Trabalho Prático 1 - Analisador Léxico

João Victor Bohrer Munhoz[16/0071101]

Universidade de Brasília, Brasília, DF, 70910-900, Brasil
160071101@aluno.unb.br

1 Motivação

Para a matéria de Tradutores da Universidade de Brasília, o trabalho obrigatório consiste em criar um tradutor que consiga transformar um arquivo escrito na linguagem especificada pela docente em um código executável através das ferramentas *Flex* e *Bison*.

A linguagem em questão é a **C-IPL**[Nal21], que foi projetada para facilitar o tratamento de listas em programas escritos em C. Para isso, foi introduzida uma nova primitiva de dados para listas e operações sobre esta nova primitiva. As demais primitivas e comandos de C têm semântica padrão.

Espera-se que ao implementar o tradutor, o aluno entenda mais profundamente de que maneira os programas são compilados e como certas características de algumas linguagens deixam esse processo mais fácil ou mais difícil.

2 Descrição

Nessa linguagem **C-IPL**, foi criada uma nova primitiva *list* para lidar com listas, uma nova constante NIL para indicar o fim de uma lista e novas operações sobre as listas, podendo-se dividi-las em construtores, operações unárias e operações binárias. A explicação detalhada dos operadores se encontra na especificação da linguagem.[Nal21]

Para realizar a análise léxica, utilizamos em grande parte as operações regulares. Portanto, começamos o arquivo com várias macros definindo algumas expressões regulares que usaremos mais para frente, tais como *DIGIT*, que identifica números, *TYPE*, que identifica os tipos de declaração, *NIL* que identifica a constante NIL, *MINUS* que identifica o símbolo de menos, *SUMOP* que identifica o símbolo de soma, *MULOP* que identifica as operações de divisão e multiplicação, *EXCLAM* que identifica o símbolo de exclamação, *LOGOP* que identifica as operações lógicas padrão de C, *RELOP* que identifica as operações relacionais usuais de C, *ASSIGN* que identifica o operador de atribuição, *KEYWORDS* que identifica as estruturas de *loop* especificadas na linguagem, *READWRITE* que identifica as operações de leitura e escrita da linguagem, *LISTOP* que identifica as operações sobre listas também especificadas na linguagem e *ID* que checa por qualquer outro identificador que não os já reservados anteriormente. Checou-se também por comentários e strings através de máquinas de estados ao longo do programa.

A partir do momento que identificamos esses lexemas, criamos uma *struct* para gerar o respectivo *token* com: o **tipo** do *token*, com todos os tipos presentes na tabela 1, a **posição**, em linha e coluna, em que se encontra o lexema no arquivo e seu **atributo opcional**, o qual pode variar de acordo com qual tipo de *token* estamos lidando.

Para qualquer *token* que seja uma declaração de função, declaração de variável ou declaração de parâmetro, esse *token* será inserido na tabela de símbolos, a qual será muito importante para as partes mais adiante do tradutor. No caso desse trabalho, pretende-se usar uma árvore para representar a tabela de símbolos, onde cada nó é uma lista com o *token*, se o *token* é uma função ou uma variável e de que tipo é essa função ou variável.

Cada nó também representa um escopo, e a cada novo escopo que surja dentro de um desses nós, esse nó gera um filho, o qual é um novo escopo dentro do escopo do pai, desse modo controlando os escopos do programa e checando por possíveis declarações repetidas ou uso de variáveis ou funções inexistentes.

3 Arquivos de Teste

Ao total foram disponibilizados 5 arquivos de teste junto com essa entrega, três deles corretos e dois deles com erros léxicos apontados pelo analisador. Os itens corretos são:

- **Ex_Correct_0.c:** Arquivo de teste fornecido junto com a descrição da linguagem.
- **Ex_Correct_1.c:** Arquivo próprio que testa metade das instruções.
- **Ex_Correct_2.c:** Arquivo próprio que testa a outra metade das instruções.

Os itens incorretos são;

- **Ex_Incorrect_1.c:** Há um ponto extra no float da linha 17 na coluna 16, há um símbolo **$** que não faz parte da linguagem na linha 27 e coluna 17 e há uma *string* aberta mas não fechada na linha 32 e coluna 13.
- **Ex_Incorrect_2.c:** Há um símbolo ^ que não faz parte da linguagem na linha 10 e coluna 33 e há um comentário em bloco não fechado que se estende até o final do arquivo na linha 20 e coluna 5.

