Trabalho 2 – Analisador Sintático

1964SCC – Compiladores

Victor Fernandes Gardini
Profa. Dra. Renata Spolon Lobato
Instituto de Biociências, Letras e Ciências Exatas, Unesp – Universidade Estadual Paulista
São José do Rio Preto - SP.
e-mail: victor.fernandes@unesp.br

12 de fevereiro de 2021

Sumário

1. Introdução

Este documento é dividido em cinco seções principais: Manual de Uso, Novas Implementações, Gramática Livre de Contexto, Exemplos e Conclusão. Na seção Manual de Uso (Seção 2), é explicitado como compilar e utilizar o analisador sintático projetado, bem como os programas necessários para sua execução. Já em Novas Implementações (Seção 3), é descrito as novas alterações promovidas para a utilização do analisador léxico nesta continuação do trabalho anterior. Na seção Gramática livre de contexto (Seção 4), são mostradas a gramática e as principais regras que geram a linguagem elaborada para este trabalho. Na seção 5, são dados exemplos de códigos da linguagem que é aceita pelo analisador sintático elaborado. Por fim, as conclusões sobre este trabalho estão presentes na Seção 6.

2. Manual de Uso

Nesta seção será explicado e demonstrado como compilar e executar os analisadores léxico e sintático, também será explicitado um manual de uso do analisador projetado para este trabalho. Por fim, recomenda-se que seja utilizado um sistema operacional Linux para a execução dos passos a seguir.

2.1 GNU Bison

Na construção do projeto foi utilizado o GNU Bison, comumente conhecido como Bison, trata-se de um gerador de analisadores que faz parte do Projeto GNU. O Bison é responsável por ler as especificações de uma linguagem livre de contexto, notificando ambiguidades detectadas durante a análise e, por fim, gera um analisador (neste caso, na linguagem C) capaz de ler uma sequência de tokens previamente definidos e decidir se a sequência está conforme a sintaxe especificada pela gramática projetada.

Para este projeto foi utilizada a versão 3.5.1 do Bison sob o sistema operacional Ubuntu 20.04 LTS. Temos que, para realizar a compilação do código do analisador sintático, é necessário executar o comando *bison -d sintatico.y* e então um arquivo em c, chamado *sintatico.tab.c* será criado, este que, por sua vez, contém em seu código as regras para análise sintática.

2.2 Flex

Temos que o analisador sintático utiliza um analisador léxico para separar o código-fonte em tokens, essa função será desempenhada pelo Flex. Para este projeto, a versão 2.6.4 do Flex foi utilizada no sistema operacional Ubuntu 20.04 LTS. Por fim, para realizar a compilação do código do analisador léxico, é necessário executar o comando *lex lexico.l* e então, um novo arquivo em c, chamado *lex.yy.c* será criado este que, por sua vez, contém em seu código a análise léxica feita no arquivo fonte.

2.3 Compilar

Por fim, para gerar o analisador executável é utilizado o compilador gcc na sua versão 9.3.0, ao executar o comando *gcc lex.yy.c sintatico.tab.c -ll -o analisador.exe*, é obtido o analisador, que pode ser executado com o *comando ./analisador exemplos/teste1.txt.*

Todo o processo é simplificado com a utilização do *Makefile*, basta executar o comando *make* no terminal do computador para gerar o executável.

3. Novas Implementações

Para que o analisador sintático conseguisse determinar os tokens encontrados pelo analisador léxico, foram realizadas novas implementações, quando comparado com o último trabalho entregue no dia 27/10/2020. Nesta seção, será destacadas as mudanças realizadas no analisador léxico desde o último trabalho entregue da disciplina.

A tabela a seguir destaca, em vermelho, os tokens já implementadas e, em verde, os novos que são aceitos pelo programa:

