Manual de Uso
e
Explicação da Gramática

1. Introdução

Este documento está dividido em três seções principais: (2)Gramática, (3)Manual de Uso e (4)Exemplos. Respectivamente teremos a descrição da linguagem utilizada para a construção deste trabalho, citando seus estados, tokens entre outros, depois será a explicação de como utilizar a linguagem por meio de simuladores léxicos e sintáticos, finalizando com exemplos de códigos possíveis na linguagem.

2. Gramática

Nesta seção serão especificadas todas as expressões, estágios e tokens utilizados dentro da linguagem, tanto do arquivo Léxico quanto do Sintático.

Expressões Regulares e Tokens

As tabelas a seguir nesta parte da seção descrevem as produções feitas pelas expressões no arquivo léxico da linguagem, gerando os tokens necessários para o analisador léxico.

Expressão Regular	TOKENS	Descrição
DIGITO	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9	Dígitos possíveis para criação de números na linguagem.
ALFABETO	a | b | c | d | e | f | g | h |…| y | z | A | B | C | D |…| Y | Z	Letras minúsculas e maiúsculas para a criação de palavras/variáveis na linguagem.
ESPACO	" " | \t	Expressão que reconhece os espaços dentro da linguagem.
MAIN	main	Token do início da função Main
IF	if	Início da função de caso If da linguagem.
ELSE	else	Início da função de caso Else da linguagem.
FOR	for	Token reconhecido para um laço de repetição For.
WHILE	while	Token reconhecido para um laço de repetição While.
DO	do	Token reconhecido para um laço de repetição Do…While.
PRINTF	printf	Início da função Printf da linguagem.
RETURN	return	Início da função Return da linguagem.
ASPAS	‘	Token de aspas simples para a escrita do Printf.
ABRE_PARENTESIS	(	Token de reconhecimento de início de parênteses.

FECHA_PARENTESIS	)	Token de reconhecimento de fechamento de parênteses.
ABRE_CHAVE	{	Token de reconhecimento de início de chaves.
FECHA_CHAVE	}	Token de reconhecimento de fechamento de chaves.
SEPARADOR	;	Token de separação entre linhas de códigos na linguagem.
VIRGULA	,	Token de vírgula utilizado dentro do Printf para escrita ou para separação das sentenças dentro do comando.
OPERADOR	+ | - | * | /	Operadores possíveis dos cálculos matemáticos dentro da linguagem.
MAIOR_MENOR	> | <	Sinais de maior e menor dentro da linguagem para comparações na linguagem.
IGUAL	=	Sinal de igual usado na linguagem.
PONTUACAO	! | ?	Sinais de pontuação utilizados no comando Printf na linguagem.
QUEBRA_LINHA	\n	Expressão que reconhece as quebra de linha dentro da linguagem.

Existem na linguagem junções de expressões regulares que formam outras ou que são específicas e não utilizam expressões criadas, são as seguintes:

Expressão Regular	**TOKENS**	**Descrição**
TIPO	“int”,”float”,”double”,”bool”,”void”, “long int”	São os tokens que declaram qual é o tipo da variável/função.
IDENTIFICADOR	{ALFABETO}+({ALFABETO}| {DIGITO})*	Tokens de criação de variáveis/funções. Onde obrigatoriamente seu primeiro caractere deve ser uma letra.
INTEIRO	{DIGITO}+	Este é a expressão que cria os números inteiros, onde são usados somente sequencia de dígitos.
REAL	{DIGITO}{DIGITO}*"."{DIGITO} {DIGITO}*	A expressão cria os tokens de números reais, que a sequência de N dígitos, seguido de um ponto, ., e uma outra sequência de M dígitos.

CITACAO	"%"{ALFABETO}+({ALFABETO}|{DIGITO})*	Esta é a expressão de criação de citação de variáveis dentro do comando printf.
ITERACAO	"+""+"|"-""-"	São os tokens de soma ou subtração de valor 1 em uma variável.
COMP	{IGUAL}{IGUAL}|"!"{IGUAL}	Tokens de comparação igual ou diferente entre variáveis e números ou outras variáveis em comandos de casos IF ou de laços de repetição.
MAIOR_MENOR	{MAIOR_MENOR}|{MAIOR_MENOR}{IGUAL}	A expressão cria os tokens de comparação maior, menor, maior igual ou menor igual entre variáveis e números ou outras variáveis em comandos de casos IF ou de laços de repetição.

