# Analisador sintático[1]

Alexandre Yuji Kajihara[1]

*Universidade Tecnológica Federal do Paraná – UTFPR*

*DACOM – Departamento Acâdemico de Computação do Curso de Bacharelado em Ciência da Computação*
*Campo Mourão, Paraná, Brasil*

[1]alexandre.ykz@gmail.com

## Resumo

*Para desenvolver um compilador é necessário várias etapas, e uma delas é o analisador sintático. Para se ter o analisador sintático o único requisito é ter a análise léxica, já que os tokens que são retornados na etapa anterior são utilizados nesse passo, em que cada token retornado verificamos se existe a compatibilidade com a estrutura aceita pela linguagem T++. Na análise sintática, definimos a notação BNF (Backus-Naur Form* ou *Backus Normal Form) e suas ações, em que isso é necessário verificar se o arquivo de entrada é compatível com o formato do código-fonte aceito pela linguagem T++. Com a BNF definida, começamos a desenvolver a análise sintática em que utilizamos o GNU Bison para essa etapa, e o Flex em que foram necessárias algumas mudanças na parte da análise léxica para obter sucesso nessa etapa, e alguns conhecimentos da linguagem de programação C, para que se possa representar a árvore sintática. Por fim, com o caminho traçado pelo programa, e com a BNF, montamos nossa árvore sinática, e o resultado é que conseguimos com sucesso montar a nossa árvore sintática, em quase todos os casos. O relatório está dividido em oito partes que são: introdução, objetivo, fundamentação, materiais, procedimentos e resultados, discussão dos resultados, conclusões e referências.*

## 1. Introdução

Para montar o analisador sintático foi necessário saber a estrutura aceita pela linguagem T++, o conhecimento visto na etapa anterior também foi utilizado, pois foram necessários algumas mudanças para realizar a análise sintática. Nessa etapa, por exemplo garante que só podemos fazer três operações fora de funções, que seria definir variáveis, atribuir valores e ter funções. Com todos esses conhecimentos iremos conseguir construir a árvore sintática.

## 2. Objetivo

O objetivo de desenvolver o analisador sintático é conseguir montar a árvore sintática, verificando se o conteúdo do código-fonte de entrada é aceito pelas notações BNF (*Backus-Naur Form* ou *Backus Normal Form*) da linguagem T++.

## 3. Fundamentação

Os fundamentos necessários para implementar o analisador léxico foi ter a BNF (*Backus-Naur Form* ou *Backus Normal Form*) para verificar se o conteúdo do código-fonte de entrada está adequado com a estrutura de um código-fonte da linguagem T++. Para validar isso utilizamos o GNU Bison, que seria um compilador de compilador, compatível com o YACC e trabalha em conjunto com o Flex. O Bison aceita arquivos no formato .Y, então criamos um arquivo chamado parser.y na qual o conteúdo dele é os tokens que foram retornados na etapa anterior mais as notações da BNF. Além disso, foi necessário relembrar alguns conceitos do Flex e da linguagem de programação C. No Flex foram necessárias algumas mudanças na análise léxica para se adequar a análise sintática, e a linguagem de programação C nos auxiliou a montar a árvore sintática. Vale a pena ressaltar, que tanto no uso do GNU Bison, quanto no uso do Flex, ambos são de fáceis manipulações.

## 4. Materiais

Os materiais utilizados foram um *laptop* com o Sistema Operacional Ubuntu 16.10 64 bits com a configuração Intel Core i7-6500U, 16 GB de memória RAM (*Random Access Memory*), um compilador para linguagem C que foi o GCC na versão 6.2.0, o LEX na versão 2.6.1, gedit na versão 3.22.0, alguns códigos em T++ e o GNU Bison versão 3.0.4 para a análise sintática. Com o gedit que é um editor de texto escrevemos alguns códigos na linguagem T++, o scanner.l que é o que é aceito pelo LEX e o parser.y que é arquivo de formato compatível com o GNU

1 Trabalho desenvolvido para a disciplina de BCC36B – Compiladores

Bison. Com o GCC, LEX e GNU Bison, criamos nosso analisador sintático que aceita a linguagem T++.

## 5. Procedimentos e resultados

Para começar o analisador sintático foi definido qual é a estrutura da nossa linguagem T++, através de notações de BNF (*Backus-Naur Form* ou *Backus Normal Form*). Em que podemos ter fora de funções declarações de variáveis, e atribuição de valores que seriam consideradas variáveis globais. Já em declarações de funções podemos ter funções com três tipos de retorno, que seria o ponto flutuante, inteiro e *void* (ou seja, não retorna nada), uma função deve ter um nome, pode ou não ter uma lista de parâmetros, e deve se ter um "corpo" nessa função. Esse "corpo" seria composto de declarações de variáveis, atribuições, expressões de condição, laços de repetição, chamada de funções (que podem ser usadas funções de leitura, escrita, ou as definidas pelo usuário), podemos ter três tipos de valores que seriam um valor de ponto flutuante (números decimais), números inteiros (números sem decimais) e números exponenciais. Todas essas características que um código-fonte da linguagem T++, foram definidas através de notações BNF, como podemos ver uma parte dela na Figura 1.

```
//start
programa:
        lista_declaracoes {fprintf(output, "lista_declaracoes\n");}
        ;

lista_declaracoes:
        lista_declaracoes declaracao {fprintf(output, "lista_declaracoes declara
        | declaracao {fprintf(output, "declaracao\n");}
        ;

declaracao:
        declaracao_variaveis {fprintf(output, "declaracao_variaveis\n");}
        | inicializacao_variaveis {fprintf(output, "inicializacao_variaveis\n");
```

