

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL  
INSTITUTO DE INFORMÁTICA  
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO E  
ENGENHARIA DE COMPUTAÇÃO

DEMÉTRIO FRANCISCO FREITAS BOEIRA  
NICOLAU PEREIRA ALFF

## **Estudo sobre OCaml**

Relatório apresentado como requisito parcial para a  
obtenção de conceito na Disciplina de Modelos de  
Linguagens de Programação.

Orientador: Prof. Dr. Leandro Krug Wives

Porto Alegre  
2020

## SUMÁRIO

<b>1 INTRODUÇÃO</b>	<b>3</b>
1.1 Sobre OCaml	3
<b>2. HISTÓRICO</b>	<b>3</b>
<b>3. UTILIZAÇÃO DA LINGUAGEM</b>	<b>4</b>
3.1 Por que utilizar OCaml	4
3.2 Indústria	5
3.2.1 Facebook	5
3.2.2 Wolfram Mathcore	5
3.2.3 Zeo Agency	5
3.2.4 Docker	5
3.2.5 Bloomberg	6
3.2.6 Cryptosense	6
3.3 Universidades	6
3.3.1 América do Norte	6
3.3.2 Europa	6
3.3.3 Ásia	6
<b>4. ELEMENTOS DA LINGUAGEM</b>	<b>7</b>
4.1 Tipos	7
4.1.1 Tipos da linguagem.	7
4.1.2 Tipos criados por usuário	11
4.2 Operadores	11
4.4 Garbage Collector	14
<b>5. ORIENTAÇÃO A OBJETOS</b>	<b>14</b>
5.1 Classes e objetos	15
5.2 Herança	15
5.3 Encapsulamento	16
<b>6 PARADIGMA FUNCIONAL</b>	<b>16</b>
6.1 Recursividade	16
6.2 Expressões Lambda	16
6.3 Currying	17
6.4 Funções de alta ordem	17
6.5 Pattern Matching	18

6.6 Polimorfismo paramétrico	18
<b>7 ANÁLISE CRÍTICA</b>	<b>19</b>
<b>8 CONCLUSÃO</b>	<b>23</b>
<b>REFERÊNCIAS</b>	<b>24</b>

## 1 INTRODUÇÃO

Este relatório tem por objetivo apresentar a linguagem de programação OCaml, destacando os conceitos trabalhados durante o semestre, bem como uma análise crítica dos principais critérios de avaliação de uma linguagem de programação e também realizando comparações com outras linguagens de programação.

Para alcançarmos nosso objetivo, iremos apresentar um breve histórico da linguagem, para, dessa forma, sabermos sua origem e a evolução que existiu até a versão atual. Seguido de uma breve apresentação de onde esta linguagem está sendo utilizada nos dias atuais, principalmente para sabermos qual o impacto científico e industrial que OCaml representa. Então, apresentaremos alguns elementos básicos da linguagem que serão complementados com a apresentação de elementos de orientação a objetos e funcionais que ficarão em tópicos separados para a melhor organização deste relatório. Ao final, faremos uma análise crítica da linguagem e uma breve conclusão.

### 1.1 Sobre OCaml

OCaml é uma linguagem de programação de propósito geral com ênfase na expressividade e segurança.[...] possui um avançado sistema de tipagem que ajuda a detectar seus erros[...].É usado em ambientes onde um único erro pode custar milhões e a velocidade importa.[...] Por todo o seu poder, OCaml também é bastante simples, sendo este um dos motivos para ser utilizado com alguma frequência como linguagem de ensino. (What is OCaml?)

Complementando a apresentação feita pelo site da linguagem, OCaml suporta programação funcional, imperativa e orientada a objetos. Tem um sistema de tipos estático com suporte a polimorfismo e a inferência de tipos. Por pertencer a família ML, compartilha diversas características desta família.

## 2. HISTÓRICO

Como apresentado em (A History of OCaml) e por (MALAQUIAS), OCaml é uma linguagem de programação nomeada desta forma em 2011. É mais uma das linguagens da família ML. Outros exemplos de linguagens dessa família são: Standard ML (SML); F#; SML#; JoCaml e Manticore. OCaml é o resultado de uma série de evoluções da linguagem ML original, sendo esta criada por Robin Milner na década de 1970. Uma das primeiras evoluções se deu na década seguinte, quando Gérard Huet e outros pesquisadores introduziram uma versão ML para a máquina CAM (Categorical Abstract Machine), que havia sido criada

por Pierre-Louis Curien. Houve algumas outras implementações buscando otimizar e melhorar a performance da linguagem, até que na década de 1990, após a inclusão de suporte a orientação a objetos, se resultou na linguagem Objective Caml. Sendo renomeada para OCaml em 2011.

### 3. UTILIZAÇÃO DA LINGUAGEM

Neste capítulo, abordaremos um pouco do uso da linguagem OCaml no mundo. Tendo uma ampla comunidade ativa, a linguagem OCaml mostra que está cada vez mais ocupando o lugar que antes era de linguagens mais antigas e tradicionais. Por ser uma linguagem *open source*, manutenções e melhorias constantes são implementadas na linguagem por sua comunidade. Alguns exemplos dessa larga comunidade ativa podem ser facilmente encontrados em famosas plataformas na web, como GitHub (Trending OCaml), Reddit (let reddit = OCaml;;) e StackOverFlow (Newest ‘ocaml’ Questions).

#### 3.1 Por que utilizar OCaml

Conforme (MINSKY) fala em seu livro, OCaml é especial por ocupar um ótimo lugar no espaço de projetos de linguagens de programação. Essa linguagem consegue fornecer uma combinação de eficiência, expressividade e praticidade muito boa, ultrapassando o de diversas outras linguagens. Todos estes pontos positivos vem da combinação realizada ao longo dos anos da evolução de OCaml com a inserção de diversos recursos, como os seguintes: *Garbage Collector* para gerenciamento automático de memória; funções de primeira classe que podem ser passadas como argumento; verificação de tipos estática; polimorfismo paramétrico; suporte a imutabilidade e inferência de tipos; *Pattern Matching*. Conceitos estes que serão abordados no transcorrer deste relatório.

