Projeto de Implementação de um Compilador para a Linguagem T++
Análise Semântica (Trabalho – 3ª parte)

Gabriel Negrão Silva[1]
[1] Universidade Tecnológica Federal do Paraná - Campo Mourão (UTFPR-CM)
Campo Mourão – PR – Brasil, 16 de Novembro de 2017

itsg_negrao@hotmail.com

Resumo. *Este artigo descreve como ocorre o processo de análise semântica de um compilador construindo um analisador para a linguagem* ***T++****, tal como procedimentos, alterações, instruções, exemplos e análise dos resultados.*

1. Introdução…… 2
2. Descrição e Procedimentos…… 2
2.1. Análise Semântica…… 2
2.2. Tabela de Símbolos…… 4
3. Resultados e Análise…… 5
4. Discussão…… 6
5. Conclusão…… 6

1. Introdução

O trabalho proposto foi aplicado de forma gradual, sendo constituído de 4 partes principais, neste artigo discorreremos sobre a terceira parte deste trabalho proposto. Este relatório tem como objetivo mostrar como foi projetado e construído o algoritmo que discorreremos sobre posteriormente junto com uma conclusão final. O código está identado e comentado, instruções auxiliares são dadas no decorrer do artigo anexado.

2. Descrição e Procedimentos

2.1. Análise Semântica

A análise semântica verifica e valida o código e as expressões de acordo com a especificação da linguagem T++ que baseia-se na linguagem C.

Esta parte do compilador verifica tipos, atribuições, escopos da funções, variáveis não utilizadas, tipos de retorno, chamadas de funções e todas as outras partes responsáveis por validar o código, evitando erros de atribuições de tipos diferentes por exemplo, diferente da análise sintática onde se verifica apenas o erros léxicos como por exemplo um “fim” a mais no código, a parte de análise semântica fica a cargo de validar o código, tal como exemplo o tipo de retorno de uma função.

A análise semântica utiliza da árvore abstrata gerada pela análise sintática *(ast)* como entrada para o programa (Figura 2), uma vez que a mesma é utilizada na expansão da árvore utilizando as chamadas de funções para tal, cada função tem a obrigação de expandir seus devidos nós filhos e se caber a ela validar o conteúdo recebido ou retornado de outra chamada de função (Figura 3, Figura 4).

O trecho de código a seguir define a função __init__ a qual é a função de inicialização do analisador, onde se tem a tabela de símbolos, uma flag de debug que quando ativa emite todos os erros do código, uma variável escopo, a qual é alterada por outras funções para manter o escopo atual e a chamada da função programa passando a o retorno do analisador sintático e 3 funções adicionar para verificações posteriores.

```
8  # Importação ply lexer
9  from graphviz import Graph
10 import ply.yacc as yacc
11 from Analisador_Sintatico import Analisador_Sintatico
12 import sys
13 import os
14
15 class Analisador_Semantico:
16
17     def __init__(self, code, flag):
18         self.tabSimbolos = {}
19         self.escopo="global"
20         self.debug = flag
21
22         self.arvoreSintatica = Analisador_Sintatico(code).ast
23         self.programa(self.arvoreSintatica)
24
25         self.verificaDeclarPrincipal(self.tabSimbolos)
26         self.verificaFuncaoUtilizada(self.tabSimbolos)
27         self.verificaVarUtilizada(self.tabSimbolos)
28
```

Figura 2. Trecho da Inicialização 'Analisador_Semantico''.

As Figuras 3 e 4 referem-se a implementação da Análise Semântica expandindo a árvore recebida através das chamadas das funções criadas.

```
281     def fator(self, node):
282         if(node.child[0].type=="var"):
283             return self.var(node.child[0])
284         if(node.child[0].type=="chamada_funcao"):
285             return self.chamadaFuncao(node.child[0])
286         if(node.child[0].type=="numero"):
287             return self.numero(node.child[0])
288         else:
289             return self.expressao(node.child[0])
```

Figura 3. Função 'fator'.

```
231     def expSimples(self, node):
232         if len(node.child)==1:
233             return self.expAditiva(node.child[0])
234         else:
235             tipoExp1 = self.expSimples(node.child[0])
236             self.opRelacional(node.child[1])
237             tipoExp2 = self.expAditiva(node.child[2])
238 
239             if(tipoExp1 != tipoExp2):
240                 print("Warning: Operação com tipos diferentes -> '"+
241 
242             return "logico"
```

Figura 4. Função 'expSimples'.

