Politechnika Wrocławska

Wydział Mechaniczny

Praca dyplomowa inżynierska

Narzędzie do rysowania schematów blokowych

Autor: Jordan Wiszniewski
nr indeksu: 247859
Promotor: Dr inż. Wojciech Myszka
charakter pracy: projektowy

Rok akademicki 2021/2022

Spis treści

Rozdział 1

Wstęp

1.1 Cele pracy

Narzędzie powstało głównie w celach dydaktycznych – ma ona na celu ułatwienie wizualizacji i tym samym zrozumienie początkującemu programiście działania algorytmów utworzonych w językach bazujących na języku C poprzez bezpośrednie pokazanie schematu blokowego odnoszącego się do napisanego kodu. Dodatkowo jest to przydatne narzędzie umożliwiające tworzenie schematów blokowych w łatwy i szybki sposób, wymagający jedynie podstaw programowania do dowolnego zastosowania. Narzędzie powinno również umożliwiać użytkownikom łatwe udostępnianie schematów innym użytkownikom w formie, którą łatwo można poddać dalszej edycji.

1.2 Istniejące rozwiązania

Przykładowe rozwiązania dla rysowania schematów blokowych typu flowchart:

- Użycie dowolnego edytora graficznego - jest to najmniej wydajne rozwiązanie - rysowanie diagramów w ten sposób jest czasochłonne oraz trudne do ewentualnej edycji. Również wymaga od użytkownika dobrej znajomości edytora. Jedyną zaletą na tle innych rozwiązań jest nieograniczenie wyniku końcowego do predefiniowanych form - daję największą swobodę.

- Narzędzie specjalnie przeznaczone do rysowania za pomocą ręcznego umieszczania pojedyńczych bloczków oraz ich opisywania i łączenia np. lucidchart - dużo wydajniejsze niż w przypadku użycia edytora graficznego dzięki gotowym elementom, które użytkownik może umieszczać i edytować w dość dużym zakresie.

- Konwerter pseudo-kodu programistycznego wprost do schematu blokowego (którego wariantem jest ten projekt). Na rynku dostępna jest komercyjna wersja takiego konwertera pod nazwą code2flow. Zaletą tych rozwiązań jest wielokrotnie wyższa wydajność rysowania schematu oraz bezpośrednie odniesienie go do napisanego kodu co daje lepszą gwarancje poprawności rozwiązania oraz najłatwiejszą edycję spośród wszystkich dostępnych metod. Na niekorzyść tej metody działa jedynie jej mała elastyczność - użytkownik ma niewielki wpływ na końcowy wygląd schematu, jest to w dużej mierze wstępnie zdefiniowane przez twórców tego typu narzędzi.

1.3 Forma pracy

Praca ma charakter projektowy i polega na stworzeniu narzędzia, za pomocą którego będzie można w łatwy sposób narysować (wygenerować) schematy blokowe typu flowchart na podstawie elementów składniowych zaczerpniętych z języka C, takich jak:

- wywołanie funkcji lub przypisanie - jako bloczek procesu
- instrukcje warunkowe typu $if/else$ - bloczek decyzyjny z dwiema gałęziami reprezentującymi wykonywanie kolejnych instrukcji w zależności od warunku podanego wewnątrz bloczka decyzyjnego,
- instrukcja warunkowego wykonywania pętli $while$ - jako bloczek decyzyjny, wraz z pętlą wskazującą na ten bloczek po zakończeniu instrukcji zawartych poniżej.

Generator (podobnie jak kompilator języka C) obsługuje zagnieżdżenie w sobie powyższych instrukcji (tzn. można rozbudować drzewko decyzyjne umieszczając jedną instrukcję warunkową wewnątrz drugiej). Dodatkowo w związku z tym, że interfejsem aplikacji jest formularz HTML, wywołujący zapytania do serwera poprzez Rest API, po udostępnieniu aplikacji np. na platformie chmurowej, będzie można w łatwy sposób udostępniać utworzony schemat wraz z kodem na podstawie którego powstał za pomocą linku. Dodatkową funkcjonalnością jest również obsługa znaków Unicode, co umożliwia pisanie kodu z użyciem m.in. polskich znaków oraz zawijanie tekstu w bloczkach, zapobiegając nadmiernemu rozrostowi bloczków przy większej ilości tekstu.

