Wstęp do Programowania
potok funkcyjny

Marcin Kubica

2016/2017

Outline

1. Podstawy Ocamla
   - Dziedzina algorytmiczna Ocaml-a i BNF
   - Wyrażenia i definicje stałych
   - Definicje procedur i definicje lokalne

Elementy języka programowania

Elementy języka programowania

W każdym języku programowania (wyższego rzędu)
mamy następujące trzy elementy:

- dziedzina algorytmiczna — typy, stałe, operacje, relacje,
- metody konstrukcji — czyli jak z prostszych całości budować bardziej skomplikowane,
- metody abstrakcji — czyli jak złożone konstrukcje mogą być nazwane i dalej wykorzystywane tak, jak podstawowe elementy.

Czy metody konstrukcji i abstrakcji można oddzielić od siebie?
Tak!

Dziedzina algorytmiczna Ocamla

Typy	Stałe	Procedury (operacje i relacje)
`bool`	`true`, `false`	`||`, `&&`, `not`
`int`	$0, 1, 42, -2$	`+`, `-`, `*`, `/`, `mod`
`float`	$2.3, -3.4, 4.5E-7$	`+.`, `-.`, `*.`, `/.`
`char`	`'a'`, `'z'`	
`string`	`"Ala ma kota"`	`^`
wszystkie		$=, \leq, \geq, <, >, <>$

BNF

Definition (Notacja Backusa-Naura)

Meta-notacja do opisu składni języka (programowania).

- konstrukcje składniowe — $\langle\texttt{konstrukcja}\rangle$,
- reguły postaci:

$$\langle\texttt{konstrukcja}\rangle \ ::= \ \textit{możliwa postać}$$

BNF

Definition (Notacja Backusa-Naura)

Prawe strony reguł:

- `tekst, który pojawia się explicite` ,
- ⟨`nazwy konstrukcji składniowych`⟩
- alternatywy $\ldots | \ldots$,
- [tekst opcjonalny],
- nawiasy meta-notacji: $\{\ldots\}$,
- { zero lub więcej powtórzeń }*,
- { jedno lub więcej powtórzeń }$^+$.

BNF

Example

Składnia adresów pocztowych

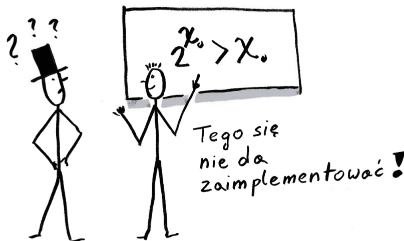
⟨adres⟩	::=	⟨adresat⟩ , ⟨adres lokalu⟩ ,
		⟨adres miasta⟩ [, ⟨adres kraju⟩]
⟨adresat⟩	::=	[W.P. | Sz.Pan.] ⟨napis⟩
⟨adres lokalu⟩	::=	⟨ulica⟩ { ⟨numer⟩ [⟨litera⟩] |
		⟨numer⟩ / ⟨numer⟩ } [m ⟨numer⟩ | / ⟨numer⟩]
⟨adres miasta⟩	::=	[⟨kod⟩] ⟨napis⟩
⟨adres kraju⟩	::=	⟨napis⟩
⟨kod⟩	::=	⟨cyfra⟩ ⟨cyfra⟩ - ⟨cyfra⟩ ⟨cyfra⟩ ⟨cyfra⟩
⟨cyfra⟩	::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨numer⟩	::=	⟨cyfra⟩ [⟨numer⟩]

Zdziwienie dnia

Zdziwienie dnia

Czy dla każdej funkcji istnieje implementująca ją procedura?
Nie!

Skończonych napisów, plików tekstowych i programów jest $\aleph_0$.
Funkcji, choćby $\mathbb{N} \to \{\texttt{true}, \texttt{false}\}$, jest continuum.
Programów jest nieskończenie razy mniej niż funkcji.



Przyrostowy tryb pracy

Przyrostowy tryb pracy:

- wczytanie fragmentu programu,
- kompilacja, dołączenie do już skompilowanych fragmentów,
- ew. wykonanie.