Expressão regular	Observação
OPERADORES_LOGICOS	Novos operadores lógicos: “and”, “or” e “not”.
VÍRGULA	A vírgula passou a ser reconhecida como um token.
OPERADORES_RELACIONAIS	Novos operadores: “==”, “>”, “<”, “>=”, “<=” e “!=”.
ABRE_CHAVES	O caractere “{” é reconhecido como um token.
FECHA_CHAVES	O caractere “{” também é reconhecido um token.
WHILE	Novo token reconhecido para uma estrutura de repetição
INICIO_PROGRAMA	Foram implementados novos tokens para sinalizar o início e fim do programa principal.
FIM_PROGRAMA	
NUMERO_INTEIRO	O número inteiro agora é reconhecido como um token.
NUMERO REAL	O número real agora é reconhecido como um token.
FOR	Estes operadores eram e continuam sendo reconhecidos pelo programa.
DEF	
RETURN	
TIPOS_VARIAVEIS	
IF	
ELSE	
ABRE_PARENTESES	
FECHA_PARENTESES	
ATRIBUIÇÃO	
DIGITO	
ALFABETO	
ID	
STRING	
ESPAÇO	
OPERADORES ARITMÉTICOS	
PONTO_E_VÍRGULA	

Tabela 1: Todos os Tokens Reconhecidos pelo Programa.

A adição dos novos tokens foi importante para a conclusão do analisador sintático construído neste trabalho, os mesmos podem ser observados nos exemplos da Seção 5.

4. Gramática Livre de Contexto

Nesta seção, encontra-se as especificações das expressões utilizadas na gramática utilizada do analisador sintático. Para um maior entendimento e visualização, as expressões estão definidas separadas pela sua finalidade. É importante destacar que as regras de produção estão em letras minúsculas, enquanto que os tokens estão em maiúsculo. Por fim, destaca-se que a cadeia vazia está representada por "%empty".

4.1 Regra Principal e Comandos

Todas as regras irão partir de duas das mais importantes, uma delas é a regra principal que pode ser observada na Figura 1. Observa-se a regra comandos delimitada por dois tokens que sinalizam o início e fim do programa.

```
/* Definição da estrutura inicial do programa */
principal: INICIO_PROGRAMA comandos FIM_PROGRAMA;
```

Figura 1: Definição da Regra "principal".

A próxima regra é a comandos, conforme pode-se observar na Figura 2, observa-se que nesta regra estão definidas as principais estruturas de recursão do programa.

```
/* Definição dos comandos possíveis */
comandos: decl_var comandos
    | ID atribuicao PONTO_E_VIRGULA comandos
    | condicional comandos
    | funcao comandos
    | loop comandos
    | %empty;
```

Figura 2: Definição da Regra "comandos".

4.2 Regra de Funções

Observa-se, na Figura 3, a principal estrutura que permite a definição das funções pelo analisador. A regra função é a de mais alto nível, ela permite ao usuário começar a definir uma função, começando token DEF, ou chamar uma função, com diferentes tokens como STRING e cont_parametros (sucessão de parâmetros) ou, a mesclagem das opções anteriores.

```
/* Declaração das funções */
funcao: DEF ID ABRE_PARENTESES lista_parametros FECHA_PARENTESES bloco_comando_funcao
    | ID ABRE_PARENTESES STRING FECHA_PARENTESES PONTO_E_VIRGULA
    | ID ABRE_PARENTESES cont_parametros FECHA_PARENTESES PONTO_E_VIRGULA
    | ID ABRE_PARENTESES ID FECHA_PARENTESES PONTO_E_VIRGULA
    | ID ABRE_PARENTESES STRING VIRGULA ID FECHA_PARENTESES PONTO_E_VIRGULA;

/* Definição da lista de parametros de uma função */
lista_parametros: TIPOS_VARIAVEIS ID cont_parametros
    | %empty;

/* Definição da continuação dos parametros de uma função*/
cont_parametros: VIRGULA TIPOS_VARIAVEIS ID cont_parametros
    | VIRGULA ID cont_parametros
    | ID cont_parametros
    | %empty;
```

Figura 3: Principais Regras da Estrutura funcao.

4.3 Declaração de Variáveis e Atribuições

Para viabilizar a declaração de variáveis no projeto, foi definida a estrutura da Figura 4. Para declarar uma variável é preciso que o token TIPOS_VARIAVEIS esteja sendo utilizado, após isso, um identificador ou uma atribuição seguido do token PONTO_E_VIRGULA.

```
/* Definição da estrutura de declaração das variáveis */
decl_var: TIPOS_VARIAVEIS ID PONTO_E_VIRGULA
    | TIPOS_VARIAVEIS ID atribuicao PONTO_E_VIRGULA;
```

Figura 4: Definição da regra “decl_var”.

