Analisador Sintático

Alexandre Yuji Kajihara[1]

[1]DACOM -Departamento Acadêmico de Computação do Curso de Bacharelado em Ciência da Computação – Universidade Tecnológica Federal do Paraná (UTFPR)
Campo Mourão – PR – Brazil

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, alguns conhecimentos da linguagem de programação C, para que se possa representar a árvore sintática, e conhecimento no formato de arquivo aceito pelo Graphviz dot para que possamos representar a nossa árvore em forma de imagem. Por fim, conseguimos montar a nossa árvore sintática e o resultado é que testamos em oito arquivo de entradas e conseguimos obter sucesso em todos eles. 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++. 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. Uma ferramenta que nos auxiliou a visualizar a árvore sintática foi a Graphviz dot, em que representamos a nossa árvore em forma de imagem.

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 Bison. Com o GCC, LEX e GNU Bison, criamos nosso analisador sintático que aceita a linguagem T++. Além disso, para facilitar a visualização da árvore sintática utilizamos a ferramenta Graphviz dot na versão 2.38.0.

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 e declarações de variáveis 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.

```
389 fator:
390         ABREPARENTESES expressao FECHAPARENTESES { $$ = criaNo("fator", 1, $2);
391         | var { $$ = criaNo("fator", 1, $1); }
392         | chamada_funcao { $$ = criaNo("fator", 1, $1); }
393         | numero { $$ = criaNo("fator", 1, $1); }
394         ;
395
396 numero:
397         NUMEROINTEIRO
398                 {
399                         auxiliar = criaNo($1, 0);
400                         $$ = criaNo("numero", 1, auxiliar);
401                 }
402         | NUMEROFLUTUANTE
403                 {
404                         auxiliar = criaNo($1, 0);
405                         $$ = criaNo("numero", 1, auxiliar);
406                 }
407         | EXPONENCIAL
408                 {
409                         auxiliar = criaNo($1, 0);
410                         $$ = criaNo("numero", 1, auxiliar);
411                 }
412         ;
413
```

Figura 1: notações BNF e ações para cada uma das notações.

Com as notações BNF definidas, colocamos ações em cada uma dessas notações, que podem ser visualizadas entre colchetes. Em cada uma dessas notações criamos um nó que no final, irá se juntar para formar a árvore sintática. Para visualizar a nossa árvore pode ser feita de duas maneiras, a primeira olhando para o terminal ou pela janela que irá abrir. No terminal iremos imprimir primeiro o pai e depois disso o nó filho ou os nós filhos, e quando tivermos a saída impressa de verde representa que aquele nó é um nó folha, isso pode ser vistao na Figura 2.

```
xandao@danoninho: ~/CompiladorEmC
ÁRVORE SINTÁTICA
programa
lista_declaracoes

lista_declaracoes
declaracao_funcao

declaracao_funcao
inteiro cabecalho

inteiro

cabecalho
fatorial lista_parametros corpo

fatorial
```

Figura 2: árvore sintática sendo exibida no terminal.

Como relatamos anteriormente, que após o término da execução do programa irá também se abrir um janela, essa janela irá mostrar uma imagem de como ficou a nossa árvore sintática, isso foi feita com a ferramenta Graphviz dot. Para realizar isso percorremos todos os nós da árvore, e colocamos os seus pais e filhos da árvore em um arquivo .dot, enfim a nossa árvore pode ser visualizada na Figura 3.

