Opis języka

Planuję zrobić język imperatywny, statycznie typowany. Jako, że nie planuję wychodzić poza tabelkę z funkcjonalnością mojego języka, omówię go punkt po punkcie. Gramatyka została zrobiona poprzez dodanie kilku reguł do gramatyki *Latte*.

1. Typy `string`, `bool`, `int`,
2. Literały typu string (wzięte w cudzysłów, jak w C++) oraz `true`, `false` (typu `bool`) oraz liczby całkowite (zapisane w systemie dziesiętnym).Ewaluowanie wyrażeń będzie odbywało się od lewej, poprzez istnienie funkcji będzie miało skutki uboczne ze skutkiem widocznym natychmiastowo. Ewaluowanie wyrażeń będzie leniwe, np `if (2 > 1 || f())` nie wywoła funkcji `f()`. Dodałem również operację dwuargumentową potęgowania `^`.
3. Będą.
4. Będzie, wraz z metodą `toString` określoną dla trzech podstawowych typów.
5. `while`, `if` oraz `if` z `else`.
6. Zaimplementowałem funkcje wraz z rekurencją. Będzie odpowiednik procedur, czyli funkcje zwracające `void`.
7. Przekazywanie przez referencję a'la C++ (słówko kluczowe `ref`, które może pojawić się przy parametrze funkcji).
8. –
9. Nie planuję mieć zmiennych globalnych, ale punkt 9ty zapewnia, że by go spełnić wystarczą funkcje zagnieżdżone, które planuję zaimplementować.
10. Zostanie zaimplementowana, wraz z type-checkingiem – planuję robić statyczne typowanie (punkt 13).
11. Zostaną zaimplementowane.
12. Planuję wykonać.
13. Zostanie zaimplementowane wraz ze statycznym wiązaniem. Deklaracja funkcji będzie taka sama syntaktycznie jak deklaracja funkcji globalnych.
14. –
15. Planuję zaimplementować typ `tuple<*typy składające się na krotkę po przecinku*>`, np `tuple<int, string>` lub `tuple<tuple<int, bool, int>`, string>

Będzie możliwość rozpakowywania krotek - patrz plik `15.fl`. Nie zaimplementowałem w gramatyce możliwości poyskania konkretnej wartości tuple, jednak jeśli starczy czasu lub będzie to konieczne by uzyskać punty za ten podpunkt, spróbuję to dodać. Dodatkowo wyobrażam sobie, że być może warto byłoby dodać słowo kluczowe `_`, tak, by można było pominąć niektóre elementy przy rozpakowywaniu (jak w Pythonie).

Nie można będzie zmienić wartości w krotce (ale będzie można przypisać do krotki nową wartość).

Przy przypisaniu krotka będzie kopiowała wartości.

16. –
17. Zaimplemetowany zostanie typ funkcyjny `function<ret_type(*lista parametrów po przecinku*)>`, np `function<void(int ref)>`, `function<string(int, bool)>`. Na chwilę obecną myślę, że funkcje przy przekazywaniu jako parametr będzie przyjmowała zawsze semantykę referencji. Obiekty tego typu można będzie tworzyć na dwa sposoby - deklarując funkcję / funkcję zagnieżdżoną lub funkcję anonimową (lambdę).

Zaimplementowane zostaną domknięcia.

W pliku `17.fl` jest przypomniany przykład z zajęć, w którym domknięcia wraz z przekazywaniem przez referencje powodują undefined behaviour. Nie planuję zabraniać użytkownikowi tworzenia takiego UB. Bardzo możliwe, że w związku, że w programie stos będzie tylko rósł (nie musimy w intepreterze nic dealokować), w rzeczywistości taka konstrukcja będzie zachowywało się w pewien określony sposób (mimo, że jesteśmy już poza blokiem w którym deklarowaliśmy zmienną, nie zajeliśmy miejsca w pamięci na które wskazujemy w lambdzie niczym nowym, więc domknięcie będzie działało).

18. –

Dodatkowe rzeczy, które planuję zaimplementować (są w gramatyce)

- Zmienne read-only (będzie można zaimplementować zmienną typu `const`, na którą ponowne przypisanie spowoduje błąd, prawdopodobnie na etapie type-checkingu).
- Konkatenacja stringów (string + string będzie zdefiniowaną operacją).

Funkcje wbudowane

Planuję dodac funkcje wbudowane:

1. `string toString(int), string toString(bool), string toString(string)` - działanie wiadome. Funkcję będę musiał rozbić na taką o trzech różnych nazwach jeśli nie zaimplementuję możliwości polimorficznych funkcji (to jest takich o tej samej nazwie ale różnym typie)
2. `void print(string)` - działanie wiadome
3. `void assert(bool)` - kończy wykonanie programu gdy napotka na `false`.

Rzeczy które zaimplementuję jak będzie szło wyjątkowo dobrze, ale nie deklaruję ich

- Dostęp do części elementu krotki za pomocą indeksu.
- Możliwość podania `_` przy rozpakowywaniu krotki (a'la Python).
- for (jak w podpunkcie 8. tabelki dla języka imperatywnego)

- Generatory. Zaimplementowałbym typ `generator<typ>`, wraz z instrukcją `yield` oraz instrukcją `next`. Dodatkowo zrobiłbym funkcje `for` lub `forG`, które działałyby tak jak w programie `opcjonalne.fl`. Generatory również przyjmowałyby semantykę referencji przy przekazywaniu jako parametr.
- Dedukcja typu wraz ze słówkiem kluczowym `auto` a'la C++.