Trabalho 4

-- Compiladores --

Analisador Semântico

Integrantes:

Caio Gomes	7239072
Diego Gomes	7656467
Fernando Cury Gorodscy	7152354
Roberty Manzini Bertolo	7573399
Vanessa Apolinário Lima	7239256

Sumário

Introdução

Esse documento tem como objetivo apresentar a gramática implementada no Analisador Semântico JavaCC, seu analisador, os respectivos casos de testes analisados e um pouco da estrutura desenvolvida sobre o trabalho.

O nome da nossa gramática chama-se **Meuphoria.** Esse nome surgiu de uma brincadeira em sala de aula, na qual nós tinhamos que criar um nome, e queríamos que fosse nossa. Daí alguém sugeriu o "Meuphoria" - "Nossa euforia".

Caberá a cada uma das seções explicar um pouco sobre o desenvolvimento e ideia da implementação dada.

Gramática

A gramática designada encontra-se abaixo com a nossa adição/característica única.

TERMINAIS:
ALPHA ::= ["a"-"z", "A"-"Z"]
DIGIT ::= ["0"-"9"]
USCORE ::= "_"
EOL ::= "\n"

TOKENS:
- Palavras reservadas:
CASE ::= "case"
DO ::= "do"
END ::= "end"
EXPORT ::= "export"
GLOBAL ::= "global"
INCLUDE ::= "include"
PUBLIC ::= "public"
SWITCH ::= "switch"
UNTIL ::= "until"
CONSTANT ::= "constant"
ELSE ::= "else"
ENTRY ::= "entry"
FALLTHRU ::= "fallthru"
GOTO ::= "goto"
LABEL ::= "label"
RETRY ::= "retry"
THEN ::= "then"
WHILE ::= "while"
BREAK ::= "break"
CONTINUE ::= "continue"
ELSEDEF ::= "elsedef"
ENUM ::= "enum"
IF ::= "if"
LOOP ::= "loop"
OVERRIDE ::= "override"
RETURN ::= "return"
TO ::= "to"
WITH ::= "with"
BY ::= "by"

ELSIF ::= "elsif"
EXIT ::= "exit"
FUNCTION ::= "function"
IFDEF ::= "ifdef"
PROCEDURE ::= "procedure"
TYPE ::= "type"
WITHOUT ::= "without"
ELSEIFDEF ::= "elsifdef"
FOR ::= "for"

- Separadores:
LPAREN ::= "("
RPAREN ::= ")"
LBRACE ::= "{"
RBRACE ::= "}"
LBRACKET ::= "["
RBRACKET ::= "]"
SEMICOLON ::= ";"
COMMA ::= ","
DOT ::= "."
#SINGLE_QUOTE ::= "\'"
SLICE ::= ".."
#QUOTE ::= "\""
#TRIPLE_QUOTE ::= "\"\"\""
COLON ::= ":"

- Operadores:
NOT_OP ::= "not"
PLUS ::= "+"
MINUS ::= "-"
STAR ::= "*"
SLASH ::= "/"
CONCAT ::= "&"
GT ::= ">"
LT ::= "<"
LE ::= "<="
GE ::= ">="
EQUAL ::= "="
NE ::= "!="
SC_OR ::= "or"
SC_AND ::= "and"
XOR_OP ::= "xor"
END_SYMBOL ::= "$"

MANIPULAÇÃO VARIÁVEIS:
IDENTIFIER ::= (ALPHA | USCORE) { ALPHA | DIGIT | USCORE }

```
ATOM ::= INTEGER | REAL
        INTEGER ::= [ PLUS | MINUS ] DIGIT { DIGIT | USCORE }
        REAL ::= FLOAT | EXP
                FLOAT ::= INTEGER DOT DIGIT { DIGIT | USCORE }
                EXP ::=  ( INTEGER | FLOAT ) ("E" | "e") INTEGER

SEQUENCE ::= LBRACE OBJECT { COMMA OBJECT } [ COMMA END_SYMBOL ] RBRACE
        OBJECT ::= ATOM |  SEQUENCE | BIN_NUMBER | OCT_NUMBER | DEC_NUMBER |
HEX_NUMBER
                BIN_NUMBER ::= "0b" BIN
                        BIN ::= ("0" | "1") { "0" | "1" }
                OCT_NUMBER ::= "0t" OCT
                        OCT ::= ["0" - "7"] { ["0" - "7"] }
                DEC_NUMBER ::= "0d" INTEGER
                HEX_NUMBER ::=  "0x" HEX
                        HEX ::= ["0" - "9", "A" - "F"] { ["0" - "9", "A" - "F"] }


PARAMLIST ::= PARAMETER {COMMA PARAMETER }
        PARAMETER ::= DATATYPE IDENTIFIER

IDENTLIST ::= IDENT { COMMA IDENT }
        IDENT ::= IDENTIFIER [ EQUAL EXPRESSION ]

VARIABLE ::= IDENTIFIER [ SLICESTMT | SUBSCRIPTING ]
        SLICESTMT ::= { LBRACKET EXPRESSION SLICE ( EXPRESSION | END_SYMBOL) RBRACKET }
        SUBSCRIPTING ::= INDEX { INDEX }
                INDEX ::= LBRACKET EXPRESSION RBRACKET
```

