Języki ezoteryczne

Marek Materzok

4 grudnia 2006

Czym są języki ezoteryczne?

- Języki nie przeznaczone do praktycznego programowania
- Motywacja
  - Sprawdzenie pewnych teoretycznych koncepcji w praktyce
  - Ćwiczenie umysłowe
  - Dowcip
- Turing tar pit - „grzęzawisko Turinga"
  - Języki uproszczone w pewnych aspektach do granic możliwości, lecz wciąż Turing-zupełne
  - *„Strzeżcie się grzęzawiska Turinga, w którym wszystko jest możliwe, ale nic nie jest łatwe"*
    Alan Perlis, „Epigrams on Programming", ACM SIGPLAN Notices 17(9)
  - Maszyna Turinga, λ-rachunek, logika kombinatorów, gramatyki generacyjne...

INTERCAL

- Don Woods i James M. Lyon, 1972 - pierwszy język ezoteryczny
- Język-parodia, obśmiewający języki FORTRAN i COBOL - jego specyfikacja jest znakomitą lekturą do poduszki
- Szalone konstrukcje językowe
  - Zamiast `GOTO` jest `COME FROM`
  - Programista musi być uprzejmy i pisać `PLEASE` - ale nie za często, bo zostanie uznany za lizusa
  - `ABSTAIN/REINSTATE` - włączanie i wyłączanie interpretowania konkretnych instrukcji
  - `IGNORE/REMEMBER` - włączanie i wyłączanie operacji na pewnych zmiennych

Brainfuck

- Urban Müller, 1993 - chęć stworzenia języka z najmniejszym możliwym kompilatorem (stworzono kompilatory Brainfucka mniejsze od 200 bajtów)
- Model obliczeń: cykliczna tablica 30000 bajtów + wskaźnik „głowica” (celowe podobieństwo do maszyny Turinga)
- Podobieństwo do języka P'' - Corrado Böhm, 1964
  - Ciekawostka: był to pierwszy język strukturalny, którego Turing-zupełność udowodniono
- Osiem znaków, siedem komend:
  - <,> - przesuń głowicę w lewo (w prawo)
  - +,− - inkrementuj (dekrementuj) bajt pod głowicą
  - `,`,`.` - wprowadź (wyprowadź) jeden bajt
  - [*kod*] - wykonuj *kod* dopóki pod głowicą nie będzie zero

Hello World w Brainfucku

```
++++++++++
[>+++++++>++++++++++>+++>+<<<<-]
>++.
>+.
+++++++.
.
+++.
>++.
<<++++++++++++++.
>.
+++.
------.
--------.
>+.
>.
```

Brainfuck - warianty

- Ook, czyli język dla orangutanów

```
Ook. Ook? Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook.
Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook.
Ook! Ook? Ook? Ook. Ook. Ook. Ook. Ook. Ook. Ook.
```

- Fuckfuck, czyli niecenzuralnie na całego

```
fuck boob!!!!!!!! arse shag boob!!!!!!! fuck tits
butt shag cock fuck boob!!!!!! arse shag boob!!!
fuck tits butt shag boob cock boob!!!!!! cock!
```

- Brainfork, czyli wielowątkowy Brainfuck
  - Dodatkowa instrukcja `Y` rozdzielająca wątek na dwa
- Boolfuck, czyli Brainfuck na bitach

Befunge

- Chris Pressey, 1993 - chęc stworzenia języka ekstremalnie trudnego w kompilacji
- Maszyna stosowa, program i dane zapisane w dwuwymiarowej tablicy o toroidalnej topologii
  - Wskaźnik instrukcji może poruszać się w dowolnym kierunku
  - Tablica jest modyfikowalna (więc kod również)
  - Rozszerzenia do większej liczby wymiarów oraz innych topologii
- Instrukcje
  - 0,..,9 - umieszczenie liczby na stosie
  - +,−,*,/ - wykonanie działania arytmetycznego na stosie
  - <,^,>,v - zmiana kierunku wykonania
  - ? - przypadkowy kierunek wykonania
  - # - przeskoczenie kolejnej komórki
  - _,| - instrukcja warunkowa: skręć w prawo (w dół) jeśli na stosie jest 0, wpw skręć w lewo (w górę)
  - I inne...

