Compilador da Linguagem T++: Relatório de Implementação

Sávio O. Camacam[1]

[1]Departamento Acadêmico de Computação –
Universidade Tecnológica Federal do Paraná (UTFPR)
Caixa Postal 271 – 87.301-899 – Campo Mourão – PR – Brazil

saviocamacam@alunos.utfpr.edu.br

Abstract. *This work describes the implementation phases of the compilation tool created during the school year in Compilers, aiming to fix the main concepts of the actual compiler construction steps. The complete work comprises 4 phases of implementation: lexical analysis, syntactic analysis, semantic analysis and, finally, code generation. Since the implementation process is incremental, the report is also done in incremental format, with the four main topics related to these steps and their respective internal stages and characteristics.*

Resumo. *Este trabalho descreve as fases de implementação da ferramenta de compilação criada durante o período letivo na disciplina de Compiladores, tendo o objetivo de fixar os principais conceitos das etapas de construção de compilador real. O trabalho completo compreende 4 fases de implementação: análise léxica, análise sintática, análise semântica e por fim, geração de código. Sendo incremental o processo de implementação, o relatório se faz também no formato incremental, com os quatro grandes tópicos relativos a estas etapas e suas respectivas etapas e características internas.*

1. Introdução

Este relatório descreve o processo de construção da ferramenta de compilação para a Linguagem T++, que possui uma extensão de funcionalidades e características, e adaptação para o português, da linguagem-exemplo TINNY, do livro-texto Compiler Construction Principles, de Kenneth Louden [Louden].

Um programa nessa linguagem tem uma estrutura relativamente simples, mas com características de línguagens como C , possuindo sequências de *statements*, possuindo declarações e chamadas de funções, tipos de dados dimensionados e operadores lógicos.

Este trabalho está organizado da forma como segue: na próxima seção é feita uma especificação mais detalhada da linguagem, sobre seus itens e dados e declarações; em seguida são descritas as palavras reservadas da linguagem que são aceitas no Português; adiante são descritos os símbolos e operadores de manipulação da linguagem; logo depois é dada uma especificação formal dos autômatos da linguagem com descrição as expressões regulares usadas para captura dos tokens, bem como os autômatos que os representam; depois é explicado sobre o processo de execução do programa, bem como é exibida uma figura de exemplo de uma saída desse processador e por fim, uma conclusão sobre esta etapa do trabalho.

2. Análise Léxica

O processo de escaneamento, ou análise lexica, é a fase do processo de compilação que a ferramenta tem a tarefa de ler o código fonte a partir de um arquivo [Louden], lendo caractere a caractere e submetendo-os às regras que geram os *Tokens* desta linguagem. Tokens são basicamente palavras em linguagem natural formado pela sequência de caracteres lida do arquivo fonte, e representa uma unidade de informação.

Uma vez que a tarefa de extração dos tokens é executada, é necessário então a aplicação dos padrões de combinação que classificam essas unidades de informação, de acordos com às classes de uma linguagem, tais como: palavras-chave, idenficadores ou constantes numéricas.

2.1. Especificação da Linguagem de Programação T++

A linguagem T++ objeto desse estudo é relativamente simples nos termos de suas funções, mas tem a complexidade inerente das questões de implementação de qualquer linguagem. Abaixo estão discriminados os símbolos, descritores, operações e opções suportadas pela linguagem.

2.1.1. Tipos de Dados

Por se tratar uma linguagem experimental para estudo de compiladores, T++ apresenta apenas os tipos de dados primitivos:

- Inteiro;
- Flutuante.

Entretanto, a linguagem permite a criação de Tipos de Vetores e Matrizes a partir desses tipos primitivos, na seguinte sintaxe, além do reconhecimento de números em notação científica:

```
inteiro : var1
flutuante : var1
var1 := 1.54334E-34
inteiro : arr1[indice1]
inteiro : arr2[indice1][indice2]
```

2.1.2. Palavras Reservadas

A linguagem atente ao parâmetro de uma linguagem discorrida no Português, o que significa o suporte à caracteres especiais dessa língua, como acentuação e "ç".

