Universidade Federal de Viçosa

Campus Florestal

Trabalho Prático
Especificação e Implementação da linguagem:

Alunos:

João Victor Magalhães Souza	3483
Lucas Ranieri Oliveira Martins	3479

Disciplina: CCF 441 - Compiladores.
Professor: Dr. Daniel Mendes Barbosa.

Nome e Origem da Linguagem

Buscando contextualizar áreas do mundo real com o objetivo de encontrar uma que fosse nosso objeto de estudo e motivação na construção da linguagem, deparamo-nos com o potencial da seara da música e seu importante papel em nossas vidas. Pensando nisso, propomos a construção deste compilador e da linguagem por ele reconhecida pensando em terminologias técnicas e populares da área musical. Dessa forma, pensamos que nomear a linguagem como "**Melodia",** podemos ter uma boa analogia com a construção de algoritmos com a programação, visto que uma melodia pode ser dada como uma construção sequencial de notas arranjadas harmonicamente e, um algoritmo, uma sequência de passos a serem seguidos.

Por fim, paralelamente aos fatos supracitados, pensamos que às vezes o aprendizado de programação pode ser não tão intuitivo, principalmente em alguns conceitos iniciais. Dessa forma, permitir a construção de código com termos musicais e, sendo assim, propiciando uma analogia clara do que está acontecendo, na nossa visão, pode ajudar e facilitar o processo de aprendizagem. Posteriormente nas seções seguintes mostraremos como pretendemos construir essas analogias e tornar o processo de programação bem intuitivo.

Tipos de Dados Primitivos

Para a representação dos tipos de dados primitivos, pensamos em uma abordagem em que há um crescimento do nível de magnitude à medida que a nota fica mais aguda.

- **do:** inteiro sem sinal;
- **re:** inteiro com sinal;
- **mi:** float8;
- **fa:** float16;
- **sol:** float32;
- **la:** boolean;
- **si:** string;
- **acorde:** array;
- **maestro:** ponteiro;
- **medley:** classe;
- **vazio:** ;

Comandos e Expressões Disponíveis

Nesta seção, iremos explicar um pouco sobre os comandos que nós julgamos necessários para a construção da linguagem. De antemão gostaríamos de salientar que alterações de formato, inserção ou deleção de comandos são totalmente possíveis durante a execução das etapas posteriores do projeto.

Comando	Definição	Explicação	Analogia em C
nota()	entrada de dados	Alusão ao músico tocar uma nota, só que nesse caso ele entrará com dados do teclado.	scanf()
show()	exibição de informações	Alusão ao show musical, que seria quando o músico mostra seu repertório.	printf()
refrão	laço de repetição	Ambos os comandos na música remetem à repetição de algo.	for
bis	laço de repetição		while
ensaio	comando condicional	O ensaio é considerado como um teste do show ou do que será feito.	if
improviso	comando condicional	Já o improviso é o contrário do ensaio, seria como tocar algo que não foi ensaiado.	else
{	abertura de um bloco	-	{
}	fechamento de um bloco	-	}

play()	definição de função	"Aperte o play", ou nesse caso, execute uma função.	int main ()
teste()	verifica o tipo de uma variável	Testando o som e os arranjos	typeof()
eco	retorno de função	Ecoar o som. Nesse caso, "ecoaremos" o resultado.	return
finale	pausa na execução de um bloco ou função	Encerramento	break
& | ~ > < == <= >= !=	E lógico OU lógico negação maior que menor que igual menor ou igual maior ou igual diferente	-	&& || ! > < == <= >= !=
+ - / * ** % //	soma subtração divisão multiplicação exponenciação resto da divisão divisão inteira	-	+ - / * ** %
=	atribuição simples	-	=
+= -= /= *=	atribuição composta	-	+= -+ /= *=
++ --	atribuição unária pós fixada	-	++ --

[]	referenciamento	-	[]
do() re() mi() fa() sol()	conversão explícita de tipos	-	(int) (double) (float) (char) ...
medley *TAD não implementado. Explicação na Parte 3 desta documentação.	criação de classe	Medley é a junção de vários trechos de música em uma só música. Nessa analogia, uma classe é uma estrutura composta de vários tipos de dados e várias funções.	struct
;	demarcador de fim de linha	-	;

Exemplos:

♪ **Definição de variáveis:**

```
do v1
re v2
do acorde nomeVariavel[2]= [v1,v2];
si nome
```

ID “acorde” LITERAL_RECEBE “[“ args “]”
nomeVariavel → [v1, v2]
ID “[“ expression “]” “=” expression
nomeVariavel.append(‘cole’)
nomeVariavel.pop()

♪ **Laço de repetição:**

```
refrao (do v1 = 0, v1 < v2, v1++) {
        v3++
}

bis (v1 < v2) {
        v3++
```

```
        v1++
    }
```

♪ **Comando condicional:**

```
    ensaio (v1 > 0) {
        v1++
    }
    improviso {
        v1--
    }
```

♪ **Definição e retorno de funções:**

```
    sol play nome(sol v1, sol v2) {
        sol v3 = v1+v2
        eco v3          }
```

♪ **Interrupção:**

```
    bis (v1<v2) {
        ensaio (v1==3) {
        finale
        }
    v1++
    }
```

♪ **Operadores lógicos:**

```
    True & False
    True | True
    ~(False) & ~(False)
```

♪ **Atribuição simples e composta:**

```
    re v1 = 0
    re v2 = 1
    v1 += v2 + 1
    re v3 = 2
    v3 *= v1
```

♪ **Referenciamento:**

```
    si nome = “João”
    si sobrenome = “Victor”
    si concatena = nome[0] + sobrenome[1]
```

♪ **Conversão explícita de tipos:**

```
    sol pi = 3.14
```

```
re re_pi
re_pi = re(pi)
```

Paradigmas de Programação

Visando herdar conceitos das linguagens C e Python, mantemos a forma e a essência imperativa que a linguagem C oferece, principalmente na finalidade sintática em termos de aprendizado que a linguagem tem a oferecer (faz o programador aprender o que ele está realmente fazendo). Por outro lado, também desejamos implementar o conceito de Orientação a Objetos, nos espelhando na linguagem Python, visto que é um conceito extremamente interessante em programação. Logo, nossa proposta é de uma linguagem **multiparadigma:** contendo o paradigma **imperativo** e o paradigma **orientado a objetos** como carros chefe.

