Topic: Parser & Building an Abstract Syntax Tree

Course: Formal Languages & Finite Automata

Author: Cretu Dumitru and cudos to the Vasile Drumea with Irina Cojuhari

Theory

Parsers, based on lexers, are software components that analyze sequences of tokens produced by lexers to construct a parse tree or abstract syntax tree, representing the syntactic structure of the input code. Lexers, or tokenizers, preprocess the input by breaking it into tokens, which are meaningful units like keywords, identifiers, and operators. This two-step process allows parsers to focus on the syntax and structure of the code, facilitating error detection and code interpretation

Objectives:

1. Get familiar with parsing, what it is and how it can be programmed.
2. Get familiar with the concept of AST.
3. In addition to what has been done in the 3rd lab work do the following:
   i. In case you didn't have a type that denotes the possible types of tokens you need to:
      a. Have a type ***TokenType*** (like an enum) that can be used in the lexical analysis to categorize the tokens.
      b. Please use regular expressions to identify the type of the token.
   ii. Implement the necessary data structures for an AST that could be used for the text you have processed in the 3rd lab work.
   iii. Implement a simple parser program that could extract the syntactic information from the input text.

Implementation description

I have created the following parser on the basis of the ELSD Project:

```
import json,re
RULES={
    'COMMENT':["COMMENT"],
    'PATIENT_INFO':['LITERAL','OPERATOR','NUMERICAL'],
}


def parse(tokens):
    index=len(tokens)
    body = []
    found_rules=[]
    for rule in RULES:
        tokens_structure=" ".join([token['type'] for token in tokens])

        ans=[(m.start(0), m.end(0),rule) for m in re.finditer(" ".join(RULES[rule]),tokens_str
        found_rules.extend(ans)

    found_rules.sort()

    index=0
    for found in found_rules:
        rule_Set=RULES[found[2]]
        size=len(rule_Set)

        body.append({found[2]:tokens[index:index+size]})
        index+=size
    assert index==len(tokens), "Not all tokens were parsed, check if all tokens belong to rule

    return {"Type":"Program","body":body}
```

The way the program works is that the user must define all available structural rules in the RULES Array, then we iterate through each of the rules and identify them in the list of tokens we got from the tokenizer.

The index where each Rule was found is stored in an array then sorted, which returns the parser the sequential structure in the tokenizer.

```
  for rule in RULES:
        tokens_structure=" ".join([token['type'] for token in tokens])

        ans=[(m.start(0), m.end(0),rule) for m in re.finditer(" ".join(RULES[rule]),tokens_str
        found_rules.extend(ans)

    found_rules.sort()
```

After that the body of the AST is being built, based on the order of the rules found they are being added.

```
   index=0
    for found in found_rules:
        rule_Set=RULES[found[2]]
        size=len(rule_Set)

        body.append({found[2]:tokens[index:index+size]})
        index+=size
```

In case that not all tokens gottena s input from the lexer were used in the AST builder an error is asserted to verify the rules in RULES and their order

```
    assert index==len(tokens), "Not all tokens were parsed, check if all tokens belong to rule
```

Example usage of the Parser:

```
from src.lab3.tokenizer_v2 import Tokenizer
from src.lab5.parser import parse
import json
t=Tokenizer("""
Patient: 1
Pregnancies: 5
Glucose: 130
BloodPressure: 80
SkinThickness: 25
Insulin: 100
BMI: 28.5
DiabetesPedigreeFunction: 0.55
Age: 40
Outcome: 1
#i dont like this

""")
t.lines_plit()
response=t.tokenize()
parse_Tree=parse(response)


parse_Tree=json.dumps(parse_Tree,indent=4)
print(parse_Tree)
```

This is the produced Output:

```
{
    "Type": "Program",
    "body": [
        {
            "PATIENT_INFO": [
                {
                    "type": "LITERAL",
                    "value": "Patient"
                },
                {
                    "type": "OPERATOR",
                    "value": ":"
                },
                {
                    "type": "NUMERICAL",
                    "value": "1"
                }
            ]
        },
        {
            "PATIENT_INFO": [
                {
                    "type": "LITERAL",
                    "value": "Pregnancies"
                },
                {
                    "type": "OPERATOR",
                    "value": ":"
                },
                {
                    "type": "NUMERICAL",
                    "value": "5"
                }
            ]
        },
        {
            "PATIENT_INFO": [
                {
                    "type": "LITERAL",
                    "value": "Glucose"
                },
                {
                    "type": "OPERATOR",
                    "value": ":"
                },
                {
                    "type": "NUMERICAL",
                    "value": "130"
                }
            ]
        },
        {
            "PATIENT_INFO": [
                {
                    "type": "LITERAL",
                    "value": "BloodPressure"
                },
                {
                    "type": "OPERATOR",
                    "value": ":"
                },
                {
                    "type": "NUMERICAL",
                    "value": "80"
                }
```

```
    ]
},
{
    "PATIENT_INFO": [
        {
            "type": "LITERAL",
            "value": "SkinThickness"
        },
        {
            "type": "OPERATOR",
            "value": ":"
        },
        {
            "type": "NUMERICAL",
            "value": "25"
        }
    ]
},
{
    "PATIENT_INFO": [
        {
            "type": "LITERAL",
            "value": "Insulin"
        },
        {
            "type": "OPERATOR",
            "value": ":"
        },
        {
            "type": "NUMERICAL",
            "value": "100"
        }
    ]
},
{
    "PATIENT_INFO": [
        {
            "type": "LITERAL",
            "value": "BMI"
        },
        {
            "type": "OPERATOR",
            "value": ":"
        },
        {
            "type": "NUMERICAL",
            "value": "28.5"
        }
    ]
},
{
    "PATIENT_INFO": [
        {
                "type": "LITERAL",
                "value": "DiabetesPedigreeFunction"
            },
            {
                "type": "OPERATOR",
                "value": ":"
            },
            {
                "type": "NUMERICAL",
                "value": "0.55"
            }
        ]
    },
    {
        "PATIENT_INFO": [
            {
                "type": "LITERAL",
                "value": "Age"
            },
            {
                "type": "OPERATOR",
                "value": ":"
            },
            {
                "type": "NUMERICAL",
                "value": "40"
            }
        ]
    },
    {
        "PATIENT_INFO": [
            {
                "type": "LITERAL",
                "value": "Outcome"
            },
            {
                "type": "OPERATOR",
                "value": ":"
            },
            {
                "type": "NUMERICAL",
                "value": "1"
            }
        ]
    },
    {
        "COMMENT": [
            {
                "type": "COMMENT",
                "value": "#i dont like this"
            }
        ]
    }
    ]
}
```

Conclusions

In conclusion, our laboratory work successfully achieved the creation of an AST parser built upon the lexer from lab 3. This is however a pretty limited example as the lexer is built upon a simplistic language. Expanding the DSL will come with expanding the lexer and parser as well. By integrating these components, we have enhanced our understanding of compiler construction and gained practical insights into parsing abstract syntax trees. This experience has enriched our skill set in software development.