Analisador Léxico
Trabalho Prático de Tradutores

Gabriel Crespo de Souza - 14/0139982

Universidade de Brasília, Distrito Federal, Brasil
Departamento de Ciência da Computação
cic@unb.br
https://cic.unb.br/

1 Objetivos

A disciplina de Tradutores tem como finalidade fornecer o entendimento sobre o funcionamento dos diferentes tipos de tradutores, seus componentes, bem como as principais formas de implementá-los. Dado isso, o projeto principal da disciplina consiste na implementação de um compilador. Essa implementação se dará em etapas e esse documento apresentará o desenvolvimento das etapas de análise léxica e sintática, bem como suas integrações.

Ao nível de simplificação, um subconjunto da linguagem C foi escolhido, a C-IPL [Nal], juntamente com uma nova primitiva que implementa e permite a manipulação de conceitos relacionados à listas. Nativamente, esses recursos não fazem parte da linguagem C, portanto, a sua inclusão visa facilitar o uso dessa estrutura de dados e suas principais operações. Listas são estruturas muito importantes devido sua flexibilidade em relação ao espaço e eficiência em relação ao tempo.

2 Desenvolvimento

2.1 Análise léxica

O desenvolvimento do Analisador Léxico também se dá em fases. Inicialmente, as palavras-chave escolhidas para compor a linguagem servem como base para a construção das primeiras expressões regulares. Logo após, é feita a análise dos símbolos auxiliares. Por fim, são retornadas as sequências de *tokens* geradas, bem como a linha e coluna que foram encontrados. Em linhas gerais, os *tokens* não são nada além de registros(*structs*), onde o seu identificador é seu único atributo. Há variáveis de controle de linha e coluna, ambas variáveis globais. Por fim, há uma função com a responsabilidade de reportar qualquer erro léxico encontrado.

2.2 Análise Sintática

A fase de análise sintática segue a de análise léxica e conforme seu próprio nome já sugere, essa etapa é a responsável por avaliar o quão correta está a

escrita de um código fonte. Isto é, a sua responsabilidade é garantir que os *lexemas* enviados pelo Analisador Léxico sejam descritos de forma adequada aos seus programs [GC08]. Os identificadores dos *tokens* devem aparecer como símbolos terminais da gramática e seus atributos direcionados para a tabela de símbolos que possui mais informações sobre os *tokens* [ALSU07].

Tabela de símbolos As tabelas de símbolos são estruturas que guardam informações sobre os identificadores que podem aparecer no código fonte. Nesse projeto, foi utilizado o conceito de listas simplesmente encadeadas para a construção da tabela de símbolos, onde cada nó da lista representa um símbolo resgatado pelo Analisador Léxico, bem como suas informações essenciais, como: identificador, escopo, tipo de dado, tipo do símbolo e linha em que foi encontrado.

Árvore sintática abstrata Apesar de parcialmente construída e, no momento, omitida devido ao seu não correto funcionamento, a árvore sintática abstrata, que representa a estrutura hierárquica de um código fonte, está sendo construída a partir dos conceitos de árvore binária, onde as derivações de um símbolo terminal poderão crescer tanto para esquerda quanto para a direita. Assim, nessa estrutura, os nós intermediários irão representar os símbolos não terminais, as folhas irão representar os tokens presentes no código fonte e sua raiz irá representar o program que está sendo analisado.

3 Funcionamento

O Analisador Léxico irá basear-se nas definições regulares para detectar os *lexemas* da linguagem, depois retornar os tokens formados e a linha em que foram encontrados para o Analisador Sintático. Ao localizar os símbolos que não pertencem à linguagem, o Analisador Léxico deverá relatar o problema e indicar o seu local de ocorrência, sem que o problema encontrado interrompa o fluxo de execução do Analisador Léxico. Qualquer tipo de comentário será desconsiderado. Após isso, o Analisador Sintático irá receber os *tokens* gerados e validar se estão descritos corretamente no programa fonte, bem com popular a tabela de símbolos e gerar a árvore sintática abstrata. Ao validar que os *tokens* não estão descritos da forma correta, o Analisador Sintático deverá relatar o problema e indicar o seu local de ocorrência, sem que o problema encontrado interrompa o fluxo de execução do Analisador Sintático.

