Python Abstract Syntax Trees (AST)

Abdus Salam Azad

What is Abstract Syntax Tree

- Tree Representation of the source code
- Every “syntactic” details of the source code do not appear in the tree
  - hence “abstract”
  - represent “semantically meaningful” aspects
  - there are “concrete” syntax trees, which capture complete textual form
- “Parse” trees are typically “concrete” syntax tree

Over-simplified (Partial) Overview of Python Interpreter

Source Code

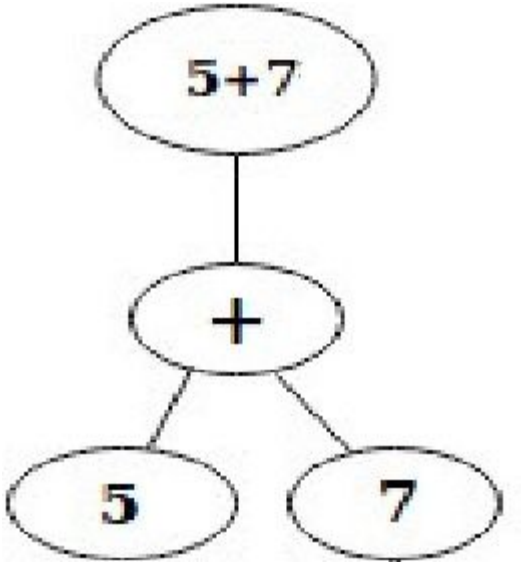
=> list of tokens

=> abstract syntax trees

=> bytecode

=> run the python code

AST for expressions: 5 + 7

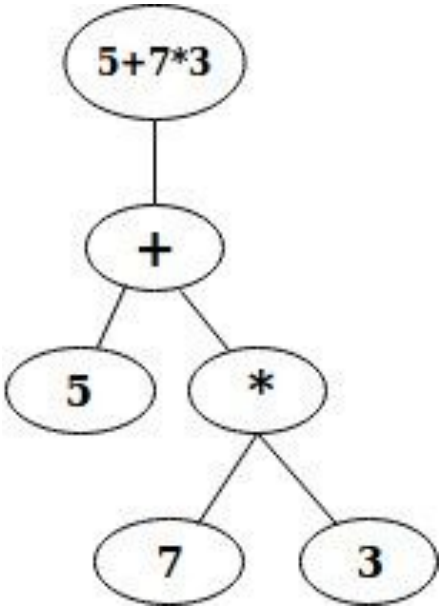


Abstract Visual Representation
(Not accurate)

- Module: {} 2 keys
  - body: {} 1 key
    - 0: {} 1 key
      - Expr: {} 1 key
        - value: {} 1 key
          - BinOp: {} 3 keys
            - left: {} 1 key
              - Constant: {} 1 key
                - value: 5
            - op: "Add"
            - right: {} 1 key
              - Constant: {} 1 key
                - value: 7
  - type_ignores: {} 0 keys

https://python-ast-explorer.com/

AST for expressions:



Abstract Visual Representation (Not accurate)

```
▼ Module: {} 2 keys
  ▼ body: {} 1 key
    ▼ 0: {} 1 key
      ▼ Expr: {} 1 key
        ▼ value: {} 1 key
          ▼ BinOp: {} 3 keys
            ▼ left: {} 1 key
              ▼ Constant: {} 1 key
                  value: 5
              op: "Add"
            ▼ right: {} 1 key
              ▼ BinOp: {} 3 keys
                ▼ left: {} 1 key
                  ▼ Constant: {} 1 key
                      value: 7
                op: "Mult"
                ▼ right: {} 1 key
                  ▼ Constant: {} 1 key
                      value: 3
  type_ignores: {} 0 keys
```

https://python-ast-explorer.com/

Python AST Examples - Try each one in the explorer

```
a = 1
```

```
b = a
```

```
c = a + b
```

```
print(c)
```

```
if c % 2 == 0:
     d = True
else:
     d = False
```

```
for i in range(5): print(i)
```

Generate ASTs from Pythons Source Code

- ast.parse(source)
  - source is a string => Python Source
  - https://docs.python.org/3/library/ast.html#ast.parse

Reading/Processing ASTs

class ast.**NodeVisitor**

A node visitor base class that walks the abstract syntax tree and calls a visitor function for every node found. This function may return a value which is forwarded by the `visit()` method.

This class is meant to be subclassed, with the subclass adding visitor methods.

visit(*node*)

Visit a node. The default implementation calls the method called `self.visit_classname` where *classname* is the name of the node class, or `generic_visit()` if that method doesn't exist.

generic_visit(*node*)

This visitor calls `visit()` on all children of the node.

Note that child nodes of nodes that have a custom visitor method won't be visited unless the visitor calls `generic_visit()` or visits them itself.

https://docs.python.org/3/library/ast.html#ast.NodeVisitor

Modifying ASTs

class `ast.`**`NodeTransformer`**

A `NodeVisitor` subclass that walks the abstract syntax tree and allows modification of nodes.

The `NodeTransformer` will walk the AST and use the return value of the visitor methods to replace or remove the old node. If the return value of the visitor method is `None`, the node will be removed from its location, otherwise it is replaced with the return value. The return value may be the original node in which case no replacement takes place.

From ASTs to Python Source

`ast.unparse(ast_obj)`

Unparse an `ast.AST` object and generate a string with code that would produce an equivalent `ast.AST` object if parsed back with `ast.parse()`.

- might throw an error (RecursionError) for highly complex ASTs

Executing ASTs

One can execute (the programs represented by) ASTS using the help of *compile* and *exec* function

```
exec(compile(<ast>, filename="", mode="exec"))
```

Modifying ASTs

`ast.`**`fix_missing_locations`**(*node*)

When you compile a node tree with `compile()`, the compiler expects `lineno` and `col_offset` attributes for every node that supports them. This is rather tedious to fill in for generated nodes, so this helper adds these attributes recursively where not already set, by setting them to the values of the parent node. It works recursively starting at *node*.

Resources Used/Acknowledgements

- https://courses.cs.washington.edu/courses/cse401/08wi/lecture/AST.pdf
- https://kamneemaran45.medium.com/python-ast-5789a1b60300

- https://www.journaldev.com/19243/python-ast-abstract-syntax-tree
- https://www.mattlayman.com/blog/2018/decipher-python-ast/
- https://kamneemaran45.medium.com/python-ast-5789a1b60300