

Dokumentacja końcowa TKOM
Temat projektu: „Język z czasem”

SPIS TREŚCI

1	OPIS TEMATU PROJEKTU	2
1.1.	OPIS „JĘZYKA Z CZASEM”	2
1.2.	ELEMENTY JĘZYKA	2
1.2.1.	TYPY W JĘZYKU	2
1.2.2.	DEFINIOWANIE ZMIENNYCH	2
1.2.3.	WYRAŻENIA ARYTMETYCZNE	3
1.2.4.	WYRAŻENIA LOGICZNE	3
1.2.5.	INSTRUKCJA WARUNKOWA	4
1.2.6.	INSTRUKCJA PĘTLI.....	4
1.2.7.	DEFINIOWANIE I WYWOŁYWANIE FUNKCJI.....	4
1.2.8.	FUNKCJE WBUDOWANE	5
1.3.	PRZYKŁADOWY PROGRAM	5
2	GRAMATYKA JĘZYKA.....	6
2.1.	SKŁADNIA	6
2.2.	LEKSYKA.....	7
3	TECHNICZNE ASPEKTY REALIZACJI PROJEKTU	8
3.1.	PRODUKT KOŃCOWY.....	8
3.2.	LEKSER.....	8
3.3.	PARSER	9
3.4.	INTERPRETER	10
3.5.	MECHANIZM OBSŁUGI BŁĘDÓW	10
3.6.	WYKORZYSTANE NARZĘDZIA.....	10

1. OPIS TEMATU PROJEKTU

1.1. OPIS „JĘZYKA Z CZASEM”

Główną, charakterystyczną funkcjonalnością oferowaną przez tworzony język jest **przetwarzanie zmiennych i wartości związanych z czasem – momentami w czasie** (np.: data) czy **okresami w czasie** (np.: sekunda). Język obsługuje również liczby całkowite i rzeczywiste oraz łańcuchy znaków (tylko w funkcji do wyświetlania tekstu na ekranie). Przetwarzanie danych możliwe jest dzięki zdefiniowanym operatorom, instrukcji warunkowej i pętli oraz mechanizmom tworzenia zmiennych czy własnych funkcji. Język oferuje również kilka funkcji wbudowanych – np.: do wypisywania tekstu na ekran czy podania aktualnej daty. Wszelkie białe znaki w kodzie są ignorowane przez język, z wyjątkiem tych w łańcuchu znaków.

1.2. ELEMENTY JĘZYKA

1.2.1. TYPY W JĘZYKU

Implementowany język oferuje obsługę wartości o następujących typach:

- Typ liczbowy (bez jednostki): `20`, `3.14`, `(-8)`, ...;
- **Typ czasowy – moment w czasie:**
 - godzina – np.: `[^17:42:36]`
 - data – np.: `[08/11/2021]`
 - timestamp – np.: `[08/11/2021 17:42:36]`
- **Typ czasowy – okres w czasie:**
 - sekunda – `s`, np.: `[s]2`, `[s]8.4`, `-[s]16`, ...
 - minuta – `m`, np.: `[m]2`, `[m]8.4`, `-[m]16`, ...
 - godzina – `h`, np.: `[h]2`, `[h]8.4`, `-[h]16`, ...
 - stała czasowa, zapisywana w `s`; np.: `[+2:30:54]`, co jest równoważne `[s]9054`
- Typ łańcucha znaków (używany jedynie w funkcji wyświetlającej tekst na ekranie): `„Ala ma kota”`, `„Witaj”`, ...;

Jak widać z powyższych przykładów, wartości niektórych kategorii wymagają podania wprost swojego typu w nawiasach klamrowych. Inne należy wpisywać bezpośrednio w nawiasy klamrowe, czasami dołączając odpowiedni prefiks.

