Construção de Compiladores em OCaml

José Romildo Malaquias

BCC328: Construção de Compiladores I

Universidade Federal de Ouro Preto
Departamento de Computação

2014–2

SUMÁRIO

Parte I

OCaml

1 Introdução a OCaml

Resumo

OCaml é uma linguagem de programação de uso geral, com ênfase na expressividade e segurança. Desenvolvida há mais de 20 anos no INRIA[1] por um grupo de pesquisadores líderes, OCaml beneficia-se de um dos sistemas de tipos mais avançados e suporta os estilos de programação funcional, imperativo e orientado a objetos, que facilitam o desenvolvimento de um software flexível e confiável. Usado em ambientes onde um único erro pode custar milhões e onde velocidade é importante, OCaml é apoiada por uma comunidade ativa que desenvolveu um rico conjunto de bibliotecas e irá ajudá-lo a tirar o máximo de possibilidades de OCaml.

OCaml é uma linguagem de programação moderna, de alto nível, com muitos recursos úteis.

Neste capítulo você se familiarizará com a linguagem OCaml, que será utilizada no desenvolvimento de um compilador.

Sumário

1.1 Algumas características de OCaml

A linguagem OCaml oferece:

- um sistema de tipos poderoso
- tipos algébricos definidos pelo usuário e casamento de padrão
- gerenciamento automático de memória
- compilação separada de aplicações autônomas
- um sistema de módulos sofisticado

Visite o site de OCaml no endereço `http://ocaml.org/`. Lá você encontrará várias informações interessantes sobre a linguagem.

1.2 Instalação das ferramentas de desenvolvimento em OCaml

Para o desenvolvimento de aplicações em OCaml é necessário o compilador de OCaml e um conjunto de bibliotecas. A distribuição de OCaml oferece dois compiladores:

- `ocamlc`: um compilador para bytecode
- `ocamlopt`: um compilador nativo

[1] Institut National de Recherche en Informatique et en Automatique

Existe também um ambiente interativo (`ocaml`) onde o programador pode carregar módulos e avaliar expressões, facilitando o desenvolvimento incremental.

Para preparar os programas é necessário um editor de texto e um terminal, ou um ambiente de desenvolvimento integrado (IDE).

O desenvolvimento baseado na linha de comando usa um terminal e um editor de textos. O programador digita os arquivos fontes em um editor de texto e compila a aplicação em um terminal executando um dos compiladores.

Emacs (juntamente com o plugin Tuareg) é um bom editor de texto com facilidades para desenvolvimento em OCaml. Porém o grau de dificuldade de sua aprendizagem é considerado elevado. Emacs está disponível para a maioria das platformas, incluindo Linux e Windows.

Notepad++ é uma opção mais simples disponível no Windows. No Linux temos outras opções como o gedit (parte do ambiente GNOME) e o kate (parte do ambiente KDE).

Eclipse é um ambiente de desenvolvimento integrado muito usado com várias linguagens de programação como Java e C++. O plugin OcaIDE oferece suporte para desenvolvimento em OCaml no Eclipse.

O compilador de OCaml e suas bibliotecas podem ser instalados de várias maneiras. As mais usadas são:

- usando `OPAM`, um gerenciador de pacotes específicos para OCaml (ainda não disponível para Windows),
- usando um gerenciador de pacotes suportado para sua plataforma, ou
- a partir do código fonte.

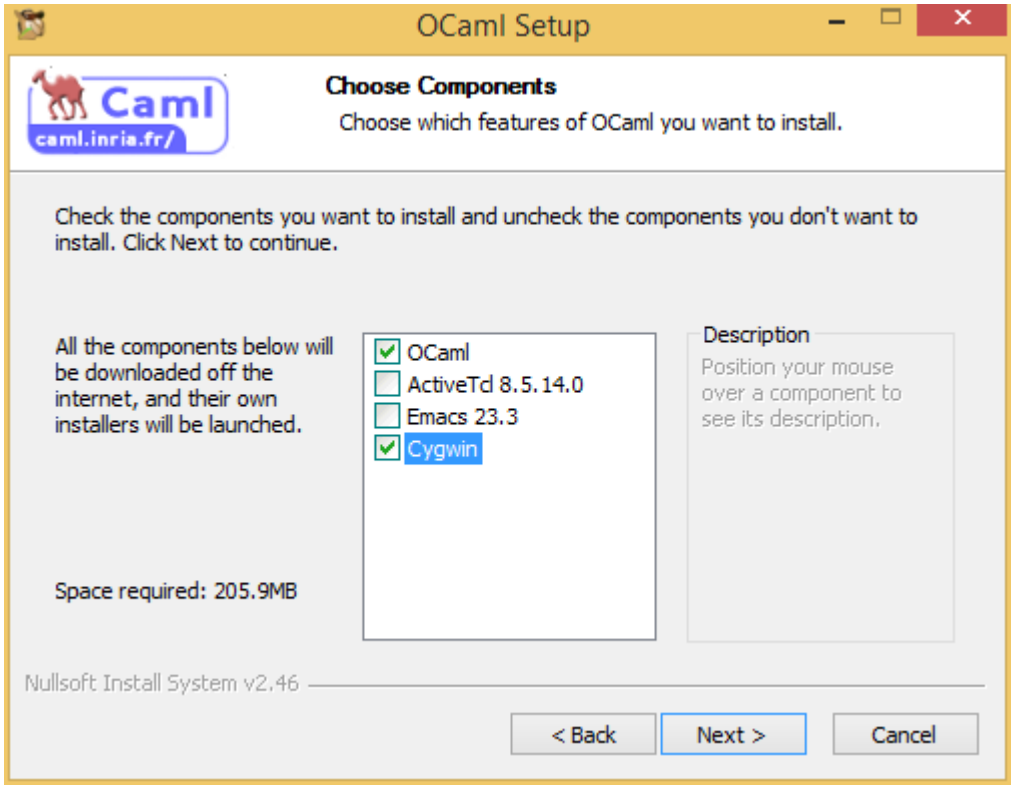
A página Install OCaml apresenta maiores detalhes.

1.2.1 Instalação do OCaml no Windows

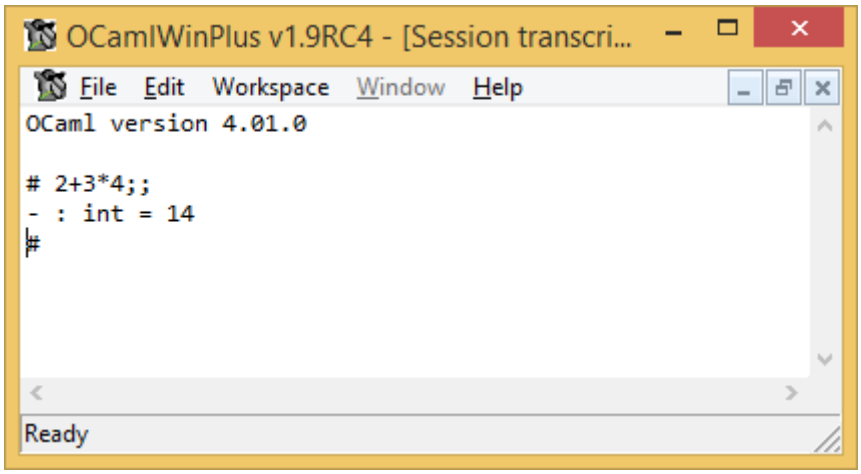
Há várias maneiras de instalar OCaml no Windows. Vamos usar o instalador oficial, disponível na página `http://protz.github.com/ocaml-installer/`.

1. Faça o donwload da versão mais recente do programa de instalação disponível na página `http://protz.github.io/ocaml-installer/`. No momento em que estas instruções foram escritas a versão mais recente era para `OCaml-4.01.0`, acessível pelo link `http://gallium.inria.fr/~protzenk/caml-installer/ocaml-4.01.0-i686-mingw64-installer3.exe`.
2. Execute o programa de instalação, que poderá instalar:
   - OCaml, findlib e flexdll.
   - Emacs (opcional), um editor de texto, com suporte para OCaml.
   - ActiveTCL (opcional), uma biblioteca para aplicações com interface gráfica. Necessário para usar OCamlBrowser ou para criar aplicações com interface gráfica usando Tcl/Tk (labltk).
   - Cygwin (opcional), uma coleção de ferramentas que permitem que o Windows possa de certa forma agir como um sistema Unix. Necessário para compilação nativa de programas em OCaml. O programa de instalação do Cygwin será executado com os pacotes necessários previamente selecionados.

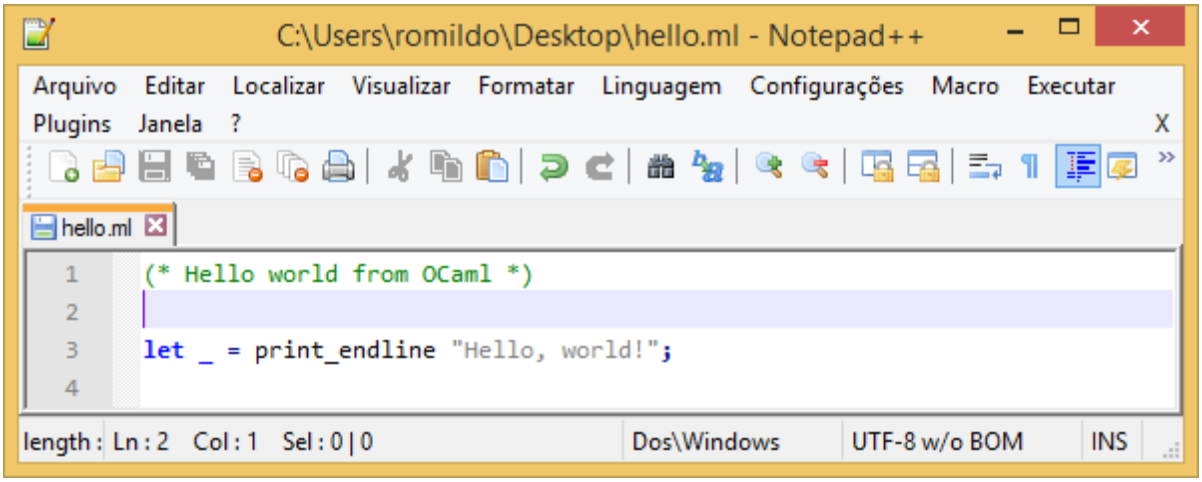
   Certifique-se de habilitar a instalação do Cygwin.



O ambiente interativo poderá ser executado usando o item OCamlWin no menu de programas. Será aberta uma janela onde você pode avaliar expressões e declarações.

