Compilador T++

Thiago Alexandre Nakao França

[1]Universidade Tecnológica Federal do Paraná (UTFPR)
Caixa Postal 271 – 87301-899 – Campo Mourão – PR – Brazil

Abstract. *The goal of this paper is to detail the development process of a T++ compiler, the processes was split into 4 main phases: lexical, syntactic and semantic analyzes, followed by the generation of intermediate code, at this moment only the lexical analyze is complete, we used the PLY framework to identify the tokens from the input programs.*

Resumo. *O objetivo deste artigo é o detalhamento do processo de desenvolvimento de um compilador na linguagem T++, tal processo se dá em quatro fases distintas, sendo elas a análise léxica, análise sintática, análise semântica e geração de código intermiário, no dado momento somente a análise léxica e sintática foram completadas, o processo de tokenização e construção da árvore sintática foram feitos com o auxílio da biblioteca PLY.*

1. Introdução

Compiladores é uma disciplina clássica na área da computação, é nela onde a maioria dos acadêmicos passa a entender como as instruções de uma linguagem de alto nível são convertidas para um código intermediário(semelhante ao assembly), compreender o funcionamento desse processo ainda pode ser útil para o desenvolvimento de linguagens interpretadas, que desfrutam de um processo deveras semelhante. Neste artigo, desenvolvemos e documentamos um compilador para a linguagem fictícia T++ em python, para tal o processo de leitura do arquivo de entrada até a geração de código intermediário se dá em etapas clássicas, que são:

- Analise léxica.
- Analise sintática.
- Analise semântica.
- Geração de código intermediário.

2. A linguagem T++

A linguagem escolhida é uma linguagem de programação em alto nível, suas palavras reservadas são em português com direito a acentuação, possui tipos numéricos, operadores lógicos, operadores matemáticos, declaração de funções e variáveis, chamada de funções, um operador condicional e um operador de loop, por ser uma linguagem imperativa a forma de raciocínio para se programar nela se assemelha a linguagens tradicionais como C, Java e Python, todas essas especificações encontram-se nesta seção.

2.1. Tipos de variáveis

A linguagem tem dois tipos de variáveis, sendo eles inteiro e flutuante.

2.2. Operadores

Existem os operadores lógicos AND e OR, representados da mesma forma que na linguagem C por && e ||, respectivamente, além disso, a linguagem conta com operadores matemáticos de soma, subtração, multiplicação e divisão, sendo +,-,* e /, respectivamente. É importante mencionar a existência dos operadores de comparação, usados para verificar igualdade, desigualdade e maioridade, sendo eles =, <>, >, <, >= e <=.

Expressões unárias também são válidas.

2.3. Declarações e atribuições de variáveis

As variáveis obrigatoriamente precisam ter o tipo a elas anexado em sua declaração, passado esse momento, dentro do escopo em questão elas podem ser invocadas sem a necessidade do seu tipo, por exemplo:

- inteiro: a,b
- a:=2
- b:=3

Para a separação da identificação do tipo de uma variável e a sua label, utiliza-se o caracter ':', além disso, mais de uma variável pode ser declarada na mesma linha, contanto que possuam o mesmo tipo. A atribuição de variáveis se dá pelo operador ':='.

2.4. Declarações de funções

Na linguagem t++ é possível a criação de funções, toda a função precisa explicitar qual o tipo de saída esperada e quais os parâmetros de entrada, segue um exemplo:

- inteiro principal(inteiro:n,flutuante:p)
- codigo
- fim

Observe que se faz necessária a declaração do tipo da função, bem como a declaração do tipo de todos os paramêtros presentes na lista de entrada, a virgula é usada como separador entre parâmetros, e a palavra reservada fim para denotar o final do escopo da função.

2.5. Laço de repetição e condicionais

A linguagem fornece uma forma de aplicação de estruturas de condicional, na forma de SE condição ENTÃO ação SENÃO ação FIM, com o bloco senão sendo opcional.

