Análise Semântica

Juan Felipe da S. Rangel[1]

[1]Universidade Tecnológica Federal do Paraná (UTFPR)
Caixa Postal 135 – 87.301-899 – Campo Mourão – PR – Brasil

juanrangel@alunos.utfpr.edu.br

Abstract. *With the objective of developing a TPP code compiler at the end of the Compilers subject, this article will see in greater detail how the third step of this process works, the Semantic analysis. For the implementation, the Python language was used, together with the Pandas libraries, and the libraries already seen in the development of the previous analyses, receiving the generated Syntax Tree from the Syntactic Analysis.*

Resumo. *Com o objetivo de desenvolver um compilador de códigos TPP ao fim da matéria de Compiladores, neste artigo será visto em maior detalhe como funciona o terceiro passo deste processo, a análise Semântica. Para a implementação foi utilizado a linguagem Python, em conjunto com as bibliotecas Pandas,e as bibliotecas já vista no desenvolvimento das análises anteriores, recebendo da análise Sintática a Árvore Sintática gerada.*

1. Introdução

Na fase denominada como Análise Semântica temos como objetivo verificar as regras semânticas da linguagem T++ e se elas estão sendo seguidas nos códigos .tpp que serão analisados, retornando por fim uma tabela de Símbolos, a árvore Árvore Sintática Abstrata podada e os erros e warnings apresentados no código de entrada, identificados de acordo com as regras semânticas. Poderá ser observado na seção a seguir maiores detalhes sobre a descrição do trabalho e um pouco mais sobre o programa desenvolvido.

2. Tabela de Símbolos

Durante a análise Semântica há dois objetivos que devem ser completados, sendo eles a construção da tabela de símbolos, e a identificação de erros e avisos, tendo como base o slide oferecido pelo professor, podendo ser observada na seção 3, assim como os erros e avisos esperados no conjunto de testes especificamente para análise semântica, e por fim, podar a árvore Sintática, o que resultará em uma árvore sintática abstrata. A criação da tabela de símbolos neste caso foi feita durante a própria análise Semântica, mais especificamente, antes de verificar as regras semânticas, já que ela será utilizada para realizar este passo. A tabela gerada ao fim é semelhante a tabela abaixo.

Token	Lexema	Tipo	dim	tam_dim1	tam_dim2	init	linha	funcao	parametros	valor
ID	a	inteiro	1	1	0	N	5	N	[]	[]
ID	b	flutuante	0	0	0	N	6	N	[]	[]

Tabela 1. Tabela de Símbolos.

Para geração desta tabela, foi utilizado o código fonte tpp a seguir (Listing 1).

Listing 1. Código fonte tpp sema-001.tpp

```
{Erro: Funcao principal n o declarada}
{Aviso: Variavel 'a' declarada e n o utilizada}
{Aviso: Variavel 'b' declarada e n o utilizada}

inteiro: a[1]
flutuante: b
```

É possível perceber que a tabela abaixo (Tabela 1) contém 11 colunas, e agora veremos o que cada uma delas significa. A coluna 'Token' representa o token da variável ou funçao que faz parte da tabela de símbolos, a coluna 'Lexema' contém o nome das variáveis declaradas ou utilizadas, assim como o nome das funções declaradas ou chamada, a coluna 'Tipo' contém o tipo das variáveis ou funções declaradas, a colunas 'dim' conterá '0' caso seja declarada uma variável que não é um vetor ou uma matriz, '1' caso a variável declarada seja um vetor ou '2' caso a variável seja uma matriz, e caso ela contenha mais de uma dimensão, os valores dessas dimensões serão armazenadas nas colunas 'tam-dim1' que representa o tamanho da primeira dimensão e 'tam-dim2', a qual representa o tamanho da segunda dimensão. Já a coluna 'init' representa se uma função ou variável foi declarada ou não, dizendo na coluna 'linha' onde ela foi encontrada. Por fim podemos observar as colunas adicionada principalmente para armazenar alguns dados das funções, sendo elas 'funcao' que indica se o lexema identificado é uma função ou não, e caso seja, na coluna é armazena os parâmetros passados. Para encerrar temos a coluna 'valor', que como o próprio nome diz, guarda os valores e tipos das atribuições feitas.