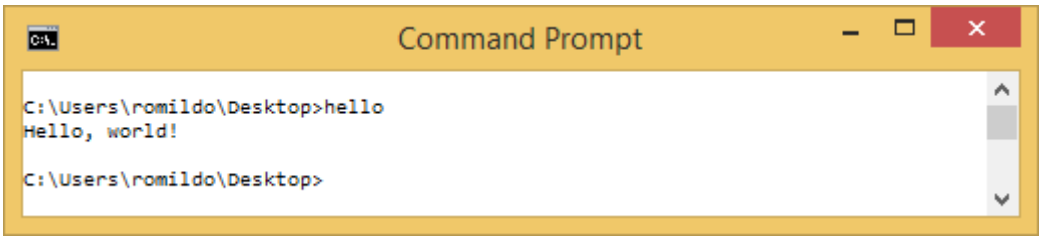


Para desenvolver um programa, edite o texto do programa em um editor de texto, como o Notepad++:



Em seguida compile o programa no terminal:

E execute o programa no terminal:



1.2.2 Instalação do OCaml no Ubuntu

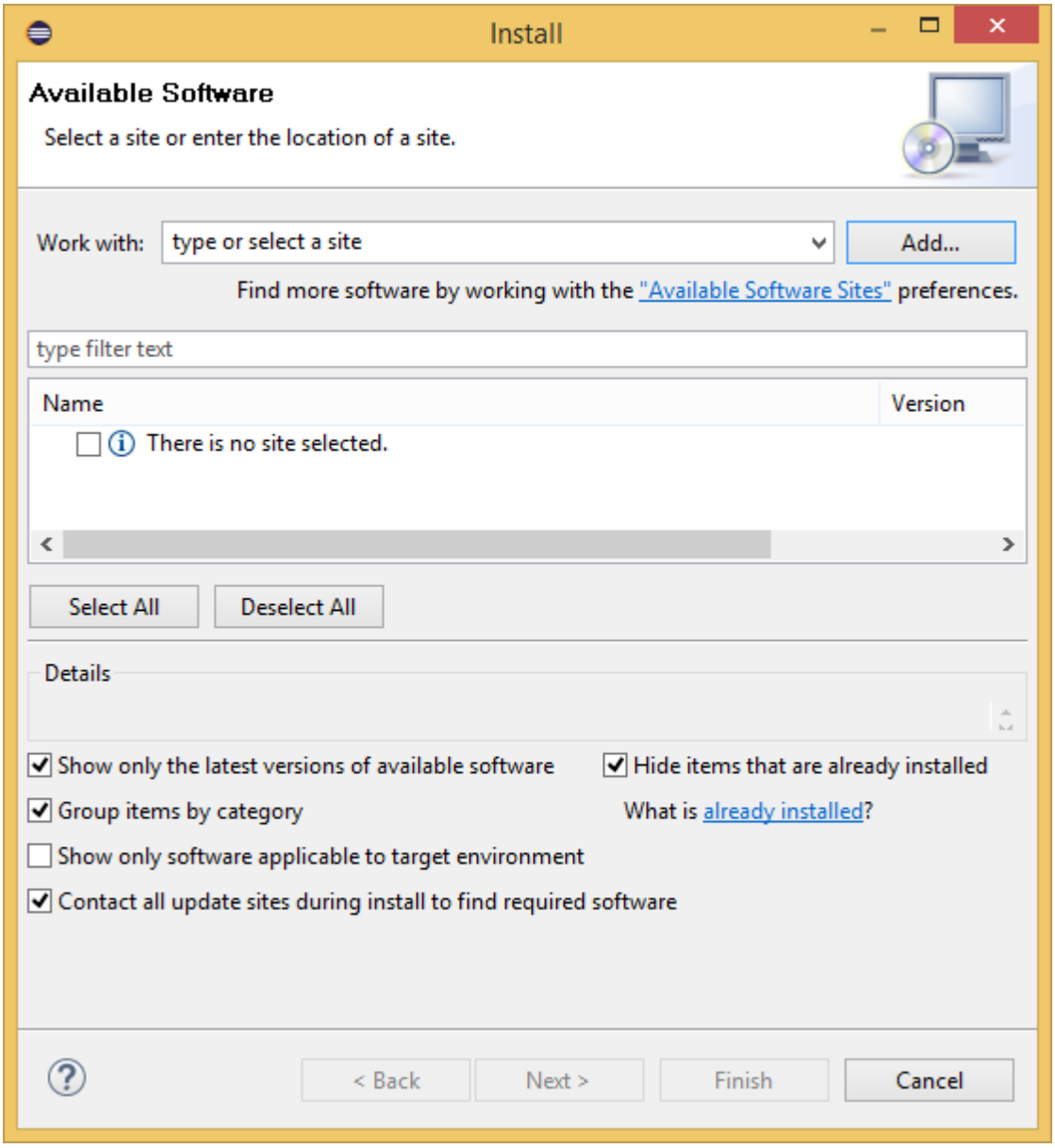
OCaml é muito fácil de ser instalado no Ubuntu. Use o gerenciador de pacotes para instalar os seguintes pacotes:

- `ocaml`, para desenvolvimento de aplicações
- `ocaml-native-compilers`, para compilação de programas para código nativo
- `ocaml-doc`, para ter acesso ao manual de referência
- `tuareg-mode`, um plugin para o editor Emacs
- `ocaml-findlib`, para instalar e usar bibliotecas e suas dependências facilmente
- `menhir`, um gerador de analisadores sintáticos
- `rlwrap`, para facilitar a edição da linha de entrada do ambiente interativo,

1.2.3 Instalação do ambiente interativo no Eclipse

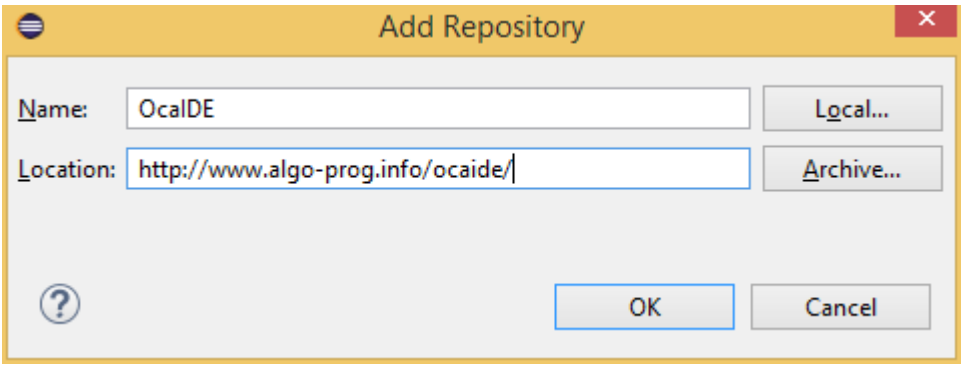
Para instalar OcaIDE, um plugin para Eclipse:

1. Instale a máquina virtual do Java versão 1.7 ou superior (`http://java.com/en/download/index.jsp`).
2. Instale uma versão recente do Eclipse (`http://www.eclipse.org/downloads/`).
3. Inicie o Eclipse.
4. No Eclipse acesse o menu `Help`→`Install New Software...`.



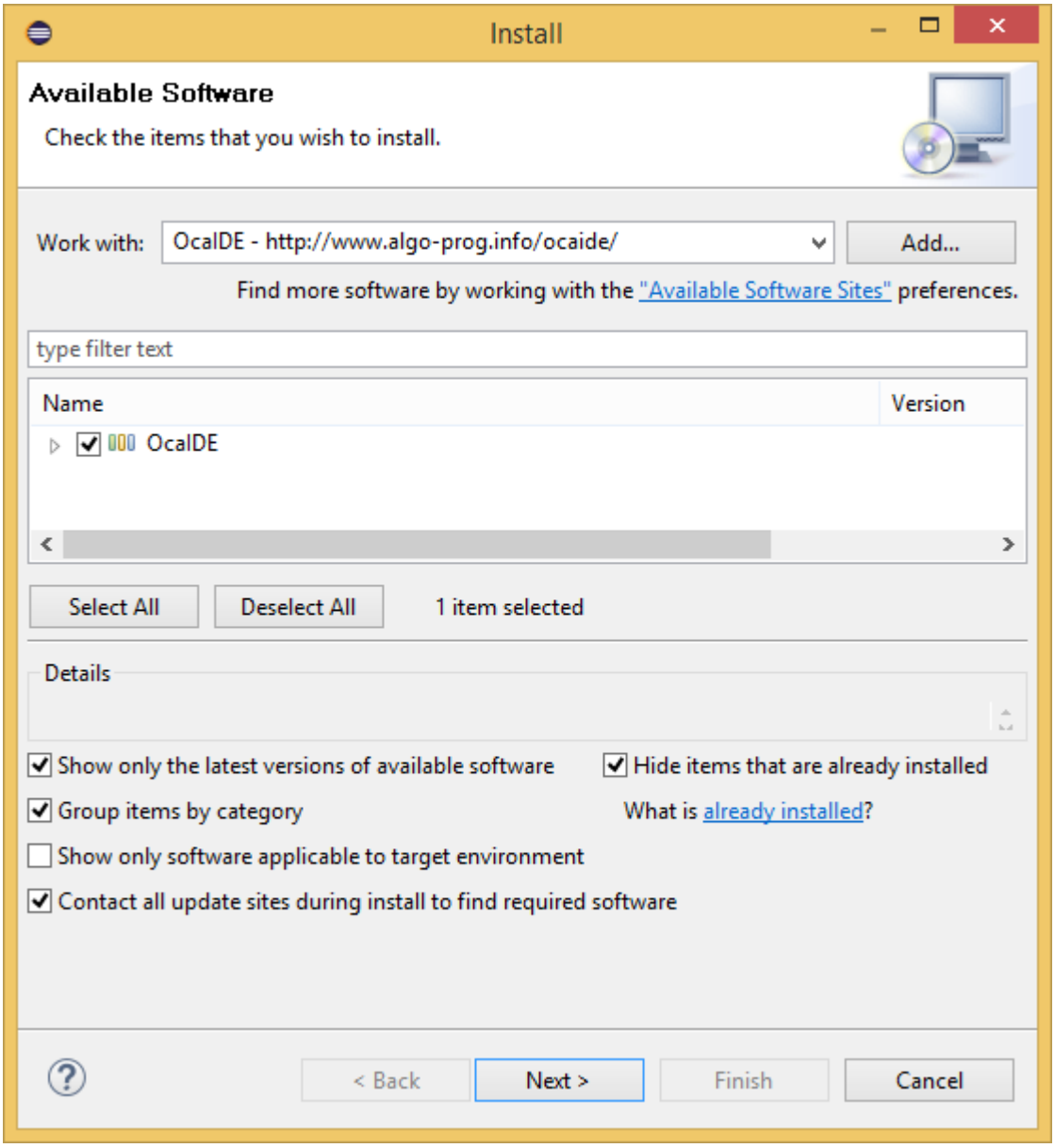
5. Clique Add para configurar um novo repositório.