Suas palavras reservadas são: **se, então, senão, fim, repita, flutuante, retorna, até, leia, escreve, e inteiro**. A seguir, um exemplo de código em T++:

```
1
2   inteiro: A[20]
3
4   inteiro busca(inteiro: n)
5
6           inteiro: retorno
7           inteiro: i
8
9           retorno := 0
10          i := 0
11
12          repita
13                  se A[i] = n então
14                          retorno := 1
15                  fim
16                  i := i + 1
17          até i = 20
18
19          retorna(retorno)
20  fim
21
22  inteiro principal()
23
24          inteiro: i
25
26          i := 0
27
28          repita
29                  A[i] := i
30                  i := i + 1
31          até i = 20
32
33          leia(n)
34          escreva(busca(n))
35          retorna(0)
36  fim
```

Figura 1. Código de exemplo em T++.

2.1.3. Símbolos e Operadores

Além do conjunto de palavras reservadas, essa linguagem também tem uma série de operadores mapeados para operações específicas:

- "*" : operador de ***multiplicação***;
- "-" : operador de ***subtração***;
- "+" : operador de soma***soma***;

- ":=" : operador de ***atribuição***;
- ":" : símbolo ***dois-pontos***;
- "<" : operador de comparação ***menor-quê***;
- ">" : operador de comparação ***maior-quê***;
- "=" : operador de ***igualdade***;
- "<=" : operador de comparação ***menor ou igual a***;
- ">=" : operador de comparação ***maior ou igual a***;
- "[" : símbolo abre ***colchetes***;
- "]" : símbolo fecha ***colchetes***;
- "(" : símbolo abre ***parênteses***;
- ")" : símbolo fecha ***parênteses***;
- "!" : operador lógico de ***negação***;
- "&&" : operador lógico ***conjunção***;
- "||" : operador lógico ***disjunção***;

2.2. Especificação Formal dos Autômatos

O sistema de varredura foi implementado com base em expressões regulares, descritas como se segue, gerando as marcas para identificação dos lexemas da linguagem:

- Palavra Reservada **t_PR = (se)|(então)|(senão)|(fim)|(repita)| (flutuante)|(retorna)|(até)|(leia)|(escreva)|(inteiro)**
  Para cada conjunto de caractere que representa uma palavra-chave é necessário criar uma regra específica que dará forma a um autômato para a aquela palavra, depois se unindo com os demais autômatos de outras regras.
- Identificador **t_ID = [a-zA-Zá-ñÁ-Ñ][a-zA-Zá-ñÁ-Ñ0-9_]**
  Regra que combina as definições de números e letras, acentuadas ou não, e underscore para identificar marcadores de identificadores da linguagem;
- Inteiro **t_INTEIRO = ?[0-9][0-9]***
  Regra genérica para captura de números inteiros;
- Flutuante **t_FLUTUANTE = ?[0-9][0-9]*[̇0-9]+([Ee][-]?[0-9]+)?**
  Regra genérica para captura de números flutuantes e em notação científica;
- Branco **t_ignore = [\t]**
  Regra para captura de símbolos de espaço e outros não imprimíveis que não fazem parte do programa;
- Símbolo **SB = (:=)|(:)|(>)|(<)|(()|())|(*)|(=)| (-)|(+)|(/)|(<=)|(>=)|([)|(]))**
  Da mesma forma como ocorre com as palavras reservadas, um autômato de símbolo é composto pelos estados com transições com os caracteres específicos dos símbolos da linguagem, que se une com as demais regras formando o autômato da linguagem.
- Comentário **t_COMMENT = {[̃}]*}**
  Regra que casa abertura e fechamento de comentários reconhecido pela linguagem;
- Erro de comentário **t_ERRORCT = [{}]**
  Regra de exceção para abertura ou fechamento sem sequência de comentários;

