

# Tytuł

Patryk Wałach

January 2022

## Contents

<b>1</b>	<b>[Wstęp]</b>	<b>2</b>
<b>2</b>	<b>Ogólne wprowadzenie</b>	<b>2</b>
2.1	Analiza leksykalna . . . . .	2
2.2	Parser . . . . .	2
2.3	System typów . . . . .	3
2.4	System typów ML . . . . .	3
2.5	Rekord z wariantami . . . . .	4
<b>3</b>	<b>Inferencja typów w teori</b>	<b>4</b>
3.1	System typów Hindley–Milner . . . . .	4
3.2	Algorytm W . . . . .	4
<b>4</b>	<b>Rescript/ReasonML jako języki realizujące podobne zadania</b>	<b>4</b>
<b>5</b>	<b>Założenia i priorytety opracowanej aplikacji</b>	<b>5</b>
5.1	Opis formalny składni języka . . . . .	9
<b>6</b>	<b>Narzędzia</b>	<b>9</b>
6.1	Język python . . . . .	9
6.2	Parsowanie i tokenizowanie przy użyciu biblioteki sly . . . . .	9
6.3	Środowisko nodejs do uruchomienia skompilowanego kodu . . . . .	9
<b>7</b>	<b>Implementacja</b>	<b>9</b>
7.1	lexer . . . . .	9
7.2	parser . . . . .	9
7.3	inferencja typów . . . . .	9
7.4	kompilacja . . . . .	9
<b>8</b>	<b>Opis działania</b>	<b>9</b>
8.1	Co działa . . . . .	9
8.2	Uwagi co do obsługi błędów . . . . .	9
<b>9</b>	<b>[Podsumowanie]</b>	<b>9</b>

## 1 [Wstęp]

## 2 Ogólne wprowadzenie

### 2.1 Analiza leksykalna

Analiza leksykalna w informatyce jest to proces rozbijania program źródłowych na jednostki logiczne (zwane leksemami) złożone z jednego lub więcej znaków, które łącznie mają jakieś znaczenie[J.E. Hopcroft(2005)]. Przykładami leksemów mogą być słowa kluczowe (np. `while`), identyfikator lub liczba składająca się z cyfr. Rozdzielaniem programu źródłowego, na leksemy, zajmuje się lexer.

Token jest strukturą reprezentującą leksem i wprost go kategoryzującą[Aho et al.(1985)Aho, Sethi, and Ullman]. Tokeny ułatwiają pracę parserowi. Tokeny kategoryzuje się na komputerowy odpowiednik tego, co lingwiści określiliby mianem części mowy. Biorąc jako przykład poniższy kod w języku C:

```
1  x = a + b * 2;
```

analiza leksykalna, zwraca tokeny:

```
1  [(identifier, x), (operator, =), (identifier, a), (operator, +),
   (identifier, b), (operator, *), (literal, 2), (separator, ;)]
```

Dwoma ważnymi przypadkami są znaki białe i komentarze. One również muszą być uwzględnione w gramatyce i przeanalizowane przez lexer, lecz mogą być odrzucone (nie produkować żadnych tokenów) i traktowane jako spełniające mało znaczące zadanie, rozdzielania dwóch tokenów (np. w `if x` zamiast `ifx`).

### 2.2 Parser

Analizator składniowy, parser – program dokonujący analizy składniowej danych wejściowych w celu określenia ich struktury gramatycznej w związku z określoną gramatyką formalną. Analizator składniowy umożliwia przetworzenie tekstu czytelnego dla człowieka w strukturę danych przydatną dla oprogramowania komputera. Wynikiem analizy składni, dokonywanej przez parser, najczęściej jest drzewo składniowe nazywane czasami drzewem wyprowadzenia[Aho et al.(1985)Aho, Sethi, and Ullman].