O projeto foi estruturado em diretórios, de modo que os arquivos principais estivessem separados dos auxiliares. Na raiz do projeto, */14_0139982*, há quatro pastas: */src*, */lib*, */doc* e */tests*, bem como os arquivos *makefile* e *README.md*. Em *src* estão os códigos dos analisadores e os arquivos de extensão *.c*. Em *tests* estão quatro arquivos de testes, *correct_1.c*, *correct_2.c*, *incorrect_1.c* e *incorrect_2.c*. Na pasta *doc* se encontra o relatório do projeto. Enquanto a pasta *lib* contêm os arquivos de cabeçalho *.h*. Para que a compilação e a execução do Analisador Léxico se deem com sucesso, será necessário seguir os seguintes passos estando no diretório raiz:

1. Executar o comando *make* e o seguinte arquivos será gerado:
   - *tradutor (arquivo executável)*;
   - *os outros arquivos gerados durante a compilação serão enviados a suas respectivas pastas*;
   - *obs.: há um problema no makefile que não permite compilar o programa por completo sem que o executável "tradutor"seja antes excluído. Portanto, caso queira compilar novamente, exclua o arquivo "tradutor"e chame o comando "make"*;
2. Agora, basta chamar o código objeto juntamente com um único arquivo de teste presente em */tests* como parâmetro por exemplo:
   - *./tradutor tests/correct_1.c.*;

4 Teste

Dos quatro arquivos disponibilizados para teste, dois apresentam erros léxicos e sintáticos(*incorrect_1.c e incorrect_2.c*) , enquanto dois estão corretos(*correct_1.c e correct_2.c*);

Referências

[ALSU07] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, Tools*. Pearson/Addison Wesley, 2nd edition, 2007.

[GC08] D. Grune and Jacobs C.J.H. *Parsing Techniques: A Pratical Guide*. Springer, 2nd edition, 2008.

[Nal] Cláudia Nalon. Trabalho prático - descrição da linguagem. https://aprender3.unb.br/mod/page/view.php?id=464034. Acessado pela última vez em 11/08/2021.

A Gramática

⟨*program*⟩ ::= ⟨*declarations*⟩

⟨*declarations*⟩ ::= ⟨*declarations*⟩ ⟨*declaration*⟩
| ⟨*declaration*⟩

⟨*declaration*⟩ ::= ⟨*variable_declaration*⟩
| ⟨*list_declaration*⟩
| ⟨*function_declaration*⟩

⟨*variable_declaration*⟩ ::= ⟨*variable_simple_declaration*⟩ ';'

⟨*list_declaration*⟩ ::= ⟨*list_simple_declaration*⟩ ';'

⟨*function_declaration*⟩ ::= ⟨*function_simple_declaration*⟩ '(' ⟨*parameters*⟩ ')' ⟨*compound_statement*⟩
| ⟨*function_simple_declaration*⟩ '(' ')' ⟨*compound_statement*⟩

⟨*parameters*⟩ ::= ⟨*list_parameter*⟩

⟨*list_parameters*⟩ ::= ⟨*list_parameter*⟩ ',' ⟨*parameter*⟩
| ⟨*parameter*⟩

⟨*parameter*⟩ ::= ⟨*variable_simple_declaration*⟩
| ⟨*list_simple_declaration*⟩
⟨*variable_simple_declaration*⟩ ::= ⟨*type_specifier*⟩ 'ID'
⟨*list_simple_declaration*⟩ ::= ⟨*type_specifier*⟩ 'LIST' 'ID'
⟨*function_simple_declaration*⟩ ::= ⟨*type_specifier*⟩ 'ID'
| ⟨*type_specifier*⟩ 'LIST' 'ID'
⟨*type_specifier*⟩ ::= 'INT'
| 'FLOAT'
⟨*compound_statement*⟩ ::= '{' ⟨*local_declarations*⟩ '}'
⟨*local_declarations*⟩ ::= ⟨*list_statements*⟩
⟨*list_statements*⟩ ::= ⟨*list_statements*⟩ ⟨*statement*⟩
| ⟨*statement*⟩
⟨*statement*⟩ ::= ⟨*expression_statement*⟩
| ⟨*expression_statement*⟩
| ⟨*compound_statement*⟩
| ⟨*conditional_statement*⟩
| ⟨*iteration_statement*⟩
| ⟨*return_statement*⟩
| ⟨*variable_declaration*⟩
| ⟨*list_declaration*⟩
| ⟨*in_out_declaration*⟩
⟨*declaration*⟩ ::= 'IF' '(' ⟨*expression*⟩ ')' ⟨*statement*⟩
| 'IF' '(' ⟨*expression*⟩ ')' ⟨*statement*⟩ 'ELSE ' ⟨*statement*⟩
⟨*declaration*⟩ ::= 'FOR' '(' ⟨*expression_statement*⟩ ⟨*expression_statement*⟩
⟨*expression*⟩ ')' ⟨*statement*⟩
⟨*expression_statement*⟩ ::= ⟨*expression*⟩ ';'
| ';'
⟨*return_statement*⟩ ::= 'RETURN' ⟨*expression*⟩ ';'
| 'RETURN' ';'
⟨*expression*⟩ ::= ⟨*assign_expression*⟩
| ⟨*simple_expression*⟩
| ⟨*list_expression*⟩
| ⟨*in_out_expression*⟩
⟨*assign_expression*⟩ ::= 'ID' '=' ⟨*expression*⟩
| 'ID' '=' 'NIL'
⟨*simple_expression*⟩ ::= '!' ⟨*relational_expression*⟩
| ⟨*simple_expression*⟩ ⟨*binary_logic_op*⟩ ⟨*relational_expression*⟩
| ⟨*simple_expression*⟩ ⟨*binary_logic_op*⟩ '!' ⟨*relational_expression*⟩
| ⟨*relational_expression*⟩
⟨*relational_expression*⟩ ::= ⟨*relational_expression*⟩ ⟨*relational_op*⟩ ⟨*arithmetic_add_expression*⟩
| ⟨*arithmetic_add_expression*⟩
⟨*arithmetic_add_expression*⟩ ::= ⟨*arithmetic_add_expression*⟩ ⟨*arithmetic_add_op*⟩
⟨*arithmetic_mul_expression*⟩
| ⟨*arithmetic_mul_expression*⟩