OBS: A expressão MAIOR_MENOR, aparece em ambas tabelas pois a sua cadeia pode possuir além de < ou >, uma junção com IGUAL, =.

Regras e Produções

Nesta parte da seção serão mostradas as especificações das expressões utilizadas na gramática utilizada do analisador sintático e suas funcionalidades. As regras de produção são nomeadas com a sua primeira letra sendo maiúscula e o restante minúscula, enquanto os tokens só possuem letras maiúsculas.

A regra de partida é a Programa, no qual são feitas as definições de funções, se houverem, a declaração da Main junto com os seus comandos, finalizando com o seu Return, tendo logo após seu termino a descrição das funções definidas no começo. Qualquer programa fora dessa sequência é considerado um erro.
Programa: Declara_Funcao MAIN ABRE_PARENTESIS FECHA_PARENTESIS Chaves_R Escreve_Funcao|error{yyerror("", linhas);};

É a regra de declaração de uma função na linguagem, onde deve ser informado o tipo que a função vai retornar, seu nome e se possui ou não variáveis de parâmetro.
Declara_Funcao: TIPO IDENTIFICADOR ABRE_PARENTESIS Lista_Var_F FECHA_PARENTESIS SEPARADOR Declara_Funcao | ;

Lista_Var_F é a regra que produz a sequência de variáveis que são usadas como parâmetros na declaração e definição nas funções.
Lista_Var_F: TIPO IDENTIFICADOR VIRGULA Lista_Var_F | TIPO IDENTIFICADOR | ;

Chaves e Chaves_R são as regras que abrem as funções e os comandos de laços e de casos, onde em todos esses casos existem comandos dentro das mesmas. A diferença entre eles é possuir ou não o comando Return em seu final. Foi necessário a criação de duas pois comandos dentro de funções não possuem return, como While.
Chaves: ABRE_CHAVE Comandos FECHA_CHAVE;
Chaves_R: ABRE_CHAVE Comandos Retornar FECHA_CHAVE;

Essa regra gera um Comando Comandos ou nada, assim as funções podem ter ou não códigos dentro de suas definições. Logo teremos nenhum, um ou vários comandos nas funções.
Comandos: Comando Comandos |;

A regra Comando é onde são escolhidos todas as opções possíveis dentro das funções criadas, tanto main quanto outras. Um Comando pode ser uma Declaração, Atribuicao, Print, Caso, Laco ou Funcao, qualquer coisa fora isso é um erro. Os comandos que não possuem separador no final é porque ou já possuem dentro de seu Comando ou não precisam. A descrição desses comandos serão feitas logo mais.
Comando: Declaracao SEPARADOR| Atribuicao SEPARADOR| Print SEPARADOR| Caso | Laco | Funcao SEPARADOR | error {yyerror("",linhas);};

Declaração é o comando de declarar uma ou muitas variáveis dentro de uma função.
Declaracao: TIPO Lista_Var;

É a regra de citar todas as variáveis que estão sendo criadas, podendo ter ou não uma atribuição de valor a ela.
Lista_Var: IDENTIFICADOR Atribui_valor;

Essa regra atribui um valor a uma variável e declarar mais uma variável do mesmo tipo sendo separadas por uma vírgula, somente atribuir um valor, somente listar mais variáveis do mesmo tipo ou nada.
Atribui_valor: IGUAL Valor VIRGULA Lista_Var| IGUAL Valor | VIRGULA Lista_Var |;

Valor é a regra de produção dos valores possíveis dentro da linguagem. Onde ele pode ser uma variável, no caso IDENTIFICADOR, um número, NUM, que pode ser do conjunto dos reais ou inteiros ou até uma chamada de função, Funcao, que retorna um valor.
Valor: IDENTIFICADOR | Num | Funcao;
Num: INTEIRO | REAL ;