*Figura 1: parte das notações BNF, definidades no arquivo parser.y.*

Com as notações BNF definidas, colocamos ações em cada uma dessas notações, que podem ser visualizadas entre colchetes. Decidimos colocar essa ações para sabermos qual foi o trajeto realizado, e as notações de BNF que compõem a linguagem foram fundamental para montarmos a árvore sintática. Isso porque, para cada caminho dessa sequência verificamos se era compatível com a nossa BNF, se sim poderiamos adicionar o nó na árvore e prosseguir, caso contrário, teriamos que voltar nos nós predecessores da árvore até encontrar uma notação que fosse válida, podemos ver uma parte da saída gerada na Figura 2.

```
ID: 103
Pai: var
Filho: IDENTIFICADOR indice

ID: 104
Pai: indice
Filho: ABRECOLCHETE expressao FECHACOLCHETE

ID: 105
Pai: expressao
Filho: expressao_simples

ID: 106
Pai: expressao_simples
Filho: expressao_aditiva

ID: 107
Pai: expressao_aditiva
Filho: expressao_multiplicativa

ID: 108
Pai: expressao_multiplicativa
Filho: expressao_unaria

ID: 109
Pai: expressao_unaria
Filho: fator

ID: 110
Pai: fator
Filho: numero

ID: 111
Pai: numero
Filho: NUMEROINTEIRO

ID: 101
Pai: tipo
Filho: TIPOINTEIRO
```

*Figura 2: árvore gerada pelo nossa analisador sintático.*

Na Figura 2 podemos verificar que temos "var" (que seria variável), tem como seu filho sendo "IDENTIFICADOR indice". Após isso, vimos que o "indice" que foi filho de "var" no nível passado, agora vai ser pai de "ABRECOLCHETE expressao FECHACOLCHETE", e assim por diante. Uma coisa que vale a pena ressaltar que o "ID" que podemos verificar na Figura 2, indica que quando temos valores de "ID" iguais, significa que aquele "ID" pertence ao "ID" de mesmo valor. Isso fica mais claro na Figura 3, em que a saída com o valor de "ID" de 101, é referente ao "ID" de 101 que está presente na Figura 3, ou seja, aquele tipo que está presente na Figura 3, com o "ID" 101 é do tipo inteiro. Em códigos-fontes que temos poucas linhas, imprimimos com a mesma cor para ficar fácil a visualização de "ID" iguais.

Após tudo isso, conseguimos realizar a árvore sintática em quase todos os casos. Além disso, também conseguimos também detectar erros de sintaxe nos códigos-fonte, em que emitimos um aviso para avisar que há um erro de sintaxe no código-fonte.

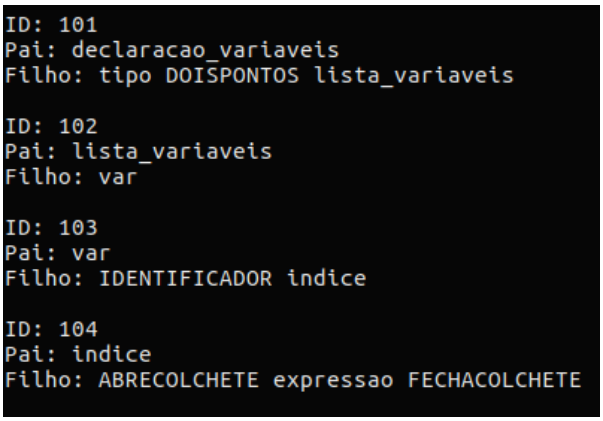


*Figura 3: outra parte da árvore sintática.*

## 6. Discussão dos resultados

Após realizar todas esses procedimentos, para a construção dessa segunda etapa de um compilador, vimos que quase alcançamos o resultado desejado, que era a construção de árvore sintática. Isso porque, em casos que código-fonte de entrada tinha poucas linhas, conseguimos gerar uma árvore sintática sem problema nenhum. Porém quando o código-fonte era grande os nós se perdiam em relação ao pai deles. Já em casos de erro da sintaxe do código-fonte detectamos o erro, mostramos uma mensagem de erro, e não construímos a árvore.

## 7. Conclusão

Concluimos, que essa segunda etapa para se construir um compilador é muito relevante e essencial. Isso porque, é o analisador sintático que vai verificar se o código-fonte de entrada respeita o formato em que a nossa linguagem T++ aceita. Conseguir a árvore sintática é muito importante nesse passo, já que para etapa que vem que é a análise semântica, consiga por exemplo, verificar se aquela variável presente no "corpo" de uma função ela foi declarada, se a função é do tipo void, então não podemos ter uma expressão de retorno. Tivemos alguns problemas logo no começo, de que seria como era estruturado um arquivo .Y, onde teria que definir as notações BNF (*Backus-Naur Form* ou *Backus Normal Form*), porém vendo alguns exemplos conseguimos resolver esse problema. Outro problema que apareceu durante as declarações dos BNF, em que tivemos alguns *warnings que* foram gerados, em que não sabiamos as causas, porém no final conseguimos resolver, em que seria que algumas regras faltavam. Com relação a árvore sintática, tivemos erros em código-fontes mais longos, pois quando tentávamos achar o pai daquele nó, havia uma perda pois não sabiamos se o pai era tal nó ou se era um ancestral dele. Entretanto, todas essas dificuldades citadas acima foram superadas, com exceção da árvore e consequentemente quase conseguimos alcançar nosso objetivo.

## 8. Referências

[1] LOUDEN, K.C. Compiladores – Princípios e Práticas. Ed. Thomson Pioneira, 2004.zEd. Thomson Pioneira, 2004.z