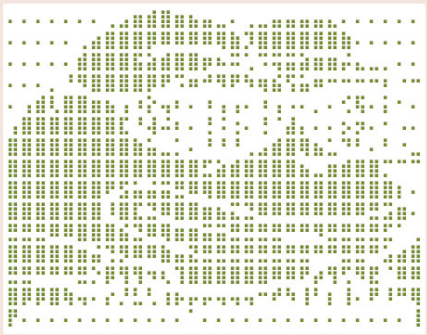
Mateusz Kotarba
Juliusz Kociński
Jakub Kierznowski

# Memecoding

## Wprowadzenie i poradnik do języka programowania GTL

The GreenText Programming Language



>use GTL
>profit

Opracowane na podstawie dokumentacji projektu GTL w ramach kursu "Kompilatory i Teoria Kompilacji

# Spis treści

# 1 Wstęp do GreenTextLang (GTL)

GreenText Programming Language (GTL) to wieloparadygmatowy język programowania zaprojektowany tak, aby pisanie w nim kodu przypominało tworzenie klasycznych internetowych historyjek w formacie "green text", spopularyzowanych przez portale takie jak 4chan. GTL, mimo że jest w pełni funkcjonalnym językiem, został stworzony głównie dla zabawy i unikalnego doświadczenia programistycznego, a niekoniecznie do tworzenia kodu produkcyjnego.

Język GTL jest implementowany w Javie przy użyciu ANTLR4. Wiele zaawansowanych funkcji i łatwość użycia zostały czasem poświęcone na rzecz podobieństwa do oryginalnego stylu green text.

## 1.1 Główna zasada GTL

Każda linia kodu w GTL musi być rozpoczęta sekwencją znaków: **spacja**, znak >, a następnie **co najmniej jedna spacja**. Na przykład:

```
> to jest poprawna linia kodu GTL
```

## 1.2 Hello World

Tradycyjny program "Hello, World!"w GTL może wyglądać następująco:

```
> spit "Hello, World!"
```

Co wyświetli na ekranie:

```
Hello, World!
```

Jeśli chcemy od razu nadać mu formę historyjki (np. w głównej funkcji programu, która często nazywa się ‘me‘):

```
> be me
> spit "Hello, World!"
> profit
```

Ten program również wyświetli:

```
Hello, World!
```

Jednakże kluczowa linia będzie zamknięta w funkcji głównej.

### Ćwiczenie

Spróbuj napisać program, który wyświetli Twoje imię na ekranie.

# 2 Czytanie i Wyświetlanie Danych (Input/Output)

GTL umożliwia interakcję z programem poprzez czytanie danych z standardowego wejścia (stdin) oraz wypisywanie danych na standardowe wyjście (stdout).

## 2.1 Czytanie z stdin: `swallow`

Polecenie `> swallow nazwa_zmiennej` służy do czytania danych wprowadzonych przez użytkownika. GTL próbuje automatycznie skonwertować wprowadzone dane na odpowiedni typ danych, jeśli to możliwe. Domyślnie dane są traktowane jako tekst (string).

## 2.2 Wypisywanie na stdout: `spit`

Do wypisywania danych na ekran używamy polecenia `> spit wartość_lub_zmienna`. Wartości są konwertowane na tekst (jeśli nim nie są) i wyświetlane.

**Przykład użycia `spit` i `swallow`:**

```
> be io_example
> # Przyklad 1: Czytanie i wypisywanie tekstu
> hear user_name
> spit "Jak masz na imie?"
> swallow user_name # Program czeka na wpisanie danych
> spit "Witaj, "
> spit user_name
> spit "!"

> # Przyklad 2: Czytanie liczby i operacja
> see number_in
> spit "Podaj liczbe, a ja pomnoze ja przez 2:"
> swallow number_in
> see result is number_in breeding like 2 times
> spit "Wynik to: "
> spit result
> profit
```

**Oczekiwany wynik programu (zakładając wpisanie "Tester"i potem 21):**

```
Jak masz na imie?
Tester
Witaj,
Tester
!
Podaj liczbe, a ja pomnoze ja przez 2:
21
Wynik to:
42
```

W przypadku, gdy interaktywne wprowadzanie danych przez `swallow` nie jest dostępne lub podczas testowania skryptów, dane wejściowe można zasymulować, np. przekazując

je do programu przez standardowe wejście (piping) lub modyfikując kod tak, by zmienne były inicjalizowane bezpośrednio.

