|  |
| --- |
|  |

Tytuł

|  |
| --- |
|  |

Spis treści

[2 Kompilator 2](#_Toc89974275)

[2.1 Język programowania 2](#_Toc89974276)

[2.2 Zastosowanie kompilatora 2](#_Toc89974277)

[2.3 Budowa kompilatora 3](#_Toc89974278)

[2.4 Struktura kompilacji 4](#_Toc89974279)

[3 Lekser 8](#_Toc89974280)

[4 Interfejs użytkownika 8](#_Toc89974281)

[4.1 Instalacja Leksera 8](#_Toc89974282)

[4.2 Dodatki 11](#_Toc89974283)

[4.2.1 Logger 11](#_Toc89974284)

[5 Implementacja leksera 12](#_Toc89974285)

[6 Testy oraz działanie 12](#_Toc89974286)

[7 Bibliografia 13](#_Toc89974287)

# Kompilator

## Język programowania

**Język programowania** – zbiór zasad określających, kiedy ciąg symboli tworzy [program komputerowy](https://pl.wikipedia.org/wiki/Program_komputerowy) oraz jakie obliczenia opisuje. (1)

Języki dzielimy na wysoko i niskopoziomowe. Języki wysokopoziomowe starają się maksymalnie ułatwić rozumienie kodu programu przez człowieka, natomiast składnia języka niskopoziomowego odpowiada instrukcją procesora.

## Zastosowanie kompilatora

Istnieje wiele języków programowania i każdy z nich potrzebuje narzędzia, które przetłumaczy je na polecenia zrozumiałe dla komputera.   
Jednym z takich narzędzi jest kompilator.  
„Kompilator jest programem, który potrafi przeczytać program sformułowany w jednym języku – języku źródłowym – i przełożyć go na równoważny program w innym języku – języku wynikowym. Ważną rolą kompilatora jest zgłaszanie wykrytych w czasie tłumaczenia dowolnych błędów w programie źródłowym.” (2)



Rysunek . Działanie kompilatora

„Program wynikowy jest programem wykonywalnym w języku maszynowym, może zostać uruchomiony przez użytkownika w celu przetwarzania wejścia i wygenerowania wyjścia.” (2)



Rysunek . Działanie programu wynikowego

Innym popularnym narzędziem jest interpreter.

Zamiast tworzenia programu wynikowego jako efektu tłumaczenia na bieżąco wykonuje polecenia przetłumaczone z kodu źródłowego.

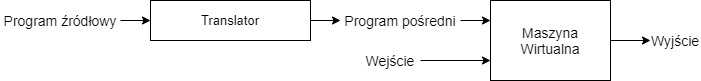


Rysunek . Działanie interpretera

Program tworzony przez kompilator jest znacznie szybszy. Z drugiej strony interpretacja może zająć mniej czasu niż kompilacja i uruchomienie oraz udostępnia lepszą diagnostykę błędów. Dlatego interpretacja jest często wykorzystywana w językach skryptowych.

Kompilator tłumaczący język wysokiego poziomu na inny język wysokiego poziomu jest nazywany translatorem source-to-source lub transkompilatorem.

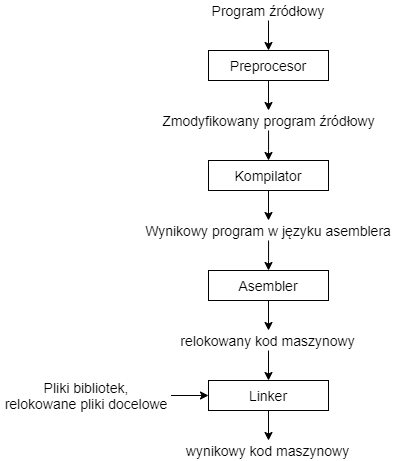
Coraz częściej wykorzystywana jest metoda hybrydowa nazywana kompilatorami just-in-tim (JIT, kompilacja na żądanie), gdzie kod źródłowy jest kompilowany do kodu pośredniego zwanym kodem bajtowym (bytecode), który jest interpretowany przez maszynę wirtualną.



Rysunek . kompilator JIT

## Budowa kompilatora

Do utworzenia programu wykonywalnego może być potrzebne kilka programów.



Rysunek . Proces tworzenia programu wykonywalnego

Preprocesor łączy kod źródłowy z kilku plików oraz rozwija skróty nazwane makrami do pełnych wyrażeń języka źródłowego.

Kompilator często jako wynik swojej pracy dostarcza program w języku asembler, gdyż jest on łatwiejszy do wykonania i debugowania.

