Arkadiusz Dawid
300199

TKOM - Projekt wstępny

Temat

Implementacja interpretera prostego języka programowania ogólnego przeznaczenia z niestandardowym typem danych - listą wraz z mechanizmem *list comprehension* znanym z pythona. Każdą listę można również odwrócić operatorem, np. *!list*.

Zasady działania języka

- Obsługa danych typu int, float oraz string.
- Poprawne wykonywanie działań matematycznych o różnym priorytecie, obsługa nawiasów.
- Proste operacje na stringach (konkatenacja, usunięcie ostatniego znaku).
- Znak ucieczki, pozwalający na umieszczenie innych znaków w stringach.
- Obsługa komentarzy jednoliniowych zaczynających się od znaku *#*.
- Praca ze zmiennymi - definiowanie, przypisywanie wartości, odczyt.
    - Typowanie statyczne.
    - Typowanie silne.
    - Zmienne mutowalne.
    - Widoczność zmiennych - wewnątrz bloku ograniczonego nawiasami klamrowymi.
- Instrukcja warunkowa *if (warunek)*.
- Instrukcja pętli *while (warunek)*.
- Praca z funkcjami - definiowanie, wołanie, rekursja.
- Dodatkowy typ danych - lista, która może zawierać wiele wartości int, float, string, ale zawsze tych samych, bez mieszania. Mechanizm *list comprehension*. Operator odwracania listy. Dodatkowe funkcje jak *max(lista)* i *min(lista)* zwracające wartości max i min z listy (tylko dla list złożonych z intów lub floatów. Operator podłogi zwracający długość listy lub stringa, np. *_lista* lub *_string*. Tylko listy jednowymiarowe.

Przykłady wykorzystania języka

Przykład 1: Podaj n-tą wartość ciągu Fibonacciego.

```
int fibonacci(int n) {
    if (n <= 2) {
        return 1;
    }
    int prev = 1;
    int result = 1;
    int i = 0;
    while (i < n-2) {
        result = result + prev;
        prev = result - prev;
        i = i + 1;
    }
    return result;
}
```

Definicja funkcji wraz z argumentem, instrukcja warunkowa, zmienne lokalne, instrukcja pętli, modyfikowanie wartości zmiennych, zwrócenie wartości przez funkcję.

Przykład 2: Mając do dyspozycji listę dodatnich liczb całkowitych, zwróć odwróconą listę z każdym elementem podniesionym do kwadratu.

```
int[] squareList(int[] nums) {
    int[] result = [n*n for n in nums];
    return !result;
}
```

Wykorzystanie mechanizmu *list comprehension*. Definicja zmiennej w postaci listy intów, definicja funkcji, która zwraca taką listę, odwrócenie listy operatorem.

Przykład 3: Stwórz dwie listy stringów i połącz je ze sobą. Na podstawie wynikowej listy stwórz drugą listę, której kolejne wartości będą odpowiadały długości kolejnych stringów z pierwszej listy. Zwróć tę listę. Wykorzystaj mechanizm *list comprehension*.

```
int[] lengthList() {
    string[] a = ["this", "is"];
    string[] b = ["example", "text"];
    string[] c = a + b;
    int[] result = [_word for word in c];
    return result;
}
```

Definicja listy wprost, przez złączenie dwóch już istniejących list oraz przy użyciu mechanizmu *list comprehension*.

To samo, tylko krócej:

```
int[] lengthList() {
    string[] a = ["this", "is"];
    string[] b = ["example", "text"];
    return [_word for word in a+b];
}
```

Bardziej złożone wyrażenie w returnie.