6. No campo Name digite `OcaIDE` e no campo Location digite `http://www.algo-prog.info/ocaide/`.



7. Clique em Ok.

8. Selecione a categoria OcaIDE na lista de plugins.



9. Clique Next.

10. Clique Next.

11. Aceite os termos do acordo de licença.

12. Clique Finish. Será feito o download do plugin.

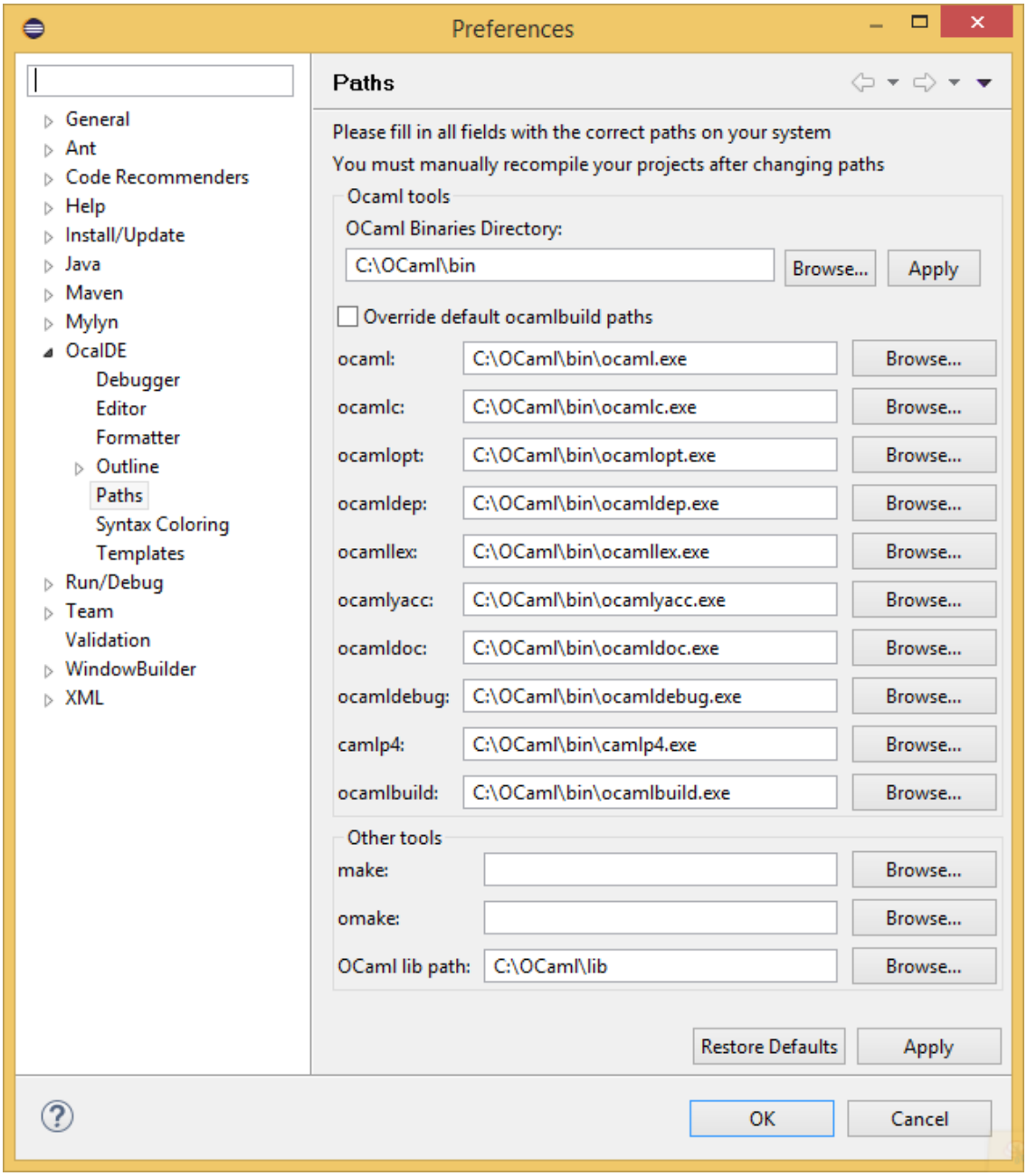
13. Aceite a instalação (o plugin não tem assinatura digital).

14. Reinicie o Eclipse quando solicitado.

15. Após o reinício do Eclipse, o ambiente já está pronto para uso.

16. Para acessar a ajuda online clique no menu `Help`→`Help Contents` e escolha *OCaml Development User Guide* na lista de tópicos disponíveis.

17. Verifique os caminhos onde o OCaml pode ser encontrado. Para tanto acesse o menu `Window`→`Preferences` e selecione a aba `OcaIDE`→`Paths`. Se necessário preencha o campo `Ocaml Binaires Directory` para inidicar a localização correta dos programas executáveis do OCaml e em seguida clique em `Apply`. Pode ser necessário indicar também o caminho das bibliotecas do OCaml no campo `Ocaml lib path`. Conclua clicando em `OK`.



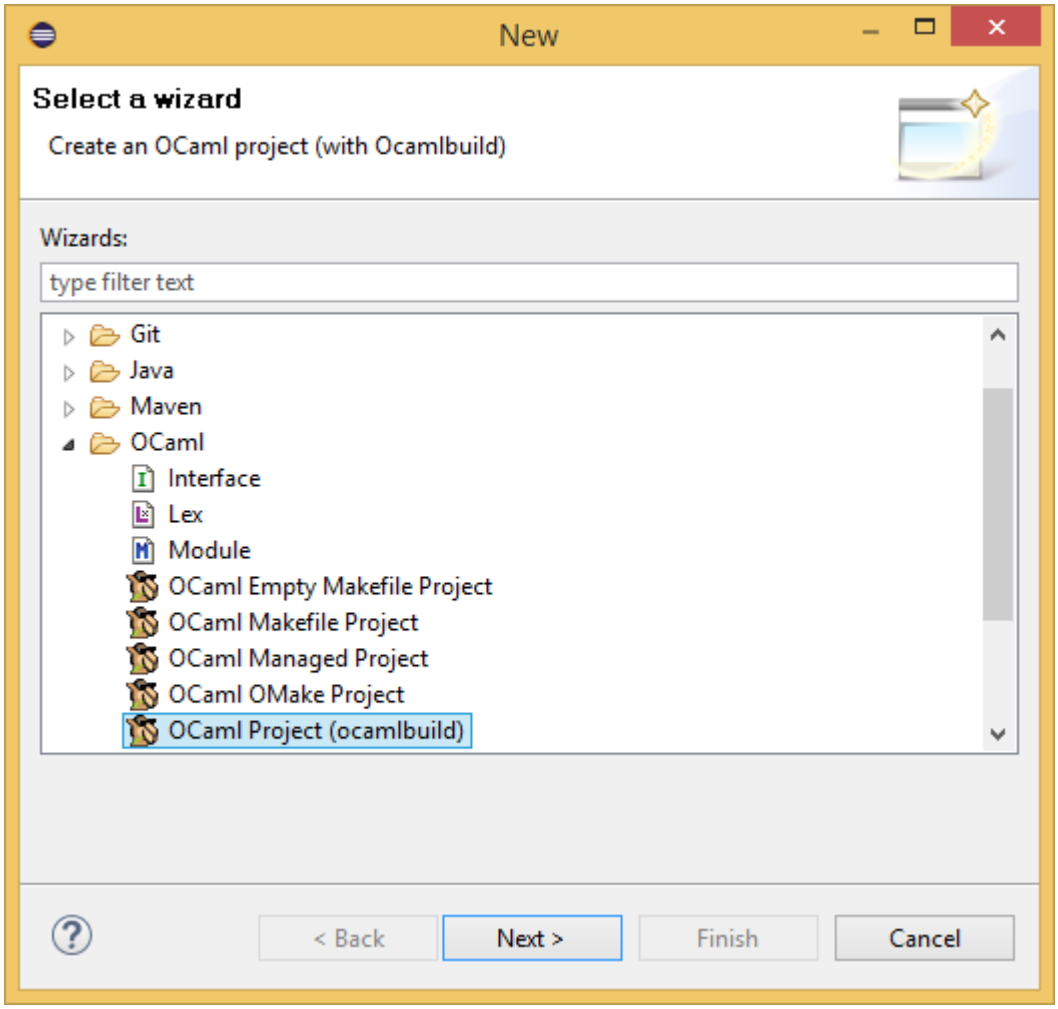
1.3 Desenvolvimento do primeiro projeto

Vamos desenvolver uma pequena aplicação para exibir uma mensagem na tela, usando o Eclipse.