(MINSKY) ainda comenta que OCaml consegue se destacar dentro das outras linguagens por conseguir entregar um grande poder computacional, se mantendo ainda bastante pragmático para o programador.

O compilador tem uma estratégia de compilação direta que produz código de alto desempenho sem exigir otimização pesada e sem as complexidades da compilação just-in-time (JIT) dinâmica. Isso, junto com o modelo de avaliação estrito do OCaml, torna o comportamento do tempo de execução fácil de prever. O coletor de lixo é incremental, permitindo evitar grandes pausas relacionadas à coleta de lixo (GC) e preciso, o que significa que ele coletará todos os dados não referenciados (ao contrário de muitos coletores de contagem de referência) e o tempo de execução é simples e altamente portátil.(MINSKY)

## 3.2 Indústria

Como apresentado no site da linguagem (Companies using OCaml), diversas empresas ao redor do mundo utilizam a linguagem OCaml, fato que nos surpreendeu bastante. Algumas utilizam em larga escala, usando OCaml como linguagem chefe de alguns projetos. Outras, em menor escala, utilizando OCaml para projetos menores ou como uma linguagem secundária. A seguir, alguns exemplos reais da utilização de OCaml na indústria.

### 3.2.1 Facebook

O Facebook construiu uma série de ferramentas de desenvolvimento importantes usando OCaml. Hack, por exemplo, é um compilador para uma variante do PHP que visa conciliar o ciclo de desenvolvimento rápido do PHP com a disciplina fornecida pela tipagem estática. O Flow é um projeto semelhante que fornece verificação de tipo estático para Javascript. Ambos os sistemas são programas paralelos altamente responsivos que podem incorporar alterações no código-fonte em tempo real. Pfff é um conjunto de ferramentas para análise de código, visualizações e transformações de origem com preservação de estilo, escrito em OCaml, mas com suporte a muitas linguagens.(Companies using OCaml)

### 3.2.2 Wolfram Mathcore

Wolfram MathCore usa OCaml para implementar seu kernel SystemModeler. A principal função do kernel é traduzir modelos definidos na linguagem Modelica em código de simulação executável. Isso envolve a análise e transformação do código Modelica, processamento matemático de equações, geração de código de código de simulação C / C ++ e cálculos numéricos de tempo de execução.3.2.3 TrustInSoft.(Companies using OCaml)

### 3.2.3 Zeo Agency

A Zeo Agency é uma empresa de marketing digital com foco em ajudar as empresas a fazer melhor em SEO (Search Engine Optimization). Devido à natureza do nosso negócio, gerenciamos bilhões de linhas em nosso banco de dados e criamos percepções usando esses dados. Para utilizar nossas necessidades de forma eficaz, usamos OCaml em nossa parte de rastreamento e processamento de dados. (Companies using OCaml)

### 3.2.4 Docker

O Docker fornece um pacote de tecnologia integrado que permite que as equipes de desenvolvimento e operações de TI criem, enviem e executem aplicativos distribuídos em qualquer lugar. Seus aplicativos nativos para Mac e Windows usam código OCaml retirado do projeto de sistema operacional da biblioteca MirageOS. (Companies using OCaml)

### 3.2.5 Bloomberg

A Bloomberg, líder global em notícias e informações financeiras e de negócios, oferece aos tomadores de decisão influentes uma vantagem crítica ao conectá-los a uma rede dinâmica de informações, pessoas e ideias. A Bloomberg emprega OCaml em um aplicativo avançado de gerenciamento de risco de derivativos financeiros fornecido por meio do seu serviço Bloomberg Professional. (Companies using OCaml)

### 3.2.6 Cryptosense

Com sede em Paris, na França, a Cryptosense cria softwares de análise de segurança com foco particular em sistemas criptográficos. Um *spin-off* do instituto de pesquisa em ciência da computação (Inria), os fundadores da Cryptosense combinam mais de 40 anos de experiência em pesquisa e indústria. A Cryptosense oferece suas soluções para uma clientela internacional, em particular nos setores financeiro, industrial e governamental. (Companies using OCaml)

## 3.3 Universidades

Assim como no mundo corporativo, OCaml é amplamente utilizado no meio acadêmico. Abaixo seguem alguns exemplos de universidades que utilizam OCaml como linguagem principal. Estas informações estão disponibilizadas no site do OCaml (Teaching OCaml) ,

### 3.3.1 América do Norte

1. Boston College
2. Brown University
3. Harvard University
4. University of California

### 3.3.2 Europa

1. University of Cambridge
2. University of Rennes
3. Université Paris-Diderot
4. University Pierre & Marie Curie

### 3.3.3 Asia

1. Indian Institute of Technology

## 4. ELEMENTOS DA LINGUAGEM

Neste capítulo do relatório abordaremos os principais tipos da linguagem e seus operadores. No tópico de funções, abordaremos apenas uma pequena base que será complementada no capítulo 6, em que aprofundamos nas características e funcionalidades suportadas pela linguagem no paradigma funcional. Porém, achamos importante apresentar alguns elementos mais básicos neste ponto do relatório para que as próximas explicações sejam melhor compreendidas pelo leitor, uma vez que já se terá um conhecimento básico da sintaxe, comandos, estruturas e funcionalidades de OCaml.

Iremos apresentar alguns exemplos contendo trechos de código. O código completo e comentado se encontra presente no repositório github criado para este trabalho disponível no link: <https://github.com/npalff/MLP-OCaml>.

### 4.1 Tipos

Apresentaremos aqui os principais tipos da linguagem. Entretanto, vale destacar que OCaml permite que o programador crie novos tipos, como apresentaremos no tópico 4.1.2.

#### 4.1.1 Tipos da linguagem.

Em (What is OCaml?) são apresentados diversos tipos de dados já integrados à linguagem. Os tipos básicos da linguagem são: inteiro; números de ponto flutuante; booleano; caractere e string. Outro tipo de dado presente em OCaml é a unidade, que, similar ao *nil* e *null* em outras linguagens, tem o objetivo de fazer cumprir nas funções da linguagem o recebimento ou devolução de algum parâmetro. Mesmo que seja sem um valor propriamente dito, o único elemento desse tipo é o valor “()”. Outros tipos da linguagem, além dos básicos apresentados no mesmo site são: tupla; lista; matriz; conjuntos; tabelas hash; filas; pilhas; exceção e função. Neste trabalho, iremos explorar um pouco mais alguns desses tipos.

