Universidade Federal de Viçosa

Campus Florestal

Trabalho Prático - Parte 1
Especificação 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:** void ;

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	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

Exemplos:

♪ **Definição de variáveis:**

```
do v1
re v2
acorde[v1,v2]
si nome
```

♪ **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(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()
```

“Sem a música a vida seria um erro”

-Friedrich Nietzsche

Referências

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

[2] DEGENER, Jutta. ***ANSI C Yacc grammar.*** *lysator.liu. 1995. 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. *19/09/2021. Disponível em: <https://www.amazon.com/Compilers-Principles-Techniques-Tools-2nd/dp/0321486811>.*