Já na Figura 5, observa-se a regra que permite a atribuição com as variáveis e funções. É importante destacar que o token ATRIBUICAO é importante para a representação de uma atribuição.

```
/* Definição das estruturas de atribuicao de variáveis */
atribuicao: ATRIBUICAO expressao_arit
    | ATRIBUICAO ID ABRE_PARENTESES cont_parametros FECHA_PARENTESES
    | var_num ;
```

Figura 5: Definição da Regra “atribuição”.

4.4 Expressões Aritméticas

Para viabilizar o uso de expressões aritméticas é definida a regra “expressão_arit”, conforme ilustrado na Figura 6. Observa-se que a regra está ligada com outras duas: “arit1” e “arit2”, isso deve-se à possível necessidade de implementar uma expressão com operadores aritméticos definidos no token OPERADORES_ARITMETICOS.

```
/* Definição das regras de expressão aritméticas */
expressao_arit: arit1 arit2 ;
arit1: ABRE_PARENTESES expressao_arit FECHA_PARENTESES arit2
    | var_num arit2 ;
arit2: OPERADORES_ARITMETICOS arit1
    | %empty ;
```

Figura 6: Definição das Regras que permitem o uso de Expressões Aritméticas.

4.5 Expressões Lógicas

Para combinar as expressões aritméticas com operadores lógicos, muito comum em operações que utilizam estruturas como “if”, “else”, “while”, entre outros, foi definida a regra ilustrada na Figura 7. Observa-se que a regra está acompanhada de outras duas regras: “log1” e “log2”.

```
/* Definição das regras de expressão lógica */
expressao_logica: expressao_arit log1 ;
log1: OPERADORES_RELACIONAIS expressao_arit log2
    | %empty;
log2: OPERADORES_LOGICOS expressao_logica
    | %empty;
```

Figura 7: Definição da Regra “expressão_logica”.

4.6 Blocos de Comandos

A regra “bloco_comando” foi definida como a regra de “comandos”, já apresentada, delimitada pelos tokens ABRE_CHAVES e FECHA_CHAVES, respectivamente, como forma de definir um bloco de código. Essa estrutura é essencial para controlar escopo de variáveis em algumas linguagens de programação, o resultado da implementação pode ser conferido na Figura 8.

```
/* Definição do bloco de comandos delimitados pelas chaves */
bloco_comando: ABRE_CHAVES comandos FECHA_CHAVES;
```

Figura 8: Definição da Regra “bloco_comando”.

Somente para o caso de uma função é permitido o uso do token RETURN e seus parâmetros (como *strings* ou variáveis), então foi definido uma regra somente para esse caso. Essa regra é utilizada somente pela regra que viabiliza funções já apresentada anteriormente e o resultado da sua implementação está ilustrado na Figura 9.

```
/* Definição do bloco de comando das funções */
bloco_comando_funcao: ABRE_CHAVES comandos FECHA_CHAVES
    | ABRE_CHAVES comandos RETURN var_num PONTO_E_VIRGULA FECHA_CHAVES
    | ABRE_CHAVES comandos RETURN STRING PONTO_E_VIRGULA FECHA_CHAVES;
```

Figura 9: Definição da regra “bloco_comando_funcao”.

4.7 Condições

Para viabilizar o uso de estruturas condicionais que possuem a presença dos tokens IF e ELSE, foi implementada a regra “condicional”. Conforme ilustrado na Figura 10, pode-se notar a presença das regras de expressões lógicas e, por fim, a regra “cond1” que é responsável pelo uso do token ELSE.

```
/* Declaração das estruturas condicionais */
condicional: IF ABRE_PARENTESES expressao_logica FECHA_PARENTESES bloco_comando cond1;
cond1: ELSE bloco_comando
    | %empty ;
```

Figura 10: Definição da regra “condicional” e “cond1”.

4.8 Laços de Repetição

Por fim, foi implementada a regra “loop” que permite a utilização de laços de repetição de linguagens de programação. Nesta gramática, é permitido o uso do for e while, conforme observado na Figura 11. Também vale destacar, a possibilidade de utilizar a regra “expressao_logica”, o que é muito útil em uma linguagem de programação.

```
/* Declaração das estruturas de repetição */
loop: FOR ABRE_PARENTESES decl_var expressao_logica PONTO_E_VIRGULA ID atribuicao FECHA_PARENTESES bloco_comando
    | WHILE ABRE_PARENTESES expressao_logica FECHA_PARENTESES bloco_comando
```

Figura 11: Regra que Define Laços de Repetição.