Jednostka kompilacji:

- fragment, który może być wczytany i skompilowany,
- zakończona `;;`

Wyrażenia

Example

```
42;;
- : int = 42

36 + 6;;
- : int = 42

3 * 14;;
- : int = 42

100 - 58;;
- : int = 42
```

Wyrażenia

Example

```
1 * 2 * 3 * 4 * 5 - ((6 + 7) * 8 * 9 / 12);;
- : int = 42

silnia 7 / silnia 5;;
- : int = 42

596.4 /. 14.2;;
- : float = 42.

"Ala" ^ " ma " ^ "kota";;
- : string = "Ala ma kota"
```

Budowa wyrażeń

Elementy składowe wyrażeń:

- stałe (liczbowe, napisy, lub nazwy stałych),
- zastosowanie procedur do argumentów (wywołanie),
- nawiasy.

+ czy ∗ to nazwy stałych, których wartościami
są procedury wbudowane w język programowania.

Składnia wyrażeń

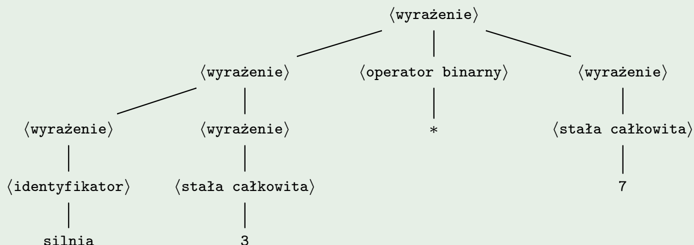
Definition

⟨jednostka kompilacji⟩ ::= ⟨wyrażenie⟩ | ...
⟨wyrażenie⟩ ::= (⟨wyrażenie⟩) | { ⟨wyrażenie⟩ }$^+$|
⟨op. unarny⟩ ⟨wyrażenie⟩ |
⟨wyrażenie⟩ ⟨op. binarny⟩ ⟨wyrażenie⟩ |
⟨identyfikator⟩ | ⟨stała całkowita⟩ |
⟨stała zmiennopozycyjna⟩ |
⟨stała napisowa⟩ | ...
⟨op. unarny⟩ ::= - | not | ...
⟨op. binarny⟩ ::= + | - | * | / | mod | +. | -. | *. | /. |
|| | && | = | < | <= | > | >= | ^ | ...

Obliczanie wartości wyrażeń

Drzewo składni wyrażenia

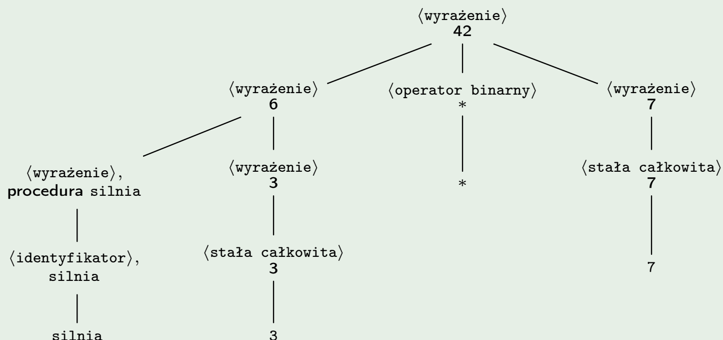
Example (silnia 3 * 7)



Obliczanie wartości wyrażeń

Drzewo składni udekorowane wartościami podwyrażeń

Example (`silnia 3 * 7`)



Składnia definicji stałych

Definicja stałej jest jedną z możliwych jednostek kompilacji.

Definition

$$\langle \texttt{jednostka kompilacji} \rangle ::= \langle \texttt{definicja} \rangle \,|\, \ldots$$
$$\langle \texttt{definicja} \rangle ::= \underline{\texttt{let}}\ \langle \texttt{identyfikator} \rangle\ \underline{\texttt{=}}\ \langle \texttt{wyrażenie} \rangle \,|\, \ldots$$

Definicja stałej jest *formą specjalną*.

Wykonanie definicji stałych

Definicje stałych są „wykonywane” następująco:

- nazwy stałych i związane z nimi wartości są pamiętane w *środowisku*,
- obliczenie wartości wyrażenia,
- rozszerzenie środowiska o identyfikator powiązany z wartością wyrażenia,
- *przysłanianie* istniejących definicji.

