UNIVERSIDADE FEDERAL DE SERGIPE
DEPARTAMENTO DE SISTEMAS DE INFORMAÇÃO
PROF. ANDRÉ LUIS MENESES SILVA
LINGUAGENS FORMAIS E TRADUTORES (SINF0012)

Alunos: Alcymar Marcolino dos Santos, Andeson Almeida Vasconcelos e Rafael Silveira de Andrade

Documentação do Projeto de Analisador Sintático

1. Introdução

Análise sintática (também conhecido pelo termo em inglês parsing) é o processo de analisar uma sequência de entrada (lida de um arquivo de computador ou do teclado, por exemplo) para determinar sua estrutura gramatical segundo uma determinada gramática formal. Essa análise faz parte de um compilador, junto com a análise léxica e análise semântica. A análise sintática, ou análise gramatical é o processo de se determinar se uma cadeia de símbolos léxicos pode ser gerada por uma gramática.

A análise sintática transforma um texto na entrada em uma estrutura de dados, em geral uma árvore, o que é conveniente para processamento posterior e captura a hierarquia implícita desta entrada. Através da análise léxica é obtido um grupo de tokens, para que o analisador sintático use um conjunto de regras para construir uma árvore sintática da estrutura.

2. Gramática

program	::= ‘procedure’ id ‘is‘ decl body | ‘procedure’ id ‘is‘ body
subprogram	::= ‘function’ id decl_param ‘is’ decl body | ‘function’ id decl_param ‘is’ body
body	::= ‘begin’ cmd_loop ‘end’ id ‘semicolon’
decl	::= var ‘semicolon’ decl | var ‘semicolon’ | subprogram decl | subprogram
var	::= id ‘colon’ ‘type’ ‘assign’ value | id ‘colon’ ‘type’ | var_loop id ‘colon’ ‘type’ | array
var_loop	::= var_loop id ‘comma’ | id ‘comma’
decl_param	::= ‘lparen’ param ‘rparen’ | ‘lparen’ param ‘rparen’ ‘return’ ’type’

param	::= id ‘colon’ ‘type’ ‘semicolon’ param | id ‘colon’ ‘type’ ‘semicolon’
function_call	::= : id ‘lparen’ param_pass ‘rparen’ ‘semicolon’
function_call_exp	::= : id ‘lparen’ param_pass ‘rparen’
param_pass	::= expression ‘comma’ param_pass | expression
value	::= number_int | number_float | number_expoent | boolean | string | char
cmd	::= if_statement | repeat_statement | puts | return | assign | function_call
cmd_loop	::= cmd_loop cmd | cmd
puts	::= ‘puts’ ‘lparen’ ‘string’ ‘rparen’ ‘semicolor’
if_statement	::= ‘if’ expression ‘then’ cmd_loop if_statement_loop
if_statement_loop	::= ‘elsif’ expression cmd_loop if_statement_loop | ‘else’ expression cmd_loop ‘end’ ‘if’ ‘semicolon’ | ‘end’ ‘if’ ‘semicolon’
repeat_statement	::= loop_statement | for_statement | while_statement
loop_statement	::= ‘loop’ cmd_loop ‘end’ ‘loop’
while_statement	::= ‘while’ expression number_inte ‘loop’ cmd_loop ‘end’ ‘loop’ ‘semicolon’
for_statement	::= ‘for’ id ‘in’ range ‘loop’ cmd_loop ‘end’ ‘loop’ ‘semicolon’
range	::= id ‘dotdot’ id

assign	::= id ‘assign’ op_arithmetic ‘semicolon’
expression	::= expression ‘and’ or_exp | or_exp
or_exp	::= or_exp ‘or’ comp_exp | comp_exp
comp_exp	::= comp_exp comp_op op_arithmetic | op_arithmetic
comp_op	::= greaterthan | greaterthanequal | lessthan | lessthanequal | notequal | equal
op_arithmetic	::= op_arithmetic PLUS factor | op_arithmetic MINUS factor | factor
factor	::= factor times power | factor divide power | power
power	::= power : power ‘power’ term | term
term	::= id | function_call_exp | ‘lparen’ expression ‘rparen’
array	::= ‘type’ id ‘is’ ‘array’ ‘lparen’ range ‘rparen’ ‘of’ ‘type’ ‘semicolon’
return	‘return’ expression ‘semicolon’

Por uma questão de otimização, deixamos apenas as regras da gramática que serão úteis ao projeto. A gramática BNF completa do ADA está disponível neste link: https://gist.github.com/Nihlus/f65e1a77487d5c571eb27f36cb024306