Além disso, é possível a utilização de laços de repetição, sendo que utiliza as palavras reservadas REPITA e ATE na forma REPITA ação ATE condição.

2.6. Leitura e escrita

Na linguagem é possível realizar a leitura de variáveis passadas pelo usuário com o comando LEIA(expressão), e pode-se escrever no console padrão com o comando ESCREVA(expressão).

2.7. Comentários

Os comentários são todos os caracteres contidos entre os caracteres { e }, nessa ordem especifica.

3. Análise léxica

O objetivo desta etapa é ler o programa de entrada, e agrupar os conjuntos de caracteres de interesse de uma forma clara e sintetizada, formando as entidades que chamamos de tokens.

3.1. Tokens

Um token nada mais é que uma estrutura de dados que armazena duas informações, um identificador de tipo e outro de valor, visualmente podem ser representados por <tipo, valor>, usaremos essa estrutura para identificar blocos de interesse no código de entrada.

Por exemplo, seria interessante se agrupássemos todas as palavras reservadas da linguagens contidas em um código na forma

<PalavraReservada, RETORNA>

<PalavraReservada, SE>

<PalavraReservada, SENAO>

E separar os identificadores de entidades em

<identificador, var1> e <identificador, var2>

Montando essas estrutura podemos separar quais palavras e/ou simbolos nos interessam do texto de entrada agrupados pela forma que nos for mais conveniente, isso facilita e muito o trabalho nas próximas etapas.

Então, na realidade o objetivo final da analise léxica é transformar o código de entrada em uma lista de tokens, e a partir desse momento o arquivo de entrada não será mais utilizado no decorrer do processo, dado que estamos lidando com a linguagem T++, precisaremos de agrupamentos para identificadores, palavras reservadas, símbolos, comentários, números inteiros e reais.

3.2. Planejamento

Agora que sabemos que devemos gerar uma lista de tokens, a próxima questão é como faremos isso, na literatura existem duas abordagens clássicas adequadas para implementar o tokenizador, sendo pela utilização de automatos ou fazendo uso de expressões regulares. Neste trabalho optamos pelo uso de expressões regulares, mas iremos exibir os autômatos de cada agrupamento de token de interesse de forma visual.

Primeiramente, daremos nomes as categorias de tokens que utilizaremos e o que elas podem armazenar, segue:

- OID = Identificador de variáveis e demais objetos
- RWO = Palavras reservadas(repita,ate,se,senao,entao,fim,inteiro,real,leia,escreva,retorna)
- SYB = Símbolos(, + - * / := < > <= >= = ( ) && || }:)
- INT = Numero inteiro
- FLT = Numero flutuante
- CMT = Comentários(Texto dentro dos carácteres { }, o token { estará incluso, mas o de fechamento excluso do valor dos tokens de match)

3.3. Materiais e métodos

Para a implementação do tokenizador, foi utilizado o framework Ply(python3.4), pois o método de implementação das expressões regulares é prático e de fácil entendimento utilizando essa ferramenta, para a utilizar basta realizar a instalação e depois importar a biblioteca ply.lex para o seu código python.

No ply.lex, a ideia é que para cada categoria de token seja definida uma expressão regular que será responsável por identificar os valores a ela pertencidos no texto de entrada, todos os grupos de tokens são definidos em um array que contém seus nomes.

Para cada nome de token, deve haver uma variável com o nome t_{nomeGrupo}, um exemplo no nosso cenário seria t_SYB, essa variável deve receber uma string indicando qual a expressão regular a ser utilizada para encontrar todos os tokens do grupo em questão.

```
tokens = ['CMT','SYB','FLT','INT','OID']+list(reservedList.values())
```