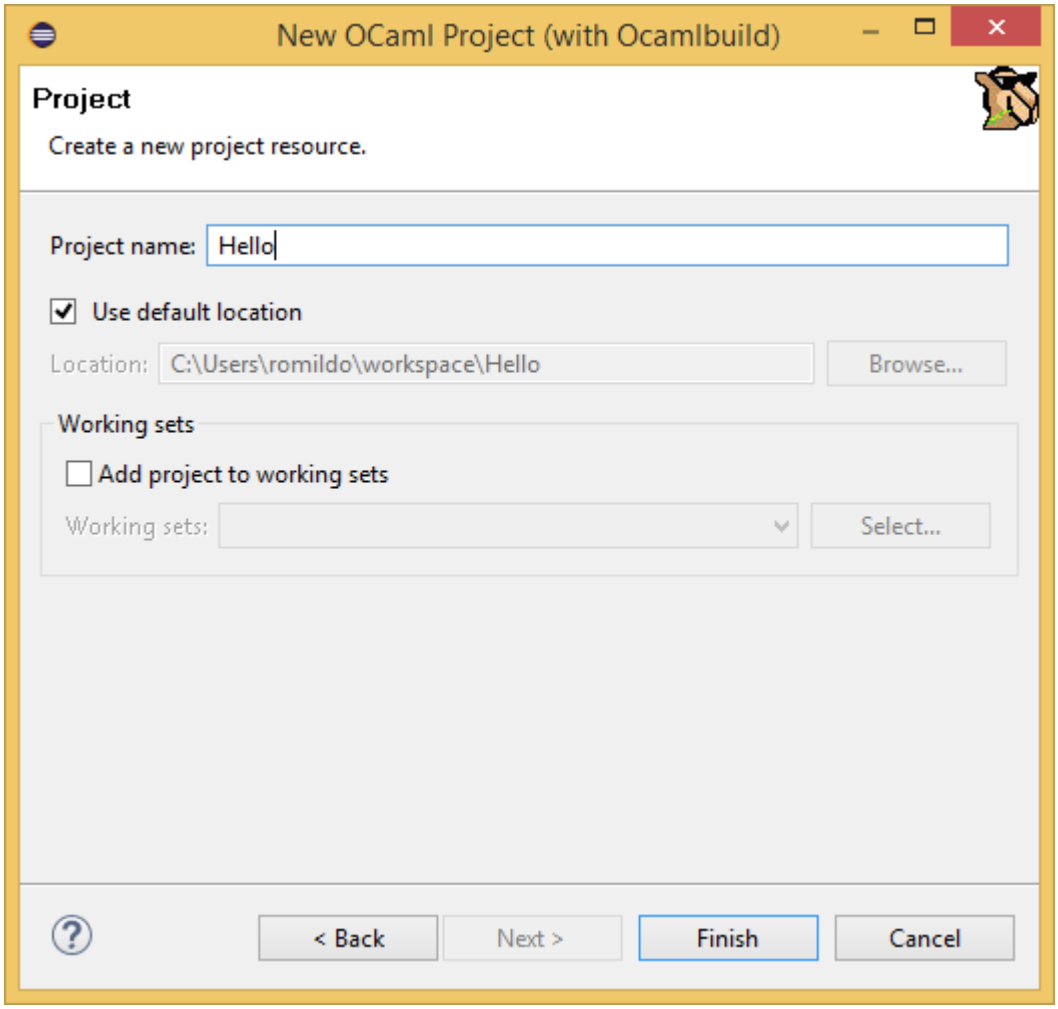
1. Para criar um novo projeto em OCaml no Eclipse use o menu `File→New→Other`.

2. Escolha a opção `OCaml Project (ocamlbuild)` na janela que se abre.

3. Em seguida clique em `Next`.

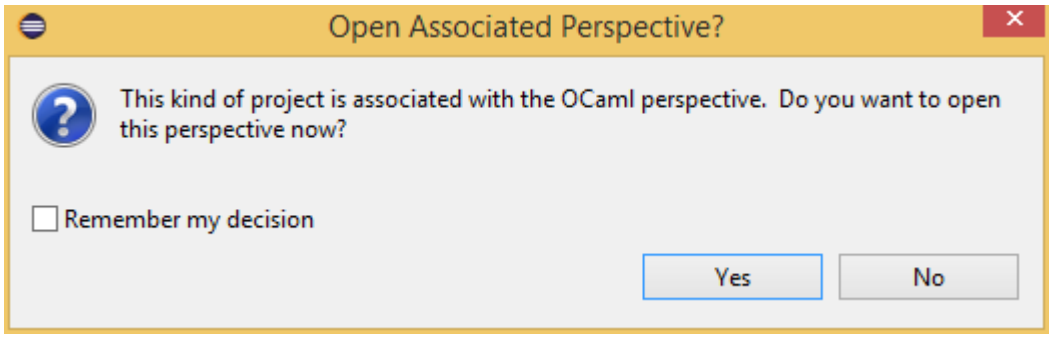


4. Digite o nome do projeto.



5. Clique em `Finish`.

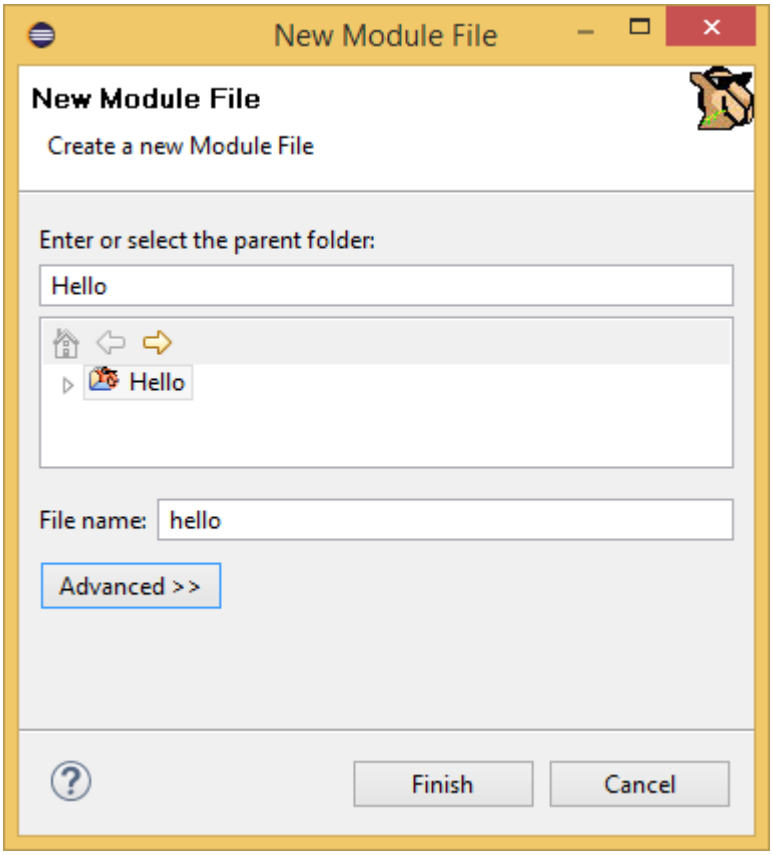
6. Aceite abrir a perspectiva OCaml associada ao projeto clicando em `Yes`.



7. Clique com o botão direito do mouse sobre o nome do projeto na visão `Navigator`. Será exibido um menu.

8. Escolha a opção `New`→`Module`. Será exibido o assistente `New Module File`.

9. No campo `File name` digite o nome do arquivo `hello`.



10. Clique em `Finish`. O módulo será adicionado ao projeto e aberto no editor de texto.

11. Vamos agora configurar algumas propriedades do projeto. Clique novamente com o botão direito do mouse sobre o nome do projeto na visão `Navigator`. Será exibido um menu.

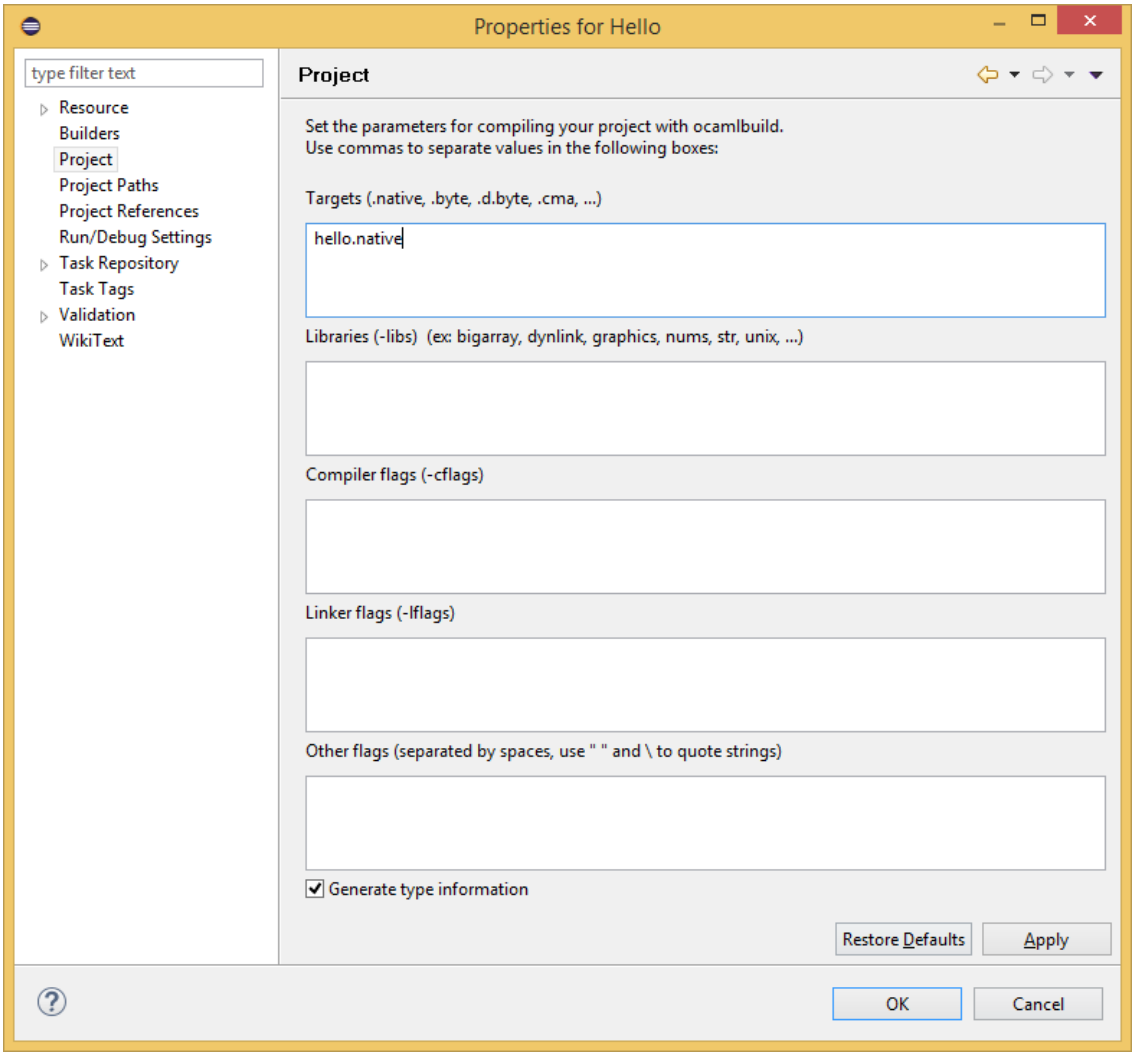
12. Escolha a opção `Properties`. Será exibido o assistente `Properties for Hello`.

