# ZPR - Dokumentacja końcowa

- Autor: Łukasz Suchołbiak

## Temat projektu

Celem projektu było stworzenie:

1. **Języka programowania**, który:
   - posiada statyczne i silne typowanie,
   - traktuje zmienne jako niemutowalne domyślnie,
   - przekazuje zmienne do funkcji przez referencję,
   - wspiera funkcje wyższego rzędu,
     - oraz udostępnia dwa wbudowane operatory wyższych funkcji (wybrane: *bind front* i *kompozycja funkcji*).
2. **Interpretera** obsługującego ten język, który potrafi przyjmować dane wejściowe zarówno z pliku, jak i ze standardowego wejścia.

Szczegółowa specyfikacja języka znajduje się w pliku `./docs/TKOM-dokumentacja-końcowa.md`.

---

## Wykorzystane narzędzia

- **C++20**
  - **Boost** (w wersji co najmniej 1.88) – wykorzystano moduły `program_options` oraz `test`.
  - **spdlog** – do logowania.
  - **fmt** – do formatowania tekstu.
- **clang-format** – do formatowania kodu (konfiguracja w pliku `.clang-format`).
- **cmake** – do budowania i linkowania projektu oraz testów.
- **doxygen** – do generowania dokumentacji z kodu źródłowego.
- **lcov** – do generowania raportów pokrycia kodu testami.
- **just** – do automatyzacji zadań (kompilacja, testy, uruchamianie, generowanie dokumentacji i pokrycia kodu).

---

## Zrealizowana funkcjonalność z dokumentacji wstępnej

### Odniesienie do wymagań języka

W trakcie trwania projektu w niewielkim stopniu zmieniła się specyfikacja języka, jednak zmiany te (językowe) nie naruszają żadnych wymagać z dokumentacji wstępnej.

#### Wymagania

1. Język statycznie typowany - wymaganie spełnione
2. Język silnie typowany - wymaganie spełnione
3. Zmienne w języku domyślnie niemutowalne - wymaganie spełnione
4. Wsparcie dla mechanizmu funkcji wyższego rzędu - wymaganie spełnione
5. 2 wbudowane operatory funkcji wyższego rzędu - spełnione - bind front oraz kompozycja funkcji

### Odniesienie do wymagań interpretera

Niespełnione zostało wymaganie o parametryzowalne ograniczenia na długość identyfikatorów oraz stałych testowych
Kolejną zmianą jest brak rozdzielenia analizy semantycznej i interpretacji w wersji finalnej - przy projekcie wstępnym niejasny był jeszcze sposób realizacji późniejszych etapów.

#### Wymagania

- Analizator leksykalny

    1. Obsługa 2 typów źródeł - wymaganie spełnione - z pliku oraz z wejścia standardowego
    2. Leniwa tokenizacja - wymaganie spełnione - generowanie tokenów na żadanie parsera
    3. Obsługa escapingu w ciągach tekstowych - wymaganie spełnione
    4. Parametryzowalne ograniczenia na długość identyfikatorów, oraz stałych tekstowych - wymaganie **nie spełnione**
- Analizator składniowy
    1. Weryfikacja poprawności składniowej - wymaganie spełnione
    2. Budowa drzewa składniowego/reprezentacji programu - wymaganie spełnione
    3. Możliwość zrzucania budowanej struktury drzewa na wyjście konsoli. - wymaganie spełnione - (printowane przy uruchomieniu programu z opcją `--verbose`)
- Interpreter
    1. Weryfikacja poprawności semantycznej reprezentacji programu - wymaganie spełnione
    2. Funkcje wbudowane traktowane jak użytkownika - wymaganie spełnione - są `Callable` - wywoływane przez `.call` tak samo jak użytkownika
- **Wymaganie przedmiotowe - projekt kompilowalny pod windowsem i linuxem:**
    - wymaganie spełnione
    - linux: w `g++-13`
    - windows: w `Visual Studio`

---

## Napotkane problemy w trakcie projektu i wnioski

W trakcie projektu główny problem stanowiła dla mnie **implementacja modułu Parsera**. Wprowadziła u mnie spore opóźnienie w stosunku do przewidywanego harmonogramu i spowodowała nawarstwienie opóźnienia przy kolejnym etapie - interpreterze.