1.2.2. DEFINIOWANIE ZMIENNYCH

„Język z czasem” pozwala tworzyć zmienne, przypisywać im wartość oraz je odczytywać. Język wykorzystuje **typowanie dynamiczne**. Nie trzeba z góry określać typ zmiennej przy jej tworzeniu. Zdefiniowanie zmiennej następuje w momencie

przypisania do niej jakiejś wartości – służy do tego **operator =**. Do zmiennych należy odwoływać się poprzez ich identyfikatory. Przykłady:

```
zmienna_liczbowa = 7
zmienna_czasowa_okres_1 = [s]14.8
zmienna_czasowa_okres_2 = [h]5
zmienna_czasowa_moment_1 = [15/04/1990]
zmienna_czasowa_moment_2 = [^11:12:43]
```

1.2.3. WYRAŻENIA ARYTMETYCZNE

Tworzony język umożliwia konstruowanie wyrażeń arytmetycznych przy użyciu określonych operatorów: *****, **/**, **+**, **-** oraz nawiasów **()**. Niektóre operatory można użyć jedynie z określonymi kombinacjami typów operandów ze względu na sensowność operacji. Poniższa tabelka pokazuje dozwolone operacje dla danych typów operandów:

Operand 2 Operand 1	Typ liczbowy	Typ czasowy - okres	Typ czasowy - moment
Typ liczbowy	* , / , + , -	*	BRAK
Typ czasowy - okres	* , /	+ , -	BRAK
Typ czasowy - moment	BRAK	BRAK	-

Operand operatora unarnego – musi mieć typ liczbowy lub czasowy – okres.

Przy obliczaniu wyrażeń algebraicznych, typ wyniku będzie wyznaczany na podstawie typów operandów. W przypadku typów czasowych – moment w czasie różnica będzie wyrażona w sekundach. W przypadku okresów w czasie typ wyniku będzie taki sam, jak bardziej podstawowy spośród typów operandów tzn. dla operacji **[h] + [s]** wynik będzie w **[s]**; dla operacji **[d] - [m]** wynik będzie w **[m]** itp. W celu konwersji wartości do innego typu, należy użyć odpowiedniej funkcji wbudowanej.

Przykłady wyrażeń arytmetycznych:

```
wyr1 = 12*(6+2.4)
wyr2 = [s]14/2
wyr3 = ([11/11/2021]-[10/11/2021])/wyr1
```

1.2.4. WYRAŻENIA LOGICZNE

Język umożliwia również konstruowanie wyrażeń logicznych przy użyciu określonych operatorów: **==**, **=\=**, **<**, **>**, **<=**, **>=**, **&** (**i**), **|** (**lub**) oraz nawiasów **()**. Operatory logiczne można używać z operandami, które mają ten sam typ lub który można sprowadzić do tego samego typu, np.: operacje pomiędzy okresami w czasie są poprawne, ale między momentem a okresem już nie. Przykłady wyrażeń logicznych zostaną pokazane w części *Instrukcja warunkowa i Definiowanie Funkcji*.

1.2.5. INSTRUKCJA WARUNKOWA

W języku istnieje instrukcja warunkowa, która może wystąpić w jednej z kilku postaci:

- `if(wyrażenie_logiczne) {wykonaj jeśli wyrażenie true}`
- `if(wyrażenie_logiczne) {wykonaj jeśli wyrażenie true}
else {wykonaj w przeciwnym przypadku}`
- `if(wyrażenie_logiczne1){wykonaj jeśli wyrażenie1 true}
elif(wyrażenie_logiczne2) {wykonaj jeśli wyrażenie2 true}`
- `if(wyrażenie_logiczne1){wykonaj jeśli wyrażenie1 true}
elif(wyrażenie_logiczne2) {wykonaj jeśli wyrażenie2 true}
else {wykonaj w przeciwnym przypadku}`

Instrukcja `if` może mieć jeden lub więcej klauzul `elif` i co najwyżej jedną klauzulę `else`. Przykład:

```
if(zmienna >= [m]6)
{
    zmienna = zmienna/10
}
else
{
    zmienna = zmienna-[m]5
}
```

1.2.6. INSTRUKCJA PĘTLI

Język udostępnia instrukcję pętli, która umożliwia wielokrotne wykonanie fragmentu kodu tak długo jak spełniony jest określony warunek.

```
while(warunek_czyli_wyrażenie_logiczne)
{
    powtarzany kod
}
```

Przykład:

```
i = 4
while(zmienna >= [m]5 & i!=0)
{
    zmienna = zmienna/5
    i = i-1
}
```