No exemplo acima, vemos que além da lista, existe uma referência a uma estrutura chamada reservedList, se trata ve uma estrutura de dados que contém todas as palavras reservadas da linguagem T++, como segue:

```
reservedList = {
    'se'          : 'RWO'  ,
    'SE'          : 'RWO'  ,
    'entao'       : 'RWO',
    'ENTAO'       : 'RWO',
    'senao'       : 'RWO',
    'SENAO'       : 'RWO',
    'fim'         : 'RWO',
    'FIM'         : 'RWO',
    'repita'      : 'RWO',
    'REPITA'      : 'RWO',
    'ate'         : 'RWO',
    'ATE'         : 'RWO',
    'inteiro'     : 'RWO',
    'INTEIRO'     : 'RWO',
    'flutuante'   : 'RWO',
    'FLUTUANTE'   : 'RWO',
    'leia'        : 'RWO',
    'LEIA'        : 'RWO',
    'escreva'     : 'RWO',
    'ESCREVA'     : 'RWO',
    'retorna'     : 'RWO',
    'RETORNA'     : 'RWO'
}
```

Isso porquê é complicado separar essas palavras por uma expressão regular comum, para tal tarefa utilizamos uma função 'coringa', é muito facil dos tokens da classe OID se confundirem com os de palavra reservada, por isso ao invez de criarmos uma expressão para OID e outra para RWO, iremos codificar ambas em na função do agrupamento OID.

Ah sim, eu esqueci de mencionar mais cedo mas além de strings, os tokens podem ser representados por funções que contenham uma expressão regular, bastando com que o nome da função seja igual a t_nomeTokenCategoria.

De volta para a função coringa, usaremos a t_OID para identificar os tokens de palavras reservadas e os identificadores.

```
def t_OID(t):
    r'_|([a-zA-Z])\w*_*\w*'
    t.type = reservedList.get(t.value,'OID')
    return t
```

Perceba que existe uma expressão regular dentro dessa função, o que ocorre é que essa expressão será utilizada para identificar todos os identificadores de variável(pois obrigatoriamente devem começar com letra ou underscore, depois podem conter qualquer sequencia de letras e/ou numeros e underscores), no entanto, se o texto do token for IDENTICO a algum texto presente na reserved List, ele terá o valor respectivo de tipo da reservedList, que no nosso caso sempre será RWO.

Mas agora que você já viu do pior, vamos voltar para as declarações das expressões regulares simples, as que usam variáveis com para representar os grupos de tokens.

```
t_CMT   =   r'{[^}]*'
t_SYB     = r'\+|-|:=|>=|<=|<>|:|\*|<|>|}|=|\(|\)|,|\[|\]|&&|\|\|'
t_FLT    = r'\d+\.\d+'
t_INT = r'\d+'
t_ignore   = ' \t\r\n'
```

Aqui temos variáveis com os nomes das categorias dos nossos tokens que restaram, dentro de cada r" está a expressão regular de cada token group.

A t_CMT contém uma expressão que ao encontrar um abre chaves, só para de casar quando encontrar o fecha chaves, por isso conseguimos pegar todos os comentários com o abre chaves, mas token fecha chaves vai ser da categoria dos simbolos.

A t_SYB é mediocremente simples, se trata apenas de um ou entre os possíveis simbolos da linguagem, nada mais. Além de que t_FLT e t_INT também são deveras modestas, utilizando dígito.dígito e somente dígito para casar.

Repare que existe o t_ignore, que não é uma categoria de token, como o nome sugere, são caracteres que devem ser ignorados pelo lexer.

Por fim, basta instanciar o lexer(entidade que leva em conta as definições acima descritas), informar a string de entrada, e chamar a função token() até que não haja mais um token a ser lido. No nosso caso, estaremos escrevendo os tokens no arquivo output.txt.

```
    lexer = lex.lex()
    f  = open('input.txt','r')
    input = f.read()
    f.close()
    lexer.input(input)

    file = open('output.txt','w');
```

```
    while True:
        tok = lexer.token()
        if not tok:
            break      # No more input
        file.write("<"+tok.type+","+tok.value+">\n")
    file.close()
```

E assim temos a essência do tokenizador, que mais adiante será testado, também abordaremos os autômatos referentes a cada expressão regular aqui exposta mais adiante, o software usado para desenhar os autômatos foi o JFLAP.

