Analisador Léxico

Gabriel Cunha Bessa Vieira - 16/0120811

Universidade de Brasília

1 Motivação

Este é o relatório inicial do trabalho que será conduzido ao longo do semestre, contando apenas com o analisador léxico. O objetivo final do trabalho será reconhecer uma linguagem baseada denominada *C-IPL* que foi descrita em [Nal]. A implementação dessa nova linguagem terá como impacto o aprendizado de todo o processo de tradução desde linguagem de alto nível até a geração da linguagem intermediária a nível de código. Saber como ocorrem tais processos é fundamental para um cientista entender como um código pode ser otimizado visando o processo de compilação, criando, assim, um programa mais performático.

A implementação de listas dentro da nova primitiva tem por objetivo principal facilitar a manipulação de dados por meio de operadores lógicos como '?', '≫', '≪', ':', '!', '%'. A lista dá a possibilidade de usar uma estrutura robusta que ainda não existe no C. Além de ser possível manipular dados dentro do código com listas, é possível manipular a memória e determinar como os arquivos de fato são armazenados dentro do SO por meio das mesmas.

2 Descrição

A análise léxica tem como objetivo atribuir lexemas [Est] para a nova primitiva a partir da *regex* criada com o intuito de identificar os possíveis *tokens* [ALSU07], os quais podem ser alfanuméricos[a-z0-9], *strings*, operadores aritméticos '+', '−', '∗', '/', operadores lógicos '&&', '||', identificadores, tipos(int—float), constantes, operadores de lista(mencionados na seção anterior) e delimitadores como ';', ',', '{', '}', '(', ')'. Quando ocorre um comportamento não esperado, é produzido uma linha de erro que aponta no formato '**(linha:coluna)** ***token***' juntamente com a descrição do erro.

Para realizar a separação de identificadores e operadores dentro do analisador foram utilizadas macros que têm o nome análogo à sua expressão regular. Tudo foi feito de acordo com a estrutura padrão do Flex [Est], que são definições, seguidas de suas regras e por fim o código que será exportado para o .c gerado a partir do arquivo Flex.

A tabela de símbolos será implementada posteriormente utilizando listas encadeadas no formato <id, valor>. Já os tokens descritos na estrutura dos tokens no apêndice B serão armazenados por meio de uma *struct*.

3 Variáveis de ambiente e versões

3.1 Compilação

Para compilar, é necessário estar na raiz e executar o seguinte comando:

```
$ make all
```

Para rodar algum programa é necessário entrar o seguinte comando:

```
$ ./tradutores tests/<nome_do_programa>.c
```

Os arquivos disponibilizados serão os seguintes:

1. teste_correto1.c
2. teste_correto2.c
3. teste_errado1.c
4. teste_errado2.c

] São fornecidos dois arquivos teste para cada caso que esteja correto ou errado. Com o **teste_errado1.c** contendo erro na linha 2, coluna 10(2:10). E o **teste_errado2.c** contendo erro na linha 2 coluna 9(2:9) e na linha 3 colunas 6 a 8 (3:6), (3:7), (3:8), retratados a seguir:

- *TesteErrado_1.c*

```
(2:10)Unidentified character: '.'
```

- *TesteErrado_2.c*

```
(2:9)Unidentified character: '#'
(3:6)Unidentified character: '$'
(3:7)Unidentified character: '#'
(3:8)Unidentified character: '@'
```

3.2 Software Utilizado

- Ubuntu LTS 20.04
- Flex 2.6.4
- Gcc 11.2.1
- GNU Make 4.3

Referências

[ALSU07] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, & Tools*. Pearson/Addison Wesley, 2nd edition, 2007. Online; acessado 10 de Agosto de 2021.

[Est] W. Estes. Flex: Fast lexical analyser generator. https://github.com/westes/flex. Online; acessado 10 de Agosto de 2021.

[Nal] Cláudia Nalon. Trabalho prático - descrição da linguagem. https://aprender3.unb.br/mod/page/view.php?id=464034. Online; acessado 10 de Agosto de 2021.