MANIPULAÇÃO DE STRINGS:

```
CHAR  ::= ["a"-"z", "A"-"Z", caracteres_especiais]

STRINGLIT ::= SIMPLESTRINGLIT | RAWSTRINGLIT
        SIMPLESTRINGLIT ::= QUOTE { CHAR | ESCCHAR } QUOTE
                ESCCHAR ::= ESCLEAD ( "n" | "r" | "t" | "\\" | "\"" | "\'" | "0" | "e" | "E" | BINARY |
        HEX2 | HEX4 | HEX8 )
                        BINARY ::= "\\b" { ["0", "1"] }
                        HEX2 ::= "\\x" ["0" - "7"] ["0" - "7"]
                        HEX4 ::= "\\u" ["0" - "7"] ["0" - "7"]["0" - "7"] ["0" - "7"]
                        HEX8 ::= "\\U" ["0" - "7"] ["0" - "7"]["0" - "7"] ["0" - "7"]["0" - "7"] ["0" -
"7"]["0" - "7"] ["0" - "7"]
        RAWSTRINGLIT ::= DQRAWSTRING | BQRAWSTRING
                DQRAWSTRING ::= "`" CHAR {CHAR} "`"
                BQRAWSTRING ::= "\"\"\"" CHAR {CHAR} "\"\"\""
```

LABELSTMT ::= LABEL STRINGLIT

DATA TYPE:
SCOPEMODIFIER ::= GLOBAL | PUBLIC | EXPORT | OVERRIDE
DATATYPE ::= "atom" | "integer" | "sequence" | "object" | IDENTIFIER

EXPRESSÕES:
EXPRESSION ::= SEQUENCE | E0
E0 ::= E1 { (SC_AND | SC_OR | SC_XOR) E1}
E1 ::= E2 { RELATIONOP E2 }
RELATIONOP ::= "<" | ">" | "<=" | ">=" | "=" | "!="
E2 ::= E3 { CONCAT E3 }
E3 ::= E4 { (PLUS | MINUS) E4 }
E4 ::= E5 { (STAR | SLASH) E5 }
E5 ::= [PLUS | MINUS | NOT_OP] E6
E6 ::= ATOM | STRINGLIT | SEQUENCE |VARIABLE | "0" | "1" | (LPAREN EXPRESSION RPAREN) | IDENTIFIER | CALL
CALL ::= IDENTIFIER "(" [ARGLIST] ")"
ARGLIST ::= EXPRESSION { "," EXPRESSION }

STATEMENT:
STMBLK ::= STATEMENT { STATEMENT }

STATEMENT ::= FLOW | LOOP | BRANCH | ASSIGNMENTSTMT | RETURN | VARDECLARE | CONSTDECLARE | ENUMDECLARE | PROCDECLARE | FUNCDECLARE | TYPEDECLARE | CALL

- Statements Básicos:
// used in the loop
WITHENTRY ::= WITH ENTRY
ENTRYSTMT ::= ENTRY [STMBLK]

- Controle de Fluxo:
FLOW ::= ((BREAK [INTEGER]) | CONTINUE | RETRY | EXIT | FALLTHRU) [STRINGLIT]

- Loop:
LOOP ::= FORSTMT | WHILESTMT | LOOPSTMT | GOTOSTMT
FORSTMT ::= FOR FORIDX [LABEL] DO [STMBLK] END FOR
FORIDX ::= IDENTIFIER EQUAL EXPRESSION TO EXPRESSION [BY EXPRESSION]
WHILESTMT ::= WHILE EXPRESSION [WITHENTRY][LABEL] DO STMBLK [ENTRYSTMT] END WHILE
LOOPSTMT ::= LOOP [WITHENTRY][LABEL] DO STMBLK [ENTRYSTMT] UNTIL EXPRESSION END LOOP
GOTOSTMT ::= GOTO LABEL