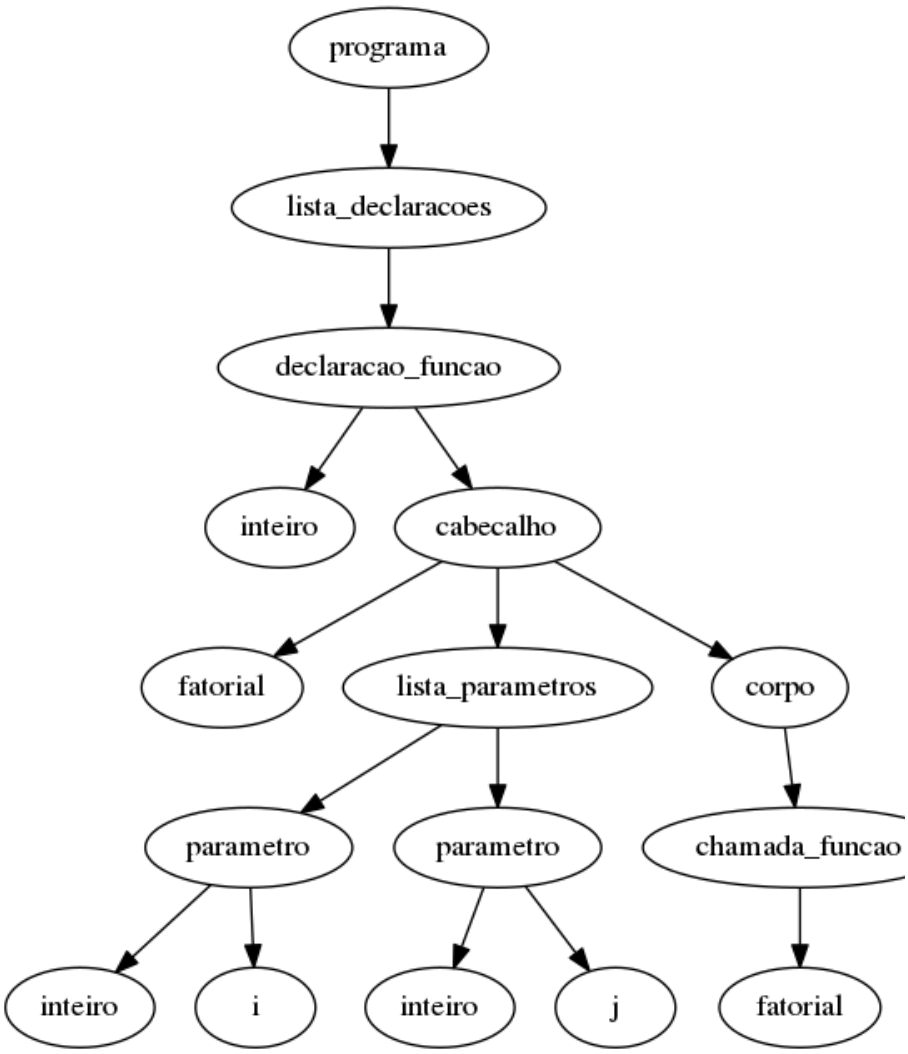

Figura 3: árvore sintática gerada pela ferramenta Graphviz dot.

Após tudo isso, realizamos teste em oito arquivos de entrada .tpp, em que cada um deles primeiro montamos essas árvores em papéis de caderno e depois, comparamos cada um deles com a saída do terminal e com a saída gerada pela ferramenta Graphviz dot, e vale a pena ressaltar que em todos eles conseguimos sucesso.

6. Discussão dos resultados

Após realizar todas esses procedimentos, para a construção dessa segunda etapa de um compilador, vimos que alcançamos o resultado desejado com todos nossos arquivos .tpp, que era a construção de árvore sintática. Vale a pena ressaltar, que quando há um erro de sintaxe no arquivo de entrada, é avisado e não se constrói a árvore sintática. Além disso, geramos um arquivo de LOG, isso caso ele caia em algum caso que possa acarretar em erros na geração da árvore sintática, porém se nenhum erro acontecer ele não irá gerar e nem avisar que foi gerado, porém se gerar emitirá um aviso avisando que foi criado um arquivo de LOG.

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, refizemos várias vezes até chegar nessa árvore, mas enfim conseguimos gerar a nossa árvore sintática sem problema nenhum. Entretanto, todas essas dificuldades citadas acima foram superadas e consequentemente conseguimos alcançar nosso objetivo.

References

Louden, K.C. Compiladores – Princípios e Práticas. Ed. Thomson Pioneira, 2004.