Абстрактный и конкретный синтаксис

Коновалов А.В.

29 марта 2023 г.

Абстрактный синтаксис

Абстрактный синтаксис (abstract syntax) — упрощённая грамматика языка, в которой отсутствует информация, гарантирующая построение уникальных деревьев вывода.

В отличие от конкретного синтаксиса, абстрактный синтаксис не предназначен для ни для вывода цепочек символов, принадлежащих языку, ни для синтаксического анализа этих цепочек.

Абстрактный синтаксис предназначен для описания *абстрактных синтаксических деревьев.*

В сущности, грамматику абстрактного синтаксиса можно считать разновидностью описания алгебраических типов данных.

Абстрактное синтаксическое дерево (1)

Абстрактное синтаксическое дерево или **дерево абстрактного синтаксиса** (abstract syntax tree) — структура данных, отражающая содержательную информацию о структуре программы. Узлами абстрактного синтаксического дерева служат конструкции языка программирования: внутренние узлы соответствуют операторам, их потомки — операндам.

В общем случае, любая конструкция языка программирования может быть обработана путём создания оператора для данной конструкции и рассмотрения семантически значащих компонентов конструкции в качестве операндов этого оператора.

Абстрактное синтаксическое дерево (2)

В отличие от конкретного синтаксического дерева (дерева вывода или дерева разбора) абстрактное синтаксическое дерево не включает вспомогательные языковые средства: скобки для группировки, разделители вроде запятых или точек с запятой и т.д.

Также в абстрактном синтаксическом дереве отсутствуют вспомогательные нетерминалы, используемые для указания приоритета.

Далее прилагательное *абстрактное* мы будем опускать. Конкретные синтаксические деревья мы будем называть деревьями вывода или деревьями разбора.

Примеры описаний абстрактного синтаксиса (1)

2. Language

In this section, we describe the language which will be used throughout this paper.

DEFINITION 2.1 (Language). The language for which the described transformations are to be performed is a simple higher-order functional language as shown in Figure 3.

prog	::=	e_0 **where** $f_1 = e_1 \ldots f_n = e_n$	Program
e	::=	v	Variable
		$\mid c\ e_1 \ldots e_n$	Constructor
		$\mid f$	Function Call
		$\mid \lambda v.e$	λ-Abstraction
		$\mid e_0\ e_1$	Application
		$\mid$ **case** e_0 **of** $p_1 : e_1 \mid \cdots \mid p_k : e_k$	Case Expression
p	::=	$c\ v_1 \ldots v_n$	Pattern

Figure 3. Language Grammar

Скриншот из статьи *Hamilton, Geoff W. "Distillation: extracting the essence of programs." ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (2007).*
https://doi.org/10.1007/978-3-642-11486-1_13

Примеры описаний абстрактного синтаксиса (2)

3.1 Syntax

Definition 3. Assume denumerable disjoint sets of symbols for constructors $\mathcal{C}$, functions $\mathcal{F}$, pattern functions $\mathcal{G}$ (ranged over by c, f, and g, respectively), and variables $\mathcal{X}$ (ranged over by x, y). Then the set of programs $\mathcal{Q}$, definitions $\mathcal{D}$, terms $\mathcal{T}$, and patterns $\mathcal{P}$ (ranged over by q, d, t, and p, respectively) are defined by the abstract syntax grammar

$$
\begin{array}{lll}
\text{(program)} & \mathcal{Q} \ni q ::= d^m & \text{(definitions)} \\
\text{(definition)} & \mathcal{D} \ni d ::= f\, x^n = t \mid (g\, p.\, x^n = t.)^m & \text{(function/matcher)} \\
\text{(pattern)} & \mathcal{P} \ni p ::= c\, x^n & \text{(flat pattern)} \\
\text{(term)} & \mathcal{T} \ni t ::= x \mid c\, t^n \mid f\, t^n \mid g\, t^m & \\
 & & \text{(variable/construction/application/match)} \\
\text{(value)} & \mathcal{V} \ni v ::= c\, v^n &
\end{array}
$$

where $n \geq 0$ and $m > 0$. We require that

Скриншот из статьи *Secher, J.P. (2001). Driving in the Jungle. In: Danvy, O., Filinski, A. (eds) Programs as Data Objects. PADO 2001. Lecture Notes in Computer Science, vol 2053. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-44978-7_12*

Примеры описаний абстрактного синтаксиса (3)

Как видно из приведённых примеров, описание абстрактного синтаксиса зачастую неформально и сочетает в себе черты математической нотации.

Абстрактный синтаксис часто подразумевает повторение некоторых дочерних конструкций несколько раз, что часто выражается через многоточие или (как в примере из статьи Secher’a) — через степень.

В абстрактном синтаксическом дереве (в его программном воплощении) повторению однородных дочерних узлов могут соответствовать массивы/списки соответствующего типа.