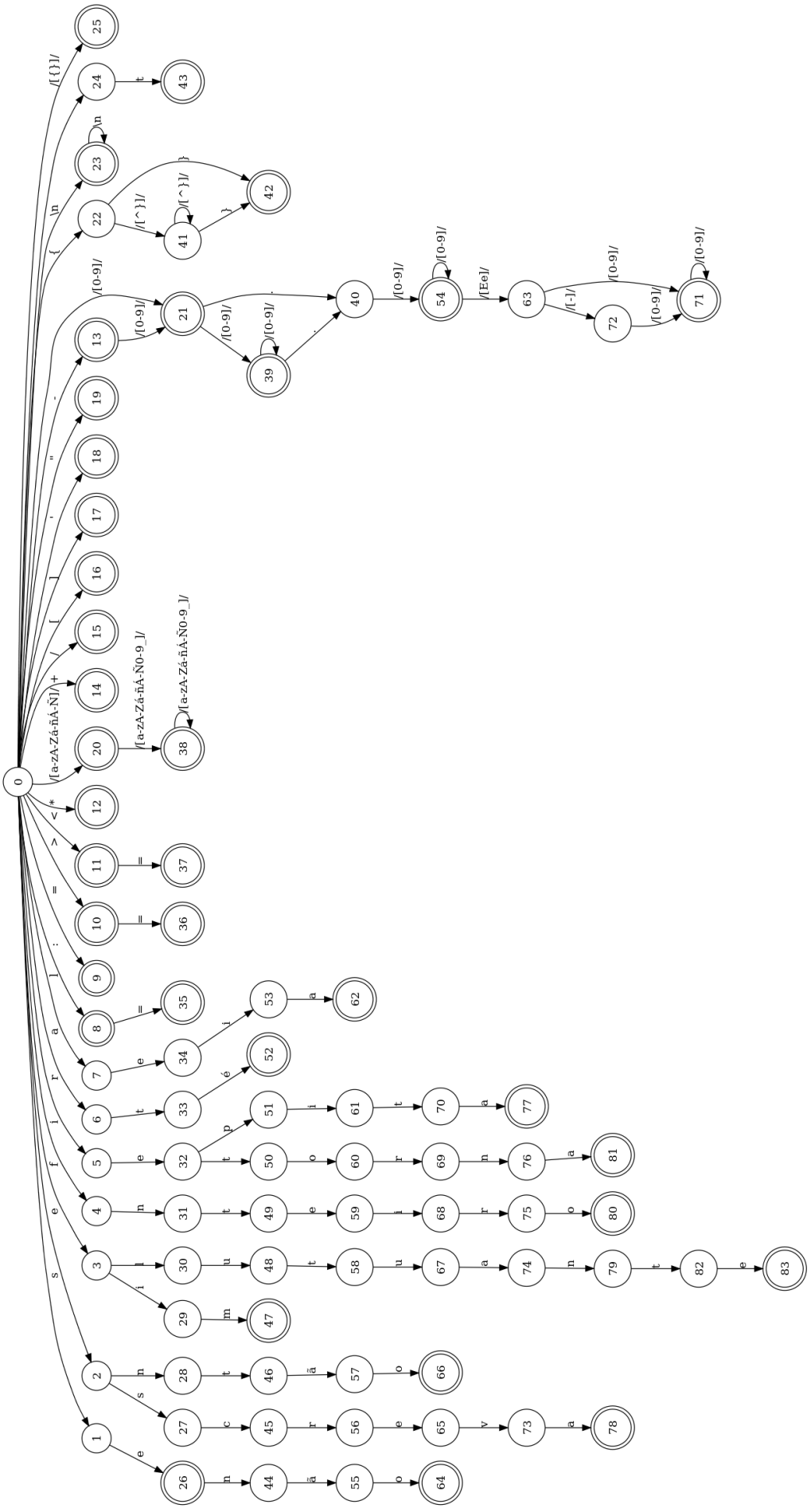

Figura 2. Autômato resultante da união das partes do analisador.

2.3. Detalhes de Implementação da Ferramenta

Considerando experiências passadas com o Java na implementação de ferramentas de compilação, para esse novo projeto foi usado o Python como linguagem-base para implementação com auxílio da biblioteca PLY.

O texto base para desenvolvimento da ferramenta pode ser encontrado em http://www.dabeaz.com/ply/ply.html [Beazley], que conta com exemplos básicos, detalhes de formulação de regras e modelo projeto.

Diferentemente da Linguagem Java, para o analisador léxico construído com o PLY é necessária a criação de um único arquivo de programação que representa uma única classe, nesse caso, chamada apenas de Lexer, que possui os métodos Test.