O comando de chamativa de uma função, que pode retornar ou não um valor e pode possuir variáveis de parâmetro ou não.
Funcao: IDENTIFICADOR ABRE_PARENTESIS Variaveis FECHA_PARENTESIS;

Variaveis é a definição dos parâmetros que serão enviados na chamativa da função, podendo ter um, vários ou nenhum.
Variaveis: Valor VIRGULA Variaveis | Valor | ;

Este comando atribui a uma variável já criada um valor, que pode ser uma operação matemática, OP_MAT, um Valor, que foi citado anteriormente, ou então somar ou subtrair um da variável.
Atribuicao: IDENTIFICADOR IGUAL OpMat | IDENTIFICADOR IGUAL Valor | IDENTIFICADOR ITERACAO;

OpMat é uma operação matemática com valores de Valor, que podem ou não estar entre parênteses.
OpMat: ABRE_PARENTESIS Valor OpMat_P FECHA_PARENTESIS | Valor OpMat_P;
OpMat_P: OPERADOR OpMat | OPERADOR Valor;

O comando de Print da linguagem, onde são escritos textos, que possuem letras, números, pontuações, citações de variáveis. Pode ter ou não texto dentro de um Print, juntamente é possível ter mais de uma sentença escrita dentro, além de também a escrita das variáveis que foram citadas.
Print: PRINTF ABRE_PARENTESIS Sentenca FECHA_PARENTESIS;

Sentenças é tudo que é escrito dentro de aspas simples, ASPAS, onde são escritas Palavras, e após a aspa é possível declarar quais variáveis foram citadas pela sentença.
Sentenca: ASPAS Palavra ASPAS Citar Sentenca | ;

Produção de texto dentro do Printf. Pode ser sinais de pontuação, PONTUACAO, vírgulas, VIRGULA, citação de variáveis, CITACAO, valores da regra Valor ou então nenhuma delas resultando em um Printf vazio.
Palavra: Valor Palavra | PONTUACAO Palavra | VIRGULA Palavra | CITACAO Palavra |;

Citar é a produção para gerar a variável, IDENTIFICADOR, citada dentro do Printf que pode ou não ter. Além disso, possuindo uma citação de variável pode ser que ela seja a última escrita antes de fechar os parênteses, ou talvez exista uma outra sentença após, necessitando de uma vírgula.
Citar: VIRGULA IDENTIFICADOR | VIRGULA IDENTIFICADOR VIRGULA |;

Caso é a produção do IF, onde ocorre uma comparação entre duas produções de Valor, podendo possuir ou não um caso Else logo após sua fecha de chaves.
Caso: IF ABRE_PARENTESIS Valor Comparacao Valor FECHA_PARENTESIS Chaves Caso_Else;

Comparação é onde escolhido qual tipo de comparação é feita entre uma variável, IDENTIFICADOR, e um Valor. Podendo ser maior, menor, maior ou igual, menor ou igual, igual ou diferente.
Comparacao: COMP | MAIOR_MENOR;

Comando que produz o caso Else após o caso If, que pode ou não existir. Onde possui também Comandos dentro de suas chaves.
Caso_Else: ELSE Chaves | ;

Comando que escolhe um laço de repetição, podendo ser For, While ou Do … While.
Laco: While_L | Do_L | For_L;

Produção do laço de repetição Do … While, onde se obtém DO, comandos entre suas chaves e, por fim, um While, na qual possui o caso que os comandos se repetem.
Do_L: DO Chaves While_D SEPARADOR;

Definição de While, que pode ser tanto usado no laço While quanto no Do … While. É o token WHILE e a variável, IDENTIFICADOR, que é comparada a um Valor.
While_D: WHILE ABRE_PARENTESIS IDENTIFICADOR Comparacao Valor FECHA_PARENTESIS ;

Produção do laço de repetição While que é a combinação de While_D junto com a abertura de chaves, os comandos e a fechamento das chaves.
While_L: While_D Chaves;