3.4. Testes

Dado o seguinte código:

```
inteiro: A[20]
inteiro busca(inteiro: n)
        inteiro: retorno
        inteiro: i
        retorno := 0{Comentario
  formoso}
        i := 0
        repita
                se A[i] = n
                        retorno := 1
                fim
                i := i + 1
        at   i = 20
        retorna(retorno)
fim
```

Após rodar o tokenizador, obtivemos a saída:

```
<RWO, inteiro> <SYB,:> <OID,A> <SYB,[> <INT,20>
<SYB,]> <RWO, inteiro> <OID, busca> <SYB,(> <RWO, inteiro>
<SYB,:> <OID,n> <SYB,)> <RWO, inteiro> <SYB,:> <OID, retorno>
<RWO, inteiro> <SYB,:> <OID,i> <OID, retorno> <SYB,:=> <INT,0>
<CMT,{ Comentario
  formoso>
<SYB,}>
<OID,i> <SYB,:=> <INT,0> <RWO, repita> <RWO, se> <OID,A> <SYB,[>
<OID,i> <SYB,]> <SYB,=> <OID,n> <OID, retorno> <SYB,:=> <INT,1>
<RWO, fim> <OID,i> <SYB,:=> <OID,i> <SYB,+> <INT,1> <RWO, at  >
<OID,i> <SYB,=> <INT,20> <RWO, retorna> <SYB,(> <OID, retorno>
<SYB,)> <RWO, fim> <RWO, inteiro> <OID, principal><SYB,(>
<SYB,)> <RWO, inteiro> <SYB,:> <OID,i> <OID,i> <SYB,:=> <INT,0>
<RWO, repita> <OID,A> <SYB,[> <OID,i> <SYB,]> <SYB,:=> <OID,i>
<OID,i> <SYB,:=> <OID,i> <SYB,+> <INT,1> <RWO, at  > <OID,i>
<SYB,=> <INT,20> <RWO, leia> <SYB,(> <OID,n> <SYB,)> <RWO, escreva>
```

<SYB,(> <OID, busca> <SYB,(> <OID, n> <SYB,) > <SYB,) > <OID, retorno>
<SYB,(> <INT,0> <SYB,) > <RWO, fim>

Acompanhando token a token, fica visível que obtivemos o resultado esperado, o comentario mesmo foi identificado ainda que houvesse em mais que uma linha.

3.5. Automatos das expressões regulares

Abaixo, seguem os autômatos das expressões regulares mencionadas neste documento, para compactação de espaço visual, utilizamos colchetes para sinalizar a presença do operador lógico OR(|). Acima estão os autômatos das expressões OID, RWO, CMT, SYB

