Analisador Semântico

Bruno Cordeiro Mendes

Universidade de Brasília
150007094@aluno.unb.br

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

Keywords: Tradutores · Analisador Léxico · Analisador Sintático · Compilador

1 Motivação

Este trabalho tem como principal motivador exercer o conhecimento adquirido na disciplina de Tradudores através da construção de um analisador semântico para uma linguagem fornecida na descrição do trabalho prático. 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 um autômato, criado a partir de expressões regulares definidas no programa, 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 "Expression *expressão* not recognized".

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 a gramática definida no Apêndice A. Os lexemas são utilizados como atributos do token para que assim eles possam ser utilizados na impressão da árvore e da tabela de símbolos. Além disso a tabela de símbolos, criada pela analisador, será usada em fases posteriores do projeto da disciplina. Para o bison armazenar informações sobre os tokens passados pelo léxico foi utilizada a seguinte estrutura de dados:

```
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[50];
  char lexeme[150];
  char type[20];
  int scope;
  int line;
  int column;
  char decl[5];
} Symbol;
```

onde nessa estrutura temos as informações de token, lexema, tipo (float, int, int list ou float list), escopo, linha, coluna 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 uma estrutura de listas chamada de *SymbolList*, que possui um ponteiro para *Symbol* e um ponteiro para o próximo elemento da lista. A estrutura criada para a lista ficou da seguinte maneira:

```
typedef struct symbolList{
    Symbol *symbol;
    struct symbolList* next;
} SymbolList;
```

Os símbolos são inseridos na lista sempre que forem encontradas regras do tipo <var_decl>, <var_decl_with_assign>, <fun_decl>, ou <param_decl >. Para armazenar a informação do escopo foi simulada uma estrutura de pilha utilizando um array. Sempre que é encontrado uma abertura de chaves é inserido

um elemento nessa pilha e quando for encontrado um fechamento de chaves ocorre a remoção do último elemento inserido na pilha. O elemento que é inserido no topo da pilha sempre é o valor de um inteiro que é incrementado toda vez que é encontrado abertura de chaves.

A seguinte estrutura de dados foi usada para a criação da árvore:

```
typedef struct node
{
  struct node* leaf1;
  struct node* leaf2;
  struct node* leaf3;
  struct node* leaf4;
  struct node* leaf5;
  struct symbol *token;
  char name[50];
} Node;
```

onde *leaf1*, *leaf2*, *leaf3*, *leaf4*, *leaf5* são os filhos que um nó da árvore pode ter; *token* sendo o ponteiro para um *Symbol*, pois um nó da árvore pode ser um token passado pelo léxico; e *name* sendo um nome que esse nó pode armazenar nos casos em que for criado um nó contendo uma regra da gramática.

4 Descrição da Análise Semântica

A análise semântica será responsável por realizar a checagem de tipos assim como a detecção de demais erros semânticos na linguagem. As verificações são feitas dentro das regras da gramática. Regras de declaração de variáveis e regras que utilizam verificadores chamam funções que percorrem a tabela de símbolo e verificam se o identificador utilizado na regra já foi declarado ou não. Além disso o escopo também é utilizado dentro dessas funções para a verificação do token na tabela de símbolos leve em consideração o escopo no qual a variável está sendo utilizada. Por enquanto o analisor léxico engloba as seguintes verificações semânticas:

- Presença de função main
- Declaração de variável
- Redeclaração de variável
- *Too many arguments*
- *Too few arguments*

5 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. Os erro do arquivo *ex3.txt* são:

```
SEMANTIC ERROR -> Undeclared 'i'. Line 6 Column 2
SEMANTIC ERROR -> Undeclared 'i'. Line 9 Column 7
SEMANTIC ERROR -> Undeclared 'i'. Line 9 Column 14
SEMANTIC ERROR -> Undeclared 'i'. Line 9 Column 26
SEMANTIC ERROR -> Undeclared 'i'. Line 9 Column 22
SEMANTIC ERROR -> Undeclared 'elem'. Line 12 Column 8
SEMANTIC ERROR -> undefined reference to "main"
```

e do arquivo 4 sao

```
SEMANTIC ERROR -> Undeclared 'elem'. Line 13 Column 14
SEMANTIC ERROR -> undefined reference to "main"
```

Quando erros são encontrados a seguinte saída é mostrada no terminal:

– Error ➔*erro_sintatico* [Line *linha*, Column *coluna*]

6 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_with_assing*⟩

⟨*var_decl*⟩ ::= **TYPE ID;**

⟨*var_decl_with_assing*⟩ ::= **TYPE ID ASSIGN** ⟨*simple_exp*⟩ **;**

⟨*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*⟩
| ⟨*var_decl_with_assing*⟩
| ⟨*for_stmt*⟩

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

⟨*exp_stmt*⟩ ::= ⟨*exp*⟩**;** | **;**

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

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

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

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

⟨*if_stmt*⟩ ::= **IF (** ⟨*simple_exp*⟩ **)** ⟨*statement*⟩ | **IF (** ⟨*simple_exp*⟩ **)** ⟨*statement*⟩
ELSE ⟨*statement*⟩

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

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

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

⟨*read*⟩ ::= **READ(ID);**

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

⟨*list_exp*⟩ ::= ⟨*factor*⟩ **:** ⟨*factor*⟩
| **EXCLAMATION** ⟨*factor*⟩
| **%** ⟨*factor*⟩
| ⟨*factor*⟩ **MAP** ⟨*factor*⟩
| ⟨*factor*⟩ **FILTER** ⟨*factor*⟩

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

⟨*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
:	:	Construtor binário infixo de listas
?	?	Operador unário infixo de listas
%	%	Operador unário infixo de listas
MAP	<<	Operador binário infixo de listas
FILTER	>>	Operador binário 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, encontraElemento.
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"