A linguagem OCaml suporta inferência de tipos, ou seja, o seu compilador consegue inferir automaticamente os tipos utilizados na programação sem precisar que o programador explicita enquanto programa. Porém, pode ser feito caso o programador queira ou precise fixar um tipo.

Para o melhor entendimento dos exemplos que apresentaremos, vamos primeiramente mostrar como são declaradas as variáveis e as funções na linguagem. A palavra reservada *let* permitirá fazer a associação do nome da variável ou função ao seu valor ou definição. Como nos exemplos a seguir, em que ambas variáveis serão do tipo `int`.

```
let var1 = 10
```



```
let fun1 = x*2
```

Para explicitar os tipos da linguagem, devemos utilizar dois-pontos (“:”) e o tipo a ser utilizado.

```
let exemplo1 (numero:int):int = numero * 2
let exemplo2 numero = numero*2
```

Note que nos dois casos acima os tipos são os mesmos, a variável número será do tipo `int` e o tipo de saída relativo a função `exemplo1` também será `int`, sendo esta uma função `int → int`. Ao verificarmos no compilador OCaml a função `exemplo2`, ele também retornará o tipo função `int → int`, de forma correta pelo seu aferidor de tipos.

Os exemplos aqui utilizados fazem parte de um código em um arquivo OCaml disponibilizado através de repositório no GitHub, acessível através do link: <https://github.com/npalff/MLP-OCaml>. Recomendamos que também realizem a leitura do código, pois lá incluímos mais exemplos e aplicações do que aqui está sendo explicado. O código e a apresentação dos tipos e operadores a seguir foram baseado em estudo realizados sobre (LEROY), (BARD), (WACLENA), (MALAQUIAS) e (MINSKY).

O tipo inteiro (*int*) é um número inteiro de precisão fixa e eles tem 2 bits a menos que outras linguagens de programação, suportando inteiros com até 30 ou 62 bits. Com isso, seu intervalo de representação é de  $-2^{30}$  a  $(2^{30} - 1)$  ou de  $-2^{60}$  a  $(2^{60} - 1)$ . Abaixo segue um exemplo de como definir uma variável do tipo `int`:

```
let exInt = 10    -- O compilador retornará: - : int = 10
```

O tipo de números de ponto flutuante (*float*) são valores aproximados de números reais. Em OCaml, eles seguem a padronização IEEE 754 com 53 bits para a mantissa e 9 bits para o expoente.

```
let exFloat = 19.4 -- O compilador retornará: - : float = 19.4
```

O tipo Caractere (*char*) “são valores definidos pela ISO-8859-1 (ISO Latin 1)” (WACLENA).

```
let exChar = 'a'
```

O tipo String (*string*) é uma sequência de caracteres, porém diferente das outras linguagens. Em OCaml, ela não é tratada como uma lista de caracteres. Inclusive, será apresentado no tópico 4.2, dentro de um dos exemplos de como podemos fazer a concatenação de um caractere com uma string, que não é tão simples quanto em outras linguagens que tratam strings de forma diferente de OCaml.

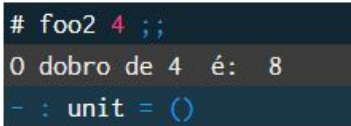
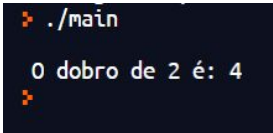
```
let exString = "Hello World!"
```

O tipo Booleano (*bool*) é o tipo dos valores lógicos *true* e *false*.

```
let exTrue = true;;    let exFalse = false;;
```

O tipo Unidade (*unit*) contém apenas um valor “()” (o de tupla vazia), e tem uma utilidade parecida ao que algumas outras linguagens dão ao *nil* ou ao *null*. Extremamente útil pela forma como OCaml funciona. Em OCaml, todas as funções devem receber pelo menos 1 argumento e também devem retornar algum argumento quando se encerram, porém existem funções, como a do exemplo *exUnit* a seguir, que não precisam de nenhum argumento de entrada, ou não precisam retornar nenhum valor, ou ainda a união dos dois casos anteriores de não precisar nenhum argumento de entrada e não precisar retornar nenhum valor. Para solucionar este problema, nestes casos o valor a ser dado como entrada de uma função ou retornado será () e o tipo *unit*. Isto inclusive faz com que possamos ter um comportamento similar ao de uma linguagem imperativa como C ou C++, como comparado no exemplo a seguir:

Tabela 1 - Exemplo de comparação entre OCaml e C++ com utilização de *unit*

OCaml	C++
<pre>let foo2 n = printf "O dobro de %d %s %d" n " é: " (dobro n); () (*caso de retorno de tipo unit*)</pre>	<pre>void foo1(int n){     std::cout &lt;&lt;"\n O dobro de "&lt;&lt;n&lt;&lt;" é: "&lt;&lt;(n*2)&lt;&lt;"\n";     return; }</pre>
 <pre># foo2 4 ;; O dobro de 4 é: 8 - : unit = ()</pre>	 <pre>./main O dobro de 2 é: 4</pre>

O tipo Tupla (*tuple*) é uma forma de agrupar elementos em uma *n-tuple*, onde *n* é o número de elementos e o separador de tuplas é a vírgula (,) sendo o uso de parênteses opcional. Este tipo da linguagem pode ser heterogêneo, ou seja, os tipos de seus elementos podem ser diferentes e isto inclusive será o que definirá o tipo dessa tupla. Uma observação muito importante é que a ordem desses tipos importará. Então, uma tupla do tipo *int\*bool* será diferente de uma tupla do tipo *bool\*int*. Seguem alguns exemplos para tornar um pouco mais visível esta diferença.

```
let tupla1 = 1,true - tipo int*bool
```

```
let tupla2 = true,1 - tipo bool*int
```

