Wrocław 13.06.2018

# Zadanie projektowe z przedmiotu Organizacja i Architektura Komputerów

## Temat: Prosty jednoprzebiegowy kompilator własnego języka programowania.

**Autorzy:**Jan Venulet 235542  
Piotr Kołeczek 234940

**Prowadzący projekt:**Mgr inż. Dominik Żelazny

# Wstęp teoretyczny.

Program, który stanowi nasze zadanie projektowe to analizator leksykalny wraz z analizatorem składniowym, tak zwanym „parserem”. Pełnią one funkcję analizatora tekstu o wcześniej określonej składni. Nasz język programowania, który został przez nas zaimplementowany wprowadza nową składnie, a co za tym idzie odgórnie wyznaczoną interpretacje ciągów znaków pliku tekstowego. Zamysłem realizacji projektu było stworzenie programu implementowanego w języku wysokiego poziomu C, gdzie znajduje się rdzeń naszego projektu, natomiast w języku niskopoziomowym, takim który bazuje na podstawowych operacjach procesora, zwanym językiem asemblera, napisane zostałyby wstawki realizujące konkretnie zadane funkcje. Program pozwoli na kompilacje, to znaczy w innym wypadku wyjdzie nie wykonując poleceń, jeśli podany do niego plik nie zawiera konstrukcji

nazwa\_pliku.l

, gdzie oczywiście *nazwa\_pliku* jest dowolnym ciągiem znaków, a znak „l” rozszerzeniem.

Kompilator został stworzony w środowisku systemowym LINUX ze względu na wygodę pracy i możliwości rozwoju naszego programu, a sam język programowania jest językiem interpretowanym, czyli język, programy, które zostały napisane w takim języku są kompilowane „w locie”, ponieważ na żądanie użytkownika uruchamiana jest pewna procedura, odpowiedzialna za pewną wykonywaną czynność.

W przypadku analizatorów leksykalnych należy znać kilka bazowych pojęć o tym w jaki sposób działają oraz o założeniach koncepcyjnych jakie towarzyszą autorom takich programów. Zaczynając od leksem-ów są to pojedynczo identyfikowane ( w sposób jednoznaczny ) ciągi znaków, które podczas analizy program będzie traktował jako naturalne części programu ( na przykładzie języka C++ : float, continue, „Hello\n”), operatory (np. „+” , „%” „^” ) oraz znaki interpunkcji programowej ( w C znak końca linii „;” bądź klamry i nawiasy „()” „{}” ) . Następnie, kiedy nasz program (język) ma już zidentyfikowane leksemy musimy przypisać im odpowiednie tokeny. Tokeny są to stanowią specjalną bazę leksemów do których odwołanie się jest zarezerwowane wyłącznie w określony przez program sposób. Kolekcję tokenów w języku C++ definiujemy słowem kluczowym let.

# Plan projektu

**Etap 1:**

* Implementacja arytmetyki czterodziałaniowej
* Operacja kończąca działanie programu

**Etap 2:**

* Zmiana dużych liter na małe
* Odbicie lustrzane stringu
* Operacje na liczbach przekraczających rozmiar rejestru architektury x86 ( np. 1024 bajtowe)
* Strumienie wejścia/wyjścia

**Etap 3:**

* Pętle oraz instrukcje warunkowe
* Deklaracja zmiennych programu

Ostateczne rezultaty założeń projektowych:

W etapie pierwszym wszystkie założenia dotyczące funkcjonalności zostały przez nas spełnione i mogą zostać przy analizie leksykalnej programu napisanego w JVPK.   
W etapie drugim nie udała się nam implementacja identyfikatorów odpowiadających za wprowadzanie modyfikacji do ciągu znaków (stringach) oraz operacji na liczbach przekraczających rozmiary rejestrów ogólnego przeznaczenia dostępnych w architekturze x86.  
W ostatnim etapie projektu wszystkie założone uprzednio funkcjonalności zostały umieszczone w możliwości naszego kompilatora (analizatora leksykalnego) za wyjątkiem instrukcji warunkowych, których implementacja jest dużo bardziej złożona od tego czego można się spodziewać.

Program własnego kompilatora analizującego język własnej implementacji okazał się dużym i trudnym wyzwaniem. O ile czytanie kolejno napisów czy symboli w pliku tekstowym, a następnie odnalezienia odpowiedniej instrukcji przypisanej do niej, nie jest trudne w implementacji, to skonstruowanie odpowiednio efektywnego skanera nie należy do zadań prostych. Istnieją nawet odpowiednie programy które umożliwiają na zautomatyzowane utworzenie i wygenerowanie analizatorów leksykalnych. Po podaniu odpowiednich specyfikacji opisującej wygląd leksemów i działanie „ukryte” pod ich powierzchnią. Przykładowo takim programem jest otwarte oprogramowanie Flex (Fast LEXical analyzer generator). Oczywiście w implementacji naszego projektu nie korzystaliśmy z takiego programowania, a jedynie przedstawiamy, poziom skomplikowania zadania projektowego, na które zdecydowaliśmy się.

# Przebieg prac.

W analizatorze leksykalnym skaner działa na zasadzie „następny leksem”. Oznacza to, że każdy wprowadzony znak, na który składa się kod jest analizowany oraz interpretowany. Dla zobrazowania tej idei rozpatrzmy prosty przykład, będący równaniem:

Jest to bardzo popularny wzór, opisujący równoważność masy i energii danego ciała. Został on zapisany w tak zwanej notacji matematycznej, który jest rzecz jasna niezrozumiały dla komputera.

Zatem spróbujmy przedstawić to samo równanie w troszkę inny sposób:

Powyższy zapis może stać się zapisem zrozumiałym dla komputera. Dlaczego? Otóż w tym zapisie równania możemy wyszczególnić każdy znak, któremu przypisujemy już jakąś funkcję.

**- „E”** – jest to znak określający lewą stronę równania;

**- „=”** - znak równości;

**- „m”** – jedna ze zmiennych równania;

**- „\*”** - operator iloczynu;

**- „c”** – druga ze zmiennych;

**- „^”** – operator potęgi;

**- „2”** – liczba.

I w taki właśnie sposób reprezentowany jest każdy kod, który interpretuje kompilator. Każdy kolejny znak jest analizowany, a następnie wyzwalana jego funkcja bądź procedura.

Poniżej znajduje się tabelka z niektórymi znanymi naszemu kompilatorowi znakami:

|  |  |  |
| --- | --- | --- |
| Leksem | Żeton | Atrybut |
| E | identyfikator | „E” |
| = | operator przypisania |  |
| m | identyfikator | „m” |
| \* | mnożenie |  |
| c | identyfikator | „c” |
| ^ | potęgowanie |  |
| 2 | liczba całkowita |  |

Tu jakieś nasze bym wstawił.

# Bibliografia

<http://edu.pjwstk.edu.pl/wyklady/jfa/scb/frames-jfa-main-node5.html>  
https://en.wikipedia.org/wiki/Flex\_(lexical\_analyser\_generator)  
<https://en.wikipedia.org/wiki/Lexical_analysis>  
https://hackernoon.com/lexical-analysis-861b8bfe4cb0