Palavras-chave e palavras reservadas

Todas nossas palavras-chave serão reservadas. Para elas podemos pensar, por exemplo, nas próprias palavras para os tipos primitivos, que serão: **do, re, mi, fa, sol, la, si, acorde, maestro, medley**. Outras palavras reservadas serão: **acorde, refrao, bis, ensaio, improviso, arpeggio, play, finale**.

Gramática

NOME_VARIAVEL→ **[a-zA-Z_][a-zA-Z0-9_]***
DIGITO → **[0-9]**
do→ *DIGITO***+** {unsigned int}
re → (*"+"* | *"-"*) *DIGITO***+** {int}
FLOAT → (((+?) {DIGITO}) | ([\-] {DIGITO})) {DIGITO} * (\. {DIGITO}+) {float}
si → ".*"{string} ***Observação:** o ponto significa qualquer caractere.
la → **True** | **False** {booleano}
acorde → "[" INTERNO "]" {array}
maestro → **"*"** *NOME_VARIAVEL* {ponteiro}
NUMERO → (do | re | mi | fa | sol)
medley → **[a-zA-Z]+** { . } {classe}

TIPOS_PRIMITIVOS → vazio
| do
| re
| mi
| fa
| sol
| la
| si
| acorde
| maestro
| medley

NOME_TIPOS_PRIMITIVOS → *"***vazio"**
| "**do"**
| "**re"**
| "**mi"**
| "**fa"**
| "**sol"**
| "**la"**
| "**si"**
| "**acorde"**
| "**maestro"**
| **[a-zA-z]+** {o tipo de retorno sendo de um objeto de uma classe}

BLOCO → "{" *STATEMENTS* "}"
STATEMENTS → *STATEMENTS STATEMENT* | **ε**
STATEMENT → *EXPRESSAO*

| **ensaio** ‘(‘ *EXPRESSAO* ‘)’ *BLOCO*
| **ensaio** ‘(‘ *EXPRESSAO* ‘)’ *BLOCO* **improviso** ‘(‘ EXPRESSAO ‘)’ BLOCO
| **refrao** ‘**(**‘ EXPRESSAO ‘;’ EXPRESSAO ‘;’ EXPRESSAO ‘**)**’ BLOCO
| **bis** ‘(‘ EXPRESSAO ‘)’ BLOCO
| **teste** ’**(**‘ NOME_VARIAVEL ‘**)**’
| **finale**
| **do** ‘**(**‘ NOME_VARIAVEL ‘**)**’
| **re** ‘**(**‘ NOME_VARIAVEL ‘**)**’
| **mi** ‘**(**‘ NOME_VARIAVEL ‘**)**’
| **fa**‘**(**‘ NOME_VARIAVEL ‘**)**’
| **sol** ‘**(**‘ NOME_VARIAVEL ‘**)**’
| *ATRIBUICAO_SIMPLES*
| *ATRIBUICAO_COMPOSTA*
| NOTA
| *BLOCO*
| *RETORNO*

ATRIBUICAO → **do** ‘**(**‘ NOME_VARIAVEL ‘**)**’
| **re** ‘**(**‘ NOME_VARIAVEL ‘**)**’
| **mi** ‘**(**‘ NOME_VARIAVEL ‘**)**’
| **fa**‘**(**‘ NOME_VARIAVEL ‘**)**’
| **sol** ‘**(**‘ NOME_VARIAVEL ‘**)**’
| EXPRESSAO

ATRIBUICAO_SIMPLES → NOME_VARIAVEL “=” ATRIBUICAO
*ATRIBUICAO_COMPOSTA → NOME_VARIAVEL **”++”***
| *NOME_VARIAVEL **”--”***
| *NOME_VARIAVEL **”/=”** NOME_VARIAVEL*
| *NOME_VARIAVEL **”*=”** NOME_VARIAVEL*
| *NOME_VARIAVEL **“+=”** NOME_VARIAVEL*
| *NOME_VARIAVEL **“-=”** NOME_VARIAVEL*
|*NOME_VARIAVEL **”/=”** NUMERO*
| *NOME_VARIAVEL **”*=”** NUMERO*
| *NOME_VARIAVEL **“+=”** NUMERO*
| *NOME_VARIAVEL **“-=”** NUMERO*

*NOTA → **“nota”** “(“* INTERNO*”)”*
INTERNO → (NOME_VARIAVEL | TIPO_PRIMITIVO)

| INTERNO “,” (NOME_VARIAVEL | TIPO_PRIMITIVO)
| **ε**

EXPRESSAO → RELACIONAL | FUNCAO
FUNCAO → NOME_TIPOS_PRIMITIVOS “play” NOME_VARIAVEL '(' LISTA_ARGUMENTOS ')' BLOCO

LISTA_ARGUMENTOS → [*NOME_TIPOS_PRIMITIVOS*][*NOME_VARIAVEL*]
| *LISTA_ARGUMENTOS “,”* [*NOME_TIPOS_PRIMITIVOS*][*NOME_VARIAVEL*]
| **ε**

