От токенов к абстрактному синтаксическому дереву

Лекция 2 — парсер и AST в PLY

Лазар В. И.

24 апреля 2025 г.

Где мы сейчас?

1. Лекция 1 — лексер: получили упорядоченный список токенов.
2. **Лекция 2 — строим AST и учимся его выполнять.**
3. Дальше — расширяем язык и пишем REPL.

Что такое AST?

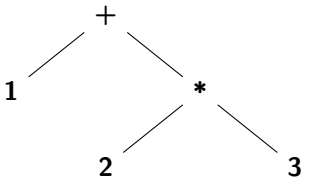
Определение

Абстрактное синтаксическое дерево (AST) — структурное представление программы, в котором опущены лишние скобки и пунктуация, а каждый узел описывает семантическую сущность (число, переменную, операцию...).

Почему не оставить «сырые» токены?

- Деревом проще анализировать и исполнять.
- Исчезают неоднозначности приоритетов.
- Можно легко добавлять новые фишки («гуляем» по дереву).

Пример 1: выражение $1 + 2 * 3$



В Python-коде это будет:

```
BinOp("+",
      Number(1),
      BinOp("*", Number(2), Number(3))
          )
```

Пример 2: условие

```
If(
  cond=BinOp("<", Identifier("x"), Number(0)),
  then_block=[Print(Identifier("x"))],
  else_block=[Print(Number(0))]
)
```

Важно: дерево отражает логику, а не текстовую форму.

Как PLY помогает строить AST

- Лексер (`lexer.py`) уже выдаёт токены.
- Теперь пишем правила `ply.yacc`: «что после чего идёт».
- Каждое правило — функция `p_...`, у которой `p[0]` возвращает узел.

Как PLY помогает строить AST

- Лексер (`lexer.py`) уже выдаёт токены.
- Теперь пишем правила `ply.yacc`: «что после чего идёт».
- Каждое правило — функция `p_...`, у которой `p[0]` возвращает узел.

```
from tinypy.ast_nodes import Number, BinOp

def p_expr_binop(p):
    """expr : expr PLUS expr
            | expr MINUS expr"""
    p[0] = BinOp(p[2], p[1], p[3])
```

Файл ast_nodes.py

```
from dataclasses import dataclass
class Node: pass
@dataclass
class Number(Node):
    value: float
@dataclass
class Identifier(Node):
    name: str
@dataclass
class BinOp(Node):
    op: str
    left: Node
    right: Node
```

Приоритеты и ассоциативность

В PLY они задаются таблицей:

```
precedence = (
  ('left', 'PLUS', 'MINUS'),
  ('left', 'STAR', 'SLASH'),
  ('right', 'POWER'),    # **
)
```

Это избавляет от ручного разбора вложенных выражений.

Отладка парсера

- `parser = yacc.yacc(debug=True)` создаёт файл `parsetab.py`.
- Команда `parser.parse(src, lexer=lexer)` возвращает корневой узел.
- Можно распечатать дерево функцией `print(ast)` — dataclass делает это красиво.

Быстрое исполнение AST

Идея: рекурсивно обойти дерево.

```
def eval_node(node, env):
    if isinstance(node, Number):
        return node.value
    if isinstance(node, Identifier):
        return env[node.name]
    if isinstance(node, BinOp):
        a = eval_node(node.left, env)
        b = eval_node(node.right, env)
        return a + b if node.op == '+' else a - b
```

‘env‘ — обычный словарь переменных.

Мини-пример «всё вместе»

```
src = "let x = 2 * 3; x + 1"
lexer.input(src)
ast = parser.parse(lexer=lexer)  # AST
env = {}
for stmt in ast:
    print(eval_node(stmt, env))
```

Практика к семинару 2

1. Завести файл `parser.py` по шаблону.
2. Реализовать правила для: числа, идентификатора, `+ - * /`, скобок.
3. Добавить таблицу приоритетов.
4. Написать функцию, печатающую полученный AST.
5. Запустить тесты `pytest tests/test_parser_basic.py` (появятся в репо).