FHLL

Opis zakładanej funkcjonalności

I. System typów

1. Język jest statycznie typowany
2. Język jest słabo typowany
3. Język wspiera inferencję typów

II. Zmienne i typy danych

1. Zmienne domyślnie są niemutowalne.
2. Język wspiera wbudowane typy danych:
   - Całkowitoliczbowe: `i16`, `i32`, `i64`, `u16`, `u32`, `u64`
   - Zmiennoprzecinkowe: `f32`
   - Boolowskie: `bool`
   - Ciągi znaków: `str` (zawierający dowolne znaki, włącznie z wyróżnikiem, nową linią oraz tabulacją)
3. Zmienne mogą występować w kontekście globalnym oraz lokalnym.
4. Zmienne w kontekście lokalnym są usuwane po wyjściu z kontekstu.
5. Zmienne są przekazywane do najbliższego kontekstu.

III. Operacje

1. Stosowanie operatorów zgodne jest z konwencją stosowaną w matematyce:
   - Nawiasowanie przed mnożeniem/dzieleniem.
   - Mnożenie/dzielenie przed dodawaniem/odejmowaniem.
   - Znak minusa przed dodawaniem/odejmowaniem.
2. Interpreter ma wbudowane operacje:
   - Dla typów liczbowych: `+`, `-`, `/`, `*`, `<`, `>`
   - Dla typów boolowskich: `&&`, `||`, `!`
   - Dla typów znakowych: `+` (konkatenacji)
   - Dla wszystkich: `==`, `!=`
3. Operacje na typach powodują wywołanie stosownej funkcji realizującej daną operację.
   - W przypadku braku znalezienia odpowiedniej funkcji zostaje wykorzystany mechanizm panikowania.
   - Wykorzystywany jest mechanizm przeciążania funkcji do definicji własnych operacji na typach.

IV. Konstrukcje

1. Język obsługuje konstrukcję warunkową `if-else`.
2. Język obsługuje pętlę `while`.
3. Język zapewnia mechanizm panikowania `panic`.
   - Użycie mechanizmu panikowania kończy interpretację programu oraz wyświetla komunikat o błędzie oraz linii kodu, z którego ona pochodzi.
4. Język pozwala na definicję funkcji.
   - Argumenty przekazywane są przez wartość.
   - Możliwe jest przeciążanie funkcji.
   - Nie można przeciążać funkcji tylko ze względu na zwracaną wartość.
   - Możliwe jest wywoływanie rekurencyjne funkcji - maksymalna głębokość regulowana jest przez flagę interpretera.

V. Struktury

1. Język obsługuje struktury:

- Struktura może posiadać pola o dowolnym typie.
- Język pozwala na odczyt/zapis pól w strukturach.
- Wszystkie pola w strukturach są publiczne.

2. Język obsługuje rekordy wariantowe.
   - Możliwe jest sprawdzenie czy rekord jest danym wariantem
   - Możliwe jest rzutowanie tylko do odpowiedniego wariantu - próba rzutowania do innego wariantu zakończy się spanikowaniem programu
3. Struktury traktowane są jako typy zdefiniowane przez użytkownika:
   - Dozwolone jest wykorzystanie struktur jako typów argumentów funkcji.

VI. Inne

1. Język wspiera komentarze jednolinijkowe - koniec komentarza wyznaczony jest przez znak nowej linii lub znak końca pliku.
2. Rzutowanie typów dokonywane przez interpreter dokonywane jest automatycznie dla predefiniowanych zestawów typów.
3. Interpreter posiada wbudowane funkcje do wyświetlania oraz pobierania danych.
4. Możliwe jest rzutowanie ręczne typów

Konstrukcje językowe

I. System typów

```
// 1. Statyczne typowanie
let mut x: i32 = 10;
x = "var"; // panic!; mismatched type

// 2. Słabe typowanie
let x: i32 = 10;
let y: f32 = x + 2.0; // y = 12.0

// 3. Inferencja
let x = 10.3; // x: f32
```