RETORNO → **“eco”** NOME_VARIAVEL
| “**eco”** *TIPOS_PRIMITIVOS*
| “**eco**” EXPRESSAO

RELACIONAL → RELACIONAL < ARITMETICO_SOMA_SUB
| *RELACIONAL <= ARITMETICO_SOMA_SUB*
| *RELACIONAL > ARITMETICO_SOMA_SUB*
| *RELACIONAL >= ARITMETICO_SOMA_SUB*
| *RELACIONAL != ARITMETICO_SOMA_SUB*
| *RELACIONAL & ARITMETICO_SOMA_SUB*
| *RELACIONAL | ARITMETICO_SOMA_SUB*
| *RELACIONAL ~ ARITMETICO_SOMA_SUB*
| *ARITMETICO_SOMA_SUB*

***Observação:** abaixo separamos as operações aritméticas com o viés de promover a precedência entre as operações. Por exemplo: a exponenciação, radiciação e resto da divisão têm precedência sobre a multiplicação e divisão e estas últimas, por sua vez, têm precedência na soma e subtração.

ARITMETICO_SOMA_SUB→ ARITMETICO_SOMA_SUB + ARITMETICO_MULT_DIV v
| *ARITMETICO_SOMA_SUB - ARITMETICO_MULT_DIV*
| *ARITMETICO_MULT_DIV*

*ARITMETICO_MULT_DIV → ARITMETICO_MULT_DIV * ARITMETICO_EXP_MOD*
| *ARITMETICO_MULT_DIV / ARITMETICO_EXP_MOD*
| *ARITMETICO_MULT_DIV // ARITMETICO_EXP_MOD*
| *ARITMETICO_EXP_MOD*

ARITMETICO_EXP_MOD → ARITMETICO_EXP_MOD % TERMO
| *ARITMETICO_EXP_MOD ** TERMO*
| *TERMO*

TERMO → (EXPRESSAO)
| NUMERO
| NOME_VARIAVEL

LEX

Enfrentamos alguns problemas em algumas Definições Regulares no LEX. Os problemas estavam circunscritos nas estruturas que recebiam parâmetros como funções, classes, entre outras. Na nossa forma de manter uma estrutura de parâmetros correta, utilizamos uma espécie de recursão para garantir que a quantidade de vírgulas fosse igual à quantidade de parâmetros - 1. Entretanto, o LEX apresentou um erro da espécie: “O comando é muito complexo para ser tratato”. Dessa forma, implementamos uma abordagem mais “simples” para essa primeira etapa e pretendemos tratar esse empecilho nas abordagens posteriores.

Para compilar o nosso analisador, basta executar os seguintes comandos via terminal:

1. *flex lex1.l*
2. *gcc lex.yy.c*
3. *./a.out < entrada1.txt > saida2.txt*

Entrada de dados(“*entrada1.txt*”):

```
do a
re b
mi c
fa d
la e = True
do a = 2+2
la h = 2>2
la k = ~True|False
do play funcao1(){}
re play funcao2(){}
sol play funcao3(){}
mi play funcao4(){}
2
+2
-2
2.3
-2.3
0
"Essa eh uma string"
True
False
["Joao","Victor",False,False,False,2,2.3,"Futebol",]
medley nossoModeloClasse{}
medley classe1{} = nossoModeloClasse
refrao(i=0 ; i<10 ; i++){}
refrao(re j=1 ; j>500 ; i--){}
bis(e==True){}
bis(a>b){}
ensaio(a>b){}
```

```
ensaio(b==False){}
improviso{}
finale
eco 0
do(v1)
re(v2)
k[2]
k[10]
nota()
show()
```

Saída(“*saida1.txt*”):

```
1. Foi encontrado uma operacao de DECLARACAO DE VARIAVEL. do a
2. Foi encontrado uma operacao de DECLARACAO DE VARIAVEL. re b
3. Foi encontrado uma operacao de DECLARACAO DE VARIAVEL. mi c
4. Foi encontrado uma operacao de DECLARACAO DE VARIAVEL. fa d
5. Foi encontrado uma operacao de DECLARACAO DE VARIAVEL. la e = True
6. Foi encontrado uma operacao de DECLARACAO DE VARIAVEL. do a = 2+2
7. Foi encontrado uma operacao de DECLARACAO DE VARIAVEL. la h = 2>2
8. Foi encontrado uma operacao de DECLARACAO DE VARIAVEL. la k = ~True|False
9. Foi econtrado a definicao de uma FUNCAO. do play funcao1(){}
10. Foi econtrado a definicao de uma FUNCAO. re play funcao2(){}
11. Foi econtrado a definicao de uma FUNCAO. sol play funcao3(){}
12. Foi econtrado a definicao de uma FUNCAO. mi play funcao4(){}
13. Foi encontrada uma variavel do tipo DO: 2
14. Foi encontrada uma variavel do tipo RE: +2
15. Foi encontrada uma variavel do tipo RE: -2
16. Foi encontrada uma variavel do tipo MI: 2.3
17. Foi encontrada uma variavel do tipo MI: -2.3
18. Foi encontrada uma variavel do tipo DO: 0
19. Foi encontrada uma variavel do tipo SI: "Essa eh uma string"
20. Foi encontrada uma variavel do tipo LA: True
21. Foi encontrada uma variavel do tipo LA: False
22. Foi encontado um TAD_ACORDE.
    ["Joao","Victor",False,False,False,2,2.3,"Futebol",]
23. Foi encontrado uma operacao de DECLARACAO DE VARIAVEL. medley
    nossoModeloClasse{}
24. Foi encontrado uma operacao de DECLARACAO DE VARIAVEL. medley classe1{} =
    nossoModeloClasse
25. Foi encontrado um LACO REFRAO. refrao(i=0 ; i<10 ; i++){}
26. Foi encontrado um LACO REFRAO. refrao(re j=1 ; j>500 ; i--){}
27. Foi encontrado um LACO BIS. bis(e==True){}
28. Foi encontrado um LACO BIS. bis(a>b){}
29. Foi encontrado a condicional ENSAIO. ensaio(a>b){}
30. Foi encontrado a condicional ENSAIO. ensaio(b==False){}
31. Foi encontrado a condicional IMPROVISO. improviso{}
32. Foi encontrada uma operacao de PARADA DE EXECUCAO. finale
33. Foi encontrada uma operacao de RETORNO DE VALOR. eco 0
34. Foi encontrada uma EXPRESSAO DE CASTING EXPLICITO. do(v1)
35. Foi encontrada uma EXPRESSAO DE CASTING EXPLICITO. re(v2)
36. Foi encontrado uma operacao de REFERENCIAMENTO. k[2]
37. Foi encontrado uma operacao de REFERENCIAMENTO. k[10]
38. Foi encontrado uma FUNCAO DE ENTRADA DE DADOS. nota()
39. Foi encontrado uma FUNCAO DE SAIDA DE DADOS. show()
```

