Insper 2021.1 - Lógica da Computação

APS - Uma Linguagem de Programação

Caio Horschutz Fauza

1 - Introdução

O objetivo deste documento é implementar uma linguagem de programação customizada, seguindo as estruturas básicas disponíveis na literatura. A linguagem contará com variáveis, condicionais, loops e funções com escopo de variáveis. A linguagem será estruturada segundo o padrão EBNF.

2 - Implementação

2.1 - Estrutura da linguagem

A linguagem idealizada, é resultado da concatenação de uma série de outras linguagens (Python, Javascript e C). A seguinte EBNF foi desenvolvida:

FUNCTION = "toolkit", "IDENTIFIER", "(", PARAMETERS, ")", BLOCK;
RETURN = "recover", IDENTIFIER, OREXPR;

BLOCK = { COMMAND }, { RETURN };
PARAMETERS = (IDENTIFIER), “,”, { PARAMETERS } | (IDENTIFIER);
COMMAND = (λ | ASSIGNMENT | PRINT | WHILE | FOR | IF | BLOCK) ;

PRINT = "show", "(", OREXPR, ")";
WHILE = "until", "(", OREXPR, ")", COMMAND;
ASSIGNMENT = IDENTIFIER, "=", OREXPR;
IF = "if", "(", OREXPR, ")", COMMAND, { ELSE };
ELSE= "else", COMMAND;

OREXPR = ANDEXPR, { ("or"), ANDEXPR};
ANDEXPR = EQEXPR, { ("and"), EQEXPR};
EQEXPR = RELEXPR, { ("==" | "!="), RELEXPR};
RELEXPR = EXPRESSION, { (">" | "<" | ">=" | "<="), EXPRESSION};
EXPRESSION = TERM, { ("+" | "-"), TERM };
TERM = FACTOR, { ("*" | "/"), FACTOR };
FACTOR = (("+" | "-" | "!"), FACTOR) | NUMBER | IDENTIFIER | IDENTIFIER, “(“, FUNCTIONPARAMS, “)” | "(", OREXPR, ")" | "door", "(", ")";
FUNCTIONPARAMS= (OREXPR), “,”, { FUNCTIONPARAMS } | (OREXPR);
IDENTIFIER = (LETTER | "_"), { LETTER | DIGIT | "_" };
NUMBER = DIGIT, { DIGIT };
LETTER = (a | ... | z | A | ... | Z);
DIGIT = (1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0);
BOOLEAN = "true" | "false";

2.2 - Análise Léxica

Para realizar a análise léxica da linguagem, utilizou-se a linguagem python em conjunto com a biblioteca PLY (ply.lex). Todos os tokens disponíveis na linguagem foram identificados e relacionados a expressões regulares específicas. Assim, quando um código fonte for inserido, será realizada a tokenização de acordo com os seguintes atributos:

```
reserved = {
    'toolkit': 'FUNCTION',
    'recover': 'RETURN',
    'door': 'INPUT',
    'show': 'PRINT',
    'until': 'WHILE',
    'over': 'FOR',
    'if': 'IF',
    'else': 'ELSE',
    'or': 'OR',
    'and': 'AND',
    'true': 'TRUE',
    'false': 'FALSE'
}

tokens = ['NUMBER', 'PLUS', 'MINUS', 'MULT', 'DIV', 'LPAR', 'RPAR', 'IDENTIFIER',
          'STRING', 'EQUAL', 'ATTRIB', 'NOTEQUAL', 'NOT', 'BIGGER', 'SMALLER', 'BIGGEREQUAL',
          'SMALLEREQUAL', 'COMMA', 'SEMICOLLON', 'OPENBLOCK', 'CLOSEBLOCK'] + list(reserved.values())

t_PLUS = r'\+'
t_MINUS = r'\-'
t_MULT = r'\*'
t_DIV = r'\/'
t_LPAR = r'\('
t_RPAR = r'\)'
t_STRING = r'\".*"'
t_ATTRIB = r'\='
t_EQUAL = r'\=='
t_NOTEQUAL = r'\!='
t_NOT = r'\!'
t_BIGGER = r'\>'
t_SMALLER = r'\<'
t_BIGGEREQUAL = r'\>='
t_SMALLEREQUAL = r'\<='
t_COMMA = r'\,'
t_SEMICOLLON = r'\;'
t_OPENBLOCK = r'\{'
t_CLOSEBLOCK = r'\}'
t_ignore = ' \t'
```

```
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t


def t_IDENTIFIER(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    t.type = reserved.get(t.value, 'IDENTIFIER')
    return t


def t_COMMENT(t):
    r'\//.*'
    pass


def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)


def t_error(t):
    print('Invalid syntax. Character {} is invalid'.format(t.value[0]))
    t.lexer.skip(1)
```

2.3 - Análise Sintática + AST + Análise semântica

Para realizar a análise sintática e posteriormente a análise semântica da linguagem, utilizou-se também a biblioteca PLY (ply.yacc), em conjunto com uma série de tipos específicos de nós, para construção da AST. Cada nó possui um método “evaluate” relacionado, que é chamado na etapa de execução do código.

