Compilador da Linguagem T++

Rafael Rampim Soratto [1]

[1]Universidade Técnológica Federal do Paraná - UTFPR CM

soratto@alunos.utfpr.edu.br

Abstract. *The present work has the objective of structuring a compiler for the T ++ language. For this a process sequence is adopted: The first process is called lexical analysis, the expected result of this process is a list of tokens with the exact value of the meaning of a certain expression. After this process is performed the parsing of the language where the result is a syntactic tree.*

Resumo. *O presente trabalho possui o objetivo de estruturar um compilador para a linguagem T++. Para isso é adotada uma sequência de processos: O primeiro processo é chamado de análise léxica, o resultado esperado deste processo é uma lista de tokens com o valor exato do significado de certa expressão. Após este processo é realizada a análise sintática da linguagem onde o resultado trata-se de uma árvore sintática.*

1. Introdução

Um compilador é o responsável por receber um código fonte e transformar em um código alvo [louden]. Para isso são realizados processos sequências que dividem as responsabilidades do compilador:

1. Análise Léxica;
2. Análise Sintática;
3. Análise Semântica;
4. Otimização de código fonte;
5. Gerador de código;
6. Otimizador de código alvo.

A primeira parte do trabalho está relacionado com o processo de varredura realizado pela análise léxica, o resultado desse processo é uma lista de tokens gerados à partir de um código fonte fornecido.

1.1. Especificação da Linguagem T++

A linguagem T++ é um exemplo de código fonte que será passado para o compilador. O primeiro processo realizado neste código fonte é a varredura realizada pelo analisador léxico.

Esta linguagem possui palavras reservadas para o seu funcionamento, elas são próprias da linguagem e sua sintaxe deve ser respeitada. Assim como as palavras reservadas existem símbolos que podem ser utilizados. Tanto as palavras reservadas e os símbolos precisam ser tratados pelo analisador léxico de maneira reconhecer todas as expressões da linguagem. Isto pode ser feito utilizando expressões regulares(autômatos finitos) [JARGAS].

Os ‘tokens’ e palavras reservadas utilizadas na linguagem são:

Tabela 1: **Tokens** da linguagem **T++**

palavras reservadas	símbolos
se	+ soma
então	- subtração
senão	* multiplicação
fim	/ divisão
repita	= igualdade
flutuante	, vírgula
retorna	:= atribuição
até	< menor
leia	> maior
escreva	<= menor-igual
inteiro	>= maior-igual
	(abre-par
	(fecha-par
	: dois-pontos
	[abre-col
	] fecha-col
	&& e-logico
	|| ou-logico
	! negação

A linguagem não possui strings e todas as marcas ou são palavras reservadas ou símbolos. Tudo aquilo que não é palavra reservada mas começa com letra são identificadores (nome de funções, variáveis, etc).

A linguagem T++ possui dois tipos de variáveis: inteiro e flutuante. Então os números podem assumir estes dois tipos.

2. Especificação formal dos Autômatos para as classes de token da linguagem

Os autômatos representam as expressões regulares reconhecidas no processo de varredura. Ele é responsável por indicar quais expressões serão aceitas para que se gere o token respectivo da expressão.

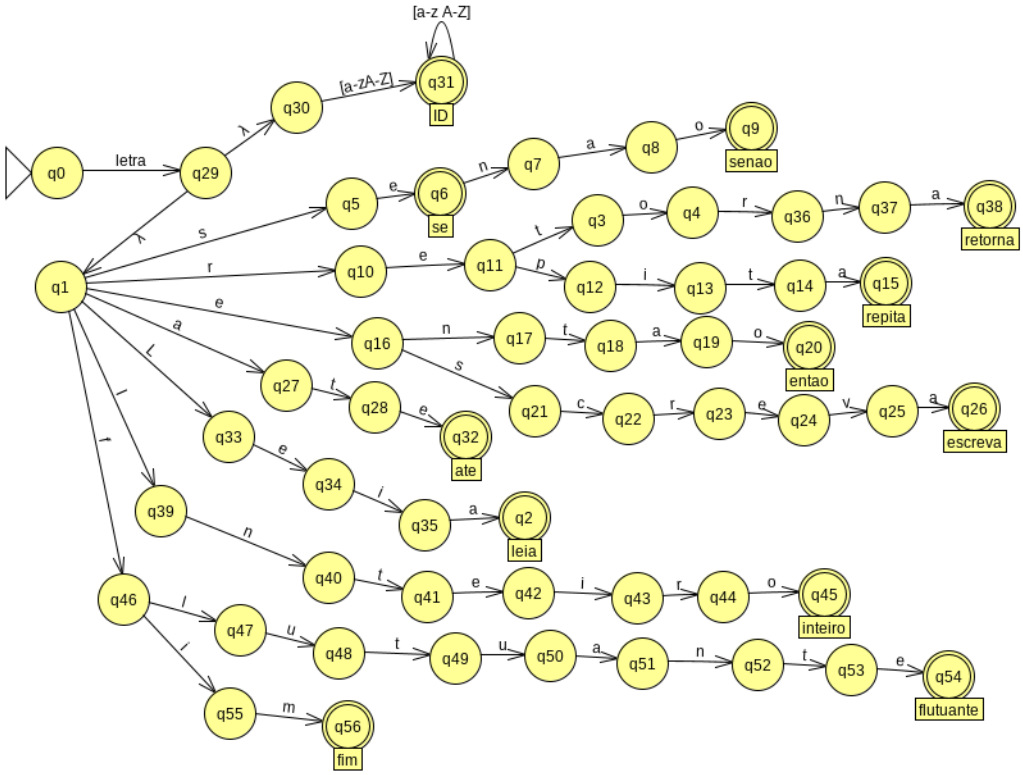
Construir o autômato de uma expressão regular permite visualizar com facilidade o funcionamento da expressão regular e também visualizar quais são os estados de aceitação do autômato.

