Laboratório de Avaliação 1 - Analisador Léxico

João Victor Bohrer Munhoz[16/0071101]

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

Abstract. Esse é o relatório da primeira parte do trabalho de tradutores de 2021-1, que consiste na implementação do Analisador Léxico da linguagem C-IPL, projetada para incluir facilidade no tratamento de listas.

Keywords: Tradutores · Análise Léxica · Flex.

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 **C-IPL** em um código executável através das ferramentas *Flex* e *Bison*. 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

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, *SUMOP* que identifica as as operações de soma e subtração, *MULOP* que identifica as as operações de divisão e multiplicação, *LOGOP* que identifica as operações lógicas padrão de C, *RELOP* que identifica as operações relacionais usuais de C, *KEYWORDS* que identifica as estruturas de *loop* especificadas na 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.

Esse analisador léxico, entretanto, possui duas definições temporárias. Devido a uma superposição do uso do símbolo **"-"**, que pode ser usado tanto como uma operação aritmética quanto para definir números negativos, e do símbolo **"!"**, que pode ser usado tanto como uma operação de listas como uma negação lógica, esses dois símbolos possuem uma categoria própria de *tokens*, Símbolo de Menos e Símbolo de Exclamação, respectivamente. Essa ambiguidade será resolvida na próxima parte do trabalho, o analisador sintático.

E por fim, uma particularidade desse analisador é que foram usadas duas máquinas de estado para checar por *strings* e comentários (do tipo `/* */`). Caso fossem usadas expressões regulares como nos outros casos, o *overhead* em caso *strings* ou comentários muito grandes causaria uma perda de desempenho muito grande.

As máquinas de estado também facilitam o tratamento de casos onde um comentário ou *string* não foram fechados antes de uma quebra de linha (no caso das *strings*) ou antes do fim do arquivo (se aplica a ambos). No caso específico da *string*, caso haja uma quebra de linha antes de se detectar o fechamento dela, o analisador sai da máquina de estados que estava e volta ao programa normal, como forma de tentar se recuperar de um erro e continuar a análise.

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 3 na coluna 18, há um símbolo **$** que não faz parte da linguagem na linha 6 e coluna 29 e há uma *string* aberta mas não fechada na linha 8 e coluna 17.
- **Ex_Incorrect_2.c:** Há um símbolo ^ que não faz parte da linguagem na linha 10 e coluna 14, há um símbolo **&** sozinho que não faz parte da linguagem na linha 25 e coluna 23 e há um comentário em bloco não fechado que se estende até o final do arquivo na linha 36 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, entre na pasta *src* onde se encontra o arquivo *C_IPL_Lex-Analyzer.l* e digite os seguintes comandos para compilar o arquivo .c e já criar seu executável:

```
$ flex C_IPL_Lex-Analyzer.l && gcc lex.yy.c -g -Wall
```

Uma vez com o arquivo *a.out* na pasta src, digite o seguinte comando para rodar o analisador léxico em arquivos de teste:

```
$ ./a.out <caminho para o arquivo>
```

References

1. Especificação da Linguagem, https://aprender3.unb.br/mod/assign/view.php?id=464058. Acessado pela última vez em 10/08/2021
2. Manual oficial do Flex, https://westes.github.io/flex/manual/. Acessado pela última vez em 10/08/2021
3. Aho, Alfred V., Lam, Monica S., Sethi, Ravi, Ullman, Jeffrey D.: Compilers: Principles, Techniques & Tools. 2nd edn. Pearson, United States (2007)
4. Heckendorn, R.: A grammar for the c- programming language (version s21). University of Idaho (2021), http://marvin.cs.uidaho.edu/Teaching/CS445/c- Grammar.pdf, Acessado pela última vez em 10/08/2021

Token	**Descrição Informal**	**Exemplos de Lexemas**
Tipo	A sequencia de caracteres: int OR float OR list	int
Constante Int	Qualquer sequência somente de números	123
Constante Float	Sequência de números, um ponto, e mais uma sequência de números	123.1
Constante NIL	A sequência de caracteres NIL	NIL
Símbolo de Menos	O caractere -	-
Operação de Soma	Os caracteres: + OR -	+
Operação Multiplicação	Os caracteres: * OR /	*
Símbolo de Exclamação	O caractere !	!
Operação Lógica	Os caracteres: && OR ||	&&
Operação Relacional	Os caracteres: <OR <= OR >OR >= OR == OR !=	>
Atribuição	O caractere =	=
Keyword	A sequência de caracteres: if OR else OR for OR return	if
I/O	A sequência de caracteres: read OR write OR writeln	read
Operação de Listas	Os caracteres: ? OR : OR % OR >>OR <<	%
ID	Letra ou underline podendo ser seguido de letras, números ou underlines	number
Ponto e Vírgula	O caractere ;	;
Parêntesis	Os caracteres: (OR)	(
Chaves	Os caracteres: { OR }	}
Vírgula	O caractere ,	,
String	Tudo que está entre um caractere ” e outro caractere ” na mesma linha	”Digite sua idade”

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 → <- **Ainda é necessário construir a gramática de if e else**
17. forStmt → <- **Ainda é necessário construir a gramática do for**
18. returnStmt → **return ;** | **return** exp **;**
19. exp → **ID** = exp | simpleExp
20. simpleExp → simpleExp || andExp | andExp
21. andExp → andExp **&&** unaryRelExp | unaryRelExp
22. unaryRelExp → **!** unaryRelExp | relExp
23. relExp → sumExp relOp sumExp | sumExp | listExp
24. relOp → < | <= | > | >= | == | ! =
25. listExp → listExp listOp listExp | **ID <- Ainda é necessário separar os tipos de operação sobre listas entre unárias, binárias e etc**
26. listOp → : | ? | % | ! | >> | <<
27. sumExp → sumExp sumOp sumExp || mulExp
28. sumOp → − | +
29. mulExp → mulExp mulOp mulExp || unaryExp
30. mulOp → ∗ | /
31. unaryExp → unaryOp factor
32. unaryOp → −
33. factor → **(** exp **)** | call | **ID**
34. call → ID **(** args **)**
35. args → argList | ϵ
36. argsList → argList **,** exp | exp