2.4. Resultados da Varredura

A entrada desse programa é composta de dois arquivos: o primeiro contém a descrição das regras de interpretação da linguagem em Lex; o segundo, contém o código da linguagem que deseja-se compilar.

O processo trata da leitura caractere a caractere do arquivo de entrada do programa - código fonte - como um *stream*, onde a partir dele, são buscadas combinações de regras léxicas dada a máquina de estados da linguagem que indica a criação das marcas dessa linguagem.

Como saída, é exibida no console do programa, a combinação <classe, valor> para cada marca válida da linguagem. Em situação de caractere ou sequência inválidos é apontado um erro, com especificação de caractere não identificado ou sequência inválida, como abertura e não fechamento de ***chaves*** que indica um comentário incompleto.

O projeto foi escrito usando a IDE PyCharm, e pode ser executado em Run > Run > nome_programa, sendo necessário também informar por parâmetro o nome do arquivo-fonte do programa a ser analisado.

Assumindo que o programa chegou até o caminho dos arquivos de entrada a saída será exibida dessa forma:

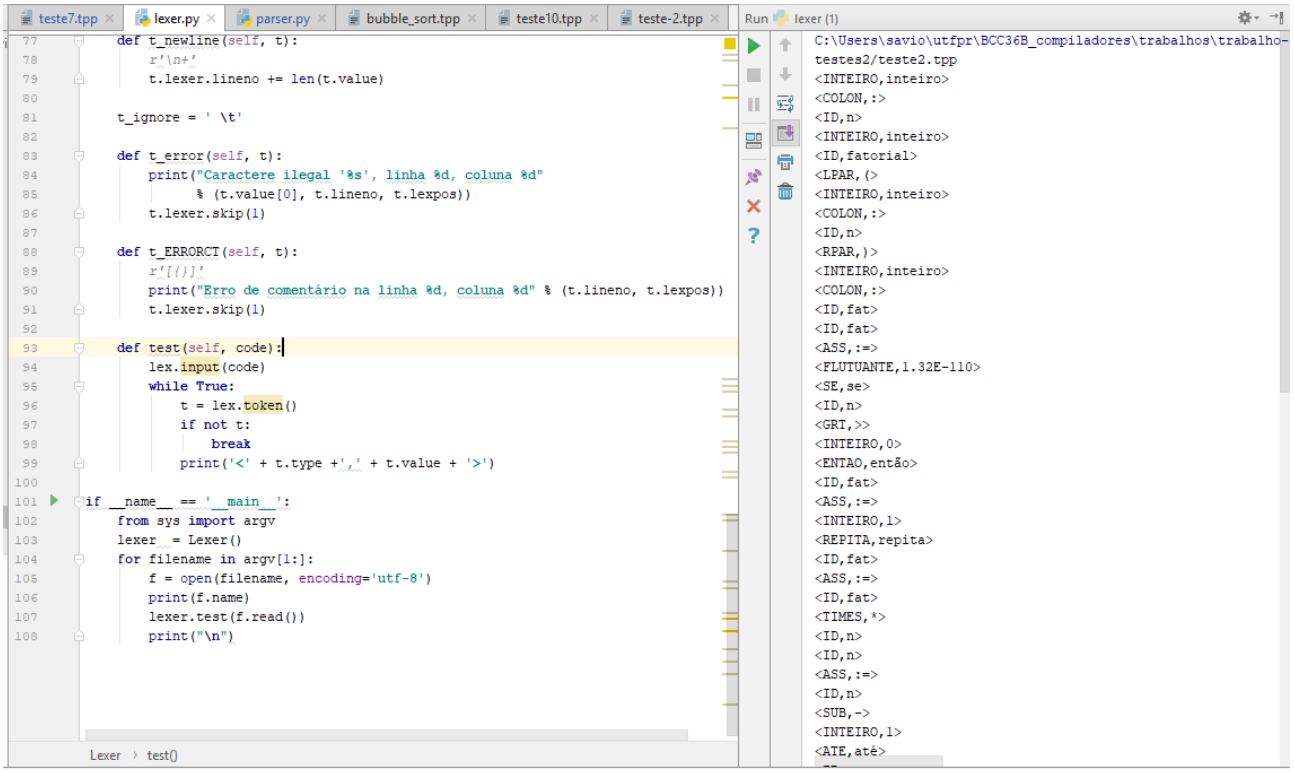

Figura 3. Situação de saída do programa.