Struktura projektu

1. Źródłem danych może być plik lub ciąg znaków wpisywanych kolejno w konsoli.
2. Analizator leksykalny wyodrębnia kolejne znaki z kodu źródłowego i grupuje je w tokeny, które następnie przekazuje do analizatora składniowego (parsera). Automatycznie usuwa białe znaki i komentarze. Oprócz tego śledzi numer linii w kodzie, żeby móc poinformować użytkownika o miejscu ewentualnego błędu. Leniwa tokenizacja. Testy jednostkowe będą polegały na podaniu kodu wejściowego, co do którego znany jest oczekiwany zbiór tokenów i porównanie z wynikiem wygenerowanym przez parser. Każdy token będzie miał odpowiadający mu test jednostkowy.
3. Analizator składniowy (parser) zstępujący grupuje tokeny w drzewa składniowe i sprawdza, czy kod jest poprawny składniowo. Zapisuje tokeny do tablicy symboli i uzupełnia ją o typy danych tokenów. Drzewa składniowe przekazuje do analizatora semantycznego. Testy jednostkowe będą polegały na podaniu analizatorowi zbioru tokenów i sprawdzeniu, czy wygenerował oczekiwane drzewo składniowe.
4. Analizator semantyczny sprawdza, czy dane wyrażenie jest poprawne pod względem semantycznym, czy typy tokenów zgadzają się w wyrażeniu.

Błędy krytyczne, np. niewłaściwe odwołanie do pamięci spowodują zakończenie programu. Błędy niekrytyczne, np. literówki w identyfikatorach, brakujące nawiasy, źle umiejscowione średniki czy niewłaściwe dopasowanie typów danych w kodzie nie spowodują zakończenia programu, a po skończeniu analizy kodu na standardowe wyjście zostanie wypisany komunikat zawierający informacje o błędach.

Gramatyka

```
program             ::==  { function }
function            ::==  signature parameters block
parameters          ::==  '(' [ signature { ',' signature } ] ')'
arguments           ::==  '(' [ expression { ',' expression } ] ')'
block               ::==  '{' { instruction ';' | statement | block } '}'
signature           ::==  type id
statement           ::==  ifStm
                        |  whileStm
instruction         ::==  returnInstr
                        |  initInstr
                        |  assignInstr
                        |  functionCall
ifStm               ::==  'if' '(' condition ')' block [ 'else' block ]
whileStm            ::==  'while' '(' condition ')' block
returnInstr         ::==  'return' expression
initInstr           ::==  signature [ assignable ]
functionCall        ::==  id arguments
assignInstr         ::==  id assignable
assignable          ::==  '=' ( expression | listDef | listComprehension )
listDef             ::==  '[' [ expression { ',' expression } ] ']'
listComprehension   ::==  '[' expression 'for' id 'in' expression ']'
expression          ::==  multExpr { addOp multExpr }
multExpr            ::==  primaryExpr { multOp primaryExpr }
primaryExpr         ::==  literal
                        |  id [ '[' expression ']' ]
                        |  '(' expression ')'
                        |  functionCall
condition           ::==  andCond { '||' andCond }
andCond             ::==  equalityCond { '&&' equalityCond }
equalityCond        ::==  relationalCond { equalOp relationalCond }
relationalCond      ::==  primaryCond [ relationOp primaryCond ]
primaryCond         ::==  [ '!' ] ( '(' condition ')' | expression )
equalOp             ::==  '==' | '!='
relationOp          ::==  '<' | '<=' | '>' | '>'
addOp               ::==  '+' | '-'
multOp              ::==  '*' | '/' | '%'
literal             ::==  '0'
                        |  [ '-' ] number
                        |  string
                        |  listDef
                        |  listComprehension
string              ::==  '"' { any ASCII character } '"'
number              ::==  naturalDigit { digit } [ '.' digit { digit } ]
                        |  digit [ '.' digit { digit } ]
id                  ::==  letter { digit | letter }
digit               ::==  '0' | naturalDigit
naturalDigit        ::==  '1' | '2' | ... | '9'
letter              ::==  'a' | 'b' | ... | 'z'
                        |  'A' | 'B' | ... | 'Z'
                        |  '_'
type                ::==  standardType
                        |  listType
listType            ::==  standardType '[]'
standardType        ::==  'int'
                        |  'float'
                        |  'string'
```