JiPP zadanie 2 - „Interpreter”

W ramach zadania 2 należy zaimplementować interpreter języka programowania.

Za zadanie można uzyskać maksymalnie 24 punkty. Na ocenę ma wpływ zarówno zakres projektu („wielkość” i „trudność” języka wybranego do implementacji) jak i jakość rozwiązania.

Harmonogram

Zadanie podzielone jest na trzy fazy (trzecia faza jest opcjonalna).

1. **Deklaracja języka do implementacji – do wtorku 10 kwietnia 2018.**

   Należy oddać (przez Moodle'a) dokument zawierający opis języka wybranego do implementacji. Format pliku: PDF, płaski tekst lub ZIP zawierający pliki wspomnianego typu. Prosimy o nazwy plików postaci imie_nazwisko.pdf. Zawarte mają być:

   a) gramatyka języka – można ją podać w notacji EBNF (szczególnie zalecane osobom chcącym skorzystać z BNFC) lub w dowolnej rozsądnej postaci „na papierze” (można zacząć od pewnego poziomu abstrakcji – nie definiować literałów, identyfikatorów itp.),

   b) kilka przykładowych programów ilustrujących różne konstrukcje składniowe,

   c) tekstowy opis języka z podkreśleniem nietypowych konstrukcji; uwaga! nie opisujemy rzeczy oczywistych, np. w przypadku zapożyczeń ze znanych języków jak Pascal, C, Haskell wystarczy je wymienić, bez dokładnego opisu.

   Za niewykonanie w terminie tego etapu zadania od oceny końcowej zostanie odjętych 6 punktów, a za wykonanie niepełne (np. brak gramatyki) do 6 punktów. Na tym etapie rozmiar i trudność języka nie są oceniane.

   W odpowiedzi na deklarację języka sprawdzający poda (w terminie do 24.4) maksymalną ocenę, jaką można uzyskać za poprawne zrealizowanie takiego języka. Ostateczny zakres projektu może jednak jeszcze zostać zmieniony w porozumieniu ze sprawdzającym.

2. **Pierwsza wersja interpretera – do wtorku 15 maja.**

   Działająca implementacja interpretera. Należy oddać plik imie_nazwisko.zip, o zawartości opisanej poniżej. Oddanie rozwiązania przez Moodle'a w terminie jest obowiązkowe. Dodatkowo sprawdzający może poprosić o osobistą prezentację rozwiązania.

3. **Ostateczna wersja interpretera – do wtorku 12 czerwca**.

   Opcjonalnie, w porozumieniu ze sprawdzającym – dodatkowe funkcjonalności w celu podwyższenia oceny.

   Uwaga, uzupełnianie istotnych błędów i braków znalezionych przez sprawdzającego może nie rekompensować w pełni odjętych punktów. Na pewno nie można liczyć na bardzo dobrą ocenę oddając początkowo „wydmuszkę” i uzupełniając rozwiązanie dopiero w drugiej iteracji.

Implementacja i zawartość paczki

Interpreter należy zaimplementować w Haskellu.

Jako rozwiązanie należy oddać plik imie_nazwisko.zip, który po rozpakowaniu tworzy katalog imie_nazwisko, w którym wywołanie polecenia make buduje bez przeszkód (na maszynie students, najlepiej przy użyciu kompilatora Haskella w wersji 8.2.2 umieszczonej w katalogu /home/students/inf/PUBLIC/MRJP/ghc-8.2.2/bin) działający interpreter. Ma się on uruchamiać poleceniem ./interpreter program, gdzie program oznacza plik z programem do interpretacji. Ponadto, jeśli w języku nie ma obsługi standardowego wejścia, interpreter wywołany bez parametru może wczytywać program ze standardowego wejścia. Wyniki mają wypisywać się na standardowe wyjście, komunikaty o błędach na standardowe wyjście błędów. Domyślnie interpreter nie powinien wypisywać żadnych dodatkowych komunikatów diagnostycznych.

Rozwiązanie ma również zawierać plik README z ogólnym opisem rozwiązania i sposobu uruchomienia (gdyby był nieoczywisty) oraz przykładowe programy:

- w podkatalogu good przykłady poprawnych programów ilustrujących **wszystkie** punktowane konstrukcje języka,

- w podkatalogu bad przykłady niepoprawnych programów, aby zilustrować działanie interpretera w sytuacjach wyjątkowych, na przykład (ale w zależności od języka listę należy wydłużyć):
  - błędy składniowe (nie za dużo),
  - nieznany identyfikator, nieznana funkcja itp.,
  - zła liczba argumentów,
  - błędy typów i inne błędy wykrywane statycznie (jeśli rozwiązanie wspiera statyczną kontrolę typów),
  - błędy czasu wykonania (dzielenie przez zero, odwołanie do indeksu spoza tablicy itp.).

Język

Nie ma wstępnych ograniczeń na język - może być imperatywny, funkcyjny, obiektowy, logiczny albo mieszany. Szczególnie cenimy nieszablonowe pomysły i dla takich „ciekawych" języków sprawdzający mogą podejmować indywidualne decyzje odnośnie oceny nawet jeśli język nie spełnia podanych poniżej wymagań dla projektów standardowych. Nie będziemy natomiast wysoko cenić języków, których oryginalność ogranicza się do wymyślnej składni.