Refatorações da Etapa Anterior

- Primeiramente, atendendo às sugestões do professor, criamos alguns arquivos de teste tanto para nos auxiliar na montagem dos processos desta parte quanto para explicitar ainda mais a nossa linguagem, mostrando suas características mais intrínsecas. Tais arquivos contendo detalhes sintáticos mais aprofundados podem ser consultados nos três arquivos de teste: *entrada.txt, entrada2.txt* e *entrada3.txt*.
- Em segundo lugar, notamos que havíamos trocado as etapas do LEX e do Yacc, visto que na primeira etapa implementamos as Definições Regulares ao invés de implementar a identificação de tokens unitariamente. Isso não foi necessariamente um problema para nós já que aproveitamos grande parte das definições previamente feitas. Portanto, tivemos que implementar a identificação dos tokens de maneira singular ao invés de identificar todo o formato da expressão como havíamos feito.
- Posteriormente, também atendendo à uma sugestão do professor e, paralelamente, fazendo algumas análises e projeções futuras do nosso Compilador através de mapeamentos utilizando a Tabela de Símbolos, optamos por não implementar o conceito de Orientação a Objetos, sobretudo na dominância de Classes e Objetos. Primeiramente, notamos que a etapa mais básica da definição das Classes era extremamente extensa, pensando que, por exemplo, na nossa concepção da linguagem nossas Classes poderiam ter ou não métodos e atributos, em como diferenciaríamos via tabela de símbolos a instanciação de uma Classe, adentrando à ideia de Objetos. Por esses motivos principais, decidimos não prosseguir com a implementação dos conceitos de OO neste trabalho, somado às limitações de tempo de implementação.
- Outro ponto a se comentar é acerca da Declaração Com Atribuição. Decidimos optar por manter uma abordagem dividida, ou seja, toda atribuição pode acontecer em algum instante de tempo depois da declaração, ou seja, não acontece no mesmo momento. O fator principal que nos levou à essa decisão foi o fato de que fizemos essa regra de maneira recursiva e essa regra

estava gerando muitos conflitos internos no nosso Compilador. Tentamos modificar a Recursão à Esquerda para Recursão à Direita como forma de combater a ambiguidade nesse ponto mas foi uma tentativa falha, visto que ainda assim havia problemas correlacionados fortemente com a recursão dessa definição.

- Por fim, adicionamos o token “;” como demarcador de fim de linha visando alicerçar as definições de formas sentenciais de maneira bem explícita ao usuário.

LEX

Como mencionado anteriormente, o arquivo do lexer sofreu algumas alterações importantes para essa parte do trabalho prático. Antes, as tarefas relacionadas ao analisador sintático (YACC) estavam sendo realizadas no próprio lexer. Agora, ao reconhecer um determinado padrão, um token é lançado para o analisador sintático que realiza as “análises de forma” necessárias.

Outra coisa importante é que, além de lançar o token, o Lexer é responsável por colocar o padrão encontrado em um arquivo. Essa funcionalidade de inserção em arquivo é responsável por salvar cada linha do código fonte e seu número de linha relativa. Após lançar todos os tokens, caso o programa não lance nenhum erro, teremos um arquivo com todo o código fonte entrado e suas respectivas linhas de código, caso contrário, se o programa gerar algum erro, conseguiremos através desse arquivo indicar onde o erro de sintaxe ocorreu no código. Para o arquivo “melodia.l” temos, então, o seguinte trecho de código:

```
#.*                 { escreve_arquivo(yytext); }

{TIPOS}                 { escreve_arquivo(yytext); return TIPO;}

"ensaio"                { escreve_arquivo(yytext); return IF; }
"improviso"             { escreve_arquivo(yytext); return ELSE; }
"bis"                   { escreve_arquivo(yytext); return WHILE; }
"refrao"                { escreve_arquivo(yytext); return FOR; }
"continue"              { escreve_arquivo(yytext); return CONTINUE; }
"finale"                { escreve_arquivo(yytext); return BREAK; }
"void"                  { escreve_arquivo(yytext); return VOID; }
"eco"                   { escreve_arquivo(yytext); return RETURN; }
"acorde"                { escreve_arquivo(yytext); return ACORDE; }
"play"                  { escreve_arquivo(yytext); return PLAY; }
```

Figura 1 - Reconhecimento de alguns tokens no LEX.