### Ćwiczenie

Napisz program, który:

1. Prosi użytkownika o podanie jego ulubionego koloru (użyj `spit` do wyświetlenia prośby).
2. Wczytuje odpowiedź do zmiennej typu `hear` (użyj `swallow`).
3. Prosi użytkownika o podanie jego szczęśliwej liczby (użyj `spit`).
4. Wczytuje liczbę do zmiennej typu `see` (użyj `swallow`).
5. Wyświetla komunikat: "Twój ulubiony kolor to [kolor], a Twoja szczęśliwa liczba to [liczba]."(użyj `spit`).

Pamiętaj, aby symulować wprowadzanie danych przez przypisanie, jeśli testujesz w środowisku bez interaktywnego wejścia lub jeśli `swallow` nie działa interaktywnie w Twoim interpreterze.

# 3 Zmienne i Typy Danych

W GTL zmienne deklaruje się używając specyficznych słów kluczowych odpowiadających typom danych. Słowa kluczowe te odpowiadają różnym zmysłom i doświadczeniom:

## 3.1 Podstawowe typy danych

- `see`: dla liczb całkowitych (integer).
- `taste`: dla liczb zmiennoprzecinkowych (double).
- `hear`: dla ciągów znaków (string).
- `smell`: dla wartości logicznych (boolean).
- `spot`: dla struktur danych (structs).

## 3.2 Deklarowanie i przypisywanie wartości

Zmienną deklaruje się według schematu: `> typ nazwa is wartość`. Słowo kluczowe `is` służy do przypisania wartości. Jeśli wartość nie jest podana, zmienna zostanie zainicjalizowana wartością domyślną dla danego typu.

**Przykłady deklaracji zmiennych:**

```
 > be example_vars
Deklaracja liczby calkowitej
 > see my_age is 25
Deklaracja liczby zmiennoprzecinkowej
 > taste pi_value is 3.14159
Deklaracja ciagu znakow
 > hear greeting is "Witaj w GTL!"
Deklaracja wartosci logicznej
 > smell is_gtl_fun is c: # c: oznacza true

 > spit my_age
 > spit pi_value
 > spit greeting
 > spit is_gtl_fun
 > profit
```

**Oczekiwany wynik programu:**

```
25
3.14159
Witaj w GTL!
c:
```

## 3.3 Konwersja Typów (Casting)

GTL obsługuje zarówno jawną, jak i niejawną konwersję typów.

### 3.3.1 Jawne Castowanie (Eksplicytne)

Jawne rzutowanie typów wykonuje się za pomocą frazy `let me`.

- **Składnia:** `let me <typ> <wyrażenie_atom>`
- **Działanie:** Konwertuje wartość wynikającą z `<wyrażenie_atom>` na określony `<typ>`.
- **Przykłady w GTL:**

```
> be explicit_cast_example
> # Rzutuje string "123" na typ see (integer)
> see my_int is let me see "123"
> spit my_int

> hear num_str is "45.67"
> # Rzutuje string num_str na typ taste (double)
> taste my_double is let me taste num_str
> spit my_double
> profit
```

**Oczekiwany wynik programu:**

```
123
45.67
```

### 3.3.2 Niejawne Castowanie (Automatyczne)

GTL często wykonuje niejawne konwersje typów w następujących sytuacjach:

- **Przypisania do zmiennych:** Gdy przypisujesz wartość do istniejącej zmiennej, GTL próbuje automatycznie skonwertować przypisywaną wartość do typu zmiennej.
- **Przypisania do elementów tablicy:** Wartość jest automatycznie konwertowana do typu elementów tablicy.
- **Operacje arytmetyczne i logiczne:** Typy mogą być promowane lub konwertowane, aby umożliwić operację (np. int do double w operacji z double). Jeśli konwersja nie jest możliwa, może zostać rzucony błąd.
- **Przekazywanie argumentów do funkcji:** Argumenty są konwertowane do typów oczekiwanych przez parametry funkcji.

**Przykład niejawnego castingu (w operacjach):**

```
> be implicit_cast_example
> see int_val is 10
> taste dbl_val is 5.5
> # int_val zostanie niejawnie rzutowany na double
> # przed operacja dodawania
```

```
 > taste result is int_val joined by dbl_val
 > spit result # Wynik: 15.5
 > profit
```

**Oczekiwany wynik programu:**

```
15.5
```

## 3.4 Kolekcje (Tablice)

GTL wspiera tablice statyczne i dynamiczne.

- `about`: Słowo kluczowe używane przy deklaracji tablicy o stałym rozmiarze. Po `about` podaje się wyrażenie określające rozmiar.
- `multiple`: Słowo kluczowe używane przy deklaracji tablicy dynamicznej.

