Piotr Jarosik

Dokumentacja końcowa

Projekt TKOM

Temat projektu:

Zadanie 16. Interpreter języka w C++ z wbudowanym typem macierzy (wymagane minimum: zmienne globalne i lokalne, funkcje, operatory arytmetyczne +,-,*,/ i logiczne >,>=,<,<=,<>, przypisania, instrukcja pętli i warunku logicznego).

1. Zakładana funkcjonalność

Realizowany projekt języka zakłada:
- typowanie silne, dynamiczne – typ zmiennej będzie określany na podstawie przypisanej (po raz pierwszy) jej wartości;
- składnię wzorowaną na języku C z elementami typowymi dla języka Pascal.

Podstawowe typy danych języka: macierz liczb ułamkowych oraz liczba całkowita i ułamkowa w zapisie dziesiętnym.

Dopuszczalne są następujące operacje arytmetyczne i porównania na typach danych:
- dodawanie, odejmowanie, mnożenie, dzielenie liczb (stało- i zmiennoprzecinkowych);
- dodawanie macierzy o takich samych wymiarach;
- mnożenie macierzy, jeżeli liczba kolumn lewej macierzy jest równa liczbie wierszy prawej macierzy;
- dowolne porównywanie liczb (dostępne operacje podane w treści zadania);
- porównywanie równościowe oraz nierównościowe macierzy.

Konwencje leksykalne:

litera → [a-zA-Z]
cyfra → [0-9]
cyfra_niezerowa → [1-9]

identyfikator → **litera** (**litera** | **cyfra** | _)*
liczba → (**cyfra_niezerowa cyfra*** **|** 0)
liczba_ułamkowa → **liczba** . [0-9]+

znak → + | -
op_przypisania → =
op_multiplikatywny → * | / | and
op_relacji → > | >= | < | <= | != | ==

Składnia języka – produkcje:
Symbol startowy: *lista_operacji*
Oznaczenia: nieterminale – oznaczone kursywą, terminale – pozostałe.

lista_operacji →
    operacja
    | *lista_operacji operacja*

operacja →
    definicja_funkcji
    | *instrukcja*

definicja_funkcji →
    defn identyfikator (*lista_parametrów_formalnych*) *blok_instrukcji*

lista_parametrów_formalnych →
    lista_identyfikatorów
    | ε

lista_identyfikatorów →
    lista_identyfikatorów , **identyfikator**
    | **identyfikator**

blok_instrukcji →
    { *lista_instrukcji* }

lista_instrukcji →
    lista_instrukcji instrukcja
    | *instrukcja*

instrukcja →
    utworzenie_zmiennej **separator**
    | ***identyfikator op_przypisania*** *wyrażenie* **separator**
    | *wywołanie_funkcji* **separator**
    | *instrukcja_warunkowa*
    | *instrukcja_pętli*

utworzenie_zmiennej →
    val identyfikator
    | **val** identyfikator **op_przypisania** *wyrażenie*

zmienna →
    identyfikator
    | **identyfikator** *'|' wyrażenie , wyrażenie* ****

Sekwencja | i, j \ umożliwia dostęp do elementu macierzy z i-tego wiersza, j-tej kolumny.

macierz →
    [*macierz_wyrażeń*]

macierz_wyrażeń →
    macierz_wyrażeń **separator** argumenty
    | *argumenty*

lista_wyrażeń →
    lista_wyrażeń , *wyrażenie*
    | *wyrażenie*

element_macierzy →
    macierz '|' wyrażenie , wyrażenie ****
    | identyfikator *'|' wyrażenie , wyrażenie* ****

wyrażenie →
    wyrażenie_arytmetyczne
    | *wyrażenie* **op_relacji** *wyrażenie_arytmetyczne*

wyrażenie_arytmetyczne →
    składnik
    | *wyrażenie_arytmetyczne* op_addytywny *składnik*

op_addytywny →
    znak | or

składnik →
| **znak** czynnik
| *czynnik*
| *składnik* **op_multiplikatywny** *czynnik*

czynnik →
liczba
| liczba_ułamkowa
| macierz
| zmienna
| *wywołanie_funkcji*
| (*wyrażenie*)
| **not** *czynnik*

wywołanie_funkcji →
identyfikator (*argumenty*)

argumenty →
lista_wyrażeń
| *ε*

instrukcja_warunkowa →
if (*wyrażenie*) *wynik_warunku*
| **if** (*wyrażenie*) *wynik_warunku* **else** *wynik_warunku*