2.1. Autômato para tratar dígitos

Quando se recebe um digito no arquivo .tpp ele pode ser enquadrar em qualquer uma das palavras reservadas e isso significa que o usuário está digitando um comando para ser interpretado.

Caso esse digito não chegue no estado de aceitação de nenhuma palavra reservada então ele será tratado como ‘ID’. Este ID é útil para nomear variáveis, nome de funções e etc .... O autômato que realiza o reconhecimento das palavras reservadas da linguagem em conjunto com reconhecimento dos ID’s pode ser visualizado na figura 1

Figura 1. Automato para reconhecer palavras reservadas e ID



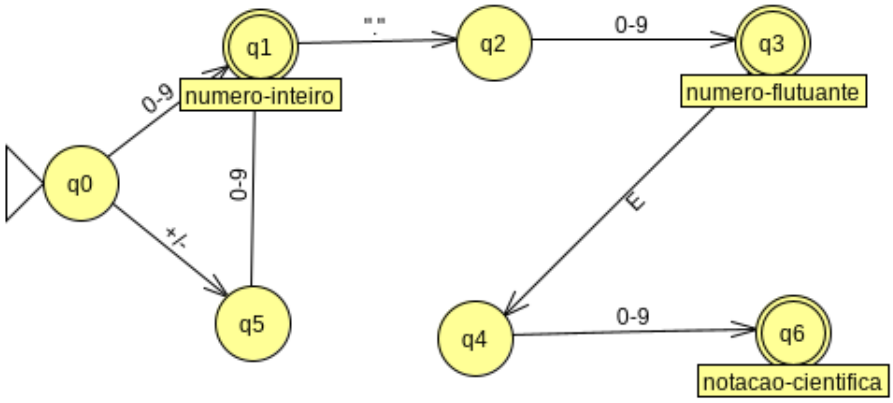
Fonte : Autoria Própria 2019.

De acordo com a figura1 é possível visualizar que as palavras reservadas podem começar com os mesmos dígitos (e.g.repita e retorna), então seu reconhecimento começa igual e quando o digito é diferente criam-se sub-árvores para definir qual token será reconhecido. Isso também ocorre com as palavras ‘se’ e ‘senão’ que começam iguais porém uma sub-árvore é gerada para o senão por possuir dígitos diferentes.

2.2. Autômato para tratar números

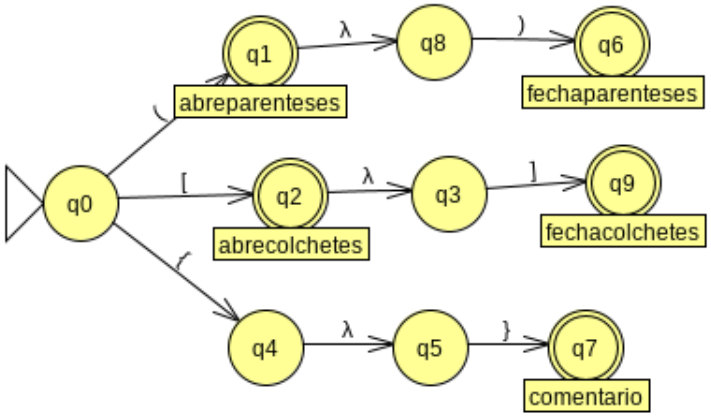
Quando recebe-se um número do teclado ou um símbolo de ‘+’ ou ’-’ então isso significa que o token a ser gerado será um dos 3 tipos de números reconhecidos pela linguagem: inteiro, flutuante e notação científica. Lembrando que se a expressão começar com dígito então não há como ser classificado como número e o nome de uma variável também não pode começar com números.

Figura 2. Autômato para reconhecer inteiros, flutuantes e notações científicas



Fonte : Autoria Própria (2019).

Figura 3. Autômato para reconhecer chaves,colchetes e comentários

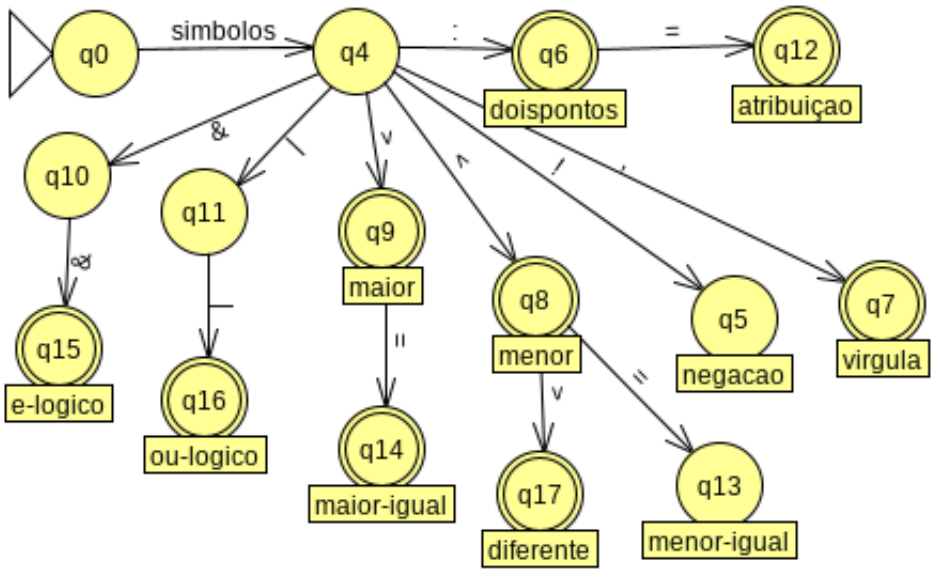


Fonte : Autoria Própria (2019).