3. Analisador Sintático

O módulo ply.yacc implementa o componente de análise sintática do PLY. O nome "yacc"é um acrônimo para "Yet Another Compiler Compiler"e foi emprestado da ferramenta de mesmo nome do Unix.

O yacc.py é usado para realizar análise sintática da linguagem, sendo que para isso é necessário ter uma gramática com especificações não ambíguas, expressa no formato BNF. As especificação da BNF para a linguagem T++ é apresentada na seção a seguir.

3.1. Descrição da Gramática

Nessa gramática, símbolos como INTEIRO, FLUTUANTE, *, -, +, / e ID são conhecidos como *terminais* e correspondem às entradas brutas de tokens. Já os identificadores como *expressão, declaração, var e índice* são usados para referencias produções que representam regras da gramática da linguagem, compreendidas por um conjunto de símbolos terminais e outras regras, são os símbolos não-terminais da linguagem.

programa	:=	lista_declaracoes
lista_declaracoes	:=	lista_declaracoes declaracao | declaracao | error
declaracao	:=	declaracao_variaveis | inicializacao_variaveis | declaracao_funcao | error
declaracao_variaveis	:=	tipo “:” lista_variaveis
declaracao_variaveis	:=	tipo “:”error
inicializacao_variaveis	:=	atribuicao
lista_variaveis	:=	lista_variaveis “,” var | var
lista_variaveis	:=	lista_variaveis “,” error
var	:=	ID | ID indice | ID lista_dimension
lista_dimension	:=	dimension | lista_dimension dimension
dimension	:=	“[”“]”
indice	:=	indice “[” expressao “]” |“[” expressao “]”
indice_error	:=	indice “[” error “]” |“[” error “]” | error “]” |“[” error
tipo	:=	INTEIRO | FLUTUANTE
declaracao_funcao	:=	tipo cabecalho | cabecalho
declaracao_funcao_error	:=	tipo cabecalho error | cabecalho error
cabecalho	:=	ID “(” lista_parametros “)” corpo FIM
cabecalho_error	:=	ID “(” lista_parametros “)” corpo error
lista_parametros	:=	lista_parametros ","parametro | parametro | vazio
parametro	:=	tipo “:” var | parametro
corpo	:=	corpo acao | vazio
acao	:=	expressao | declaracao_variaveis | se | repita | leia | escreva | retorna

se	:=	SE expressao ENTAO corpo FIM | SE expressao ENTAO corpo SENAO corpo FIM
se_error	:=	SE expressao error corpo FIM | error SENAO corpo FIM
repita	:=	REPITA corpo ATE expressao
repita_error	:=	REPITA corpo error
atribuicao	:=	var simbolo_atribuicao expressao | condicional | NOT condicional
atribuicao	:=	var simbolo_atribuicao error
condicional	:=	expressao_simples operador_relacional expressao_aditiva |“(” condicional “)” | condicional op_condicional condicional | condicional op_condicional error |“(” error “)” | error op_condicional condicional
simbolo_atribuicao	:=	“:=”
simbolo_condicional	:=	“||” | “&&”
leia	:=	LEIA “(”ID“)”
escreva	:=	ESCREVA “(” expressao “)”
retorna	:=	RETORNA “(” expressao “)”
expressao	:=	expressao_simples | atribuicao
expressao_simples	:=	expressao_aditiva | expressao_simples operador_relacional expressao_aditiva
expressao_aditiva	:=	expressao_multiplicativa | expressao_aditiva operador_soma expressao_multiplicativa
expressao_multiplicativa	:=	expressao_unaria | expressao_multiplicativa operador_multiplicacao expressao_unaria
expressao_unaria	:=	fator | operador_soma fator
operador_relacional	:=	“<” | “>” | “=” | “<>” | “<=” | “>=”
operador_soma	:=	“+” | “-”
operador_multiplicacao	:=	“*” | “/”

fator	:=	“(” expressao “)” | var | chamada_funcao | numero
numero	:=	NUM_INTEIRO | NUM_PONTO_FLUTUANTE
chamada_funcao	:=	ID “(” lista_argumentos “)”
lista_argumentos	:=	lista_argumentos “,” expressao | expressao | vazio