1.2.7. DEFINIOWANIE I WYWOŁYWANIE FUNKCJI

„Język z czasem” umożliwia tworzenie własnych funkcji. Deklaracja i definicja funkcji odbywają się w tym samym momencie – nie ma możliwości ich

rozdzielenia. W nawiasach `()` można podać nazwy argumentów funkcji. Argumenty są przekazywane do funkcji **poprzez wartość**. W ciele funkcji można użyć **RET()** do ustalenia wartości zwracanej. Zmienne stworzone w ciele funkcji są widoczne tylko w jej obrębie. Sama definicja powinna się rozpoczynać od **FUNC**.

```
FUNC nazwa_funkcji(arg1, arg2, ..., argN) { ciało funkcji }
```

Przykład:

```
FUNC podzielPrzez5(zmienna)
{
    RET(zmienna/5)
}
```

Aby wywołać funkcję, należy użyć jej nazwy i podać argumenty. Przykład:

```
inna_zmienna = podzielPrzez5(zmienna)
```

1.2.8. FUNKCJE WBUDOWANE

Język oferuje kilka funkcji wbudowanych, które można wywołać tak jak każdą inną funkcję:

- `SHOW(arg1, arg2, ..., argN)` - wypisuje tekst na ekran. Argumentem funkcji może być łańcuch znaków, zmienna lub wyrażenie arytmetyczne.
- `getTime()` - zwraca aktualny czas,
- `getDate()` - zwraca aktualną datę,
- funkcje do konwersji typów – zostaną one wyspecyfikowane w trakcie trwania projektu w tej sekcji dokumentacji.

1.3. PRZYKŁADOWY PROGRAM

```
FUNC zwrocIleSekundTemu(data)
{
    dzis = getDate()
    RET(dzis - data)
}

dataUrodzin = [01/01/2000]
ilosc_sekund = zwrocIleSekundTemu(dataUrodzin)
SHOW(„Ilość przeżytych sekund to ”, ilosc_sekund)

i = 1
iloscGodzin = 0
while(i == 1)
{
    if(ilosc_sekund >= [s]3600)
    {
        ilosc_sekund = ilosc_sekund - [s]3600
        iloscGodzin = iloscGodzin + 1
    }
    else
```

```

        {
            i = 0
        }
    }

```

```
SHOW(„Ilosc przezytych godzin to ”, iloscGodzin)
```

2. GRAMATYKA JĘZYKA

2.1. SKŁADNIA

```

program          = polecenie, {polecenie};

polecenie        = definicja funkcji
                  | instrukcja;

definicja funkcji = 'FUNC', identyfikator, '('\,
                  [identyfikator, {'\,', identyfikator}],
                  '\)', blok instrukcji;

blok instrukcji  = '{\, {instrukcja}, '\)';

instrukcja       = (przypisanie
                  | instrukcja warunk
                  | pętla
                  | wywołanie funkcji
                  | wyświetlenie tekstu
                  | zwrócenie wartości);

przypisanie      = identyfikator, '=', (wyrażenie arytm
                  | wywołanie funkcji);

instrukcja warunk = 'if', warunek, blok instrukcji
                  {'elif', warunek, blok instrukcji},
                  [ 'else', blok instrukcji];

pętla            = 'while', warunek, blok instrukcji;

warunek          = '('\, wrażenie logiczne, '\)';

wywołanie funkcji = '\.', identyfikator, '('\, [wyrażenie arytm,
                  {'\,', wyrażenie arytm }], '\)';

wyświetlenie tekstu = 'SHOW', '('\, (string
                  | wyrażenie arytm ), {'\,', (string
                  | wyrażenie arytm ) }, '\)';

zwrócenie wartości = 'RET', '('\, wyrażenie arytm , '\)';

wyrażenie logiczne = podwyrażenie logic, {operator logiczny,
                  wyrażenie logiczne };

podwyrażenie logic = wyrażenie arytm, operator porównania,
                  wyrażenie arytm;

wyrażenie arytm   = składnik, { operator addytywny,