Uwaga!

To jest uproszczony model.

Definicje stałych — przykłady

Example

```
let a = 4;;
val a : int = 4

let b = 5 * a;;
val b : int = 20

let pi = 3.14;;
val pi : float = 3.14

pi *. 4.0 *. 4.0;;
val - : float = 50.24

let a = "Ala";;
val a : string = "Ala"
```

Definicje równoczesne

Możliwe jest równoczesne definiowanie wielu stałych.

Definition

$$\langle \texttt{definicja} \rangle ::= \underline{\texttt{let}}\ \langle \texttt{identyfikator} \rangle\ \underline{\texttt{=}}\ \langle \texttt{wyrażenie} \rangle \\ \{\ \underline{\texttt{and}}\ \langle \texttt{identyfikator} \rangle\ \underline{\texttt{=}}\ \langle \texttt{wyrażenie} \rangle\ \}^*$$

Najpierw obliczane są wszystkie wyrażenia.
Następnie ich wartości są przypisywane identyfikatorom.

Definicje równoczesne

Example

```
let a = 4 and b = 5;;
val a : int = 4
val b : int = 5

let a = a + b and b = a * b;;
val a : int = 9
val b : int = 20
```

Wyrażenia warunkowe

Definition (Składnia wyrażeń `if-then-else`)

⟨`wyrażenie`⟩ ::= <u>`if`</u> ⟨`wyrażenie`⟩ <u>`then`</u> ⟨`wyrażenie`⟩ <u>`else`</u> ⟨`wyrażenie`⟩

- Wartością pierwszego wyrażenia musi być wartość logiczna.
- Pozostałe dwa wyrażenia muszą być tego samego (lub uzgadnialnego) typu.
- Najpierw obliczane jest pierwsze wyrażenie.
- W zależności od jego wartości, obliczane jest drugie lub trzecie wyrażenie.

Leniwość `if-then-else`

- `If-then-else` jest formą specjalną.
- `If-then-else` jest w pewnym sensie leniwe — obliczane jest tylko to, co niezbędne.

Example

```
let x = 0.0238095238095238082;;
val x : float = 0.0238095238095238082

if x = 0.0 then 0.0 else 1.0 /. x;;
- : float = 42.
```

Operatory logiczne

- `&&` — koniunkcja,
- `||` — alternatywa,
- `not` — negacja.

`&&` i `||` są leniwe.

```
 x && y ≡ if x then y else false
x || y ≡ if x then true else y
```

Są to formy specjalne.
Ale `not` jest już zwykłą procedurą.

Rodzaje konstrukcji językowych

Mamy tylko trzy podstawowe konstrukcje językowe:

- **użycie** wartości zdefiniowanej w środowisku, np. stałych lub procedur,
- **zastosowanie** procedur do argumentów, tzn. wywołanie procedury,
- **definiowanie** nowych wartości (abstrakcja), stałych, procedur, typów, ...

„Konstytucyjne" prawa procedur

Konstytucja programowania funkcyjnego

Procedury są obywatelami pierwszej kategorii,
czyli równie dobrymi wartościami, jak wszystkie inne.

Wynika stąd, że:

- można definiować nazwane procedury (stałe proceduralne),
- procedura może być wartością wyrażenia,
- procedury mogą być argumentami i wynikami procedur!

Definicje procedur

Definition

Składnia definicji procedur

$$\langle \texttt{definicja} \rangle ::= \underline{\texttt{let}}\ \langle \texttt{identyfikator} \rangle \{ \langle \texttt{identyfikator} \rangle \}^{+}\ \underline{=}\ \langle \texttt{wyrażenie} \rangle$$

```
let nazwa parametry = treść
```

Definicje procedur

Example

```
let abs x =
  if x < 0 then -x else x;;
val abs : int -> int = <fun>

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

let pow r = pi *. (square r);;
val pow : float -> float = <fun>

pow 4.0;;
- : float = 50.24
```

Definicje procedur

Example

```
let twice x = 2 * x;;
val twice : int -> int = <fun>

twice (twice 3);;
- : int = 12

let mocium s = "mocium panie, " ^ s;;
val mocium : string -> string = <fun>

mocium (mocium "me wezwanie");;
- : string = "mocium panie, mocium panie, me wezwanie"
```

Typy argumentów i wyniku procedury

Example

```
let plus x y = x + y;;
val plus : int -> int -> int = <fun>

let inc x = plus 1 x;;
val inc : int -> int = <fun>
let inc = plus 1;;
val inc : int -> int = <fun>
```

typ argumentu -> *typ argumentu* -> . . .-> *typ wyniku*

'a, 'b, 'c, . . . — reprezentują dowolne typy
(więcej o tym w przyszłości).

