Kompilator języka z funkcjami anonimowymi

1. **Sposób realizacja zadania**:

Program kompilujący/interpretujący mój język z funkcjami lambda będzie napisany w języku C++. Przyjmował będzie on dane wejściowe w postaci plików lub jako dane przesyłane strumieniem. Oprócz wspomnianej funkcji anonimowej język wspiera:

- inicjowanie zmiennych ✓
- przypisywanie wartości zmiennym ✓
- operacje matematyczne z zapewnionymi odpowiednimi priorytetami ✓
- wyrażenia logiczne ✓
- wypisywanie danych na konsole ✓
- dwie instrukcje pętli ✓
- jedną instrukcję warunkową ✓
- definiowanie funkcji oraz ich rekurencyjne wykonywanie ✓
- przekazywanie zmiennych do funkcji przez referencje jak i wartość ✓
- operacje na stringach ✓
- obsługę komentarzy ✓

2. **Przyjęte założenia**:

Przy wykonywaniu projektu przyjąłem pewne założenia, mianowicie:

(a) Liczba w programie może zaczynać sie od „0”, wtedy po prostu wszystkie je pomijamy.
(b) liczba rzeczywista może wygladać w ten sposób: „3.”
(c) Zmienne są typowane statycznie
(d) Zmienne są typowane silnie
(e) Zmienne są mutowalne
(f) Wchodzimy w nowy zakres widoczności zmiennych gdy: rozpoczynamy program, wywołujemy funkcję, wchodzimy w blok pętli, blok warunku lub blok lambdy. Oczywiście bloki widzą wcześniej zadeklarowane zmienne.
(g) Komentarze obowiązują od miejsca ich ustawienia do kolejnego znaku nowej linii.
(h) Zmienne są przekazywane do funkcji za pomocą wartości lub referencji.
(i) Wyrażenie lambda traktuję jako zmienną, to znaczy mogę je przypisywać do identyfikatora czy też podawać w parametrach i argumentach funkcji.

3. **Przykładowy kod języka**:

```
//definicja funkcji
DEF fun1(string ref tak)
        tak = "nie";
END

DEF fun2(function L)
        L(3);
END


//głowna funkcja programu
START
// przyklady przypisania i wypisania
        int x = 2;
        int y = 5;
        print(x); //2
        print ("\n");

// operacje matematyczne
        x = x + 10;
        print (x); //12
        print ("\n");

        x = x+y*2-y;
        print (x); //17
        print ("\n");

// wywoływanie funkcji i przekazywanie argumentów
        string tekst = "tak";
        fun1(ref tekst);
        print (tekst) //nie
        print ("\n");

// użycie funkcji anonimowej
        function L = lambda[x](int a)
                a = a + x;
                print (a);
        END(3);  //
        L(1);       //18
        fun2(L);  //20

// użycie przykładowej pętli
        int a = 0;
        FOR i IN RANGE 5
                a = a+i;
        END
        print (a) //15

//uzycie przykladowego warunku
        IF (a < x)
                print ("pierwszy");
        ELSE
                print ("drugi");
        END

STOP
//koniec działania programu
```

4. **Gramatyka**:

Od momentu złożenia dokumentacji wstępnej, gramatyka uległa dość dużym zmianom. Zdecydowana większość z nich wynikała z prób uzyskania jak najprostszej implementacji Parsera. Z tego chociaż powodu doszły takie wyrażenia jak „przypisanie_wywołanie_wyrażenie", które miały na celu zapobiegnięciu cofania się podczas parsowania. Wprowadzenie takich dodatkowych wyrażeń wynikało z tego, że poprzedni układ zaczynał się od tego samego elementu np. identyfikator był klasą która rozpoczynała i „przypisanie do zmiennej" i „wywołanie funkcji".

Inne zmiany w gramatyce dotyczyły nowych funkcjonalności. Dla przykładu, rozszerzenie implementacji petli for o mozliwosc umieszczenia w jej zasięgu dowolnego wyrażenia złożonego.

Jeszcze inne zmiany wynikały z wcześniej popełnionych błędów, np. brak możliwości powrotu z funkcji bez zwracania wartości.

