Informacje wstępne

1. Projekt realizowany jest w grupach dwuosobowych.
2. O ile nie zaznaczono inaczej w temacie, język implementacji dowolny
3. O ile z tematu nie wynika inaczej, w projektach należy wykorzystać (dowolny) generator skanerów i parserów, np. *PLY, AntLR, flex/bison*.
4. Dokumentacja projektu powinna zawierać m.in.:
   - tutorial do zaproponowanego języka,
   - specyfikację gramatyki języka w notacji ANTLR'a lub notacji BNF kub EBNF (*dot. projektów, w których specyfikuje się język*)
   - opis ewolucji języka (*dot. projektów, w których specyfikuje się język*)
   - opis przyjętego systemu typizacji i scopingu (*dot. projektów, w których specyfikuje się język*)
   - opis obsługi i użytkowania programu
   - opis architektury i komponentów programu
   - opis środowiska implementacji, w tym krótki opis użytych bibliotek innych niż generatory parserów
   - opis napotkanych problemów, specyficznych dla projektu i sposobów ich rozwiązania
5. Można zaproponować własny temat

Tematy projektów.

1. Translacja języka strukturalnego z użyciem infrastruktury LLVM do kodu pośredniego LLVM IR. Język implementacji C/C++.

2. Translacja języka strukturalnego z użyciem infrastruktury LLVM do kodu pośredniego LLVM IR. Język implementacji Python, LLVMPY

3. Optymalizacja kodu pośredniego.
W ramach projektu należy zaimplementować co najmniej jedną przykładową optymalizację kodu prostego języka programowania (np. eliminacja martwego kodu, eliminacja nieosiągalnego kodu, constant folding, propagacja stałych, eliminacja wspólnych podwyrażeń).
Optymalizacje kodu powinny być wykonane w infrastrukturze LLVM, poprzez stworzenie własnego przebiegu (Pass).
Jako język można przyjąć *kaleidoscope* oraz istniejący dla niego front-end i back-end (parsing oraz generacja kodu). Dla języka innego niż *kaleidoscope* dopuszcza się korzystanie z dostępnych specyfikacji gramatyk.

4. Implementacja i porównanie 2 algorytmów kompresji tablic skanera lub parsera np. metodą przemieszczania wierszy oraz metodą kolorowania grafów.

Tablica nie musi być wyliczana własnoręcznie przez program lecz może być uzyskana poprzez odwołanie się do odpowiednich struktur generatora parserów lub sparsowanie plików wynikowych genratora parserów

5. Eliminacja funkcji zagnieżdżonych metodą lambda-lifting.

6. Kompilator własnego języka (np. podzbiór języka C) do asemblera procesorów ARM [zestaw instrukcji ARM]

7. Kompilator własnego języka (np. podzbiór języka C) do asemblera procesorów ARM [zestaw instrukcji Thumb-2]

8. Efektywny interpreter prostej maszyny wirtualnej używający tzw. threaded code i superinstrukcji. Maszyna wirtualna może być wg własnego pomysłu.
Język implementacji: C/C++ z rozszerzeniami gcc

9. Program do tworzenia i ilustrowania struktur wykorzystywanych w analizie i optymalizacji kodu pośredniego (graf przepływu (CFG), drzewo dominacji, granica dominacji, graf wywołań (call graph))

10. Porównanie szybkości parsingu 2 metod:
- własnoręcznie implementowango parsera i skanera metodą recursive descent parsing,
- parsera (LA)LR implementowanego przy użycia flex/bison
na przykładzie własnych implementacji prostego języka.
Język implementacji: C/C++

11. Kompilator własnego języka do kodu pośredniego maszyny wirtualnej LUA

12. Kompilator własnego języka do kodu pośredniego maszyny wirtualnej Parrot

13. Program do eliminacji szablonów (template'ów) z kodu języka Java lub C++ (do wyboru).

14. Program umożliwiający upraszczanie wyrażeń matematycznych oraz symboliczne różniczkowanie funkcji.
15. Program do upraszczania wyrażeń regularnych.
16. Program do upraszczania formuł logicznych.

14-16. W projektach 14-16 wymagany język implementacji to SCALA (wykorzystanie pattern matching). Sugerowane narzędzia parsingu to Parsing Combinator lub ANTLR.