3. Regras Semânticas

Para que seja possível criar a Árvore Sintática Abstrata, é necessário tem um maior conhecimento sobre as regras semânticas aplicada à linguagem. Neste caso a regra semântica foi oferecida pelo professor no documento de especificação do desenvolvimento desta análise, podendo então ser observada a seguir.

3.1. Funções e Procedimentos

- Verificação se existe uma função principal no código que serve como função de inicialização
- Verificar se o tipo retornado da função principal é o mesmo do tipo da função principal
- Verificar se a quantidade de parâmetros de uma chamada de função é igual a quantidade de parâmetros formais de sua definição
- Verificar para todas as funções se o tipo do valor retornado é compatível com o tipo de retorno declarado
- Verificar se as funções são declaradas antes de ser chamadas
- Uma função qualquer não pode chamar a função principal
- É possível declarar uma função e não utilizá-la
- Se a função principal chamar ela mesma é necessário emitir um aviso

3.2. Variáveis

- Variáveis devem ser declaradas e inicializadas antes de serem utilizadas
- Não é possível escrever ou ler uma variável se ela não for declarada
- Não é possível ler uma variável declarada mas não inicializada

- Uma variável não deve ser declarada mais de uma vez

3.3. Atribuição

- É necessário que o tipo e o valor atribuído a uma variável sejam compatíveis
- A atribuição de um retorno de uma função seja compatível com o tipo da função

3.4. Coerções implícitas

- Atribuição de variáveis ou resultados de expressão de tipos distintos não é permitido (inteiro <=> flutuante)

3.5. Arranjos

- Um índice de um arranjo deve ser inteiro
- Verificar se o acesso ao elemento de um arranjo está fora de sua definição, e caso esteja é necessário emitir um erro

4. Árvore sintática Abstrata

A árvore sintática abstrata é a árvore gerada ao fim da Análise Semântica, tendo como base a Árvore Sintática obtida da Análise Sintática. Como citado anteriormente ela pode ser definida como a Árvore Sintática, porém, com alguns de seus nós podados, como uma forma de representar o código analisado utilizando a estrutura de árvore a qual conterá em seus nós apenas os símbolos terminais, diferente da árvore sintática que também apresenta a derivação dos nós a partir dos símbolos não terminais

5. Detalhes da implementação

Nesta seção veremos alguns detalhes sobre a implementação do projeto, mais especificamente sobre a construção da tabela de Símbolos e a identificação das regras semânticas. Após receber como parâmetro a árvore construída na Análise Sintática, a primeira coisa que faço é verificar a declaração de variáveis e a declaração de funções percorrendo a árvore, como é possível observar abaixo (Listing 2).

Listing 2. Código que encontra as de.

```
def monta_tabela_simbolos(tree, tabela_simbolos):
    dimensao_1 = ''
    dimensao_2 = ''
    dimensao = 0

    for filho in tree.children:
        if ('declaracao_funcao' in filho.label):
            # preciso do valor retornado e tipo do retorno
            tipo = ''
            nome_funcao = ''
            tipo_retorno = ''
            parametros = []
            retorno = []
            tipos = []

            # Encontrando os parametros
            # VERIFICAR SE ELES TEM DIMENSAO IGUAL A 1 OU MAIOR
            parametros = encontra_parametro_funcao(filho, parametros)

            # N o se esquecer de verificar tamb m os par metros da fun   o
```

```
            linha_declaracao = filho.label.split(':')
            linha_declaracao = linha_declaracao[1]

            tipo,nome_funcao, _,retorno,tipo_retorno,linha_retorno = encontra_dados_funcao(filho,'','','','','','')

            linha_dataframe = ['ID',nome_funcao,tipo,0,0,0,escopo,'N',linha_declaracao,'S',parametros,[]]
            tabela_simbolos.loc[len(tabela_simbolos)] = linha_dataframe

            # Declara as vari veis passada por parametro
            for p in parametros:
                for nome_param, tipo_param in p.items():

                    linha_dataframe=['ID',nome_param,tipo_param,0,0,0,escopo,'S',linha_declaracao,'N',[],[]]
                    tabela_simbolos.loc[len(tabela_simbolos)] = linha_dataframe

            if (retorno != ''):
                # Verifica o tipo do retorno
                pos = 0
                muda_tipo_retorno_lista = []
                for ret in retorno:
                    for nome_retorno, tipo_retorno in ret.items():

                        # tipos_variaveis_retorno.append(tipo_retorno)

                        # procura na tabela de s mbolos as vari veis
                        tipo_retorno = tabela_simbolos.loc[tabela_simbolos['Lexema'] == nome_retorno]

                        tipo_variaveis_retorno = tipo_retorno['Tipo'].values
                        if len(tipo_variaveis_retorno) > 0:
                            tipo_variaveis_retorno = tipo_variaveis_retorno[0]

                        muda_tipo_retorno = {}
                        muda_tipo_retorno[nome_retorno] = tipo_variaveis_retorno
                        muda_tipo_retorno_lista.append(muda_tipo_retorno)


                        tipos.append(tipo_variaveis_retorno)
                        pos += 1

                if len(tipos) > 0:
                    if ('flutuante' in tipos):
                        tipo = 'flutuante'
                    else:
                        tipo = 'inteiro'

                # Verificar se realmente veio algo no retorno
                linha_dataframe=['ID','retorna',tipo,0,0,0,escopo,'N',linha_retorno,'S',[],muda_tipo_retorno_lista]
                tabela_simbolos.loc[len(tabela_simbolos)] = linha_dataframe
```