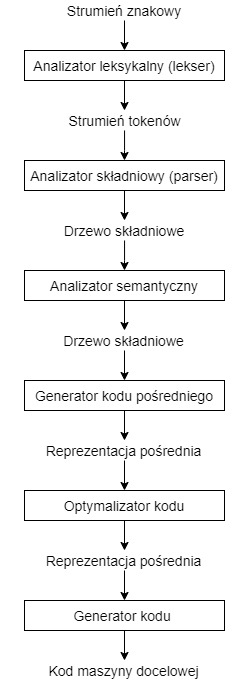
„Asembler generuje relokowany kod maszynowy jako swoje wyjście” (2)

Linker zwany również konsolidatorem łączy nasz kod maszynowy z kodem maszynowym bibliotek wymaganych do działania programu

## Struktura kompilacji

Kompilacja odbywa się w 2 częściach: analizy (front-end) i syntezy (back-end)

Część analityczna dzieli kod źródłowy na części składowe, stosując strukturę gramatyczną. W tej części szukane są błędy składniowe lub niejednoznaczności semantyczne. Na tym etapie zbierane są również informacje o kodzie źródłowym. Część syntezy na podstawie danych zebranych w fazie analizy tworzy program wynikowy. W każdej z części możemy wyszczególnić kilka faz, w których przekształcana jest jedna reprezentacja kodu źródłowego w kolejną.



Rysunek . Fazy kompilacji

„W praktyce wiele faz może być grupowanych razem i reprezentacje pośrednie między zgrupowanymi fazami nie muszą być jawnie konstruowane” (2)

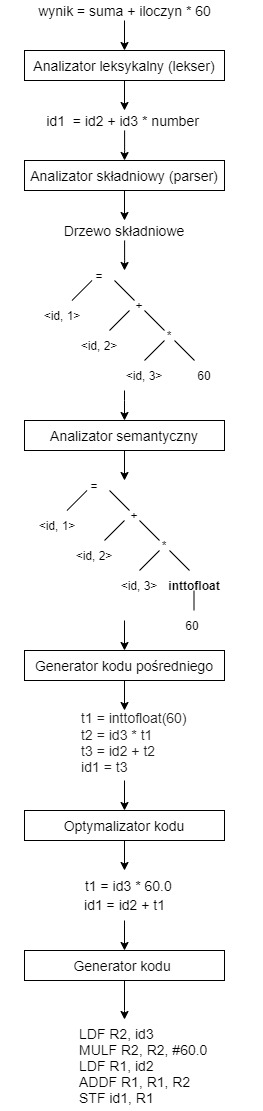
**Analizator leksykalny (lekser) -**  „odczytuje strumień znaków budujących program źródłowy i grupuje te znaki w znaczące sekwencje nazywane leksemami. Dla każdego leksemu analizator leksykalny tworzy wyjście w postaci tokenu.” (2)

**Analizator składniowy (parser) -** „używa tokenów utworzonych przez analizator leksykalny do zbudowania pośredniej reprezentacji przypominającej drzewo, odwzorowującej gramatyczną strukturę strumienia tokenów.” (2)

**Analiza semantyczna -** „sprawdza kod źródłowy pod kątem spójności semantycznej programu z definicją języka. Ponadto gromadzi on informacje o typach i zapisuje je.” (2)

**Generowanie kodu pośredniego -**  „generuje jawną niskopoziomową reprezentację pośrednią zbliżoną do kodu maszynowego”. (2) Kod ten powinien być łatwy do utworzenia i przetłumaczenia na kod maszynowy.

**Optymalizacja kodu** ***-*** „Faza niezależnej od architektury maszynowej optymalizacji kodu ma na celu ulepszenie kodu pośredniego, dzięki czemu lepszy będzie również kod wynikowy.” (2)  
**Generowanie kodu** ***-*** instrukcje pośrednie są tłumaczone na sekwencje instrukcji maszynowych wykonujących to samo zadanie. Krytycznym aspektem generowania kodu jest rozważne przypisanie rejestrów do przechowywanych zmiennych.