instrukcja_pętli →
while *(wyrażenie) wynik_warunku*

wynik_warunku →
blok_instrukcji
| *instrukcja*

Uwagi do semantyki poszczególnych elementów składni języka:
- instrukcje nie należące do definicji żadnej z funkcji są traktowane jako instrukcje głównego programu;
- realizowany język nie umożliwia realizacji procedur – każda funkcja musi coś zwracać; wartością zwracaną przez funkcję jest wartość zmiennej lokalnej o takiej samej nazwie jak ta funkcja; domyślnie na początku wywołania funkcji zmienna ta jest inicjalizowana na wartość 0;
- możliwe jest definiowanie zmiennych poprzez użycie operatora **val;**
  w definicji funkcji oznacza to utworzenie zmiennej lokalnej, w pozostałych przypadkach tworzona jest zmienna globalna;
- komentarze są umieszczone między /* */.

2. Wymagania funkcjonalne i niefunkcjonalne

- **interpretacja pliku wejściowego** – użytkownik będzie miał możliwość wskazania pliku wejściowego, stanowiącego wejściowy ciąg operacji;
- **powiadamianie o błędach** – realizowane będzie powiadamianie o błędach składniowych, semantycznych, oraz błędach w czasie wykonania programu (np. nieprawidłowy typ operandu);

wymagania niefunkcjonalne
- **przenośność** – zaimplementowany interpreter będzie możliwy do skompilowania i uruchomienia niezależnie od systemu operacyjnego;
- **język implementacji interpretera** – C++.

3. Projekt realizacji interpretera

W projekcie wykorzystane zostaną generatory: analizatorów leksykalnych Flex oraz analizatorów składniowych Bison.

Analiza leksykalna:

Analizator leksykalny wykonuje następujące działania:

- pobiera kolejne znaki z wejścia programu;
- grupuje pobrane znaki w tokeny; rozróżniane atomy leksykalne są przedstawione w punkcie pierwszym „Konwencje leksykalne”;
- przekazuje analizatorowi składniowemu kolejne tokeny.

Tokeny przekazywane do analizatora składniowego będą składać się z par: typ symbolu oraz odpowiadająca wartość. Wartość odpowiadająca symbolowi jest przekazywana do parsera w postaci zmiennej yylval.

Dodatkowo przekazywana jest do parsera obecna pozycja w strumieniu wejściowym (w postaci zmiennej yylloc). Pozwala to na wytwarzanie przez parser informacji o błędach oraz pozycji tych błędów.

Analiza składniowa i semantyczna:

Przed wykonaniem wprowadzonego programu interpreter powinien sprawdzić cały tekst wejściowy pod względem składniowym oraz semantycznym. Dlatego parser LR na podstawie tokenów pobieranych od analizatora leksykalnego sprawdza poprawność konstrukcji składniowych wejściowego ciągu tokenów. Dodatkowo tworzona jest struktura drzewa wyprowadzenia dla wejściowego strumienia tokenów – drzewo to jest wykorzystywane w dalszej interpretacji programu. Węzły drzewa są zdefiniowane w folderze projektu `./environment/ast/`. Każdy węzeł reprezentujący *wyrażenie* lub *instrukcję* zapamiętuje informację o swojej pozycji – jest to wykorzystywane w trakcie interpretacji programu, w celu wyświetlenia czytelnych komunikatów, w którym miejscu popełniony został „Błąd wykonania” (błąd zauważony w momencie wykonywania programu).

Dokonywane jest również sprawdzenie, czy wprowadzone na wejście macierze spełniają warunek, że każdy wiersz (*lista_wyrażeń*) jest tej samej długości – poprzez zliczanie w odpowiednich węzłach liczby wyrażeń, jakich one dotyczą (już w trakcie interpretacji).

Kontrola typów jest dokonywana w momencie interpretacji programu.

Interpretacja:

Implementacja wykonania instrukcji dla drzewa wyprowadzanie programu znajduje się w folderze `./environment`.