2.3. Autômato para tratar operadores lógicos, relacionais e de atribuição

Os símbolos reconhecidos pelo autômato e pelas expressões regulares tratam-se de operadores lógicos(e, ou, negação) e operadores relacionais (maior, maior igual, menor, menor igual). Também são tratados como símbolos os dois pontos e a atribuição (dois pontos seguido do sinal de igual). Todos operadores podem ser visualizados na Figura 4:

Figura 4. Autômato para reconhecer símbolos

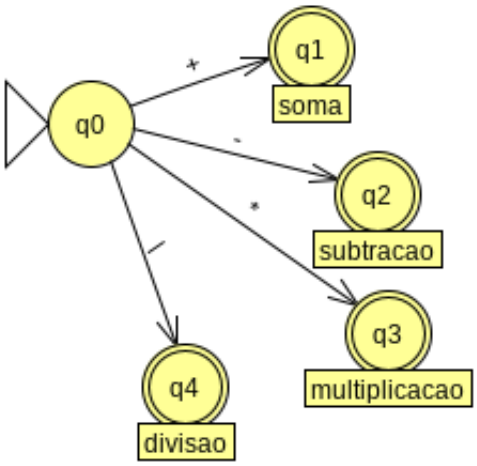


Fonte : Autoria Própria (2019).

2.4. Autômato para tratar operadores aritiméticos

Os operadores aritiméticos representam operações entre os números ou variáveis (soma, subtração, multiplicação, divisão), como ilustrado na Figura 5:

Figura 5. Autômato para reconhecer operações matemáticas



Fonte : Autoria Própria (2019).

3. Implementação do Analisador Léxico

A função do analisador léxico é receber a linguagem T++ (código fonte) e de acordo com o código lido deve ser gerada uma lista de *tokens* correspondentes com cada expressão da linguagem. O analisador léxico foi implementado utilizando a linguagem Python e o pacote PLY.

3.1. Lexer.py

Foi criado um pacote lexer.py para ser utilizado como ferramenta de análise léxica.

```
# -*- coding: UTF-8 -*-
import ply.lex as lex
from ply.lex import TOKEN
import sys
import re
# Palavras Rservadas
reserved = {
'se': 'SE',
'então': 'ENTAO',
'senão': 'SENAO',
'fim': 'FIM',
'leia': 'LEIA',
'escreva': 'ESCREVA',
'retorna': 'RETORNA',
'até': 'ATE',
'flutuante': 'FLUTUANTE',
'inteiro': 'INTEIRO',
'repita': 'REPITA',
}
# Lista de nomes dos tokens
tokens = [
# Logicals
'E', 'OU', 'NAO',
# Types
'NUMERO_PONTO_FLUTUANTE', 'NUMERO_INTEIRO',
# Arithmeticals
'SOMA', 'SUBTRACAO', 'MULTIPLICACAO', 'DIVISAO',
# Relationals
'MENORIGUAL', 'MAIORIGUAL', 'IGUALDADE', 'DIFERENTE', 'MENOR', 'MAIOR',
# Simbolos
'VIRGULA', 'ATRIBUICAO', 'ABREPARENTESES', 'FECHAPARENTESES', '
   ABRECOLCHETE', 'FECHACOLCHETE',
'ABRECHAVE', 'FECHACHAVE', 'DOISPONTOS',
# Outros
'ID', 'COMENTARIO']+list(reserved.values())

# NUMEROS
t_NUMERO_PONTO_FLUTUANTE = r'((\d+)(\.\d+)(e(\+|-)?(\d+))?|(\d+)e(\+|-)
   ?(\d+))'
t_NUMERO_INTEIRO = r'\d+'
```

Código 1. Analisador Léxico

3.1.1. Funções do Lexer

Além das palavras reservadas, símbolos e expressões regulares a classe Lexer possui duas funções essenciais:

1. Função para Buildar o Lexer;
2. Função para testar algum arquivo .tpp e retornar a lista de tokens.

Essas funções podem ser visualizadas no código a baixo:

```
lexer = lex.lex(debug=False)

if __name__ == '__main__':
lista_tokens = []
new_token = {}
code = open(sys.argv[1])
code_text = code.read()
lex.input(code_text)
while True:
tok = lex.token()
if not tok:
break
new_token['Tipo'] = tok.type
new_token['Linha'] = tok.lineno
new_token['Valor'] = tok.value
lista_tokens.append(new_token)
print(new_token)
new_token = {}
```

Código 2. Funções da Classe MyLexer em Python

4. Execução

Para executar um arquivo teste .tpp basta executar o arquivo principal 'lexer' e passar o nome do arquivo. Isso gera uma lista de tokens relacionados com este arquivo passado. Pode-se ainda passar a opção 'com' para gerar uma lista de tokens personalizada com o tipo e o nome dos tokens:

- python3 lexer.py teste.tpp

4.1. Exemplo

```
{recebe o vetor do usuario}
inteiro: vet[100]

recebeVetor(inteiro: n)
inteiro: i
i := 0
repita
leia(vet[i])
i := i + 1
até i < n
fim

{ função insertion sort }
insertion_sort(inteiro: n)
inteiro: i
inteiro: j
inteiro tmp
i := 1
repita
j := i
repita
tmp = vet[j]
vet[j] = vet[j-1]
vet[j-1] = tmp
```

```
j := j - 1
até (j > 0) && (vet[j-1] > vet[j])
i := i +1
até i < n
fim

{funcao principal}
inteiro principal()
recebeVetor(10)
insertion_sort(10)
retorna(0)
fim
```

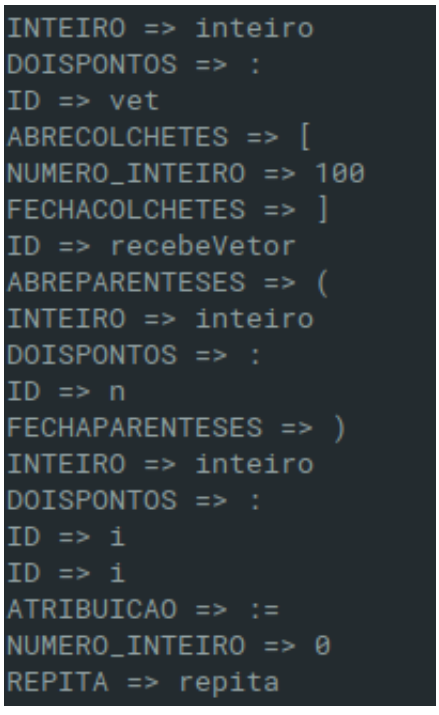
Código 3. Insertion Sort em t++

De acordo com o Código 3 podemos visualizar um arquivo.tpp contendo um algoritmo para ordenação de vetores "**insertion sort**".