**Przykład deklaracji tablic:**

```
 > be example_arrays
Tablica statyczna 5 liczb calkowitych
 > see about 5 numbers is 1, 2, 3, 4, 5
Tablica dynamiczna stringow
 > hear multiple names is "Anon", "Pepe", "Wojak"

 Dostp  do elementu tablicy (indeksowanie od 0)
 > spit numbers'th 0 # Wyswietli pierwszy element
 > spit names'th 1   # Wyswietli drugi element

Modyfikacja elementu tablicy
 > names'th 1 is "Chad"
 > spit names'th 1

 > profit
```

**Oczekiwany wynik programu:**

```
1
Pepe
Chad
```

Uwaga: Dostęp do elementów tablicy odbywa się za pomocą konstrukcji `nazwa_tablicy'th indeks`.

## 3.5 Dostęp przez kopię i referencję

Domyślnie w GTL zmienne są przekazywane i przypisywane przez kopię. Aby uzyskać dostęp przez referencję (głównie w kontekście argumentów funkcji), używa się frazy `someone elses` przed nazwą zmiennej. Pełna implementacja ogólnego przypisania przez referencję może nie być jeszcze dostępna.

### Ćwiczenie

Zadeklaruj tablicę statyczną trzech ulubionych potraw (jako ciągi znaków). Następnie wyświetl drugą potrawę z tej tablicy. Zmień pierwszą potrawę na inną i wyświetl ją ponownie.

# 4 Wyrażenia i Operacje Matematyczne

GTL obsługuje standardowe operacje matematyczne i logiczne, opakowane w charakterystyczny dla języka styl.

## 4.1 Wartości logiczne (booleans)

- `:c` oznacza `false` (fałsz).
- `c:` oznacza `true` (prawda).

## 4.2 Operatory porównania

- `>` (większy niż): `beats`
- `<` (mniejszy niż): `beaten by`
- `==` (równy): `vibe with`
- `!=` (różny od): `doesn't vibe with` (domniemane, nie wymienione wprost, ale typowe)
- `>=` (większy lub równy): `unbeaten by`
- `<=` (mniejszy lub równy): `doesn't beat`

Pamiętaj: Operator przypisania to `is`. Do porównania równości używaj `vibe with`.

## 4.3 Podstawowe operatory matematyczne

- Dodawanie (`X+Y`): `X joined by Y`
- Mnożenie (`X*Y`): `X breeding like Y times`
- Negacja (`-X`): `the literal opposite of X`
- Odejmowanie (`X-Y`): W GTL *de iure* nie istnieje, a *de facto* jest realizowane jako `X joined by the literal opposite of Y`, więc dodawanie liczby przeciwnej.
- Modulo (`X%Y`): `X whatever left from Y`
- Odwrotność (`1/X`): `X flipped`
- Dzielenie (`X/Y`): W GTL *de iure* nie istnieje, a *de facto* jest realizowane jako `X breeding like Y flipped times`, więc mnożenie przez odwrotność.

## 4.4 Nawiasy

Nawiasy `( )` są używane do grupowania wyrażeń w standardowy sposób, aby kontrolować kolejność operacji. Z uwagi na ich wybijający charakter często psujący spójność z historyjkowym stylem GTLa zalecamy ich użycie sporadycznie, tylko gdy jest to konieczne.

## 4.5 Kolejność działań

Standardowa kolejność działań jest zachowana:

1. Nawiasy `()`
2. Operatory unarne: `flipped`, `the literal opposite of`
3. Operatory multiplikatywne: `breeding like Y times`, `whatever left from Y`
4. Operatory addytywne: `joined by Y`
5. Operatory porównania: `beats`, `beaten by`, itd.
6. Logiczne NOT: `not`
7. Logiczne AND: `also`
8. Logiczne OR: `alternatively`

Operatory na tym samym poziomie pierwszeństwa są wykonywane od lewej do prawej.

**Przykład operacji matematycznych i logicznych:**

```
> be math_example
> see a is 10
> see b is 5
> see sum_val is a joined by b
> spit sum_val # Wynik: 15

> see product_val is a breeding like b times
> spit product_val # Wynik: 50

> see diff_val is a joined by (the literal opposite of b)
> spit diff_val # Wynik: 5

> taste num_c is 10.0
> taste num_d is 4.0
> taste div_val is num_c breeding like num_d flipped times
> spit div_val # Wynik: 2.5

> smell check_true is (a beats b) also (sum_val vibe with 15)
> spit check_true # Wynik: c:

> profit
```

**Oczekiwany wynik programu:**

```
15
50
5
2.5
c:
```

### Ćwiczenie

Napisz program, który oblicza pole prostokąta. Zadeklaruj dwie zmienne, `wtą` i `wtamtą`, przypisz im wartości, a następnie oblicz pole i wyświetl wynik. Dodatkowo, sprawdź, czy długość jest większa od szerokości i wyświetl wynik tego porównania.