3.2. Formato de Análise

Yacc usa um formato de análise conhecido como “LR-parsing” ou análise por “shift-reduce”, que é uma técnica de análise *de baixo pra cima* que tenta reconhecer os valores de tokens associados aos valores da regra de produção do lato direito. Dessa forma, se um entrada válida for combinada com o lado direito de alguma das produções, um código de ação apropriado é ativado e os símbolos de gramática são substituídos pelo símbolo do lado esquerdo da regra de produção reconhecida.

```
Grammar                          Action
--------------------------------  ------------------------------------------
expression0 : expression1 + term   expression0.val = expression1.val + term.val
            | expression1 - term   expression0.val = expression1.val - term.val
            | term                 expression0.val = term.val

term0       : term1 * factor       term0.val = term1.val * factor.val
            | term1 / factor       term0.val = term1.val / factor.val
            | factor               term0.val = factor.val

factor      : NUMBER               factor.val = int(NUMBER.lexval)
            | ( expression )       factor.val = expression.val
```

Figura 4. Gramática de exemplo.

A análise LR é normalmente implementada usando a troca de símbolos da gramática numa pilha e sempre procurando nessa pilha e pelo próximo token de entrada, para combinar um padrão das regras da gramática. O exemplo a seguir ilustra o passo-a-passo que é executado na análise de uma expressão aritmética simples como 3 + 5 * (10 - 20) usando a gramática de exemplo definida na seção anterior. Nesse exemplo, o símbolo especial *dólar* representa o final da entrada.

```
Step Symbol Stack           Input Tokens            Action
---- ---------------------  ---------------------   --------------------------------
1                           3 + 5 * ( 10 - 20 )$    Shift 3
2    3                        + 5 * ( 10 - 20 )$    Reduce factor : NUMBER
3    factor                   + 5 * ( 10 - 20 )$    Reduce term   : factor
4    term                     + 5 * ( 10 - 20 )$    Reduce expr : term
5    expr                     + 5 * ( 10 - 20 )$    Shift +
6    expr +                     5 * ( 10 - 20 )$    Shift 5
7    expr + 5                     * ( 10 - 20 )$    Reduce factor : NUMBER
8    expr + factor                * ( 10 - 20 )$    Reduce term   : factor
9    expr + term                  * ( 10 - 20 )$    Shift *
10   expr + term *                  ( 10 - 20 )$    Shift (
11   expr + term * (                  10 - 20 )$    Shift 10
12   expr + term * ( 10                  - 20 )$    Reduce factor : NUMBER
13   expr + term * ( factor              - 20 )$    Reduce term : factor
14   expr + term * ( term                - 20 )$    Reduce expr : term
15   expr + term * ( expr                - 20 )$    Shift -
16   expr + term * ( expr -                20 )$    Shift 20
17   expr + term * ( expr - 20                )$    Reduce factor : NUMBER
18   expr + term * ( expr - factor            )$    Reduce term : factor
19   expr + term * ( expr - term              )$    Reduce expr : expr - term
20   expr + term * ( expr                     )$    Shift )
21   expr + term * ( expr )                     $    Reduce factor : (expr)
22   expr + term * factor                       $    Reduce term : term * factor
23   expr + term                                $    Reduce expr : expr + term
24   expr                                       $    Reduce expr
25                                              $    Success!
```

Figura 5. Exemplo do processo de análise por shif-reduce.

3.3. Implementação e Utilização do YACC

Para a implementação dessa segunda etapa do projeto da linguagem T++ foram usadas especificações da documentação do PLY disponíveis em http://www.dabeaz.com/ply/ply.html [Beazley] a partir da seção 5 que trata de Conceitos Básicos de Análise Sintática, YACC e Manipulação de Erros.