Para executar a tradução da linguagem e gerar a lista de tokens correspondentes (varredura) basta utilizar o comando :
python3 lexer.py insertionsort.tpp.

Ao utilizar os comandos à cima será gerada lista de token. Na figura 6:

Figura 6. Lista de tokens



5. Analisador Sintático

A Análise Sintática determina a sintaxe ou estrutura de um programa. A sintaxe de uma Linguagem de Programação é normalmente dada pelas regras gramaticais de uma gramática livre de contexto (GLC). Assim como a estrutura léxica é determinada por expressões regulares.

Uma GLC utiliza convenções para nomes e operações muito similares às usadas por expressões regulares, a diferença é que as regras de uma gramática livre de contexto podem ser recursivas (permite aninhamento de ifs). A classe de estruturas reconhecíveis por GLC é significativamente maior que com Expressões Regulares.

Existem duas categorias gerais de algoritmos para análise sintática:

- Análise Sintática Ascendente: começa das folhas até a raiz (é a análise utilizada pelo pacote PLY do python também presente neste projeto);
- Análise Sintática Descendente (deriva as regras da raiz para as folhas);

O Analisador Sintático ou '**parser**' é responsável por determinar a estrutura sintática de um programa a partir das marcas (tokens) produzidos pelo processo de varredura (scanner) mostrado na seção anterior. O objetivo é construir, explícita ou implicitamente, uma árvore sintática que representa essa estrutura.

O analisador sintático pode ser visto como uma função de sequências de marcas produzidas pela varredura para a árvore sintática. Nem todas as sequências de tokens são programas válidos, o analisador sintático tem que distinguir entre sequências válidas e inválidas.

5.1. Gramáticas Livres de Contexo

Como apresentado na seção 2 durante o processo de análise léxica são utilizadas expressões regulares. Porém, na análise sintática não podemos utilizar expressões regulares pois não há como definir a precedência de operadores. Para resolver este problema são utilizadas as gramáticas livres de contexto, notação 'mais poderosa'.

Uma Gramática Livre de Contexo (CFG) é composta por:

- Conjunto de terminais T;
- Conjunto de não terminais V;
- 1 não terminal inicial S;
- Um conjunto de produções P;

  - Uma produção é um par de um não-terminal e uma cadeia (possivelmente vazia) de terminais e não-terminais

  - Podemos considerar produções como regras; o não-terminal é o lado esquerdo da regra e a cadeia é o lado direito
- Existe a linguagem denotada pela gramática G:L(G).

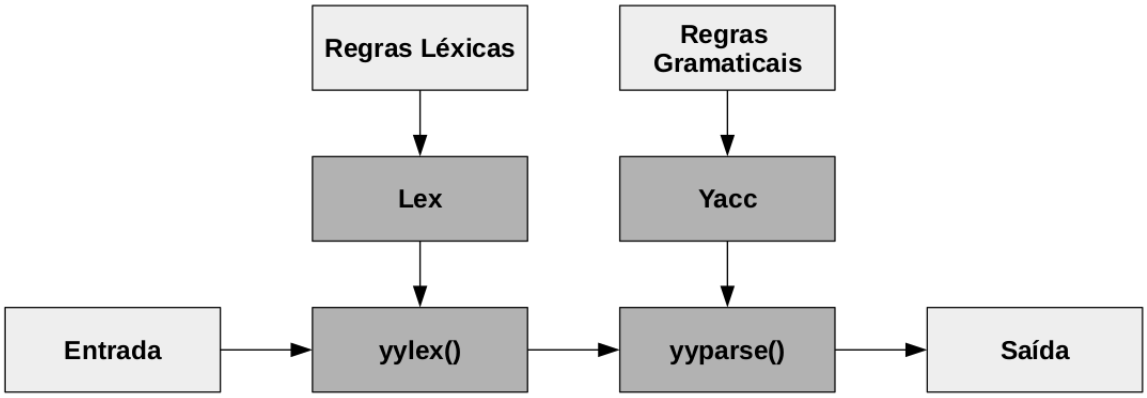
5.2. Formato da Análise sintática LALR(1)

De acordo com [Yacc] o Yacc (yet another compiler-compiler) é um gerador automático de Analisadores Sintáticos do tipo LALR(1). A maioria das ferramentas de análise sintática são expressadas convenientemente por analisadores LALR(1) pois ela permite diminuir os estados da tabela de símbolos do LR(1). Logo é construída uma máquina de estados referente a linguagem e então passa as informações de estados e transições para uma tabela de símbolos. Nesta tabela encontram-se os estados e para cada símbolo a célula da tabela mostra pra qual estado ir ... A diferença é que os estados iguais são unidos em um único estado na utilização do LALR.

5.3. Implementação e Yacc

O Yacc trabalha em conjunto com o Lex, o que é possível de ser visualizado na Figura:

Figura 7. Funcionamento do Yacc



5.4. Implementação da Árvore Sintática

Uma Árvore de Análise Sintática ou Árvore de 'parse' é uma árvore rotulada que possui:

- não-terminais nos nós interiores
- nós-folha são terminais
- filhos de cada nó interno representam a substituição do não-terminal associado ao passo de derivação.

Percorrer a árvore em ordem dá a cadeia sendo derivada A árvore sintática dá a estrutura e associatividade das operações que a cadeia original não mostra. Para construir a arvore em Python foi utilizada a estrutura de nós fornecida pelo pacote **anytree**. A raiz da árvore representa o programa principal e as folhas representam as variáveis e ID's.

