Politechnika Warszawska, WEiTI
Techniki kompilacji
Dokumentacja Końcowa

Interpreter własnego języka obsługującego Rzymski system zapisywania liczb

Autor: Halyna Polekha (nr alb. 294866)

Oświadczam, że niniejsza praca stanowiąca podstawę do uznania osiągnięcia efektów uczenia się z przedmiotu Techniki kompilacji została wykonana przeze mnie samodzielnie.

1 Opis projektu

Ideą projektu jest stworzenie interpretera własnego prostego języka, wyposażonego w proste instrukcje sterujące (instrukcja warunkowa i pętli), możliwość definiowania oraz wywołania własnych funkcji oraz wbudowany typ liczby rzymskie.
Przyjęte zostały następujące zasady:

- Język własny jest imperatywnym językiem typowanym dynamicznie.
- Istnienie wbudowanego typu liczbowego całkowitego i rzeczywistego oraz możliwość wykonywania na nich operacji matematycznych (dodawanie, odejmowanie, mnożenie, dzielenie)
- Jednym z typów wbudowanych jest typ liczb rzymskich w zakresie od 1 do 9999
- Liczby o większym zakresie są przekształcane do typu int
- Istnienie liczb rzymskich ujemnych (poprzedzenie liczby znakiem -)
- Istnienie możliwości wykonania operacji matematycznych na danych typu liczby rzymskie (dodawanie, odejmowanie, mnożenie, dzielenie)
- Wynik operacji arytmetycznych łączących typy danych liczby rzymskie i liczby całkowite będzie typem danych liczby całkowite
- Wynik operacji arytmetycznych łączących typy danych liczby rzymskie i liczby rzeczywiste będzie typem danych liczby rzeczywiste
- Istnienie wbudowanego typu znakowego, możliwość wykonania operacji konkatenacji oraz możliwość porównania danych tego typu

2 Przykład

Poniżej przedstawiono przykład użycia języka w realizacji operacji potęgowania.

```
(fun power_of real (real base) (int exponent) {
      (real result)
      (= result 1)
        (loop (!= exponent 0) {
               (= result (* result base))
               (= exponent (- exponent 1))
        })
      (ret result)
})
(power_of 3.4 5)
```

3 Krótki kurs języka

Zaimplementowany język jest językiem skryptowym, który umożliwia zdefiniowanie funkcji w jednym pliku lub całkowite unikanie funkcji i stosowanie imperatywnego stylu programowania.
Zgodnie z gramatyką języka poszczególne instrukcje są zawsze ujęte w nawiasy okrągłe, np:

```
(int a)
```

Ważne jest to że wyrażenia muszą być zapisywane z wykorzystaniem notacji polskiej, np

- Prawidłowy zapis

  ```
  (+ a b)
  ```

- Nieprawidłowy zapis

  ```
  (a + b)
  ```

3.1 Typy danych

W języku zaimplementowane są następujące typy danych:

- text (maksymalna długość 4294967295 bajtów)
  - Tekstowy typ danych służący do przechowywania ciągu znaków (zmiennych łańcuchowych)
- int (maksymalna wielkość 2147483647)
  - Typ całkowitoliczbowy
- real (maksymalna precyzyjność 15 znaków)
  - Typ zmiennoprzecinkowy
- rom (maksymalna wielkość 9999]
  - Typ reprezentujący liczbę rzymską

Obsługa wartości logicznych jest realizowana za pomocą typu całkowitoliczbowego na takiej zasadzie, że

- *fałsz* jest reprezentowana jako liczba 0
- *prawda* jest reprezentowana jako liczba nieujemna, domyślnie 1.

W przypadku wartości tekstowych jests brana pod uwagę jej liczba znaków.

Język także realizuje automatyczne rzutowanie typów liczbowych do typów o większym zakresie w trakcie przypisania wartości liczbowej do zmiennej typu liczbowego.

3.2 Zmienne

Nazwa zmiennej może zawierać litery, cyfry oraz znak podkreślenia. Musi zaczynać się z litery niebędącej cyfrą rzymską

- Prawidłowe nazwy zmiennych: a, b_, ab_13
- Nieprawidłowe nazwy zmiennych: 1a, _b, x, i, ,v
    - Uwaga: ciągi znaków takie jak i, x, v, xi itd są odczytywane jako dane typu liczby rzymskie.

