

# Синтаксис и семантика. Абстрактные синтаксические деревья

Косарев Дмитрий

матмех

9 и 16 сентября 2024 г.

Дата сборки: 8 сентября 2024 г.



Посмотрим на понятия *синтаксиса* и *семантики*.

Примеры языков и построение AST.

Операционная семантика (т.е. как написать интерпретатор?)

Синтаксис — это язык, на котором мы записываем что-либо

Семантика<sub>1</sub> — смысл, который имеем в виду, записывая слова на языке

Семантика<sub>2</sub> — смысл, конкретного предложения в языке

Интерпретация — сопоставление некоторому синтаксису некоторой семантики.

## Пример. Синтаксис арифметических выражений

Неформальное описание: *арифметические выражения с 4мя бинарными операциями (+, \*, −, / и стандартными приоритетами) от целых чисел и переменной  $x$*

Какие выражения являются или не являются арифметическими?

- $1 + 2 * 3$
- $(1 + 2) * (3 - 4)$
- $(-x)/2$
- $\frac{1 + 2 * x + x * x}{2}$
- $x^0 + 42$
- $3.1415$
- $*$

## Пример: Тип данных для языка арифметических выражений

Неформальное описание: *арифметические выражения с 4мя бинарными операциями (+, \*, −, / и стандартными приоритетами) от целых чисел и переменной x.*

```
type expr =  
  | Const of int  
  | Var_called_x  
  | Plus of expr * expr  
  | Asterisk of expr * expr  
  | Dash of expr * expr  
  | Slash of expr * expr
```

```
type op =  
  | Plus  
  | Asterisk  
  | Dash  
  | Slash  
type expr =  
  | Const of int  
  | Var_called_x  
  | Bin_op of op * expr * expr
```

Это другой, также вполне  
допустимый вариант

# Пример: формулы с целыми числами и кванторами

Неформальное описание:

- константы-числа и инфиксные бинарные операции ( $+$ ,  $/$ ,  $*$ ,  $-$ )
- кванторы  $\forall$ ,  $\exists$  и переменные ( $x$ ,  $y$ ,  $z$ , ...)
- (так называемые) *предикатные символы* для чисел ( $>$ ,  $<$ ,  $\leq$ ,  $\equiv$ ,  $\neq$ ), все бинарные
- логические связки:  $\vee$  — «или»,  $\wedge$  — «и»,  $\Rightarrow$  — «если ... то ...»,  $\neg$  — «отрицание»
- иногда добавляют логические константы:  $\top$  (истина, true, top) и  $\perp$  (ложь, false, bottom)

Примеры формул и не формул

- $\forall x \exists y (x > y)$
- $\exists x \forall y (x > y)$
- $\forall x \forall y ((x \equiv y) \Leftrightarrow (y \equiv x))$
- $\exists y (x > y)$
- $\exists x y (x + 1 > y)$
- $\exists 1 (2 > 1)$

## Пример: тип для формул

```
type var_name = char
```

```
type expr =
```

```
| Const of int
```

```
  (** 42 *)
```

```
| Var of var_name
```

```
  (** x *)
```

```
| Plus of expr * expr
```

```
  (** e1 + e2 *)
```

```
| Slash of expr * expr
```

```
  (** e1 / e2 *)
```

```
| ...
```

```
type phormula =
```

```
| Top
```

```
| Bottom
```

```
| For_all of var_name * phormula
```

```
  (**  $\forall x f$  *)
```

```
| Conj of phormula * phormula
```

```
  (**  $f \wedge g$  *)
```

```
| Negation of phormula
```

```
  (**  $\neg f$  *)
```

```
| GreaterThen of expr * expr
```

```
  (**  $e1 > e2$  *)
```

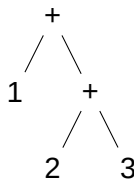
```
| ...
```

Когда будете делать допуск, нужно приписывать комментарии к объявлениям типов и функций в .mli файле

# Деревья абстрактного синтаксиса (англ. abstract syntax trees, AST)

Вспомним синтаксис арифметических выражений. Пример:  $1 + (2 + 3)$

```
type expr =  
  | Const of int  
  | Plus of expr * expr  
  | ...
```



Это выражение будет выглядеть в AST как

```
Plus (Const 1,  
      Plus (Const 2,  
            Const 3))
```

Мы абстрагировались в том числе от:

- префиксности и инфиксности
- приоритетов и скобочек
- конкретных позиций в файле



# Что может пойти не так с AST

- Может получиться AST не того языка
  - Рекомендуется в комментариях писать, что имелось в виду под тем или иным конструктором в AST

# Что может пойти не так с AST

- Может получиться AST не того языка
  - Рекомендуется в комментариях писать, что имелось в виду под тем или иным конструктором в AST
- Надо избегать случаев, когда некоторым значениям AST сложно придать смысл. Например:

```
type expr = Plus of expr * expr | Const of int | ...  
let x = Plus (Const 1, Plus (Const 2, Const 3))
```

(\* vs \*)

```
type expr = Plus of expr list | Const of int | ...  
let y = Plus [ Const 1; Const 2; Const 3 ]  
let z = Plus [ ]
```

# Что может пойти не так с AST

- Может получиться AST не того языка
  - Рекомендуется в комментариях писать, что имелось в виду под тем или иным конструктором в AST
- Надо избегать случаев, когда некоторым значениям AST сложно придать смысл.
- Надо избегать случаев, когда одну и ту же информацию можно представить двумя различными способами

```
type expr =  
| Plus of expr * expr  
| Op of op * expr list  
| ...
```

## Интерпретация : Синтаксис $\longrightarrow$ Семантика

В домашнем задании будут подзадачи

- Задать синтаксис (AST) мини-языка
- Сделать интерпретатор (задать «адекватную» семантику)
- ...