Hello World w Befunge

```
>              v
v  ,,,,,"Hello"<
>48*,          v
v,,,,,,"World!"<
>25*,@
```

Hello World w Unlambda

```
`r```````````.H.e.l.l.o. .w.o.r.l.di
```

Thue

- John Colagioia, 2000
- Gramatyki generacyjne typu 0 (bez ograniczeń)
  - A wręcz semi-systemy Thuego - brak rozróżnienia pomiędzy terminalami a nieterminalami
  - Turing-zupełny - dowód konstruktywny, przekształcenie dowolnej maszyny Turinga (również niedeterministycznej) do gramatyki typu 0
- Program w Thue - ciąg produkcji oraz słowo początkowe
- Wynikiem programu jest dowolne słowo wyprowadzone ze słowa początkowego za pomocą produkcji (niedeterminizm)
- Możliwość przeprowadzania wejścia-wyjścia w trakcie pracy programu
  - Ciąg `:::` jest zastępowany przez wiersz odczytany z wejścia
  - Symbol `~` powoduje wysłanie na wyjście wszystkich znaków na prawo od niego w momencie zastosowania produkcji

Hello World w Thue

```
a::=~Hello World!
::=
a
```

Inne dziwne języki

- Malbolge
  - Język z piekła rodem! Nazwany od ósmego kręgu piekła z „Boskiej komedii" Dantego
  - Maszyna rejestrowa z 10-tritowymi (trit - cyfra trójkowa) słowami
  - Znak ma różne znaczenie w zależności od położenia w programie
  - Pierwszy Hello World został wygenerowany metodami ewolucyjnymi (!)
- Whitespace
  - Program złożony z białych znaków, wszystkie inne znaki to komentarz
- Shakespeare
  - Programy przypominają dramaty Szekspira
- i tak dalej...

Translacja do Brainfucka

- Chcemy tłumaczyć prosty język podobny do C do kodu Brainfuckowego
- Dla uproszczenia rezygnujemy z indeksowania pamięci, czyli też z:
  - Wskaźników do pamięci
  - Dynamicznej alokacji
  - Zmiennych globalnych
  - Funkcji zagnieżdżonych
- Nawet przy tym uproszczeniu da się zaimplementować rekurencję i tablice

Idea 1: taśma Brainfuckowa - stos

- Głowica wskazuje szczyt stosu
- Nie mamy wskaźnika ramki stosu - kompilator musi śledzić położenie wskaźnika stosu względem początku rekordu aktywacji
- Operacje na stosie:
  - `LOAD n` = `<`^{n}`[->`^{n}`+>+<`$^{n+1}$`]>`$^{n+1}$`[-<`$^{n+1}$`+>`$^{n+1}$`]`
  - `STORE n` = `<`^{n}`[-]>`$^{n-1}$`[-<`$^{n-1}$`+>`$^{n-1}$`]`
  - `DUP` = `<[->+>+<<]>>[-<<+>>]`
  - `POP` = `<[-]`
- Jako pamięć tymczasową dla wykonywanych operacji wykorzystujemy szczyt stosu
- Efekt: możemy już zaimplementować wyrażenia arytmetyczne i logiczne oraz instrukcję przypisania

Operacje arytmetyczne i logiczne

- `ADD` = `<[-<+>]`
- `SUB` = `<[-<->]`
- `MUL` = `<<[->>+<<]<[->[-<<+>>>+<]>[-<+>]<<]`
- `NEG` = `<[->-<]>[-<+>]`
- `NOT` = `+<[[-]>-<]>[-<+>]`
- `OR` = `<[[-]>+<]<[[-]>>+<<]>>[[-]<<+>>]<`
- `AND` = `++<[[-]>-<]<[[-]>>-<<]+>>[-<<->>]<`
- `EQ` = `SUB NOT`
- itp. (niektóre operacje - np. dzielenie - są trudne)