Zadanie parsera sprowadza się do sprawdzenia czy i jak dane wejściowe mogą zostać wyprowadzone z symbolu startowego. To zadanie można zrealizować na dwa sposoby:

- Analiza zstępująca (ang. top-down parsing) to strategia znajdowania powiązań między danymi przez stawianie hipotez dotyczących drzewa rozbioru składniowego i sprawdzanie, czy zależności między danymi są zgodne z tymi hipotezami.
- Analiza wstępująca (ang. bottom-up parsing) – ogólna metoda analizy składniowej, w której zaczyna się od słowa wejściowego i próbuje się zredukować je do symbolu startowego. Drzewo wyprowadzenia jest konstruowane od liści do korzenia (stąd nazwa). W każdym momencie w trakcie tego procesu mamy formę zdaniową, która zawiera segment, powstały w ostatnim kroku wyprowadzenia. Segment ten (nazywany uchwytym (ang. handle) jest prawą stroną produkcji i powinien zostać w tym kroku zredukowany do jej lewej strony, w wyniku czego powstanie poprzednia forma zdaniowa z wyprowadzenia. Główna trudność w analizie wstępującej polega właśnie na odpowiednim znajdowaniu uchwytów. Analiza wstępująca może przebiegać w określonym kierunku (np. od lewej do prawej), lub w sposób bezkierunkowy, wtedy analizowane jest całe słowo naraz. Jednym z bardziej znanych przedstawicieli metody bezkierunkowej jest algorytm CYK. Do metod kierunkowych zalicza się między innymi parsery shift-reduce czyli LR, LALR, SLR, BC, pierwszeństwa.

## 2.3 System typów

System typów jest to system klasyfikacji wyrażeń w zależności od rodzajów wartości, jakie one generują[Pierce(2002)]. Każdej obliczonej wartości przypisywany jest pewien typ, który jednoznacznie definiuje, jakie operacje można na niej wykonać. Śledząc przepływ wartości, system typów stara się udowodnić, że w programie występuje poprawne typowanie, tzn. nie dochodzi do sytuacji, w której na wartości określonego typu próbujemy wykonać niedozwoloną operację.

## 2.4 System typów ML

System typów ML jest to silny system typów stosowany w językach rodziny ML (Ocaml, Standard ML) oparty na inferencji.

Podstawowy system typów jest następujący: istnieją typy proste, takie jak string, int, bool, unit (typ pusty) itd. Z dowolnych typów można też generować typy złożone – przez krotki (typ1 \* typ2, typ1 \* typ2 \* typ3 itd.), konstruktory typów (typ list, typ tree itd.) i funkcje (typ1 → typ2).

System próbuje nadać typy każdemu wyrażeniu języka, i nie licząc kilku rzadkich przypadków, udaje mu się to całkiem dobrze.

Generalnie system taki wyklucza polimorfizm (nie licząc typów polimorficznych), jednak w Standard ML stworzono specjalne reguły umożliwiające polimorfizm dla wyrażeń arytmetycznych.

System typów ML jest interesujący z teoretycznego punktu widzenia – wiele problemów ma bardzo wysoką złożoność, jednak w praktyce inferencja zachodzi bardzo szybko – typy, które są rzeczywiście używane, są zwykle bardzo proste – rzadko używa się funkcji rzędów wyższych niż trzeci-czwarty, oraz liczby argumentów większej niż kilkanaście.

W rzeczywistych implementacjach dochodzą do tego bardziej złożone problemy typizacji obiektów, modułów itd.

## 2.5 Rekord z wariantami

Rekord z wariantami jest to rodzaj rekordu, posiadającego tę właściwość, że zbiór rekordów posiada wspólny typ, lecz różną postać, określoną aktualną wartością specjalnego pola znacznikowego.