II. Zmienne i typy danych

```
// 1. Niemutowalne zmienne
let x: i32 = 3;
let mut y: i32 = 4;
y = y + 1; // y = 5;
x = x + 1; // panic!: cannot assign twice to immutable variable 'x'

// 2. Typy danych
let x: i16 = 10;
let y: i64 = 4000000000;
let z: f32 = 3.14;
let d: bool = true;
let e: str = "Print\n\"line\"" // escaping \n \"

// 3. & 5. Kontekst oraz przesłanianie
let y: i32 = 10;

fn x() {
```

```
    let y: i32 = 20;
    let x: i32 = y + 0; // x = 20;
}

// 4. Usuwanie zmiennych po wyjściu z kontekstu
fn x(y: i32) {
    if (y > 10) {
        let z: i32 = 20;
    }

    let d: i32 = z + 0; // panic!: undefined variable 'z';
}
```

III. Operacje

```
// 1. Priorytety
let x: i32 = (1 * 3) + 4; // x = (1 * 3) + 4 => x = 3 + 4 => x = 7;

// 2. Wbudowane operacje
let y: f32 = 3.14 + 10; // y = 13.14;
let z: bool = y > 10 && y < 20; // z = true;
let d: str = "Hello" + "World"; // d = "HelloWorld";

// 3. Wywołanie funkcji wbudowanej
let x: i32 = (1 * 3) + 4; // x = __add(__mul(1, 3), 4);
let y: i32 = "Example" / "E"; // panic!: undefined operation "/" for "str" and "str"

fn __mul(x: str, y: str) -> str {
    // ...
}
```

IV. Konstrukcje

1. Konstrukcja warunkowa `if-else`

```
// <> - miejsce do wstawienia
// [] - opcjonalne
if (<condition>) {
    <if-block>
} [else {
    <else-block>
}]
```

- W przypadku spełnienia warunku konstrukcji warunkowej `<condition>` zostanie wykonany blok `<if-block>`.
- W przeciwnym wypadku zostanie wykonany blok `<else-block>`.

```
let x: i32 = 6;

if (x > 5) {
    println("gt");
} else {
    println("le");
}
```

2. Pętla `while`

```
// <> - miejsce do wstawienia
// [] - opcjonalne
while (<condition>) {
    <while-block>
}
```

- W przypadku spełnienia warunku `<condition>` zostanie wykonany blok `<while-block>`.
- Po każdym wykonaniu bloku `<while-block>` ponownie zostanie wykonany krok poprzedni.
- Wyjście z pętli (przekazanie sterowania dalej) może nastąpić jedynie poprzez niespełnienie warunku `<condition>`.

```
let mut x: i32 = 0;
while (x < 10) {
    x = x + 1;
    println("x = " + x);
}
```

3. Mechanizm panikowania

```
fn div(x: i32, y: i32) -> i32 {
    if (y == 0) {
        panic("Cannot divide by zero");
    }

    return x / y;
}

div(3, 0); // panic!: Cannot divide by zero. Panicked at line: 3, main.fhll
```

4. Definiowanie funkcji

```
// <> - miejsce do wstawienia
// [] - wielokrotność / opcjonalność
fn <name> ([, <arg>]) -> [type] {
    <fn-block>
}
```

- Funkcja definiowana jest poprzez nazwę `<name>` oraz zbiór typów argumentów `[<arg>]`.
- Argumenty przekazywane są przez wartość.
- Możliwe jest przeciążanie funkcji.
- Niemożliwe jest przeciążanie jedynie przez wartość zwracaną.

```
fn square(mut x: i32) -> i32 {
    x = x * x;
    return x;
}

let mut x: i32 = 0;
square(x);
let z: i32 = x; // z = 0;

fn square(x: f32) -> f32 {
    return x + 1.0;
}

fn square(x: i32) -> f32 {
    return x * x + 1.0;
} // Error: Cannot overload only by return value
```

V. Struktury

1. Struktury

```
// <> - miejsce do wstawienia
// [] - wielokrotność / opcjonalność
struct <name> {
    [<fieldName>: <fieldType>]
}
```

- Struktura posiada swoją nazwę `<name>` oraz listę publicznych pól `[<fieldName>]`.
- Każde pole struktury musi mieć określony typ.

```
struct Item {
    name: str;
    amount: i32;
}

let item = Item { name: "Axe"; amount: 1; };
println(item.name);
item.amount = 4;
```