13. Clique na opção `Project`.

14. No campo `Targets (.native, .byte, .d.byte, .cma, ...)` digite os alvos a serem construídos pelo projeto. Algumas possibilidades são:

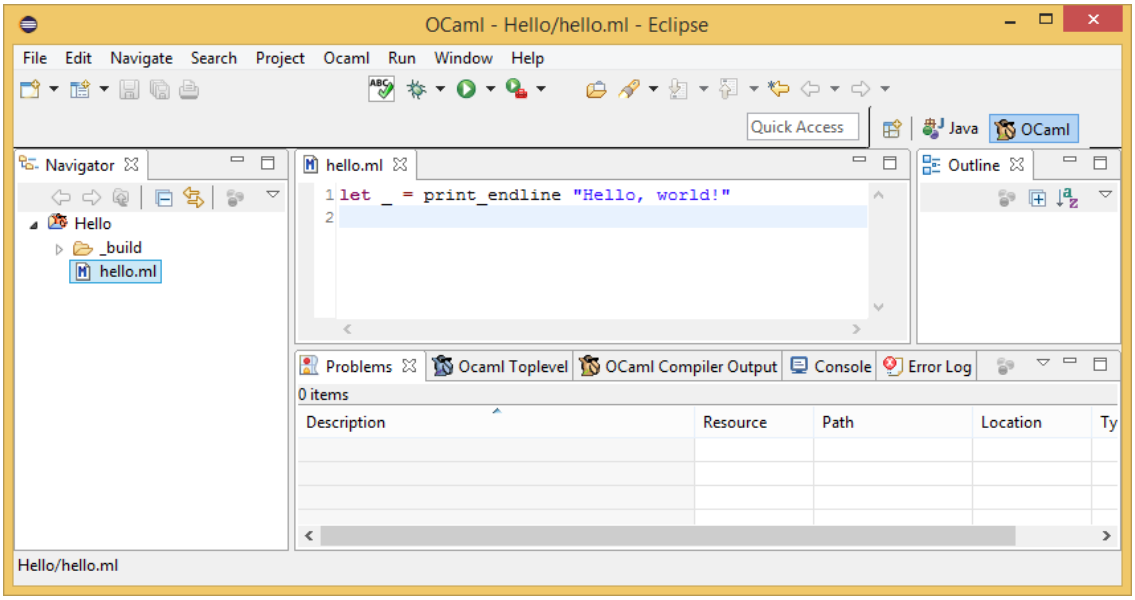
    - `hello.native`: programa executável nativo
    - `hello.byte`: programa executável em bytecodes
    - `hello.d.byte`: programa executável em bytecodes com código adicional para depuração

    Escolha `hello.native`.



15. Clique em `OK`.

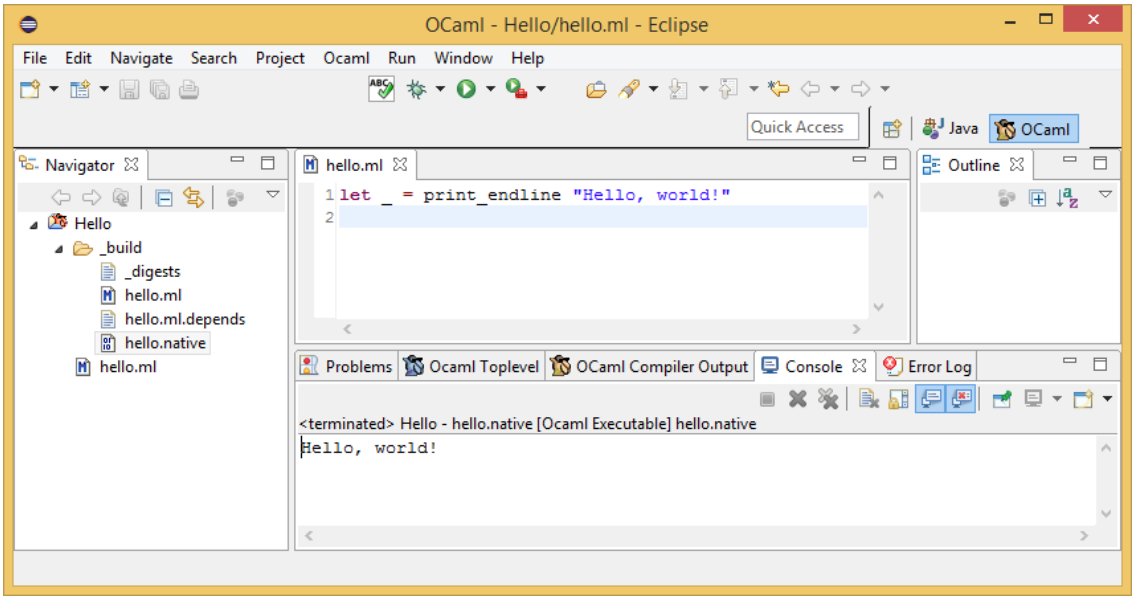
16. Digite o texto do programa fonte no arquivo do módulo usando o editor de texto.



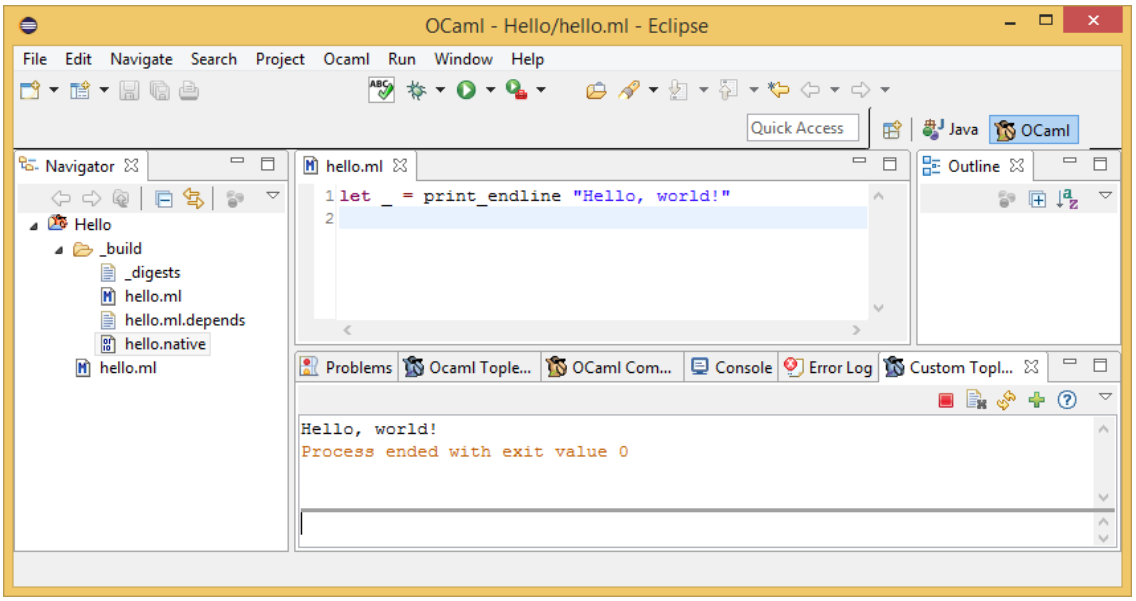
17. Salve o arquivo (menu `File`→`Save`). O módulo será automaticamente compilado.

18. Para executar o programa, clique com o botão direito do mouse sobre o nome do módulo executável na visão `Navigator`: `Hello`→`_build`→`hello.native`. Será exibido um menu.

19. Escolha a opção `Properties`→`Run As`→`Ocaml Executable` para executar o módulo na visão `Console`.



ou escolha a opção `Properties→Run As→Ocaml Toplevel` para carregar o módulo na visão `Custom Toplevel`.



1.4 Atividades

Tarefa 1.1: Produto de três números inteiros

Escreva um programa que solicita ao usuário três números inteiros, lê os números, e calcula e exibe o produto dos números.

Tarefa 1.2: Conversão de temperatura

Escreva um programa em OCaml que solicita ao usuário uma temperatura na escala Fahrenheit, lê esta temperatura, converte-a para a escala Celsius, e exibe o resultado.

Para fazer a conversão, defina uma função `celsius : float -> float` que recebe a temperatura na escala Fahrenheit e resulta na temperatura correspondente na escala Celsius. Use a seguinte equação para a conversão:

$$C = \frac{5}{9} \times (F - 32)$$

onde F é a temperatura na escala Fahrenheit e C é a temperatura na escala Celsius.

Use a função `celsius` na definição de uma função principal chamada `main`.

Tarefa 1.3: Peso ideal

Escrever um programa em OCaml que recebe a altura e o sexo de uma pessoa e calcula e mostra o seu peso ideal, utilizando as fórmulas constantes na tabela a seguir.

sexo	**peso ideal**
masculino	$72.7 \times h - 58$
feminino	$62.1 \times h - 44.7$

onde h é a altura da pessoa.

Tarefa 1.4: Situação de um aluno

Faça um programa que receba três notas de um aluno, e calcule e mostre a média aritmética das notas e a situação do aluno, dada pela tabela a seguir.

média das notas	**situação**
menor que 3	reprovado
entre 3 (inclusive) e 7	exame especial
acima de 7 (inclusive)	aprovado

Tarefa 1.5: Raízes da equação do segundo grau

Faça um programa que leia os coeficientes de uma equação do segundo grau e calcule e mostre suas raízes reais, caso existam.

Tarefa 1.6: Linha de crédito

A prefeitura de Contagem abriu uma linha de crédito para os funcionários estatutários. O valor máximo da prestação não poderá ultrapassar 30% do salário bruto.

Fazer um programa que permita entrar com o salário bruto e o valor da prestação, e informar se o empréstimo pode ou não ser concedido.

Tarefa 1.7: Classe eleitoral

Crie um programa que leia a idade de uma pessoa e informe a sua classe eleitoral:

não eleitor abaixo de 16 anos;

eleitor obrigatório entre 18 (inclusive) e 65 anos;

eleitor facultativo de 16 até 18 anos e acima de 65 anos (inclusive).

Tarefa 1.8: Impostos

Faça um programa que apresente o menu a seguir, permita ao usuário escolher a opção desejada, receba os dados necessários para executar a operação, e mostre o resultado.

```
--------------------------------
Opções:
--------------------------------
1. Imposto
2. Novo salário
3. Classificação
--------------------------------
Digite a opção desejada:
```

Verifique a possibilidade de opção inválida.