2.2. Tabela de Símbolos

A tabela de símbolos é necessária para que se tenha um controle das variáveis declaradas e suas manipulações para a geração dos erros e warnings esperados na análise semântica.

A tabela é implementada através da adição de tuplas a mesma e acessada através de um hash ou implicitamente acessando a posição desejada (Figura 5).

```
52      def declaracaoVar(self, node):
53          tipo = node.child[0].type
54          string=""
55          i=0
56          complemento=""
57 
58          for son in self.listaVars(node.child[1]):
59              if ("[" in son):
60                  for i in range(len(son)):
61                      if (son[i]=="["):
62                          break
63                      string += son[i]
64                  complemento = son[i:]
65                  son = string
66 
67              if (son  in self.tabSimbolos.keys()):
68                  print("Erro: Já existe uma função declarada como '"+node.value
69                  exit(1)
70 
71              if("global-"+son  in self.tabSimbolos.keys()):
72                  print("Erro: Variável '"+son+ "' já declarada.")
73                  exit(1)
74 
75              if (self.escopo+"-"+son  in self.tabSimbolos.keys()):
76                  print("Erro: Variável '"+son+ "' já declarada.")
77                  exit(1)
78 
79              self.tabSimbolos[self.escopo+"-"+son]=["variavel",son,False,False,
80 
81          return "void"
```

Figura 5. Função 'declaracaoVar'.

3. Resultados e Análise

Dado o algoritmo de teste escrito na linguagem T++ o qual recebe um número inteiro e escreve seu fatorial na tela como entrada para o analisador léxico programando em Python para a linguagem. (Figura 1).

O algoritmo a seguir (Figura 1) escrito na linguagem T++ anexado ao projeto, o qual escreve na tela o fatorial de um número inteiro n, é a entrada de teste para o analisador sintático programado em Python executado via terminal com **Python 3**.

```
1   {Atribuição de tipos distintos}
2
3 - inteiro func(inteiro: x, inteiro: y)
4     retorna (x + y)
5   fim
6
7 - inteiro principal()
8     flutuante: a
9     flutuante: c
10    inteiro: b
11
12    b := c
13
14    a := func(10,5)
15  fim
```

Figura 1. Algoritmo de teste sema-006 em T++.

A saída do programa criado na linguagem Python para a análise sintática da linguagem T++ usando o programa de teste (Figura 1) escrito na linguagem T++ incorporado ao projeto é dada por mensagens de erros e warnings via terminal, a saída final executando o programa '*Analisador_Semantico.py*' via terminal de comando utilizando **Python 3** e passando o programa **sema-006.tpp** como parâmetro foi a seguinte.

Exemplo de execução via terminal:

---

root@root-PC:~/$ python3 Analisador_Semantico.py semantica-testes/sema-006.tpp --debug=OFF
Erro: Variável 'principal-c' chamada em 'principal' não foi atribuída.

---

4. Discussão

A saída dos testes executados foram satisfatórias, tal como este demonstrado anteriormente. Há mais algoritmos programados em T++ para testes do analisador e o resultado obtido com a execução dos mesmos foram satisfatoriamente boas (cumpriram os requisitos mínimos).

Os experimentos foram realizados em computador pessoal móvel (Notebook) próprio com processador intel quad core e seu tempo de execução foi baixo e olho nú aparentemente de término instantâneo dado tamanho do teste de entrada.

Nesta parte não utilizamos mais a ferramenta PLY para a análise semântica uma vez que este é implementado pelo programador.

O maior problema nesta parte do compilador para a linguagem é que todas as implementações e validações precisam ser feitas pelo programador seguindo as especificações da linguagem.

5. Conclusão

Nesta parte do projeto do compilador para a linguagem T++ não tivemos auxílio de ferramenta portando o entendimento do conteúdo e da função do analisador semântico ficam a par do programador. Após ocorrido o entendimento do mesmo, a programação se torna mais fluida.

Este projeto teve como contribuição o entendimento da parte Semântica de um compilador referindo-se a disciplina de Compiladores, das funcionalidades e construção de um compilador e das ferramentas de suporte ao projeto, mais especificamente a parte de um analisador Semântico, parte integrante de um compilador.