```

```

wyrażenie arytym };

składnik          = czynnik, { operator multiplikatywny,
                              składnik};

czynn timer       = ['-'], (identyfikator
                           | stała
                           | '(', wyrażenie arytym, ') '
                           | wywołanie funkcji);

```

2.2. LEKSYKA

```

identyfikator      = [a-zA-Z][a-zA-Z0-9_]*

operator logiczny  = \ | | &

operator porównania = == | =\= | < | > | <= | >=

operator addytywny = \+ | -

operator multiplikatywny = \* | /

string             = „[a-zA-Z0-9[ \t]]*„

stała              = moment | (jednostka? liczba)

liczba               = liczba całkowita (\.[0-9]*)?

liczba całkowita     = 0|([1-9][0-9]*)

jednostka            = \[(s | m | h)\]

moment               = data | godzina | timestamp
                    | timeperiod

data                 = \[liczba całkowita/liczba
                    całkowita/liczba całkowita\]

godzina              = \[^liczba całkowita:liczba
                    całkowita:liczba całkowita\]

timestamp            = \[liczba całkowita/liczba
                    całkowita/liczba całkowita[\t]
                    liczba całkowita:liczba
                    całkowita:liczba całkowita\]

timeperiod           = \[\\+liczba całkowita:liczba
                    całkowita:liczba całkowita\]

```


3. TECHNICZNE ASPEKTY REALIZACJI PROJEKTU

3.1. PRODUKT KOŃCOWY

Celem projektu jest zaimplementowanie interpretera dla opisanego wcześniej języka. Interpreter będzie pracował w trybie wsadowym. Danymi wejściowymi będzie plik `.txt` z kodem źródłowym „języka z czasem”. Plik ten należy przekazać do interpretera przy wywołaniu:

```
./tln <nawa_pliku.txt
```

Tekst z pliku będzie zaciągany przez specjalny moduł dotyczący źródła danych. Z modułem tym będzie komunikował się lekser, który będzie przekazywał ciąg rozpoznanych tokenów parserowi. Parser z kolei będzie budował drzewa składniowe, na podstawie których będzie wykonywany kod przez moduł interpretacyjny. Translacja kodu źródłowego będzie więc przebiegać w ramach współpracy poszczególnych modułów interpretera.

3.2. LEKSER

Klasa *Lexer* ściśle współpracuje z klasą *CodeSource*, która dostarcza strukturę *CharAndPosition*, zawierającą pojedynczy, przeczytany ze źródła znak wraz z jego pozycją we źródle – numerem linii i kolumny. *Lexer* buduje z kolejnych dostarczanych znaków ciąg tak długi, aż rozpozna w nim któryś z poniższych tokenów:

Kategoria	Token	Wartość
Słowa kluczowe	<code>T_FUNC</code>	FUNC
	<code>T_IF</code>	if
	<code>T_ELIF</code>	elif
	<code>T_ELSE</code>	else
	<code>T_WHILE</code>	while
	<code>T_RET</code>	RET
	<code>T_SHOW</code>	SHOW
Literały	<code>T_IDENTIFIER</code>	Np.: abc
	<code>T_INT</code>	Np.: 12
	<code>T_DOUBLE</code>	Np.: 3.14
	<code>T_SEC_U</code>	[s]
	<code>T_MIN_U</code>	[m]
	<code>T_HOUR_U</code>	[h]
	<code>T_STRING</code>	Np.: "Witaj"
	<code>T_DATE</code>	Np.: [17/01/2022]
	<code>T_TIMESTAMP</code>	Np.: [17/01/2022 12:00:00]
	<code>T_CLOCK</code>	Np.: [^12:00:00]
	<code>T_TIME_PERIOD</code>	Np.: [+3:55:12]
Operatory	<code>T_ASSIGN</code>	=
	<code>T_PLUS</code>	+

