Analisador Sintático

Bruno Cordeiro Mendes

Universidade de Brasília
150007094@aluno.unb.br

Abstract. Neste trabalho é apresentada a especificação do analisador sintático para a linguagem C-IPL, que traz facilidade no tratamento de listas em programas escritos em C. O analisador léxico será parte do compilador construído ao longo da disciplina de Tradutores ministrada pela professora Cláudia Nalon.

Keywords: Tradutores · Analisador Léxico · Compilador

1 Motivação

Este trabalho tem como principal motivador exercer o conhecimento adquirido na disciplina de Tradudores através da construção de um analisador sintático para uma linguagem fornecida na descrição do trabalho. O analisador sintático foi construído utilizando como base os conhecimentos adquiridos na leitura do livro-texto [ALSU06]. A linguagem em questão utiliza um nova primitiva de dados para listas e foi projetada para facilitar as operações entre os mesmos em programas escritos na linguagem C. Nessa linguagem temos um novo tipo de dado chamado "list", e novas funções e operadores para trabalhar com listas como o *filter*, *map*, *header* e *tail*. A gramática dessa linguagem se encontra no Apêndice A. Ela foi construída com base na gramática da própria linguagem C e na especificação da linguagem C-IPL fornecida na especificação do projeto [Nal].

2 Descrição da Análise Léxica

O analisador trabalha de forma a identificar lexemas dentro do programa utilizando REGEX e classificando-os em tokens. A lista de tokens e descrição dos mesmos se encontra no Apêndice B. O código flex criado para gerar o analisador teve como referência o tutorial fornecido por [Lev09]. Para gerar as cores no terminal de saída foi utilizado como referência [Rab]. A saída para os tokens obtidos ficou da seguinte forma:

– (*linha*, *coluna*) TOKEN: *token*, LEXEMA: *lexema*

onde *linha* representa o número da linha do token, *coluna* o número da coluna do token, *token* o próprio token encontrado, impresso no terminal com cor magenta, e *lexema* sendo o lexema encontrado, impresso na cor amarela. Os erros aparecem de vermelho com a frase "Expressão *expressão* não reconhecida".

3 Descrição da Análise Sintática

A análise sintática será responsável por pegar os lexemas e *tokens* passados pelo analisador léxico e criar uma árvore utilizando da gramática definida no Apêndice A. Além disso o analisador irá criar a tabela de símbolos que será usada em fases posteriores do projeto da disciplina. Por enquanto neste envio só possuo a implementação da tabela de símbolos. Para isso foi utilizada a seguinte estrutura de dados para armazenar informação sobre os *tokens*:

```
struct Token{
    int line;
    int column;
    int scope;
    char lexeme[100];
} token;
```

a qual contêm informações sobre linha, coluna, escopo e lexema lido do arquivo contendo a linguagem C-IPL.

Para criar a tabela de símbolos, foi definido um outro tipo de estrutura de dados:

```
typedef struct{
  char token[20];
  char lexeme[50];
  char type[20];
  int scope;
  int line;
  int column;
  int filled;
  char decl[3];
} Symbol;
```

nessa estrutura temos as informações de token, lexema, tipo, que poder inteiro, float, int, int list ou float list, escopo, linha, coluna, *filled* que nos informa se o token já foi ou não inserido na tabela e *decl* que possui informação se o símbolo é uma variável, função ou parâmetro de função. Para popular a tabela de símbolos com a estrutura *Symbol* foi criado um array de 1000 posições do tipo *Symbol*. Os símbolos são inseridos no array sempre que forem encontradas regras do tipo <var_decl>ou <fun_decl>.

4 Descrição dos arquivos de teste

O analisador vem acompanhado de quatro arquivos de teste, sendo eles *ex1.txt* e *ex2.txt* com código correto e *ex3.txt* e *ex4.txt* com código contendo erros. O erro do arquivo *ex3.txt* é a declaração de função *int read_list(int n,y)*, e o erro do arquivo 2 se encontra na declaração *int i = {14};*.

5 Instruções para compilação e execução do programa.

O arquivo flex se encontra dentro da pasta **src** no arquivo *flex.l*. E o arquivo bison se encontra no arquivo bison.y. O diretório principal já contém um arquivo *makefile*, portanto para compilar e gerar o executável, chamado *tradutor*, digite:

```
make run
```

lembrando que os comandos no makefile estão configurados para o uso do gcc-11 portanto certifique-se de que você tenha o mesmo instalado na sua máquina.

Execute o programa passando como argumento um dos arquivos de teste que se encontram na pasta "tests", como no exemplo abaixo:

```
./tradutor < tests/ex1.txt
```

Com isso serão gerados os tokens identificados dentro do arquivo de exemplo passado para o programa.

References

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[Lev09] John Levine. *Flex & Bison*. O'Reilly Media, Inc., 1st edition, 2009. Último acesso em 15 de agosto de 2021.

[Nal] Claudia Nalon. Tradutores - 2021/1 - trabalho prático - descrição da linguagem. https://aprender3.unb.br/mod/page/view.php?id=464034. Último acesso em 10 de agosto de 2021.

[Rab] RabaDabaDoba. The entire table of ANSI color codes working in C! https://gist.github.com/RabaDabaDoba/145049536f815903c79944599c6f952a. Último acesso em 15 de agosto de 2021.