5. Exemplos

Nesta seção encontra-se alguns exemplos projetados referentes à a linguagem projetada para este trabalho. Cada exemplo aborda um aspecto desde o considerado como inicial, o primeiro contato com a linguagem, até o mais avançado, que contempla a criação de funções. Todos os exemplos foram submetidos ao analisador sintático que reportou que foram aceitos e se enquadram com a linguagem projetada, o resultado pode ser observado à seguir.

5.1 Olá Mundo!

Trata-se de um código simples, comum entre os iniciantes em uma linguagem de programação. Neste caso, utiliza-se a função *display* para simular a escrita do texto entre parênteses no terminal principal.

```
1  %INICIO_PROGRAMA%
2      // A função display representa
3      // uma escrita no terminal principal.
4      display("Olá, mundo!");
5  %FIM_PROGRAMA%
```

Figura 12: Exemplo básico, olá mundo.

5.2 Declarando Variáveis

Um exemplo um ainda simples. Observa-se a utilização das palavras reservadas *int*, *float* e *double*, ressaltando que a palavra *char* também poderia ser utilizada, para a definição de qual valor será armazenado nas variáveis.

```
1  %INICIO_PROGRAMA%
2      // Programa exemplo de
3      // como declarar variáveis
4      int var1;
5      var1 = funcao2();
6      int var2 = 5;
7      float var3;
8      double teste;
9  %FIM_PROGRAMA%
```

Figura 13: Exemplo de Como Declarar Variáveis.

5.3 Estruturas de Condições e de Repetição

A linguagem permite utilizar estruturas de condição, como *if* e *else*, e estruturas de repetição, como *while* e *for*, conforme destacado anteriormente. A Figura 14 exemplifica o uso dessas estruturas além de combinar o que já foi visto de exemplos nessa seção.

```
%INICIO_PROGRAMA%
    // Programa exemplo utilizando estruturas de
    // repetição e de condição
    int var1 = 1;
    int var2 = 5;
    int var3;
    float var4;

    if (var1 >= var2){
        display("O valor de var1 é maior que o de var2.");
    } else {
        display("O valor de var1 é menor que o de var2.");
    }

    // Exemplo utilizando o laço for
    for (int i=0; i!=10; i=i+1) {
        if (i < 5) {
            var3 = var1 / i;
        }
        else {
            var2 = var1 / i;
        }
    }

    // Exemplo utilizando o laço while
    while (var3 < 10) {
        var3 = var3 + 1;
        if (var1 != 4 or var2 <= 8) {
            teste = var1 + (var2 * var3);
        } else {
            teste = (var1 - var3) / var2;
        }
    }
%FIM_PROGRAMA%
```

Figura 14: Exemplo de como utilizar estruturas de controle e de repetição.

5.4 Funções

Por fim, temos que o analisador também suporta a utilização de funções com ou sem parâmetros, conforme destacado na Figura 15.

```
%INICIO_PROGRAMA%
    // Programa exemplo de como
    // declarar funções

    // Exempo de função com parâmetros
    def funcao1 (int var2, float var3){
        return "retornando uma string";
    }

    // Exemplo de função sem parâmetros
    def funcao2 (){
        i = 1;
        return i;
    }

    // Armazenando o valor retornando
    // em uma variável
    var1 = funcao2();
%FIM_PROGRAMA%
```

Figura 15: Exemplo de como declarar funções.

6. Conclusão

Conclui-se que, por meio deste trabalho, foi possível compreender o funcionamento de um analisador sintático, exercitando as habilidades obtidas nas disciplinas 1959SCC-Linguagens Formais e Autômatos e 1964SCC-Compiladores, ministradas pela prof. Dr. Renata Spolon Lobato para projetar uma linguagem de programação e um compilador. Ademais, com a utilização do Flex e do Bison, foi possível realizar a conexão do analisador léxico e sintático de uma mesma linguagem, percebendo sua importância de tais ferramentas ao projetar uma linguagem. Por fim, conclui-se que foi possível construir um analisador sintático e léxico com base na gramática planejada para este trabalho.