17. Translacja kodu do postaci Static Single Assignment (SSA).

SSA to rodzaj kodu pośredniego, który spełnia dwa warunki:
-każda definicja zmiennej posiada odrębną nazwę
-każde użycie zmiennej odnosi się do osobnej nazwy
Forma SSA wymaga wprowadzenia tzw. funkcji ϕ w miejscach, gdzie łączą się różne ścieżki grafu przepływu. Przy translacji do postaci SSA należy zwrócić uwagę, aby nie wygenerować zbyt dużej ilości ϕ-funkcji. W tym celu należy dokonać translacji do wersji *semipruned SSA*.

18. Optymalizacja kodu metodą inliningu.

Optymalizacja kodu metodą inliningu polega na zastąpieniu (kosztownego) wywołania funkcji/procedury jej ciałem, uwzględniając przy tym wiązanie parametrów. Istotnym elementem optymalizacji jest odpowiednie podjęcie decyzji, kiedy należy dokonać inliningu. W tym celu należy stworzyć heurystykę, która może uwzględniać wiele kryteriów, m.in.:
-wskazówki od programisty (słowo kluczowe *inline*),
-rozmiar wywoływanej funkcji,
-rozmiar funkcji wywołującej,
-liczba parametrów wywoływanej funkcji,
-liczba stałych parametrów aktualnych wywoływanej funkcji,
-liczba wywołań funkcji,
-głębokość zagnieżdżenia funkcji w pętlach.
Kod wynikowy może być generowany w języku wysokiego poziomu.

19. Optymalizacja kodu metodą procedure-placement.

Optymalizacja polega na zreorganizowaniu położenia procedur celem uniknięcia konfliktów w pamięci cache. Jeśli procedura *p* wywołuje procedurę *q*, to procedury te powinny zajmować bliskie sobie (adjacentne) lokacje w pamięci. Optymalizacja nie zapewni oczywiście spełnienia wszystkich warunków adjacencji (przykładowo procedura *p* wywołująca procedury *q,r,s* może być adjacentna tylko do jednej z nich).

Optymalizacja położenia procedur składa się z dwóch faz: analizy i transformacji i wymaga stworzenia grafu wywołań (call graph).
Kod wynikowy może być generowany w języku wysokiego poziomu.

20. Optymalizacja kodu metodą LCM (lazy-code-motion).

Celem optymalizacji jest przemieszczenie instrukcji celem eliminacji nadmiarowych lub częściowo nadmiarowych obliczeń. Optymalizacja wykonywana jest na kodzie pośrednim oraz grafie przepływu (control flow graph, CFG). Optymalizacja wymaga rozwiązania równań przepływu, w tym równań dotyczących wyrażeń dostępnych oraz wyrażeń antycypowanych.

21. Optymalizacja kodu poprzez rozwijanie i fuzję pętli (loop-unrolling).

Celem projektu jest opracowanie optymalizatora prostych pętli poprzez ich rozwijanie oraz ewentualne łączenie.
Rozwijanie pętli polega na kilkukrotnym kopiowaniu ciała pętli, dzięki czemu rzadziej sprawdzany jest warunek zakończenia pętli. Kopiowanie ciała pętli wymaga modyfikacji indeksów tablic. Konieczne może też być wygenerowanie prologu/epilogu pętli.
Przy rozwijaniu pętli zewnętrznych pętlę wewnętrzne powinny być łączone (tzw. fuzja).
Kod wynikowy może być generowany w języku wysokiego poziomu.

22. Badanie podobieństwa programów metodą strukturalną.

Celem projektu jest stworzenie programu do badania podobieństwa dwóch programów. Należy założyć, że celem ukrycia podobieństwa programy mogą być napisane w różnych językach programowania (np. w C lub Pythonie). Program powinien analizować

podobieństwo grafów przepływu oraz grafów wywołań a także brać pod uwagę inne metryki jak np. liczba użytych zmiennych.

23. Badanie podobieństwa programów metodą przesiewania (winnowing'u).

Celem projektu jest stworzenie programu do badania podobieństwa programów. Analizie powinny podlegać tylko fragmenty tekstu, reprezentowane w postaci haszy.

24. Program do wykrywania plagiatów w tekście metodą statystyczną. Język implementacji: C/C++.

25. Instrumentacja kodu dyrektywami preprocesora

W ramach projektu należy opracować zestaw pragm preprocesora służących instrumentacji kodu źródłowego programu w C (choć od strony technicznej nic nie stoi na przeszkodzie, by był to inny język strukturalny ew. OO) oraz preprocesor, który na podstawie pragm wygeneruje kod gromadzący i raportujący informacje o przebiegu wykonania programu. Na przykład kod postaci:

```
// here we clear a timer MyTimer, or create it if it did not exists thus
far

#pragma instr clear timer MyTimer

/*

some CPU intensive code goes here

*/

// and now we report the value of the timer MyTimer

#pragma instr report timer MyTimer with "ok, it took:"
```

może służyć raportowaniu czasu wykonania pewnego fragmentu programu. Mechanizm instrumentacji kodu musi być oczywiście bardziej rozbudowany. Np. powinien umożliwiać raportowanie nie tylko czasu wykonania, lecz także ilości iteracji pętli/wywołań funkcji, wartości zmiennych skalarnych/wskaźników w programie (w pętlach, zarówno wartości chwilowe jak i agregowane, tj. uśrednione/posumowane/maksymalne/minimalne), fakt niespełnienia assertu, i inne.