`let tupla3 = (19,false)` - tipo `int*bool`, note neste exemplo que o uso dos parênteses para tuplas é opcional

O tipo Lista (*list*) é outra forma de agrupar elementos da linguagem é usando o separador dos elementos quando utilizada a definição entre colchetes se dá por ponto-e-vírgula (;). Listas, por sua vez, devem ser homogêneas, isto é, todos os seus elementos devem ser do mesmo tipo. Por exemplo, na ExLista2 a seguir, representamos uma lista inválida que geraria erro ao se compilar o programa, isto pois os tipos de seus elementos não são os mesmos. Uma observação é que neste ponto, assim como a maioria das linguagens de programação, a indexação de listas inicia por 0 (zero). Isto se tornará importante para quando fomos apresentar os operadores de listas. De acordo com (WACLENA), Ocaml utiliza a mesma implementação de Lisp para listas.

```
let ExLista1 = [1,2;3,8] - lista válida
let ExLista2 = [true;2;false;3] - lista inválida
```

O tipo Matriz (*array*), assim como listas, também deve conter apenas elementos do mesmo tipo. Como (WACLENA) fala na página Built-In Data Types, o tratamento que Ocaml tem com arrays é semelhante ao que Perl e Tcl faz com o tipo *list* destas linguagens.

```
let mat3 = [| [1;2;3]; [4;5;6]; [7;8;9] |] - tipo: matriz de listas de inteiros
```

Ocaml ainda tem um tratador de exceções, o que gera o tipo exceção (*exn*), já existem algumas exceções definidas na linguagem e o programador ainda pode criar novas e definir tratamento para elas.

As exceções são uma construção muito dinâmica. [...]. As exceções podem (opcionalmente) receber um parâmetro, para que possam carregar um valor que pode ser extraído e examinado se a exceção for detectada. Os nomes de exceção devem começar com uma letra maiúscula inicial. Eles são construtores que retornam um determinado valor de exceção. (WACLENA)

Novas exceções podem ser definidas de 2 formas: `exception BugHere;` e `let ex1:exn = Not_found;` Para levantar alguma exceção utilizamos o comando *raise* e passamos o nome da exceção como argumento, por exemplo `raise BugHere.`

E por fim, sendo esta uma linguagem funcional, um dos tipos mais importantes da linguagem, o tipo função (*fun*). Este tipo pode ser armazenado, passado para outras funções como argumentos e também retornados de outras funções. Um exemplo de função é:

```
let exFuncao1 = (fun n->n+1)
```

Este comportamento de funções na linguagem OCaml é de extrema importância para alguns elementos da programação funcional como suporte a currying e a funções de alta ordem que serão apresentados no capítulo 6 deste relatório. Também como outras linguagens funcionais ela é baseada no cálculo lambda e suporta funções lambda.

O construtor utilizado na criação de funções é *fun* e o seu tipo genérico é  $T \rightarrow T'$ , onde  $T$  e  $T'$  são quaisquer outros tipos da linguagem podendo inclusive serem de mesmo tipo. O tipo  $T$  será o tipo de entrada esperado para esta função e  $T'$  será o tipo de saída., em OCaml a forma com que o compilador explicita que pode ser de tipo qualquer é utilizando `'a`, isto é, o tipo mais genérico de função em OCaml vai ser representado da seguinte forma pelo compilador: `('a → 'a)`.

#### 4.1.2 Tipos criados por usuário

Tipos podem ser criados pelo programador, podemos definir esse novo tipo através da construção `type <nome do tipo>`. E isto permite uma das aplicações que podem ser realizadas com OCaml, que é a definição de novas linguagens de programação. Um dos motivos dos quais é uma linguagem amplamente utilizada em universidades. É possível criar uma inferidor de tipos, avaliador e até mesmo compilador para uma máquina abstrata a partir disso. Segue abaixo um exemplo extraído do código da implementação da linguagem L0 descrita no Capítulo 3 do livro *Types and Programming Languages* de Benjamin C. Pierce. Temos na tabela a seguir a esquerda a criação do tipo `tipo`, em que define os novos tipos da linguagem que estamos implementando, a direita estamos definindo quais as operações dessa linguagem.

Tabela 2 - Exemplo de definições de novos tipos em OCaml

<pre>type tipo =   TyInt     TyBool     TyFn of tipo * tipo     TyPair of tipo * tipo</pre>	<pre>type op =   Sum     Sub     Mult     Eq     Gt     Lt     Geq     Leq</pre>
---	--

## 4.2 Operadores

Existem diversos operadores para cada um dos tipos da linguagem. Bastante similar a outras linguagens de programação. Na tabela a seguir, que é apresentado por (MALAQUIAS) no seu capítulo 2, irei apresentar alguns dos principais operadores. Logo após irei discorrer dos principais aspectos e o que se difere de diversas outras linguagens de programação.

Tabela 3 - Alguns operadores sobre os tipos da linguagem

Tipo	Alguns operadores
int	+ , - , * , / , mod , min_int , max_int
float	+. , -. , *. , **. , ceil , floor , sqrt , exp , log , log10 , sin , cos , tan , asin , acos , atan
bool	not , && ,    , = , <> , < , <= , > , >=
string	^ , length , get , uppercase , lowercase

Fonte: MALAQUIAS,2015

Em OCaml, assim como em algumas outras linguagens funcionais, os operadores podem ser chamados também como funções da linguagem. Sendo então os exOp1 e exOp2 abaixo corretos e calculando o mesmo valor.

```
let exOp1 = (+) 5 14
```

```
let exOp2 = 5 + 14
```

O programador também pode criar e/ou redefinir os operadores já existentes na linguagem. Ressalto aqui que a linguagem permite isso, cabendo ao programador ter o cuidado e boas práticas para documentar todas essas modificações. No exemplo abaixo mostrarei uma modificação sobre a operação de adição nos inteiros, que passará a subtrair.

```
let exOp3 = 5+3 - o resultado calculado para exOp3 será 8
```

```
let exOp3Modificado = let (+) a b = a-b in
```

```
5+3;; - O resultado calculado para esta operação será 2, graças a modificação feita.
```