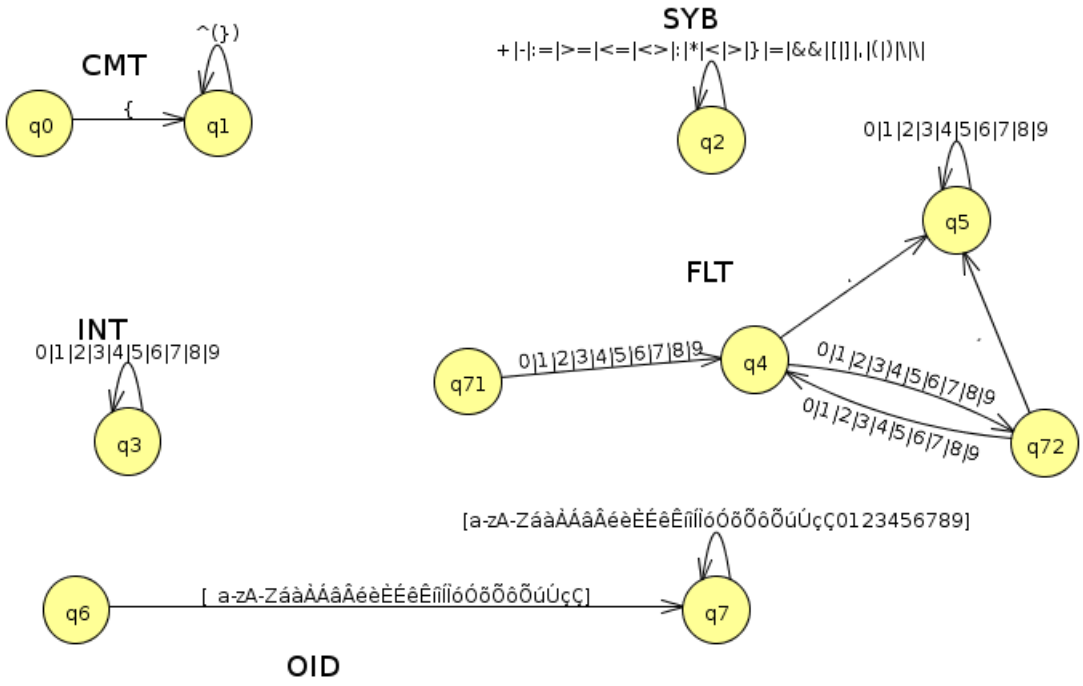

Figura 1. Autômatos das expressões OID, RWO, CMT, SYB, INT, FLT

e INT, todos finitos e determinísticos.

Seguindo, temos os autômatos das palavras reservadas. A aqui temos os autômatos finitos determinísticos das palavras reservadas do lexer.

4. Analise sintática

O objetivo dessa etapa é o processamento dos tokens da análise léxica, que resulta em uma validação deles e em uma árvore sintática abstrata.

4.1. Gramáticas livres de contexto e a BNF do T++

Nesse momento, nos deparamos em uma situação onde não é mais possível a utilização de somente expressões regulares, isso porquê não seria possível o controle de sequenciamento e recursão somente dessa maneira, por essa razão foi preciso a utilização de