A imagem acima identifica o nó nomeado de 'declaracao_funcao', realizando então as operações necessários para verificar a declaração de uma função e seu retorno, caso ela tenha.

Em seguida a análise semântica é realizada, seguindo uma sequência de passos para verificar as regras semânticas citadas acima (Seção 3). Na minha implementão há uma função 'verifica_regras_semanticas' que recebe como parâmetro a tabela de símbolos, com o objetivo de realizar as vericações semânticas. Abaixo podemos ver um trecho do código desta função, onde estou identificando se há variáveis que não foram declarada (Listing 3).

Listing 3. Código que verifica se há variáveis não declaradas

```
def verifica_regras_semanticas(tabela_simbolos):
    # variaveis = tabela_simbolos['Lexema'].unique()

    # pegar s  as vari veis
    variaveis = tabela_simbolos.loc[tabela_simbolos['funcao'] == 'N']

    # Valores  nicos  das vari veis declaradas e inicializadas
    variaveis_repetidas_valores = variaveis['Lexema'].unique()

    # Tirando vari veis do mesmo escopo difente, ficar com a declara  o
    for var_rep in variaveis_repetidas_valores:
        variaveis_repetidas = variaveis.loc[variaveis['Lexema'] == var_rep]

        if len(variaveis_repetidas) > 1:
            variaveis_repetidas_index = variaveis_repetidas[variaveis_repetidas['init'] == 'N'].index
            variaveis_repetidas_linhas = variaveis_repetidas[variaveis_repetidas['init'] == 'N']

            # Checar se elas s o do mesmo escopo
            # Pego os escopos
            escopos_variaveis = variaveis_repetidas_linhas['escopo'].unique()
```

```
21
22              # Passo por todas os escopos
23              for esc in escopos_variaveis:
24                  variaveis_repetidas_linhas=variaveis_repetidas_linhas.loc[variaveis_repetidas_linhas['escopo']==esc]
25                  .index
26                  variaveis_repetidas_escopo_igual_index = variaveis_repetidas_linhas
27                  variaveis.drop(variaveis_repetidas_escopo_igual_index[0] , inplace=True)
28
29          elif len(variaveis_repetidas) == 0:
30              print("Erro: Vari vel '%s' n o declarada" % var_rep)
```

Por fim a poda da árvore é feita, através da função 'poda_arvore' retira_no, a qual dada uma lista de labels, ela a retira tais nós que estão contidas nesta lista de labels da árvore, e depois rearranja os nós. Abaixo podemos verificar a função desenvolvida (Listing 4).