1.4 Wykorzystane narzędzia i technologie wraz z opisem

Wszystkie użyte technologie i narzędzia są open-source i do ich użytku (nawet komercyjnego) nie wymagają żadnych dodatkowych obligacji:

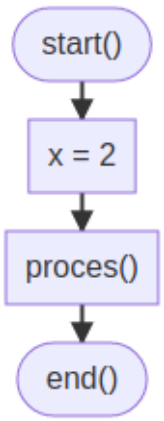
- HTML5, CSS, Thymeleaf - interfejsem aplikacji będzie prosty formularz w formie strony internetowej uzyskanej metodą szablonu HTML, komunikującej się z serwerem poprzez Rest API.
- Mermaid - narzędzie napisane w JavaScript umożliwiające rysowanie schematów na stronie HTML za pomocą specjalnej składni, unikalnej dla tego narzędzia.
- Java 11 SE, Spring Boot framework - część back-endowa aplikacji w którą m. in. wchodzą: obsługa zapytań Rest API, implementacja klas i zawartych w nich algorytmów interpretujących język C-podobny oraz zamieniających go na język wymagany przez narzędzie do rysowania schematów (Mermaid)
- Użyte narzędzia programistyczne: IntelliJ IDEA Community Edition, Visual Studio Code, system kontroli wersji Git.

Rozdział 2

Typy bloczków oraz ich przykładowe zastosowania

2.1 Bloczki procesu oraz start/end

Bloczek procesu jak nazwa wskazuje odpowiada za każdą akcje. W przypadku analizy kodu programistycznego bloczek ten odpowiada wywołaniu funkcji lub przypisaniu. Bloczek start/end w przypadku tej aplikacji od bloczka procesu różni się jedynie tym, że sygnalizuje on punkt startowy oraz wszystkie możliwe zakończenia algorytmu w danym schemacie. Aplikacja automatycznie zastępuje bloczki procesu występujące na początku i na końcu bloczkiem start/end. Dodatkowo aplikacja ignoruje wszystkie znaki typu *whitespace* . Jeśli chcemy wprowadzić znak spacji do schematu należy użyć znaku "_"(underscore), który zostanie zastąpiony. Aplikacja nie wymusza użycia elementów składniowych implikujących, że jest to wywołanie funkcji (zakończenie "...();") lub przypisanie (operator "="), do narysowania tego bloczku wymagane jest jedynie, by użyte były wyłącznie znaki należące do alfabetu w dowolnym języku (mające swój odpowiednik w Unicode),



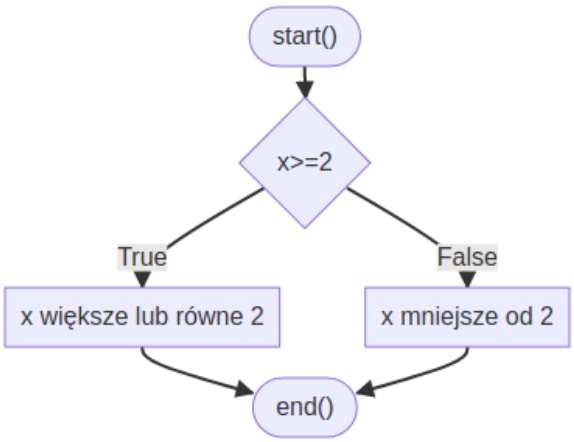

```
1 start();
2 x _ = _ 2;
3 proces();
4 end();
```

Rysunek 2.1: Bloczki procesu, start/end wraz z kodem, który je generuje

2.2 Bloczek warunkowy

- dla instrukcji $if - else$:

  Aplikacja rozpoznaje składnie instrukcji i z podanego warunku tworzy bloczek decyzyjny. Następnie tworzy dwa rozgałęzienia odpowiadające za akcje (dowolna ilość bloczków procesu) wykonywane w przypadku spełnienia warunku (bezpośrednio pod scope'm intrukcji if) lub nie spełnienia (pod scope'm instrukcji $else$, a następnie instrukcje znajujące się poza intrukcją if)