3.3 Deklaracja zmiennych

Deklaracja zmiennych jest instrukcją składającą się z dwóch elementów: typ definiowanej zmiennej oraz jej nazwa, np.

- Definicja zmiennej o nazwie *a* i typie *int*

  ```
  (int a)
  ```

- Definicja zmiennej o nazwie *b* i typie *rom*

  ```
  (rom b)
  ```

3.4 Operatory

3.4.1 Operator przypisania

Operatorem przypisania jest operator = .
Przykładem przypisania wartości całkowitoliczbowej zmiennej *a* jest instrukcja

```
(= a 1)
```

3.4.2 Operatory arytmetyczne

Operatorami arytmetycznymi są operatory dwuargumentowe:

- +
    - Operator dodawania
- -
    - Operator odejmowania
- /
    - Operator dzielenia

- *
    - Operator mnożenia

Zgodnie z notacją polską operatory arytmetyczne poprzedzają swoje argumenty.
Przykłady wykorzystania operatorów arytmetycznych:

- Realizacja operacji arytmetycznej (a + b)/2 jest możliwa za pomocą instrukcji

```
(/(+ a b) 2)
```

- Realizacja operacji arytmetycznej 2 * (VI - 2) jest możliwa za pomocą instrukcji

```
(* 2 (- VI 2))
```

3.4.3 Operatory relacji

Operatorami relacji są operatory:

- Dwuargumentowe
    - >
        - Operator nierówności ostrej większe
    - <
        - Operator nierówności ostrej mniejsze
    - >=
        - Operator nierówności ostrej większe lub równe
    - <=
        - Operator nierówności ostrej mniejsze lub równe
    - ==
        - Operator równości
    - !=
        - Operator nierówności
- Jednoargumentowe
    - !
        - Operator negacji

Operatory relacji zawsze zwracają wartości całkowite 1 lub 0 odpowiadające wartościom logicznym prawda lub fałsz.
Przykłady użycia operatorów relacji:

```
(> a b)
(!VI)
(= a (== b X))
```

3.4.4 Operator konkatenacji

Operator konkatenacji dotyczy typu łańcuchowego txt
Przykład wykorzystania operatora konkatenacji:

```
(. "abc" "def")
```

3.5 Instrukcje read i write

Używamy instrukcji *write* do wysyłania danych do standardowego urządzenia wyjściowego (ekranu).
Używamy instrukcji *read* do wczytywania danych ze standardowego wejścia.
Przykłady:

```
(read a)
(write "Hello world!")
```

3.6 Instrukcje warunkowe

Instrukcje warunkowe zdefiniowane w przedstawionym języku:

- (if (<*warunek*>) {<*właściwa instrukcja blokowa*>})
- (ifel (<*warunek*>) {<*właściwa instrukcja blokowa*>} {<*alternatywna instrukcja blokowa*>})

Warunkiem może posłużyć zarówno operacja porównywania realizowana za pomocą relacji jaki wartość zmiennej oraz wartość liczbowa lub tekstowa
Przykładami użycia instrukcji warunkowych są

```
(if (“abc”) { (* b 10) })
(ifel (a > 0) { (= c (+ c V)) } { (= c 0) (write "Error") })
```

3.7 Instrukcja pętli

Instrukcja pętli jest zdefiniowana jako

- (loop (<*warunek*>) { <*właściwa instrukcja blokowa*>})

Warunkiem może posłużyć zarówno operacja porównywania realizowana za pomocą relacji jaki wartość zmiennej oraz wartość liczbowa lub tekstowa
Przykłady:

```
(loop (!= k 1) { (write "Not equal.") })
```

3.8 Funkcje

W podanym języku jest realizowane grupowanie sekwencji instrukcji w osobne funkcje.
Definicja funkcji jest możliwa w taki sposób:

- (fun *<nazwa funkcji> <typ zwracany> <argumenty>* {*<instrukcja bloku posiadająca instrukcje wyjścia z funkcji ret>*})

Przyzkład:

```
(fun koncatenate txt (txt a) (txt b) { (ret (. a b)) })
```

Wywołanie funkcji:

- (*<nazwa funkcji> <parametry funkcji>*)

Przykład:

```
(= a (concatenate "abc" "def"))
```

4 Przykłady obrazujące dopuszczalne konstrukcje językowe

Przykład 1: Program do wyliczania członów sekwencji Fibonacciego

```
(fun fib int (int n) {
     (if (<= n 1) {
        (ret n)
     })
     (ret (+(fib (- n 1) ) (fib (- n 2)))
)})

(write (fib 7))
```