Para cada token lido será gerada uma sub-árvore que representa uma regra. No final, quando todos tokens passarem pelo processo de '**parse**' então é esperável que todas as sub-árvores possuam como raiz o programa pŕincipal, '*main*'. Desta forma, basta imprimir a raiz que representa o programa para visualizar a árvore sintática do programa.

```
def criar_variavel(pai,line2,p):
  return Node('ID-' + p[1], parent=pai, line=line2)

# arvore sintática é implementada com nós
# um nó tem as seguintes informacoes:
# (nome, pai, numero)
def criar_no(name, parent=None, line=None):
# estrutura para mostrar o número e o tipo do nó
  if parent and line:
    return Node(name, parent=parent, line=line) # ID
  elif parent:
    return Node(name, parent=parent)
  else:
    return Node(name)
```

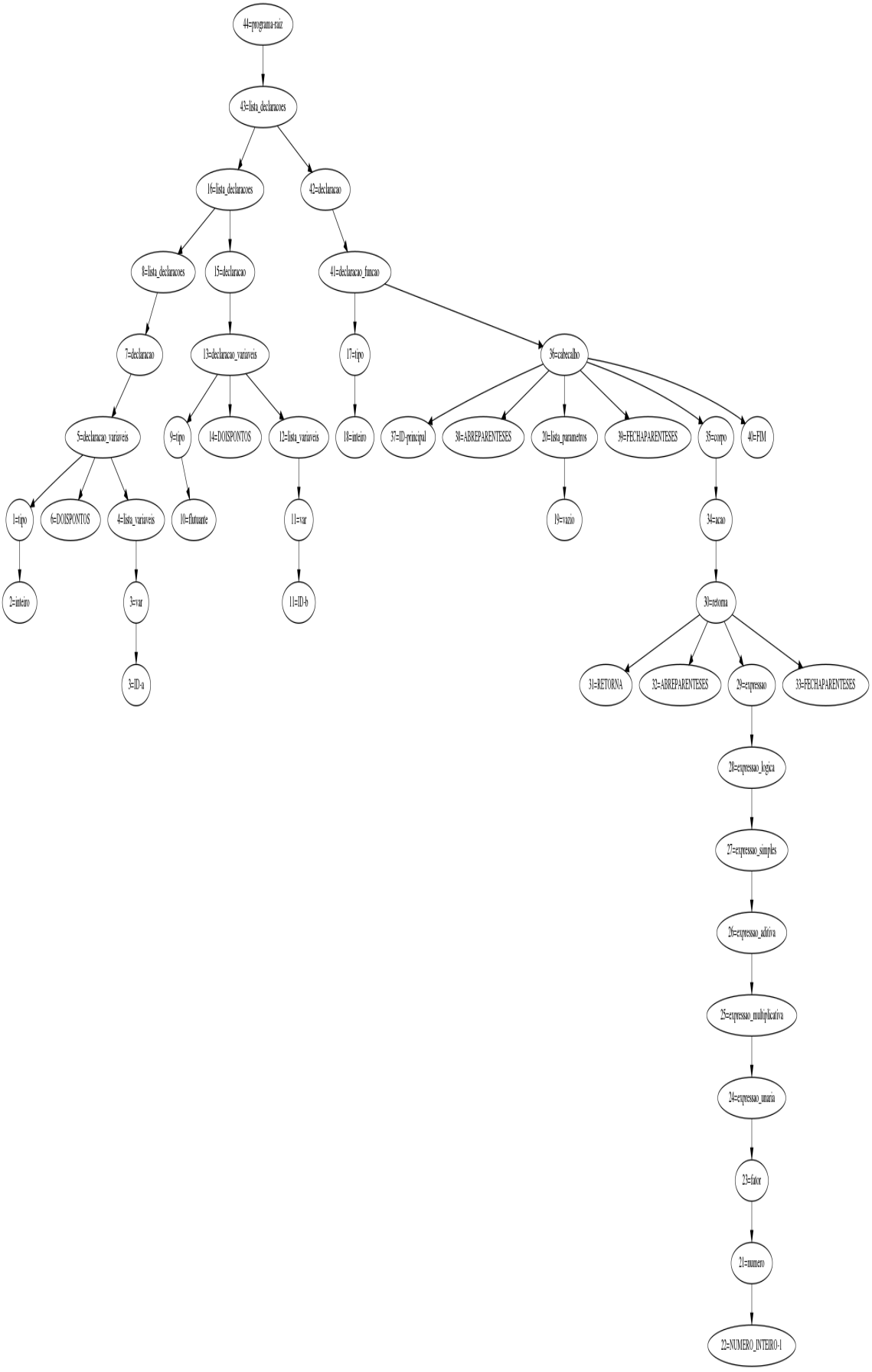
Código 4. Criação de Nós na Árvore Sintática

```
1
2  # funcao de erro
3  def p_erro(p):
4  # se for error de sintaxe
5    if p:
6  # mostra uma exeção indicando a linha e o token
7  # # print('Erro sintático na linha %d - Posição %d: '%s'' % (p.lineno,
       p.lexpos, p.value))
8    raise Exception('Erro sintático na linha {} no token '{}''.format(p.
       lineno, p.value))
9    else:
10   # reinicia o parser
11     parser.restart()
12     print('Erro sintático nas definições!')
13     exit(1)
14 # gera uma  execao de erro
15     raise Exception('Erro')
16
17 if __name__ == '__main__':
18 # main
19 #
20 # ativa o parser
21 parser = yacc.yacc(debug=True, tabmodule='fooparsetab')
22 #recebe o arquivo com códgio tpp
23 code = open(sys.argv[1])
24 # le arquivo
25 code_text = code.read()
26 # realiza a analise sintatica do código
27 try:
28 result = parser.parse(code_text, debug=False)
29 print('A raíz do programa: ',result)
30 except Exception as e:
31 raise e
32 code.close()
33 # se houver uma raiz então pode-se mostrar a ávore sintática dessa raiz
34 # se não houver uma raíz possui erro de construção sintática
35 if (raiz):
36 print('Gerando imagem da árvore...')
37 DotExporter(raiz).to_picture('arvore-sintatica.png')
38 else:
39 raise Exception('Houve erro ao tentar gerar a árvore'
```