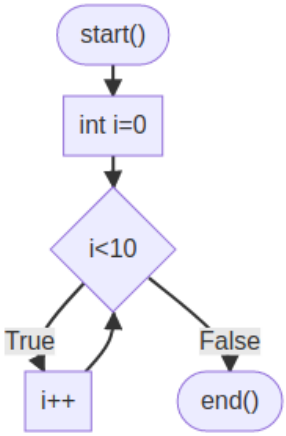



```
start();
if(x >= 2){
  x_większe_lub_równe_2;
}
else{
  x_mniejsze_od_2;
}
end();
```

Rysunek 2.2: Bloczek warunkowy utworzony przy użyciu instrukcji if

- dla instrukcji $while$:

  Interpretacja warunku działa analogicznie jak przy instrukcji if, podobnie również działa dołączanie kolejnych instrukcji wykonujących się po spełnieniu warunku z tą różnicą, że ostatni proces tej gałęzi łączony jest automatycznie z bloczkiem decyzyjnym instrukcji $while$. Nie spełnienie warunku odpowiada gałęzi na której znajdują się procesy umieszczone poza scope'm tego typu instrukcji warunkowej.

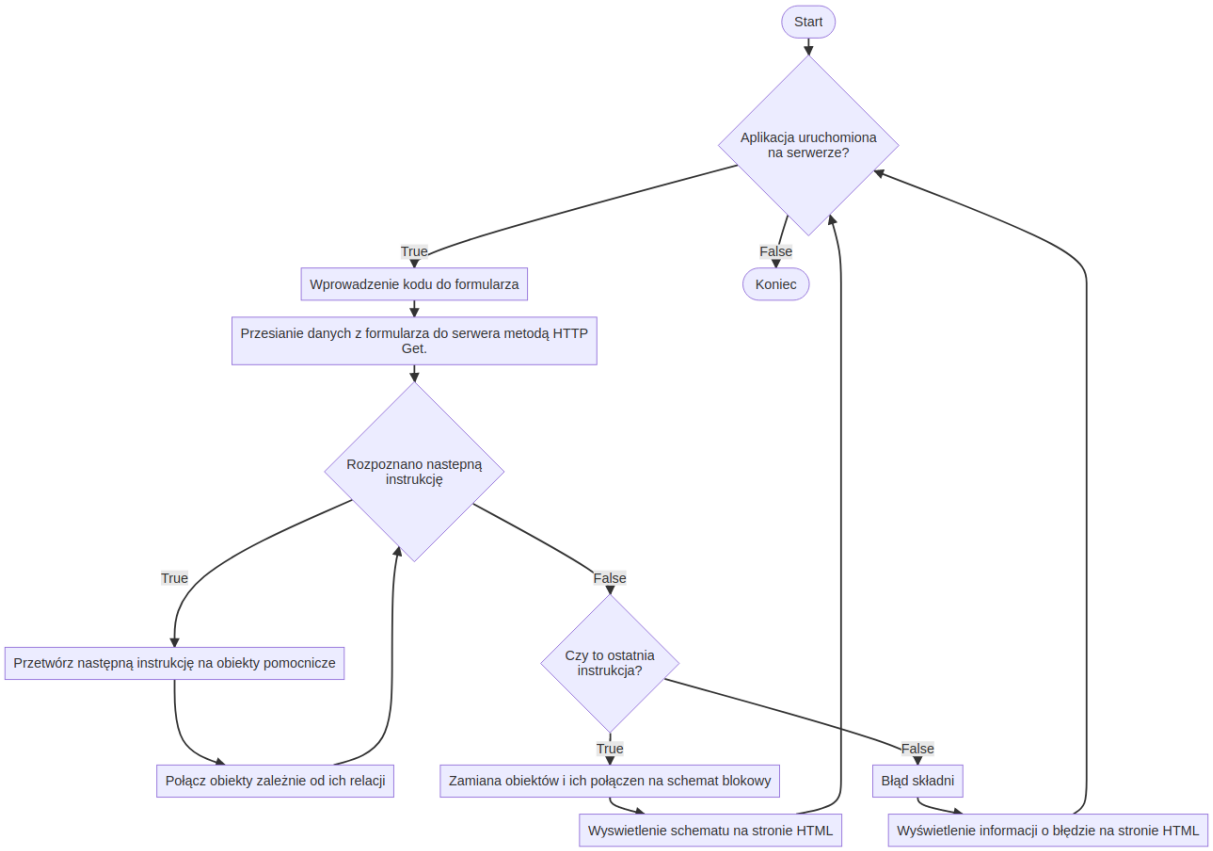



```
start();
int_ i = 0;
while(i < 10){
  i++;
}
end();
```