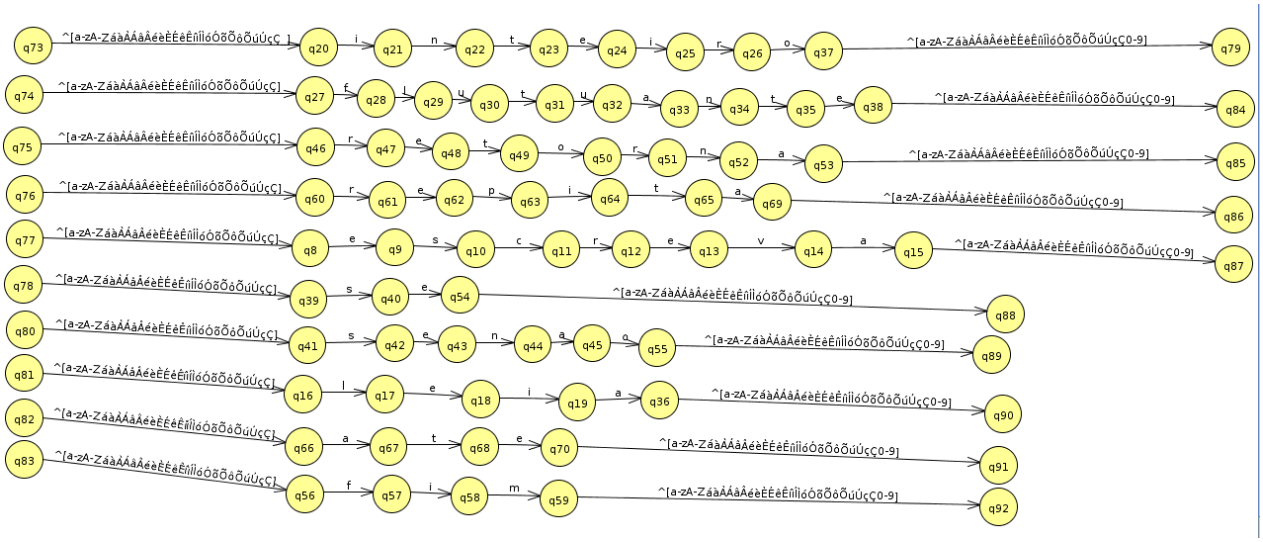

Figura 2. Autômatos das palavras reservadas

gramáticas livres de contexto para a construção da árvore sintática. A gramática utilizada está presente logo abaixo.

```
programa : lista_declaracoes

lista_declaracoes : lista_declaracoes declaracao
                  | declaracao

declaracao : declaracao_variaveis
                              | inicializacao_variaveis
                              | declaracao_funcao

declaracao_variaveis : tipo DOISPONTOS lista_variaveis

inicializacao_variaveis : atribuicao

lista_variaveis : lista_variaveis VIRGULA var
                | var

var : ID
    | ID indice

indice : indice LCOLCHETE expressao RCOLCHETE
       | LCOLCHETE expressao RCOLCHETE

tipo : INTEIRO
    | FLUTUANTE

declaracao_funcao : tipo cabecalho
```

```
                  | cabecalho

cabecalho : ID LPARENTHESYS lista_parametros RPARENTHESYS corpo FIM

lista_parametros : lista_parametros VIRGULA parametro
                 | parametro
                 | vazio

parametro : tipo DOISPONTOS ID
          | parametro LCOLCHETE RCOLCHETE

corpo : corpo acao
      | vazio

acao : expressao
     | declaracao_variaveis
     | se
     | repita
     | leia
     | escreva
     | retorna
     | error

se : SE expressao ENTAO corpo FIM
   | SE expressao ENTAO corpo SENAO corpo FIM

repita : REPITA corpo ATE expressao

atribuicao : var ATRIB expressao

leia : LEIA LPARENTHESYS var RPARENTHESYS

escreva : ESCREVA LPARENTHESYS expressao RPARENTHESYS

retorna : RETORNA LPARENTHESYS expressao RPARENTHESYS

expressao : expressao_logica
          | atribuicao

expressao_logica : expressao_simples
    | expressao_logica operador_logico expressao_simples

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_negacao fator

operador_relacional : LESSER
                    | GREATER
                    | EQUAL
                    | DIFF
                    | LESSEREQ
                    | GREATEREQ

operador_soma : PLUS
              | MINUS

operador_negacao : NEGATION

operador_logico : AND
                | OR

operador_multiplicacao : MULT
                       | DIV

fator : LPARENTHESYS expressao RPARENTHESYS
      | var
      | chamada_funcao
      | numero

numero : INT
       | FLOAT

chamada_funcao : ID LPARENTHESYS lista_argumentos RPARENTHESYS

lista_argumentos : lista_argumentos VIRGULA expressao
                 | expressao
                 | vazio

vazio :
```

4.2. O formato da Análise sintática realizado pelo PLY

O PLY.yacc utiliza uma técnica de parsing conhecida como LR-parsing ou shift-reduce parsing. Se trada de uma técnica de baixo para cima que tenta reconhecer o "lado direito

da mão"de várias regras gramaticais. Toda vez que o lado direito de uma regra é encontrado na entrada, a ação apropriada é invocada e os simbolos da gramática são substituidos pelo simbolo gramatical do lado esquerdo da declação.