For_L é a produção do laço de repetição For. A sequência de produções é o próprio FOR onde dentro de seus parênteses ocorre a criação de uma variável que recebe um valor tendo um ponto e virgula, SEPARADOR, para finalizar essa sentença, vem seguido de um caso para o laço ocorrer entre a variável criada e um Valor e um outro SEPARADOR, por fim é declarado qual o tipo de iteração, ITERACAO, ocorrerá, fechando assim os parênteses e abrindo as chaves e tendo os comandos necessários dentro.
For_L: FOR ABRE_PARENTESIS TIPO IDENTIFICADOR IGUAL Valor SEPARADOR IDENTIFICADOR MAIOR_MENOR Valor SEPARADOR IDENTIFICADOR ITERACAO FECHA_PARENTESIS Chaves;

Comando para gerar ou não um retorno com valor ou vazio de uma função.
Retornar: RETURN Valor SEPARADOR | RETURN SEPARADOR |;

Descrição da função fora da Main na linguagem. Onde seu tipo, seu nome, possuindo ou não parâmetros e então a abertura das chaves, os comandos que ocorrem dentro da função e o então o seu Return se possuir.
Escreve_Funcao: TIPO IDENTIFICADOR ABRE_PARENTESIS Lista_Var_F FECHA_PARENTESIS Chaves_R Escreve_Funcao | ;

3. Manual de Uso

Aqui será mostrado como compilar e executar os arquivos dos analisadores léxico e sintático para a análise de arquivos pela linguagem criada neste trabalho. A execução e criação dos arquivos foi feita em Linux, o sistema operacional recomendado pelos desenvolvedores, logo os passos a seguir são feitos neste S.O.

Flex
O analisador léxico utilizado foi o Flex, no qual são declaradas os tokens possíveis na linguagem, ou seja, os valores terminais válidos. Isso é feito por meio de código onde são definidas os valores terminais possíveis e logo após as expressões possíveis que eles podem gerar, como por exemplo é definido a regra DIGITO, que produz os números entre 0 até 9, já na seção de cadeias possíveis possui a sequência {DIGITO}+ { return (INTEIRO);}, na qual é a sequência INTEIROS, que reconhece os qualquer número nesse formato, como por exemplo 01, 236, 10000, etc. Após isso é possível implementar ao seu final comandos de verificação de erros dentre outros.

Bison
A escolha para o analisador sintático foi o GNU Bison, mais conhecido como somente Bison, no qual consegue analisar os arquivos de entrada e verificar se correspondem ou não sintaticamente a linguagem. Neste analisador são declarados primeiramente os valores terminais possíveis em sua linguagem, que devem estar contidos dentro do arquivo Flex utilizado pois é de lá que são tirados os tokens possíveis em sua análise. Logo após temos a declaração das cadeias possíveis de produção, onde são gerados os estágios da linguagem. Para exemplificar ambas partições temos a produção Valor: IDENTIFICADOR | Num, onde um valor utilizado em alguma função pode ser tanto uma variável quando um número, sendo uma variável o Flex analisa se a sequência de caracteres é possível dentro da regra IDENTIFICADOR, ela sendo a análise continua, se não é um erro. Em sua última partição é onde ficam os códigos de chamada de análise e report de erros no analisador.

Arquivos Necessários

Os arquivos necessários para que seja possível a leitura de arquivos de entrada pela linguagem são dois: O arquivo léxico fonte contendo todos os tokens, os valores terminais, da linguagem e o arquivo sintático fonte no qual possui todas as cadeias possíveis, sequência de valores terminais e não-terminais, e os estágios da linguagem. Com estes arquivos é possível criar os arquivos que serão lidos pelo pelo compilador gcc. Os arquivos devem estar em formatos .l, arquivo flex e .y, arquivo Bison, respectivamente.