Rysunek 2.3: Bloczek warunkowy utworzony przy użyciu instrukcji $while$

2.3 Bardziej złożony przykład

Schemat blokowy przedstawiający (uproszczony) algorytm działania aplikacji w której został narysowany:




```
Start;
while(Aplikacja _ uruchomiona _ na _ serwerze?){
    Wprowadzenie_ kodu_ do_ formularza;
    Przesianie_ danych_ z_ formularza_ do_ serwera_ metodą_ HTTP_ Get.;
    while(Rozpoznano_ nastepną_ instrukcję){
        Przetwórz_ następną_ instrukcję_ na_ obiekty_ pomocnicze;
        Połącz_ obiekty_ zależnie_ od_ ich_ relacji;
    }
    if(Czy_ to_ ostatnia_ instrukcja?){
    Zamiana_ obiektów_ i_ ich_ połączeń_ na_ schemat_ blokowy;
    Wyswietlenie _ schematu _ na _ stronie _ HTML;
    } else {
        Błąd _ składni;
        Wyświetlenie _ informacji _ o _ błędzie _ na _ stronie _ HTML;
    }
}
Koniec;
```

Rysunek 2.4: Przykład pokazujący obsługę zagnieżdżonych instrukcji warunkowych, zawijanie tekstu oraz wprowadzanie polskich znaków

Rozdział 3

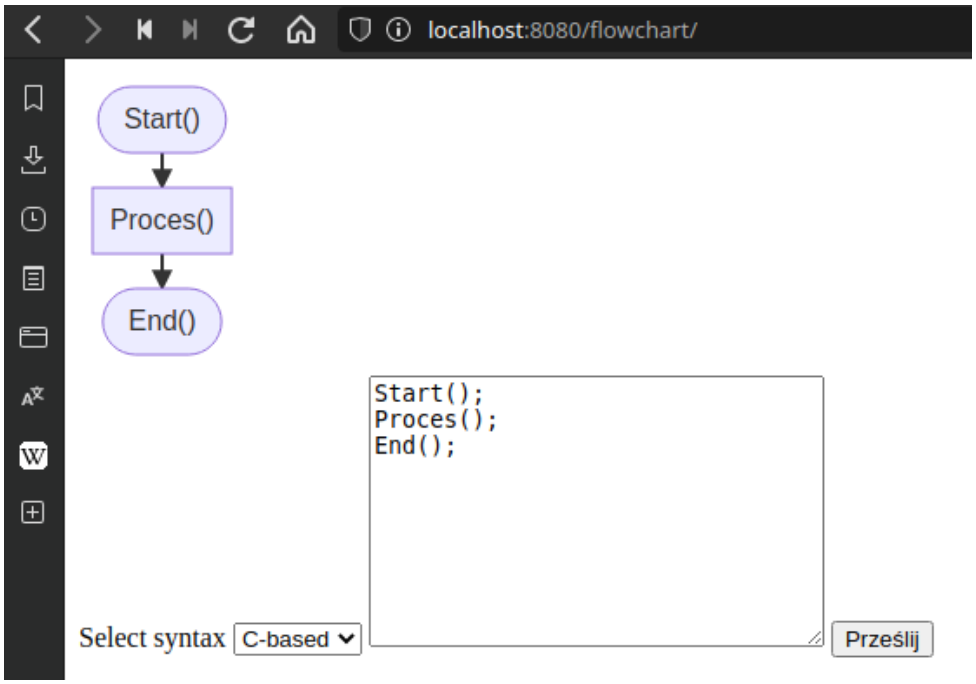
Opis działania aplikacji

Rozdział poświęcony jest ogólnemu opisowi działania poszczególnych fragmentów aplikacji zarówno od strony jej użytkownika jak i kodu źródłowego.

3.1 Interfejs użytkownika

Graficznym interfejsem użytkownika (GUI) jest prosty formularz HTML, domyślnie dostępny w oknie przeglądarki internetowej po uruchomieniu aplikacji pod adresem lokalnego hosta na porcie 8080:
localhost:8080/flowchart/.