Kolejnym problemem była dla mnie **kompatybilność kodu pod windowsem i linuxem oraz zarządzanie zależnościami**. Przez większość czasu projektu korzystałem z clanga w wersji 19.7.1 i z `-stdlib=libc++`, co dawało mi dostęp do `std::ranges::for_each`. Aby program był kompilowalny przez `g++` oraz w `Visual studio` pod windowsem w finalnej wersji projektu przeszedłem na `std::for_each` z biblioteki standardowej. Tak na prawdę mogłem od początku "nie wydziwiać" i pracować pod `g++`.

Jeszcze innym problemem była **reprezentacja wartości w interpreterze**: wartość/odwołanie do zmiennej. Szczególnie zamienie wprowadziło zapewnienie tego, że funkcja przyjmująca argument niemutowalny może przyjąć referencję do zmiennej mutowalnej, ale wewnątrz tej funkcji zawartość nie może być zmodyfikowana. Rozwiązałem to wprowadzając obiekt `VariableHolder`, który trzyma wskaźnik do zmiennej(obiektu `Variable`) oraz informację, czy dany holder zezwala na modyfikację zmienną. Pomimo, że efekt osiągnięty przez to jest fajny i działa poprawnie uważam, że moje rozwiązanie jest mało finezyjne i skomplikowało znacznie zarządzanie zmiennymi i wartościami - wprowadziło sporo dodatkowego kodu - uważam, że można było to zrobić lepiej.

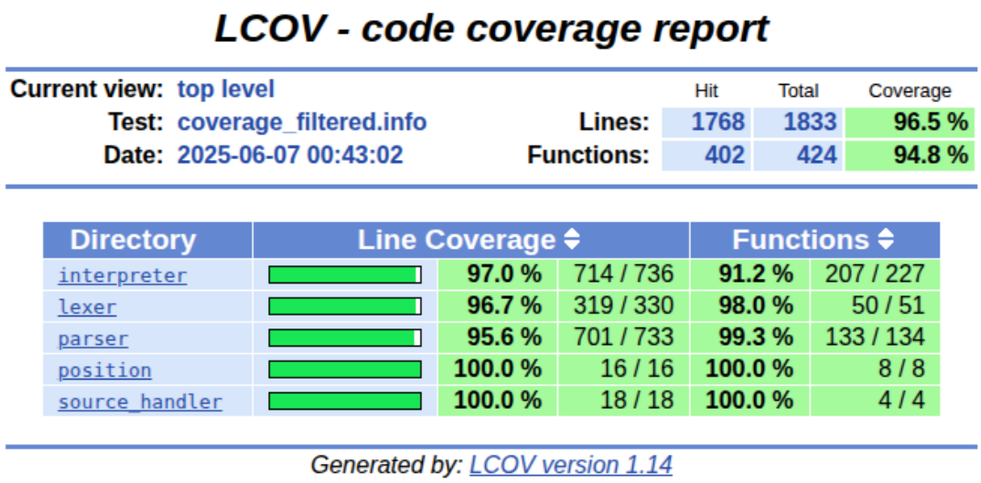
Projekt był zdecydowanie najbardziej pracochłonnym projektem na całych studiach, ale też najbardziej satysfakcjonującym.

