Analisador Sintático - Tradutores

Adelson Jhonata Silva de Sousa - 18/0114913

[1] Universidade de Brasília
[2] Departamento de Ciência da Computação
180114913@aluno.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 [ALSU06] 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 conforme descrição da linguagem em [Nal]. Essa nova linguagem 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 análise léxica é feita utilizando o gerador léxico FLEX [Est], e as regras da análise 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 segunda parte deste trabalho, o analisador Sintático, e podem ser verificados 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 análise léxica é exibida uma mensagem falando qual foi o erro.

Cada *token* tem uma linha, uma coluna, um símbolo, e um valor de escopo. O símbolo é o identificador do *token*. Os *tokens* são enviados para o analisador sintático e tratado para ser inseridos na tabela de símbolos.

O analisador léxico irá analisar a linguagem fornecida, utilizando regras 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 aritméticas, operações lógicas e relacionais, e as operações da lista *map, filter, header, tail* e o construtor da lista.

3 Descrição da análise sintática

A análise sintática foi feita utilizando um arquivo descrito em yacc e o programa *bison* [Bis], na versão 3.7.4. O *bison* é *software* que faz o parser da gramática descrita no arquivo yacc com regras de derivação LR(1).

No topo do arquivo foi definida a regra **%define lr.type canonical-lr** para que a derivação seja feita utilizando as regras LR(1) canonical. LR(1) é *bottom-up*, ou seja, de baixo para cima, e a árvore é gerada do final até o início.

O desenvolvimento do projeto está sendo feito seguindo algumas instruções descritas em [Aab].

3.1 Tabela de Símbolos

A tabela de símbolos é responsável por armazenar todos os símbolos durante o parser da gramática. São armazenadas em um vetor as informações/tokens necessárias para a utilização na análise semântica, como por exemplo variáveis declaradas, chamadas de função, escopo das variáveis, entre outros. Uma parte do código da tabela de símbolos está descrita no apêndice C.

Os *tokens* recebidos da analise léxica são tratados e inseridos na tabela de simbolo sempre que é declarada uma variável ou função, além de chamadas de funções também.

Além do identificador de cada *token* também é armazenada a linha, coluna, o tipo e o escopo em que se encontra.

Na figura 1 está a representação gráfica da tabela de símbolos do arquivo testeCerto.c.

Figura 1. Tabela de Símbolos.

Tabela de Simbolos

Simbolo	Tipo Entrada	Escopo	Linha	Coluna
x	Variavel	1	2	9
y	Variavel	1	3	11
a	Variavel	1	4	11
m	Variavel	1	6	9
m	Variavel	2	8	13
m	Variavel	3	10	17
asd	Variavel	3	11	17
abc	Variavel	1	21	9
main	Funcao	0	1	5

3.2 Árvore Sintática Abstrata

A árvore sintática abstrata representa o fluxo do parser bottom-up da gramática descrita. Onde cada nó representa um não terminal e seus filhos representam os terminais, que são os símbolos. Cada nó tem um ponteiro apontando para o próximo nó e pode ter ou não filhos.

Assim podemos percorrer a árvore em profundidade e verificar se está tudo de acordo com o descrito no programa.

Ao finalizar o parser da gramatica e do input, a árvore é exibida graficamente se não tiver nenhum erro, se tiver erro a arvore não é mostrada e mostra apenas a tabela de símbolos.

4 Descrição dos arquivos testes

Os arquivos de testes desenvolvidos para a aplicação estão dentro da pasta *tests*. Os arquivos testeCerto.c e testeCerto2.c são os testes que não contém nenhum erro léxico ou sintático. O arquivo testeCerto2.c usa o exemplo disponibilizado na descrição da linguagem.