- Branching:
BRANCH ::= IFSTMT | SWITCHSTMT | IFDEFSTMT

IFSTMT ::= IFTEST { ELSIFSTMT } [ELSESTMT] ENDIF
IFTEST ::= IF EXPRESSION [LABEL] THEN [STMBLK]
ELSIFSTMT ::= ELSIF EXPRESSION THEN [STMBLK]
ELSESTMT ::= ELSE [STMBLK]
ENDIF ::= END IF
SWITCHSTMT ::= SWITCHTEST CASESTMT { CASESTMT } [CASEELSE] ENDSWITCH
SWITCHSTMT ::= SWITCHTEST CASESTMT { CASESTMT } [ELSE {STMBLK}] ENDSWITCH
SWITCHTEST ::= SWITCH EXPRESSION [WITHFALL] [LABEL] DO
WITHFALL ::= (WITH | WITHOUT) FALLTHRU
CASESTMT ::= CASE CASELIST THEN [STMBLK]
CASELIST ::= EXPRESSION {"," EXPRESSION }
ENDSWITCH ::= END SWITCH
IFDEFSTMT ::= IFDEFTEST [ELSDEFIFSTMT {CHAR}] [ELSEDEFSTMT] ENDDEFIF
IFDEFTEST ::= IFDEF DEFEXPR THEN [STMBLK]
ELSDEFIFSTMT ::= ELSEIFDEF DEFEXPR THEN [STMBLK]
ELSEDEFSTMT ::= ELSEDEF [STMBLK]
ENDDEFIF ::= END IFDEF
DEFEXPR ::= DEFTERM [DEFOP DEFTERM]
DEFTERM ::= [NOT] IDENTIFIER
DEFOP ::= SC_AND | SC_OR

- Atribuição:
ASSIGNMENTSTMT ::= ASSIGNMULTI | ASSIGNWITHOP
ASSIGNMULTI ::= IDENTIFIER {COMMA IDENTIFIER} EQUAL EXPRESSION {COMMA EXPRESSION}
ASSIGNWITHOP ::= IDENTIFIER (PLUS | MINUS | SLASH | STAR | CONCAT) EQUAL EXPRESSION

- Return:
RETURNSTMT ::= RETURN EXPRESSION { COMMA EXPRESSION } ***--Característica Meuphoria***

- Declaração de Variáveis:
VARDECLARE ::= [SCOPEMODIFIER] DATATYPE IDENTLIST

- Declaração de Constantes:
CONSTDECLARE ::= [SCOPEMODIFIER] CONSTANT IDENTLIST

- Declaração de Enumeração:
ENUMDECLARE ::= [SCOPEMODIFIER] (ENUMVAL | ENUMTYPE)
ENUMVAL ::= ENUM IDENTLIST
ENUMTYPE ::= ENUM TYPE IDENTLIST END TYPE

- Declaração de Processos:
PROCDECLARE ::= [SCOPEMODIFIER] PROCEDURE IDENTIFIER LPAREN [PARAMLIST] RPAREN [STMBLK] END PROCEDURE

- Declaração de Função:
FUNCDECLARE ::= [SCOPEMODIFIER] FUNCTION IDENTIFIER LPAREN [PARAMLIST] RPAREN [STMBLK] END FUNCTION

- Declaração de Tipo:
TYPEDECLARE ::= [SCOPEMODIFIER] TYPE IDENTIFIER LPAREN [PARAMLIST] RPAREN [STMBLK] END TYPE

Casos de Testes

Realizamos uma bateria de testes léxicos, sintáticos e semânticos, entretanto, para este relatório iremos apenas designar como pedido os casos de testes referentes à bateria semântica, os arquivos podem ser encontrados na pasta *test_cases(happy path, sad paths e resultados).*

Semântico: Happy Path Tests (Sucesso!)

Teste: Switch Case (modificado o CASE ELSE -> CASE, conforme pedido) [Teste 10]

Entrada:

```
atom nota = 'C'
switch nota do
  case 'A' then
    puts(1, "Excelente!\n" )
  case 'B', 'C' then
    puts(1, "Muito bom!\n" )
  case 'D' then
    puts(1, "Aprovado!\n" )
  case 'F' then
    puts(1, "Oh, nao! Fui reprovado!\n" )
  else
    puts(1, "Nota invalida!\n" )
end switch
```

Saída:

Análises léxica, sintática e semântica concluídas com sucesso!