# 5 Instrukcje Warunkowe

Instrukcje warunkowe w GTL przypominają standardowe konstrukcje `if-else_if-else`.

## 5.1 Składnia

Podstawowa struktura:

```
> implying {wyrazenie_logiczne_1}
> # Cialo instrukcji if
> or {wyrazenie_logiczne_2}
> # Cialo instrukcji else if
> or not
> # Cialo instrukcji else
> or sth # Zakonczenie bloku warunkowego
```

## 5.2 Szczegóły

- **Instrukcja `if`:**

  ```
  > implying {wyrazenie}
  > # Cialo instrukcji
  > or sth
  ```

  Blok kodu jest wykonywany, jeśli `wyrazenie` jest `c:` (true) lub niezerowe. Linia `> or sth` zamyka blok `if`, jeśli nie ma dalszych części `else if` / `else`.

- **Instrukcja `if / else`:**

  ```
  > implying {wyrazenie}
  > # Cialo if
  > or not
  > # Cialo else
  > or sth
  ```

  `> or not` przechodzi do bloku `else`. `> or sth` zamyka cały blok warunkowy.

- **Instrukcja `if / else if / else`:**

  ```
  > implying {wyrazenie1}
  > # Cialo if
  > or {wyrazenie2}
  > # Cialo pierwszego else if
  > or {wyrazenie3}
  > # Cialo kolejnego else if
  > or not
  ```

```
> # Cialo else
> or sth
```

---

**Przykład instrukcji warunkowej:**

```
> be conditional_example
> see score is 85

> implying score unbeaten by 90
> spit "Ocena A"
> or score unbeaten by 80
> spit "Ocena B"
> or score unbeaten by 70
> spit "Ocena C"
> or not
> spit "Ojjj sabiutko  poszo "
> or sth

> profit
```

---

**Oczekiwany wynik programu:**

```
Ocena B
```

### Ćwiczenie

Napisz program, który pyta użytkownika o jego wiek (użyj `spit` do zapytania i `swallow` do wczytania wieku do zmiennej typu `see`). Następnie, używając instrukcji warunkowych:

- Jeśli wiek jest mniejszy niż 18, wyświetl "Jesteś niepełnoletni."
- Jeśli wiek jest równy 18, wyświetl "Masz 18 lat!"
- Jeśli wiek jest większy niż 18, wyświetl "Jesteś pełnoletni."

(Pamiętaj o symulacji wprowadzania danych przez przypisanie, jeśli testujesz w środowisku bez interaktywnego wejścia, lub jeśli `swallow` nie działa interaktywnie w Twoim interpreterze GTL.)

# 6 Pętle

GTL posiada jeden podstawowy rodzaj pętli: `think that`, która jest odpowiednikiem pętli `while` znanej z innych języków programowania.

## 6.1 Składnia pętli `think that`

```
> think that {wyrazenie_logiczne}
> # Cialo petli
> reconsider # Zakonczenie petli
```

Pętla wykonuje się tak długo, jak `wyrazenie_logiczne` jest prawdziwe (tzn. nie jest `:c` (false) ani zerem). Ciało pętli stanowi [zakres lokalny](scopes.md). Jeśli warunek na początku nie jest spełniony, pętla nie wykona się ani razu.

**Przykład pętli `think that`:** Program, który wyświetla liczby od 1 do 5.

```
> be loop_example
> see counter is 1
> think that counter doesn't beat 5 # Warunek: counter <= 5
> spit counter
> counter is counter joined by 1 # Inkrementacja counter
> reconsider
> profit
```

**Oczekiwany wynik programu:**

```
1
2
3
4
5
```

### Ćwiczenie

Napisz program, który używa pętli `think that` do obliczenia sumy liczb od 1 do 10. Wyświetl końcową sumę.

# 7 Funkcje

Funkcje w GTL pozwalają na organizację kodu w reużywalne bloki.

## 7.1 Deklaracja funkcji

Funkcję deklaruje się zaczynając od `> be nazwa_funkcji`.

