Análise Semântica

Brendon Vicente Rocha Silva
Graduando em Ciências da Computação
Universidade Federal de Santa Catarina - UFSC
Florianópolis, SC, Brasil
<brendon.vicente@grad.ufsc.br>

Para este trabalho foi necessário o desenvolvimento de um analisador semântico para a linguagem *LCC-2022-2*, gerada pela gramática *CC-2022-2*.

Para executar tal tarefa, foi preciso a implementação de um analisador sintático e um analisador léxico, para que fosse gerada a sequência de *tokens* utilizada para alimentar o analisador semântico.

O projeto foi desenvolvido em linguagem de programação *Python 3.10*

MODIFICAÇÕES EM *CC-2022-2*

Para esta parte do presente trabalho foram feitas algumas modificações na gramática proposta, a fim de proporcionar melhor usabilidade e mais comodidade ao se desenvolver códigos na linguagem estudada.

As seguintes modificações foram feitas:

```
STATEMENT        → (VARDECL; | ATRIBSTAT; | PRINTSTAT; | READSTAT;
                    | RETURNSTAT; | IFSTAT | FORSTAT | {STATELIST}
                    | break; | FUNCCALL; | ;)
PARAMLISTCALL    → (ident, PARAMLISTCALL | ident | FACTOR)?
IFSTAT           → if(EXPRESSION) STATEMENT (else STATEMENT)? endif
FUNCCALL         → invoke ident(PARAMLISTCALL)
```

ANÁLISE SEMÂNTICA

Optou-se pela realização da análise léxica, sintática e semântica de forma concorrente. O analisador léxico produz *tokens* que são avaliados pelo analisador sintático e posteriormente consumidos como nós da árvore de derivação, pelo analisador semântico. Quando um erro é encontrado, são lançadas *Exceptions* referentes ao tipo de erro detectado.

O programa é alimentado, de forma estática, com uma gramática provida de ações semânticas; cada nó da árvore de derivação é analisado de forma a determinar a ação a ser executada antes e depois de derivá-lo, assim como possivelmente o lexema e sua posição no arquivo.

EXPA E SDD L-ATRIBUÍDA

EXPRESSION → NUMEXPRESSION EXPRESSION'	EXPRESSION'.node = NUMEXPRESSION.root EXPRESSION.root = EXPRESSION'.sin
EXPRESSION' → RELOP NUMEXPRESSION	EXPRESSION'.sin = exp_node(RELOP.operator, "op", EXPRESSION'.node, NUMEXPRESSION.root)
EXPRESSION' → &	EXPRESSION'.sin = EXPRESSION'.node
NUMEXPRESSION → TERM NUMEXPRESSION'	NUMEXPRESSION'.node = TERM.root NUMEXPRESSION.root = NUMEXPRESSION'.sin
NUMEXPRESSION' → SUM TERM NUMEXPRESSION'	NUMEXPRESSION'¹.node = exp_node(SUM.operator, "op", NUMEXPRESSION'.node, TERM.root) NUMEXPRESSION'.sin = NUMEXPRESSION'¹.sin
NUMEXPRESSION' → &	NUMEXPRESSION'.sin = NUMEXPRESSION'.node
TERM → UNARYEXPR TERM'	TERM'.node = UNARYEXPR.node TERM.root = TERM'.sin
TERM' → MULTI UNARYEXPR TERM'	TERM'¹.node = exp_node(MULTI.operator, "op", TERM'.node, UNARYEXPR.node) TERM'.sin = TERM'¹.sin
TERM' → &	TERM'.sin = TERM'.node
RELOP → < OREQ	RELOP.operator = <.lexeme + OREQ.value
RELOP → > OREQ	RELOP.operator = >.lexeme + OREQ.value
RELOP → = =	RELOP.operator = =.lexeme + =.lexeme
RELOP → ! =	RELOP.operator = !.lexeme + =.lexeme
OREQ → =	OREQ.value = =.lexeme
OREQ → &	OREQ.value = ""
SUM → +	SUM.operator = +.lexeme
SUM → -	SUM.operator = -.lexeme
MULTI → *	MULTI.operator = *.lexeme
MULTI → /	MULTI.operator = /.lexeme
MULTI → %	MULTI.operator = %.lexeme
UNARYEXPR → SUM FACTOR	set_signal(FACTOR.node, SUM.operator) UNARYEXPR.node = FACTOR.node
UNARYEXPR → FACTOR	UNARYEXPR.node = FACTOR.node
FACTOR → int_constant	FACTOR.node = exp_node(int_constant.lexeme, "int")
FACTOR → float_constant	FACTOR.node = exp_node(float_constant.lexeme, "float")
FACTOR → string_constant	FACTOR.node = exp_node(string_constant.lexeme, "string")
FACTOR → null	FACTOR.node = exp_node("null", "null")