Dla ułatwienia studentom decyzji, a sprawdzającym ujednolicenia ocen, przedstawiamy poniżej dwie linie standardowych projektów i wymagania na poszczególne oceny (przy założeniu poprawnej i porządnie napisanej implementacji).

Oczywiście w celu uzyskania pełnej liczby punktów na danym poziomie należy oprócz danych cech języka również zaimplementować naturalnie wynikające kombinacje tych cech, np. dla tablic i rekordów również tablice rekordów i rekordy zawierające tablice itp.

Docelowa gramatyka języka powinna być w miarę możności wolna od konfliktów. Jeśli niektórych konfliktów nie da się (w miarę łatwo) uniknąć, należy je udokumentować - podać przykłady wyrażeń prezentujących konflikt oraz skutki rozwiązania konfliktu przyjętego przez użyty generator parserów.

Powinna być możliwość umieszczania w programach komentarzy.

Jako generator parsera polecamy BNFC w wersji dostępnej na maszynie students w katalogu /home/students/inf/PUBLIC/MRJP/bin lub https://github.com/BNFC/bnfc/tree/176-source-position.

Język imperatywny

Język imperatywny, najlepiej w składni opartej o Pascala lub C. W przypadku braku własnych oryginalnych pomysłów można wzorować się na języku Latte
https://www.mimuw.edu.pl/~ben/Zajecia/Mrj2017/Latte/

Na 6 punktów

1. Jeden typ wartości, np. `int`.
2. Zmienne, operacja przypisania.
3. `if`.
4. `while` lub `goto`.
5. Wyrażenia z arytmetyką `+ - * / ( )`.
6. Porównania (dopuszczalne tylko w warunkach lub z interpretacją liczbową 0/1 jak w C).

Wykonanie może polegać na wykonaniu ciągu instrukcji i wypisaniu stanu końcowego.

Na 10 punktów

J.w., a dodatkowo:

7. Funkcje lub procedury z parametrami przez wartość, rekurencja.

Na 14 punktów

1. Co najmniej dwa typy wartości w wyrażeniach: `int` i `bool`
   (to znaczy `if 2+2 then _` parsuje się, ale wyrażenie ma niepoprawny typ).

2. Arytmetyka, porównania.
3. `while`, `if` (z `else` i bez, może być też składnia `if _ elif _ else _ endif`).
4. Funkcje lub procedury (bez zagnieżdżania), rekurencja.
5. Jawne wypisywanie wartości na wyjście (instrukcja lub wbudowana procedura `print`).
6. Dwie wybrane rzeczy z poniższej listy lub coś o porównywalnej trudności:
   a) dwa sposoby przekazywania parametrów (przez zmienną / przez wartość),
   b) pętla `for` w stylu Pascala,
   c) typ `string`, literały napisowe, wbudowane funkcje pozwalające na rzutowanie między napisami a liczbami,
   d) wyrażenia z efektami ubocznymi (przypisania, operatory języka C `++`, `+=` itd).

Na 17 punktów

J.w, a ponadto:

7. Statyczne typowanie (tj. zawsze terminująca faza kontroli typów przed rozpoczęciem wykonania programu). (Wymaganie nie dotyczy nietrywialnych projektów, w których dynamiczne typowanie jest istotną cechą wybranego języka, np. Smalltalk, JavaScript).

Na 20 punktów

J.w., a ponadto:

8. Przesłanianie identyfikatorów ze statycznym ich wiązaniem (np. zmienne globalne i lokalne w funkcjach lub lokalne w blokach).
9. Jawnie obsłużone dynamiczne błędy wykonania, np. dzielenie przez zero.
10. Funkcje zwracające wartość (tzn. nie tylko procedury; za to mogą być tylko funkcje – jak w języku C).
11. Dwie dodatkowe rzeczy z poniższej listy lub coś o porównywalnej trudności:
    a) rekordy,
    b) tablice indeksowane `int` lub coś à la listy,
    c) tablice/słowniki indeksowane dowolnymi porównywalnymi wartościami; typ klucza należy uwzględnić w typie słownika,
    d) dowolnie zagnieżdżone krotki z przypisaniem jak w Pythonie (składnia wedle uznania),
    e) operacje przerywające pętlę `while` - `break` i `continue`,
    f) funkcje jako parametry,
    g) zwracanie funkcji w wyniku, domknięcia à la JavaScript.
    h) funkcje anonimowe (szczególnie sensowne w połączeniu z punktem f).

Na 24 punkty

J.w., a ponadto wymagane:

12. Dowolnie zagnieżdżone definicje funkcji / procedur z zachowaniem poprawności statycznego wiązania identyfikatorów (jak w Pascalu).
13. Jeszcze jedna-dwie funkcjonalności z listy powyżej.

Język funkcyjny

Język funkcyjny, najlepiej w składni opartej o składnię SML/Camla lub Haskella (lub Lisp/Scheme dla prostszych).