- **Typ zwracany i nazwa zmiennej zwracanej:** `> typ-ing nazwa_zmiennej_zwracanej` (np. `> seeing result` dla funkcji zwracającej `int` w zmiennej `result`).
- **Argumenty:** Deklarowane po słowie kluczowym `likes`. Każdy argument to `typ-ing nazwa_argumentu`. Argumenty oddziela się przecinkiem `,` lub słowem `and`. Można kontynuować listę argumentów w nowej linii, zaczynając ją od `and`.
- **Zakończenie funkcji i zwrot wartości:** Linia `> profit` kończy definicję funkcji i zwraca wartość zapisaną w zadeklarowanej zmiennej zwracanej. Oznacza to, że efektywnie funkcja ma jeden punkt wyjścia. Aby emulować wielokrotne zwroty, należy przypisywać różne wartości do zmiennej zwracanej w ciele funkcji, a `profit` wywołać na samym końcu.

Domyślne wartości parametrów nie są obecnie obsługiwane.

**Przykład prostej funkcji bez argumentów i wartości zwracanej:**

```
> be greet_user
> spit "Hello from function!"
> profit
```

**Przykład funkcji z argumentami i wartością zwracaną:**

```
> # Definicja funkcji dodajacej dwie liczby
> be add_numbers
> seeing sum_val # Zmienna, w ktorej bedzie przechowywany wynik
> likes seeing num1, seeing num2 # Argumenty
> sum_val is num1 joined by num2
> profit

> # Glowna czesc programu (symulacja funkcji main)
> be main_program
> see result_add
> # Wywolanie funkcji add_numbers
> call add_numbers regarding 10 and 20
> result_add is sum_val # Pobranie wyniku funkcji (jesli funkcja zwraca wartosc
  )
                        # W GTL, po wywolaniu funkcji, jej zmienna zwracana
                        # (tutaj sum_val z add_numbers) moze byc dostepna
                        # w zasiegu wywolujacym pod ta sama nazwa, jesli nie ma
   konfliktu.
                        # Lub, wynik jest przypisywany bezposrednio:
                        # result_add is call add_numbers regarding 10 and 20 (
  to zalezy od specyfiki interpretera)
```

```
                        # Zalozmy, ze wynik jest dostepny w zmiennej zwracanej
   funkcji
                        # lub przypisywany bezposrednio.
                        # Dla jasnosci, czsto wynik przypisuje si do nowej
   zmiennej:
> see number_a is 10
> see number_b is 20
> see calculation_result is call add_numbers regarding number_a and number_b
> spit calculation_result
> profit
```

---

Dokumentacja nie precyzuje dokładnie, jak wartość zwracana przez `call` jest przechwytywana. Przyjmijmy, że `call` zwraca wartość, którą można przypisać: Poprawiony przykład wywołania:

---

```
> # Definicja funkcji dodajacej dwie liczby
> be add_numbers
> seeing sum_val
> likes seeing num1, seeing num2
> sum_val is num1 joined by num2
> profit

> be main_program_corrected
> see number_a is 10
> see number_b is 20
> # Wywolanie funkcji i przypisanie wyniku
> see calculation_result is call add_numbers regarding number_a and number_b
> spit "Suma liczb:"
> spit calculation_result

> # Wywolanie funkcji bez argumentow
> call greet_user # zdefiniowanej wczesniej
> profit
```

---

**Oczekiwany wynik programu (dla poprawionego przykładu):**

```
Suma liczb:
30
Hello from function!
```

## 7.2 Wywołania funkcji

Funkcje wywołuje się za pomocą: `> call nazwa_funkcji regarding arg1, arg2, ...`. Jeśli funkcja nie przyjmuje argumentów, używa się: `> call nazwa_funkcji`.

### Ćwiczenie

Napisz funkcję o nazwie `multiply`, która przyjmuje dwie liczby całkowite jako argumenty (`factor1`, `factor2`) i zwraca ich iloczyn. Następnie w głównej części programu wywołaj tę funkcję z przykładowymi wartościami i wyświetl wynik.

# 8 Struktury Danych (structs)

GTL pozwala na definiowanie własnych złożonych typów danych, zwanych strukturami. Struktury mogą również zawierać metody, działając podobnie do klas w innych językach.

## 8.1 Definiowanie struktur

Definicję struktury rozpoczyna się od `> look around`, po czym w nowej linii podaje się nazwę struktury. Następnie deklaruje się pola (zmienne członkowskie) w standardowy sposób. Definicję kończy się linią `> lose interest`.

**Przykład definicji struktury:**

```
> look around
> Point # Nazwa struktury
> see x_coord
> see y_coord
> lose interest
```

To odpowiadałoby strukturze w C++:

```
struct Point {
    int x_coord;
    int y_coord;
};
```

## 8.2 Definiowanie struktur z metodami

Metody (funkcje członkowskie) deklaruje się wewnątrz definicji struktury, tak jak zwykłe funkcje.