No código acima podemos ver que antes de retornar o token “TIPO” é feito uma função com o padrão reconhecido como argumento, essa função recebe esses argumentos, inclusive a quebra de linha, e quando há uma quebra de linha, o número da linha é adicionado ao arquivo também.

```

\n                          { escreve_arquivo(yytext); char stringNum[20];
sprintf(stringNum,"%d",linha); escreve_arquivo(stringNum); linha +=1;}
.
```

Figura 2 - Escrita do “\n” no arquivo de saída.

Observe que nenhum token é enviado ao receber a quebra de linhas, mas ela é escrita no arquivo e posteriormente, na próxima linha do arquivo, é inserido o número da linha (indicado pela variável linha). Após ser inserida no arquivo, a linha é incrementada.

A função responsável por inserção da string no arquivo é uma simples inserção em um arquivo com abertura “a”, para fazer um *append* no conteúdo já existente :

```
1.
2. void escreve_arquivo(char *texto){
3.     FILE *pont_arq;
4.     pont_arq = fopen("impresso.txt", "a");
5.     fprintf(pont_arq, "%s ", texto);
6.     fclose(pont_arq);
7. }
```

Figura 3 - Montagem e enumeração do código-fonte.

Após todas inserções, basta fazer uma leitura desse arquivo no programa YACC, e caso o programa esteja sintaticamente correto, a função de impressão será chamada na função “main”, e caso haja algum erro de sintaxe, ela será chamada na função “yyerror”. Caso haja algum erro de sintaxe, o arquivo foi preenchido até o erro ocorrer, que indicará onde o analisador sintático encontrou um erro, então podemos indicar a linha em que o erro ocorreu e seu ponto de ocorrência no código fonte. Assim, a função de impressão, o main e o yyerror ficam da seguinte forma:

```c
void yyerror(char *c){
    printf("Erro %s na linha: %d\n",c,linha);
    imprimeCodigoFonte();
    printf("\nPrograma sintaticamente incorreto proximo a linha %d!\n", linha-1);
    exit(1);
}

void imprimeCodigoFonte(){
    FILE *pont_arq;
    pont_arq = fopen("impresso.txt", "r");
    char linha[100];

    while (!feof(pont_arq)){
        fgets(linha, 100 , pont_arq);
        printf("%s", linha);
    }

    fclose(pont_arq);
}

int main(){
    FILE *pont_arq;
    pont_arq = fopen("impresso.txt", "w");
    fprintf(pont_arq, "%s ", " 1 ");
    fclose(pont_arq);
    yyparse();

    imprimeCodigoFonte();
    printf("\nPrograma sintaticamente correto!\n");
    printf("\n");
    return 1;
}

```

Figura 4 - Funções para exibição de informações acerca do programa-fonte compilado.

YACC

Como comentamos anteriormente, já havíamos implementado primeiramente o YACC. Entretanto, foi necessário implementar o LEX para o reconhecimento dos tokens e isso foi gerando necessidade de adaptação e modificação nas definições que havíamos feito na primeira parte deste projeto. Além disso, observamos que algumas definições estavam muito complexas e não intuitivas e, somado ao fato de querermos simplificar um pouco a linguagem, como dissemos anteriormente, projetamos definições mais simples. Outro ponto interessante que, na nossa concepção, é fruto de nosso amadurecimento e *expertise*, foi o fato de irmos criando nossas Expressões Regulares guiadas à exemplos, de maneira a irmos criando primeiramente as expressões mais simples e que são a base das demais e irmos subindo na hierarquia sintática da linguagem. Dessa forma, conseguimos implementar as ER's bem mais fluídas e mais legíveis, mantendo um mesmo nível de funcionalidade quando comparada a etapa anterior.

Não é nosso intuito mostrar toda a estrutura do YACC implementado nesta documentação mas, a seguir, mostraremos sucintamente a forma como pensamos para construção da nossa linguagem. Focaremos na parte de **declarações**, que julgamos possuir um aspecto de utilização abrangente e um ótimo exemplo.

```
1.  program: commands|program commands;
2.
3.  commands: declarations | statements ;
4.
5.  variable: ID | pointer ID | ID array ;
6.
7.  pointer: pointer MULOP | MULOP ;
8.
9.  array: array LCOLCH ICONSTANTE RCOLCH | LCOLCH ICONSTANTE RCOLCH ;
10.
11. /* DECLARAÇÕES */
12.
13. declarations:  declaration declarations| declaration ;
14.
15. declaration: TIPO declaration_names LITERAL_PONTO_E_VIRGULA |function_def ;
16.
17.
18. declaration_names: declaration_variable|declaration_names LITERAL_VIRGULA declaration_variable;
19.
20. declaration_variable: ID|ACORDE ID array|pointer ID ;
21.
```

Figura 5 - Algumas expressões no YACC

.

1. Primeiramente temos um *programa*, simulando o código-fonte.
2. Um *programa* pode possuir *declarações* ou *statements*.
3. As *declarações* podem ser *declarações simples* (do a;) ou *múltiplas* (do a,b,c;).

Como mostrado na figura, temos alguns tipos de declarações bem específicas e, dessa forma, decidimos não abordar todos os rumos a serem tomados a partir do passo 3.