A Gramática da Linguagem

⟨*S*⟩ ::= ⟨*decl_list*⟩

⟨*decl_list*⟩ ::= ⟨*decl_list*⟩ ⟨*decl*⟩ | ⟨*decl*⟩

⟨*decl*⟩ ::= ⟨*var_decl*⟩ | ⟨*fun_decl*⟩

⟨*var_decl*⟩ ::= **TYPE ID;**

⟨*fun_decl*⟩ ::= **TYPE ID (** params **)** ⟨*block_stmt*⟩
| **TYPE ID ( )** ⟨*block_stmt*⟩

⟨*params*⟩ ::= ⟨*params*⟩ **,** ⟨*param_decl*⟩ | ⟨*param_decl*⟩

⟨*param_decl*⟩ ::= **TYPE ID**

⟨*statement*⟩ ::= ⟨*exp_stmt*⟩
| ⟨*block_stmt*⟩
| ⟨*if_stmt*⟩
| ⟨*return_stmt*⟩
| ⟨*write_stmt*⟩
| ⟨*writeln_stmt*⟩
| ⟨*read_stmt*⟩
| ⟨*var_decl*⟩
| ⟨*for_stmt*⟩

⟨*for_stmt*⟩ ::= FOR **(** ⟨*assing_exp*⟩ **;** ⟨*rel_exp*⟩ **;** ⟨*assign_exp*⟩ **)** ⟨*block_stmt*⟩

⟨*exp_stmt*⟩ ::= ⟨*exp*⟩**;** | **;**

⟨*exp*⟩ ::= ⟨*assing_exp*⟩ | ⟨*simple_exp*⟩

⟨*assing_exp*⟩ ::= **ID ASSIGN** ⟨*exp*⟩

⟨*block_stmt*⟩ ::= **{** ⟨*stmt_list*⟩ **}**

⟨*stmt_list*⟩ ::= ⟨*statement*⟩ ⟨*stmt_list*⟩ | ⟨*statement*⟩

⟨*if_stmt*⟩ ::= **IF (** ⟨*bin_exp*⟩ **)** ⟨*block_stmt*⟩ | **IF (** ⟨*bin_exp*⟩ **)** ⟨*block_stmt*⟩
ELSE ⟨*block_stmt*⟩

⟨*return_stmt*⟩ ::= **RETURN ;** |**RETURN** ⟨*exp*⟩**;**

⟨*write*⟩ ::= **WRITE(**⟨*simple_exp*⟩**);**

⟨*writeln*⟩ ::= **WRITELN(**⟨*simple_exp*⟩**);**

⟨*read*⟩ ::= **READ(ID);**

⟨*simple_exp*⟩ ::= ⟨*bin_exp*⟩ | ⟨*list_exp*⟩

⟨*list_exp*⟩ ::= ⟨*list_exp*⟩ **LIST_OP** ⟨*unary_list_exp*⟩ | ⟨*unary_list_exp*⟩

⟨*unary_list_exp*⟩ ::= **UNARY_LIST_OP ID**

⟨*bin_exp*⟩ ::= ⟨*bin_exp*⟩ **LOG_OP** ⟨*unary_log_exp*⟩ | ⟨*unary_log_exp*⟩

⟨*unary_log_exp*⟩ ::= **UNARY_LOG_OP** ⟨*unary_log_exp*⟩
| **EXCLAMATION** ⟨*unary_log_exp*⟩
| ⟨*rel_exp*⟩

⟨*rel_exp*⟩ ::= ⟨*rel_exp*⟩ **REL_OP** ⟨*sum_exp*⟩ | ⟨*sum_exp*⟩

⟨*sum_exp*⟩ ::= ⟨*sum_exp*⟩ **SUM_OP** ⟨*mul_exp*⟩ | ⟨*mul_exp*⟩

⟨*mul_exp*⟩ ::= ⟨*mul_exp*⟩ **MUL_OP** ⟨*factor*⟩ | ⟨*factor*⟩ | **SUM_OP** ⟨*factor*⟩

⟨*factor*⟩ ::= ⟨*immutable*⟩ | **ID**

⟨*immutable*⟩ ::= **(** simple_exp **)** | ⟨*call*⟩ | ⟨*constant*⟩

⟨*call*⟩ ::= **ID(** ⟨*args*⟩ **)** | **ID()**

⟨*args*⟩ ::= ⟨*args*⟩ **,** ⟨*simple_exp*⟩ | ⟨*simple_exp*⟩

⟨*constant*⟩ **NIL** | **INT** | **FLOAT** | **STRING**

B Léxico

Token	Regex	Descrição
IF	if	Condicional
ELSE	else	Condicional
TYPE	int |float |int list |float list	Palavra reservada para tipo de dado
FOR	for	Palavra reservada para iteração
WRITE	write	Comando de saída
WRITELN	writeln	Comando de saída com quebra de linha
READ	read	Comando de entrada
RETURN	return	Palavra reservada para retorno de função
LOG_OP	&&|||	Operador lógico "OR" ou "AND"
SUM_OP	[+-]	Operador binário de soma ou subtração
MUL_OP	[*/]	Operador binário de multiplicação ou divisão
REL_OP	< | <= | > | >= | == |! =	Operador binário relacional
LIST_OP	: | << | >>	Construtor binário infixo de listas
UNARY_LIST_OP	%|?	Construtor unário infixo de listas
EXCLAMATION	!	Operador lógico de negação ou operador unário para listas
ASSIGN	=	Operador de atribuição
,	,	Vírgula
;	;	Ponto e vírgula
(	}	Abertura de escopo
)	}	Fechamento de escopo
{	(	Abertura de parênteses
}	)	Fechamento de parênteses
ID	[_a-zA-Z][_a-zA-Z0-9]*	Nome de função ou variável. Ex: x, var1, encontraElement
INT	-?[0-9]*	Número inteiro. Ex: 1, 23, -23.
FLOAT	-?[0-9]*"."[0-9]*	Número flutuante. Ex: -13.4, 4.87.
NIL	NIL	Constante para representar lista vazia.
STRING	\"(\\.|[^"\\])*\"	Literal. Ex: "Banana", "Hello World\n"