# Analisador Léxico

João Victor Cabral de Melo[1]

Universidade de Brasília

**Abstract.** This project is to show, what we've done and learned in the course of compilers and the grammar of experimental language C-IPL.

**Keywords:** Compilers · Grammar · Lexical Analyser · Flex

## 1 Introdução

### 1.1 Motivação

A maior motivação para a implementação de um tradutor é o entendimento por trás das abstrações do compilador facilitando assim a percepção do aluno enquanto as linguagens de programação.

### 1.2 Nova Primitiva

A motivação para entrada da nova primitiva *list* é a possibilidade de uma pequena abordagem do paradigma funcional. A importância da primitiva se da em problemas onde se precisa de uma estrutura de dados com um tamanho não definido previamente. Facilitando esse conjunto específico de problemas pela não implementação da estrutura de dados manualmente.

## 2 Análise Léxica

A função do *scanner* do compilador é identificar *tokens* na linguagem e coloca-los na tabelas de símbolos para a avaliação do sintático. Neste projeto foi utilizado o **flex** para tal função, além disso foram feitas tais modificações no arquivo como as variáveis para contagem de linhas, colunas e erros além tratamento do caso em que não consiga abrir o arquivo a ser escaneado, para ser input de usuário a ser lido. A tabela de símbolos será implementada como uma lista ligada sendo cada valor dessa uma lista um:

```
struct node {
    char *Value;
    char *TOKEN;
    node *next;
};
```

## 3 Testes

No diretório de testes possui dos arquivos com erros léxicos **error.c** e **error1.c** e dois arquivos sem erros léxicos **success.c** e **success1.c**. Sendo os erros são

```
Error: $ Unidentified token at line: 2 and column: 19
Error: 42jujuba Not a valid token at line: 1 and column: 6
```

## 4 Instruções

Para se compilar o analisador léxico utiliza-se os seguintes comandos:

```
lex lexico.l
cc -Wall lex.yy.c -lfl -o tradutor
./tradutor ../test/error.c
./tradutor ../test/error1.c
./tradutor ../test/success.c
./tradutor ../test/success1.c
```

Caso de algum problema com a compilção do arquivo **lex.yy.c** descomente o seguinte código:

```
/*%option noyywrap*/
```

E tente compilar novamente sem a *flag* **-lfl** dos comandos acima.

## References

1. AV Aho, R Sethi, JD Ullman: Compilers: Principles, Techniques, and Tools
2. Levine J.: Flex and Bison
3. Flex Manual: https://westes.github.io/flex/manual/. Acessado pela última vez: 09/08/2020 – 16:24

# Appendices

# Appendix A

# Tabelas de Símbolos

| Token | Informal Description | Sample Lexemes |
|---|---|---|
| ID | qualquer letra com '_' seguido de um número | jujuba_doce |
| LIST | qualquer tipo do list | list |
| TYPES | tipos int e float | int float |
| NULL | constante para listas vazias | NIL |
| STRING | qualquer caracter dentro de " " | "chocolate" |
| ASSIGN | Um único caractere = | = |
| KEYWORD | Palavras como if else for e return | if else for return |
| OUTPUT | Palavras da saída de dados da linguagem | write writeln |
| INPUT | Palavra de entrada de dados | read |
| CONSTRUCTOR | Palavras para construtores de lista | rvalue lvalue |
| INFIX | Caracter para fazer construtor uma operação infixa | : |
| FUNCTION | Caracteres para as função de map e filter | >><< |
| COMMENT | Qualquer comentário em linha com //<br>ou comentário multilinha com /**/ | //pudim<br>/*<br>pudim é<br>bom<br>/ |
| + — - — * — / | Operações aritmeticas | + - * / |
| —— — && | Operações Lógicas | && —— |
| >—<—>=—<=—==—!= | Operação de comparação | <<= >>= == != |
| {} | Escopo | {} |
| () | Expressão | () |
| ; | Separador de Expressão | ; |
| ? | Operação head da lista | ? |
| ! | Operação tail da lista ou negação | ! |
| % | Operação tail destrutiva da lista | % |
| 123 | Número inteiro | 123 |
| .5 | Número decimal | .5 |