A Gramática

1. $< PROGRAM > \rightarrow < DECLARATION\_LIST >$
2. $< DECLARATION\_LIST > \rightarrow < DECLARATION\_LIST >< DECLARATION >$ $| < DECLARATION >$
3. $< DECLARATION > \rightarrow < VAR\_DECLARATION > \ | \ < FUNCTION\_DECLARATION >$ $| \ < LIST\_DECLARATION >$
4. $< VAR\_DECLARATION > \rightarrow < SIMPLE\_TYPE >< ID >';' \ | \ <$ $SIMPLE\_TYPE >< ID > [< DIGIT >]';'$
5. $< FUNCTION\_DECLARATION > \rightarrow < TYPE > < ID > \ '(' <$ $PARAMS > \ ')' \ < BRACKETS >$
6. $< LIST\_DECLARATION > \rightarrow < TYPE > < LIST\_TYPE > \ ' =' <$ $ID >$
7. $< PARAMS > \rightarrow < TYPE >< ID > \ ','$
8. $< IF\_STATEMENT > \rightarrow \ if \ '(' < EXPRESSION\_STMT > \ ')' \ <$ $BRACKETS > < STATEMENT > < BRACKETS >$
9. $< FOR\_STATEMENT > \rightarrow \ for \ '(' < EXPRESSION\_STMT > \ ')' \ <$ $BRACKETS > < STATEMENT > < BRACKETS >$
10. $< RETURN\_STATEMENT > \rightarrow \ return \ ';' | return \ < EXPRESSION >$ $';'$
11. $< EXPRESSION\_STMT > \rightarrow < EXPRESSION >';'$
12. $< BRACKETS > \rightarrow \ '\{' \ | \ '\}'$
13. $< KEYWORD > \rightarrow \ if|else|for|return$
14. $< FILTER > \rightarrow \ ' <<'$
15. $< MAP > \rightarrow \ ' >>'$
16. $< TAIL > \rightarrow \ '!'|'\%'$
17. $< HEADER > \rightarrow \ '?'$
18. $< BINARY\_CONSTRUCTOR > \rightarrow \ ' :'$
19. $< OUTPUT > \rightarrow \ write|writeln$
20. $< INPUT > \rightarrow \ read$
21. $< BINARY\_COMP\_OP > \rightarrow \ \ll" \ | \ \ll =" \| \ \gg" \ | \ \gg =" \ | \ "! =" \ | \ " ==$ "
22. $< LOGIC\_OP > \rightarrow \ '|||' | \ '\&\&'$
23. $< BINARY\_BASIC\_OP > \rightarrow \ [+ * / -]$
24. $< ID > \rightarrow \ [a - zA - Z\_][a - z0 - 9A - Z\_]*$
25. $< STRING > \rightarrow ("([^\backslash"\backslash'])*")$
26. $< SIMPLE\_TYPE > \rightarrow < int > \ | \ < float >$
27. $< float > \rightarrow < DIGIT > +'.' < DIGIT > +$
28. $< int > \rightarrow < DIGIT > +$
29. $< DIGIT > \rightarrow \ [0 - 9]$
30. $< LIST\_CONSTANT > \rightarrow \ NIL$
31. $< LIST\_TYPE > \rightarrow \ list$

B Estrutura dos Tokens

Token	Lexema	Expressão Regular
<int, >	12	DIGIT+
<float, >	12.5	DIGIT+"."DIGIT+
<list, >	list	list
<list_constant, >	NIL	NIL
<digit, >	2	[0-9]
<string, >	"blablabla"	(\"([^\"\']) * \")
<id, >	int	$[a-zA-Z\_][a-z0-9A-Z\_]*$
<binary_basic_op, >	+	[+*/-]
<logic_op, >	&&	—"|"&&"
<binary_comp_op, >	≤	
<keyword, >	if	if | else | for | return
<input, >	INPUT	read
<output, >	OUTPUT	write|writeln
<binary_constructor, >	:	:
<header, >	?	?
<tail, >	%	"!"|"%"
<map, >	≫	"≫"
<filter, >	≪	"≪"