FACTOR → (NUMEXPRESSION)	FACTOR.node = NUMEXPRESSION.root
FACTOR → LVALUE	FACTOR.node = LVALUE.node
LVALUE → ident NUM_INDEX	NUM_INDEX.indexes = [] LVALUE.node = exp_node(ident.lexeme, get_ident_type(ident.lexeme), indexes=NUM_INDEX.indexes)
NUM_INDEX→ [NUMEXPRESSION] NUM_INDEX	append(NUM_INDEX.indexes, NUMEXPRESSION.root) NUM_INDEX¹.indexes = NUM_INDEX.indexes
NUM_INDEX → &	

* *set_signal* define o sinal de um operador salvo no nodo;
* *get_ident_type* obtém o tipo de um identificador.

SDT (*EXPA*)

- O início e fim do espaço de ação semântica é delimitado pelo caractere §;
- Quando são feitas referências à atributos de símbolos, eles são cercados pelo caractere #;
- É sempre apresentado junto a um símbolo a notação de ordem na produção, iniciado por 0.

```
EXPRESSION → NUMEXPRESSION § #EXPRESSION'_0.node# = #NUMEXPRESSION_0.root#
§ EXPRESSION' § #EXPRESSION_0.root# = #EXPRESSION'_0.sin# §

EXPRESSION' → RELOP NUMEXPRESSION § #EXPRESSION'_0.sin# = exp_node(
#RELOP_0.operator# , "op", #EXPRESSION'_0.node# , #NUMEXPRESSION_0.root# )
§

EXPRESSION' → & § #EXPRESSION'_0.sin# = #EXPRESSION'_0.node# §

NUMEXPRESSION → TERM § #NUMEXPRESSION'_0.node# = #TERM_0.root# §
NUMEXPRESSION' § #NUMEXPRESSION_0.root# = #NUMEXPRESSION'_0.sin# §

NUMEXPRESSION' → SUM TERM § #NUMEXPRESSION'_1.node# = exp_node(
#SUM_0.operator# , "op", #NUMEXPRESSION'_0.node# , #TERM_0.root# ) §
NUMEXPRESSION' § #NUMEXPRESSION'_0.sin# = #NUMEXPRESSION'_1.sin# §

NUMEXPRESSION' → & § #NUMEXPRESSION'_0.sin# = #NUMEXPRESSION'_0.node# §

TERM → UNARYEXPR § #TERM'_0.node# = #UNARYEXPR_0.node# § TERM' §
#TERM_0.root# = #TERM'_0.sin# §

TERM' → MULTI UNARYEXPR § #TERM'_1.node# = exp_node( #MULTI_0.operator# ,
"op", #TERM'_0.node# , #UNARYEXPR_0.node# ) § TERM' § #TERM'_0.sin# =
#TERM'_1.sin# §

TERM' → & § #TERM'_0.sin# = #TERM'_0.node# §

RELOP → < OREQ § #RELOP_0.operator# = #<_0.lexeme# + #OREQ_0.value# §

RELOP → > OREQ § #RELOP_0.operator# = #>_0.lexeme# + #OREQ_0.value# §

RELOP → = = § #RELOP_0.operator# = #=_0.lexeme# + #=_1.lexeme# §

RELOP → ! = § #RELOP_0.operator# = #!_0.lexeme# + #=_0.lexeme# §

OREQ → = § #OREQ_0.value# = #=_0.lexeme# §

OREQ → & § #OREQ_0.value# = "" §

SUM → + § #SUM_0.operator# = #+_0.lexeme# §

SUM → - § #SUM_0.operator# = #-_0.lexeme# §

MULTI → * § #MULTI_0.operator# = #*_0.lexeme# §

MULTI → / § #MULTI_0.operator# = #/_0.lexeme# §

MULTI → % § #MULTI_0.operator# = #%_0.lexeme# §

UNARYEXPR → SUM FACTOR § set_signal( #FACTOR_0.node# , #SUM.operator# ) ;
#UNARYEXPR_0.node# = #FACTOR_0.node# §

UNARYEXPR → FACTOR § #UNARYEXPR_0.node# = #FACTOR_0.node# §

FACTOR → int_constant § #FACTOR_0.node# = exp_node(
#int_constant_0.lexeme# , "int" ) §
```

```
FACTOR → float_constant § #FACTOR_0.node# = exp_node(
#float_constant_0.lexeme# , "float") §

FACTOR → string_constant § #FACTOR_0.node# = exp_node(
#string_constant_0.lexeme# , "string" ) §

FACTOR → null § #FACTOR_0.node# = exp_node( "null" , "null" ) §

FACTOR → ( NUMEXPRESSION ) § #FACTOR_0.node# = #NUMEXPRESSION_0.root# §

FACTOR → LVALUE § #FACTOR_0.node# = #LVALUE_0.node# §

LVALUE → ident § #NUM_INDEX_0.indexes# = [] § NUM_INDEX § #LVALUE_0.node#
=  exp_node( #ident_0.lexeme# , get_ident_type( #ident_0.lexeme# ), indexes
= #NUM_INDEX_0.indexes# ) §

NUM_INDEX → NUM_INDEX → [ NUMEXPRESSION § assert_expression(
#NUMEXPRESSION_0.root# , #[_0# ) § ] § append( #NUM_INDEX_0.indexes# ,
#NUMEXPRESSION_0.root# ) ; #NUM_INDEX_1.indexes# = #NUM_INDEX_0.indexes# §
NUM_INDEX

NUM_INDEX → &
```