Rysunek . Translacja wyrażenia wynik = suma + iloczyn \* 60

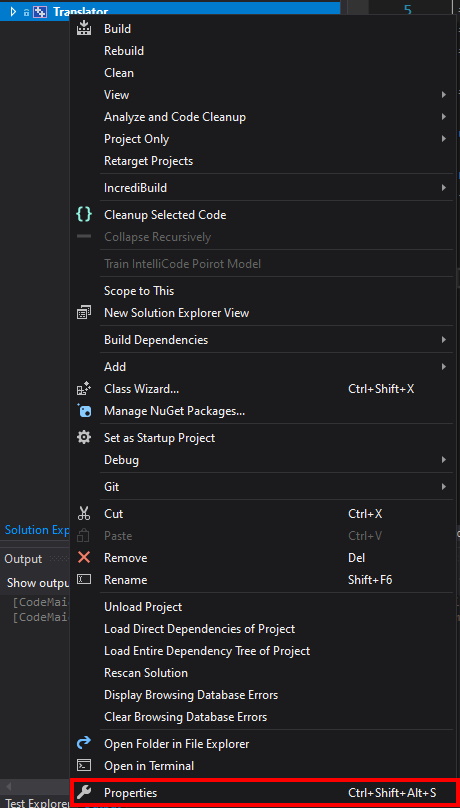
# Lekser

# Interfejs użytkownika

## Instalacja Leksera

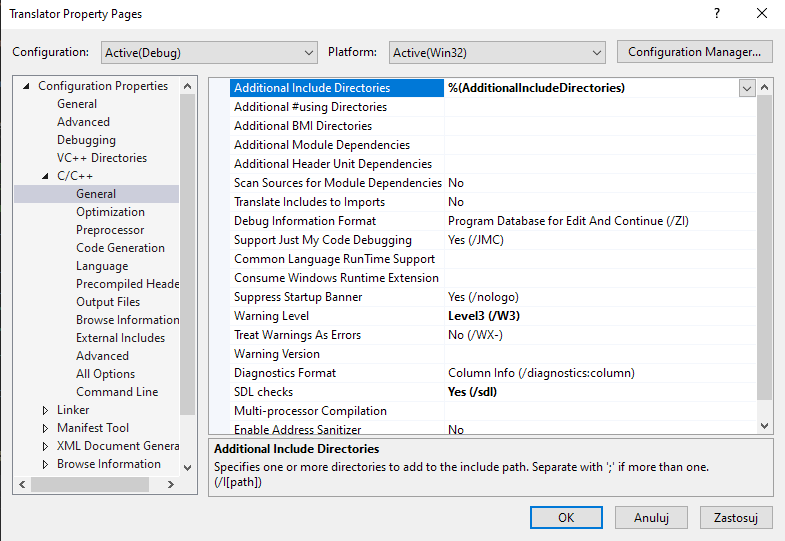
Lekser++ jest dynamiczną biblioteką c++. Pokażę jak poprawnie dodać ją do naszego projektu na przykładzie visualStudio2019.

Na początku musimy dodać nagłówki biblioteki, w tym celu otwieramy właściwości projektu, klikając prawym przyciskiem na jego nazwę w Solution Explorer (Eksploratorze rozwiązań).

.

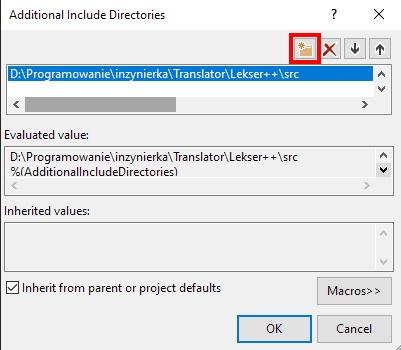
****Rysunek 4.1 Uruchomienie Właściwości.****

W zakładce C/C++ szukamy własności Additional Include Directories (Dodatkowe katalogi plików nagłówkowych).



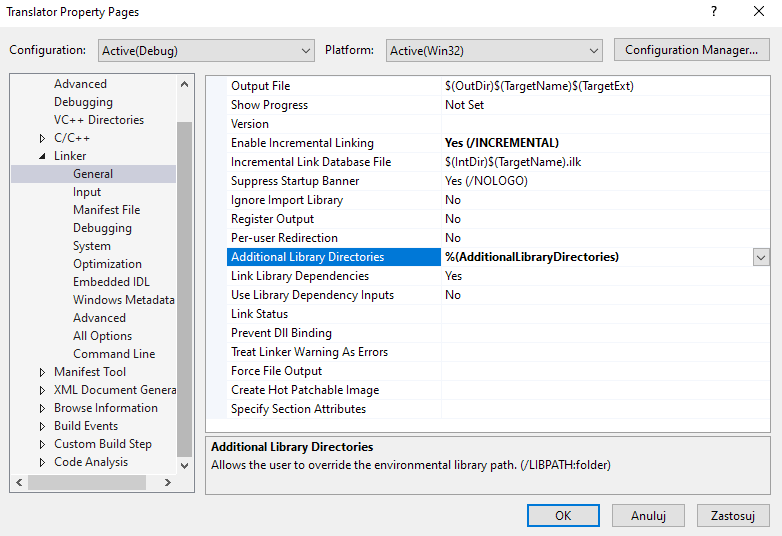
Rysunek . Additional Include Directories.