Na 5 punktów

Język wyrażeń

1. Jeden typ wartości, np. `int`.
2. Zmienne, lokalna deklaracja (`let _ in`). Wiązanie może być dynamiczne lub statyczne.
3. Wyrażenie warunkowe `if`.
4. Wyrażenia z arytmetyką `+ - * / ( )`.
5. Porównania (dopuszczalne tylko w warunkach lub z interpretacją liczbową 0/1 jak w C).

Wykonanie może polegać na ewaluacji wyrażenia i wypisaniu wyniku.

Na 8 punktów

J.w., a dodatkowo:

6. Funkcje anonimowe lub nazwane, chociaż jednoargumentowe, rekurencja (może być zrobiona jako jawny fixpoint z funkcją anonimową).

Na 10 punktów

Jak na 5 punktów, a dodatkowo:

7. Nazwane funkcje wieloargumentowe, normalna rekurencja.

Na 12 punktów

J.w., a dodatkowo **jedna** z poniższych rzeczy:

a) Funkcje anonimowe, częściowa aplikacja i funkcje wyższego rzędu.

b) Struktura pozwalająca budować i przetwarzać listy `int`ów (wystarczy `[]` i `:` lub `nil` i `cons`) z wbudowanym w język pattern matchingiem (składnia może być bardzo uproszczona) lub zestawem funkcji (patrz niżej).

Na 16 punktów

1. Co najmniej dwa typy wartości w wyrażeniach: `int` i `bool`
   (to znaczy `if 2+2 then _` parsuje się, ale wyrażenie ma niepoprawny typ).
2. Arytmetyka, porównania.
3. Wyrażenie warunkowe `if`.
4. Funkcje wieloargumentowe, rekurencja.
5. Funkcje anonimowe, częściowa aplikacja i funkcje wyższego rzędu.
6. Listy z
   a) pattern matchingiem `[] | x:xs` (składnia może być inna niż w Haskellu),
   b) **lub** zestawem wbudowanych operacji: `empty`, `head`, `tail`,
   c) mile widziany lukier syntaktyczny do wpisywania stałych list, np.: `[1,2,3]`.

Na 20 punktów

J.w., a ponadto:

7. Listy dowolnego typu, także listy zagnieżdżone i listy funkcji
8. **Lub** ogólne rekurencyjne typy algebraiczne (jak `data` w Haskellu) z pattern matchingiem. Mogą być monomorficzne a pattern matching jednopoziomowy.
9. Statyczne wiązanie identyfikatorów przy dowolnym poziomie zagnieżdżenia definicji.
10. Statyczne typowanie (tj. zawsze terminująca faza kontroli typów przed rozpoczęciem wykonania programu). Na tym poziomie można wymagać jawnego podawania typów, nawet dla każdego wprowadzanego identyfikatora.

Na 24 punkty

J.w., a ponadto:

11. Ogólne **polimorficzne i rekurencyjne** typy algebraiczne.

    Mile widziane wykonane w samym języku definicje typów `List` (składnia może być inna niż w Haskellu, lukier syntaktyczny nie wymagany), `Maybe` i `Either` oraz ich zastosowania w przykładowych programach.
12. Typy polimorficzne w stylu ML (jak Caml lub Haskell bez klas) z algorytmem rekonstrukcji typów. W tej wersji nie jest konieczna (ale zabroniona też nie) składnia do deklarowania typów.
13. Dowolne zagnieżdżenie wzorców w pattern matchingu. Ostrzeżenie przy próbie zdefiniowania funkcji częściowej.

Zapożyczenia

Projekt zaliczeniowy ma być pisany samodzielnie. Wszelkie przejawy niesamodzielności będą karane. W szczególności nie wolno oglądać kodu innych studentów, pokazywać, ani w jakikolwiek sposób udostępniać swojego kodu.

Dopuszczalne są jawne (z podaniem źródła i poszanowaniem praw autorskich) zapożyczenia elementów rozwiązania nie negujące własnego wkładu pracy i intelektu w całokształt projektu, np.:

- wykorzystanie ogólnie dostępnej gramatyki języka,
- oparcie się w swoim rozwiązaniu o dostępny (i wskazany w rozwiązaniu) opis pewnej techniki lub algorytmu (np. realizacja przesłaniania identyfikatorów, algorytm rekonstrukcji typów); kod ma być w tym przypadku napisany samodzielnie.

W rozwiązaniu można do woli korzystać z **własnych** projektów realizowanych przy innej okazji, np. na potrzeby innych przedmiotów czy w poprzedniej edycji JiPP. Należy jednak pamiętać, że oczekiwany jest interpreter (a nie np. kompilator) i że projekt będzie oceniany od nowa.

Przydział grup

Koordynatorem merytorycznym zadania jest Jacek Chrząszcz.

Sprawdzającymi dla poszczególnych grup są:

- grupa 1, 6, 9 – Jacek Chrząszcz
- grupa 2 i 3 – Vincent Michielini
- grupa 5 – Lorenzo Clemente / Daria Walukiewicz-Chrząszcz (podział poprzez moodle)
- grupa 7 – Lorenzo Clemente
- grupa 8 – Daria Walukiewicz-Chrząszcz