Compilação

Possuindo estes dois arquivos a compilação pode começar. Primeiramente deve ser iniciar o Bison com o arquivo-fonte sintático, .y, para criar um arquivo-fonte da linguagem C, o .tab.c, e também é gerado um arquivo biblioteca que faz a ligação entre o léxico e o sintático, que é o .tab.h. Isso é feito pelo código no terminal: *bison -d nome_arquivo.y* .
Logo após isso é iniciado o Flex com o seu arquivo-fonte léxico, .l, para que o arquivo chamado lex.yy.c, no qual é um arquivo-fonte em linguagem c que usaremos para gerar um executável. Isso é feito pelo código no terminal: *flex nome_arquivo.l* .
Tendo os três arquivos resultantes da execução dos simuladores, utilizamos o compilador gcc para criar o executável de ambos analisadores, para no final juntar ambos para fazer o analisador sintático desejado. Isso é feito pela seguinte sequência de comandos:
gcc -c lex.yy.c -o comp.flex.o
gcc -c fonte.tab.c -o comp.y.o
gcc -o comp comp.flex.o comp.y.o -lfl -lm
Com isso obtemos todos os arquivos possíveis para a leitura dos arquivos de entrada da linguagem, onde essa entrada é feita por meio do comando *./comp arquivo* .

Para facilitar a compilação desses arquivos foi feito a criação de um Script, que ao executá-lo no terminal executa esses passos para facilitar a utilização do funcionário. As figuras a seguir mostra a execução do Script e os aquivos resultantes desta execução. Vale ressaltar que uma execução sem o Script não resultará no arquivo .output, pois é um comando não necessário para a execução dos compiladores, mas interessante de se possuir, pois ele mostra todos os estágios, reduções e movimentos da linguagem atual. Além disso será necessário mudar no arquivo Flex o *analisador.h* para *fonte.tab.h*, porque ele não será renomeado automaticamente sem ser pelo Script.

```
italo@Ubuntin:~$ cd 'Área de Trabalho'/'Compiladores'/'Projetos'/'Projeto2'/'Italo'
italo@Ubuntin:~/Área de Trabalho/Compiladores/Projetos/Projeto2/Italo$ chmod +x exe.sh
italo@Ubuntin:~/Área de Trabalho/Compiladores/Projetos/Projeto2/Italo$ ./exe.sh
Iniciando o BISON
Criando Arquivo de Estados da Linguagem
Renomeando arquivos
Iniciando FLEX
Renomeando arquivos
Compilando arquivos
Para executar o compilador digite: ./comp [nome do arquivo a ser testado]
italo@Ubuntin:~/Área de Trabalho/Compiladores/Projetos/Projeto2/Italo$
```

Figura 1: Execução do Script para compilar os arquivos Flex e Bison

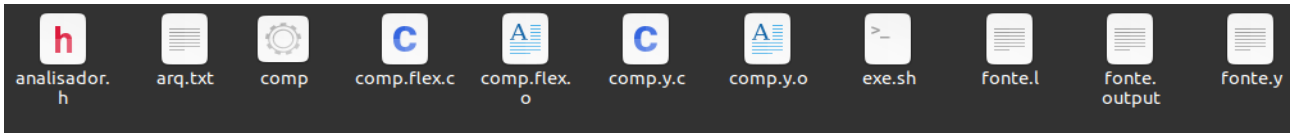

Figura 2: Arquivos resultantes após a execução do Script

4. Exemplos

Nesta seção serão mostrados exemplos de programas possíveis na linguagem projetada. Cada exemplo contem um aspecto implementado na linguagem, indo desde um programa iniciante, com a escrita do “Olá mundo”, até um mais complexo com chamada de funções, loops e casos de condições.

Exemplo ‘Olá Mundo’

Código simples de mostrar na tela a mensagem “Ola Mundo”, mostrando a funcionalidade de escrita da linguagem por meio do *printf*.

```
main(){
  printf('Ola Mundo');
}
```

Figura 3: Exemplo de Código "Olá Mundo" na Linguagem.

```
italo@Ubuntin:~/Área de Trabalho/Compiladores/Projetos/Projeto2/Italo$ ./comp Ola_Mundo.txt
Análise concluída com sucesso
```

Figura 4: Leitura do Código "Olá Mundo" pelo Bison.