⟨*arithmetic_mul_expression*⟩ ::= ⟨*arithmetic_mul_expression*⟩ ⟨*arithmetic_mult_op*⟩ ⟨*unary_sub_expression*⟩
| ⟨*unary_sub_expression*⟩

⟨*unary_sub_expression*⟩ ::= '-' ⟨*factor*⟩
| ⟨*function_call*⟩
| 'ID'
| ⟨*INT_CONST*⟩
| ⟨*FLOAT_CONST*⟩

⟨*factor*⟩ ::= '(' ⟨*expression*⟩ ')'
| ⟨*factor*⟩

⟨*relational_op*⟩ ::= '<'
| '<='
| '>'
| '>='
| '=='
| '!='

⟨*arithmetic_add_op*⟩ ::= '+'
| '-'

⟨*arithmetic_mult_op*⟩ ::= '*'
| '/'

⟨*binary_logic_op*⟩ ::= '||'
| '&&'

⟨*list_expression*⟩ ::= ⟨*constructor*⟩
| ⟨*header*⟩
| ⟨*tail*⟩
| ⟨*map*⟩
| ⟨*filter*⟩
| ⟨*list_comparation*⟩

⟨*constructor*⟩ ::= ⟨*expression*⟩ ':' 'ID'

⟨*header*⟩ ::= '?' 'ID'

⟨*tail*⟩ ::= '%' 'ID'

⟨*map*⟩ ::= 'ID' '>>' 'ID'

⟨*filter*⟩ ::= 'ID' '<<' 'ID'

⟨*list_comparation*⟩ ::= 'ID' '==' 'NIL'
| 'ID' '!=' 'NIL'

⟨*in_out_expression*⟩ ::= ⟨*read*⟩ ';'
| ⟨*write*⟩ ';'

⟨*read*⟩ ::= 'READ' '(' 'ID' ')'

⟨*write*⟩ ::= 'WRITE' '(' ⟨*var*⟩ ')'
| 'WRITE' '(' ')'

⟨*function_call*⟩ ::= 'ID' '(' ⟨*arguments*⟩ ')'
| 'ID' '(' ')'

⟨*arguments*⟩ ::= ⟨*list_arguments*⟩

⟨*list_arguments*⟩ ::= ⟨*list_arguments*⟩ ‘**,**’ ⟨*expression*⟩
| ⟨*expression*⟩

⟨*var*⟩ ::= ⟨*STRING*⟩
| ⟨*CHAR*⟩
| ⟨*list_expression*⟩
| ⟨*simple_expression*⟩

B Expressões regulares

Token	Expressão
letter	[a-zA-Z]
digit	[0-9]
alphanumeric	{letter} | {digit}
identifier	{letter}+({alphanumeric}|_)*
integer	digit}*
float	[0-9]*(\.[0-9]+)
char	(\‘(.|\\a|\\b|\\f|\\n|\\r|\\t|\\v|\\\\\|\\‘|\\“|\\?)\‘)
string	\“[^\“]*\“ | ‘[^‘]*‘
main	“main“
int_type	“int“
float_type	“float“
list_type	“list“
const_nil	“NIL“
if	“if“
else	“else“
for	“for“
return	“return“
read	“read“
write	“write“
writeln	“writeln“
sum_op	“+“
sub_op	“-“
mult_op	“*“
div_op	“/“
list_header	“?”
list_constructor	“:“
list_tail	“%“
list_map	“>>“
list_filter	“<<“
exc_op	“!“
or_op	“||“
and_op	“&&“
equal_op	“==“
diff_op	“!=“
grt_op	“>“
lst_op	“<“
grt_eq_op	“>=“
lst_eq_op	“<=“
assign_op	“=“
r_paren	“(“
l_paren	“)“
r_brack	“{“
l_brack	“}“
semi	“.“
comma	“,“
reffer	“&“