No programa de análise sintática, cada regra da gramática é definida por uma função em Python onde um *docstring* para cada função contém uma especificação apropriada da gramática livre de contexto da linguagem. As declarações que formam o corpo da função implementam as ações semânticas da regra. Cada função aceita um único argumento "p" que é a sequência contendo os valores de cada símbolo da gramática da regra correspondente. Os valores de p[i] são então mapeados para os símbolos da regra na gramática, como mostrado a seguir:

```
def p_expression_plus(p):
    'expression : expression PLUS term'
    #   ^            ^        ^    ^
    #  p[0]         p[1]     p[2] p[3]

    p[0] = p[1] + p[3]
```

Figura 6. Exemplo de uma função de produção no PLY.

Quando qualquer erro é detectado na especificação da gramática, o programa irá produzir uma mensagem de diagnóstico e possivelmente levantar uma excessão no processo de análise. Alguns dos erros que podem ser detectados inclui:

- Nomes de funções duplicados;
- Conflitos shift/reduce e reduce/reduce gerados por ambiguidades;
- Especificações mal-formadas de regras da gramática;
- Recursões infinitas;
- Regras ou Tokens não utilizados; e
- Regras ou Tokens indefinidos.

3.3.1. Produções Vazias e Manipulação de Erros Sintáticos

O programa pode manipular produções vazias com a definição da seguinte regra:

```
def p_empty(p):
    'empty :'
    pass
```

A partir disso é possível usar a produção vazia, simplesmente usando 'empty' como símbolo:

```
def p_optitem(p):
    'optitem : item'
    '        | empty'
    ...
```

No programa foram criadas algumas produções extras que faziam o tratamento de alguns erros específicos de combinação das regras da linguagem.

O principal modo de manuseio de erros é exibido a seguir. Ele segue o comportamento de casamento de erro com uma possível regra da gramática, para que assim seja possível, notificar a produção específica que o erro o occoreu, e assim sugerir o símbolo ou formato correto esperado:

```
def p_statement_print(p):
    'statement : PRINT expr SEMI'
    ...
```

```
def p_statement_print_error(p):
    'statement : PRINT error SEMI'
    print("Syntax error in print statement. Bad expression")
```

```
def p_statement_print_error(p):
    'statement : PRINT error'
    print("Syntax error in print statement. Bad expression")
```

3.4. Implementação da Árvore Sintática Abstrata

O PLY não fornece nenhuma função especial para construção de árvores sintáticas abstratas. Dessa forma, foi usado o modelo de árvore genérica, que permitiria uma travessia mais simples pela árvore:

```
class Node:
    def __init__(self,type,children=None,leaf=None):
         self.type = type
         if children:
              self.children = children
         else:
              self.children = [ ]
         self.leaf = leaf

def p_expression_binop(p):
    '''expression : expression PLUS expression
                  | expression MINUS expression
                  | expression TIMES expression
                  | expression DIVIDE expression'''

    p[0] = Node("binop", [p[1],p[3]], p[2])
```

Figura 7. Exemplo de classe que representa nós da árvore e uma chamada de produção que forma este nó.

Depois que o processo de análise é concluído e uma árvore sintática está na memória, o programa faz uma travessia a partir do nó raiz e todos os nós subsequentes, para a partir daí gerar uma estrutura parentesada de onde pode ser visualizada a árvore do programa. O site usado para visualizar a árvore é o http://mshang.ca/syntree/.