Абстрактный синтаксис и синтаксическое дерево

Абстрактный синтаксис определяет типы узлов синтаксических деревьев и допустимые типы потомков для каждого из узлов.

Пример. Грамматика абстрактного синтаксиса с двумя типами узлов для выражений — арифметических и логических:

```
ArithExpr → NUMBER
          | ArithExpr BinOp ArithExpr
          | - Expr
          | BoolExpr ? ArithExpr : ArithExpr
BoolExpr → TRUE | FALSE
         | ArithExpr RelOp ArithExpr
         | BoolExpr BoolOp BoolExpr
         | NOT BoolExpr
BinOp → + | - | * | /
RelOp → = | ≠ | > | < | ≤ | ≥
BoolOp → AND | OR
```

Представление синтаксического дерева в программе (1)

В объектно-ориентированных языках нетерминалам абстрактного синтаксиса соответствуют абстрактные базовые классы (или интерфейсы), продукциям — конкретные классы-наследники. Атрибутами конкретных классов являются семантически значимые компоненты продукций.

В примере выше нам потребуются абстрактные классы `ArithExpr` и `BoolExpr` для представления выражений, типы данных для `BinOp`, `RelOp` и `BoolOp` можно представить при помощи перечислений.

Для продукции для тернарного оператора `ArithExpr → BoolExpr ? ArithExpr : ArithExpr` будет написан класс, полями которого будет одно логическое и два арифметических выражения. Знаки `?` и `:` суть разделители, используемые для удобства записи.

Представление синтаксического дерева в программе (2)

В статически типизированных функциональных языках (Haskell, Standard ML и др.) для описания абстрактного синтаксиса разумно использовать алгебраические типы данных — нетерминалы соответствуют типам, продукции — конструкторам типов.

В объектно-функциональном языке Scala синтаксическое дерево естественно выражается при помощи case-классов.

Переход от абстрактного синтаксиса к конкретному

1. Абстрактный синтаксис описывает синтаксическое дерево — его нужно преобразовать в грамматику, способную порождать цепочки символов данного языка, пусть даже и неоднозначную.
2. Написать вспомогательные нетерминалы + правила, описывающие итерацию при помощи рекурсии.
3. Написать вспомогательные нетерминалы + правила для выражения приоритета и ассоциативности.

Приоритет и ассоциативность (1)

Рассмотрим ситуацию, когда в выражении две операции op_1 и op_2 имеют общий операнд b:

a op_1 b op_2 c

Тогда, если в языке операции op_1 и op_2 имеют разный **приоритет**, «спорный» аргумент b достанется той из них, чей приоритет выше.

В случае операций с равным приоритетом решение принимается на основе соответствующей их уровню приоритета **ассоциативности**: b принадлежит op_1 в случае **левой** ассоциативности, op_2 — в случае **правой.**

Приоритет и ассоциативность (2)

Левой ассоциативностью обладает, например, операция вычитания:

$$10 - 5 - 2 = (10 - 5) - 2 = 3 \quad \neq \quad 10 - (5 - 2) = 7$$

Правой ассоциативностью обладает, например, операция присваивания в Си (`a = b = c` эквивалентно `a = (b = c)`) или операция возведения в степень в Python:

```
>>> 2 ** 3 ** 2
512
```

Если операцию нельзя объединять в цепочку вида `a op b op c`, то говорят, что операция **неассоциативна**. Например, в Паскале неассоциативны операции отношения, т.е. выражение вида `a > b > c` синтаксически некорректно.

Пример перехода к конкретному синтаксису (1)

Пусть мы имеем следующий абстрактный синтаксис:

$$
\begin{aligned}
Expr \rightarrow\ & Expr + Expr \mid Expr - Expr \mid \\
& \mid Expr * Expr \mid Expr / Expr \\
& \mid Expr \hat{} Expr \\
& \mid v \\
& \mid f(Expr_1, \ldots, Expr_N)
\end{aligned}
$$

Здесь v и f — имена переменных и функций, идентификаторы, $\hat{}$ — операция возведения в степень.

В спецификации языка говорится, что вызов функции должен принимать как минимум один параметр (т.е. $N > 0$) и для указания порядка выполнения операций можно использовать круглые скобки.

Пример перехода к конкретному синтаксису (2)

Запишем (неоднозначную) грамматику, позволяющую выводить строки символов рассматриваемого языка. Повторение параметров пока напишем условно:

```
Expr → Expr + Expr | Expr - Expr
     | Expr * Expr | Expr / Expr
     | Expr ^ Expr
     | IDENT
     | IDENT(Expr, …, Expr)
     | (Expr)
```

В грамматику мы добавили круглые скобки, которые можно использовать для указания порядка действий.