Diferente de diversas outras linguagens que sobrecarregam as operações aritméticas para inteiros e floats, em OCaml haverá distinção dos operadores. Sendo que os operadores de floats receberam um ponto à direita dos operadores aritméticos de inteiros, isto é, +; - e \* , por exemplo, são apenas operadores de inteiros. Enquanto que para floats para realizar estas mesmas operações deve-se codificar com +. ; -. e \*. . A única exceção a isto é o operador unário (-) quando empregado para tornar o número negativo, de forma que -5 e -5.6 são valores válidos da linguagem de tipo int e float respectivamente. Floats ainda tem os operadores logaritmos e trigonométricos que podem ser utilizados. E para inteiros acho que vale ressaltar as operações *min\_int* e *max\_int*, que retornam menor e o maior valor representável da linguagem para inteiros, sendo isso bastante interessante para implementações de alguns algoritmos como os de sorting.

Em strings o operador de concatenação é o acento circunflexo (^).

```
let novaStr2 = "Hello"^" World!" - retorna a string "Hello World!"
```

Porém como em OCaml strings não são tratadas como lista de caracteres é necessário o comando (*String.make 1 <char>*) para transformar esse caractere em uma string q 1 caractere para só então concatenar. Como no exemplo a seguir:

```
let novaStr3 = (String.make 1 'E')^"xemplo de String" - retornando então "Exemplo de String".
```

Sobre o tipo lista irei explorar um pouco mais. O construtor de listas é ::, porém também podemos definir uma lista através de um açúcar sintático entre colchetes com o separador ponto-e-vírgula como demonstrado no tópico 4.1. Ou seja, as seguintes listas resultaram em listas iguais:

```
let exLista1 = [1;2;3]
```

```
let exLista2 = 1::2::3::[] - Note que sempre devemos acabar esse tipo de inicialização com a lista vazia [].
```

Para adicionarmos novos elementos a lista podemos fazer através do construtor, lembrando que o tipo deve ser o mesmo da lista de forma que esta seja homogênea.

```
let exLista3 = 'z'::['a';'g';'k';'p'] - resultará em ['z';'a';'g';'k';'p']
```

Para acessar o n-ésimo termo de uma lista usamos a função já definida na linguagem *List.nth*, passando então uma lista e o índice do termo da seguinte forma: `let nth1 = List.nth lst2 0` que retornará o primeiro elemento, o mais à esquerda, da lista.

Já existem funções de alta ordem definidas na linguagem para a utilização em listas, sendo elas a *map*, que recebe uma função e a aplica em todos os elementos da lista também passada como argumento, e a *filter* que dada uma função de filtro retorna uma lista filtrada. Para o melhor entendimento segue exemplos:

Tabela 4 - Exemplos de utilização de funções de alta ordem

Funções	Lista retornada
<code>let mapex1 = List.map (fun x -&gt; x+3) [1;2;3]</code>	<code>[4;5;6]</code>
<code>let filtex1 = List.filter (fun x -&gt; x &gt;= 'h') ['z';'a';'g';'k';'p']</code>	<code>['z'; 'k'; 'p']</code>

#### 4.4 Garbage Collector

OCaml tem implementado um sistema de garbage collector, de forma que o programador não precise alocar e desalocar memória explicitamente. Como apresentado por (MISNKY) o OCaml já faz o gerenciamento do ciclo de vida das variáveis. Fazendo verificações regulares de todos os valores alocados em memória, de forma que no momento que um espaço ocupado possa ser desalocado ele seja.

Um dos algoritmos de garbage collection que nos é apresentado no livro do (MINSKY) é o Mark and Sweep.

Quando não há memória suficiente disponível para satisfazer uma solicitação de alocação do pool de blocos de heap alocados, o sistema de tempo de execução invoca o garbage collector (GC). Um programa OCaml não pode liberar explicitamente um valor quando é feito com ele. Em vez disso, o GC determina regularmente quais valores estão ativos e quais valores estão mortos, ou seja, não estão mais em uso. Os valores mortos são coletados e sua memória disponibilizada para reutilização pelo aplicativo. (MINSKY)

### 5. ORIENTAÇÃO A OBJETOS

Como frisamos em boa parte deste documento, OCaml destaca-se pela sua versatilidade e aplicabilidade em diversos contextos. E para confirmar esta hipótese, iremos abordar neste capítulo aspectos da programação orientada a objetos em OCaml. Lembrando que o “O” em “OCaml” faz alusão à aderência ao paradigma orientado a objetos da linguagem, e este nome se oficializou em 2011.

OCaml permite escrever programas em um estilo orientado a objetos. Seguindo a filosofia da linguagem, a camada orientada a objetos obedece ao paradigma da “tipagem forte”: assim, é impossível enviar uma mensagem a algum objeto que não possa respondê-la. Essa segurança não tem um custo em expressividade: graças a recursos como herança múltipla e classes paramétricas, os padrões de projeto mais complexos podem ser expressos de maneira natural. (What is OCaml?)

Como (LEROY) apresenta em seu capítulo 3, “A relação entre objeto, classe e tipo em OCaml é diferente das demais linguagens orientadas a objetos convencionais, como Java e C++.” (LEROY). Os features de orientação a objetos são muito menos utilizadas em OCaml em relação ao seu paradigma funcional. Porém ainda assim existe este suporte na linguagem.

## 5.1 Classes e objetos

Uma classe pode ser definida como o exemplo a seguir, note entretanto que ao utilizar de orientação a objetos uma variável declarada para ter seu valor alterado deve conter a palavra reservada *mutable*

```
class pessoa =
  object
    val mutable nome = ""
    val cpf = 021
    method getNome = nome
    method mudaNome = nome <- nome
  end;;
```

let p = new pessoa - Aqui instanciamos um objeto da classe pessoa.

Ao executarmos o comando *p#getNome* receberemos a string contendo o nome de pessoa, neste caso iniciado com “”.

## 5.2 Herança

Nesta parte, falaremos sobre um conceito fundamental para o paradigma da orientação a objetos: herança.