Idea 2: podział funkcji na bloki kodu

- Instrukcja pętli Brainfucka jest niewystarczająca do wyrażenia wywołań funkcji wprost
  - Chcemy symulować wskaźnik instrukcji
  - Spostrzeżenie: nie musimy adresować każdej instrukcji, wystarczą punkty, do których może nastąpić skok
  - Dzielimy więc kod funkcji na bloki, w których wykonanie odbywa się liniowo
- Struktura sterująca: selektor bloku
  - Idea: jeśli pod wskaźnikiem stosu jest numer bloku, wykonujemy ten blok, po czym wracamy do selektora; jeśli jest zero, kończymy

```
genblocks l = "[>+<" ++ gb l ++ "<]" where
    gb [] = "[[-]>-]"
    gb (h : t) = "[-" ++ gb t ++ ">[-" ++ h ++ ">]<]"
```

Translacja funkcji

- Wskaźnik instrukcji dwuczłonowy: numer funkcji, numer bloku kodu
- Kod wynikowy: dwa zagnieżdżone poziomy selektorów; funkcja main ma numer 1

$$gencode\ c = \texttt{">+>+"} +\!\!+\ genblocks\ (map\ genblocks\ c)$$

- Instrukcja skoku: odkładamy na stos numer kolejnego bloku
- Powrót z funkcji: zapisujemy wartość zwracaną w zarezerwowanym miejscu, czyścimy stos, odkładamy zero na stos
- Wywołanie funkcji
  - Rezerwujemy miejsce na stosie na wartość zwracaną
  - Odkładamy argumenty faktyczne funkcji na stos
  - Odkładamy na stos numer wywoływanej funkcji, jedynkę i zero
- Powrót z wywołania
  - Zdejmujemy argumenty faktyczne ze stosu

Przykład

```
byte fac(byte n) {
    if (n == 0)
        return 1
    else
        return n*fac(n-1)
}
```

```
Params: byte n
LOAD n
PUSH 0
EQ
JFALSE else
PUSH 1
RETURN
else:
LOAD n
LOAD n
LOAD 1
SUB
CALL fac
MUL
RETURN
```

Przykład

```
Params: byte n
LOAD n
PUSH 0
EQ
JFALSE else
PUSH 1
RETURN
else:
LOAD n
LOAD n
PUSH 1
SUB
CALL fac
MUL
RETURN
```

Przykład

```
pos Params: byte n at 3
  0 LOAD 3+0
  1 PUSH 0
  2 EQ
  1 JFALSE else
  0 PUSH 1
  1 RETURN
    else:
  0 LOAD 3+0
  1 LOAD 3+1
  2 PUSH 1
  3 SUB
  2 CALL fac
  2 MUL
  1 RETURN
```

Przykład

```
blk pos Params: byte n at 3
  0   0 LOAD 3+0
      1 PUSH 0
      2 EQ
      1 JFALSE 2 NEXT 1
  1   0 PUSH 1
      1 RETURN

  2   0 LOAD 3+0
      1 LOAD 3+1
      2 PUSH 1
      3 SUB
      2 CALL 1 NEXT 3
  3   2 MUL
      1 RETURN
```

Translacja tablic

- Dla każdego pola tablicy rezerwujemy pole pomocnicze oraz komórkę indeksu; dla uproszczenia zajmijmy się tylko tablicami bajtów
- Odczyt z tablicy
  - Zapis indeksu do pierwszego pola indeksowego
  - Znalezienie docelowej pozycji
  - Skopiowanie danej do pola pomocniczego
  - Powrót do pozycji startowej, przenosząc daną pomiędzy polami pomocniczymi
  - Przeniesienie danej z pierwszego pola indeksowego na stos
- Zapis do tablicy
  - Analogicznie, lecz kopiujemy w przeciwnym kierunku

Translacja do Unlambdy

- Chcemy tłumaczyć prosty język podobny do SML do kodu Unlambdy
- Dla uproszczenia rezygnujemy ze zmiennych
- Zmienne wymagałyby symulowania pamięci (np. za pomocą binarnego drzewa poszukiwań) oraz pracowitego przekazywania reprezentacji pamięci przez cały program
- Garbage collection?
- Tłumaczenie będzie dwustopniowe: najpierw wygenerujemy program w rachunku lambda odpowiadający programowi wejściowemu, po czym pozbędziemy się z niego lambd