As árvores de expressões são impressas durante a execução do programa.

DEC E SDD L-ATRIBUÍDA

FUNCDEF → def ident (PARAMLIST) { STATELIST }	PARAMLIST.count = 0 add_func(ident, PARAMLIST.count)
PARAMLIST → TYPE ident PARAMLIST'	add_var(TYPE.type, ident) PARAMLIST'.count = PARAMLIST.count + 1 PARAMLIST.count = PARAMLIST'.count
PARAMLIST → &	
PARAMLIST' → , PARAMLIST	PARAMLIST.count = PARAMLIST'.count PARAMLIST'.count = PARAMLIST.count
PARAMLIST' → &	
TYPE → int	TYPE.type = "int"
TYPE → float	TYPE.type = "float"
TYPE → string	TYPE.type = "string"
VARDECL → int ident INT_INDEX	add_var("int", ident)
VARDECL → float ident INT_INDEX	add_var("float", ident)
VARDECL → string ident INT_INDEX	add_var("string", ident)

* *add_func* adiciona uma função à tabela de símbolos dado o número de parâmetros;
* *add_var* adiciona um identificador à tabela de símbolos dado seu tipo.

SDT (*DEC*)

- O início e fim do espaço de ação semântica é delimitado pelo caractere §;
- Quando são feitas referências à atributos de símbolos, eles são cercados pelo caractere #;
- É sempre apresentado junto a um símbolo a notação de ordem na produção, iniciado por 0.

```
FUNCDEF → def ident § scope( #ident_0# ) § ( § #PARAMLIST_0.count# = 0 § PARAMLIST ) § add_func( #ident_0# , #PARAMLIST_0.count# ) § { STATELIST } § unscope() §

PARAMLIST → &

PARAMLIST → TYPE ident § add_var( #TYPE_0.type# , #ident_0# ) ; #PARAMLIST'_0.count# = #PARAMLIST_0.count# + 1 § PARAMLIST' § #PARAMLIST_0.count# = #PARAMLIST'_0.count# §

PARAMLIST' → &

PARAMLIST' → , § #PARAMLIST_0.count# = #PARAMLIST'_0.count# § PARAMLIST § #PARAMLIST'_0.count# = #PARAMLIST_0.count# §

TYPE → int § #TYPE_0.type# = "int" §

TYPE → float § #TYPE_0.type# = "float" §

TYPE → string § #TYPE_0.type# = "string" §

VARDECL → int ident § add_var("int" , #ident_0# ) § INT_INDEX

VARDECL → float ident § add_var("float" , #ident_0# ) § INT_INDEX

VARDECL → string ident § add_var("string" , #ident_0# ) § INT_INDEX
```

VERIFICAÇÃO DE TIPOS

Para garantir que expressões aritméticas admitam apenas variáveis do mesmo tipo basta avaliar os nodos da árvore de expressão, um a um. A função *assert_expression* é responsável por tal processo; ela é invocada via ação semântica - por símbolos que derivam expressões - e, caso encontre incompatibilidade de tipos, lança uma exceção.

VERIFICAÇÃO DE IDENTIFICADORES POR ESCOPO

Para fazer o controle de declaração de variáveis, foi implantado um sistema hierárquico de tabelas de símbolos: cada tabela possui um escopo e símbolos são salvos apenas na tabela sendo avaliada no momento. As funções *scope* e *unscope* são responsáveis por descer ou subir de nível na hierarquia e as funções *add_func* e *add_var* são responsáveis por adicionar variáveis ou funções à tabela atual. Todas as funções são invocadas via ação semântica e, caso haja divergências, uma exceção é lançada.

COMANDOS DENTRO DE ESCOPO

Para lidar com a verificação do comando *break* no escopo de um comando de repetição foi necessário adicionar um novo atributo à tabela de símbolos: *is_loop*. Quando um *token* que deriva um laço de repetição é avaliado, sua tabela de símbolos é criada com tal indicador e, quando um lexema *break* é encontrado, a função *assertLoop* verifica se a tabela atual pertence a um laço de repetição; caso não seja verdadeiro, uma exceção é lançada.