Пример перехода к конкретному синтаксису (3)

Один или более параметров функции изобразим при помощи вспомогательного нетерминала `Args` с рекурсивным правилом:

```
Expr → Expr + Expr | Expr - Expr
     | Expr * Expr | Expr / Expr
     | Expr ^ Expr
     | IDENT
     | IDENT(Args)
     | (Expr)
Args → Expr | Args, Expr
```

Пример перехода к конкретному синтаксису (4) — приоритет

Разберёмся с приоритетами операций, игнорируя пока ассоциативность.

В рассматриваемом языке наивысшим приоритетом обладает возведение в степень, затем следуют умножение и деление, затем сложение и вычитание.

Для простоты разберёмся сначала со сложением и вычитанием. Операндами + и – у нас могут быть любые выражения, операндами операций `*`, `/` и `^` выражения вида `a + b` и `a - b` быть не могут (если они, конечно, не заключены в скобки).

Разделим нетерминал `Expr` на два: $\texttt{Expr}_1$, который может содержать любые операции, и $\texttt{Expr}_2$, который операции + и – содержать не может.

Пример перехода к конкретному синтаксису (5) — приоритет

Любое выражение $\texttt{Expr}_2$ будет выражением $\texttt{Expr}_1$. Это можно выразить правилом $\texttt{Expr}_1 \rightarrow \texttt{Expr}_2$.

Операндами операций + и – могут быть выражения, содержащие любые знаки операций, т.е. $\texttt{Expr}_1$. Это можно выразить правилами

$$\texttt{Expr}_1 \rightarrow \texttt{Expr}_1 \texttt{ + } \texttt{Expr}_1 \mid \texttt{Expr}_1 \texttt{ - } \texttt{Expr}_1$$

Выражения $\texttt{Expr}_1$ операндами операций *, / и ^ быть не могут. Поэтому для нетерминала $\texttt{Expr}_2$ просто повторяем исходные продукции для Expr, заменяя в них Expr на $\texttt{Expr}_2$ (кроме выражения в скобках — там должно остаться $\texttt{Expr}_1$).

Пример перехода к конкретному синтаксису (6) — приоритет

Получим следующую грамматику:

```
Expr₁ → Expr₁ + Expr₁ | Expr₁ - Expr₁ | Expr₂
Expr₂ → Expr₂ * Expr₂ | Expr₂ / Expr₂
      | Expr₂ ^ Expr₂
      | IDENT
      | IDENT(Args)
      | (Expr₁)
Args → Expr₁ | Args, Expr₁
```

В полученной грамматике выражения вида `a + b` и `a - b` не могут быть операндами операций `*`, `/` и `^`, следовательно, в выражениях вида `a + b * c` приоритет у `*` будет выше, чем у `+` (и аналогично для других операций).

Пример перехода к конкретному синтаксису (7) — приоритет

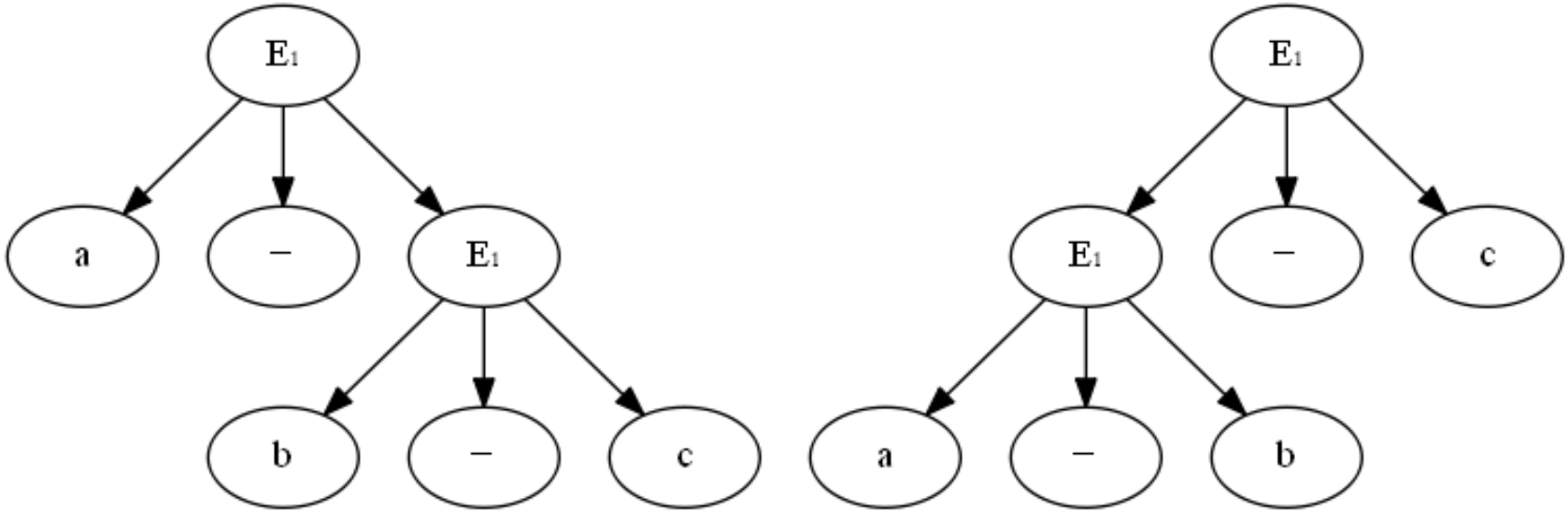
Повторяя те же рассуждения к операции ^ с наивысшим приоритетом (её операндами не могут быть выражения вида `a * b` или `a / b`), получим грамматику

```
Expr₁ → Expr₁ + Expr₁ | Expr₁ - Expr₁ | Expr₂
Expr₂ → Expr₂ * Expr₂ | Expr₂ / Expr₂ | Expr₃
Expr₃ → Expr₃ ^ Expr₃
      | IDENT
      | IDENT(Args)
      | (Expr₁)
Args → Expr₁ | Args, Expr₁
```