Przykład - Zakładając, że chcemy stworzyć drzewo binarne intów. W języku ML, zrobilibyśmy to tworząc nowy typ danych w ten sposób:

```
1 datatype tree = Leaf
2   | Node of (int * tree * tree)
```

Leaf i Node są konstruktorami, które pozwalają nam na stworzenie konkretnego drzewa, np.

```
1 Node(5, Node(1, Leaf, Leaf), Node(3, Leaf, Node(4, Leaf, Leaf)))
```

## 3 Inferencja typów w teorii

### 3.1 System typów Hindley–Milner

### 3.2 Algorytm W

## 4 Rescript/ReasonML jako języki realizujące podobne zadania

## 5 Założenia i priorytety opracowanej aplikacji

Tworząc aplikację, chciałem, by język posiadał podstawowe typy danych (liczby, stringi, wartość logiczna), kilka typów generycznych (funkcje, tablice), typ ‘Option’, oraz możliwość tworzenia własnych typów. Dodatkowo nie powinno być potrzeby podawania typów zmiennych w większości przypadków, kompilator sam powinien wykrywać typy zmiennych na podstawie ich użycia.

Język poza zmiennymi, potrzebuje możliwości wykonywania operacji na danych, dlatego ważne było dla mnie, by zaimplementować operatory binarne, oraz unarne. Operatory te miały też spełniać ważną rolę w trakcie inferencji typów. W języku javascript operator ‘+’ może być wykorzystywany do dodawania liczb jak i konkatencji stringów, ważne więc było by stworzyć dwa oddzielne operatory.

Prymitywne typy danych:

- string

```
1 greeting = 'Hello world!'
```

examples/string.uwu

Do konkatencji stringów służy operator ++

- Wartość logiczna ma typ `Bool` i wartość `True` lub `False`. Powiązane operacje:

- `<>`, równość pomiędzy dwiema liczbami
- `>`, `<`
- `!=` równość

- liczby Powiązane operacje: `+`, `-`, `*`, `/`, `*`, `%`, `//`

Chciałem również by funkcje wieloargumentowe kompilowane były jako funkcje jednoargumentowe zwracające kolejne funkcje, co pozwala na wywoływanie funkcji bez wszystkich argumentów w celu zwrócenia funkcji przyjmującej resztę argumentów tzw. currying.

```

1 def classy_greeting(first_name, last_name) do
2   "The name's " ++ last_name ++ ', ' ++ first_name ++ ' ' ++
    last_name
3 end
4
5 def compose1(f, g, a) do
6   f(g(a))
7 end
8
9 def compose2(f, g, a, b) do
10  f(g(a, b))
11 end
12
13 yell_greetings = compose2(to_upper, classy_greeting)
14 yell_greetings('James', 'Bond') # "THE NAME'S BOND, JAMES BOND"
15
16 compose1(compose1(abs, add(1)), multiply(2))(-4) # 7

```

examples/currying.uwu

```

1 const classy_greeting = (first_name) => (last_name) => {
2   return "The name's " + last_name + ", " + first_name + " " +
    last_name;
3 };
4 const compose1 = (f) => (g) => (a) => {
5   return f(g(a));
6 };
7 const compose2 = (f) => (g) => (a) => (b) => {
8   return f(g(a)(b));
9 };
10 const yell_greetings = compose2(to_upper)(classy_greeting);
11 yell_greetings("James")("Bond");
12 compose1(compose1(abs)(add(1.0)))(multiply(2.0))(-4.0);

```

examples/currying.uwu.js

Jednym z ważniejszych elementów każdego języka jest możliwość wykonywania różnego zbioru instrukcji, warunkowo. W tym celu planowałem zaimplementowanie instrukcji 'if', oraz 'case'. Instrukcja 'case' wykonywać ma dopasowanie do wzorca (tzw. pattern-matching), wykonywać, odpowiedni zbiór instrukcji zależnie od wprowadzonych danych. Kompilator, powinien ostrzegać, jeżeli ścieżka dla jednego z typów danych nie została zaimplementowana.