**Przykład struktury z metodą:**

```
> look around
> Rectangle
> see width
> see height
> # Metoda obliczajaca pole
> be calculate_area
> seeing area_val
> area_val is width breeding like height times # Dostp do pl struktury
> profit
> lose interest
```

## 8.3 Dostęp do pól i metod struktur

Do pól i metod struktury odwołujemy się za pomocą operatora `'s` (apostrof + s).

**Przykład użycia struktury:**

```
> be struct_example
```

```
> # Definicja struktury Point (jak wyzej)
> look around
> Point
> see x_coord
> see y_coord
> lose interest

> # Utworzenie instancji struktury Point
> spot my_point is Point # "spot" dla typow zloonych
> # Przypisanie wartosci do pol
> my_point's x_coord is 10
> my_point's y_coord is 20

> spit "Punkt X:"
> spit my_point's x_coord
> spit "Punkt Y:"
> spit my_point's y_coord

> # Uzycie struktury Rectangle z metoda
> look around
> Rectangle
> see width
> see height
> be calculate_area
> seeing area_val
> area_val is width breeding like height times
> profit
> lose interest

> spot my_rect is Rectangle
> my_rect's width is 5
> my_rect's height is 8
> see rect_area is call my_rect's calculate_area
> spit "Pole prostokata:"
> spit rect_area

> profit
```

---

**Oczekiwany wynik programu:**

```
Punkt X:
10
Punkt Y:
20
Pole prostokata:
40
```

### Ćwiczenie

Zdefiniuj strukturę `Book` z polami `title` (typu `hear`), `author` (typu `hear`) i `pages` (typu `see`). Stwórz instancję tej struktury, przypisz wartości do jej pól, a następnie wyświetl

informacje o książce. Dodaj metodę `display_info` do struktury `Book`, która wyświetla wszystkie informacje o książce, i wywołaj tę metodę.

# 9 Zasięgi Zmiennych (Scopes)

Zrozumienie zasięgów jest kluczowe dla zarządzania widocznością zmiennych i funkcji.

## 9.1 Zasięg globalny

W zasięgu globalnym przechowywane są:

- **Definicje funkcji**: Funkcje zdefiniowane poza innymi funkcjami lub strukturami są globalne.
- **Definicje struktur**: Definicje struktur obowiązują w całym programie/pliku.

Zmienne globalne jako takie nie istnieją w GTL. Aby obejść to ograniczenie i stworzyć coś na kształt "zmiennej globalnej", można zdefiniować strukturę przechowującą potrzebne dane, a następnie utworzyć instancję tej struktury (choć dokumentacja sugeruje, że instancje są zwykle tworzone w zasięgach lokalnych, np. w funkcji `me`).

## 9.2 Zasięgi lokalne

Zasięgi lokalne tworzone są przez:

- Ciała funkcji
- Ciała definicji struktur (dla metod i ich zmiennych)
- Ciała pętli
- Ciała instrukcji warunkowych

Zmienne zadeklarowane w zasięgu lokalnym są widoczne tylko w tym zasięgu i jego podzakresach (zagnieżdżonych blokach).

### 9.2.1 Tworzenie pustego zasięgu

Można utworzyć pusty zasięg (np. do ograniczenia widoczności zmiennych) za pomocą zawsze prawdziwej instrukcji `if`:

```
> implying c: # c: to true
> # Kod w tym nowym, lokalnym zasiegu
> or sth
```

## 9.3 Redeklaracja zmiennych i dostęp do zmiennych przesłoniętych

Zmienna może być redeklarowana w podzakresie. W takim przypadku odwołanie do nazwy zmiennej będzie dotyczyć jej najbliższej (najgłębiej zagnieżdżonej) deklaracji. Aby odwołać się do zmiennej z zasięgu nadrzędnego, która została przesłonięta, używa się słowa kluczowego `parent`. Można użyć `parent parent` itd., aby cofać się o kolejne poziomy zasięgów.

**Przykład przesłaniania zmiennych i użycia `parent`:**

```
> be scope_example
> see flour is 1 # Zmienna w zasiegu funkcji scope_example
> spit flour     # Wyswietli 1

> implying c:    # Nowy zasieg lokalny
> see flour is 2 # Przeslania 'flour' z zasiegu nadrzednego
> spit flour     # Wyswietli 2
> spit parent flour # Odwoluje sie do 'flour' z scope_example, wyswietli 1

> implying c:    # Kolejny zagniezdzony zasieg
> see flour is 3 # Przeslania 'flour' z poprzedniego zasiegu
> spit flour     # Wyswietli 3
> spit parent flour # Odwoluje sie do 'flour' z wartoscia 2
> spit parent parent flour # Odwoluje sie do 'flour' z wartoscia 1
> or sth
> or sth
> profit
```

