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 · Analisador Semântico · 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. 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 symbol
{
    int line;
    int column;
    int scope;
    int numberOfParams;
    char token[50];
    char lexeme[150];
    char type[20];
    char decl[5];
    int typeParameters[150];
} 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. Eu poderia ter criado um ponteiro para *token* nas minhas estruturas de *Node* e *Symbol*, para não ter que armazenar as duas strings na árvore, porém quando me dei conta disso já era tarde demais, e espero poder fazer essa correção nas minhas próximas entregas.

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. Desde a última entrega do trabalho foram adicionado dois novos atributos, um chamado *numberOfParams* responsável por armazenar a quantidade de parâmetros de uma função e o outro chamado *typeParameters* para armazenar os tipos dos parâmetros de uma função. 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>, <fun_decl>, ou <param_decl >. Para armazenar a informação do escopo foi simulada uma estrutura de pilha utilizando um array. Sempre que é encontrada 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 token *token;
    char name[50];
    int type;
} 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. Além disso foi adicionado um atributo *type* para armazenar o tipo do nó, assim como o casting que pode estar ocorrendo neste mesmo nó. va

4 Descrição da Análise Semântica

A análise semântica será responsável por realizar a checagem de tipos assim como a detecção de erros semânticos na linguagem. Para facilitar o armazenamento e comparação entre tipos optei por associar uma id para cada tipo da linguagem, sendo 0 para int, 1 para float, 2 para int list e 3 para float list. Para tipos não reconhecidos é utilizado o id -1. As verificações são feitas dentro das regras da gramática em apensas **uma passagem**. Regras de declaração de variáveis e demais regras que utilizam identificadores (ID) chamam funções que percorrem a tabela de símbolo e verificam se o identificador utilizado na regra já foi declarado ou não. O escopo também é utilizado dentro dessas funções para que se leve em

consideração o escopo no qual a variável está sendo utilizada. Em verificações de retorno de função, conversão implícita de tipos, verificação de número de parâmetros de função e etc, são utilizadas funções que recebem como argumento um nó da árvore, esse nó então é percorrido utilizando uma função recursiva e fazendo as devidas verificações e *casting's* . O analisador semântico detecta os seguintes erros:

- Ausência de função main
- Variável não declarada
- Redeclaração de variável
- Quantidade de argumentos passados em chamada de função maior que quantidade de parâmetros esperado (*Too many arguments*)
- Quantidade de argumentos passados em chamada de função menor que quantidade de parâmetros esperado (*Too few arguments*)
- Primeiro argumento de operador MAP não sendo função unária
- Primeiro argumento de operador FILTER não sendo função unária
- Tipo de retorno da função diferente do tipo declarado para função (Exceto em casos envolvendo int e float, nesses casos ocorre *cast*)
- Uso de listas em expressões que envolvem operadores de soma (+, -) ou multiplicação(*, /).
- Tentativa de atribuição de uma expressão resultante em int ou float em uma lista. Exemplo: a = 6, com a sendo uma variável do tipo float list.
- Argumento com tipo incorreto em chamada de função.
- Uso de variáveis que não sejam do tipo *float list* ou *int list* em expressões binárias e unárias de listas. Exemplo: ?a, com a sendo uma variável do tipo float.

4.1 Conversão implícita de tipos

A árvore é anotada com os tipos de cada nó e ocorre conversão implícita em 3 situações. A primeira é quando existe uma operação entre *int* e *float*, nesse caso como *float* é um tipo de maior hierarquia, ocorre uma conversão de int para float no fator contendo o tipo *int*. Essa conversão é anotada na árvore como **intToFloat**. A segunda ocorre nos casos em que é declarada uma variável do tipo *int* e acontece uma atribuição de uma variável ou constante do tipo float nessa variável. Nesses casos há uma conversão de *float* para *int* com perda da parte decimal do *float*, e é apresentada na árvore como **floatToInt**. E o terceiro caso acontece quando temons uma constante NIL sendo utilizada em operadores de lista. Nessas situações NIL acaba sendo carregada com o tipo da variável com a qual está interagindo. Exemplo: "FIL != NIL", com FIL sendo uma variável do tipo *float list*. Nessa expressão NIL é anotada na árvore como sendo do tipo *float list*. Vale lembrar que expressões lógicas e relacionais podem ter dois retornos, uma para indicar que a expressão é verdadeira e outra para indicar que a expressão é falsa. Para representar esses valores será utilizado o tipo *int*, portando nos nós da árvore contendo operadores lógicos estarão indicadas essas tipagem.

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:

- Função espero retorno dp tipo "float" porém retorno é do tipo "int list". Linha 17, Coluna 9.
- Função "belo" deve ser unária. Linha 38, Coluna 23.
- write sem argumentos. Linha 43, Coluna 12;
- Atribuição sem nada depois do sinal "=". Linha 49, Coluna 12.

e os erros do arquivo 4 são:

- Declaração de parâmetro de função sem acompanhamento do tipo. Linha 4, Coluna 24.
- *else* sem precedência de *if*. Linha 13, Coluna 14.
- Variável *elem* não declarada. Linha 13, Coluna 14.
- Referência não definida para *main*.

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

- SYNTATIC ERROR ➔*erro_sintatico* [Line *linha*, Column *coluna*]

ou:

- SEMANTIC ERROR ➔*erro_semantico*. 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* contendo os seguintes comandos:

```
run:
    bison -o src/bison/bison.tab.c -v -d src/bison/bison.y -Wcounterexamples -g
    flex -o src/flex/lex.yy.c src/flex/flex.l
    gcc-11 -g -c lib/symbol.table.c -o lib/symbol.table.o
    gcc-11 -g -c lib/scope.stack.c -o lib/scope.stack.o
    gcc-11 -g -c lib/node.c -o lib/node.o
    gcc-11 -g -o tradutor src/bison/bison.tab.c src/flex/lex.yy.c lib/symbol.table.o lib/scope.stack.o lib/node.o -I lib -I src/bison -I src/flex -lfl -g
    -Wall
```

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

Algumas das sugestões de correções feitas nas entregas anteriores não puderam ser feitas na gramática pois ocasionava um grande número de conflitos. Outras correções puderam ser feitas resultando na seguinte gramática:

⟨*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*⟩
| ⟨*for_stmt*⟩

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

⟨*exp_stmt*⟩ ::= ⟨*exp*⟩**;** | **;**

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

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

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

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

⟨*var_or_statement*⟩ ::= ⟨*statement*⟩ | ⟨*var_decl*⟩

⟨*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*⟩ | ⟨*bin_list_exp*⟩

⟨*bin_list_exp*⟩ ::= ⟨*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*⟩ | ⟨*unary_list_exp*⟩

⟨*factor*⟩ ::= ⟨*immutable*⟩ | **ID** | **SUM_OP** ⟨*factor*⟩

⟨*unary_list_exp*⟩ ::= **EXCLAMATION** ⟨*factor*⟩
| **%** ⟨*factor*⟩

⟨*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, float][\t]+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"