Teste: Procedure [Teste 16]

Entrada:

```
procedure DigaOi(sequence nome,atom  idade)
  printf(1, "%s tem %d anos.", {nome, idade})
end procedure


-- chama a procedure definida em cima.
DigaOi("zara", 8)
```

Saída:

Análises léxica, sintática e semântica concluídas com sucesso!

Teste: Function [Teste 17]

Entrada:

```
function DigaOi()
  puts(1, "Oi Professora")
  return 1
end function

-- Chama a funcao acima
DigaOi()
```

Saída:

Análises léxica, sintática e semântica concluídas com sucesso!

Teste: Our function [Teste 18]

Entrada:

```
sequence nome
integer idade

nome, idade = "Vanessa", 10
```

Saída:

Análises léxica, sintática e semântica concluídas com sucesso!

Teste: Our function [Teste 18]

Entrada:

```
sequence nome
integer idade

nome, idade = "Vanessa", 10
```

Saída:

Análises léxica, sintática e semântica concluídas com sucesso!

Teste: Precedência e Prioridade [Teste 20]

Entrada:

```
--Precedencia e Prioridade

puts(1,"Deve imprimir 14, 14, 24, 7 e 3. Comeca aqui ->")


integer a = 5
integer b = 3
integer c = 3
integer resultado

resultado = a + (b * c)
puts(1,"a + (b * c) = ")
?resultado

resultado = a + b * c
puts(1,"a + b * c = ")
?resultado

resultado = (a + b) * c
puts(1,"(a + b) * c = ")
?resultado

resultado = 6 + b / c
puts(1,"6 + b / c = ")
?resultado

resultado = (6 + b) / c
puts(1,"(6 + b) / c = ")
?resultado
```

Saída:

Análises léxica, sintática e semântica concluídas com sucesso!

Teste: Retorno variável e variaveis com mesmo nomes em funções distintas [Teste 21]

Entrada:

```
function sayHello()
  sequence nome
  nome = "Oi Vanessa"
  return nome
end function

function sayBye()
  sequence nome
  nome = "Tchau Vanessa"
  return nome
end function

puts(1,sayHello())
puts(1,sayBye())
```

Saída:

Análises léxica, sintática e semântica concluídas com sucesso!

Teste: Vários parâmetros - Função [Teste 22]

Entrada:

```
function qualquerUma(integer b, integer c)
  atom number
  number = 1 + b + c
  return number
end function

integer b = 5
integer c = 6
integer a = qualquerUma(b, c)
```

Saída:

Análises léxica, sintática e semântica concluídas com sucesso!

Teste: Vários parâmetros receber – Função Retorno [Teste 23]

Entrada:

```
function qualquerUma(integer w)
  atom number
  number = 1 + w
  return number, w
end function

integer c = 6
integer d = 6
integer a,b = qualquerUma(c)
```

Saída:

Análises léxica, sintática e semântica concluídas com sucesso!

Sad Path Tests (Erro :/)

Teste: Variável não declarada [Teste 19]

Entrada:

```
integer b = a
integer c = d
--Variavel nao Declarada
```

Saída:

[1, 13] Erro Semantico: Variável a não declarada.

[2, 13] Erro Semantico: Variável d não declarada.

Análise encontrou 2 erro(s).

Teste: Variável redeclarada [Teste 20]

Entrada:

```
integer a = 4
integer b = 2
integer a = 5
--Redeclaracao de Variavel nao pode!
```

Saída:

[3, 9] Erro Semantico: Variavel 'a' ja declarada.

Análise encontrou 1 erro(s).

Teste: Variável redeclarada tipos diferentes [Teste 21]

Entrada:

```
integer a = 4
sequence a = {1, 2}
--Redeclaracao de Variavel nao pode!
```

Saída:

[2, 10] Erro Semantico: Variavel 'a' ja declarada.

Análise encontrou 1 erro(s).

