Interpreter - Robert Suchocki

Sposób uruchomienia

Po wykonaniu make w katalogu (przy problemach z make uwaga na końcu dokumentu: Uwagi końcowe - punkt drugi) pojawi się wykonywalny plik interpreter, program akceptuje ścieżkę pliku podaną jako argument uruchomienia lub czyta z wejścia, gdy ścieżka nie zostanie podana. Obie poniższe komendy są równoważne

- ./interpreter ./folder/program
- ./interpreter < ./folder/program

Krótki opis

Interpreter korzysta z monad Reader, Except, State i IO. W monadzie Reader przechowywane jest środowisko zmiennych oraz funkcji, które może ulec zmianie tylko i wyłącznie w części deklaracji bloku, która wypełnia środowisko zmiennymi, a następnie w tym środowisku uruchamia resztę bloku, czyli instrukcje. Monada State przechowuje “store” z wartościami zmiennych, na początku zapisywane są do niego wartości inicjalizujące zmienne w czasie deklaracji, a później może (i często ulega) zmianie. Środowisko obecne podczas deklaracji funkcji jest wraz z jej argumentami i blokiem zapamiętywane i odtwarzane na czas wykonania tej funkcji, podczas gdy “store” jest globalny i wszystkie zmiany wykonane w dowolnym momencie w nim pozostają. Do tego monady Except używam przy błędach czasu wykonania programu, które zostają przy pomocy tej monady rzucone, a następnie przerywają wykonanie reszty programu i zostają wypisane. Na końcu jest monada IO służąca do obsługi funkcji print i komunikatów diagnostycznych. Do tego wykonanie programu poprzedza faza weryfikacji typów zmiennych, która podobnie jak program idzie po drzewie instrukcji programu i wykrywa błędy w typowaniu przed wykonaniem samego programu

Zmiany względem deklaracji gramatyki

- Rezygnacja z dwóch sposobów przekazywania parametrów na rzecz dodania operacji przerywających pętle while
- Podział bloku na listę deklaracji i instrukcji, brak możliwości deklarowania między instrukcjami (powodują błąd parsowania)
- Zmiana arrayów na listy
    - Nie jest podawana długość przy tworzeniu listy
    - Można pobierać wartości na indeksach 0..len-1
    - Można zapisywać na indeksach 0..len, gdzie zapis na len zapisuje na koniec listy i ją wydłuża o jeden
- Zmiana składni dict z {} na {{}} celem uniknięcia konfliktów w bnfc

Cechy języka Breve

- Typy zmiennych: int, bool, str, list, dict, statycznie typowane
- Zmienne, operacje przypisania, działania arytmetyczne i logiczne, porównania, wyrażenia z efektami ubocznymi (‘++’, ‘--’)
- Instrukcje if, if/else, while, “pascalowy” for
- Funkcje z parametrami przez wartość i zmienną, z rekurencją i zagnieżdżaniem (z przesłanianiem, z zachowaniem poprawności statycznego wiązania identyfikatorów)
- Obsługa dynamicznych błędów wykonania, wbudowane funkcje do wypisywania na wyjście, funkcje rzutujące między typami int i str

Zakres

Wszystkie (nie rozwijające się na listę podpunktów do wyboru) wymagania na 24 punkty

Z punktu 6. (3)

~~a) dwa sposoby przekazywania parametrów (przez zmienną / przez wartość)~~

b) pętla for w stylu Pascala

c) typ str, literały napisowe, wbudowane funkcje pozwalające na rzutowanie między napisami a liczbami

d) wyrażenia z efektami ubocznymi (przypisania, operatory języka C ++, += itd)

Z punktu 11. (3)

b) tablice indeksowane int lub coś à la listy

c) tablice/słowniki indeksowane dowolnymi porównywalnymi wartościami; typ klucza należy uwzględnić w typie słownika

e) operacje przerywające pętlę while - break i continue

Szczegóły konstrukcji rozszerzających Latte

- Array (indeksowany wartościami int)
  - Typ [typ_wartości]
  - Tworzenie array = []
  - Przypisanie array[indeks] = wartość
  - Dostęp array[indeks]
  - Funkcje length
- Dict
  - Typ {{typ_klucza, typ_wartości}}
  - Tworzenie dict = {{}}
  - Przypisanie dict{{klucz}} = wartość
  - Dostęp dict{{klucz}}
  - Funkcje has_key, delete_key

- “Pascalowy” for
    - Składnia for i = first..last do statement
    - Zarówno inkrementacyjny jak i dekrementacyjny
- Operacje przerywające pętlę while (oraz pętlę for)
    - break;
    - continue;
- Zagnieżdżanie funkcji

Uwagi końcowe

- Konflikt shift/reduce wynika z domyślnej gramatyki Latte i wynika z obecności reguł na if then oraz if then else, które mogą być zagnieżdżane bez konieczności tworzenia bloku, który rozwiązuje konflikty w takich konstrukcjach
- Ostatecznie nie udało mi się skorzystać z ghc i bnfc udostępnionego w folderze PUBLIC na students, ghc nie widzi modułu Control.Monad.Reader mimo tego, że mtl jest zainstalowany przez cabal, JEDNAKŻE nie znam cabala dokładnie, więc zakładam, że możliwy jest scenariusz, w którym cabal ciągle widzi tylko domyślne ghc, ale nie udało mi się go skonfigurować. W pliku Makefile korzystam więc z domyślnych ghc (oczywiście wymagane jest cabal install mtl) i bnfc, ale aby ewentualnie oszczędzić Panu kilku kliknięć dołączam dodatkowo plik Makefile_public, gdzie podane są bezwzględne ścieżki i być może przy problemach ze zwykłym Makefilem podmiana na Makefile_public zadziała
- Zaimplementowane łączenie list ze słownikami jest mocno ograniczone przez gramatykę, która zakłada dostęp do tych struktur na poziomie identyfikatorów, przez co dostęp do elementów listy list (lub słownika słowników) wymaga zapisanie listy wewnętrznej (lub słownika wewnętrznego) na pomocniczy identyfikator i dostęp do wewnętrznego elementu przez ten identyfikator, poza tym ograniczeniem wszystko działa poprawnie, szczegóły użycia w plikach adv_dicts.bv i adv_lists.bv w folderze good
- Problematycznym okazało się dla mnie zapewnienie, że z danej funkcji typu niebędącego voidem zostanie wykonany return (głównie przez break i continue), dlatego na etapie statycznego typowania optymistycznie sprawdzam istnienie returna, tzn. sprawdzam jego obecność w pętlach, mimo że nie musi do nich wejść oraz w obu gałęziach if then else’a, co powinno wykrywać sytuacje, w których nie ma absolutnie żadnego przejścia przez funkcję kończącego się returnem. W przeciwnym przypadku, jeśli ten return jednak nie nastąpi, np. przez dodanie breaka w klauzuli if (jak w przypadku bad/runtime_return.bv)
- Nie wykrywam i nie wyrzucam błędów w przypadku przedeklarowania zmiennej (good/redeclaration.bv), jeżeli zmienia się typ zmiennej to nie wpływa to na poprawne otypowanie pozostałej części, tylko powoduje pewnego rodzaju “przesłanianie” dla reszty bloku. Nie postrzegałem tego za wadę, dlatego nie ograniczałem tego, ale w razie wątpiwości co do zasadności tego wyjścia zrobię to przy ewentualnej drugiej wersji