	<i>T_MINUS</i>	-
	<i>T_MULTIPLY</i>	*
	<i>T_DIVIDE</i>	/
	<i>T_OR</i>	
	<i>T_AND</i>	&
	<i>T_EQUAL</i>	==
	<i>T_NOT_EQUAL</i>	=\=
	<i>T_GREATER</i>	>
	<i>T_GREATER_E</i>	>=
	<i>T_LESS</i>	<
	<i>T_LESS_E</i>	<=
Metaznaki	<i>T_DOT</i>	.
	<i>T_COMMA</i>	,
	<i>T_PARENTHESSES_1</i>	(
	<i>T_PARENTHESSES_2</i>)
	<i>T_BRACE_1</i>	{
	<i>T_BRACE_2</i>	}
Specjalne	<i>T_END</i>	EOF
	<i>T_NUMBER_TOO_LARGE</i>	tekst "TOO LARGE"
	<i>T_UNKNOWN</i>	jakiś tekst

Lexer zwraca na raz zawsze pojedynczy token, jedynie na żądanie parsera (poprzez wywołanie metody leksera *getNextToken()*).

3.3. PARSER

Parser pobiera od leksera kolejne tokeny i układa je w większe struktury, które następnie są dołączane do budowanego drzewa składniowego *ProgramTree*. Węzły tego drzewa odpowiadają głównie wysokopoziomowym konstrukcjom zaimplementowanym w tworzonym języku: pętli *while* (*WhileLoop*), instrukcji *if* (*IfStatement*, *ElifStat*), operacjom binarnym i unarnej (*OperatorOperation*), definicji funkcji (*FuncDef*), wywołaniu funkcji (*FuncCall*), specjalnej funkcji *SHOW* (*ShowFunc*) czy instrukcji *RET* (*ReturnInstr*). Istotnym węzłem jest również *Literal*, który przechowuje pojedynczą wartość (w obiekcie klasy *Value*). Każdy z węzłów zawiera odpowiednie pola dla danej konstrukcji np.: *WhileLoop* zawiera wskazanie na *Expression* będące warunkiem pętli i listę wskazań na instrukcje znajdujące się w bloku pętli.

Pierwotnie wszystkie węzły drzewa dziedziczyły jedynie po jednej klasie bazowej, a funkcje parsujące przyjmowały jako argument goły wskaźnik na rodzica. Obecnie istnieją dwie hierarchie dziedziczenia: po klasie *Phrase* i po klasie *Expression*. Różnią się one między sobą implementowaną metodą czysto wirtualną, wykorzystywaną w fazie interpretacji. Podczas interpretacji obiekty dziedziczące po klasie *Phrase* nic nie zwracają, natomiast te dziedziczące po klasie *Expression* zwracają klasę *Value*. Zmiany objęły również funkcje parsujące – ostatecznie nic nie przyjmują jako argument, ale zwracają gotowe węzły opakowane w *unique* pointery.

3.4. INTERPRETER

Klasa *Interpreter* przechodzi po kolei po wszystkich węzłach *ProgramTree*, wywołując odpowiednio metodę wirtualną *execute* dla obiektów dziedziczących po *Phrase*, albo *evaluate* dla obiektów dziedziczących po *Expression*. Metody te zawierają odpowiedni kod służący do interpretowania konkretnej klasy.

W celu zrealizowania zakresu widoczności zmiennych i wywołań funkcji, niektóre z wywoływanych w toku interpretacji metod zapisują lub też odczytują zmienne bądź definicje funkcji z klasy *Context*.

Interpreter pracuje w trybie wsadowym. Można go wywołać z opcją służącą do wypisywania tekstowej reprezentacji drzewa programu.

3.5. MECHANIZM OBSŁUGI BŁĘDÓW

W procesie interpretacji kodu mogą się pojawić różne błędy:

- analiza leksykalna – np.: nieznane ciągi znaków,
- analiza składniowa – np.: instrukcja *while* bez warunku,
- analiza semantyczna – np.: próba przemnożenia dwóch okresów czasowych.

W przypadku wykrycia jakiegokolwiek błędu zgłaszany jest wyjątek, w wyniku przechwycenia którego wypisywany jest na ekran odpowiedni komunikat o błędzie. Po napotkaniu błędu proces interpretacji jest natychmiast przerywany.

3.6. WYKORZYSTANE NARZĘDZIA

Narzędzia wykorzystane w trakcie realizacji projektu:

- Język C++, standard 17,
- CMake,
- Do testów jednostkowych – biblioteka *Doctest* (<https://github.com/doctest/doctest>)
- Git i Gitlab.