JiPP zadanie 2 - „Interpreter”

W ramach zadania 2 należy zaimplementować interpreter języka programowania.

Za zadanie można uzyskać maksymalnie 30 punktów. 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 piątku 12 kwietnia 2019.**

   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

   d) liczbę punktów, jaką autor spodziewa się dostać za interpreter, o ile poprawnie zaimplementuje całą podaną funkcjonalność – patrz (dużo) niżej.

   Za niewykonanie w terminie tego etapu zadania od oceny końcowej zostanie odjętych 8 punktów, a za wykonanie niepełne (np. brak gramatyki) do 8 punktów.

   W odpowiedzi na deklarację języka sprawdzający potwierdzi (lub nie :) 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 piątku 17 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 11 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.

Ze względu na łatwość obsługi błędów 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 C lub Pascal. W przypadku braku własnych oryginalnych pomysłów można wzorować się na języku Latte https://www.mimuw.edu.pl/~ben/Zajecia/Mrj2018/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 12 punktów

J.w., a dodatkowo:

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

Na 15 punktów

1. Co najmniej trzy typy wartości: `int`, `bool` i `string`
   (to znaczy `if 2+2 then _` parsuje się, ale wyrażenie ma niepoprawny typ).
2. Literały, arytmetyka, porównania.
3. Zmienne, operacja przypisania
4. Jawne wypisywanie wartości na wyjście (instrukcja lub wbudowana procedura `print`).
5. `while`, `if` (z `else` i bez, może być też składnia `if _ elif _ else _ endif`).
6. Funkcje lub procedury (bez zagnieżdżania), rekurencja.
7. Jedna rzecz z poniższej listy lub coś o porównywalnej trudności:
   a) co najmniej dwa sposoby przekazywania parametrów (przez zmienną / przez wartość / in/out),
   b) zmienne „read-only” i użycie ich np. w implementacji pętli `for` w stylu Pascala
   (`for i = pocz to kon` - wewnątrz pętli nie można zmienić wartości zmiennej sterującej, wartość `kon` liczona tylko raz - przed wejściem do pętli)

Na 20 punktów

J.w., a ponadto:

8. Przesłanianie identyfikatorów ze statycznym ich wiązaniem (zmienne lokalne i globalne lub zagnieżdżone procedury/funkcje).
9. Obsługa błędów wykonania, np. dzielenie przez zero (może być elegancki komunikat i zatrzymanie interpretera).
10. Funkcje przyjmujące i zwracające wartość dowolnych obsługiwanych typów (tzn. nie tylko procedury; za to mogą być tylko funkcje – jak w języku C).

Do 30 punktów wg cennika...

4pkt	statyczne typowanie (tj. zawsze terminująca faza kontroli typów przed rozpoczęciem wykonania programu)
2pkt	dowolnie zagnieżdżone definicje funkcji / procedur z zachowaniem poprawności statycznego wiązania identyfikatorów (jak w Pascalu)
1pkt	rekordy,
1pkt	tablice indeksowane `int` lub coś à la listy,
2pkt	dowolnie zagnieżdżone krotki z przypisaniem jak w Pythonie (składnia wedle uznania),
1pkt	operacje przerywające pętlę `while` - `break` i `continue`,
4pkt	funkcje jako parametry, zwracanie funkcji w wyniku, domknięcia à la JavaScript. funkcje anonimowe
3pkt	procedury generujące i składnia do ich używania (np. jak w Pythonie - instrukcja `yield` oraz `next` i `for i in generator(...)`)

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 9 punktów

J.w., a dodatkowo:

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

Na 12 punktów

Jak na 5 punktów, a dodatkowo:

7. Nazwane funkcje wieloargumentowe, normalna rekurencja.

Na 15 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 20 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, funkcje wyższego rzędu, domknięcia.
6. Obsługa błędów wykonania, np. dzielenie przez zero (może być elegancki komunikat i zatrzymanie interpretera).
7. 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 25 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.

Na 30 punktów

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. Dowolne zagnieżdżenie wzorców w pattern matchingu. Ostrzeżenie przy próbie zdefiniowania funkcji częściowej.

Bonus do 5 punktów

13. Typy polimorficzne w stylu ML (jak Caml lub Haskell bez klas) z algorytmem rekonstrukcji typów. Nie jest konieczna (ale zabroniona też nie) składnia do deklarowania (skrótów) typów.

    Za interpreter implementujący ten algorytm można dostać więcej niż 30 punktów.

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.