Para armazenar variáveis foi implementada uma estrutura de dicionário local para cada função, denominada “Symbol Table”. Para implementar funções foi implementada uma estrutura global de dicionário, denominada “Function Table”. Os nós são:

String_val:

- **Definição:** nó correspondente a um token de string.
- **Método evaluate:** retorna o valor atribuído.

Bool_val:

- **Definição:** nó correspondente a um token booleano.
- **Método evaluate:** retorna o valor atribuído.

Int_val:

- **Definição:** nó correspondente a um token inteiro.
- **Método evaluate:** retorna o valor atribuído.

Identifier_val:

- **Definição:** nó correspondente a um token identificador de variável.
- **Método evaluate:** consulta a symbol table para retornar o valor correspondente.

Un_op:

- **Definição:** nó correspondente a uma operação unária.
- **Método evaluate:** executa a operação unária e retorna o valor correspondente.

Bin_op:

- **Definição:** nó correspondente a uma operação binária.
- **Método evaluate:** executa a operação binária e retorna o valor correspondente.

Assignment_op:

- **Definição:** nó correspondente a uma operação de atribuição de valor para uma variável.
- **Método evaluate:** salva um novo objeto na symbol table com o valor correspondente.

Condition_op:

- **Definição:** nó correspondente a uma operação condicional.
- **Método evaluate:** executa a expressão condicional e executa a operação correspondente (se a expressão for verdadeira, executa o bloco “if” e se a expressão for falsa, executa o bloco “else”).

While_op:

- **Definição:** nó correspondente a uma operação de loop.
- **Método evaluate:** executa a expressão condicional e enquanto ela for verdadeira, executa posteriormente o código correspondente ao loop.

Input_op:

- **Definição:** nó correspondente a uma operação de leitura da entrada padrão.
- **Método evaluate:** recebe um valor na entrada padrão.

Print_op:

- **Definição:** nó correspondente a uma operação de exibição na saída padrão.
- **Método evaluate:** exibe na saída padrão o valor correspondente.

Statement_op:

- **Definição:** nó correspondente a uma estrutura de bloco.
- **Método evaluate:** percorre os nós filhos do bloco e executa seus métodos “evaluate” de forma sequencial.

Var_dec:

- **Definição:** nó correspondente a uma declaração de assinatura de função.
- **Método evaluate:** retorna.

Func_call:

- **Definição:** nó correspondente a uma chamada de função.
- **Método evaluate:** consulta a function table e executa a função recebida em conjunto com seus parâmetros.

Func_dec:

- **Definição:** nó correspondente a uma declaração de função.
- **Método evaluate:** salva um novo objeto na function table com o escopo da função correspondente.

Para realizar as análises, determinou-se expressões semelhantes a EBNF para contemplar todas as possibilidades estruturadas em 2.1. Como por exemplo, para as regras de produção “COMMAND”, desenvolveu-se a seguinte função:

```
def p_command(p):
    '''command : IDENTIFIER ATTRIB orexpr SEMICOLLON
               | PRINT LPAR orexpr RPAR SEMICOLLON
               | RETURN IDENTIFIER orexpr SEMICOLLON
               | IDENTIFIER LPAR inputparams RPAR SEMICOLLON
               | IDENTIFIER LPAR RPAR SEMICOLLON
               | block
               | WHILE LPAR orexpr RPAR command
               | IF LPAR orexpr RPAR command
               | IF LPAR orexpr RPAR command ELSE command'''

    if(len(p) == 2):
        p[0] = p[1]
    else:
        if(p[2] == '='):
            p[0] = Assignment_op(p[2], [Identifier_val(p[1]), p[3]])
        elif(p[1] == 'show'):
            p[0] = Print_op(p[1], [p[3]])
        elif(p[1] == 'until'):
            p[0] = While_Op(p[1], [p[3], p[5]])
        elif(p[1] == 'recover'):
            p[0] = Return_val('RETURN', [p[2], p[3]])
        elif(p[1] != 'if' and p[2] == '(' and p[4] == ')'):
            p[0] = Func_Call(p[1], p[3])
        elif(p[1] != 'if' and p[2] == '(' and p[3] == ')'):
            p[0] = Func_Call(p[1], [])
        elif(p[1] == 'if'):
            if(len(p) == 6):
                p[0] = Condition_Op(p[1], [p[3], p[5]])
            else:
                p[0] = Condition_Op(p[1], [p[3], p[5], p[7]])
```

2.4 - Execução

Para executar o código, utilizou-se da estrutura de árvore implementada anteriormente. Na etapa de construção da AST, cada função definida (Func_Dec) foi amarrada a um nó raiz (Statement_Op) e por fim, adicionou-se um nó do tipo chamada de função (Func_Call) para executar a função denominada “main”. Ao chamar o método “evaluate” do nó raiz, todos os nós filhos são executados em ordem e por fim, a função main é chamada realizando o comportamento correspondente ao código fonte utilizado.

3 - Exemplos

Alguns exemplos foram disponiblizados na pasta “examples” no repositório do projeto e podem ser executados de acordo com o seguinte comando:

python3 main.py examples/{filename}