O LR-parsing geralmente é implementado utilizando enviando simbolos gramaticas para uma pilha, adicionalmente olhando para a pilha e para o proximo token de entrada para ver se casou com alguma das regras gramaticais, a seção abaixo exemplifica como se dá o processo dado a gramática hipotética.

```
#Regras gramaticais - exemplo
Gramática                   Ação
--------------------        ----------------------------------
exp0 : exp1 + term           exp0.val = exp1.val + term.val
     | exp1 - term           exp0.val = exp1.val - term.val
     | term                  exp0.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)
       | ( exp )             factor.val = exp.val


#Processamento do parsing para a entrada '3 + 5 * (10 - 20)'
#OBS: Sh = shift / Rd = Reduce
St Symbol Stack       Input Tokens             Action
-- ------------------ ---------------------  ------------------
1                     3 + 5 * ( 10 - 20 )$   sh 3
2  3                   + 5 * ( 10 - 20 )$    Rd factor : NUMBER
3  factor              + 5 * ( 10 - 20 )$    Rd term   : factor
4  term                + 5 * ( 10 - 20 )$    Rd exp : term
5  exp                 + 5 * ( 10 - 20 )$    sh +
6  exp +                 5 * ( 10 - 20 )$    sh 5
7  exp + 5                 * ( 10 - 20 )$    Rd factor : NUMBER
8  exp + factor            * ( 10 - 20 )$    Rd term   : factor
9  exp + term              * ( 10 - 20 )$    sh *
10 exp + term *              ( 10 - 20 )$    sh (
11 exp + term * (              10 - 20 )$    sh 10
12 exp + term * ( 10              - 20 )$    Rd factor : NUMBER
13 exp + term * ( factor          - 20 )$    Rd term : factor
14 exp + term * ( term            - 20 )$    Rd exp : term
15 exp + term * ( exp             - 20 )$    sh -
16 exp + term * ( exp -             20 )$    sh 20
17 exp + term * ( exp - 20             )$    Rd factor : NUMBER
18 exp + term * ( exp - factor         )$    Rd term : factor
19 exp + term * ( exp - term           )$    Rd exp : exp - term
20 exp + term * ( exp                  )$    sh )
```

```
21 exp + term * ( exp )                    $    Rd factor : (exp)
22 exp + term * factor                     $    Rd term : term * factor
23 exp + term                              $    Rd exp : exp + term
24 exp                                     $    Rd exp
25                                         $    Success!
```

No parsing de uma expressão, um estado de máquina e o token corrente determinam qual será a próxima espressão, se o próximo token parecer com parte de uma regra gramatical válida, ele é geralmente enviado para a pilha, se o topo da pilha contém um valor válido de "lado direito"de uma regra gramatical, ele é reduzido e os simbolos são substituidos pelo simbolo do lado esquerdo, quando essa redução ocorre, a ação apropriada é disparada(se definida). Se o token de entrada não puder ir para a pilha e o topo não condiz com nenhuma das regras gramaticais, um erro sintático ocorreu e o parser deve partir para um passo de recuperação, o parse só é bem sucedido quanado alcançar o estado onde a pilha de simbolos está vazio e não existem mais tokens de entrada.

O PLY também utiliza uma técnica chamada LALR(Look-ahead LR parser), se trata de uma versão simplificada do LR parser tradicional.

4.3. Implementação e o PLY.yacc

A primeira tarefa da análise sintática foi a escrita da gramática de acordo com as regras do PLY.yacc, cada gramatica deve estar dentro de um bloco de comentários, e cada bloco deve estar dentro de uma função, cada função tem acesso ao objeto do PLY responsável pela construção da árvore, e em cada chamada deve-se decidir que "nó"será pendurado na árvore quando o caso de redução esperado ocorrer, como segue:

```
def p_operador_soma(self,p):
        '''
            operador_soma : PLUS
                          | MINUS
        '''
        p[0] = NkTree([p[1]],'operador_soma')
```

Por enquanto, ignore a existência da classe NkTree, apenas leve em conta que no PLY.yacc, toda vez que ocorrer uma redução de acordo com o lado direito da regra, o que p[0] receber no corpo da função será o "nó"pendurado na árvore sintática.

A documentação dp PLY instrui a utilização de uma estrutura de dados auxiliar para a montagem da árvore, nossa implementação a chamou de NkTree, se trata de uma árvore simples, o código da sua estrutura está logo abaixo:

```
    class NkTree:

    def __init__(self, children, type, leaf=None):
        self.type = type
        self.children = children
        self.leaf = leaf
        self.label = ""
```

```
    def __str__(self):
        return self.type

    def getNextCount(self):
        self.count += 1
        return str(self.count)

    def setLabelId(self,value):
        self.label = self.type+str(value)
```

Toda vez que uma redução for ocorrer, umas das funções que contém as gramaticas será chamada, e sempre a saída irá ser uma NkTree, que recebe como parametros as próximas regras a serem executadas em cadeia e o nome da regra em questão.