Reprezentacja danych

Wygodnie jest użyć kodowania Churcha. Będę używać do zapisu pseudo-Haskella.

$$
\begin{aligned}
&\mathit{true} = \lambda a\ b \to a \\
&\mathit{false} = \lambda a\ b \to b
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{pair} = \lambda a\ b\ c \to c\ a\ b \\
&\mathit{fst} = \lambda a \to a\ \mathit{true} \\
&\mathit{snd} = \lambda a \to a\ \mathit{true}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{nzero} = \lambda f\ a \to a \\
&\mathit{nsucc} = \lambda n\ f\ a \to f\ (n\ f\ a) \\
&\mathit{nnum}\ n \mid n > 0 = \mathit{nsucc}\ [\![\mathit{nnum}\ (n-1)]\!] \\
&\qquad\quad \mid \mathit{otherwise} = \mathit{nzero}
\end{aligned}
$$

Reprezentacja danych

$$
\begin{aligned}
&\mathit{nadd} = \lambda x\ y\ f\ a \rightarrow x\ f\ (y\ f\ a)\\
&\mathit{nmul} = \lambda x\ y\ f\ a \rightarrow x\ (y\ f)\ a\\
&\mathit{ndec} = \lambda x \rightarrow \mathit{snd}\ (x\ (\lambda y \rightarrow \mathit{pair}\ (\mathit{nsucc}\ (\mathit{fst}\ y))\ (\mathit{fst}\ y))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{pair}\ \mathit{nzero}\ \mathit{nzero}))\\
&\mathit{nsub} = \lambda x\ y \rightarrow y\ \mathit{ndec}\ x\\
&\mathit{niszero} = \lambda x \rightarrow x\ (\lambda_ \rightarrow \mathit{false})\ \mathit{true}\\
&\mathit{niseven} = \lambda x \rightarrow x\ (\lambda x \rightarrow \mathit{notl}\ x)\ \mathit{true}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{and} = \lambda x\ y\ a\ b \rightarrow x\ (y\ a\ b)\ b\\
&\mathit{or} = \lambda x\ y\ a\ b \rightarrow x\ a\ (y\ a\ b)\\
&\mathit{notl} = \lambda x\ a\ b \rightarrow x\ b\ a
\end{aligned}
$$

Reprezentacja liczb całkowitych

Zdefiniujmy relację równoważności na zbiorze $\mathbb{N} \times \mathbb{N}$ oraz działania na klasach abstrakcji tej relacji.

$$(a, b) \sim (c, d) \iff a + d = b + c$$
$$[(a, b)]_\sim + [(c, d)]_\sim = [(a + c, b + d)]_\sim$$
$$[(a, b)]_\sim - [(c, d)]_\sim = [(a + d, b + c)]_\sim$$
$$[(a, b)]_\sim \cdot [(c, d)]_\sim = [(ac + bd, ad + bc)]_\sim$$

Otrzymany w ten sposób system jest izomorficzny z $\mathbb{Z}$.

$$\Phi([(a, b)]_\sim) = a - b$$

Dowód izomorfizmu

- Różnowartościowość. Niech $\Phi([(a,b)]_\sim) = \Phi([(c,d)]_\sim)$. Stąd $a-b=c-d$, czyli $a+d=b+c$, zatem $(a,b)\sim(c,d)$.
- Na. Niech $n\in\mathbb{Z}$. Jeśli $n\geqslant 0$, to $\Phi([(n,0)]_\sim)=n$, w przeciwnym wypadku $\Phi([(0,-n)]_\sim)=n$.
- Homomorfizm.

$$\begin{aligned}
\Phi([(a,b)]_\sim\cdot[(c,d)]_\sim) &= \Phi([(ac+bd,ad+bc)]_\sim) = \\
= ac+bd-ad-bc &= (a-b)(c-d) = \\
&= \Phi([(a,b)]_\sim)\Phi([(c,d)]_\sim) \\
\Phi([(a,b)]_\sim+[(c,d)]_\sim) &= \Phi([(a+c,b+d)]_\sim) = \\
= (a+c)-(b+d) &= (a-b)+(c-d) = \\
&= \Phi([(a,b)]_\sim)+\Phi([(c,d)]_\sim)
\end{aligned}$$

Dla $-$ dowód jest analogiczny.