Tabela de Símbolos

Na parte da implementação da Tabela de Símbolos, decidimos manter a abordagem horizontal presente na literatura que é a utilização de uma Tabela Hash com Lista Encadeada. Na nossa concepção, decidimos manter tal abordagem pelo fato de possuir uma simples implementação, serem estruturas que já vimos e trabalhamos em algum momento do curso e, nesse caso, fornecer um desempenho aceitável, pensando em uma implementação mais robusta da nossa linguagem. Dessa forma, adentrando em conceitos de escalabilidade, não teríamos que nos preocupar muito com a estrutura da Tabela.

Em alto nível, cada nó das Listas da Tabela apresentam a seguinte estrutura:

```
typedef struct {
    char *tipo;
    int escopo;
    int has_value;
    char *id;
    char *valor;
} Entidade; //

```

Figura 6 - Representação de uma entidade de programação na Tabela de Símbolos.

Denominamos esses nós como sendo uma Entidade no nosso código-fonte. Cada entidade possui um **tipo, escopo, id e valor.** Criamos também uma flag **has_value** para identificar se determinada Entidade possui um valor associado. A variável de **escopo** é para podermos rastrear o escopo de referenciamento das variáveis no código-fonte.

Sendo assim, mantemos a perspectiva de sinonímia em nossa linguagem.

Entretanto, até o presente momento, não conectamos a Tabela de Símbolos ao LEX e YACC por questões de tempo e imprevistos na obtenção das formas sentenciais digitadas pelo usuário via YACC. Mesmo assim, verificamos que ela está funcionando normalmente e essa integração irá efetivamente acontecer na próxima etapa de desenvolvimento, sendo a nossa primeira tarefa a ser realizada.

Exemplos de entrada e saída e instruções de execução

Para executar os casos de teste criados pelo grupo, basta executar os arquivos **.sh** gerados no diretório. Exemplo: **bash comandos.sh, comandos2.sh, comandos3.sh** ou **comandos4.sh**. Todos os comandos geram uma saída em arquivos **.txt** mostrando o código-fonte de entrada e se ele é correto ou não, sintaticamente falando. Caso não seja, mostramos também a linha onde ocorreu o erro.

Agora serão apresentados alguns arquivos de entrada criados pelo grupo e suas respectivas saídas.

Entrada 1: Para o primeiro exemplo de entrada, temos o seguinte código em melodia.

```
#Declaracao
do a;
do b;
re c;
mi d;
fa k;
sol broFazSol;
la emCasa;
si variable;
do *a;  #Ponteiro
do acorde vetor[20]; #Declaracao de Vetor

ensaio (2>3) { a=2; do a; } #Semelhante ao IF em C
improviso {do a;}




ensaio (a>3){
    a = 2;
    a=a++;
    a+=1;
    ensaio (b<4){
        b+=2;
        c+=3;

    }
    improviso a = 20;
 }
improviso { a = 0; }
do a;
refrao (a; a<2; a++) {b = b+1;} #For
bis (c<500){
 c = c+1;
}
do play func1 (do v1, do v2, re v3){
    a = 1;
}

a = func1();
a = 2 + func1(v1,v2,v3) + 3;
ensaio (a>0) a=b;

func47();

a = funcv1(v1);

a = do(a);

nota();
show();

```

Figura 7 - Entrada 1.

Saída 1: Para o código em melodia anterior, temos a seguinte saída.

```
1.  1  #Declaracao
2.  2 do a ;
3.  3 do b ;
4.  4 re c ;
5.  5 mi d ;
6.  6 fa k ;
7.  7 sol broFazSol ;
8.  8 la emCasa ;
9.  9 si variable ;
10. 10 do * a ; #Ponteiro
11. 11 do acorde vetor [ 20 ] ; #Declaracao de Vetor
12. 12
13. 13 ensaio ( 2 > 3 ) { a = 2 ; do a ; } #Semelhante ao IF em C
14. 14 improviso { do a ; }
15. 15
16. 16
17. 17
18. 18
19. 19 ensaio ( a > 3 ) {
20. 20 a = 2 ;
21. 21 a = a ++ ;
22. 22 a += 1 ;
23. 23 ensaio ( b < 4 ) {
24. 24 b += 2 ;
25. 25 c += 3 ;
26. 26
27. 27 }
28. 28 improviso a = 20 ;
29. 29 }
30. 30 improviso { a = 0 ; }
31. 31 do a ;
32. 32 refrao ( a ; a < 2 ; a ++ ) { b = b + 1 ; } #For
33. 33 bis ( c < 500 ) {
34. 34 c = c + 1 ;
35. 35 }
36. 36 do play func1 ( do v1 , do v2 , re v3 ) {
37. 37 a = 1 ;
38. 38 }
39. 39
40. 40 a = func1 ( ) ;
41. 41 a = 2 + func1 ( v1 , v2 , v3 ) + 3 ;
42. 42 ensaio ( a > 0 ) a = b ;
43. 43
44. 44 func47 ( ) ;
45. 45
46. 46 a = funcv1 ( v1 ) ;
47. 47
48. 48 a = do ( a ) ;
49. 49
50. 50 nota ( ) ;
51. 51 show ( ) ;
52. 52
53. 53
54. Programa sintaticamente correto!
55.
```

Figura 8 - Saída 1.

Note que para a saída anterior, a primeira numeração é do editor de texto e a segunda foi a inserida no analisador.

Entrada 2: Para o segundo exemplo de entrada, temos o seguinte código em melodia.

```
#Função para cálculo da hipotenusa

sol play calculaHipotenusa (sol catetoOposto, sol catetoAdjacente){
    sol hipotenusa;
    hipotenusa = catetoOposto**2 + catetoAdjacente**2;
    hipotenusa = hipotenusa**(1/2);
    eco hipotenusa;
}

do play metodoPrincipal(){
    show(calculaHipotenusa(2,2));

}

```

Figura 9 - Entrada 2.