Código 5. Função de Erros e Main do Parser

5.5. Exemplo de saída (árvore sintática)

Figura 8. Árvore Sintática



6. Análise Semântica

Dado que a árvore sintática foi construída sintaticamente de maneira correta, podemos construir uma tabela de símbolos contendo algumas informações essenciais sobre as variáveis e as funções do programa:

- **Funções, Procedimentos e Variáveis**:
  O identificador de função (nome e quantidade de parâmetros formais), além de que os parâmetros formais devem ter um apontamento para o identificador de variáveis. O identificador de variáveis locais e globais: nome, tipo e escopo devem ser armazenados na Tabela de Símbolos. Variáveis devem ser declaradas, inicializadas e antes de serem utilizadas (leitura). Lembrando que uma variável pode ser declarada no escopo global ou no escopo de uma função ou procedimento. Durante a criação da tabela de símbolos todas essas informações devem ser atribuídas nas variáveis e funções:
  - Token (id ou func);
  - Tipo;
  - Nome;
  - Linha;
  - Dimensão (arrays);
  - Retorno (funções);
  - Estado;
  - Número de parâmetros reais e formais (funções);

  Para armazenar, as funções e as variáveis é utilizada uma tabela de símbolos implementada como um dicionário em Python.

Com a tabela de símbolos construída e a árvore sintática correta é possível realizar as verificações semânticas do programa:

1. **Função Principal**: todo programa precisa ter uma função principal que retorna um valor inteiro.
2. **Parâmetros de uma função**: A quantidade de parâmetros reais de uma chamada de função/procedimento func deve ser igual a quantidade de parâmetros formais da sua definição.
3. **Retorno da Função**: o tipo de retorno da função deve ser equivalente ao tipo de declaração da função.
4. Uma função qualquer não pode fazer uma chamada à função principal. Se a função principal fizer uma chamada para ela mesmo, a mensagem de aviso deve ser emitida.
5. Uma função pode ser declarada e não utilizada. Se isto acontecer uma aviso deverá ser emitido.
6. **Variáveis**: A variável que foi declarada deve ser utilizada. Se a variável for utilizada sem ser declarada deve mostrar um erro. Não se pode declarar duas variáveis com o mesmo nome.
7. **Atribuições e Coerções implícitas**: As atribuições entre variáveis deve ser compatível, ou seja, uma variável inteiro deve receber um inteiro ou algo que retorne um inteiro. Da mesma forma para as variáveis do tipo flutuante. Porém não é gerado um erro quando isso ocorre, apenas um aviso... A diferença é que os avisos não interrompem a execução do programa como fazem os erros.

8. **Arranjos e Arrays**: Os vetores na linguagem tpp não podem ter o índice flutuante caso contrário deve-se retornar um erro. Se o array é definido com o índice 2 então na utilização desse vetor não se pode exceder o índice formal. Neste caso o erro (index out of range) é mostrado.

De acordo com a lista de erros e avisos que o analisador semântico deve mostrar para o usuário da linguagem Tpp é garantida que a árvore do programa não possua erros de funcionamento podendo então prosseguir para a fase de geração de código intermediário. Portanto a entrada do processo de análise semântica é a árvore gerada na sintática que é analisada na semântica utilizando uma tabela de símbolos como apoio e então gerando uma árvore coerente à um programa executável. O código que representa isso é:

```
# se houver uma raiz então pode-se mostrar a ávore sintática dessa raiz
# se não houver uma raíz possui erro de construção sintática
if (raiz):

  DotExporter(raiz).to_picture(arvore-sintatica1.png)
  # lista as funcoes e variáveis do programa
  tableofsymbols = tabela_variaveis(raiz)
  #print(tableofsymbols)
  verify_main(tableofsymbols)
  verify_functions(tableofsymbols,raiz)
  verify_variables(tableofsymbols,raiz)
  verify_index_var(tableofsymbols,raiz)
  verify_assignments(tableofsymbols,raiz)
  simplify_tree(raiz)

  DotExporter(raiz).to_picture(arvore-sintatica2.png)
else:
  raise Exception('Houve erro ao tentar gerar a árvore')
```

Código 6. Funções semânticas

7. Exemplo de utilização

Com o simples exemplo de código na linguagem tpp:

```
{Aviso: Variável 'a' declarada e não utilizada}
{Aviso: Variável 'b' declarada e não utilizada}
inteiro: a
flutuante: b
inteiro principal()
  retorna(1)
fim
```

Código 7. Código em TPP

Obtemos a tabela de símbolos e os avisos correspondentes:

Figura 9. Tabela de símbolos e funções

```
A raiz do programa:  Node('/44=programa-raiz')
Gerando imagem da árvore...
Aviso! Variavel 'a' da linha 5 foi declarada porém não utilizada
Aviso! Variavel 'b' da linha 6 foi declarada porém não utilizada
a : {'numero': 1, 'tipo': 'inteiro', 'token': 'ID', 'nome': 'a', 'linha': 5, 'dimensao': 0, 'indice': 0, 'estado': 'inicializada'
, 'escopo': 'global'}
b : {'numero': 2, 'tipo': 'flutuante', 'token': 'ID', 'nome': 'b', 'linha': 6, 'dimensao': 0, 'indice': 0, 'estado': 'inicializad
a', 'escopo': 'global'}
principal : {'numero': 3, 'token': 'func', 'tipo': 'inteiro', 'nome': 'principal', 'linha': 7, 'retorno': 'NUMERO_INTEIRO', 'esta
do': 'inicializada', 'parametros-formais': 0}
Gerando imagem da árvore...
```

De acordo com a figura 9 é possível visualizar que as variáveis são implementadas em tuplas na linguagem Python e cada tupla possui os atributos:

- Número
- Tipo: tipo da variável é inteiro ou flutuante;
- Token: ID se for variável e func se for função;
- Nome: nome da variável ou da função;
- Linha;
- Dimensão: 0 se for variável e 1 se for vetor, assim consecutivamente...
- Índice: O tamanho do array. Se for variável é 0.
- Estado: inicializada ou utilizada.
- Escopo: nome da função onde foi declarada ou global.