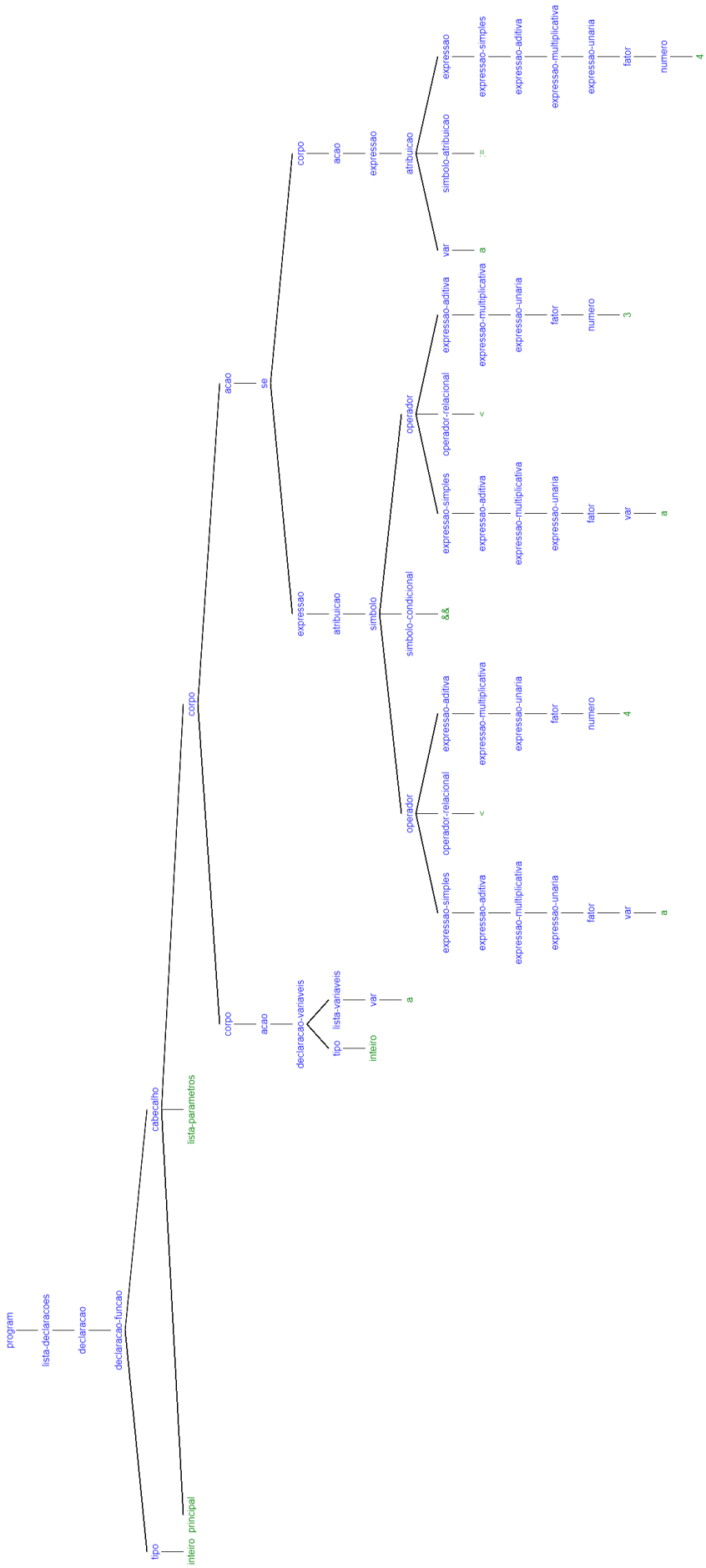

Figura 8. Exemplo de saída do programa para a linguagem T++.

4. Conclusão

4.1. Etapa de Análise Léxica

Com a conclusão desta etapa da implementação da ferramenta de compilação para a linguagem T++, os conceitos de design de compiladores continuam a ser fixados e aprimorados com a revisão da estrutura do analisador, agora no uso da biblioteca PLY, o que mostrou ser mais simples, ao contrário do uso do Java CUP que envolvia uma quantidade muito maior de classes e arquivos, que complicava o entendimento do processo de geração automática e agravava a migração do projeto para a fase seguinte, de implementação do analisar sintático.

4.2. Etapa de Análise Sintática

Nesta etapa do trabalho foram revistos alguns dos conceitos de técnicas de análise, bem como a implementação da gramática para a linguagem T++ com base numa especificação em BNF que foi construída em sala. Foram realizadas melhorias na gramática proposta com a adição de expressões condicionais, o que implicou na criação de novas regras e modificações das já existentes, como foram realizadas melhorias nas regras para lista de parâmetros de cabeçalho de funções, para suporte de quantidade ilimitadas de dimensões de *arrays*. A recuperação de erros passou a ser melhor documentada com especificação das regras da gramática, que implicou na criação de estruturas adicionais.

Referências

Beazley, D. M. Ply version: 3.9.

Louden, K. C. *Compiler Construction Principles*.