- Заданы в операционном стиле (*операционные семантики*)
  - назначение: смоделировать как что-то вычисляется
  - большого или малого шага
- В денотационном стиле для некоторого домена  $\mathcal{D}$  (*денотационные семантики*)
  - Объяснить смысл синтаксиса в уже известных терминах из  $\mathcal{D}$
- Всекие другие

# Операционная семантика для арифметических выражений (одна из)

```
type expr =  
  | Const of int  
  | Plus of expr * expr  
  | Slash of expr * expr
```

# Операционная семантика для арифметических выражений (одна из)

```
type expr =  
  | Const of int  
  | Plus of expr * expr  
  | Slash of expr * expr
```

У нас в синтаксисе присутствуют целые числа, но нам никто не запрещает считать их вещественными...

# Операционная семантика для арифметических выражений (одна из)

```
type expr =  
  | Const of int  
  | Plus of expr * expr  
  | Slash of expr * expr
```

У нас в синтаксисе присутствуют целые числа, но нам никто не запрещает считать их вещественными...

```
let rec interpret : expr -> float = function  
  | Const n -> float_of_int n  
  | Plus (l,r) -> (interpret l) +. (interpret r)  
  | Slash (l,r) -> (interpret l) /. (interpret r)
```

Возможное улучшение данной семантики:

```
interpret: expr -> float option
```



# Малого и большого шага



## Семантика **малого** шага

- каким-то образом демонстрирует промежуточные этапы

## Семантика **большого** шага

- выдает сразу результат
- более эффективная (но менее познаваемая), чем малого

P.S.

Там ниже дополнительные примеры про написание типов

- Числа
- Строки
- Массивы
- Объекты как набор пар «ключ-значение»

- Числа
- Строки
- Массивы
- Объекты как набор пар «ключ-значение»

```
type json =  
  | Null  
  | Number of float  
  | String of string  
  | List of json list  
  | Assoc of (string * json) list
```

## Пример про почту (1/2)

```
type contact =  
  { name : name  
    ; emailContactInfo : email_contact_info  
    ; postalContactInfo : postal_contact_info  
  }
```

## Пример про почту (1/2)

```
type contact =  
  { name : name  
    ; emailContactInfo : email_contact_info  
    ; postalContactInfo : postal_contact_info  
  }
```

Хочется, чтобы у контакта был *хотя бы один* адрес: либо электронной, либо физической почты.

## Пример про почту (1/2)

```
type contact =  
  { name : name  
    ; emailContactInfo : email_contact_info  
    ; postalContactInfo : postal_contact_info  
  }
```

Хочется, чтобы у контакта был *хотя бы один* адрес: либо электронной, либо физической почты.

Что вы думаете о вот таком?

```
type contact =  
  { name : name  
    ; email_contact_info : email_contact_info option  
    ; postal_contact_info : postal_contact_info option  
  }
```

## Пример про почту (2/2)

```
type contact_info =  
  | Email_only of email_contact_info  
  | Post_only of postal_contact_info  
  | Email_and_post of email_contact_info * postal_contact_info
```

```
type contact =  
  { name name  
    ; contactInfo : contact_info  
  }
```

Если, посмотрев на тип, сразу понятно какие состояния корректные, а какие нет, то это считается хорошим дизайном.

Пример взят [отсюда](#). [1]



## Содержательный пример (1/2) из [2]

```
type connection_state =  
  | Connecting  
  | Connected  
  | Disconnected
```

```
type connection_info =  
  { state: connection_state  
    ; server: inet_addr  
    ; last_ping_time: time option  
    ; last_ping_id: int option  
    ; session_id: string option  
    ; when_initiated: time option  
    ; when_disconnected: time option  
  }
```

## Содержательный пример (2/2)

```
type connecting = { when_initiated: time }
type connected =
  { last_ping : (time * int) option
  ; session_id: string
  }
type disconnected = { when_disconnected: time }

type connection_state =
| Connecting of connecting
| Connected of connected
| Disconnected of disconnected

type connection_info =
  { state: connection_state
  ; server: inet_addr
  }
```

- [1] Scott Wlaschin. *Designing with types: Making illegal states unrepresentable*. 2013. URL: <https://fsharpforfunandprofit.com/posts/designing-with-types-making-illegal-states-unrepresentable>.
- [2] Yaron Minsky. «OCaml for the masses». В: *Commun. ACM* 54.11 (2011), с. 53–58. DOI: 10.1145/2018396.2018413. URL: <https://queue.acm.org/detail.cfm?id=2038036>.