Wyliczanie wartości procedury

Zastosowanie procedury do argumentów:

- obliczenie (wyznaczenie) procedury, którą należy wywołać,
- obliczenie wartości argumentów
- tymczasowe środowisko, w którym parametrom przyporządkowane są wartości argumentów
  na podstawie środowiska, w którym procedura jest definiowana
- obliczenie treści w tymczasowym środowisku,
- wartość treści jest wynikiem zastosowania procedury do argumentów.

Statyczne wiązanie symboli

Example

```
let a = 24;;
let f x = 2 * x + a;;
let a = 15;;
f 9;;
- : int = 42
```

Treść procedury może mieć dostęp do przysłoniętej definicji.

λ-abstrakcja

Zwykle funkcje są zawsze przedstawiane wraz z ich nazwą.
Tymczasem definicja funkcji nic nie mówi o nazwie funkcji.
Czy mogą istnieć funkcje bez nazwy?
Tak, na przykład: $x \mapsto x + 1$

Definition

Składnia λ-abstrakcji:

$$\begin{array}{lcl}\langle\texttt{wyrażenie}\rangle & ::= & \underline{\texttt{function}}\ \langle\texttt{identyfikator}\rangle\ \underline{\texttt{->}}\ \langle\texttt{wyrażenie}\rangle\ | \\ & & \underline{\texttt{fun}}\ \{\ \langle\texttt{identyfikator}\rangle\ \}^{+}\ \underline{\texttt{->}}\ \langle\texttt{wyrażenie}\rangle\end{array}$$

λ-abstrakcja

Example

Oto alternatywna definicja procedur z poprzedniego przykładu:

```
(function x -> x * (x + 1)) (2 * 3);;
- : int = 42

let abs = function x -> if x < 0 then -x else x;;
val abs : int -> int = <fun>

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

let pow = function r ->
            pi *. ((function x -> x *. x) r);;
val pow : float -> float = <fun>
```

λ-abstrakcja

Example

```
let twice = function x -> 2 * x;;
val twice : int -> int = <fun>

let foo x y = x * (y +2);;
val foo : int -> int -> int = <fun>

let foo = function x -> function y -> x * (y +2);;
val foo : int -> int -> int = <fun>

let foo = fun x y -> x * (y +2);;
val foo : int -> int -> int = <fun>
```

Procedury rekurencyjne

Definition

Definiując procedury rekurencyjne dodajemy słówko `rec`.

$$\langle\texttt{definicja}\rangle ::= \underline{\texttt{let}}\ \underline{\texttt{rec}}\ \{\langle\texttt{identyfikator}\rangle\}^{+}\ \underline{\texttt{=}}\ \langle\texttt{wyrażenie}\rangle$$

- Definiowana procedura jest obecna w środowisku, w którym później będzie obliczana.
- Procedury wzajemnie rekurencyjne definiujemy równocześnie, używając `and`.
- W definicji podajemy parametry formalne lub λ-abstrakcję.

Procedury rekurencyjne

Example

Przykłady definicji rekurencyjnych:

```
let rec silnia num =
  if num < 2 then 1 else num * silnia (num - 1);;
val silnia : int -> int = <fun>

silnia 7 / silnia 5;;
- : int = 42

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

fib 10 - fib 7;;
- : int = 42
```

Procedury rekurencyjne

Example

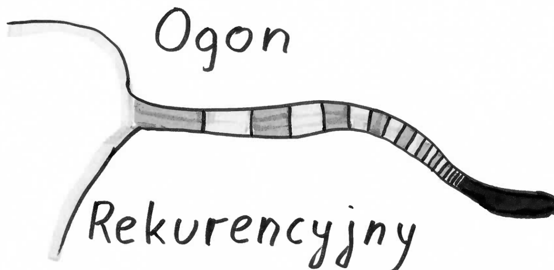
Przykłady definicji rekurencyjnych c.d.:

```
let rec petla x = x + petla x;;
val petla : int -> int = <fun>

let rec p = (function x -> p (x + 1));;
val p : int -> 'a = <fun>

let rec x = x + 2;;
This kind of expression is not allowed as right-hand side
of ‘let rec’
```

Istnieją też inne obiekty rekurencyjne, niż procedury...