Os arquivos testeErrado.c e testeErrado2.c, contém erros léxicos e sintáticos em sua execução.
testeErrado.c
- Linha: 8, Coluna: 12 Construtor da Lista: :
- syntax error, unexpected CONSTRUTOR_LISTA, expecting ID
testeErrado2.c
- Linha: 3, Coluna: 11 Ponto e virgula: ;
- syntax error, unexpected PONTOVIRGULA
- Linha: 6, Coluna: 12 Igual de atribuicao: =
- syntax error, unexpected ATRIBUICAO, expecting ID or INT or FLOAT or ABRE_PARENTESES

5 Instruções para compilação e execução do programa

Dentro do diretório principal tem um arquivo *makefile* gera o autômata (em C) a partir do arquivo de entrada do 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 os 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. bison -d sintatico.y
3. flex lexico.l
4. gcc-11 -g -c arvore.c -o arvore.o
5. gcc-11 -g -c tabelaSimbolo.c -o tabelaSimbolo.o
6. gcc-11 sintatico.tab.c lex.yy.c arvore.o tabelaSimbolo.o -g -Wall -o ../tradutor

7. cd ../
8. ./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
4. bison 3.7.4

Referências

[Aab] Anthony A. Aaby. Compiler construction using flex and bison. https://dlsiisv.fi.upm.es/traductores/Software/Flex-Bison.pdf. Acessado em 06 de Setembro de 2021.

[ALSU06] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, & Tools.* Pearson/Addison Wesley, second edition, 2006.

[Bis] GNU Bison. The yacc-compatible parser generator. manual of bison. https://www.gnu.org/software/bison/manual/bison.html. Acessado em 28 de Agosto de 2021.

[Est] Will Estes. Lexical analysis with flex, for flex 2.6.2. https://westes.github.io/flex/manual/. Acessado em 09 de Agosto de 2021.

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

A Gramática

⟨*inicio*⟩ ::= ⟨*programa*⟩

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

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

⟨*TIPO*⟩ ::= ⟨*TIPO_INT*⟩ | ⟨*TIPO_FLOAT*⟩ | ⟨*TIPO_LIST_INT*⟩ | ⟨*TIPO_LIST_INT*⟩

⟨*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*⟩ , ⟨*lista de parâmetros*⟩ | ⟨*TIPO*⟩ ⟨*ID*⟩ |
vazio

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

⟨*expressão*⟩ ::= ⟨*exp*⟩ ; | ⟨*expressão List*⟩ | ⟨*ID*⟩ = ⟨*expressão*⟩ | ⟨*ID*⟩ = ⟨*NIL*⟩;

⟨*exp*⟩ ::= ⟨*expressao logica*⟩ | !exp ;

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

⟨*return*⟩ :: = return expressão; | ;

⟨*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*⟩

⟨*expressao logica*⟩ :: = expressao logica OR exp
| expressao logica AND exp
| exp

⟨*expressao relacional*⟩ ::= ⟨*expressao relacional*⟩ ⟨*operacoes relacionais*⟩ ⟨*exp*⟩

⟨*operaçao soma_sub*⟩ ::= ⟨*operaçao soma_sub*⟩ OP_SOMA_SUB ⟨*exp*⟩;

⟨*operaçao mult_div*⟩ ::= ⟨*operaçao mult_div*⟩ OP_MULT_DIV ⟨*argumento*⟩;

⟨*argumento*⟩ ::= ⟨*ID*⟩
| ⟨*NUMERO*⟩
| (exp)
| ⟨*Chamada de Função*⟩

⟨*Chamada de Função*⟩ ::= ID (exp)

⟨*NUMERO*⟩ :: ⟨*INT*⟩ | ⟨*FLOAT*⟩

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 TabelaSimbolo{
    int escopo;
    char simbolo[100];
    int linha;
    int coluna;
    char tipoEntrada[50];
    struct TabelaSimbolo* proximo;
} TabelaSimbolo;

TabelaSimbolo* insereSimbolo(TabelaSimbolo* id,
int escopo, char* simbolo, char* tipoEntrada,
int linha, int coluna, int parametros);

void mostraTabela(TabelaSimbolo* id);

void limpaTabela(TabelaSimbolo* id);

#endif
```