Exemplo de Casos de Condições

Este exemplo mostra a utilização de caso de condição dentro do programa por meio de *if* e *else*. Mostra também como são criadas as variáveis e como são alocados valores a elas, além de também mostrar como são feitas contas matemáticas, com ou sem a utilização de parenteses.

```
main(){
 int x = 2, y = 3;
 x = 3;
 if (x != 2){
   x = x * 2;
 }
 else{
   y = (3/6 + (5*x));
 }
}
```

Figura 5: Exemplo de Código com Declarações e Atribuições de Variáveis e Casos de Condição na Linguagem.

```
italo@Ubuntin:~/Área de Trabalho/Compiladores/Projetos/Projeto2/Italo$ ./comp Caso_IF.txt
Análise concluída com sucesso
```

Figura 6: Leitura do Código de Casos de Condição pelo Bison.

Exemplo de Laços de Repetição

O exemplo a seguir demonstra como é feito laços de repetição, os loops, dentro da linguagem. Possuem três possibilidades para laços: *For*, *Do…While* e *While*. Cada um desses laços é possível escrever dentro deles comandos, como chamada de funções, outros laços de repetição, alocação de valores, etc.

```
main(){
 int x = 2, y = 3;
 for(int i = 0; i<y; i++){
   x = x * i;
 }
 do{
   y++;
 }while(y != 10);
 while(x != y){
   x--;
 }
}
```

Figura 7: Exemplo de Código com Loops na Linguagem

```
italo@Ubuntin:~/Área de Trabalho/Compiladores/Projetos/Projeto2/Italo$ ./comp Casos_Loop.txt
Análise concluída com sucesso
```

Figura 8: Leitura do Código de Casos de Loop pelo Bison.

Exemplo com Funções fora da Main

O exemplo demonstra como são feitas as declarações, chamadas e definidas de funções de um programa na linguagem. As funções são declaradas no início do arquivo antes da função *main*, podem ser chamadas dentro de outras funções ou da própria *main*, por que a chamada está dentro da regra *Comandos* da linguagem e são definidas após a escrita da função *main.*

```
int soma(int v1, int v2);
void ola();

main(){
 int x = 2;
 float y;
 double w, z = 2.847;
 w = soma(x , y);
 ola();
}

int soma(int v1, int v2){
 int aux;
 aux = v1 + v2;

 return aux;
}

void ola(){
 printf('Ola mundo');
}
```

Figura 9: Exemplo de Código com Declarações, Chamadas e Escrita de Funções na Linguagem

```
italo@Ubuntin:~/Área de Trabalho/Compiladores/Projetos/Projeto2/Italo$ ./comp Caso_Func.txt
Análise concluída com sucesso
```

Figura 10: Leitura do Código de Funções pelo Bison.

Exemplo de Erro

Por fim temos um exemplo que demonstra erros possíveis dentro da linguagem, não respeitando a sequência sintática criada para a linguagem. Neste exemplo em especial temos: Erro de alocação de valor em uma variável por não possuir um ‘;’ no seu final (x = 2); Erro de alocação de valor, pois não possui uma variável para receber o resultado da soma, além de também não possuir um ‘;’ no seu final(1+2); E por fim um erro no *printf*, onde temos uma palavra/variável fora das aspas, que necessitaria de uma virgula e estar ou não dentro de um novo par de aspas(printf(‘Ola mundo!’AAA).

```
int soma(int v1, int v2);
void ola();

main(){
 int x = 2;
 float y;
 double w, z = 2.847;
 w = soma(x , y);
 ola();
 x = 2
}

int soma(int v1, int v2){
 int aux;
 aux = v1 + v2;
 1 + 2
 return aux;
}

void ola(){
 printf('Ola mundo!'AAA);
}
```

Figura 11: Exemplo de Código que possui Erros Sintáticos

```
italo@Ubuntin:~/Área de Trabalho/Compiladores/Projetos/Projeto2/Italo$ ./comp Caso_Func.txt
Erro sintático
O erro aparece próximo à linha 11
Erro sintático
O erro aparece próximo à linha 16
O erro aparece próximo à linha 16
O erro aparece próximo à linha 16
O erro aparece próximo à linha 16
Erro sintático
O erro aparece próximo à linha 21
O erro aparece próximo à linha 21
O erro aparece próximo à linha 21
O erro aparece próximo à linha 21
Análise com erros
Total de erros encontrados: 3
```

Figura 12: Erros encontrados do arquivo lido aparecendo no Terminal