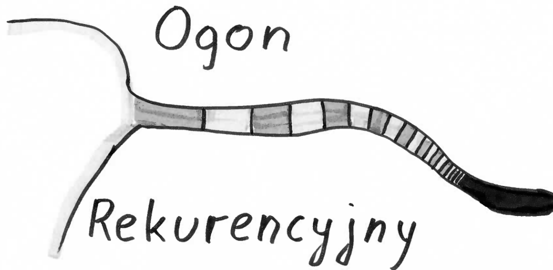
Rekurencja ogonowa

- Warunek: wynik wywołania rekurencyjnego jest zwracany bez żadnego dalszego przetwarzania (uproszczenie).
- Oszczędność pamięci — wywołanie rekurencyjne nie wymaga dodatkowej pamięci.
- Zamiast wywołania rekurencyjnego — podmiana argumentów i skok do początku treści.



Rekurencja ogonowa

- Dokładna analiza złożoności pamięciowej — w przyszłości.
- Zamiana rekurencji na ogonową:
  - Dodatkowe argumenty, akumulatory.
  - Pomocnicze procedury rekurencyjne.



Rekurencja ogonowa

Example

Silnia z akumulatorem:

```
let rec silnia_pom a n =
  if n <= 1 then a else silnia_pom (a * n) (n - 1);;
val silnia_pom : int -> int -> int = <fun>

let silnia num = silnia_pom 1 num;;
val silnia : int -> int = <fun>

silnia 3;;
- : int = 6
```

Rekurencja ogonowa

Example

Liczby Fibonacciego z akumulatorem:

```
let rec fib_pom a b num =
  if num = 0 then a else fib_pom b (a + b) (num - 1);;
val fib_pom : int -> int -> int -> int = <fun>

let fib num = fib_pom 0 1 num;;
val fib : int -> int = <fun>

fib 5;;
- : int = 5
```

Definicje lokalne

Definition

Składnia definicji lokalnych:

$$\langle\texttt{wyrażenie}\rangle ::= \langle\texttt{definicja}\rangle\ \underline{\texttt{in}}\ \langle\texttt{wyrażenie}\rangle$$

- Definicji lokalnych można używać wszędzie tam, gdzie wyrażeń.
- Zakres definicji lokalnej jest ograniczony do wyrażenia po `in` .
- Definicje `let-in` można zagnieżdżać.

Definicje lokalne

Example

```
let pow r =
  let pi = 3.14 and s = r *. r
  in pi *. s;;
val pow : float -> float = <fun>

pow 4.0;;
- : float = 50.24
```

Definicje lokalne

Example

```
let pitagoras a b =
  let square x = x * x
  in square a + square b;;
val pitagoras : int -> int -> int = <fun>

pitagoras 3 4;;
- : int = 25
```

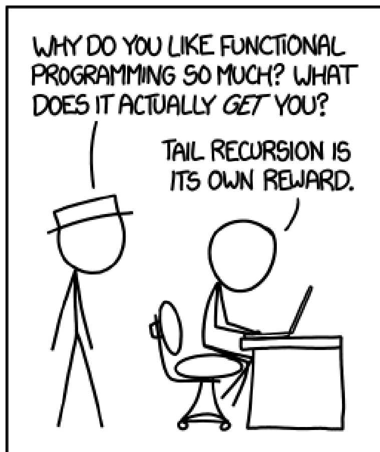
Komentarze

Komentarze mają podobną postać, jak w Pascalu, ale można je zagnieżdżać.

Example

```
(* To jest komentarz. *)
(* To też jest komentarz (* ale zagnieżdżony *). *)
```

Deser



http://xkcd.com/1270/