Ciekawy przykład rozbudowy języka programowania poprzez pragmy preprocesora studenci znajdą np. w standardzie OpenMP (tutaj pragmy służą implementacji obliczeń wielowątkowych).

26. Translator prostego języka obliczeń macierzowych na platformę NVIDIA CUDA.

W ramach projektu należy opracować prosty język obliczeń macierzowych, najlepiej jako podzbiór języka Octave/Matlab (z modyfikacjami, np. jawnymi deklaracjami typów), a następnie zaimplementować translator tego języka na platformę CUDA (platforma obliczeń na GPU opracowana przez firmę NVIDIA). Język powinien obejmować podstawowe elementy programowania strukturalnego (zmienne, stałe dosłowne, instrukcje sterujące, operatory logiczne, funkcje, ...), jeden całkowitoliczbowy i jeden zmiennoprzecinkowy typ skalarny oraz typ reprezentujący dwuwymiarowe tablice liczb zmiennoprzecinkowych (macierze) wraz z podstawowymi operatorami macierzowymi. Dodatkowo, język należy wyposażyć w podstawowe funkcje biblioteki standardowej (łączenie macierzy wzdłuż kolumn/wierszy, konstrukcja macierzy o zadanych rozmiarach, której każdy element zawiera tą samą liczbę zmiennoprzecinkową, wyznaczanie elementów maksymalnych/minimalnych wiersza/kolumny, ...).
Zespół, który wybierze ten projekt, powinien dysponować odpowiednią kartą graficzną.

27. Kompilator prostego języka do Common Intermediate Language (CIL), języka dla platformy .NET

28. Kompilator prostego języka do bytecodu JVM lub do jasmin.

29. Interpreter języka obiektowego

30. Interpreter języka funkcyjnego

31. Interpreter języka z leniwą ewaluacją

32. Interpreter bardzo prostego języka programowania logicznego.

33. Interpreter języka do obliczeń naukowych

Język powinien pozwalać na definiowanie zmiennych, wektorów, macierzy, list, funkcji, ze szczególnym uwzględnieniem operacji na macierzach.

34. Tworzenie diagramów klas w oparciu o analizę kodu źródłowego języka obiektowego (istniejącego lub zaproponowanego)

Program stworzony w tym projekcie powinien analizować kod i rysować diagramy klas z uwzględnieniem relacji dziedziczenia, powiązań, etc. Dodatkowe elementy: wyróżnianie składników (prywatnych, publicznych, statycznych, etc.), krotność i kierunek powiązań, etc. Diagramy powinny być zgodne ze standardem UML.

35. Program graficzny tworzący tablicę parsera ograniczonego kontekstu BC(m,n) dla podanej gramatyki
36. Program graficzny tworzący tablicę parsera LC (left-corner) dla podanej gramatyki
37. Program graficzny tworzący tablicę probabilistycznego parsera CYK dla podanej gramatyki

38. Program graficzny tworzący tablicę probabilistycznego parsera Earley’a dla podanej gramatyki
39. Program graficzny tworzący tablicę probabilistycznego parsera Ungera dla podanej gramatyki

35-39. Program powinien umożliwiać wczytywanie gramatyki z pliku tekstowego lub z GUI aby stworzyć dla niej odpowiednią tablicę. Dodatkowo powinna istnieć możliwość wpisania dowolnego słowa – parser powinien zbadać czy słowo to należy do języka generowanego przez gramatykę i podać możliwe drzewa parsingu słowa (z ich prawdopodobieństwami w przypadku parserów probabilistycznych). Gramatyka powinna zostać uprzednio przekształcona, np. poprzez usuwanie lewostronnej rekursji, lewostronną faktoryzację, sprowadzanie do postaci normalnej Chomsky’ego, jeśli zachodzi taka potrzeba parsera dla konkretnego parsera. Opis: ftp://ftp.cs.vu.nl/pub/dick/PTAPG_1st_Edition