Tytuł

Patryk Wałach

January 2022

1 [Wstęp]

2 Ogólne wprowadzenie

2.1 Parsowanie i tokenkizowanie

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. Przykładami leksemów mogą być słowa kluczowe (np. while), identyfikator lub liczba składająca się z cyfr. Rozdzielaniem programu źródłowego, na leksemy, zajmuje się lekser.

Token jest strukturą reprezentującą leksem i wprost go kategoryzującą, co ułatwia późniejszą pracę parserowi. Tokeny kategoryzuje się na komputerowy odpowiednik tego, co lingwiści określiliby mianem części mowy. Biorąc jako przykład poniższy kod w języku C:

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

analiza leksykalna, zwraca tokeny:

```
[(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`).

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 binarane, oraz unarne. Operatory te miały też spełniać ważną rolą w trakcie inferencji typów. W języku javascript operator '+' może być wykorzystywany do dodawania liczb jak i konkatenacji stringów, ważne więc było by stworzyć dwa oddzielne operatory.

Prymitywne typy danych:

- string

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

examples/string.uwu

  Do konkatenacji 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.

```
def classy_greeting(first_name, last_name) do
    `"The name's "` ++ last_name ++ ', ' ++ first_name ++ ' ' ++
    last_name
end

def compose1(f, g, a) do
    f(g(a))
end

def compose2(f, g, a, b) do
    f(g(a, b))
end

yell_greetings = compose2(to_upper, classy_greeting)
yell_greetings('James', 'Bond') # "THE NAME'S BOND, JAMES BOND"

compose1(compose1(abs, add(1)), multiply(2))(-4) # 7
```

examples/currying.uwu

```
const classy_greeting = (first_name) => (last_name) => {
  return "The name's " + last_name + ", " + first_name + " " +
    last_name;
};
const compose1 = (f) => (g) => (a) => {
  return f(g(a));
};
const compose2 = (f) => (g) => (a) => (b) => {
  return f(g(a)(b));
};
const yell_greetings = compose2(to_upper)(classy_greeting);
yell_greetings("James")("Bond");
compose1(compose1(abs)(add(1.0)))(multiply(2.0))(-4.0);
```

examples/currying.uwu.js

Jednym z ważniejszych elemenów 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

```
def merge<A>(a, b) do
    merge2: Callable<Array<A>, Callable<Array<A>, Array<A>>> =
     `merge`

    case Tuple(get_head(a), get_head(b)) of
        Tuple(Empty(), _) do b end
        Tuple(_, Empty()) do a end
        Tuple(Head(head_a, rest_a), Head(head_b, rest_b)) do
            if head_a < head_b then
                [head_a] | merge2(rest_a, b)
            else
                [head_b] | merge2(a, rest_b)
            end
        end
    end
end
```

examples/merge.uwu

```
const merge = (a) => (b) => {
  const merge2 = merge;
  return (() => {
    const $ = { TAG: "Tuple", _0: get_head(a), _1: get_head(b)
     };
    if (typeof $ !== "string" && $.TAG === "Tuple") {
      if ($._0 === "Empty") {
        const _ = $._1;
        return b;
      }
      if ($._1 === "Empty") {
        const _ = $._0;
        return a;
      }
      if (typeof $._0 !== "string" && $._0.TAG === "Head") {
        if (typeof $._1 !== "string" && $._1.TAG === "Head") {
          const head_b = $._1._0;
          const rest_b = $._1._1;
          const head_a = $._0._0;
          const rest_a = $._0._1;
          return (() => {
            if (head_a < head_b) {
              return [head_a].concat(merge2(rest_a)(b));
            } else {
              return [head_b].concat(merge2(a)(rest_b));
            }
          })();
        }
        throw new Error("Non-exhaustive pattern match");
      }
      throw new Error("Non-exhaustive pattern match");
    }
    throw new Error("Non-exhaustive pattern match");
  })();
};
```

examples/merge.uwu.js

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

- Przykład

```
message = if is_morning then
  'Good morning!'
else
  'Hello!'
end


result = do
    arr1 = [1, 2, 3]
    arr2 = map(arr1, add_one)

    filter(arr1, def is_even(x) do
        x % 2 == 0
    end)
end
```

examples/return.uwu

```
const message = (() => {
  if (is_morning) {
    return "Good morning!";
  } else {
    return "Hello!";
  }
})();
const result = (() => {
  const arr1 = [1.0, 2.0, 3.0];
  const arr2 = map(arr1)(add_one);
  const is_even = (x) => {
    return x % 2.0 === 0.0;
  };
  return filter(arr1)(is_even);
})();
```

examples/return.uwu.js