Na **opção 1** receba o salário de um funcionário, calcule e mostre o valor do imposto sobre o salário usando as regras a seguir:

salário	**taxa de imposto**
Abaixo de R$500,00	5%
De R$500,00 a R$850,00	15%
Acima de R$850,00	10%

Na **opção 2** receba o salário de um funcionário, calcule e mostre o valor do novo salário, usando as regras a seguir:

salário	**aumento**
Acima de R$1.500,00	R$25,00
De R$750,00 (inclusive) a R$1.500,00 (inclusive)	R$50,00
De R$450,00 (inclusive) a R$750,00	R$75,00
Abaixo de R$450,00	R$100,00

Na **opção 3** receba o salário de um funcionário e mostre sua classificação usando a tabela a seguir:

salário	**classificação**
Até R$750,00 (inclusive)	mal remunerado
Acima de R$750,00	bem remunerado

Tarefa 1.9: Terno pitagórico

Em Matemática um **terno pitagórico** (ou trio pitagórico, ou ainda tripla pitagórica) é formado por três números naturais a, b e c tais que $a^2 + b^2 = c^2$. O nome vem do *teorema de Pitágoras* que afirma que se as medidas dos lados de um triângulo retângulo são números inteiros, então elas formam um terno pitagórico.

Codifique um programa que leia três números naturais e verifique se eles formam um terno pitagórico.

Tarefa 1.10: Palíndromes

Escreva um programa em OCaml que solicita ao usuário para digitar uma frase, lê a frase (uma linha) da entrada padrão e testa se a string lida é uma palíndrome, exibindo uma mensagem apropriada.

1.5 Soluções

Tarefa 1.1 on page 1-12: Solução

```
(* Produto de três números inteiros *)

let main () =
  try
    print_string "Digite um número inteiro: ";
    let x = read_int () in
    print_string "Digite outro número inteiro: ";
    let y = read_int () in
    print_string "Digite outro número inteiro: ";
    let z = read_int () in
    let p = x * y * z in
    print_newline ();
    print_string "Produto dos números digitados: ";
    print_int p;
    print_newline ()
  with
    Failure "int_of_string" -> print_endline "Dados incorretos"

let _ = main ()
```

Tarefa 1.2 on page 1-12: Solução

```
(* conversão de temperaturas *)

open Printf

let celsius f =
  5.0 /. 9.0 *. (f -. 32.0)

let main () =
  print_string "Digite a temperatura em F: ";
  let temp = read_float () in
  printf "%.2f F equivale a %.2f F\n" temp (celsius temp)

let _ = main ()
```

Tarefa 1.3 on page 1-13: Solução

```
(* Cálculo do peso ideal *)

let main () =
  try
    print_endline "peso ideal";
    print_endline "---------";
    print_string "altura (em metros): ";
    let altura = read_float () in
    print_string "sexo (m/f): ";
    let sexo = String.lowercase (String.trim (read_line ())) in
    if sexo = "m" then
      Printf.printf "peso ideal: %.2f Kg\n" (72.7 *. altura -. 58.0)
    else if sexo = "f" then
      Printf.printf "peso ideal: %.2f Kg\n" (62.1 *. altura -. 44.7)
    else
      print_endline "sexo inválido"
  with
    Failure _ -> print_endline "dados inválidos"

let _ = main ()
```

Tarefa 1.4 on page 1-13: Solução

```
(* Cálculo do peso ideal *)

let main () =
  try
    print_endline "peso ideal";
    print_endline "---------";
    print_string "altura (em metros): ";
    let altura = read_float () in
    print_string "sexo (m/f): ";
    let sexo = String.lowercase (String.trim (read_line ())) in
    if sexo = "m" then
      Printf.printf "peso ideal: %.2f Kg\n" (72.7 *. altura -. 58.0)
    else if sexo = "f" then
      Printf.printf "peso ideal: %.2f Kg\n" (62.1 *. altura -. 44.7)
    else
      print_endline "sexo inválido"
  with
    Failure _ -> print_endline "dados inválidos"

let _ = main ()
```

Tarefa 1.5 on page 1-13: Solução

```
(* solução da equação do segundo grau *)

let main () =
  try
    print_endline "digite os coeficientes da equação do segundo grau:";
    print_string "a: ";
    let a = read_float () in
    if a = 0.0 then
      print_endline "a equação não é do segundo grau"
    else
      begin
        print_string "b: ";
        let b = read_float () in
        print_string "c: ";
        let c = read_float () in
        let delta = b *. b -. 4.0 *. a *. c in
        if delta < 0.0 then
          print_endline "a equação não possui raízes reais"
        else if delta = 0.0 then
          let r = -. b /. (2.0 *. a) in
          Printf.printf "raíz: %f\n" r
        else
          let r1 = (-. b +. sqrt delta) /. (2.0 *. a)
          and r2 = (-. b -. sqrt delta) /. (2.0 *. a) in
          Printf.printf "raízes: %f e %f\n" r1 r2
      end
  with
    Failure "float_of_string" -> print_endline "dados inválidos"

let _ = main ()
```

Tarefa 1.6 on page 1-13: Solução

```
(* linha de crédito *)

let _ =
  try
    print_endline "Linha de crédito";
    print_endline "=================";
    print_string "salário bruto: ";
    let sal_bruto = read_float () in
    print_string "valor da prestação: ";
    let valor_prestacao = read_float () in
    if valor_prestacao > 0.3 *. sal_bruto then
      print_endline "o empréstimo não pode ser concedido"
    else
      print_endline "o empréstimo pode ser concedido"
  with
    Failure "float_of_string" -> print_endline "dados inválidos"
```

Tarefa 1.7 on page 1-13: Solução

```
let _ =
  try
    print_endline "Classe eleitoral";
    print_endline "================";
    print_string "idade do eleitor: ";
    let idade = read_int () in
    if idade < 16 then
      print_endline "não eleitor"
    else if idade >= 18 && idade < 65 then
      print_endline "eleitor obrigatório"
    else
      print_endline "eleitor facultativo"
  with
    Failure "int_of_string" -> print_endline "dados inválidos"
```

Tarefa 1.8 on page 1-14: Solução

```
let imposto () =
  print_endline "Imposto";
  print_string "salário: ";
  let salario = read_float () in
  let taxa =
    if salario < 500.0 then
      5.0
    else if salario < 850.0 then
      10.0
    else
      15.0
  in
  let imp = salario *. taxa /. 100.0 in
  Printf.printf "imposto: %.2f\n" imp

let novo_salario () =
  print_endline "Novo salário";
  print_string "salário: ";
  let salario = read_float () in
  let aumento =
    if salario > 1500.0 then
      25.0
    else if salario >= 750.0 then
      50.0
    else if salario >= 450.0 then
      75.0
    else
      100.0
  in
  Printf.printf "novo salário: %.2f\n" (salario +. aumento)

let classificacao () =
  print_endline "Classificação";
  print_string "salário: ";
  let salario = read_float () in
  if salario <= 750.0 then
    print_endline "mal remunerado"
  else
    print_endline "bem remunerado"

let main () =
  try
    print_endline "-----------------------------";
    print_endline "Opções:";
    print_endline "-----------------------------";
    print_endline "1. Imposto";
    print_endline "2. Novo salário";
    print_endline "3. Classificação";
    print_endline "-----------------------------";
    print_string "Digite a opção desejada: ";
    let opcao = read_int () in
    match opcao with
    | 1 -> imposto ()
    | 2 -> novo_salario ()
    | 3 -> classificacao ()
    | _ -> print_endline "Opção inválida"
  with
    Failure _ -> print_endline "dados inválidos"

let _ = main ()
```

Tarefa 1.9 on page 1-14: Solução

```
let _ =
  try
    print_endline "Verificação de ternos pitagóricos";
    print_string "digite o primeiro número: ";
    let a = read_int () in
    print_string "digite o segundo número: ";
    let b = read_int () in
    print_string "digite o terceiro número: ";
    let c = read_int () in
    if a*a = b*b + c*c || b*b = a*a + c*c || c*c = a*a + b*b then
      print_endline "Os números formam um terno pitagórico"
    else
      print_endline "Os números não formam um terno pitagórico"
  with
    Failure "int_of_string" -> print_endline "dados inválidos"
```

Tarefa 1.10 on page 1-14: Solução

```
let palindrome str =
  let n = String.length str in
  let rec loop i =
    if i = n/2 then
      true
    else
      if str.[i] = str.[n - i - 1] then
        loop (i+1)
      else
        false
  in
  loop 0

let main () =
  print_endline "Digite uma frase:";
  let str = read_line () in
  if palindrome str then
    print_endline "é palíndrome"
  else
    print_endline "não é palíndrome"

let _ = main ()
```

2 Núcleo da Linguagem OCaml

Resumo

Esta parte do manual[1] é um tutorial introdutório para a linguagem OCaml. Uma boa familiaridade com programação em linguagens convencionais (por exemplo, Pascal ou C) é assumido, mas não é necessária uma exposição prévia a linguagens funcionais. O presente capítulo apresenta a linguagem principal.

Sumário

2.1 Básicos

Para esta visão geral do OCaml, usamos o sistema interativo, que é iniciado executando `ocaml` do shell do sistema operacional, ou iniciando o aplicativo `OCamlwin.exe` no Windows. Este tutorial é apresentado como a transcrição de uma sessão com o sistema interativo: linhas que começam com # representam a entrada do usuário; as respostas do sistema são impressos abaixo, sem um # no início da linha.

Sob o sistema interativo, as frases do OCaml que o usuário digita devem ser terminadas por `;;` em resposta ao prompt `#`, e o sistema compila-as em tempo real, executa-as, e exibe o resultado da avaliação. Frases são expressões simples, ou definições `let` de identificadores (valores ou funções).

```
# 1+2*3;;
- : int = 7

# let pi = 4.0 *. atan 1.0;;
val pi : float = 3.14159265358979312

# let square x = x *. x;;
val square : float -> float = <fun>

# square (sin pi) +. square (cos pi);;
- : float = 1.
```

O sistema OCaml calcula tanto o valor como o tipo para cada frase. Até mesmo os parâmetros de função não necessitam de declaração explícita do tipo: o sistema infere seus tipos a partir de seu uso na função. Observe também que números inteiros e números de ponto flutuante são tipos distintos, com operadores distintos: `+` e `*` operam em números inteiros, mas `+.` e `*.` operam em números em ponto flututante.

```
# 1.0 * 2;;
Error: This expression has type float but an expression was expected of type
         int
```

Funções recursivas são definidas com a declaração `let rec`:

[1] Tradução/adaptação do manual da linguagem OCaml, disponível em `http://caml.inria.fr/pub/docs/manual-ocaml/`.

```
# let rec fib n =
    if n < 2 then n else fib (n-1) + fib (n-2);;
val fib : int -> int = <fun>

# fib 10;;
- : int = 55
```

2.2 Tipos de dados

Além de números inteiros e números em ponto flutuante, OCaml oferece os tipos de dados básicos usuais: booleanos, caracteres e strings (sequências de caracteres).

```
# (1 < 2) = false;;
- : bool = false

# 'a';;
- : char = 'a'

# "Hello world";;
- : string = "Hello world"
```

Estruturas de dados predefinidas incluem tuplas, matrizes e listas. Mecanismos gerais para definir suas próprias estruturas de dados também são fornecidos. Eles serão abordados em mais detalhes mais tarde; por agora, nós nos concentramos em listas. Listas são dadas (como uma extensão sintática) por uma sequência entre colchetes de elementos separados por ponto e vírgula, ou construída a partir de uma lista vazia [] (*nil*), adicionando elementos na frente usando o operador `::` (*cons*).

```
# let l = ["is"; "a"; "tale"; "told"; "etc."];;
val l : string list = ["is"; "a"; "tale"; "told"; "etc."]

# "Life" :: l;;
- : string list = ["Life"; "is"; "a"; "tale"; "told"; "etc."]
```

Tal como acontece com todas as outras estruturas de dados em OCaml, listas não precisam ser explicitamente alocadas e desalocadas da memória: todo o gerenciamento de memória é totalmente automático em OCaml. Da mesma forma, não há manipulação explícita dos ponteiros: o compilador OCaml silenciosamente introduz ponteiros quando necessário.