2. Rekordy wariantowe

```
enum <name> {
    <variant> {
        <variantField>: <variantFieldType>
    }
}
```

- Rekordy wariantowe posiadają swoją nazwę `<name>` oraz mogą zawierać dowolną liczbę wariantów `<variant>`.
- Każdy wariant posiada listę swoich pól `[<variantField>]` wraz z ich typami.

```
enum Entity {
    Player {
        firstname: str;
    }

    Animal {
        name: str;
    }
}

let e: Entity = Entity::Player { firstname = "John"; };
if (e is Entity::Player) {
    let f = e as Entity::Player;
}
```

3. Struktury jako typy zdefiniowane przez użytkownika

```
fn calculate_cost(item: Item, cost_per_item: i32) -> i32 {
    return item.amount * cost_per_item;
}
```

VI. Inne

1. Komentarze

```
// comment
```

2. Rzutowanie automatyczne

```
let y: f32 = 1; // y = 1.0;
```

3. Wbudowane funkcje i/o

```
println("Hello World");
let x: i32 = readi32();
```

4. Rzutowanie typów

```
let x: 3 as f32;
```

Gramatyka

Znaki

```
letter              ::== "a" ... "z" | "A" ... "Z";
digit               ::== "0" ... "9";
unicode             ::== // whole unicode;
```

Symbole terminalne

```
identifier          ::== letter, { letter | digit };

builtin_type        ::== "u16"
                      | "u32"
                      | "u64"
                      | "i16"
                      | "i32"
                      | "i64"
                      | "f32"
                      | "bool"
                      | "str";

keyword             ::== "fn"
                      | "struct"
                      | "enum"
                      | "mut"
                      | "let"
                      | "is"
                      | "if"
                      | "while"

integer_literal     ::== "0"
                      | ("1" ... "9"), { digit };
float_literal       ::== integer_literal, ".", digit, { digit };
string_literal      ::== \" { unicode } \";
literal             ::== integer_literal
                      | float_literal
                      | string_literal;

and_op              ::== "&&";
or_op               ::== "||";
relation_op         ::== "=="
```

```
                     | "!="
                     | "<"
                     | ">";
additive_op        ::== "+"
                     | "-";
multiplicative_op  ::== "*"
                     | "/";
unary_op           ::== "-"
                     | "!";
```

Symbole nieterminalne

```
Program             ::== { FunctionDeclaration | StructDeclaration | EnumDeclaration };

FunctionDeclaration ::== "fn", identifier, "(", [ Arguments ], ")", [ "->", Type ], Block;
Arguments           ::== Argument, { ",", Argument };
Argument            ::== [ "mut" ], identifier, ":", Type;

StructDeclaration   ::== "struct", identifier, "{", { FieldDeclaration }, "}";
FieldDeclaration    ::== identifier, ":", Type, ";";

EnumDeclaration     ::== "enum", identifier, "{", { VariantDeclaration }, "}";
VariantDeclaration  ::== identifier, "{", { FieldDeclaration }, "}";

Block               ::== "{", StatementList, "}";

StatementList       ::== { Statement, ";" };
Statement           ::== Declaration
                      | Assignment
                      | FnCall
                      | NewStruct
                      | Block
                      | ReturnStatement
                      | IfStatement
                      | WhileStatement;

Declaration         ::== [ "mut" ], "let", identifier, [ ":", Type ], [ "=", Expression ];
Assignment          ::== Access, "=", Expression;
FnCall              ::== identifier, "(", [ FnArguments ], ");
FnArguments         ::== Expression, { ",", Expression };
NewStruct           ::== VariantAccess, "{", [ Assignment, ";" ], "}"
ReturnStatement     ::== "return", [ Expression ];
IfStatement         ::== "if", "(", Expression", ")", Block, [ "else", Block ];
WhileStatemtn       ::== "while", "(", Expression, ")", Block;

Access              ::== identifier, { ".", identifier };
VariantAccess       ::== identifier, { "::", identifier };
Type                ::== builtin_type
                      | VariantAccess;

Expression          ::== AndExpression, { or_op, AndExpression };
AndExpression       ::== RelationExpression, { and_op, RelationExpression };
RelationExpression  ::== AdditiveTerm, [ relation_op, AdditiveTerm ];
AdditiveTerm        ::== MultiplicativeTerm, { additive_op, MultiplicativeTerm };
```

```
MultiplicativeTerm  ::== UnaryTerm, { multiplicative_op, UnaryTerm }
UnaryTerm           ::== [ unary_op ], Term;
Term                ::== literal
                       | Access, [ "is", Type ], [ "as", Type ]
                       | FnCall
                       | NewStruct
                       | "(", Expression, ")";
```