A diferença entre as tuplas de variáveis e funções é que as funções possuem os atributos:

- Retorno: tipo do retorno;
- Número de parâmetros formais e reais;
- Retorno real utilizado ao ser chamada. Utilizado para comparar o tipo de retorno real com o definido na declaração da função.

Ainda na Figura 9 é possível visualizar dois avisos sobre o programa que refere-se a duas variáveis definidas e não utilizadas. Se não houver erros semânticos então o resultado será duas árvores: uma completa e uma simplificada:

Figura 10. Árvore completa

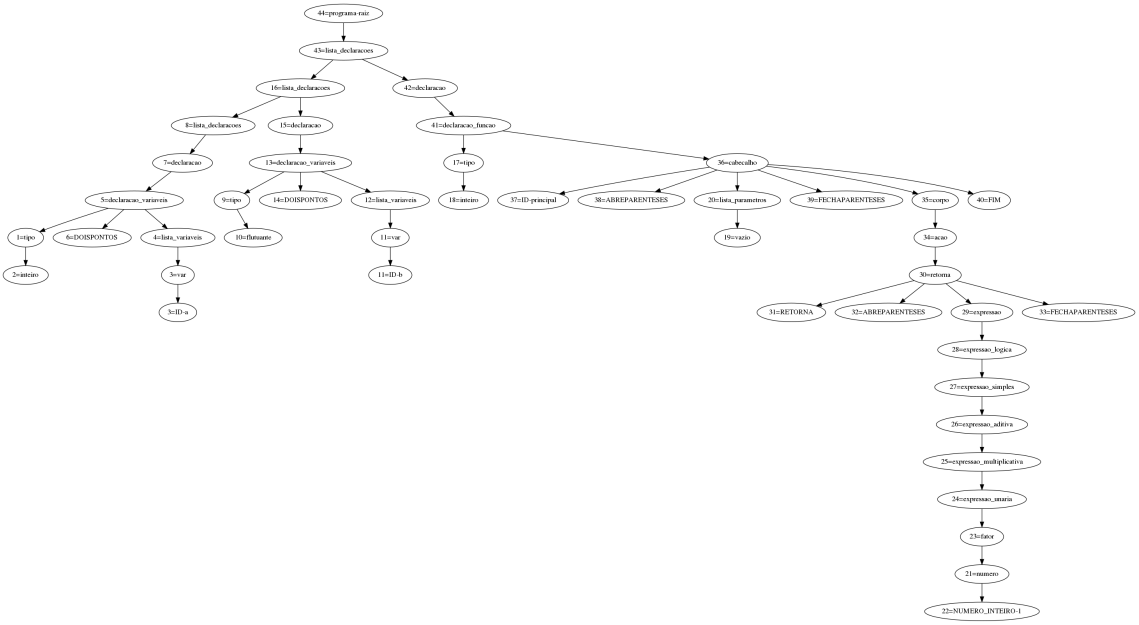
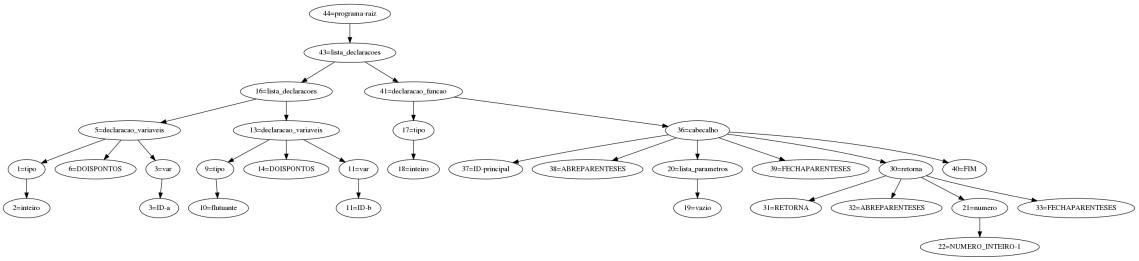


Figura 11. Árvore Simplificada sem erros Semânticos



8. Geração de Código

A última etapa do processo de compilação da linguagem T++ trata-se da geração de código executável à partir do código binário. Neste caso usaremos a árvore sintática (após o processamento semântico) para gerar o código binário e então o executável do programa.

O arquivo "gencode.py" contém as informações sobre a biblioteca para geração de código no Python:

```
from llvmlite import ir
from llvmlite import binding as llvm

def codegen():
  llvm.initialize()
  llvm.initialize_all_targets()
  llvm.initialize_native_target()
  llvm.initialize_native_asmprinter()
  module = ir.Module('geracao-codigo-tpp.bc')
  module.triple = llvm.get_default_triple()

  target = llvm.Target.from_triple(module.triple)
  target_machine = target.create_target_machine()
  module.data_layout = target_machine.target_data

  t_int = ir.IntType(32)
  t_func = ir.FunctionType(t_int, ())
  # Cria o módulo.
  module = ir.Module('meu_modulo.bc')
  arquivo = open('meu_modulo.ll', 'w')
  arquivo.write(str(module))
  arquivo.close()
  print(module)
```