Saída 2: Para o código em melodia anterior, temos a seguinte saída.

```
 1  #Função para cálculo da hipotenusa
 2
 3 sol play calculaHipotenusa ( sol catetoOposto , sol catetoAdjacente ) {
 4 sol hipotenusa ;
 5 hipotenusa = catetoOposto ** 2 + catetoAdjacente ** 2 ;
 6 hipotenusa = hipotenusa ** ( 1 / 2 ) ;
 7 eco hipotenusa ;
 8 }
 9
 10 do play metodoPrincipal ( ) {
 11 show ( calculaHipotenusa ( 2 , 2 ) ) ;
 12
 13 }
 14
 15
Programa sintaticamente correto!

```

Figura 10 - Saída 2.

Entrada 3: Para o terceiro exemplo de entrada, temos o seguinte código em melodia.

```
#Função para cálculo de do Fibonacci não-recursivo

do n;
n = do(nota());

do t1,t2,t3;

t1 = 0;
t2 = 1;
t3 = 0;

bis (t3 <= n){
    show(t3);
    t3 = t1 + t2;
    t1 = t2;
    t2 = t3;
}

```

Figura 11 - Entrada 3.

Saída 3: Para o código em melodia anterior, temos a seguinte saída.

```
 1  #Função para cálculo de do Fibonacci não-recursivo
 2
 3 do n ;
 4 n = do ( nota ( ) ) ;
 5
 6 do t1 , t2 , t3 ;
 7
 8 t1 = 0 ;
 9 t2 = 1 ;
10 t3 = 0 ;
11
12 bis ( t3 <= n ) {
13 show ( t3 ) ;
14 t3 = t1 + t2 ;
15 t1 = t2 ;
16 t2 = t3 ;
17 }
18
19
Programa sintaticamente correto!

```

Figura 12 - Entrada 3.

Entrada 4: Para o quarto exemplo de entrada, usaremos um código com erro de sintaxe, para isso, segue a seguinte entrada em melodia.

```
#Função para cálculo de do Fibonacci não-recursivo

do n;
n = do(nota());

do t1,t2,t3;

t1 = 0;
t2 = 1;
t3 = 0;

bis (t3 <= n){
    show(t3);;;;
    t3 = t1 + t2;
    t1 = t2;
    t2 = t3;
}

```

Figura 13 - Entrada 4.

Saída 4: Note que na linha 13, há um erro de sintaxe, visto que existem 4 ponto e vírgulas em sequência. Para essa entrada, temos a seguinte saída.

```
Erro syntax error na linha: 14
 1  #Função para cálculo de do Fibonacci não-recursivo
 2
 3 do n ;
 4 n = do ( nota ( ) ) ;
 5
 6 do t1 , t2 , t3 ;
 7
 8 t1 = 0 ;
 9 t2 = 1 ;
 10 t3 = 0 ;
 11
 12 bis ( t3 <= n ) {
 13 show ( t3 ) ; ;
Programa sintaticamente incorreto proximo a linha 13!
```

Figura 14 - Saída 4.

Parte Final

Integração com a Tabela de Símbolos: para essa parte, conseguimos integrar a Tabela de Símbolos com os módulos do LEX e YACC. Dessa forma, todas as declarações e atualizações de valores de variáveis são gerenciadas na Tabela de Símbolos. Nesta parte, aderimos à estratégia de mapear o escopo de onde as variáveis são declaradas dentro do programa. Dessa forma, toda vez que é identificado o token "{", o escopo do programa é incrementado e, analogamente, quando identificamos o "}", além de decrementarmos o escopo, nós excluímos os dados na Tabela de Símbolos referente às variáveis que foram declaradas ou atualizadas naquele escopo.

Análise Semântica: Para realização da análise semântica foi criado uma função chamada "analiseSemantica" que é chamada em todas as vezes que uma operação de atribuição é realizada. Nessa função, recebemos o identificador, a expressão e o escopo atual como argumentos e realizamos a análise em cima dos tipos de dados: inteiro (do), ponto flutuante (re) e string (si). Para cada um dos tipos propostos, verificamos se o que está no parâmetro "expressão" é um literal do tipo correto e se for, atribuímos a variável prevista no parâmetro "identificador", no caso de não ser, é verificado se o que foi passado existe na tabela de símbolos, e caso positivo, isso indica a existência de uma variável com aquele nome. No caso de a expressão ser uma variável, é feita uma busca na tabela de símbolos e o valor da mesma é retornado e passada para variável que está recebendo a atribuição. Vale ressaltar que a busca na tabela de símbolos ocorre pesquisando o escopo atual e os escopos menores do que ele, até chegar no global.

Decisões Gerais

Na tentativa de entregar o maior número de funcionalidades da linguagem o possível, reduzimos um pouco o suporte a estruturas de dados primitivas da linguagem Melodia. Suportes como vetores e funções existem somente na declaração mas a utilização de fato se mostrou um processo extremamente complexo e inviável para o escopo desta disciplina. Logo, decidimos atuar com suporte aos tipos mais básicos (textuais, numéricos e lógicos).

Problemas e Soluções

Aqui, explicaremos um pouco acerca das limitações que a nossa linguagem possui e o passo a passo de como poderíamos resolver tais problemas posteriormente:

Atribuição composta(+=,-=,*=,/=): Atualmente, estamos trabalhando apenas com atribuições simples, não obtendo valores de variáveis para algumas operações. Dessa forma, a atribuição composta seria uma implementação não muito complexa de se fazer em nossa linguagem caso houvesse uma disponibilidade maior de entrega. No nosso caso, deveríamos observar a expressão de ***assigment*** e tratar caso a caso (quando for +=, quando for -=, etc), cobrindo todas as possibilidades. Teríamos um *"if"* varrendo tais possibilidades, após isso, seria apenas recuperar o(s) valor(es) das variáveis referenciadas após o operador de atribuição composta via Tabela de Símbolos, visto que já possuímos uma função que faz tal tarefa.