Obsługa błędów

Lekser

- W przypadku napotkania nieprawidłowego symbolu lub ciągu znaków, lekser zgłasza błąd leksykalny `LexerError.UnexpectedCharacter`
- Lekser zgłasza błędy zbyt długiego lub nieprawidłowego identyfikatora: `LexerError.InvalidIdentificator`
- Lekser zgłasza błędy niezamkniętego ciągu znaków: `LexerError.UnclosedStringLiteral`
- Zgłaszane błędy zawierają informację o położenie problematycznego znaku
- Zgłoszenie błędu kończy dalszą analizę programu

Parser

- Parser zgłasza błędy związane z nieprawidłową składnią `ParserError.SyntaxError`
- Błędy zawierają informacje o nieoczekiwanym tokenie, oczekiwanym tokenie oraz obowiązkowo o lokalizacji
- Zgłoszenie błędu kończy dalszą analizę programu

Interpreter

- Interpreter pełni rolę również analizatora semantycznego
- Interpreter w przypadku wystąpienia błędu będzie zwracać stosowny komunikat w zależności od błędu:
  - `InterpreterError.NameError` w przypadku braku nazwy np. zmiennej lub funkcji
  - `InterpreterError.TypeError` w przypadku nieprawidłowego typu, którego nie można automatycznie skonwertować
- Błedy zawierają informację o położeniu oraz kontekst wystąpienia danego problemu np. `undefined name 'x'`
- Zgłoszenie błędu kończy dalszą analizę programu

Mechanizm panikowania

- Mechanizm panikowania kończy interpretację programu oraz wyświetla stosowny komunikat np.: `panic!: unknown variable 'z' at <line>:<column>`
- Mechanizm wykorzystywany jest przez funkcje wbudowane oraz może być używany przez użytkownika

Analiza wymagań funkcjonalnych

---

1. Interpreter musi przechowywać informacje o typie oraz wartości zmiennych.
   - Informacje o typie są wykorzystywane do dokonywania automatycznej konwersji typów.
2. Interpreter musi posiadać wbudowane funkcje konwersji typów pomiędzy typami wbudowanymi.

from to	i16	i32	i64	u16	u32	u64	f32	bool	str
i16		+	+	+	+	+	+	*2	+

from to	i16	i32	i64	u16	u32	u64	f32	bool	str
i32	*1		+	*1	+	+	+	*2	+
i64	*1	*1		*1	*1	+	+	*2	+
u16	*1	+	+		+	+	+	*2	+
u32	*1	*1	+	*1		+	+	*2	+
u64	*1	*1	*1	*1	+		+	*2	+
f32	+	+	+	+	+	+		*2	+
bool	-	-	-	-	-	-	-		+
str	-	-	-	-	-	-	-	*3	

*1 - konwersja może prowadzić do utraty danych lub przepełnienia

*2 - wartość prawda dla niezerowych wartości

*3 - wartość prawda dla niepustego ciągu znaków

3. Interpreter musi implementować podstawowe operacje dla typów wbudowanych.
   - Interpreter wyszukuje implementację danej operacji dla danych typów wejściowych.
   - W przypadku nie znalezienia takiej implementacji, próbuje dokonać konwersji drugiego argumentu tak, aby miał typ zgodny z pierwszym argumentem.
   - W przypadku niepowodzenia, sprawdzane są możliwe konwersje drugiego, a następnie pierwszego argumentu, np. `3 && 4` - zostanie wywołane `__and(bool, bool) -> bool`.
   - Jeżeli operator przyjmuje wiele typów i pozwala na zwracanie wielu, zwracany jest typ zgodny z pierwszym argumentem.

operator(y)	argument	argument	typ zwracany
+ - * /	i16 / i32 / i64 / u16 / u32 / u64	i16 / i32 / i64 / u16 / u32 / u64	i16 / i32 / i64 / u16 / u32 / u64
+ - * /	f32	i16 / i32 / i64 / u16 / u32 / u64	f32
< >	i16 / i32 / i64 / u16 / u32 / u64 / f32	i16 / i32 / i64 / u16 / u32 / u64 / f32	bool
&&	bool	bool	bool
+	str	str	str
== !=	any	any	bool

4. Interpreter musi implementować podstawowe funkcje wbudowane.

funkcja	argument(y)	typ zwracany
println	str	-
readU64	-	u64
readI64	-	i64
readStr	-	str

Konfiguracja oraz uruchomienie

Pliki konfiguracyjne

- Konfiguracja interpretera odbywa się za pomocą pliku w formacie JSON o następującej strukturze:

```
{
    "identifierMaxLength": 32,
    "recursionMaxDepth": 32
}
```

- `identifierMaxLength`: Określa maksymalną długość identyfikatora.
- `recursionMaxDepth`: Określa maksymalne zagnieżdżenie w rekursji.