Teste: Operador Lado Esquerdo [Teste 22]

Entrada:

```
--Realizar operacoes do lado esquerdo do '='
integer a = 8
integer b = 1
integer teste
teste + a = b
```

Saída:

```
Encountered " <IDENTIFIER> "a "" at line 5, column 9.

Was expecting:

   "=" ...


Ocorreu uma excecao!
```

Teste: Função sem retorno [Teste 23]

Entrada:

```
--Funcao nao possui retorno
integer a = 8
function teste(integer arg)
?arg
end function

integer resultado = teste(a)
```

Saída:

```
[3, 1] Erro Semantico: function nao retorna nenhum valor.

Análise encontrou 1 erro(s).
```

Teste: Função com número errado parametros [Teste 24]

Entrada:

```
--Numero errado de argumentos na funcao:
integer a = 5
integer b = 2
function teste(integer arg)
?arg
return arg
end function


integer resultado = teste(a, b)
```

Saída:

```
[8, 21] Erro semantico: Procedimento 'teste' com numero de parametros invalidos.

Análise encontrou 1 erro(s).
```

Teste: Procedure não existente [Teste 25]

Entrada:

```
integer tempo
tempo = 10

-- tentar chamar a variavel como se fosse uma procedure/função

tempo()
```

Saída:

```
[6, 1] Erro semantico: Procedimento 'tempo' não declarado.

Análise encontrou 1 erro(s).
```

Teste: Redeclaração de argumentos dentro de funções [Teste 26]

Entrada:

```
--Nao deve redeclarar argumentos
integer a

procedure teste(integer arg)
integer arg
end procedure


teste(a)
```

Saída:

[4, 9] Erro Semantico: Variavel 'arg' ja declarada.

Análise encontrou 1 erro(s).

Análise do projeto

Em suma, atingimos os resultados esperados como pode-se ver pelos casos de testes que conseguem efetivar todos os erros e acertos semânticos que pudemos imaginar, entretanto, o trabalho demandou muito mais trabalho do que o previsto ou suposto.

Algumas decisões que tomamos:

- **Estrutura da Tabela de Símbolos:** formato ad hoc utilizando um tabela hash, para facilitar a leitura e adição dos elementos. Essa escolha foi feita, pois, facilita o armazenamento das informações na memória permitindo uma melhor busca para arquivos pequenos e grandes, o que dificultaria caso fosse utilizado uma lista ordenada.
    - São 3 os principais arquivos: Symbol.java, SymbolTable.java e SemanticRoutines.java
        - Symbol.java: é a informação do símbolo em si(variável, constante, parâmetro, procedimento ou função)
        - SymbolTable.java: é a TAD da tabela de símbolos, é possível procurar ou inserir arquivos nessa tabela.
        - SemanticRoutines.java: são as rotinas semânticas, adicionamos todos os builtins (http://openeuphoria.org/docs/builtins.html#_783_builtinmethods) possíveis nessa tabela. Além disso, criamos funções para verificar o número de parâmetros, o nível em que o escopo se encontra(0 ou 1) e será aqui que faremos as checagens de tipos ligeiramente testadas.
- **Estrutura do código:**
    - O analisador em si ele é feito baseado nos exemplos em sala de aula, explica-lo detalhadamente demandaria muito tempo e não vemos necessário. As essências estão explicadas nos outros tópicos.
- **Definimos Categoria e Tipos:**
    - **Categorias:** variáveis, parâmetros, procedimentos, funções e constantes
    - **Tipos:** átomo, int, sequência, objeto, constante, boolean e enum

- **Built-Ins definidos:**
    - http://openeuphoria.org/docs/builtins.html#_783_builtinmethods
- **Declaração de variáveis:** descobrimento de declarações de variáveis - passando duas vezes pelo código, primeiro encontra declarações, depois faz a análise semântica, como é feito em C++, que é atualizando e verificando se a declaração está completa. Foi uma decisão de projeto que achamos mais fácil e como o projeto é para fins didáticos acreditamos válido essa representação.

Sugestões, dúvidas, críticas e conclusão

A separação da checagem de tipos dificultou muito o entendimento do que “era para ser feito” e o que “podia ser feito depois”.

Alguns dos casos que tivemos dúvidas:

- Overflow e Underflow: Apenas conseguimos verifica-los com efetividade após a definição dos tipos.
- Junção de sequências:
- Limitadores: Superior e inferior.

Entendemos que para esses deverão ser feitos no próximo, entretanto, sem um definição sólida do tipo, seria feito algum muito geral o que poderia prejudicar a semântica do trabalho.

Como sugestão talvez, seria extender o prazo e realizar os dois trabalhos em um só.

Nos aspectos gerais, demandamos muito trabalho para entender como implementar a parte semântica mesmo com o embasamento teórico. Entretanto, foi possível feito após um grande tempo de trabalho em cima dele.