**Oczekiwany wynik programu:**

```
1
2
1
3
2
1
```

### Ćwiczenie

Stwórz funkcję, w której zadeklarujesz zmienną `x` o wartości 100. Wewnątrz tej funkcji utwórz pętlę `think that`, która wykona się 2 razy. Wewnątrz pętli zadeklaruj (redeklaruj) zmienną `x` z wartością będącą numerem iteracji (np. 1, potem 2). W każdej iteracji pętli wyświetl wartość lokalnej zmiennej `x` oraz wartość zmiennej `x` z zasięgu funkcji (używając `parent`).

# 10 Komentarze

Komentarze w GTL służą do dodawania wyjaśnień i notatek do kodu, które są ignorowane przez interpreter.

## 10.1 Komentarze pełnoliniowe

Każda linia, w której pierwszy nie-biały znak (poza obowiązkowym prefiksem >) jest inny niż >, jest traktowana jako komentarz. Dokumentacja precyzuje jednak: "Każda linia, w której pierwszy nie-biały znak jest inny niż >, jest traktowana jako komentarz."To zdanie wydaje się sprzeczne z główną zasadą GTL, że każda linia kodu zaczyna się od >. Poprawniejsza interpretacja, zgodna z duchem GTL i typowymi językami, byłaby taka, że specjalny znak lub sekwencja oznacza komentarz. Jednak, plik `comments.md` mówi też: "Komentarze są ignorowane na etapie parsowania, nie ingerując w żaden sposób w kod."Biorąc pod uwagę przykład z 'variables.md' (`> Deklaracja liczby calkowitej`), preferowanym sposobem komentowania jest komentarz w linii.

Jeśli interpretować dosłownie "Każda linia, w której pierwszy nie-biały znak jest inny niż >", to oznacza to linie, które *nie są* kodem GTL i są po prostu ignorowane przez parser szukający linii GTL. To bardziej dotyczy tego, co parser GTL robi z liniami niepasującymi do formatu GTL, niż sposobu na komentowanie *wewnątrz* kodu GTL.

## 10.2 Komentarze w linii (liniowe)

Komentarze w linii są oznaczane znakiem #. Wszystko od znaku # do końca linii jest ignorowane przez interpreter. To jest standardowy i zalecany sposób dodawania komentarzy.

**Przykład użycia komentarzy:**

```
> be comment_example
> # To jest komentarz pelnoliniowy (jesli zaczyna sie od # po >)
> see my_variable is 42 # To jest komentarz w linii.
> spit my_variable      # Wyswietla wartosc zmiennej

# Ta linia (bez poczatkowego > ) bylaby zignorowana przez parser
# jako nie bedaca kodem GTL, zgodnie z jedna interpretacja.
# Jednakze, bezpieczniej jest uzywac > # dla komentarzy pelnoliniowych.

> # Kolejny przyklad:
> hear message is "Hello" # Inicjalizacja wiadomosci
> # Nastepna linia wyswietli wiadomosc
> spit message
> profit
```

**Oczekiwany wynik programu:**

```
42
Hello
```

**Ćwiczenie**

Przejrzyj jeden z wcześniej napisanych programów (np. dotyczący funkcji lub struktur) i dodaj komentarze wyjaśniające działanie kluczowych fragmentów kodu. Użyj zarówno komentarzy pełnoliniowych (zaczynających się od `> #`), jak i komentarzy na końcu linii kodu.

# A Instalacja i Uruchamianie Programów GTL

Poniżej znajdują się informacje dotyczące instalacji interpretera GTL oraz uruchamiania programów.

## A.1 Wymagania systemowe

- **Java 23** (lub wersja określona w dokumentacji projektu, musi być dostępna w systemowej zmiennej `PATH`). Zalecane jest użycie najnowszej wersji LTS Javy.
- Dostęp do terminala (Windows, Linux, lub macOS).

## A.2 Kroki instalacji

1. **Zainstaluj Javę:** Sprawdź, czy Java (np. w wersji 23) jest zainstalowana:

   ```
   java -version
   ```

   Jeśli nie, pobierz Javę z Oracle Java Archive lub innego zaufanego źródła. Dodaj katalog `bin` Javy do systemowej zmiennej `PATH` oraz ustaw zmienną środowiskową `JAVA_HOME` wskazującą na katalog instalacji Javy.

2. **Pobierz aplikację GTL:** Przejdź do sekcji "Releases"w repozytorium projektu GTL. Pobierz najnowszą wersję. Rozpakuj archiwum do wybranego folderu. Interpreter GTL to plik wykonywalny (np. `GTL.exe`) znajdujący się w folderze `bin` rozpakowanego archiwum.