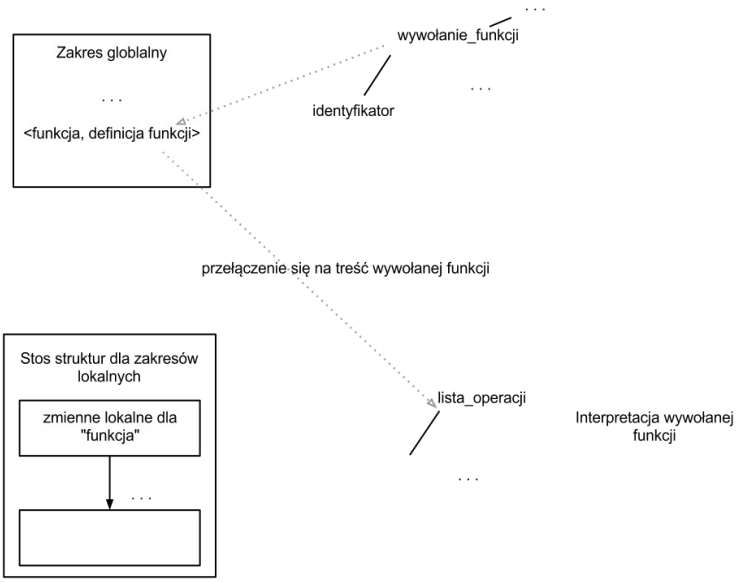
W trakcie interpretacji są wykorzystywane:

- struktura danych odpowiadająca zakresowi globalnemu: przechowuje typ, identyfikator, wartość zmiennych globalnych, oraz identyfikatory i definicje (odnośniki na korzeń drzewa wyprowadzania) poszczególnych funkcji;
- struktury danych odpowiadające zakresom lokalnym, przechowujące: typ, identyfikator, wartość zmiennych lokalnych.

Struktury dla zakresów lokalnych przechowywane są na stosie; wierzchołek stosu odpowiada obecnie interpretowanej funkcji (jej zakresowi).

Interpretacja programu polega na:

- przechodzeniu drzewa wyprowadzenia dla ciągu tokenów głównego kodu programu;
- w przypadku napotkania wywołania funkcji: nowym źródłem tokenów staje się definicja odpowiedniej funkcji. Na stosie umieszczona zostanie struktura dla nowego zakresu lokalnego („ramka stosu” dla danej funkcji), do której wstawione są również parametry funkcji (w tym parametr – wartość zwracana).



– każde kolejne wywołanie funkcji powoduje odłożenie na stosie kolejnej struktury. Napotkanie identyfikatora w ciągu instrukcji (np. w przypisaniu) powoduje przeszukiwanie struktury umieszczonej na szczycie stosu; jeżeli nie znaleziono wymaganego identyfikatora to przeglądany jest następnie zakres globalny. Zakończenie wykonania bloku instrukcji danej funkcji powoduje zdjęcie wierzchołka stosu.

4. Przykłady testowe

Przykład testowy nr 1 – kod główny programu, definicje funkcji oraz ich wywołania:

```
/* kod główny programu to każda instrukcja, która nie należy do żadnej definicji funkcji */

macierz = [1, 2;
           3, 4] * [ 5, 6;
                     7, 8 ];  /* instrukcja należąca do kodu głównego programu */
defn foo ( ) {
  macierz = [1, 0; 0, 1];          /* odwołanie się do zmiennej globalnej */

  /* aby można było utworzyć zmienną lokalną o takiej samej nazwie jak zmienna globalna należy
       użyć słowa "val" */

  val macierz = [0, 1;
                 1, 0];      /* zmienna lokalna z zakresu funkcji "foo", przysłaniająca zmienną
                                globalną macierz */
  foo = macierz;             /* każda funkcja musi coś zwracać – zwracana wartość zmiennej
                                lokalnej */
}

macierz = foo( );            /* kolejna instrukcja (po „macierz = [1, (…)” ) kodu głównego programu */
                             /* Wywołanie funkcji powoduje pobieranie ciągu tokenów
                                odpowiadających definicji funkcji foo */
```

Przykład testowy nr 2 – (pseudo) silnia macierzy:

```
/*
   Funkcja realizuje mnożenie macierzy zmniejszając kolejno wartości jej elementów
   aż któryś z elementów macierzy będzie == 1.
   @param macierz – macierz liczb 2x2
*/
defn silnia (macierz) {
   if (macierz | 0, 0 \ == 1 or          /* sekwencja | i, j \ umożliwia dostęp do elementów macierzy –  */
      macierz | 0, 1 \ == 1  or          /* –  jest odpowiednikiem [ i ] [ j ] z języka C */
      macierz | 1, 0 \ == 1  or
      macierz | 1, 1 \ == 1 )
         silnia = macierz;

   else silnia = silnia([macierz |0, 0\ - 1, macierz |0, 1\  - 1;
                         macierz |1, 0\ - 1, macierz |1, 1\ - 1] ) * macierz;
}

wynik = silnia ( [2, 3;
                  4, 5] );
/* Po wykonaniu programu: wynik = [1, 2; 3, 4] * [ 2, 3; 4, 5]  → wynik = [10, 13; 22, 29] */
```