I dodajemy ścieżkę do folderu Lekser++/src.



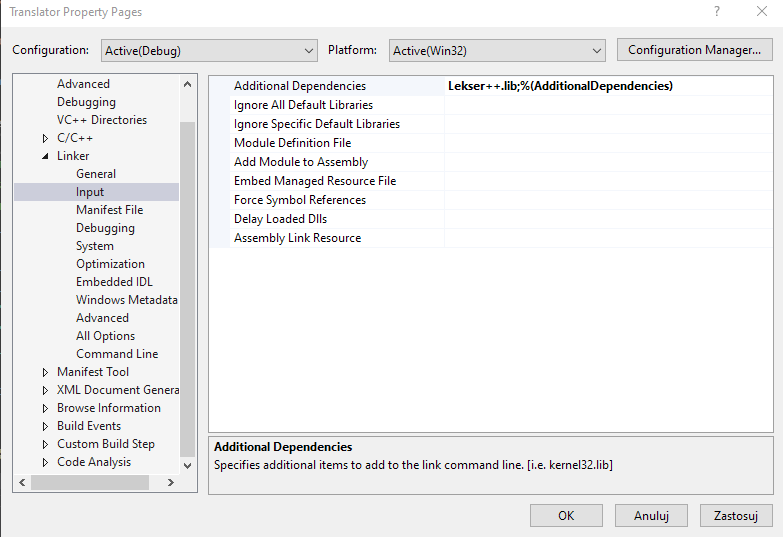
Rysunek . Dodawanie ścieżki.

W zakładce Linker (Konsolidator) w własności Additional Library Directories (Dodatkowe katalogi biblioteki) dodajemy ścieżkę do katalogu Lekser++/lib.



Rysunek . Additional Library Directories.

Przechodzimy do Linker (Konsolidator)->Input (Dane wejściowe) i właściwość Additional Dependencies (Dodatkowe zależności) wzbogacamy o Lekser++.lib.



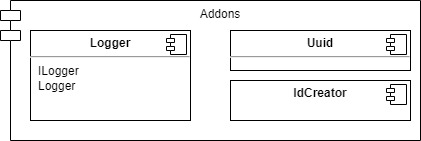
Rysunek . Dodanie Lekser++.lib

Tak skonfigurowany projekt jest gotowy do korzystania z Leksera++. Aby korzystać z funkcjonalności biblioteki należy importować plik Lekser++.h

#include "Lekser++.h"

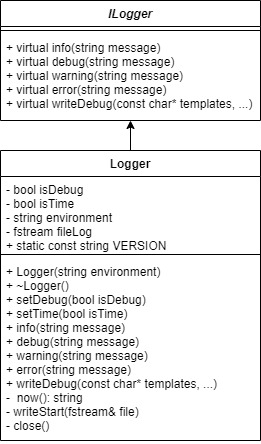
## Dodatki

Biblioteka korzysta z kilku dodatkowych funkcjonalności nie będących typowym działaniem leksera. Klasy te wydzialone sa do osobnej przestrzeni nazw i są umieszczone w folderze addons



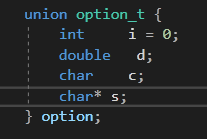
### Logger

Do łatwiejszego śledzenia i debuggowania w bibliotece logowane są wykonywane akcje oraz ich rezultaty. Mamy możliwość przekazania do konstruktora leksera własny obiekt spełniający interfejs deklarowany przez abstrakcyjną klasę ILekser w przeciwnym przypadku wykorzystana domyślna implementacja.



Domyślna implementacja loggera zapisuje informacje do pliku log.txt. Tworzy go w folderze którego nazwe definiujemy przez przekazanie go do konstruktora.

Ilość logowanych informacji możemy modyfikując flagę isDebug poprzez funkcje setDebug. Jeżeli flaga ta jest ustawiona na true to funkcja debug wpisze do pliku przekazaną wiadomość w argumencie dodając na początku wiadomości słowo [Debug]. Podobne działanie posiada funkcja writeDebug która przyjmuje minimum dwa argumenty i na ich podstawie buduje wiadomość do wypisania. Pierwszy argument tej funkcji jest ciągiem znaków który informuje o typach kolejnych argumentów.



# Implementacja leksera

# Testy oraz działanie

# Bibliografia

1. **Ben-Ari Mordechai.** *Understanding Programming Languages.* Chichester : John Wiley & Sons, 1996.

2. **Alfred V. Aho Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman.** *Kompilatory: reguły, metody i narzędzia .* Warszawa : Wydawnictwo Naukowe PWN SA, 2019.