No exemplo abaixo definimos uma classe funcionário herdeira da classe pessoa definida acima. Ao executar a classe pessoa passará a receber também todos os parâmetros e métodos da classe pessoa.

```
class funcionario =
  object
    inherit pessoa
    val mutable nrFunc = 5
    method getFunc = nrFunc
  end
```

OCaml também suporta herança múltipla, isto é, uma classe é vinculada ao ancestral de seu ancestral. O nome da classe será um identificador de pseudo-valor que só poderá ser usado para chamar métodos da superclasse.



### 5.3 Encapsulamento

O encapsulamento pode ser realizado através de métodos privados, os quais permitiram restringir o acesso a alguns componentes do objeto e também através da interface de classes que a orientação a objetos em OCaml suporta. Dessa forma, a representação interna do objeto ficará escondida da visão externa. Isso ajuda principalmente em aspectos de segurança dos objetos.

## 6 PARADIGMA FUNCIONAL

Assim como em Haskell, funções em OCaml são elementos de primeira ordem. Ou seja, funções podem ser criadas, passadas e recebidas como parâmetros ou retornos de outras funções.

Pela inferência de tipos utilizada pela linguagem e que é apresentada sempre ao término da execução e definição na linguagem, algumas padronizações foram definidas em OCaml para que essa apresentação não ficasse poluída demais. Com isso, quando apresentado tipos na linguagem, o construtor de funções  $\rightarrow$  terá sua associação a direita, mas a aplicação de funções tem sua associação a esquerda. Então, por exemplo, o tipo *int list*  $\rightarrow$  *bool*  $\rightarrow$  *int* significaria (*int list*  $\rightarrow$  (*bool*  $\rightarrow$  *int*)). Enquanto *List.map fun1 lista1* seria executada como se fosse ((*List.map fun1*) *lista1*).

### 6.1 Recursividade

“Recursividade é o mecanismo de programação no qual uma definição de função ou de outro objeto refere-se ao próprio objeto sendo definido.” (MALAQUIAS). A linguagem OCaml suporta realizar funções recursivas através do como *rec* na definição da função. Como no exemplo a seguir na definição de fatorial:

```
let rec fatorial n = if n = 0 then 1 else n * fatorial(n-1)
```

### 6.2 Expressões Lambda

OCaml sendo mais uma linguagem funcional, tem sua base no cálculo lambda. E tem suporte a expressões lambda, que são funções anônimas. A construção dessas expressões se dá da mesma forma que funções através do uso de *fun*, como demonstra o exemplo a seguir:

```
(fun y -> (y+2)*y) 8 - retornando o valor inteiro 80.
```

### 6.3 Currying

Também existe a possibilidade de utilizarmos Currying na linguagem OCaml, isto é, podemos passar menos argumento há uma função e isto retornará uma nova função que poderá ser aplicada a outro argumento. Segue exemplo:

```
let soma (x:int) (y:int) = x+y
```

`let inc10 = soma 10` - Neste ponto a função passou a ser *curried* e conseguimos definir uma nova função baseada, na original soma.

Um outro exemplo dessa utilização vai ser dada no item a seguir em Funções de alta ordem, pois isso nos permite, por exemplo, dada uma função de *mapping* qualquer para uma lista definir novas funções que serão aplicadas sobre listas apenas usando currying. Precisando passar apenas a lista para essa nova função.

### 6.4 Funções de alta ordem

“Uma função é conhecida como função de ordem superior quando ela tem uma função como argumento ou resulta em uma função”(MALAQUIAS). Isso é extremamente útil, pois nos permite realizar composição de funções em OCaml, de forma que não precisamos definir uma grande função extremamente complexa, sendo que a composição de funções bastante simples e facilmente testadas cumprem o mesmo trabalho, de forma que simplifique e facilite a programação na linguagem. Já mostrei algumas utilizações desse tipo de função nos operadores do tipo lista, que já tem implementados no core da linguagem algumas dessas funções prontas. Porém também podemos definir funções desta forma como seguem os exemplos:

A função `myMap` mapeia uma função recebida para todos os elementos de uma lista também passada como argumento, a definição utiliza pattern matching assunto do próximo tópico.

```
let rec myMap fn l=
  match l with
  [] -> []
  | head::tail -> (fn head) :: myMap fn tail
```

Uma utilização interessante através do Currying e de expressões lambda para funções de alta ordem é implementar uma função que seja aplicada diretamente a uma lista e tenha um comportamento já esperado, não precisando fazer várias chamadas da função de *map* e nem da função de parâmetro por parte do programador, como exemplifica a função abaixo *Soma1ALista*:

```
let soma1ALista l= myMap (fun x -> x+1) l
```

Quando executarmos então `soma1ALista [1;2;3]` teremos como resultado `[2;3;4]`, acho esse exemplo interessante por unirmos tantos conceitos e funcionalidades que OCaml tem.

Voltando a função `myMap` temos ela sendo do tipo `('a -> 'b) -> 'a list -> 'b list = <fun>`. O que acho importante ressaltar que ela está esperando uma função que tenha como entrada o mesmo tipo da lista a ser passada como argumento e tem o tipo de retorno da função o mesmo tipo da lista de retorno da `myMap`. Acho interessante mostrar como o inferidor de tipos consegue inclusive diferenciar quando esses tipos podem diferir utilizando 'a e 'b para distingui-los.