- Przykład - funkcja łącząca dwie posortowane tablice

```

1 def merge<A>(a, b) do
2   merge2: Callable<Array<A>, Callable<Array<A>, Array<A>>> =
3     'merge'
4
5   case Tuple(get_head(a), get_head(b)) of
6     Tuple(Empty(), _) do b end
7     Tuple(_, Empty()) do a end
8     Tuple(Head(head_a, rest_a), Head(head_b, rest_b)) do
9       if head_a < head_b then
10        [head_a] | merge2(rest_a, b)
11      else
12        [head_b] | merge2(a, rest_b)
13      end
14    end
15  end
end

```

examples/merge.uwu

```

1 const merge = (a) => (b) => {
2   const merge2 = merge;
3   return (() => {
4     const $ = { TAG: "Tuple", _0: get_head(a), _1: get_head(b) };
5     if (typeof $ !== "string" && $.TAG === "Tuple") {
6       if ($. _0 === "Empty") {
7         const _ = $. _1;
8         return b;
9       }
10      if ($. _1 === "Empty") {
11        const _ = $. _0;
12        return a;
13      }
14      if (typeof $. _0 !== "string" && $. _0.TAG === "Head") {
15        if (typeof $. _1 !== "string" && $. _1.TAG === "Head") {
16          const head_b = $. _1._0;
17          const rest_b = $. _1._1;
18          const head_a = $. _0._0;
19          const rest_a = $. _0._1;
20          return (() => {
21            if (head_a < head_b) {
22              return [head_a].concat(merge2(rest_a)(b));
23            } else {
24              return [head_b].concat(merge2(a)(rest_b));
25            }
26          })();
27        }
28        throw new Error("Non-exhaustive pattern match");
29      }
30      throw new Error("Non-exhaustive pattern match");
31    }
32    throw new Error("Non-exhaustive pattern match");
33  })();
34 };

```

examples/merge.uwu.js

Kolejnym dość ważnym elementem języka jest brak wyrażenia ‘return’, które jest wykorzystywane do zwrócenia wartości z funkcji. Zamiast tego każdy blok instrukcji powinien zwracać ostatnie wyrażenie. Pozwoli to na łatwiejsze inicjowanie zmiennych, w przypadku gdy inicjalizacja wymaga więcej niż jednej linii kodu.

- Przykład

```
1 message = if is_morning then
2   'Good morning!'
3 else
4   'Hello!'
5 end
6
7
8 result = do
9   arr1 = [1, 2, 3]
10  arr2 = map(arr1, add_one)
11
12  filter(arr2, def is_even(x) do
13    x % 2 == 0
14  end)
15 end
```

examples/return.uwu

```
1 const message = (() => {
2   if (is_morning) {
3     return "Good morning!";
4   } else {
5     return "Hello!";
6   }
7 })();
8 const result = (() => {
9   const arr1 = [1.0, 2.0, 3.0];
10  const arr2 = map(arr1)(add_one);
11  const is_even = (x) => {
12    return x % 2.0 === 0.0;
13  };
14  return filter(arr2)(is_even);
15 })();
```

examples/return.uwu.js



## 5.1 Opis formalny składni języka

## 6 Narzędzia

### 6.1 Język python

### 6.2 Parsowanie i tokenizowanie przy użyciu biblioteki sly

### 6.3 Środowisko nodejs do uruchomienia skompilowanego kodu

## 7 Implementacja

### 7.1 lexer

### 7.2 parser

### 7.3 inferencja typów

### 7.4 kompilacja

## 8 Opis działania

### 8.1 Co działa

### 8.2 Uwagi co do obsługi błędów

## 9 [Podsumowanie]

## 10 [spisy – rysunków, tabel, listingów itp.]

## References

[Aho et al.(1985)] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers*. Addison Wesley, Boston, MA, January 1985. ISBN 9780201101942.

[J.E. Hopcroft(2005)] J.D. Ullman J.E. Hopcroft, R. Motwani. *Wprowadzenie do Teorii automatów, języków i obliczeń*. Wydawnictwo Naukowe PWN, 2005. ISBN 8301145021.

[Pierce(2002)] Benjamin C Pierce. *Types and Programming Languages*. The MIT Press. MIT Press, London, England, January 2002. ISBN 0262162091.