## Appendix B

# Gramática

⟨program⟩ ::= ⟨paramlist⟩ (1)

⟨paramlist⟩ ::= ⟨param⟩ ⟨paramlist⟩ | ⟨param⟩ (2)

⟨param⟩ ::= ⟨variableParam⟩ | ⟨functionParam⟩ (3)

⟨variableParam⟩ ::= ⟨type⟩ ⟨id⟩ ⟨d⟩ | ⟨type⟩ ⟨listType⟩ ⟨id⟩ ⟨d⟩ (4)

⟨vFunction⟩ ::= ⟨type⟩ ⟨id⟩ | ⟨type⟩ ⟨listType⟩ ⟨id⟩ (5)

⟨pList⟩ ::= ⟨vFunction⟩ ⟨v⟩ ⟨pList⟩ | ⟨vFunction⟩ (6)

⟨functionParam⟩ ::= ⟨type⟩ ⟨id⟩ ⟨p⟩⟨pList⟩ ⟨p⟩ ⟨s⟩ (7)

⟨expression⟩ ::= ⟨arithmExpression⟩ | ⟨logicalExpression⟩ | (8)

⟨comparisonExpression⟩ | (9)

⟨expression⟩ ⟨d⟩⟨expression⟩ (10)

⟨inputExpression⟩ ::= ⟨input⟩ ⟨p⟩ ⟨p⟩ ⟨d⟩ (11)

⟨outputExpression⟩ ::= ⟨output⟩ ⟨p⟩ ⟨string⟩⟨p⟩ ⟨d⟩ (12)

⟨arithmExpression⟩ ::= ⟨number⟩⟨arOP⟩⟨number⟩ (13)

⟨logicalExpression⟩ ::= ⟨loOp⟩ ⟨id⟩ | ⟨id⟩ ⟨loOp⟩ ⟨id⟩ (14)

⟨comparisonExpression⟩ ::= ⟨id⟩ ⟨coOp⟩⟨id⟩ (15)

⟨listExpression⟩ ::= (⟨head⟩ | ⟨tail⟩ | ⟨dTail⟩) ⟨id⟩ | (16)

⟨id⟩ ⟨infix⟩ ⟨id⟩ | (17)

⟨cosntructor⟩ ⟨id⟩ | (18)

⟨id⟩ ⟨function⟩ ⟨id⟩ (19)

⟨assignExpression⟩ ::= ⟨id⟩ ⟨assign⟩ ⟨id⟩ | (20)

⟨id⟩ ⟨assign⟩ ⟨number⟩ (21)

⟨keywordExpression⟩ ::= ⟨keyword⟩ ⟨p⟩ ⟨expression⟩ ⟨p⟩ ⟨s⟩ (22)

⟨arOp⟩ ::= `+` | `-` | `*` | `/` (23)

⟨loOp⟩ ::= `!` | `&&` | `||` (24)

⟨coOp⟩ ::= `>` | `<` | `>=` | `<=` | `!=` | `==` (25)

⟨s⟩ ::= `{` | `}` (26)

⟨p⟩ ::= `(` | `)` (27)

⟨d⟩ ::= `;` (28)

⟨head⟩ ::= `?` (29)

⟨tail⟩ ::= `!` (30)

⟨dTail⟩ ::= `%` (31)

⟨type⟩ ::= `int` | `float` (32)

⟨listType⟩ ::= `list` (33)

⟨v⟩ ::= `,` (34)

⟨null⟩ ::= `NIL` (35)

⟨assign⟩ ::= `=` (36)

⟨keyword⟩ ::= `for` | `if` | `else` | `return` (37)

⟨output⟩ ::= `writeln` | `write` (38)

⟨input⟩ ::= `read` (39)

⟨constructor⟩ ::= `rvalue` | `lvalue` (40)

⟨infix⟩ ::= `:` (41)

⟨function⟩ ::= `>>` | `<<` (42)