## 6.5 Pattern Matching

Pattern Matching “é o ato de verificação da presença de um padrão em um dado ou em um conjunto de dados. O padrão é rigidamente especificado. O pattern matching é usado para testar se o objeto de estudo possui a estrutura desejada”(MALAQUIAS). Em OCaml a estrutura que permite isso ser feito é através de um padrão que deve ser testado em relação ao elemento passada a esquerda de uma seta (`->`) e a direita a expressão correspondente ao que o programador deseja fazer em caso de encontrado o padrão. É um uso bastante similar ao switch-case da linguagem C. Note também que os casos apresentados devem ser exaustivos, porém se não se pretende encontrar todos os casos, utilizamos a notação underline ( `_` ) que denotará todos os casos restantes. Um exemplo de como pode ser utilizado pattern matching é o da função abaixo `tamanhoLista`, que retornará o número de elementos na lista passada como argumento.

```
let rec tamanhoLista l =
  match l with
  [] -> 0
  | head::tl -> 1 + (tamanhoLista tl)
```

## 6.6 Polimorfismo paramétrico

OCaml também tem suporte ao polimorfismo paramétrico, que inclusive se faz extremamente importante inclusive para outros recursos da linguagem como para a função `map`, que pode receber funções e listas de qualquer tipo e operar normalmente. Outro exemplo, é o de `tamanhoLista` definida no item anterior que independente do tipo de lista passada também consegue operar normalmente. Isso faz com que a linguagem tenha um ganho na sua expressividade sem perda na sua segurança de tipos.

## 7 ANÁLISE CRÍTICA

Tabela 5 - Análise Crítica com as características da linguagem

<b>Característica</b>	<b>Nota</b>	<b>Explicação para as notas atribuídas</b>
Simplicidade	7	Estando muito relacionado a ortogonalidade da linguagem, faz com que as notas sejam semelhantes. Porém, como quanto menor a variedade sintática ou semântica, melhor. A sobrecarga que não ocorre dos operadores de inteiros e floats, aspecto negativo por conta de aumentar a variedade de comandos com objetivos similares. E por haver suporte há algumas formas de apresentar o comando para executar a mesma coisa também se perde alguns pontos na simplicidade
Ortogonalidade	8	OCaml consegue atingir boa parte dos objetivos de uma boa ortogonalidade, mantendo uma sintaxe pequena e algumas estruturas de controle. Além da inferência de tipos conseguir solucionar uma série de problemas de exceção que poderiam haver, além do polimorfismo paramétrico que permite a reutilização de funções para tipos diferentes. Porém, alguns aspectos contam negativamente o que fez perder alguns pontos como por exemplo, não sobrecarregar alguns operadores de inteiros para operarem sobre floats.
Estruturas de Controle (disponibilidade, variedade, adequabilidade, grau de abstração)	6	Existem poucas estruturas de controle definidas na linguagem, porém apenas com elas e se aproveitando de recursões e do paradigma funcional se consegue programar diversos programas sem tanta dificuldade, principalmente pela simplicidade do funcionamento das estruturas existentes. Porém dada a baixa variedade e ao pequeno grau de abstração devemos retirar pontos desta característica.

Tipos e Estruturas de Dados  (disponibilidade, variedade, expressividade, extensibilidade, mecanismos de definição e reaproveitamento)	10	OCaml dispõe de vários tipos nativos, além de dispor de diversas estruturas e funções já existentes para esses tipos. Inclusive algumas funções de alta ordem como a <i>List.map</i> e a <i>List.filter</i> . Além disso ainda permite como já explicado a criação de novos tipos e de novos operadores, além de permitir a modificação dos já existentes. Ao ponto de por exemplo permitir a implementação de uma outra linguagem dentro de um programa OCaml. Também é uma linguagem multiparadigma, aceitando tanto o paradigma funcional quanto o orientado a objetos.
Mecanismos de definição e gerência de escopo (adequabilidade, variedade, expressividade e grau de proteção)	8	Contém bons mecanismos de definição, a gerência de escopo pode inclusive ser feita através do aninhamento realizado através do construtor “ <i>let ... in</i> ”, como utilizado no exemplo de modificação do operador de soma no capítulo 4 deste relatório.
Expressividade	10	Como esta é uma característica que se refere a facilidade de expressar computações em uma linguagem, consideramos que a flexibilidade que OCaml dá na utilização dos operadores. Outro ponto é a proximidade com a linguagem natural de sua sintaxe. Além de ter características funcionais e de orientação a objetos.
Mecanismos de Especificação e Verificação de tipos	10	Tem tipagem forte e estática, feita através da inferência de tipos nativa da linguagem. Com isso poucas inconsistências ocorrem e quando o fazem o compilador consegue detectá-las. Ainda permite que o programador explicita o tipo, de forma a forçar aquele tipo a estrutura (variável ou função). O compilador consegue ainda inferir tipos através de contexto, quando identificado que mais de um tipo

		pode ser utilizado na situação o compilador retorna um tipo genérico expressando isso.
Suporte ao tratamento de exceções  (há pré-definidas, verificadas ou não, há mecanismos de definição de novas?)	10	Existem exceções já implementadas na linguagem, além de permitir a criação de novas. Em ambos os casos permite que se faça a verificação e o tratamento dessas exceções.
Reusabilidade (variabilidade e expressividade de mecanismos, suporte a OO)	7	OCaml tem suporte para orientação a objetos, inclusive o O vem da inserção desse suporte. Além de permitir o uso de bibliotecas e módulos também.
Suporte e documentação (do fabricante, da comunidade, exemplos, facilidade de acesso)	10	Conta com uma documentação extensa, e inclusive com tutoriais em seu site (LEARN, OCaml), também contando com diversos exemplos de código. Além de uma comunidade bastante ativa. Conta ainda como apresentado no relatório com comunidades participativas em diversas plataformas como github, StackOverFlow e Reddit.
Generalidade	10	OCaml é uma linguagem de programação multiparadigma de propósito geral. Apesar do enfoque funcional, já demonstramos aqui neste relatório diversos aspectos de orientação a objetos também.
Portabilidade, longevidade	10	Apesar de ter apenas 9 anos com o nome de OCaml, ele descende de uma família que existe e vem evoluindo desde a década de 1970. Com a sua