Listing 4. Código que realiza poda da árvore

```
1
2   def retira_no(no_remove):
3       auxilixar_arvore = []
4       pai = no_remove.parent
5
6       if no_remove.name in remover_nos:
7           for filho in range(len(pai.children)):
8               # Verifico se est  na lista de n s que quero remover
9               if pai.children[filho].name ==  no_remove.name:
10                  auxilixar_arvore += no_remove.children
11
12              # Caso nao esteja eu adiciono na lista do auxiliar
13              else:
14                  auxilixar_arvore.append(pai.children[filho])
15          pai.children = auxilixar_arvore
16
17      elif no_remove.name.split(':')[0] in remover_nos:
18          for filho in range(len(pai.children)):
19              # Verifico se est  na lista de n s que quero remover
20              if pai.children[filho].name ==  no_remove.name:
21                  auxilixar_arvore += no_remove.children
22
23              # Caso nao esteja eu adiciono na lista do auxiliar
24              else:
25                  auxilixar_arvore.append(pai.children[filho])
26
27          pai.children = auxilixar_arvore
28
29      if no_remove.name in verificar_nos:
30          if len(no_remove.children) == 0:
31              for filho in range(len(pai.children)):
32              # Verifico se est  na lista de n s que quero remover
33                  if pai.children[filho].name ==  no_remove.name:
34                      auxilixar_arvore += no_remove.children
35
36                  elif pai.children[filho].name.split(':')[0] == no_remove.name:
37                      auxilixar_arvore += no_remove.children
38
39
40                  # Caso nao esteja eu adiciono na lista do auxiliar
41                  else:
42                      auxilixar_arvore.append(pai.children[filho])
43
44              pai.children = auxilixar_arvore
45
46      elif no_remove.name.split(':')[0] in verificar_nos:
47          if len(no_remove.children) == 0:
48              for filho in range(len(pai.children)):
49              # Verifico se est  na lista de n s que quero remover
50                  if pai.children[filho].name ==  no_remove.name:
51                      auxilixar_arvore += no_remove.children
52
53                  elif pai.children[filho].name.split(':')[0] == no_remove.name:
54                      auxilixar_arvore += no_remove.children
55
56
57                  # Caso nao esteja eu adiciono na lista do auxiliar
58                  else:
59                      auxilixar_arvore.append(pai.children[filho])
60
61              pai.children = auxilixar_arvore
62
63
64
65  def poda_arvore(arvore_abstrata):
66      for no in arvore_abstrata.children:
67
68          poda_arvore(no)
69      retira_no(arvore_abstrata)
```

6. Exemplos

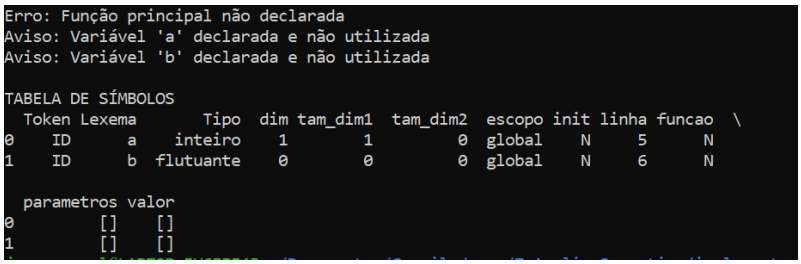
Veremos agora um exemplo de entrada e saída gerada ao passar como parâmetro um código tpp para realização da análise Semântica.
Abaixo é possível observar o código de entrada, com extensão tpp. Este código está presente no conjunto de testes oferecidos pelo professor, e pode ser localizado pelo nome de ”sema-004.tpp”.

Listing 5. Código fonte tpp

```
1 {Erro: Fun  o principal n o declarada}
2 {Aviso: Vari vel 'a' declarada e n o utilizada}
3 {Aviso: Vari vel 'b' declarada e n o utilizada}
4
5 inteiro: a[1]
6 flutuante: b
```

Como saída teremos os seguintes erros e avisos, podendo ser observados abaixo. Podemos observar nas linhas 1,2 e 3 no código acima que os erros e avisos retornados são os mesmos esperados pelo código (Listing 5).

A tabela gerada ao fim da execução é a seguinte.



```
Erro: Função principal não declarada
Aviso: Variável 'a' declarada e não utilizada
Aviso: Variável 'b' declarada e não utilizada

TABELA DE SÍMBOLOS
  Token Lexema       Tipo  dim tam_dim1  tam_dim2  escopo init linha funcao  \
0    ID      a    inteiro    1        1         0  global    N     5      N
1    ID      b  flutuante    0        0         0  global    N     6      N

  parametros valor
0         []    []
1         []    []
```

Figura 1. Tabela de Símbolos gerada.

Como já citado acima, é esperado ao fim da execução a Árvore Sintática Abstrata, podendo então ser observada abaixo. O arquivo criado tem o nome de ”sema-001.tpp.prunned.unique.ast.png”.