Reprezentacja liczb całkowitych

Aby rozmiar reprezentacji nie rósł w nieskończoność, wprowadzamy normalizację.

$$
\begin{aligned}
&\mathit{inum}\ n = \mathit{pair}\ [\![\mathit{nnum}\ n]\!]\ [\![\mathit{nnum}\ (-n)]\!] \\
&\mathit{iadd} = \lambda a\ b \to \mathit{pair}\ (\mathit{nadd}\ (\mathit{fst}\ a)\ (\mathit{fst}\ b))\ (\mathit{nadd}\ (\mathit{snd}\ a)\ (\mathit{snd}\ b)) \\
&\mathit{isub} = \lambda a\ b \to \mathit{pair}\ (\mathit{nadd}\ (\mathit{fst}\ a)\ (\mathit{snd}\ b))\ (\mathit{nadd}\ (\mathit{snd}\ a)\ (\mathit{fst}\ b)) \\
&\mathit{imul} = \lambda a\ b \to \mathit{pair}\ (\mathit{nadd}\ (\mathit{nmul}\ (\mathit{fst}\ a)\ (\mathit{fst}\ b))\ (\mathit{nmul}\ (\mathit{snd}\ a)\ (\mathit{snd}\ b))) \\
&\qquad (\mathit{nadd}\ (\mathit{nmul}\ (\mathit{fst}\ a)\ (\mathit{snd}\ b))\ (\mathit{nmul}\ (\mathit{snd}\ a)\ (\mathit{fst}\ b))) \\
&\mathit{inorm} = \lambda a \to \mathit{pair}\ (\mathit{nsub}\ (\mathit{fst}\ a)\ (\mathit{snd}\ a))\ (\mathit{nsub}\ (\mathit{snd}\ a)\ (\mathit{fst}\ a))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{iiszero_n} = \lambda a \to \mathit{and}\ (\mathit{niszero}\ (\mathit{fst}\ a))\ (\mathit{niszero}\ (\mathit{snd}\ a)) \\
&\mathit{iispos_n} = \lambda a \to \mathit{notl}\ (\mathit{niszero}\ (\mathit{fst}\ a)) \\
&\mathit{iisneg_n} = \lambda a \to \mathit{notl}\ (\mathit{niszero}\ (\mathit{snd}\ a))
\end{aligned}
$$

Reprezentacja podstawowych operacji

Uwaga: Unlambda stosuje gorliwą ewaluację, musimy zarówno uważać na porządek ewaluacji, jak i rozleniwić niektóre konstrukcje.

$$
\begin{aligned}
&\mathit{comp}\ (\mathit{ConstBool}\ b) \mid b \equiv \mathit{True} = \mathit{true} \\
&\qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise} = \mathit{false} \\
&\mathit{comp}\ (\mathit{ConstInt}\ i) = [\![\mathit{inum}\ i]\!] \\
&\mathit{comp}\ (\mathit{Seq}\ a\ b) = (\lambda_ \to [\![\mathit{comp}\ b]\!])\ [\![\mathit{comp}\ a]\!] \\
&\mathit{comp}\ (\mathit{If}\ p\ a\ b) = [\![\mathit{comp}\ p]\!]\ (\lambda_ \to [\![\mathit{comp}\ a]\!]) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\lambda_ \to [\![\mathit{comp}\ b]\!])\ (\lambda a \to a) \\
&\mathit{comp}\ (\mathit{Let}\ l\ c) = [\![\mathit{clet2}\ (\mathit{map}\ \mathit{snd}\ l)]\!]\ \textbf{where} \\
&\qquad \mathit{clet1}\ [\,] = [\![\mathit{comp}\ a]\!] \\
&\qquad \mathit{clet1}\ (n : t) = \lambda[\![n]\!] \to [\![\mathit{clet1}\ t]\!] \\
&\qquad \mathit{clet2}\ [\,] = [\![\mathit{clet1}\ (\mathit{reverse}\ (\mathit{map}\ \mathit{fst}\ l))]\!] \\
&\qquad \mathit{clet2}\ (e : t) = [\![\mathit{clet2}\ t]\!]\ [\![\mathit{comp}\ e]\!]
\end{aligned}
$$