Atribuição unária(++,--): É um problema semelhante ao caso anterior. Nesse caso, somamos sempre 1 na variável referenciada. Entretanto, na nossa condição atual, apenas pegamos o valor inicial da variável. Dessa forma, também deveríamos consultar a Tabela de Símbolos para obter o valor que já está lá, somar 1, e colocar esse novo valor na Tabela.

Entrada e saída com tabela de símbolos

Para executar o arquivo de teste *entrada.txt*, basta executar o comando no terminal: *bash comandos.sh*

Entrada:

```
#Declaracao
do a;
do b;
re c;
mi d;
fa k;
sol v1;
la v2;
si v3;
do *a;


ensaio (2>3) { a=2; do a; } #Semelhante ao IF em C
improviso {do a;}




ensaio (a>3){
    a = 2;
    a=a++;
    a+=1;
    ensaio (b<4){
        b+=2;
        c+=3;

    }
    improviso a = 20;
}

```

Figura 15 - Entrada.

Saída:

```
Programa sintaticamente correto!


| ---------------------- TABELA DE SIMBOLOS --------------------------- |
* ID:v1          |       Tipo:sol        |       Valor:(null)    |       Escopo:0 *
* ID:v2          |       Tipo:la         |       Valor:(null)    |       Escopo:0 *
* ID:v3          |       Tipo:si         |       Valor:(null)    |       Escopo:0 *
* ID:    |       Tipo:do         |       Valor:(null)    |       Escopo:1 *
* ID:    |       Tipo:do         |       Valor:(null)    |       Escopo:1 *
* ID:a   |       Tipo:do         |       Valor:20        |       Escopo:0 *
* ID:b   |       Tipo:do         |       Valor:2         |       Escopo:0 *
* ID:c   |       Tipo:re         |       Valor:3         |       Escopo:0 *
* ID:d   |       Tipo:mi         |       Valor:(null)    |       Escopo:0 *
* ID:k   |       Tipo:fa         |       Valor:(null)    |       Escopo:0 *
* ID:*a          |       Tipo:do         |       Valor:(null)    |       Escopo:0 *
| ---------------------------- FIM TABELA ------------------------------- |
joao@joaoPC:~/Documentos/TP_Compiladores/TP3-COMP$
```

Figura 16 - Saída.

Como é possível observar, há duas variáveis cujo ID é vazio(“ “). Isso acontece porque, como dissemos anteriormente, ao sair de um escopo, as variáveis são deletadas da tabela. Dessa forma, estamos inutilizando tais variáveis quando um escopo é fechado.

Conclusões

Acerca do seguinte trabalho, tivemos alguns desafios ao longo desta árdua jornada. Entretanto, estivemos sempre motivados a dar o nosso melhor e a tentar, ao máximo, entregar módulos coerentes com o esperado pelo professor e com as requisições da especificação deste trabalho prático. Tivemos ciência de que, em todas as entregas, cometemos alguns erros que, com certeza, nos fizeram aprender e melhorar para a etapa posterior. Dessa forma, mesmo que como resultado final apresentamos uma linguagem com certas limitações, estamos contemplados com todo o trabalho empenhado no processo como um todo.

Ademais, gostaríamos de ilustrar que de fato, não é/foi um trabalho com um nível de complexidade fácil para nenhum de nós deste grupo mas, acima de tudo, ficamos felizes e satisfeitos em ter participado no desenvolvimento, mesmo que mínimo, de algo tão complexo e obter resultados que nos satisfazem. Além disso, gostaríamos de agradecer pelos *feedbacks* tanto do professor, quanto do monitor, que foram essenciais para entendimento e resolução desta proposta de implementação.

"Sem a música a vida seria um erro"

-Friedrich Nietzsche

Referências

[1] AUTOR DESCONHECIDO. ***Full Grammar specification.*** *python.org*. Acesso em: *19/09/2021*. *Disponível em:* <*https://docs.python.org/3/reference/grammar.html*>.

[2] DEGENER, Jutta. ***ANSI C Yacc grammar.*** *lysator.liu. 1995*. Acesso em: *19/09/2021*. Disponível em: <https://www.lysator.liu.se/c/ANSI-C-grammar-y.html#declaration-specifiers>.

[3] AHO, Alfred; LAM, Monica; SETHI, Ravi; ULLMAN, Jeffrey. ***Compilers: Principles, Techniques, & Tools.*** *amazon.com*. 31/08/2006. Acesso em: *19/09/2021*. *Disponível em:* <*https://www.amazon.com/Compilers-Principles-Techniques-Tools-2nd/dp/0321486811*>.

[4] AUTOR DESCONHECIDO. ***Writing a simple Compiler on my own - Combine Flex and Bison***. *Steemit*. Acesso em: 12/10/2021. Disponível em: <https://steemit.com/utopian-io/@drifter1/writing-a-simple-compiler-on-my-own-combine-flex-and-bison?fbclid=IwAR0qL0Pf9fVLdD4OnUIMCij_6jENi-vRcUoRHNg6CJ0DaRgacBK0eR-skd0>.

[5] AUTOR DESCONHECIDO. ***Bison Grammar Files.*** Acesso em 24/10/2021. Disponível em: <https://www.math.utah.edu/docs/info/bison_6.html>.