---

## Statystyki

- Liczba linii w całym projekcie (włącznie z komentarzami i przerwami): `11564`, z czego:
    - w plikach nagłówkowych( `.hpp` ) - zawierają docstringi: `2660`
    - w plikach implementacji ( `.cpp` ): `3782`
    - w testach: `5122`
        - z czego `171` zajął wizytator wykorzystywany przy testach parsera
- Liczba godzin poświęconych na projekt: około `190`
- Coverage:
  - liczone bez:
  - plików nagłówkowych
  - modułu core stanowiącego obsługę cli programu
  - modułu exceptions posiadającego tylko konstruktory błędów
  - Line coverage: `96.5%`

- Function coverage: `94.8%`



LCOV - code coverage report

| | | | Hit | Total | Coverage |
|---|---|---|---|---|---|
| Current view: | top level | | | | |
| Test: | coverage_filtered.info | Lines: | 1768 | 1833 | 96.5 % |
| Date: | 2025-06-07 00:43:02 | Functions: | 402 | 424 | 94.8 % |

| Directory | Line Coverage | | Functions | |
|---|---|---|---|---|
| interpreter | 97.0 % | 714 / 736 | 91.2 % | 207 / 227 |
| lexer | 96.7 % | 319 / 330 | 98.0 % | 50 / 51 |
| parser | 95.6 % | 701 / 733 | 99.3 % | 133 / 134 |
| position | 100.0 % | 16 / 16 | 100.0 % | 8 / 8 |
| source_handler | 100.0 % | 18 / 18 | 100.0 % | 4 / 4 |

Generated by: LCOV version 1.14

---

## Testy

### Analizator leksykalny

Testy budowania pojedynczych tokenów, oraz budowanie tokenów dla przykładowego programu wraz ze sprawdzaniem zgodności pozycji zbudowanych tokenów

### Analizator składniowy

Testy z wykorzystaniem udawanego leksera z weryfikacją, czy zbudowana struktura przez parser jest taka, jak oczekiwana(z wykorzystaniem wizytatora do testów zczytującego strukturę programu).

### Interpreter

Testy interpretacji programu na podstawie tekstowego strumienia imitującego źródło przy wykorzystaniu `Lexera` oraz `Parsera`. Weryfikacja poprawności wyniku programu oraz przebadanie, czy sytuacje błędne skutkują rzuceniem odpowiednich wyjątków (Testy na około 2000 linii).
Testy poszczególnych metod wykorzystywanych przez interpreter.

---

## Czynności projektowe przy użyciu `just`

### Budowanie projektu

W katalogu głównym repozytorium należy uruchomić:

```
just build
```

Gotowa kompilacja znajdzie się w katalogu `build/`.

### Uruchamianie interpretera

Możliwe są dwie metody:

1. Z użyciem `just`

```
just tkm_interpreter [file]
# lub z dodatkowymi opcjami
just tkm_interpreter "-v [file]"
```

Przykład:

```
just tkm_interpreter "-v ./example_programs/counter.tkm"
```

2. Ręcznie z katalogu `build/`

```
./tkm_interpreter [file] [options]
```

### Dostępne opcje

```
usage: ./tkm_interpreter [file] [options]
available options::
  -h [ --help ]         display help info
  -s [ --stdin ]        read data from standard input
  -v [ --verbose ]      enable verbosity
  --input arg           input filename
```

## Uruchamianie testów

Żeby uruchomić testy:

```
just test [target]
```

Możliwe targety:

- all - (default) - wszyskie testy
- source_handler
- position
- lexer
- parser
- interpreter

## Generowanie raportu pokrycia testami

Aby wygenerować raport pokrycia kodu:

```
just gen_coverage
```

Po wygenerowaniu raport automatycznie otworzy się w przeglądarce.

Jeśli raport został już wcześniej wygenerowany, można go otworzyć przez:

```
just disp_coverage
```

Jeśli nie istnieje, to zostanie wygenerowany od nowa.

## Generowanie dokumentacji z kodu

Aby wygenerować dokumentację:

```
just docgen
```

Dokumentacja zostanie zapisana w katalogu `doxydoc` i otworzy się automatycznie w przeglądarce.

Aby otworzyć wcześniej wygenerowaną dokumentację:

```
just open_doc
```

## Czyszczenie projektu

Aby wyczyścić projekt i usunąć wygenerowane dane (build, coverage, dokumentację):

```
just clean
```

Usuwane są: katalog `build`, katalog `out` (z pokryciem), pliki `coverage.info`, `coverage_filtered.info` oraz katalog `doxydoc`.