Rekursja

Standardowe podejście: przepisujemy funkcję f z użyciem kombinatora punktu stałego.

$$Y = \lambda f \to (\lambda x \to f\ (x\ x))\ (\lambda x \to f\ (x\ x))$$
$$comp\ (Letrec'\ n\ v\ c) = [\![comp\ (Let\ n\ v'\ c)]\!] \textbf{ where}$$
$$v' = Y\ (\lambda [\![n]\!] \to [\![v]\!])$$

Ale chcielibyśmy mieć wzajemną rekursję. Można jednak zaobserwować, iż

$$f' = (\lambda f \to (\lambda f \to fc)\ (f\ f)), f'\ f' \equiv_{\beta\eta} Y\ (\lambda f \to fc)$$
$$(\lambda f \to c)\ (Y\ (\lambda f \to fc)) \equiv_{\beta\eta} (\lambda f \to c)\ (f'\ f')$$
$$\equiv_{\beta\eta} (\lambda f \to (\lambda f \to c)\ (f\ f))\ f' \equiv_{\beta\eta} (\lambda f \to c')\ f'$$

A to podejście można już łatwo uogólnić na kilka funkcji:

$$f' \equiv_{\beta\eta} (\lambda f\ g \to (\lambda f\ g \to fc)\ (f\ f\ g)\ (g\ f\ g))$$
$$g' \equiv_{\beta\eta} (\lambda f\ g \to (\lambda f\ g \to gc)\ (f\ f\ g)\ (g\ f\ g))$$
$$c' \equiv_{\beta\eta} (\lambda f\ g \to c)\ (f\ f\ g)\ (g\ f\ g)$$

Eliminacja abstrakcji

$$
\begin{aligned}
&\mathit{abselim}\ (\lambda x \to x) = I \\
&\mathit{abselim}\ (\lambda x \to y) = K\ y \qquad \text{-- jeśli y jest zmienną lub kombinatorem} \\
&\mathit{abselim}\ (\lambda x\ y \to e) = [\![\mathit{abselim}\ (\lambda x \to [\![\mathit{abselim}\ (\lambda y \to [\![e]\!])]\!])]\!] \\
&\mathit{abselim}\ (\lambda x \to e1\ e2) = S\ [\![\mathit{abselim}\ (\lambda x \to [\![e1]\!])]\!]\ [\![\mathit{abselim}\ (\lambda x \to [\![e2]\!])]\!]
\end{aligned}
$$

Pokażę, że $\mathit{abselim}\ e \equiv_{\beta\eta} e$ i $\mathit{abselim}\ e$ nie zawiera lambd. Dowód przez indukcję, miara - liczba lambd w wyrażeniu e.

- Przypadki 1 i 2 trywialne.
- $e = \lambda x\ y \to e'$.

$$
\begin{aligned}
\mathit{abselim}\ e &\equiv_{\beta\eta} \mathit{abselim}\ (\lambda x \to \mathit{abselim}\ (\lambda y \to e'))\ | \\
&\equiv_{\beta\eta} \mathit{abselim}\ (\lambda x \to \lambda y \to e') \equiv_{\beta\eta} \lambda x\ y \to e' \equiv_{\beta\eta} e
\end{aligned}
$$

- $e = \lambda x \to e1\ e2$.

$$
\begin{aligned}
\mathit{abselim}\ e &\equiv_{\beta\eta} S\ (\mathit{abselim}\ (\lambda x \to e1))\ (\mathit{abselim}\ (\lambda x \to e2)) \\
&\equiv_{\beta\eta} S\ (\lambda x \to e1)\ (\lambda x \to e2) \\
&\equiv_{\beta\eta} (\lambda a\ b\ x \to a\ x\ (b\ x))\ (\lambda x \to e1)\ (\lambda x \to e2) \\
&\equiv_{\beta\eta} \lambda x \to e1\ e2
\end{aligned}
$$