Język wspierający operacje na listach

Jakub Zieliński 299041

Opis funkcjonalny

Głównym celem języka będzie ułatwienie pracy z listami. Wsparcie definiowania list oraz operacji modyfikujących zawartość listy przy pomocy operatorów. Dodatkowe operacje dotyczące list to sortowanie oraz filtracja listy (przy pomocy predykatu zdefiniowanego przez użytkownika) .

Elementy języka:

- język będzie wspierał podstawowe typy danych : int, float, string oraz typ specjalny dla języka, czyli list<type>. Typ string reprezentuje zmienne tekstowe w podwójnym cudzysłowie.
- Instrukcja warunkowa if else
- Pętla while
- funkcje definiowane przez użytkownika przy pomocy sygnatury opisanej w "gramatyce"
- Operacje wbudowane w język wspierające pracę z listami celujące w pracę z listami
  - filtrowanie przy pomocy predykatu : someList.filter(ele -> ele > 4)
  - usuwanie elementu na danej pozycji : someList.remove(index)
  - modyfikacja każdego elementu listy : someList.foreach(ele -> ele * 2)
  - dodanie elementu na koniec listy : someList.add(123)
- Operatory specjalne dla typu list:
  - konkatenacja list: operator '+'
  - dostęp indeksowy : operator []
- Operacje arytmetyczne na liczbach całkowitych oraz zmiennoprzecinkowych
- Operacje na stringach, mianowicie konkatenacja oraz dostęp indeksowy
- Standardowe wyjście przy pomocy wbudowanej funkcji print(value)

Zasady dotyczące zmiennych:

- typowane statycznie
- typowane silnie
- zmienne są mutowalne
- widoczne wewnątrz bloku ograniczonego nawisami `{` , `}`

Przykłady

```
int main(){

      int i = 1;
      int j = 2;
      print( i + j );

      return 0;
}
```

```
void main(int i, int j){

      print("hello world") ;

      int ij = i + j;
}
```

Punktem startowym dla interpretera jest fukcja o nazwie main, oprócze warunku na nazwę nie ma innchy wymagań co do sygnatury.

```
int fibb(int n){
      if ( n < 2)
            return 1;

      return fibb( n - 1) + fibb(n - 2);
}

void main(){

      int nthFib = 5;
      print(nthFib);
}
```

Języki będzie wspierał rekurenyjne wywołania zadeklarowanych funckji.

```
void main(int para){

      list<int> intList = [1 , 2 , 3 , 4];
      intList = intList.filter(ele -> ele > 3);
      intList = intList.foreach(ele -> ele * 2);
      intList.add(123);
      intList.add(para);
      int result = intList[0] + intList[1]
      print(result);
}
```

Wykorzystanie wbudowanych metod do manipulaowania listami. Inicializujemy listę, później zamieniamy listę przy pomocy filtra. Następnie każdy element listy mapujemy na jego podwojoną wartość. Dodajemy liczbę 12345 na koniec listy, a następnie wypisujemy zmienną result, wcześniej utworzoną po zsumowaniu elementów listy spod indeksu 0 i 1.