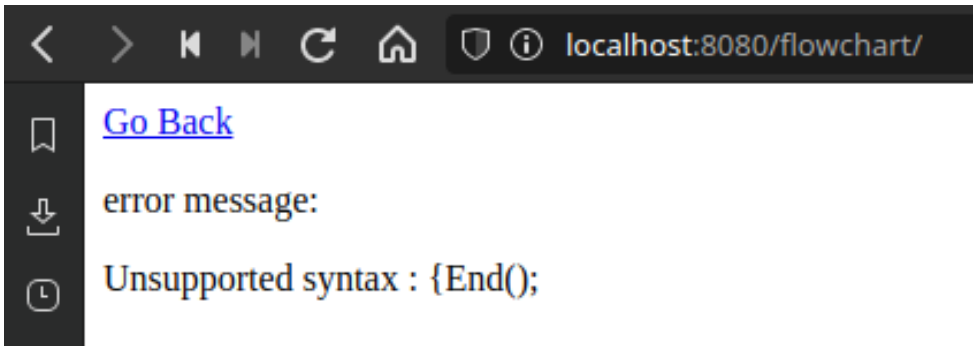
Domyślnie GUI zawiera w górnej części pole z wygenerowanym diagramem blokowym, a pod nim menu rozwijane umożliwiające wybór składni kodu zarówno Mermaid jak i C-podobnego (dropdown select menu), pole tekstowe (text area) do wprowadzania samego kodu oraz przycisk potwierdzający przesłanie typu wybranej składni i kodu przez Rest API. Żądanie typu GET obsługiwane przez kontroler Rest dopuszczający dwa opcjonalne parametry: typ składni (*type*) - domyślnie przyjmujący składnie Mermaid, kod umieszczony w polu tekstowym (*originalCode*) - domyślnie rysujący dwa proste bloczki. Jakiekolwiek wyrzucenie wyjątku przez program zostaje obsłużone właśnie w kontrolerze, po czym przekazana zostaje wiadomość w nim zawarta do ekranu błędu.



Rysunek 3.1: Interfejs graficzny użytkownika wyświetlony przy pomocy przeglądarki Vivaldi 5.

Przykładowy schemat blokowy dostępny pod adresem:
http://localhost:8080/flowchart/?type=C&originalCode=Start();Proces();End();

Dodatkowym ekranem jest strona HTML, która wyświetla komunikat o błędzie składniowym oraz pokazuje fragment kodu w którym ten błąd nastąpił.



Dla kodu:

```
1 Start();
2 Proces();{
3 End();
```

Rysunek 3.2: Przykład ekranu wyświetlającego błąd składniowy

```java
@GetMapping("/flowchart")
public ModelAndView flowchart(
    @RequestParam(value = "type", defaultValue = "mermaid") String type,
    @RequestParam(value = "originalCode", defaultValue = "A --> B") String code,
    ModelAndView mv
    ){
    try {
        mv.setViewName("flowchart.html");
        flowchartParser.setType(type);
        flowchartParser.code2flowchart(code);
        mv.addObject("flowchart", flowchartParser);
        return mv;
    } catch (Exception e){
        mv.setViewName("errorpage.html");
        mv.addObject("message", e.getMessage());
        return mv;
    }
}
```

Rysunek 3.3: Kontroler Rest obsługujący zapytania metodą GET protokołu HTTP z dwoma parametrami, który przekierowuje na stronę z narysowanym schematem lub ekran błędu

Do lepszego śledzenia przebiegu działania programu i ewentualnych błędów w terminalu, w którym uruchomiona jest aplikacja wyświetlają się szczegółowe logi z każdego etapu działania programu:

```
2021-12-17 00:04:31.451  INFO 39612 --- [nio-8080-exec-6] c.e.m.r.CParser
    : program instruction of type: EXPRESSION originalCode: Start();Proces();{End();
2021-12-17 00:04:31.452  INFO 39612 --- [nio-8080-exec-6] c.e.m.m.cToMermaidConverter
    : expression entered: "Start()"
2021-12-17 00:04:31.452  INFO 39612 --- [nio-8080-exec-6] c.e.m.r.CParser
    : program instruction of type: EXPRESSION originalCode: Proces();{End();
2021-12-17 00:04:31.452  INFO 39612 --- [nio-8080-exec-6] c.e.m.m.cToMermaidConverter
    : expression entered: "Proces()"
2021-12-17 00:04:31.452  INFO 39612 --- [nio-8080-exec-6] c.e.m.r.CParser
    : program instruction of type: UNKNOWN originalCode: {End();
```

Rysunek 3.4: Przykładowy zapis w terminalu z przebiegu programu zakończonego błędem.