Agora você deve estar se perguntando, mas que valores são armazenados no child de cada NkTree em cada chamada de função? A resposta é simples, são passados todas as entidades do lado direito das regras gramaticais como filhos de cada nó NkTree, para todas as regras definidas nas funções, seguem alguns exemplos.

```
def p_declaracao(self, p):
    '''
        declaracao : declaracao_variaveis
                   | inicializacao_variaveis
                   | declaracao_funcao
    '''
    p[0] = NkTree([p[1]],'declaracao')

def  p_declaracao_variaveis(self,p):
    '''
        declaracao_variaveis : tipo DOISPONTOS lista_variaveis
    '''
    p[0] = NkTree([p[1],p[2],p[3]],'declaracao_variaveis')

def p_inicializacao_variaveis(self,p):
    '''
        inicializacao_variaveis : atribuicao
    '''
    p[0] = NkTree([p[1]],'inicializacao_variaveis')

def p_lista_variaveis(self,p):
    '''
        lista_variaveis : lista_variaveis VIRGULA var
                        | var
    '''
    if len(p)==4:
        p[0] = NkTree([p[1],p[2],p[3]],'lista_variaveis')
    elif len(p)==2:
        p[0] = NkTree([p[1]],'lista_variaveis')
```

Note que o atributo children do NkTree é um simples array com os statements do lado direito das regras, e que alguams condicionais sobre atributos do objeto em questão foram usadas para definir quais parametros deveriam ser passados como filhos, dependendo da situação.

Outro detalhe importante é que a precedência entre os tokens do lexer teve que ser configurada, quanto mais próximo do topo(sequencialmente falando), menor a precedência, e elementos no mesmo grupo compartilham da mesma precedência. Como multiplicação e divisão tem prioridade sobre soma e subtração, eles estão em um conjunto posterior ao das somas.

```
self.precedence = (
('left','EQUAL','GREATEREQ','GREATER','LESSEREQ','LESSER'),
('left','PLUS','MINUS'),
('left','MULT','DIV')
)
```

4.4. A árvore sintática

Por fim, uma vez que todas as funções das regras gramaticais foram definidas, e as atribuições referentes as instâncias da classe NkTree foram atribuidas a cada nó da árvore sintática, obtivemos a árvore sintática no PLY, no entanto ainda se fazia necessária uma forma de representação gráfica da mesma, para tal optamos pela utilização da biblioteca treelib.

A biblioteca requer um objeto Tree() para funcionar, então tudo o que tivemos que fazer foi uma função que monta esse objeto como um espelho do NkTree populado pelo PLY, e depois chamar o método show da biblioteca, segue o código.

```
def mountTree(self,nktree):
  actuals = []
  nexts = nktree.children
  nktree.setLabelId(self.getNextCount())
  self.tree.create_node(nktree.type,nktree.label)
  actuals.append(nktree)

  while len(nexts)!=0:
      nexts = []
      parentName=""
      for node in actuals:
        if(type(node) is NkTree):
          for next in node.children:
            if(type(next) is NkTree):
              next.setLabelId(self.getNextCount())
              self.tree.create_node(next.type,next.label,node.label)
            else:
              if(next is None):
                self.tree.create_node(next,count,node.label)
            nexts.append(next)
      actuals = nexts
```

```
def print(self, **kwargs):
  self.mountTree(self.parsed)
  self.tree.show(**kwargs)
```

Segue um exemplo de impressão da Árvore Sintática abstrata gerada, invocando a função print acima descrita.

```
Input:
inteiro soma(inteiro: a,inteiro :b)
    retorna (a+b)
fim
```

A figura 3 mostra a árvore gerada para essa entrada trivial.

5. Referências

Seguem as referências utilizadas nesse trabalho:

[1]LOUDEN, Kenneth C. Compiladores: princípios e práticas. São Paulo, SP: Thomson, c2004. xiv, 569 p. ISBN 8522104220

Referências

Figura 3. Arvore Gerada