пока после каждого T идёт минус. Если после очередного вхождения T не идёт минуса, то, значит, нетерминал X закончился.

Вот какой код может получиться из этой идеи:

```
let rec parse_x pos =
    parse_x_impl (parse_t pos)

and parse_x_impl pos =
    match s.[pos] with
        '-' -> parse_x_impl (parse_t (pos + 1))
      | _ -> pos
```

Понятно, что данную функцию несложно дополнить параметрами и вызовами, позволяющими выполнять разбору какое-то полезное действие — строить дерево, вычислять выражение и т.п.