3.2 Lexer i parser kodu języka C-podobnego

W języku Java 11 SE stworzony został:

- Lexer kodu zintegrowany jest z parserem i jego podstawowym zadaniem jest rozpoznawanie instrukcji obsługiwanych przez aplikacje za pomocą wyrażeń regularnych oraz zapewnienia zrównoważonego użycia nawiasów zarówno okrągłych jak i klamrowych w odpowiedniej kolejności, zgodnie z zasadami programowania języka C.

```
ELSE_PATTERN = "\\}else\\{.*";
IF_PATTERN = "if\\(.*";
WHILE_PATTERN = "while\\(.*";
EXPRESSION_PATTERN = "([\\w\\+\\-=\\(\\)\\.,<>/]+?;).*";
END_SCOPE_PATTERN = "\\}.*";
```

Rysunek 3.5: Lista rozpoznawalnych przez lexer wyrazów regularnych (REGEX)

- Parser kodu napisany jest w oparciu o recursive descent parser użyty m. in. w kompilatorach języka C takich jak: GCC oraz Clang. Polega on na rekursywnym wywoływaniu funkcji wczytującej kolejne instrukcje rozpoznane przez lexer oraz zależnie od rodzaju tych instrukcji wyciąga treść argumentu (w nawiasach okrągłych) oraz zawartość zakresu wewnętrznego (tzw. scope) instrukcji (w nawiasach klamrowych).

```
private void handleIfStatement(){
    String statement = programBuilder.toString();
    String condition = getStringInOuterParenthesis(statement);
    String expressions = getStringInOuterCurly(statement);
    programBuilder.delete(0, condition.length() + 5);
    onIfStatementEnter(condition, expressions);
    parseProgramInstruction();
}
```

Rysunek 3.6: Przykładowa funkcja, która izoluje z instrukcji if jej argument (condition) oraz zawartość jej obszaru wewnętrznego (expressions), a następnie przekazuje te wartości do tzw. metody obserwowanej (onIfStatementEnter)

Otwarcie nowego zakresu w tym przypadku możliwe jest jedynie w instrukcjach $if - else$ oraz $while$. Aby umożliwić obsługę zagnieżdżeń tych instrukcji, enum reprezentujący jej typ z każdorazowym wejściem w jej zakres wewnętrzny (scope) jest odkładany na stos, a z wyjściem z zakresu zdejmowany z tego stosu co ma na celu późniejsze użycie samoczynnie wywołujących się metod obserwowanych (observable) we wzorcu projektowym *Obserwator* (*Observer*). W przypadku instrukcji odpowiadającym procesom (dowolny tekst bez znaków specjalnych zakończony średnikiem) metoda obserwująca informuje jedynie o samym jej wywołaniu, natomiast w przypadku instrukcji warunkowych występują metody dedykowane wejściu oraz wyjściu z obszaru tej instrukcji. Domyślna metoda obserwowana jedynie wyświetla w terminalu podstawowe informacje o danej instrukcji, natomiast informacje te mogą być dowolnie wykorzystane wersji tej klasy z nadpisanymi definicjami jej metod obserwowanych.

```
public void onIfStatementEnter(String condition, String expressions){
        log.info("IF entered : " + condition + " than : " + expressions);
}

public void onIfStatementExit(){
        log.info("IF exited");
}

public void onElseStatementEnter(String expressions){
        log.info("ELSE entered : " + expressions);
}
```

Rysunek 3.7: Przykład domyślnych definicji metod obserwowanych dla instrukcji $if/else$

3.3 Akcelerator budowania kodu w języku wymaganym przez Mermaid

Do ułatwienia tworzenia schematów blokowych powstał akcelerator - pakiet, umożliwiający łatwe tworzenie i zarządzanie poszczególnymi elementami schematów blokowych, a następnie konwersja tych elementów w pamięci programu na kod docelowy, wymagany przez Mermaid. Składa się on z listy węzłów (nodes) odpowiadającym poszczególnym bloczkom oraz połączeniom (links) generowanym na podstawie relacji między tymi węzłami.

3.4 Konwerter kodu języka C-podobnego na język Mermaid