3. **(Opcjonalnie) Skojarz rozszerzenie .gtl (Tylko Windows):** Aby pliki `.gtl` otwierały się domyślnie za pomocą interpretera, w systemie Windows przejdź do Ustawienia > Aplikacje > Aplikacje domyślne. Wyszukaj rozszerzenie `.gtl`, kliknij "Wybierz domyślną"i wskaż plik `GTL.exe`.

## A.3 Uruchamianie programów GTL

Aby uruchomić program napisany w GTL:

1. Otwórz terminal lub wiersz poleceń.
2. Przejdź do folderu `bin`, w którym znajduje się interpreter GTL (np. `GTL.exe`).
3. Uruchom program za pomocą komendy:

   ```
   gtl sciezka/do/twojego/pliku.gtl
   ```

   Lub, jeśli interpreter (np. `GTL.exe`) jest w `PATH`:

   ```
   GTL.exe sciezka/do/twojego/pliku.gtl
   ```

### A.3.1 Opcje linii komend

- `-h`: Wyświetla pomoc.
- `-d`: Włącza tryb debugowania.
- `-v`: Wyświetla wersję GTL.

Przykład:

```
gtl -d sciezka/do/pliku.gtl
```

# B Debugger GTL

GTL dostarcza prosty debugger uruchamiany z opcją `-d`. Gdy debugger jest aktywny, dostępne są następujące komendy:

- **ENTER**: Przechodzi do następnego kroku (step forward).
- **liczba** (np. `5`): Przechodzi do przodu o podaną liczbę linii.
- **line liczba** (np. `line 10`): Równoważne wpisaniu samej liczby; przechodzi do określonej linii (może oznaczać wykonanie do tej linii lub skok, zależnie od implementacji).
- **resize** a następnie **ENTER**: Przelicza rozmiar wyświetlania (użyj, jeśli wyjście wygląda na źle wyrównane).
- **clear**: Czyści okno wyjścia programu.

Używanie debuggera może znacznie pomóc w znajdowaniu błędów i zrozumieniu przepływu wykonania programu GTL.

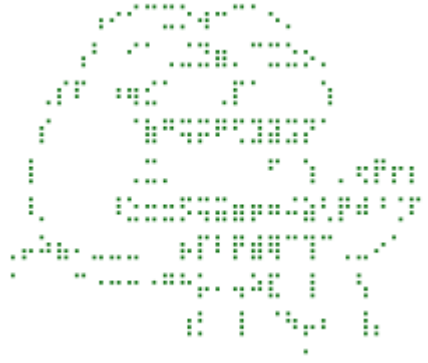
# C Błędy i Komunikaty o Błędach

Ze względu na memiczną konwencję języka, błędy w GTL mogą być prezentowane w dwóch trybach: **memicznym** i **technicznym**.

## C.1 Struktura komunikatu o błędzie

Komunikaty o błędach składają się z kilku części:

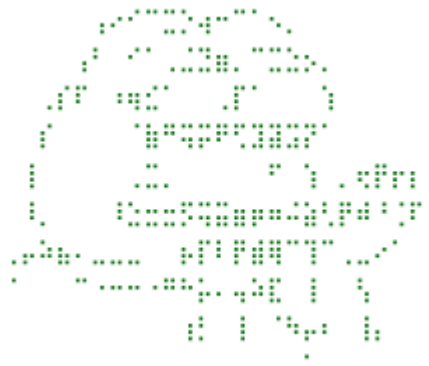
1. Grafika ASCII przedstawiająca smutną żabę Pepe:



2. Ścieżka do pliku, w którym wystąpił błąd.
3. Linia kodu, w której błąd się pojawił, z tokenem powodującym błąd podświetlonym za pomocą `^^^`.
4. Komunikat o błędzie w formie memicznej, np. `Womp Womp at [X,Y]: ....`
5. Komunikat o błędzie w formie technicznej, np. `Error at [X,Y]: ....`

Błędy mogą dotyczyć składni, semantyki lub czasu wykonania. Komunikaty często zawierają sugestie dotyczące naprawy, ale należy podchodzić do nich z ostrożnością.

## C.2 Przykład komunikatu o błędzie



```
(..)GTL/examples/tests/nested.gtl:72:20
 > implying yeet'th number doesn't vibe with 0
                      ^
Womp womp at [72, 20]: You are not long enough you have only 5 but you wanted 5
Error at [72, 20]: Index 5 is out of bounds for an array of length 5
```