		utilização em vários segmentos da indústria para diversas aplicações, e muito utilizada academicamente também.
Escalabilidade e desempenho	9	Como (MINSKY) fala, essa linguagem consegue fornecer uma combinação de eficiência, expressividade e praticidade muito boa, ultrapassando o de diversas outras linguagens.
Custo  (para usar, para instalar, para desenvolver, para aprender, de manutenção)	10	Possui um processo de instalação simples para instalação na máquina própria e ainda contam com alguns editores e compiladores online, como o tryOcaml ( <a href="https://try.ocamlpro.com/">https://try.ocamlpro.com/</a> ) e através da plataforma repl.it . Além disso como apontado em outras características possui boas qualidades, como escritabilidade e simplicidade, que auxiliam no desenvolvimento e na manutenção de softwares em OCaml. Além de contar com um bom suporte da comunidade, documentação e tutoriais no site da própria linguagem que faz com que o aprendizado também seja facilitado.
Confiabilidade	10	OCaml consegue entregar uma execução confiável, trabalhando dentro do que for esperado. Contém suporte a tratamento de exceções e inferência de tipos, além de uma boa legibilidade e expressividade. Fazendo com que o compilador consiga identificar erros, e quando não identificado pelo compilador o sistema de exceções consiga tratá-los.
Legibilidade	9	Outras características como a simplicidade, ortogonalidade, tipos e estruturas de dados ajudam a tornar esta uma linguagem com boa legibilidade. A facilidade de fazer módulos, em sua maioria funções por se tratar de uma linguagem funcional, mas

		podem ser também objetos torna a leitura do código melhor, além de ao executar tornar os tipos explícitos, mesmo quando o programador não o fez. Neste ponto não posso deixar de unir o código em si, de sua execução. Que pela forma com que o compilador OCaml funciona auxilia em muito este aspecto. Um pouco da nota foi retirada, pois alguns aspectos da sintaxe e a flexibilização da utilização de alguns elementos da linguagem permitem que o mesmo programador varie muito sua forma de se expressar, com isso pode atrapalhar a legibilidade do código, além da não obrigatoriedade de explicitar o tipo no corpo do código mesmo.
Escritabilidade	9	Sua alta expressividade, simplicidade e ortogonalidade fazem com que a escritabilidade também seja boa. Com a flexibilidade de formas de escrita, e diversas formas de poder se desenvolver um programa em OCaml torna a nota dessa característica bastante alta.

## 8 CONCLUSÃO

Após a pesquisa e o estudo sobre a linguagem OCaml, consegue-se perceber que OCaml é linguagem amplamente utilizada tanto na indústria quanto no meio acadêmico. Sendo ela uma linguagem multiparadigma e fruto da evolução de vários anos de desenvolvimento sobre a linguagem ML original chegando num estágio de linguagem poderosa, com alta eficiência e ao mesmo tempo de fácil aprendizado e de simplificada codificação. Alcançando dessa forma bom trade-off entre a eficiência de computação e a eficiência do programador.

Com todos os tópicos e conceitos apresentados, acredita-se ter conseguido apresentar os principais elementos da linguagem OCaml. Recomenda-se, neste sentido, a leitura e execução de nosso código, disponibilizado no github, para que também se tenha experiência prática com esta linguagem de programação.



Durante a realização do trabalho, não tivemos muitas dificuldades em encontrar material, pois a partir do site da linguagem e poucas pesquisas já se encontra uma quantidade grande de conteúdo. Cabendo então, o momento de filtrar para as melhores bibliografias. Isto sim sendo um desafio de como melhor apresentar a linguagem. Outra dificuldade encontrada foi referente a orientação a objetos em OCaml, esta área ainda tem pouco conteúdo. Muitas das referências apenas fazendo breves citações sobre o suporte a este paradigma.

## REFERÊNCIAS

OCaml. **OCAML**. Disponível em: <https://ocaml.org/> . Acesso em: 20 nov. 2020.

OCaml. **What is OCaml?**. Disponível em: <https://ocaml.org/learn/description.html> . Acesso em: 20 nov. 2020.

OCaml. **Companies using OCaml**. Disponível em: <https://ocaml.org/learn/companies.html> . Acesso em: 20 nov. 2020.

OCaml. **A History of OCaml**. Disponível em: <https://ocaml.org/learn/history.html> . Acesso em: 20 nov. 2020.

OCaml. **Learn**. Disponível em: <https://ocaml.org/learn/> . Acesso em: 20 nov. 2020.

OCaml. **Teaching OCaml**. Disponível em: <https://ocaml.org/learn/teaching-ocaml.html> . Acesso em: 20 nov. 2020.

LEROY, Xavier, et al. **The OCaml system release 4.11**: Documentation and user's manual. Le Chesnay-Rocquencourt: INRIA, 2020.

BARD, Adam, BATURIN. Daniil, **Learn OCaml in Y Minutes**. Disponível em: <https://learnxinyminutes.com/docs/ocaml/> . Acesso em: 20 nov. 2020.

WACLENA, Keith. **OCaml for the Skeptical**. Disponível em: <https://www2.lib.uchicago.edu/keith/ocaml-class/data.html> . Acesso em: 20 nov. 2020.

WACLENA, Keith. **Exception Handling**. Disponível em: <https://www2.lib.uchicago.edu/keith/ocaml-class/exceptions.html> . Acesso em: 20 nov. 2020.

MALAGUIAS, José Romildo. **Programação Funcional em OCaml**. Ouro Preto: [s.n.], 2015. Disponível em: <http://www.decom.ufop.br/romildo/2015-1/bcc222/slides/progfunc.pdf> . Acesso em: 20 nov. 2020.

MINSKY, Yaron; MADHAVAPEDDY, Anil; HICKEY, Jason. **Real World OCaml: Functional programming for the masses**. Disponível em: <https://dev.realworldocaml.org/index.html> . Acesso em: 20 nov. 2020 . " O'Reilly Media, Inc.", 2013.

**Trending OCaml**. Disponível em: <https://github.com/trending?l=ocaml&since=monthly> . Acesso em: 20 nov. 2020.

**let reddit = OCaml;;**. Disponível em: <https://www.reddit.com/r/ocaml/> . Acesso em: 20 nov. 2020.

**Newest 'ocaml' Questions**. Disponível em: <https://stackoverflow.com/questions/tagged/ocaml> . Acesso em: 20 nov. 2020

ALFF, Nicolau Pereira; BOEIRA, Demétrio Freitas Boeira. **MLP-OCaml**. Disponível em: <https://github.com/npalff/MLP-OCaml> . Acesso em: 20 nov. 2020.