Таким образом, если операции в грамматике имеют различный приоритет, для каждого уровня приоритета нужно ввести вспомогательный нетерминал.

Пример перехода к конкретному синтаксису (8) — ассоциативность

Полученная грамматика по-прежнему неоднозначная. Для выражений вида `IDENT - IDENT - IDENT` можно построить два дерева разбора



Очевидно, правое дерево будет правильным, а левое — нет.

Пример перехода к конкретному синтаксису (9) — ассоциативность

Т.к. ассоциативность операции `-` — левая, то правым операндом не может быть выражение вида `b - c`.

А это означает, что правым операндом операций + и – (у них одинаковый приоритет и ассоциативность) не может быть выражение, содержащее + или –, т.е. правым операндом должно быть $\texttt{Expr}_2$.

Пример перехода к конкретному синтаксису (10) — ассоциативность

Получаем грамматику с правильной ассоциативностью для операций + и −:

```
Expr₁ → Expr₁ + Expr₂ | Expr₁ - Expr₂ | Expr₂
Expr₂ → Expr₂ * Expr₂ | Expr₂ / Expr₂ | Expr₃
Expr₃ → Expr₃ ^ Expr₃
      | IDENT
      | IDENT(Args)
      | (Expr₁)
Args → Expr₁ | Args, Expr₁
```

Пример перехода к конкретному синтаксису (11) — ассоциативность

Аналогично исправляем ассоциативность для умножения и деления:

```
Expr₁ → Expr₁ + Expr₂ | Expr₁ - Expr₂ | Expr₂
Expr₂ → Expr₂ * Expr₃ | Expr₂ / Expr₃ | Expr₃
Expr₃ → Expr₃ ^ Expr₃
      | IDENT
      | IDENT(Args)
      | (Expr₁)
Args → Expr₁ | Args, Expr₁
```

Пример перехода к конкретному синтаксису (12) — ассоциативность

Ассоциативность операции возведения в степень ^ — правая. Из тех же соображений получаем, что её левым операндом не может быть выражение вида `a ^ b`. Для обеспечения этого вводим ещё один нетерминал $Expr_4$:

$$Expr_1 \to Expr_1 + Expr_2 \mid Expr_1 - Expr_2 \mid Expr_2$$
$$Expr_2 \to Expr_2 * Expr_3 \mid Expr_2 / Expr_3 \mid Expr_3$$
$$Expr_3 \to Expr_4 \text{ ^ } Expr_3 \mid Expr_4$$
$$Expr_4 \to IDENT \mid IDENT(Args) \mid (Expr_1)$$
$$Args \to Expr_1 \mid Args, Expr_1$$

Пример перехода к конкретному синтаксису (13)

Давайте теперь дадим более осмысленные имена нетенминалам (term — слагаемое, factor — сомножитель, base — основание степени):

```
Expr → Expr + Term | Expr - Term | Term
Term → Term * Factor | Term / Factor | Factor
Factor → Base ^ Factor | Base
Base → IDENT | IDENT(Args) | (Expr)
Args → Expr | Args, Expr
```

Можем сделать **вывод**, что левая ассоциативность соответствует левой рекурсии в правилах, правая — правой.

Ещё заметим, что в этой грамматике мы можем считать, что запятая имеет приоритет ещё ниже, чем сложение.

Переход от абстрактного синтаксиса к конкретному

Выводы:

- Повторение выражается рекурсией.
- Для каждого уровня приоритета вводится вспомогательный нетерминал.
- Левая ассоциативность соответствует в правилах левой рекурсии, правая — правой.