Analisador Léxico - Tradutores

Adelson Jhonata Silva de Sousa[18/0114913]

[1] Universidade de Brasília
[2] Departamento de Ciência da Computação cic@unb.br

1 Motivação

Esta primeira parte do trabalho de tradutores que aborda o analisador léxico tem como objetivo/motivação aplicar na prática os conceitos teóricos vistos em [1] e estudados até o momento na matéria.
Utilizando um subconjunto da linguagem C e adicionando uma nova primitiva de dados do tipo *list* temos uma nova linguagem que visa facilitar o tratamento de listas com operações necessárias para essa nova primitiva, como por exemplo operações de *filter, map, header e tail* da lista.
A lista é uma das estruturas de dados mais importantes que existem, pois serve para armazenar os dados em sequência, em C não existe uma primitiva do tipo lista, temos que manipular vários elementos para ter disponível essa estrutura de dados. A nova primitiva *list* adicionada a linguagem vem para facilitar a construção de programas que necessitam de listas, já que com as operações e os construtores disponíveis não precisamos utilizar ponteiros para o próximo elemento da lista e outras operações que fazemos em C.

2 Descrição da análise léxica

A analise léxica é feita utilizando o gerador léxico FLEX[2], e as regras da analise léxica estão detalhadas no arquivo scr/lexico.l, dentro desse mesmo arquivo também estão os contadores de linha e coluna para identificação de erro e uma função *main* para leitura do arquivo de teste. Todas as regras utilizam expressões regulares(regex) para identificar os padrões e sequências e todas as regras estão especificadas com seus respectivos nomes na tabela 1. Os *tokens* encontrados são exibidos na tela e já estão estruturados visando a tabela de símbolos utilizada na parte 2 desse trabalho, o analisador Sintático, e pode ser verificado no apêndice C e no arquivo **tabelaSimbolo.h** dentro do diretório *lib*.
Todos os *tokens* e erros são exibidos juntamente com a linha e a coluna correspondente, caso ocorra erro na analise léxica é exibida uma mensagem falando qual o foi o erro.
O analisador léxico irá analisar as regras da linguagem fornecida, que contém tipos básicos de variáveis(int e float) além de declaração da lista(int ou float list), estrutura condicional *if, else*, laço de repetição *for*, operações aritmética, operações logicas e relacionais e as operações da lista *map, filte, header, tail* e o construtor da lista.

3 Descrição dos arquivos testes

Os arquivos de testes desenvolvidos para a aplicação estao dentro da pasta *tests*. Os arquivos testeCerto.c e testeCerto2.c são os teste que não contém nenhum erro léxico. O arquivo testeCerto2.c usa o exemplo disponibilizado na descrição da linguagem. Os aqruivos testeErrado.c e testeErrado2.c, contém erros léxicos em sua execução.
testeErrado.c
- Linha: 2, Coluna: 13 Erro no token .
- Linha: 4, Coluna: 17 Erro no token .
- Linha: 4, Coluna: 18 Erro no token .
- Linha: 4, Coluna: 19 Erro no token .
- Linha: 6, Coluna: 14 Erro no token .

testeErrado2.c
- Linha: 2, Coluna: 9 Erro no token @
- Linha: 5, Coluna: 8 Erro no token @
- Linha: 5, Coluna: 18 Erro no token .
- Linha: 9, Coluna: 12 Erro no token .

4 Instruções para compilação e execução do programa

Dentro do diretório principal tem um arquivo *makefile* que gera o flex e compila o .c, bastando apenas entrar com o comando *make* no terminal. Para facilitar eu também coloquei dentro desse mesmo arquivo *make* todos o testes disponíveis. Para rodar todos os testes basta entrar com o comando *make run* no terminal. Caso não queira utilizar o make poderá usar os seguintes comandos no terminal para executar o programa:

1. cd src/
2. flex lexico.l
3. gcc-11 lex.yy.c -g -Wall -o tradutor ou gcc lex.yy.c -g -Wall -o tradutor
4. ./tradutor ../tests/nome_do_teste.c

O programa foi desenvolvido no ambiente Linux Ubuntu 20.04. Tentei me aproximar mais das versões descritas pela professora, mas teve algumas que não consegui. Então segue a configuração:

1. Kernel: 5.13.7-051307-generic
2. gcc-11: gcc version 11.0.1
3. flex: 2.6.4

References

1. A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. Compilers: Principles, Techniques, & Tools. Pearson/Addison Wesley, 2nd edition, 2006.
2. Lexical Analysis With Flex, for Flex 2.6.2, https://westes.github.io/flex/manual/. Acessado em 09 de Agosto de 2021.
3. Trabalho Prático - Descrição da Linguagem, https://aprender3.unb.br/mod/page/view.php?id=464034. Acessado em 09 de Agosto de 2021.