Tal como acontece com a maioria das estruturas de dados em OCaml, a inspeção e a decomposição de listas é realizada por casamento de padrão (*pattern-matching*). Padrões para lista têm a mesma forma de expressões lista, com identificadores representando partes não especificadas da lista. Como um exemplo, aqui é feita a ordenação por inserção de uma lista:

```
# let rec sort lst =
    match lst with
      [] -> []
    | head :: tail -> insert head (sort tail)
  and insert elt lst =
    match lst with
      [] -> [elt]
    | head :: tail -> if elt <= head then elt :: lst else head :: insert elt tail
  ;;
val sort : 'a list -> 'a list = <fun>
val insert : 'a -> 'a list -> 'a list = <fun>

# sort l;;
- : string list = ["a"; "etc."; "is"; "tale"; "told"]
```

O tipo inferido pora `sort`, `'a list -> 'a list`, significa que `sort` pode realmente ser aplicada a listas de qualquer tipo, e retorna uma lista do mesmo tipo. O tipo `'a` é uma variável do tipo, e representa qualquer

tipo. A razão pela qual a `sort` pode ser aplicada a qualquer tipo de listas é que as comparações =, <=, etc., são polimórficas em OCaml: elas operam com quaisquer dois valores do mesmo tipo. Isso faz com que `sort` também seja polimórfica sobre todos os tipos de lista.

```
# sort [6;2;5;3];;
- : int list = [2; 3; 5; 6]

# sort [3.14; 2.718];;
- : float list = [2.718; 3.14]
```

A função de classificação acima não modifica a sua lista de entrada: ela constrói e retorna uma nova lista contendo os mesmos elementos que a lista de entrada, em ordem crescente. Não há realmente nenhuma maneira em OCaml para modificar uma lista uma vez que ela é construída: dizemos que as listas são estruturas de dados imutáveis. A maioria das estruturas de dados em OCaml são imutáveis, mas algumas (sobretudo as matrizes) são mutáveis, o que significa que elas podem ser modificadas em qualquer momento.

2.3 Funções como valores

OCaml é uma linguagem funcional: funções no sentido matemático completo são suportadas e podem ser passados livremente como qualquer outro dado. Por exemplo, aqui temos uma função `deriv` que recebe qualquer função de **`float`** em **`float`** como argumento e retorna uma aproximação da sua função derivada:

```
# let deriv f dx = function x -> (f (x +. dx) -. f x) /. dx;;
val deriv : (float -> float) -> float -> float -> float = <fun>

# let sin' = deriv sin 1e-6;;
val sin' : float -> float = <fun>

# sin' pi;;
- : float = -1.00000000013961143
```

Mesmo a composição de função é definível:

```
# let compose f g = function x -> f (g x);;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

# let cos2 = compose square cos;;
val cos2 : float -> float = <fun>
```

As funções que utilizam outras funções como argumentos são chamadas de *funcionais*, ou *funções de ordem superior*. Funcionais são especialmente úteis para fornecer iteradores ou operações genéricas similares sobre uma estrutura de dados. Por exemplo, a biblioteca OCaml padrão proporciona um funcional **`List`**`.map` que aplica uma determinada função a cada elemento de uma lista e devolve a lista dos resultados:

```
# List.map (function n -> n * 2 + 1) [0;1;2;3;4];;
- : int list = [1; 3; 5; 7; 9]
```

Este funcional, juntamente com uma série de outros funcionais para listas e matrizes, é pré-definido, porque muitas vezes é útil, mas não há nada mágico com ele: ele pode ser facilmente definido da seguinte forma:

```
# let rec map f l =
    match l with
      [] -> []
    | hd :: tl -> f hd :: map f tl;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

2.4 Registros e variantes

Estruturas de dados definidas pelo usuário incluem registros e variantes. Ambos são definidos com a declaração **type**. Aqui, nós declaramos um tipo de registro para representar números racionais.

```
# type ratio = {num: int; denom: int};;
type ratio = { num : int; denom : int; }

# let add_ratio r1 r2 =
    {num = r1.num * r2.denom + r2.num * r1.denom;
     denom = r1.denom * r2.denom};;
val add_ratio : ratio -> ratio -> ratio = <fun>

# add_ratio {num=1; denom=3} {num=2; denom=5};;
- : ratio = {num = 11; denom = 15}
```

A declaração de um tipo de variante lista todas as formas possíveis para valores desse tipo. Cada caso é identificado por um nome, chamado de construtor, que serve tanto para a construção de valores do tipo variante, como para inspecioná-los por casamento de padrão. Nomes de construtores são capitalizados para distingui-los dos nomes de variáveis (que devem começar com uma letra minúscula). Por exemplo, aqui está um tipo variante para fazer aritmética mista com inteiros e floats:

```
# type number = Int of int | Float of float | Error;;
type number = Int of int | Float of float | Error
```

Esta declaração expressa que um valor do tipo `number` é um inteiro, um número em ponto flutuante, ou a constante de erro**Error** que representa o resultado de uma operação inválida (por exemplo, uma divisão por zero).

Os tipos enumerados são um caso especial de tipos de variantes, em que todas as alternativas são constantes:

```
# type sign = Positive | Negative;;
type sign = Positive | Negative

# let sign_int n = if n >= 0 then Positive else Negative;;
val sign_int : int -> sign = <fun>
```

Para definir operações aritméticas para o tipo `number`, usamos casamento de padrão nos dois números envolvidos:

```
# let add_num n1 n2 =
    match (n1, n2) with
      (Int i1, Int i2) ->
        (* Check for overflow of integer addition *)
        if sign_int i1 = sign_int i2 && sign_int (i1 + i2) <> sign_int i1
        then Float(float i1 +. float i2)
        else Int(i1 + i2)
    | (Int i1, Float f2) -> Float(float i1 +. f2)
    | (Float f1, Int i2) -> Float(f1 +. float i2)
    | (Float f1, Float f2) -> Float(f1 +. f2)
    | (Error, _) -> Error
    | (_, Error) -> Error;;
val add_num : number -> number -> number = <fun>

# add_num (Int 123) (Float 3.14159);;
- : number = Float 126.14159
```

O uso mais comum dos tipos variantes é para descrever estruturas de dados recursivas. Considere, por exemplo, o tipo de árvores binárias:

```
# type 'a btree = Empty | Node of 'a * 'a btree * 'a btree;;
type 'a btree = Empty | Node of 'a * 'a btree * 'a btree
```

Esta definição é lida como segue: uma árvore binária contendo valores do tipo `'a` (um tipo arbitrário) é vazia, ou é um nó que contém um valor do tipo `'a` e duas sub-árvores contendo também os valores do tipo `'a`, ou seja, dois `'a btree`.

Operações em árvores binárias são naturalmente expressas como funções recursivas seguindo a mesma estrutura da própria definição do tipo. Por exemplo, aqui temos funções de pesquisa e inserção em árvores binárias ordenadas (elementos aumentam da esquerda para a direita):

```
# let rec member x btree =
    match btree with
      Empty -> false
    | Node(y, left, right) ->
        if x = y then true else
        if x < y then member x left else member x right;;
val member : 'a -> 'a btree -> bool = <fun>

# let rec insert x btree =
    match btree with
      Empty -> Node(x, Empty, Empty)
    | Node(y, left, right) ->
        if x <= y then Node(y, insert x left, right)
                  else Node(y, left, insert x right);;
val insert : 'a -> 'a btree -> 'a btree = <fun>
```

2.5 Características imperativas

Apesar de todos os exemplos até agora terem sido escritos em estilo puramente aplicativo, OCaml também é equipado com recursos imperativos completos. Isto inclui os laços de repetição **`while`** e **`for`** habituais, bem como estruturas de dados mutáveis, tais como matrizes. Matrizes ou são dadas em uma notação estendida onde os elementos são escritos entre [| e |], ou alocadas e inicializadas com a função , e então preenchidas mais tarde por atribuições. Por exemplo, a função abaixo soma dois vetores (representados como matrizes de float).