Opisana poniżej gramatyka jest notacją EBNF. Wytłuszczone słowa oznaczają symbole terminalne. Symbolem głownym jest „Start".

```
Start                    = {definicja_funkcji | definicja_zmiennej},
                             "START", {działanie}, "STOP";

definicja_funkcji        = "DEF", identyfikator, "(", [lista_parametrów], ")",
                             {działanie}, "END";

działanie                = (definicja_zmiennej

                         | powrót_z_funkcji

                         | przypisanie_wywołanie_wyrażenie

                         | wyrażenie_złożone), ";"

                         | (blok_petli_for

                         | blok_petli_while

                         | blok_if);

przypisanie_wywołanie_wyrażenie = identyfikator, przypisanie_do_zmiennej |
                                  wywołanie_funcji | reszta_wyrażenia_złożonego;

definicja_zmiennej       = typ_zmiennej, identyfikator, [przypisanie do zmiennej];

przypisanie_do_zmiennej = "=", wyrażenie_złożone;

powrót_z_funkcji         = "return", [wyrażenie_złożone];

wywołanie_funkcji        = "(", [lista_argumentów], ")";

blok_petli_for           = "FOR", identyfikator, "IN RANGE", wyrażenie_złożone,
                         {działanie}, "END";

blok_petli_while         = "WHILE", "(", wyrażenie_złożone, ")", {działanie}, "END";
blok_if                  = "IF", "(", wyrażenie_złożone, ")", {działanie},
                         ("END" | "ELSE", {działanie}, "END");


(***WYRAŻENIA***)
reszta_wyrażenia_złożonego = [operator_porównania, wyrażenie]

wyrażenie_złożone        = [negacja], wyrażenie, [operator_porównania, wyrażenie];
```

```
wyrażenie               = [znak_liczby], składnik_dodawania, {operator_dodawania,
                           składnik_dodawania};

składnik_dodawania      = składnik_mnożenia, {operator_mnożenia, składnik_mnożenia};


składnik_mnożenia       = element | (zmienna, [wywołanie_funkcji])  |
                          "(", wyrażenie_złożone, ")";
```

```
(***TYPY***)

element                       = string | liczba | liczba_rzeczywista | lambda;
typ_zmiennej                  = "int" | "float" | "string" | "function";

referencja                    = "ref", zmienna;

lambda                        = "lambda", "[", [domknięcie], "]",
                                "(", [lista_parametrów], ")", {działanie}, "END",
                                ["(", [lista_argumentów], ")"];

lista_parametrów              = typ_zmiennej, (referencja | identyfikator),
                                {",", typ_zmiennej, (referencja | identyfikator)};

lista_argumentów              = (wyrażenie_złożone), {",", (wyrażenie_złożone)};

domknięcie                    = (referencja | zmienna), {",", (referencja | zmienna)};

zmienna                       = identyfikator;
```

```
(***PROSTE JEDNOSTKI***)

identyfikator           = litera, {litera | cyfra};

string                  = """, {znak_string | znak_specjalny_string}, """;
liczba_rzeczywista      = liczba, ".", {cyfra}
liczba                  = [znak_liczby], cyfra, {cyfra};
liczba_bez_znaku        = cyfra, {cyfra};
operator_porównania     = "==" | "!=" | "<" | "<=" | ">" | ">=";
znak_liczby             = "+" | "-";
negacja                 = "!";
operator_dodawania      = znak_liczby;
operator_mnożenia       = "*" | "/";
litera                  = ?A-Z?, | ?a-z?, | "_"
cyfra                   = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
znak_specjalny_string   = "\n" | "\"" | "\\" | "\t";
white                   = biały_znak, {biały_znak};

znak_string             = ? wszystkie znaki oprócz " oraz \ ?;
dowolny_znak            = ? wszystkie znaki ?;
biały_znak              = ? dowolny biały znak ?;
```

5. **Obsługa błędów**:

W przypadku zaistnienia błędu, użytkownik jest informowany odpowiednim typem zgłoszenia oraz podaną ~~linijką kodu~~ pozycją w kodzie w której doszło do błędu. Do obsługiwanych błędów należą

- Błędy leksykalne - np. Niezamknięty string
- Błędy składniowe – np. Brak słowa kluczowego, niezamknięcie nawiasu …
- Błędy semantyczne – np. niepoprawne przypisanie wartości do zmiennej

6. **Testowanie kodu**:
   Oprócz testów manualnych (przygotowane wcześniej pliki programów do kompilacji) w celu sprawdzenia poprawności kodu posłużę się testami jednostkowymi dostarczanymi przez bibliotekę Boost.test. Obecnie testy liczą ponad 1000 linijek, a pokrycie kodu nimi szacuję na 95%>>90%