A Gramática

⟨*inicio*⟩ ::= ⟨*programa*⟩

⟨*programa*⟩ ::= ⟨*Lista de Declaração*⟩

⟨*Lista de Declaração*⟩ ::= ⟨*Lista de Declaração*⟩⟨*declaração*⟩ | ⟨*declaração*⟩

⟨*declaração*⟩ ::= ⟨*declaração de variável*⟩ | ⟨*declaração de função*⟩

⟨*declaração de variável*⟩ ::= ⟨*TIPO*⟩ ⟨*ID*⟩;

⟨*declaração de função*⟩ ::= ⟨*TIPO*⟩ ⟨*ID*⟩ (lista de parâmetros) { declarações }

⟨*lista de parâmetros*⟩ ::= (⟨*TIPO*⟩ ⟨*ID*⟩,)* | ⟨*TIPO*⟩ ⟨*ID*⟩ | vazio

⟨*declarações*⟩ ::= ⟨*declarações*⟩ | ⟨*declaração de variável*⟩
| ⟨*condicional*⟩
| ⟨*entrada*⟩
| ⟨*saída*⟩
| ⟨*for*⟩
| ⟨*return*⟩
| ⟨*expressão*⟩
| ⟨*declaração list*⟩
| ⟨*expressão list*⟩
| ⟨*NIL*⟩

⟨*expressão*⟩ ::= ⟨*expressão*⟩ =* ⟨*expressão*⟩ | ⟨*expressão*⟩;

⟨*declaração list*⟩ ::= ⟨*TIPO*⟩ ⟨*ID*⟩;

⟨*return*⟩ :: = return ⟨*INT*⟩; | ;

⟨*for*⟩ ::= for (⟨*expressão*⟩; ⟨*expressão*⟩; ⟨*expressão*⟩) ⟨*declarações*⟩

⟨*condicional*⟩ ::= if(⟨*expressão*⟩){ ⟨*declarações*⟩ }
| if(⟨*expressão*⟩){ declarações } else { ⟨*declarações*⟩ }

⟨*entrada*⟩ ::= read(⟨*ID*⟩);

⟨*saída*⟩ ::= write(⟨*string*⟩ | ⟨*expressão*⟩); | writeln(⟨*string*⟩ | ⟨*expressão*⟩);

⟨*expressão list*⟩ ::= ?⟨*ID*⟩ | !⟨*ID*⟩ | ⟨*ID*⟩ = ⟨*ID*⟩ >> ⟨*ID*⟩ | ⟨*ID*⟩ = ⟨*ID*⟩ << ⟨*ID*⟩

⟨*NIL*⟩ ::= ⟨*NIL*⟩

⟨*TIPO*⟩ ::= ⟨*INT*⟩ | ⟨*FLOAT*⟩ | ⟨*INT LIST*⟩ | ⟨*FLOAT LIST*⟩

⟨*INT*⟩ ::= [0-9]+

⟨*FLOAT*⟩ ::= [0-9]+“.”[0-9]+

⟨*operações aritméticas*⟩ ::= `+|-|*|/`

⟨*operações logicas*⟩ ::= `&&|!||`

⟨*operações relacionais*⟩ ::= `<|<=|>|>=|==|!=`

⟨*construtor list*⟩ ::= `:`

⟨*operações list*⟩ ::= `?|%`

⟨*funções list*⟩ ::= `>>|<<`

⟨*ID*⟩ ::= [a-zA-Z_][_a-z0-9A-Z]*

B Tabela de expressões Regulares

Tabela 1. Regras e expressões regulares utilizadas.

Nome	Expressão Regular
COMENTARIO	”//”.*|”/*”.*|”*/”.*
ID	[a-zA-Z_][_a-z0-9A-Z]*
INT	[0-9]+
FLOAT	[0-9]+”.”[0-9]+
TIPOS	int|float|”int list”|”float list”
OP_B_ARITMETICAS	”+”|”-”|”*”|”/”
OP_LOGICAS	”&&”|”!”|”||”
OP_B_RELACIONAIS	”<”|”<=”|”>”|”>=”|”==”|”!=”
CONTROLEFLUXO	if|else|for|return
ENTRADA	read
SAIDA	write|writeln
CONSTRUTOR_LISTA	”:”
OP_LISTA	”?”|”%”
FUNCOES_LISTA	”>>”|”<<”
STRING	\”[^\”]*\”|’[^’]*’
NIL	NIL
PARENTESES	”(”|”)”
CHAVES	”{”|”}”
ATRIBUICAO	”=”
VIRGULA	”,”
PONTOVIRGULA	”;”
COLCHETES	”[”|”]”
QUEBRALINHA	\n|\r
TABESPACO	[\t]+|[ ]+

C Estrutura da tabela de símbolos

```
#ifndef TABELA_SIMBOLO_H
#define TABELA_SIMBOLO_H
```

```
typedef struct{
    char *id;
    char *token;
    struct TabelaSimbolo *proximo;
} TabelaSimbolo;

TabelaSimbolo *inicio, *final;

#endif
```