ARKADIUSZ DAWID
300199

Techniki Kompilacji

DOKUMENTACJA KOŃCOWA

Temat

Implementacja interpretera prostego języka programowania bazującego na C z niestandardowym typem danych - listą wraz z mechanizmem *list comprehension* znanym z pythona.

Cechy języka

- Obsługiwane typy danych: int, float, string.
- Obsługiwane listowe typy danych: lint (lista intów), lfloat (lista floatów), lstring (lista stringów).
- Implementacja mechanizmu *list comprehension* znanego z pythona.
- Zmienne mutowalne, silnie typowane, widoczne tylko w danym bloku ograniczonym przez nawiasy klamrowe.
- Instrukcje if-else oraz while.
- Poprawne wykonywanie działań matematycznych o różnym priorytecie, obsługa nawiasów
- Obsługa komentarzy jednoliniowych zaczynających się od znaku *#*.
- Znak ucieczki pozwalający na wstawienie do stringów takich znaków jak znak nowej linii *n*, tabulacji *t* czy cudzysłowu \".
- Niestandardowe operatory:
  - ![lista] - zwraca odwróconą listę,
  - _[lista] - zwraca długość listy.
- Kod musi być zawarty w funkcji *main*.
- Funkcja *main* może zwracać dowolny typ danych (int, float, string, lint, lfloat, lstring).
- Wartość zwracana przez funkcję *main* zostaje wypisana na standardowe wyjście.

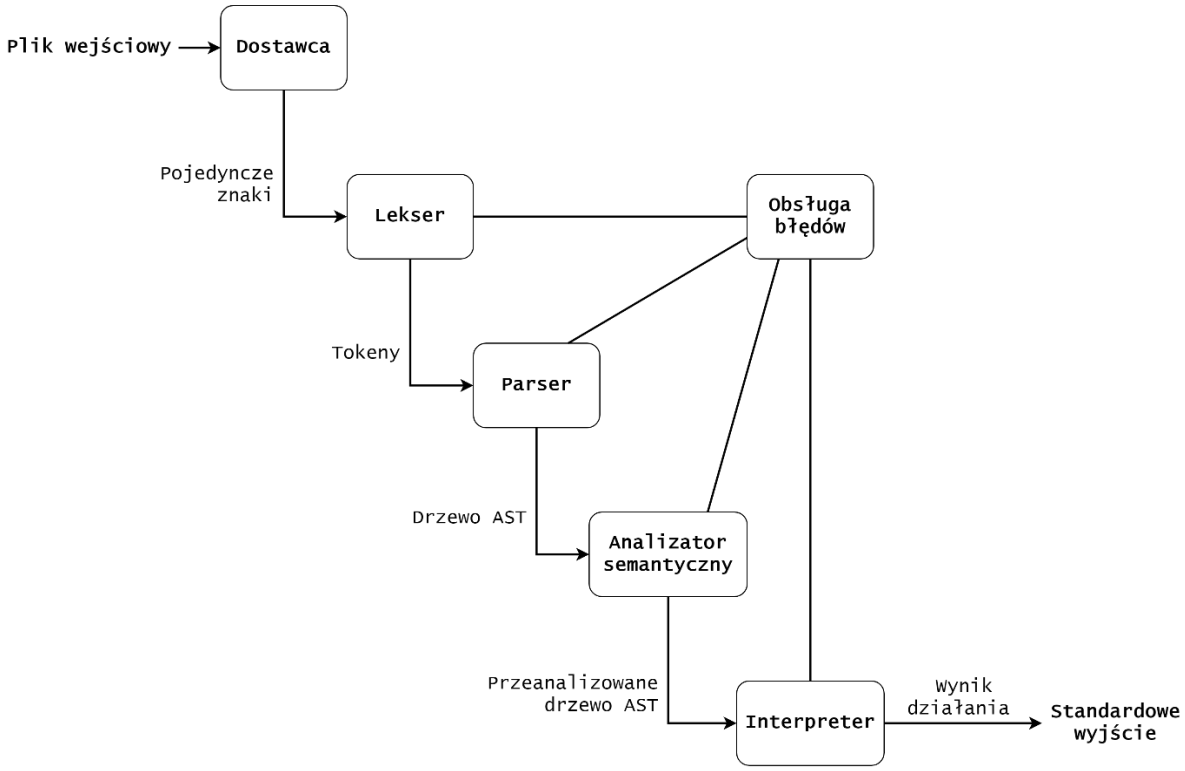
Implementacja

Opis

Implementacja całego systemu została wykonana w języku Python3 w IDE PyCharm. System jest podzielony na 5 głównych modułów, które łącznie tworzą potok przetwarzania:

1. Dostawca odpowiada za dostarczanie pojedynczych znaków kodu wejściowego do Leksera. Dodatkowo śledzi aktualne miejsce w kodzie (nr linii i kolumny).
2. Lekser z dostarczonych znaków buduje Tokeny. Pomija białe znaki i komentarze. Zgłasza błędy leksykalne, np. liczba z literami wśród cyfr albo string bez zamykającego cudzysłowu.
3. Parser przyjmuje Tokeny i buduje z nich węzły drzewa składniowego. Zgłasza błędy składniowe, np. deklaracja zmiennej o zakazanej nazwie, brakujące średniki, nawiasy, itp.
4. Analizator semantyczny przyjmuje drzewo składniowe i sprawdza poprawność programu na poziomie znaczenia poszczególnych instrukcji oraz programu jako całości. Wykrywa takie błędy jak np. próba redeklaracji zmiennej, próba przypisania wartości o niewłaściwym typie do zmiennej, użycie niezadeklarowanej wcześniej zmiennej, brak wyrażenia *return* w ciele funkcji, itp. Jeśli nie ma żadnych błędów, analizator semantyczny jest przezroczysty, tzn. zwraca dokładnie to samo drzewo składniowe, które przyjął.
5. Interpreter przyjmuje przeanalizowane drzewo składniowe i wykonuje kolejne instrukcje. Na bieżąco aktualizuje słownik zadeklarowanych zmiennych wraz z ich aktualnymi wartościami. Gdy natrafi na wyrażenie *return* w funkcji *main*, przerywa działanie i wypisuje zwracaną wartość na standardowe wyjście. Interpreter zgłasza błędy tylko w dwóch przypadkach - przy dzieleniu przez zero oraz przy próbie dostępu do elementu listy poza jej zakresem.

Potok przetwarzania



Testowanie

Na potrzeby testowania kodu została wykorzystana biblioteka *unittest*. Wszystkie testy znajdują się w folderze *tests*. Łącznie napisano 141 testów jednostkowych z czego 55 testów leksera, 72 testy parsera, 9 testów analizatora semantycznego i 5 testów interpretera.

Przykłady wykorzystania języka

Przykład 1:

```
int main() {
    int n = 8;
    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;
}
```

Program zwraca n-tą wartość ciągu Fibonacciego. Sam kod w tym przykładzie niczym nie różni się od zwykłego C, ale pokazuje podstawowe rzeczy, które język potrafi zrobić - zadeklarować zmienną, nadać jej inną wartość, wykonać pewne instrukcje warunkowo i w pętli, zwrócić końcową wartość.

Wartość zwracana: 21.

Przykład 2

```
lint main() {
    lint nums = [1, 2, 3, 4, 5];
    lint nums2 = [6, 7, 8];

    nums = nums + nums2;

    lint res = [n*n for n in nums];
    return !res;
}
```

Program na podstawie listy liczb naturalnych tworzy listę ich kwadratów, a następnie zwraca tę listę w odwrotnej kolejności. W tym przykładzie widzimy już elementy odróżniające język od C: definicja zmiennej listowej, wykorzystanie mechanizmu *list comprehension* oraz odwrócenie listy niestandardowym operatorem wykrzyknika.

Wartość zwracana: [64, 49, 36, 25, 16, 9, 4, 1].

Przykład 3

```
int main(){
    lstring a = ["Hello","world"];
    lstring b = [s + " " for s in a];

    b = a + b;

    return _b;
}
```

Program tworzy dwie listy stringów - pierwsza jest zadeklarowana wprost, a druga z wykorzystaniem *list comprehension*. Następnie łączy te dwie listy i zwraca długość wynikowej listy z wykorzystaniem niestandardowego operatora podłogi.

Wartość zwracana: 4.

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            ::=   'l' standardType
standardType        ::=   'int'
                        |   'float'
                        |   'string'
```

Uwagi

1. Ze względu na pewne trudności i ograniczenia czasowe nie zaimplementowano deklaracji i wywoływania własnych funkcji wewnątrz funkcji *main*. Dodatkowo w parserze i analizatorze semantycznym występują niewielkie błędy, które w pewnych sytuacjach zgłaszają błąd mimo poprawnego kodu wejściowego.
2. W folderze *examples* zostały umieszczone pliki o rozszerzeniu .cl (skrót od C with List) z przykładowymi kodami wejściowymi. Aby je uruchomić, wystarczy podać ścieżkę do pliku w argumencie wywołania funkcji *main* głównego programu, np. *examples/example1.cl*.