CSE341 FALL 2024 – Programming Languages Assignment 3 Report -Cemal BOLAT-

- Introduction: Tools that I used

Lex is a lexical analyzer generator that processes input text and converts it into tokens, which are the basic elements (like keywords, identifiers, and operators) used by a compiler or interpreter.

Yacc (Yet Another Compiler Compiler) is a parser generator that takes tokens from Lex and builds a syntax tree, ensuring the input follows the defined grammar rules. It helps to define the structure and logic of a programming language or script.

- Expression Evaluation Strategy

I have designed my parse tree for the eager evaluation strategy. That is, as soon as an expression is defined, it is immediately evaluated and the result is calculated. Since the evaluation of expressions is immediate, there are no unnecessary calculations and the result is obtained immediately.

- Scope

I would prefer to use static scoping technique because it increase the predictability and readability of the code. With static scoping, the scope of variables is determined at compile time, allowing developers to easily trace where a variable is defined and where it can be accessed, making the code more transparent and easier to debug.

- Algorithm for building a parse tree.

I have used a top-down parser and its recursive descent algorithm because it offers a clear and intuitive structure for parsing the grammar of a language. This method closely mirrors the grammatical rules, making the parsing logic easy to understand and implement. By breaking down each non-terminal in the grammar into a corresponding function, the recursive descent approach allows for modular and maintainable code, where each function handles a specific grammar rule.

- The Conflicts I have overcome.
    - The description of the non-terminal token explist is completely difficult to handle and it is almost impossible to resolve conflicts on “yacc”, so I changed it without breaking the requirements of the language.

My version is

```
EXPLIST:
        EXP |
        EXPLIST EXP;
```

-
    - Yacc uses an LALR(1) parsing strategy, which can lead to issues when dealing with loop-like structures. This limitation arises because LALR(1) parsers may struggle with certain recursive or ambiguous grammar patterns, requiring extra care in designing grammars to avoid conflicts or reduce/reduce errors.

○

CFG that I used for G++.

```
START:
        /* empty */ |
        INPUT;
INPUT:
        EXPLIST;
EXPLIST:
        EXP |
        EXPLIST EXP;
EXP:
        OP_OP OP_PLUS EXP EXP OP_CP |
        OP_OP OP_MINUS EXP EXP OP_CP |
        OP_OP OP_MULT EXP EXP OP_CP |
        OP_OP OP_DIV EXP EXP OP_CP |
        OP_OP KW_IF EXPB EXP EXP OP_CP |
        OP_OP KW_IF EXPB EXP OP_CP |
        OP_OP KW_FOR OP_OP IDENTIFIER EXP EXP OP_CP EXPLIST OP_CP
        OP_OP KW_WHILE EXPB EXP OP_CP |
        OP_OP KW_PRINT EXP OP_CP |
        OP_OP KW_EXIT OP_CP |
        OP_OP KW_DEFFUN IDENTIFIER OP_OP FUNC_PARAMS OP_CP EXPLIST OP_CP |
        EXPB |
        LIST_INPUT |
        SET |
        OP_OP KW_DEFVAR IDENTIFIER EXP OP_CP |
        FCALL |
        OP_OP KW_LOAD STRING OP_CP |
        COMMENT | VALUEF | VALUEI;
LIST_INPUT:
        OP_OP KW_APPEND LIST_INPUT LIST_INPUT OP_CP |
        OP_OP KW_CONCAT LIST_INPUT LIST_INPUT OP_CP |
        IDENTIFIER |
        LIST;
SET:
        OP_OP KW_SET IDENTIFIER EXP OP_CP;
LIST:
        OP_OP KW_LIST VALUES OP_CP |
        OP_APOSTROPHE OP_OP OP_CP |
        KW_NIL |
        OP_APOSTROPHE OP_OP VALUES OP_CP;
VALUES:
        VALUEI | VALUEF | IDENTIFIER |
        VALUES OP_COMMA VALUEI |
        VALUES OP_COMMA VALUEF |
        VALUES OP_COMMA IDENTIFIER;
FCALL:
        OP_OP IDENTIFIER OP_CP |
        OP_OP IDENTIFIER EXPLIST OP_CP;
FUNC_PARAMS:
        /* empty */ |
        FUNC_PARAMS IDENTIFIER;
EXPB:
        OP_OP KW_AND EXP EXP OP_CP |
        OP_OP KW_OR EXP EXP OP_CP |
        OP_OP KW_NOT EXPB OP_CP |
        OP_OP KW_LESS EXP EXP OP_CP |
        OP_OP KW_EQUAL EXP EXP OP_CP |
        KW_TRUE | KW_FALSE;
```