```
# let add_vect v1 v2 =
    let len = min (Array.length v1) (Array.length v2) in
    let res = Array.create len 0.0 in
    for i = 0 to len - 1 do
      res.(i) <- v1.(i) +. v2.(i)
    done;
    res;;
Warning 3: deprecated: Array.create
Use Array.make instead.
val add_vect : float array -> float array -> float array = <fun>

# add_vect [| 1.0; 2.0 |] [| 3.0; 4.0 |];;
- : float array = [|4.; 6.|]
```

Campos de registro também podem ser modificados por atribuição, desde que sejam declarados mutáveis na definição do tipo do registro:

```
# type mutable_point = { mutable x: float; mutable y: float };;
type mutable_point = { mutable x : float; mutable y : float; }

# let translate p dx dy =
    p.x <- p.x +. dx; p.y <- p.y +. dy;;
val translate : mutable_point -> float -> float -> unit = <fun>

# let mypoint = { x = 0.0; y = 0.0 };;
val mypoint : mutable_point = {x = 0.; y = 0.}

# translate mypoint 1.0 2.0;;
- : unit = ()
```

```
# mypoint;;
- : mutable_point = {x = 1.; y = 2.}
```

OCaml não tem nenhuma noção de variáveis atualizáveis como nas linguagens convencionais – identificadores cujo valor atual pode ser alterado por atribuição. (A declaração não é uma atribuição, mas apenas introduz um novo identificador com um novo escopo.) No entanto, a biblioteca padrão fornece referências, que são células mutáveis (ou matrizes de um elemento), com os operadores para acessar o conteúdo atual da referência, e para atribuir um novo conteúdo à célula. As variáveis podem, então, ser emuladas por declarações de uma referência. Por exemplo, aqui está uma função para ordenação de matrizes por inserção local:

```
# let insertion_sort a =
    for i = 1 to Array.length a - 1 do
      let val_i = a.(i) in
      let j = ref i in
      while !j > 0 && val_i < a.(!j - 1) do
        a.(!j) <- a.(!j - 1);
        j := !j - 1
      done;
      a.(!j) <- val_i
    done;;
val insertion_sort : 'a array -> unit = <fun>
```

Referências também são úteis para escrever funções que mantêm um estado atual entre duas chamadas para a função. Por exemplo, a sequência do gerador de números pseudo-aleatórios mantém o último número retornado em uma referência:

```
# let current_rand = ref 0;;
val current_rand : int ref = {contents = 0}

# let random () =
    current_rand := !current_rand * 25713 + 1345;
    !current_rand;;
val random : unit -> int = <fun>
```

Novamente, não há nada de mágico, com referências: elas são implementadas como um registro mutável de campo único, como se segue.

```
# type 'a ref = { mutable contents: 'a };;
type 'a ref = { mutable contents : 'a; }

# let ( ! ) r = r.contents;;
val ( ! ) : 'a ref -> 'a = <fun>

# let ( := ) r newval = r.contents <- newval;;
val ( := ) : 'a ref -> 'a -> unit = <fun>
```

Em alguns casos especiais, pode ser necessário armazenar uma função polimórfica em uma estrutura de dados, mantendo o seu polimorfismo. Sem anotações de tipo fornecido pelo usuário, isso não é permitido, pois o polimorfismo só pode ser introduzido em um nível global. No entanto, você pode dar tipos explicitamente polimórficas para campos de registro.

```
# type idref = { mutable id: 'a. 'a -> 'a };;
type idref = { mutable id : 'a. 'a -> 'a; }

# let r = {id = fun x -> x};;
val r : idref = {id = <fun>}

# let g s = (s.id 1, s.id true);;
val g : idref -> int * bool = <fun>

# r.id <- (fun x -> print_string "called id\n"; x);;
```

```
- : unit = ()

# g r;;
called id
called id
- : int * bool = (1, true)
```

2.6 Exceções

OCaml provê exceções para sinalização e manipulação de condições excepcionais. Exceções também podem ser utilizadas como uma estrutura de controle não-local de uso geral. Exceções são declaradas com a construção , e levantadas (sinalizadas) com o operador . Por exemplo, a função abaixo para tirar a cabeça de uma lista usa uma exceção para sinalizar o caso em que uma lista vazia é dada.

```
# exception Empty_list;;
exception Empty_list

# let head l =
    match l with
      [] -> raise Empty_list
    | hd :: tl -> hd;;
val head : 'a list -> 'a = <fun>

# head [1;2];;
- : int = 1

# head [];;
Exception: Empty_list.
```

As exceções são utilizadas na biblioteca padrão para sinalizar casos em que as funções da biblioteca não podem completar normalmente. Por exemplo, a função , que retorna os dados associados a uma determinada chave em uma lista de pares (lista de associações), levanta a exceção predefinida quando a chave não aparece na lista:

```
# List.assoc 1 [(0, "zero"); (1, "one")];;
- : string = "one"

# List.assoc 2 [(0, "zero"); (1, "one")];;
Exception: Not_found.
```

Exceções podem ser capturadas com a construção :

```
# let name_of_binary_digit digit =
    try
      List.assoc digit [0, "zero"; 1, "one"]
    with Not_found ->
      "not a binary digit";;
val name_of_binary_digit : int -> string = <fun>

# name_of_binary_digit 0;;
- : string = "zero"

# name_of_binary_digit (-1);;
- : string = "not a binary digit"
```

A parte é na verdade um casamento de padrão regular sobre o valor de exceção. Assim, várias exceções podem ser capturadas por uma construção . Além disso, a finalização pode ser realizada pela captura de todas as exceções, e então realizando a finalização, para em seguida levantar novamente a exceção:

```
# let temporarily_set_reference ref newval funct =
    let oldval = !ref in
```

```
    try
      ref := newval;
      let res = funct () in
      ref := oldval;
      res
    with x ->
      ref := oldval;
      raise x;;
val temporarily_set_reference : 'a ref -> 'a -> (unit -> 'b) -> 'b = <fun>
```

2.7 Processamento simbólico de expressões

Nós terminamos esta introdução com um exemplo representativo mais completo do uso de OCaml para processamento simbólico: manipulações formais de expressões aritméticas contendo variáveis. O seguinte tipo de variante descreve as expressões que devemos manipular:

```
# type expression =
      Const of float
    | Var of string
    | Sum of expression * expression    (* e1 + e2 *)
    | Diff of expression * expression   (* e1 - e2 *)
    | Prod of expression * expression   (* e1 * e2 *)
    | Quot of expression * expression   (* e1 / e2 *)
  ;;
type expression =
    Const of float
  | Var of string
  | Sum of expression * expression
  | Diff of expression * expression
  | Prod of expression * expression
  | Quot of expression * expression
```

Nós primeiro definimos uma função para calcular uma expressão dado um ambiente que mapeia nomes de variáveis para os seus valores. Para simplicidade, o ambiente é representado como uma lista de associação.

```
# exception Unbound_variable of string;;
exception Unbound_variable of string

# let rec eval env exp =
    match exp with
      Const c -> c
    | Var v ->
        (try List.assoc v env with Not_found -> raise (Unbound_variable v))
    | Sum(f, g) -> eval env f +. eval env g
    | Diff(f, g) -> eval env f -. eval env g
    | Prod(f, g) -> eval env f *. eval env g
    | Quot(f, g) -> eval env f /. eval env g;;
val eval : (string * float) list -> expression -> float = <fun>

# eval [("x", 1.0); ("y", 3.14)] (Prod(Sum(Var "x", Const 2.0), Var "y"));;
- : float = 9.42
```

Agora, para um processamento simbólico real, se define a derivada de uma expressão em relação a uma variável :

```
# let rec deriv exp dv =
    match exp with
      Const c -> Const 0.0
    | Var v -> if v = dv then Const 1.0 else Const 0.0
    | Sum(f, g) -> Sum(deriv f dv, deriv g dv)
    | Diff(f, g) -> Diff(deriv f dv, deriv g dv)
```

```
    | Prod(f, g) -> Sum(Prod(f, deriv g dv), Prod(deriv f dv, g))
    | Quot(f, g) -> Quot(Diff(Prod(deriv f dv, g), Prod(f, deriv g dv)),
                         Prod(g, g))
  ;;
val deriv : expression -> string -> expression = <fun>

# deriv (Quot(Const 1.0, Var "x")) "x";;
- : expression =
Quot (Diff (Prod (Const 0., Var "x"), Prod (Const 1., Const 1.)),
 Prod (Var "x", Var "x"))
```

2.8 Exibição e análise sintática

Como mostrado nos exemplos acima, a representação interna (também chamado de sintaxe abstrata) de expressões rapidamente se torna difícil de ler e escrever à medida que as expressões se tornam maiores. Precisamos de um *printer* e um *parser* (analisador sintático) para converter entre a sintaxe abstrata e a sintaxe concreta, o que no caso das expressões é a notação algébrica conhecida (por exemplo, `2 * x + 1`).

Para a função de impressão, devemos levar em conta as regras de precedência habituais (por exemplo * liga mais fortemente do que +) para evitar a impressão de parênteses desnecessários. Para isso, mantemos a precedência do operador atual e imprimimos parênteses em torno do operador só se a sua precedência é menor que a precedência atual.

```
# let print_expr exp =
    (* Local function definitions *)
    let open_paren prec op_prec =
      if prec > op_prec then print_string "(" in
    let close_paren prec op_prec =
      if prec > op_prec then print_string ")" in
    let rec print prec exp =     (* prec is the current precedence *)
      match exp with
        Const c -> print_float c
      | Var v -> print_string v
      | Sum(f, g) ->
          open_paren prec 0;
          print 0 f; print_string " + "; print 0 g;
          close_paren prec 0
      | Diff(f, g) ->
          open_paren prec 0;
          print 0 f; print_string " - "; print 1 g;
          close_paren prec 0
      | Prod(f, g) ->
          open_paren prec 2;
          print 2 f; print_string " * "; print 2 g;
          close_paren prec 2
      | Quot(f, g) ->
          open_paren prec 2;
          print 2 f; print_string " / "; print 3 g;
          close_paren prec 2
    in print 0 exp;;
val print_expr : expression -> unit = <fun>

# let e = Sum(Prod(Const 2.0, Var "x"), Const 1.0);;
val e : expression = Sum (Prod (Const 2., Var "x"), Const 1.)

# print_expr e; print_newline ();;
2. * x + 1.
- : unit = ()

# print_expr (deriv e "x"); print_newline ();;
2. * 1. + 0. * x + 0.
- : unit = ()
```

2.9 Programas completos

Todos os exemplos dados até agora foram executadas no âmbito do sistema interativo. Códigos em OCaml também podem ser compilados separadamente e executados de forma não interativa utilizando os compiladores de lote `ocamlc` e `ocamlopt`. O código-fonte deve ser colocado em um arquivo com extensão `.ml`. Ele consiste de uma seqüência de frases, que serão avaliadas em tempo de execução em sua ordem de aparição no arquivo fonte. Ao contrário do modo interativo, tipos e valores não são impressos automaticamente; o programa deve chamar as funções de impressão explicitamente para produzir alguma saída. Aqui está um exemplo de programa autônomo para imprimir números de Fibonacci:

é uma matriz de strings contendo os parâmetros da linha de comando. é, assim, o primeiro parâmetro de linha de comando. O programa acima é compilado e executado com os seguintes comandos shell:

```
$ ocamlc -o fib fib.ml
$ ./fib 10
89
$ ./fib 20
10946
```

Programas OCaml autônomos mais complexo são geralmente compostos de vários arquivos fonte, e pode ser ligados com bibliotecas pré-compiladas. Capítulos 8 e 11 explicam como usar os compiladores `ocamlc` e `ocamlopt`. Recompilação de projetos OCaml multi-arquivo pode ser automatizado usando o gerenciador de compilação `ocamlbuild`, documentado no capítulo 18.