Centro Universitário de Belo Horizonte – Uni-BH
Curso: Ciência da Computação
UC: Teoria da Computação e Compiladores
Professor: Gustavo Alves Fernandes

Gramática (Corrigida) da linguagem PasC
(com ações semânticas)

prog → “program” “id” {TS.setTipo(ID.lexval, VOID)} body
body → decl-list “{“ stmt-list “}”
decl-list → decl “;” decl-list | ε
decl → type {id-list.t = type.t} id-list
type → “num” {type.t = NUM} | “char” {type.t = CHAR}
id-list → “id” {TS.setTipo(ID.lexval, id-list.t)} id-list’
id-list’ → “,” id-list | ε

stmt-list → stmt “;” stmt-list | ε
stmt → assign-stmt | if-stmt | while-stmt | read-stmt | write-stmt
assign-stmt → “id” {se TS.getTipo(ID.lexval) == null: **sinalizar erro para ID não declarado**}
“=” simple-expr
{se simple-expr.t != TS.getTipo(ID.lexval): **sinalizar erro de atribuição incompatível**}
if-stmt → “if” “(“ expression “)”
{se expression.t != BOOL: **sinalizar erro de expressão lógica mal formada**}
“{“ stmt-list “}” if-stmt’
if-stmt’ → “else” “{“ stmt-list “}” | ε
while-stmt → stmt-prefix “{“ stmt-list “}”
stmt-prefix → “while” “(“ expression “)”
{se expression.t != BOOL: **sinalizar erro de expressão lógica mal formada**}
read-stmt → “read” “id” {se TS.getTipo(ID.lexval) == null: **sinalizar erro para ID não declarado**}
write-stmt → “write” simple-expr
{se simple-expr.t != CHAR | | simple-expr.t != NUM: **sinalizar erro de incompatibilidade para impressão de valores**}

expression → simple-expr expression’
{se expression’.t == VOID: expression.t = simple-expr.t;
senao se expression’.t == simple-expr.t && simple-expr.t == BOOL: expression.t = BOOL;
senao: expression.t = **ERRO**}
expression’ → logop simple-expr expression’
{se expression’Filho.t == VOID && simple-expr.t == BOOL: expression’.t = BOOL;
senao se expression’Filho.t == simple-expr.t && simple-expr.t == BOOL: expression’.t = BOOL;
senao expression’.t = **ERRO**}
| ε {expression’.t = VOID}
simple-expr → term simple-expr’
{se simple-expr’.t == VOID: simple-expr.t = term.t;
senao se simple-expr’.t == term.t && simple-expr’.t == NUM: simple-expr.t = BOOL;
senao: simple-expr.t = **ERRO**}
simple-expr’ → relop term simple-expr’
{se simple-expr’Filho.t == VOID && term.t == NUM: simple-expr’.t = NUM;
senao se simple-expr’Filho.t == term.t && term.t == NUM: simple-expr’.t = NUM;
senao simple-expr’.t = **ERRO**}

| ε {simple-expr'.t = VOID}

term → factor-b term'
{se term'.t == VOID: term.t = factor-b.t;
senao se term'.t == factor-b.t && term'.t == NUM: term.t = NUM;
senao: term.t = **ERRO**}

term' → addop factor-b term'
{se term'Filho.t == VOID && factor-b.t == NUM: term'.t = NUM;
senao se term'Filho.t == factor-b.t && factor-b.t == NUM: term'.t = NUM;
senao term'.t = **ERRO**}
| ε {term'.t = VOID}

factor-b → factor-a factor-b'
{se factor-b'.t == VOID: factor-b.t = factor-a.t;
senao se factor-b'.t == factor-a.t && factor-b'.t == NUM: factor-b.t = NUM;
senao: factor-b.t = **ERRO**}

factor-b' → mulop factor-a factor-b'
{se factor-b'Filho.t == VOID && factor-a.t == NUM: factor-b'.t = NUM;
senao se factor-b'Filho.t == factor-a.t && factor-a.t == NUM: factor-b'.t = NUM;
senao factor-b'.t = **ERRO**}
| ε {factor-b'.t = VOID}

factor-a → factor {factor-a.t = factor.t}
| not factor {se factor.t != BOOL: factor-a.t = **ERRO;**
sinalizar erro de expressão lógica mal formada;
senão: factor-a.t = BOOL}

factor → "id" {factor.t = TS.getTipo(ID.lexval)}
| constant {factor.t = Constant.t}
| "(" expression ")" {factor.t = Expression.t}

logop → "or" | "and"
relop → "==" | ">" | ">=" | "<" | "<=" | "!="
addop → "+" | "-"
mulop → "*" | "/"
constant → "num_const" {Constant.t = NUM} | "char_const" {Constant.t = CHAR}