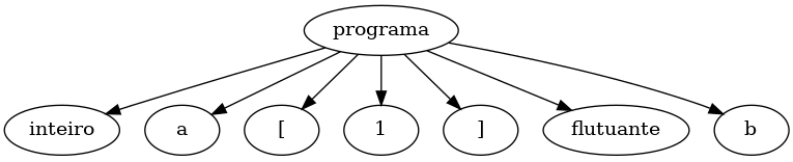


Figura 2. Árvore Sintática.

Esta Arvore Sintática Abstrata podada foi obtida podando os nós que podemos observar no código abaixo.

Listing 6. Lista de nós que serão podados

```
# N s que ser o retirados na poda
remover_nos = ['ID', 'var', 'lista_variaveis', 'dois_pontos', 'tipo',
    'INTEIRO', 'NUM_INTEIRO','lista_declaracoes', 'declaracao', 'indice',
    'numero', 'fator','abre_colchete', 'fecha_colchete', 'menos', 'menor_igual',
    'maior_igual','expressao', 'expressao_logica', 'ABRE_PARENTESE', 'FECHA_PARENTESE',
    'MAIS', 'chamada_funcao', 'MENOS','expressao_simples', 'expressao_aditiva', 'expressao_multiplicativa',
    'expressao_unaria', 'inicializacao_variaveis', 'ATRIBUICAO','NUM_NOTACAO_CIENTIFICA', 'LEIA',
    'abre_parentese', 'fecha_parentese', 'atribuicao', 'fator', 'cabecalho', 'FIM','operador_soma',
    'mais', 'chamada_funcao', 'lista_argumentos', 'VIRGULA','virgula', 'lista_parametros', 'vazio',
    '(', ')', ':', ',', 'FLUTUANTE', 'NUM_PONTO_FLUTUANTE', 'RETORNA', 'ESCREVA', 'SE', 'ENTAO', 'SENAO',
    'maior','menor', 'REPITA', 'igual', 'menos', 'menor_igual', 'maior_igual', 'operador_logico',
    'operador_multiplicacao', 'vezes','id', 'declaracao_variaveis', 'atribuicao', 'operador_relacional', 'MAIOR']
```

A árvore sintática original gerada ao fim da análise Sintática, a qual foi utilizada para realização da poda pode ser encontrada logo em seguida.

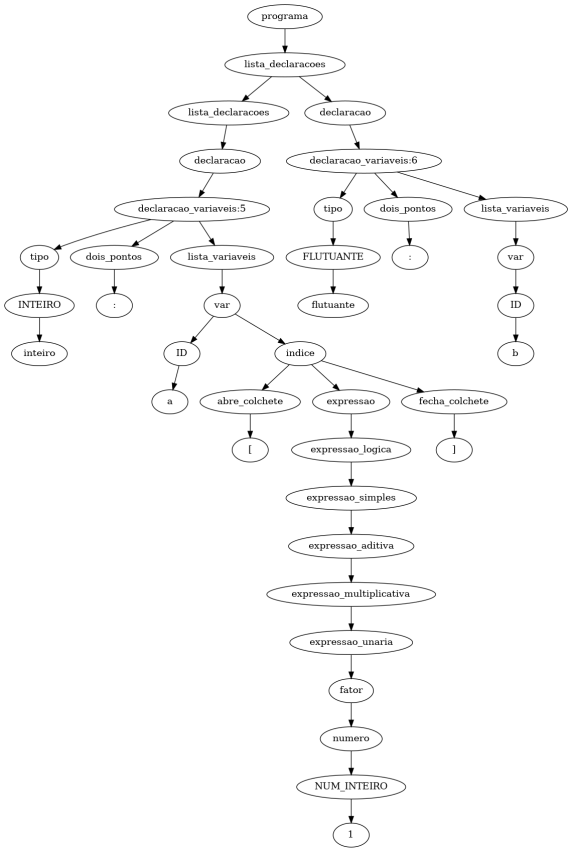


Figura 3. Árvore Sintática.

Referências

Gonçalves, R. A. Aula 016 - análise semântica. ttps://moodle.utfpr.edu.br/pluginfile.pp/1373325/mod$_r esource/content/0/aula - 16 - analise - semantica - regras - semanticas - tpp.md.slides.pdf$.

Gonçalves, R. A. (2021). Projeto de implementação de um compilador para a linguagem t++. ttps://moodle.utfpr.edu.br/pluginfile.pp/183223/mod$_r esource/content/16/trabalho - 03.md.notes.pdf$.