Código 8. Bibliotecas para geração de código em Python

Para exemplificar o funcionamento da biblioteca llvmlite basta visualizar o seguinte código:

```
'''
Será gerado um código em LLVM como este em C:
int soma (int a, int b){
return a + b;
}
```

```
6  void teste (){
7  }
8  int main(){
9  int a = 1;
10 int b = 2;
11 int res = soma(a,b);
12 teste();
13
14 return res;
15 }
16 '''
17
18 # Cria o módulo.
19 module = ir.Module('meu_modulo.bc')
20
21 # Cria o cabeçalho da função soma
22 t_soma = ir.FunctionType(ir.IntType(32), [ir.IntType(32), ir.IntType
       (32)])
23 soma = ir.Function(module, t_soma, 'soma')
24 soma.args[0].name = 'a'
25 soma.args[1].name = 'b'
26
27 # Cria o corpo da função soma
28 entryBlock = soma.append_basic_block('entry')
29 builder = ir.IRBuilder(entryBlock)
30
31 res = builder.add(soma.args[0], soma.args[1])
32 builder.ret(res)
33
34 # Cria o cabeçalho da função teste
35 t_teste = ir.FunctionType(ir.VoidType(), [])
36 teste = ir.Function(module, t_teste, 'teste')
37
38 # Cria o corpo da função teste
39 entryBlock = teste.append_basic_block('entry')
40 builder = ir.IRBuilder(entryBlock)
41
42 builder.ret_void()
43
44 # Cria o cabeçalho da função main
45 t_main = ir.FunctionType(ir.IntType(32), [])
46 main = ir.Function(module, t_main, 'main')
47
48 # Cria o corpo da função main
49 entryBlock = main.append_basic_block('entry')
50 builder = ir.IRBuilder(entryBlock)
51
52 # int a = 1;
53 a = builder.alloca(ir.IntType(32), name='a')
54 builder.store(ir.Constant(ir.IntType(32), 1), a)
55
56 # int b = 2;
57 b = builder.alloca(ir.IntType(32), name='b')
58 builder.store(ir.Constant(ir.IntType(32), 2), b)
59
60 # int res = soma(a,b);
```

```
61 res = builder.alloca(ir.IntType(32), name='res')
62
63 call = builder.call(soma, [builder.load(a), builder.load(b)])
64 builder.store(call, res)
65
66 # teste();
67 builder.call(teste, [])
68
69 # return res;
70 builder.ret(res)
71
72 arquivo = open('meu_modulo.ll', 'w')
73 arquivo.write(str(module))
74 arquivo.close()
75 print(module)
```

Código 9. Bibliotecas para geração de código em Python

8.1. Execução dos testes de geração de código

Para visualização do processo de geração de código será utilizado o seguinte programa na linguagem nova tpp:

```
1
2 inteiro soma(inteiro: a, inteiro:b)
3   retorna(a + b)
4 fim
5
6 inteiro principal()
7   inteiro: a
8   inteiro: b
9   inteiro: c
10
11  leia(a)
12  leia(b)
13
14  c := soma(a, b)
15
16  escreva(c)
17
18  retorna(0)
19 fim
```

Código 10. Programa na linguagem tpp

Após percorrer o todos os processos (léxico, sintático, semântico e geração de código) então o executável do programa é gerado. Os testes podem ser executados como o seguinte comando:

```
1 $ python3 main.py geracao-codigo-testes/gencode-007.tpp
2 $ sudo ./run.sh
```

Código 11. Execução do códgio tpp no terminal

O primeiro comando no terminal gera o código binário e salva em um arquivo .ll. Então o segundo comando pega este arquivo .ll e gera o executável do programa.

O seguinte arquivo será gerado em “root.ll”:

```
; ModuleID = "geracao-codigo-tpp.bc"
target triple = "x86_64-unknown-linux-gnu"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"

declare i32 @"leiaInteiro"()

declare float @"leiaFlutuante"()

declare void @"escrevaInteiro"(i32 %".1")

declare void @"escrevaFlutuante"(float %".1")

define i32 @"soma"(i32 %"a", i32 %"b")
{
entry:S
br label %"exit"
exit:
%".7" = add i32 %"a", %"b"
ret i32 %".7"
}

define i32 @"main"()
{
entry:
%"a" = alloca i32, align 4
%"b" = alloca i32, align 4
%"c" = alloca i32, align 4
%".2" = call i32 @"leiaInteiro"()
store i32 %".2", i32* %"a"
%".4" = call i32 @"leiaInteiro"()
store i32 %".4", i32* %"b"
%".6" = load i32, i32* %"a"
%".7" = load i32, i32* %"b"
%".8" = call i32 @"soma"(i32 %".6", i32 %".7")
store i32 %".8", i32* %"c"
%".10" = load i32, i32* %"c"
call void @"escrevaInteiro"(i32 %".10")
br label %"exit"
exit:
ret i32 0
}
```

O segundo comando é o responsável por gerar o executável do programa para então realizar a utilização do programa de acordo com a Figura 12.

Figura 12. Programa executável

```
rafael@rafael-Z450LA:~/Documentos/compiler/compiler/compiler$ sudo ./run.sh
[sudo] senha para rafael:
rafael@rafael-Z450LA:~/Documentos/compiler/compiler/compiler$ ./root.exe
Digite um número inteiro
2
Digite um número inteiro
1998
2000
rafael@rafael-Z450LA:~/Documentos/compiler/compiler/compiler$
```

9. Conclusão

De acordo com este trabalho é possível visualizar o processo de compilação de um programa na linguagem T++ bem como todos os processos necessários para isso: análise léxica, sintática, semântica e geração de código.

Referências

LOUDEN, Kenneth C. 2004. *Compiladores*: Princípios e Práticas. 1st ed. São Paulo, SP: Thomson.

Syntax Highlight Guide. Acesso em 2019. Disponível em: https://code.visualstudio.com/api/language-extensions/syntax-highlight-guide

[JARGAS] JARGAS, Aurélio Marinho. 2012. Expressões regulares: uma abordagem divertida. 4 ed. São Paulo, SP: Novatec.

[Yacc]. Johnson, Stephen C. [1975]. Yacc: Yet Another Compiler Compiler. Computing Science Technical Report No. 32, Bell Laboratories, Murray hill, New Jersey. A PDF version is available at ePaperPress.