Uruchomienie

- Interpretacja pliku odbywa się poprzez wywołanie polecenia:

```
fhll <filename>
```

- Możliwe jest również uzyskanie listy tokenów lub drzewa składniowego programu poprzez flagi:

```
fhll <filename> --display-tokens
fhll <filename> --display-syntax-tree
```

- W przypadku korzystania z pliku konfiguracyjnego, można to zrobić za pomocą flagi `--config`:

```
fhll --config myconfig.json
```

Realizacja

Środowisko

Język: Python 3.12+

Biblioteki: pytest

Struktura

1. **BufferRead**
   - Odpowiedzialny za odczyt znaków z wejścia programu lub innych źródeł danych.
   - Pozwala na buforowanie znaków.
   - Dokonuje automatycznej konwersji znaków końca linii (wsparcie dla `\r\n`, `\n`).
   - Informuje o położeniu znaku w pliku - linia, kolumna oraz offset od początku pliku.
2. **Lexer**
   - Odpowiedzialny za analizę źródłowego kodu i przekształcenie go na sekwencję tokenów.
   - Generowanie tokenów wykonywane jest na żądanie.
   - Komunikuje się z BufferRead oraz Modułem Błędów.
3. **Parser**
   - Odpowiedzialny za analizę składniową tokenów wygenerowanych przez lekser i przekształcenie ich w formę drzewa składniowego.
   - Komunikuje się z Lekserem oraz Modułem Błędów.
4. **Interpreter**
   - Odpowiedzialny za wykonanie kodu źródłowego na podstawie reprezentacji w postaci drzewa składniowego.
   - Komunikuje się z Parserem oraz Modułem Błędów.
5. **Moduł Błędów**
   - Odpowiedzialny za gromadzenie i wypisywanie informacji o wszystkich błędach, które wystąpiły podczas procesu interpretacji.
   - Pozwala na wyświetlanie dodatkowych informacji o błędzie, np. wskazówek.

Testowanie

Testy jednostkowe

1. BufferRead
   - Test 1: Odczyt kilku znaków z pliku.
   - Test 2: Odczyt kilku znaków z `stdout`.
   - Test 3: Sprawdzenie buforowania znaków.
   - Test 4: Sprawdzenie pozycji znaku w pliku.
   - Test 5: Reakcja na różne znaki końca linii.
2. Lexer
   - Test 1: Rozpoznawanie literałów liczbowych oraz znakowych.
   - Test 2: Rozpoznawanie identyfikatorów oraz słów kluczowych.
3. Parser
   - Test 1: Budowa tabeli symboli.
   - Test 2: Rozpoznawanie wyrażeń na podstawie sekwencji tokenów.
   - Test 3: Rozpoznawanie przypisań na podstawie sekwencji tokenów.
   - Test 4: Rozpoznawanie definicji funkcji na podstawie sekwencji tokenów.
4. Interpreter
   - Test 1: Wykonanie wyrażeń na podstawie gotowego drzewa składniowego.
   - Test 2: Działanie mechanizmu panic.

Testy integracyjne

1. Lexer
   - Test 1: Tokenizacja całego pliku z programem.
2. Parser
   - Test 1: Budowa drzewa dla całego pliku z programem.
3. Interpreter
   - Test 1: Działanie dla kilku przykładowych programów - testowanie błędów oraz przypadków pozytywnych.

Testy end-to-end

- Sprawdzenie działania interpretera dla przykładowych programów:
  - Obliczanie silni rekurencyjnie.
  - Wykorzystanie pętli while.
  - Tworzenie prostego drzewa za pomocą struktur.
- Testowanie polegać będzie na podaniu wejścia na `stdin` i odczycie (z timeoutem) wyników działania.