Gramatyka

```
program         ::== { function }
function        ::== signature parameters block
parameters      ::== `(` [ signature ] {`,` signature } ] `)`
arguments       ::== `(` [ expression {`,` expression } `)`
block           ::== `{` { instruction `;` | statement | block } `}`
signature       ::== type identifier
statement       ::== ifStatement | loopStatement
ifStatement     ::== `if` `(` condition `)` block [ `else` block ]
whileStatement ::== `while` `(` condition `)` block
instruction     ::== returnInst
                   | initInst
                   | assignInst
                   | functionCall
                   | listFuncCall
returnInst      ::== `return` expression
initInst        ::== signature [ assign ]
assignInst      ::== identifier assign
assign          ::== `=` expression
functionCall    ::== identifier arguments
listFuncCall    ::== identifier `.` listOpp
listOpp         ::== filter
                   | foreach
                   | remove
                   | add
filter          ::== `filter` `(` lambda `)`
predicate       ::== identifier `->` condition
foreach         ::== `foreach` `(` lambda )`
lambda          ::== identifier `->  expression
remove          ::== `remove` `(` expression `)`
add             ::== `add` `(` expression `)`
condition       ::== baseCond { condOpp baseCond }
baseCond        ::== expression compOpp expression
compOpp         ::== relationOpp |  equalOpp
listDef         ::== `[` [ expression { `,` expression } ] `]`
expression      ::== baseExp { mathOpp baseExp}
baseExp         ::== literal
                   | identifier [`[` expression `]`]
                   | `(` expression `)`
                   | functionCall
                   | listFuncCall
mathOpp         ::== addOpp | multOpp
literal         ::== `0` | [ `-` ] number | string | listDef
condOpp         ::== `&&` | `||`
equalOpp        ::== `==` | `!=`
relationOpp     ::== `<` | `<=` | `>` | `>=`
addOpp          ::== `+` | `-`
multOpp         ::== `*` | `/`| `%`
```

```
string          ::== `"` { ASCII character } `"`
number          ::== naturalDigit { digit } [ `.` digit { digit } ]
                   | digit [ `.` { digit } ]
identifier      ::== letter { letter | digit }
type            ::== stdType | listType
stdType         ::== `int` | `float` | `string`
funcRetType     ::== type | `void`
listType        ::== `list` `<` type `>`
letter          ::== `a` | `b` | ...  | `z`
                   | `A` | `B` | ...  | `Z`
digit           ::== `0` | posDigit
naturalDigit    ::== `1` | `2` | ...  | `9`
```

Opis techniczny realizacji

- Przy implementacji projektu zostanie wykorzystany język Java w wersji 11
- Struktura folderów będzie typowa jak dla projektu wspieranego przez narzędzie Maven
- IDE wykorzystane przy pracy nad projektem to Intellij IDEA

Pakiety

1. ***DataSource*** bedzie interfejsem zapewniającym ciąg znaków dla dalszego potoku przetwarznia.W zależności od uruchomienia interpretera źródłem może być ciąg z klawiatury lub plik tekstowy.
2. ***Lexer*** będzie miał za zadanie wyodrębnić z zadanego ciągu wejściowego wyrazy, a następnie zamienić je na tokeny, które zostaną przekazne do *Parsera.* W trakcie analizy tekstu *Lexer* ignoruje białe znaki oraz komentarze. Oprócz informacji typowych dla danej klasy tokenu, każda instancja będzie miała zapisane w sobie informacje o swoim położeniu w tekście (wiersz, kolumna), w przypadku trybu interaktywnego tokeny nie potrzebują tych informacji. Tokenizacja będzie leniwa.
3. ***Parser*** grupuje tokeny w drzewo składniowe i sprawdza, czy struktura jest poprawna składniowo. Tokeny zostaną zapisane do tabeli symboli wraz z informacją o typie. Poprawnie skonstruowane drzewo zostanie przekazane do I*nterpretera.*
4. ***Interpreter*** sprawdza semantykę konstrukcji stworzonych przez *Parser* oraz wykonuje zadane instrukcje.
5. ***ErrorHandler*** zajmie się się wypisanie w odpowiedni sposób informacji o błędach przychodzących z poszczególnych modułów. Na przykład błędów o niewłaściwej konstrukcji pochodzących z *Lexera,* lub błędów wynikających z niewłaściwego źródła danych podanego do *DataSource.*

Testowanie

1. DataSource : testy jednostkowe sprawdzające czy dane zostały poprawnie zczytane ze źródła.
2. Lexer : testy jednostkowe sprawdzające, czy zbiór tokenów utowrzony z zadanego ciągu wejściowego jest zgodny ze wcześniej przygotowanym (poprawnym) zbiorem tokenów. Również każdy typ tokenu powinien mieć swój oddzielny test jednostkowy.
3. Parser : sprawdzenie czy drzwo składniowe wygenerowane ze wcześniej przygotowanego zbioru tokenów jest poprawne. Dodatkowo można przygotować moduł wizualizujący wygenerowane drzewo w celu manualnej analizy