4 Instruções de Execução

O programa foi desenvolvido e compilado utilizando as seguintes versões das ferramentas:

- **OS:** Fedora 34 (Workstation Edition) x86_64
- **Kernel:** 5.13.8
- **Flex:** 2.6.4
- **GCC:** 11.2.1

Para compilar, simplesmente entre na pasta raiz e digite:

```
$ make tradutor_lex
```

Uma vez com o arquivo *tradutor* já na pasta raiz, digite o seguinte comando para rodar os programas no arquivo de teste:

```
$ ./tradutor <caminho para o arquivo>
```

References

[Nal21] Claudia Nalon. Trabalho prático - descrição da linguagem. https://aprender3.unb.br/mod/page/view.php?id=464034, Agosto 2021. Acessado pela última vez em 10/08/2021.

Table 1. Projeto Léxico do trabalho

Token	**Expressão Regular**
Tipo	int | float
Constante Int	[0-9]+
Constante Float	[0-9]+"."[0-9]+
NIL	(NIL)
Símbolo de Menos	[-]
Operação de Soma	[+]
Operação de Multiplicação	[*/]
Símbolo de Exclamação	[!]
Operação Lógica	(&&) | (||)
Operação Relacional	[<] | (<=) | [>] | (>=) | (==) | (!=)
Atribuição	[=]
Keyword	if | else | for | return
I/O	read | write | writeln
String	\"(\\.|[^"\\])*\"
Operação de Listas	[:?%] | (>>) | (<<)
ID	[A-Za-z_][A-Za-z0-9_]*
Ponto e vírgula	[;]
Abre parêntesis	[(]
Fecha parêntesis	[)]
Abre chaves	[{]
Fecha chaves	[}]
Vírgula	[,]

1. program → declList
2. declList → declList decl | decl
3. decl → varDecl | funDecl
4. varDecl → typeList varDecl | **ID**
5. typeList → type typeList | type
6. type → **int** | **float** | **list**
7. funDecl → typeList **ID** (params) stmt | **ID** (params) stmt
8. params → paramList | ϵ
9. paramList → paramTypeList **,** paramList | paramTypeList
10. paramTypeList → typeList **ID**
11. stmt → expStmt | compoundStmt | ifStmt | forStmt | returnStmt
12. expStmt → exp **;** | **;**
13. compoundStmt → **{** localDecls stmtList **}**
14. localDecls → localDecls varDecl | ϵ
15. stmtList → stmtList stmt | ϵ
16. ifStmt → **if** simpleExp compoundStmt | **if** simpleExp compoundStmt **else** elseStmt
17. elseStmt → ifStmt | compoundStmt
18. forStmt → **for** (simpleExp **;** relExp **;** simpleExp) compoundStmt
19. returnStmt → **return ;** | **return** exp **;**
20. exp → **ID** = exp | simpleExp
21. simpleExp → simpleExp || andExp | andExp
22. andExp → andExp **&&** unaryRelExp | unaryRelExp
23. unaryRelExp → **!** unaryRelExp | relExp
24. relExp → sumExp relOp sumExp | sumExp
25. relOp → < | <= | > | >= | == | ! =
26. sumExp → sumExp sumOp sumExp | mulExp
27. sumOp → − | +
28. mulExp → mulExp mulOp mulExp | listExp
29. mulOp → ∗ | /
30. listExp → listExp listOp listExp | unaryExp
31. listOp → : | >> | <<
32. unaryExp → unaryListOp factor | factor
33. unaryOp → − | ? | ! | %
34. factor → (exp) | call | constant | readFunc | writeFunc | **ID**
35. call → **ID** (args)
36. readFunc → **read** (**ID**) **;**
37. writeFunc → writeType (**ID**) | writeType (constant)
38. writeType → **write** | **writeln**
39. args → argList | ϵ
40. argsList → argList **,** exp | exp
41. constant → **NUMCONST** | **FLOATCONST** | **STRINGCONST** | **NIL**