Przykład 2: Program do odczytania nazwiska ze standardowego strumienia wejścia, konkatenacje z imieniem podanym jako parametr oraz spacją

```
(fun read_conc txt (txt name) {
      (txt surname)
      (read surname)
      (txt t)
      (= t (. (. name ' ') surname))
      (ret t)
})

(write (read_conc 'Anna'))
```

Przykład 3: Program który cztery razy wywoła funkcję z Przykładu 2 oraz wypisze zwrócone przez nią dane na standardowy strumień wyjścia

```
(int kount)
(= kount I)
```

```
(loop (<= kount IV) {
      (write (read_conc 'Anna'))
      (= kount (+ kount 1))
})
```

Przykład 4: Liczby rzymskie (operacje arytmetyczne)

```
(write "XLI: ")
(write (+ V (* XVIII (- III I))))
```

Przykład 5: Kod niepoprawny. Wygenerowany zostanie odpowiedni komunikat o błędzie

Przykład 5.1: Każda poszczególna instrukcja musi być ujęta w nawiasy

```
(write 'Hello'
```

Przykład 5.2: Błąd - liczba rzymska nie istnieje

```
(rom romanNumber)
(= romanNumber VX)
```

Przykład 5.3: Wyrażenia muszą być zapisywane z wykorzystaniem notacji polskiej.

```
(int a)
(real b)
(+ a b)
```

5 Formalna specyfikacja i składnia

5.1 Specyfikacja języka

- Poszczególne wyrażenia (instrukcji oraz wywołania) są ujęte w nawiasy
- Oprócz nawiasów separatorami są też znaki białe takie jak znak spacji oraz znak nowej linii
- Wyrażenia są zapisywane z wykorzystaniem notacji polskiej.

5.2 Wymagania funkcjonalne

- Odczytywanie, parsowanie i analiza plików źródłowych
- Kontrola poprawności wprowadzonych danych oraz odpowiednie zgłaszanie błędów wykrytych podczas kolejnych etapów analizy plików
- Poprawne wykonywanie wszystkich poprawnie zapisanych instrukcji w plikach źródłowych
- Możliwość definiowania własnych funkcji oraz ich późniejszego wywołania
- Możliwość tworzenia, definiowania oraz używania zmiennych
- Przestrzeganie logicznego porządku instrukcji sterujących
- Przeprowadzenie operacji arytmetycznych i logicznych
- Wyświetlanie wyników wykonania na standardowym wyjściu terminala

5.3 Wymagania niefunkcjonalne

- Proste i przejrzyste komunikaty o błędach analizy kodu wejściowego

5.4 Gramatyka języka

program ={ funDefStatement | statement }

funDefStatement = ‘(‘, ‘fun’, funName, dataType|’nil’, {arg}, blockFun, ‘)’;
funName = correctName;
correctName = letter, {symbol};
letter = “a” .. “z” | “A” .. “Z”;
symbol = digit | letter | “_”;
text = ‘“‘, { *any_char* }, ‘“‘;
dataType = ‘int’ | ‘real’ | ‘text’ | ‘rom’;
arg = ‘(‘, dataType, variableName, ‘)’;
blockFun = ‘{‘, { statement | retStatement } , ‘}’;
statement = conditionalStatement | loopStatement | communicateStatement | arithmeticStatement | funCallStatement | varCreateStatement | varAssignmentStatement | concatStatement, conditionStatement;
conditionalStatement = ifStatement | ifElStatement;
ifStatement =’(‘, ‘if’, condStatement, block, ‘)’;
ifElStatement = ’(‘, ‘ifel’, conditionStatement, block, block, ‘)’;
loopStatement = ’(‘, ‘loop’, conditionStatement’, block, ‘)’;
block = ‘{‘, { statement }, ‘}’;
communicateStatement = readStatement | writeStatement;
readStatement = ‘(‘, ‘read’, variableName, ‘)’;
writeStatement = ‘(‘, ‘write’, variableName|text|funCallStatement|arithmeticStatement|concatStatement|conditionStatement, ‘)’;
varCreateStatement = ‘(‘, dataType, variableName, ‘)’;

retStatement = ‘(‘, ‘ret’ + separator + variableName | literal | ‘nil’ | funCallStatement|arithmeticStatement|concatStatement|conditionStatement, ‘)’;

literal = number | realNumber | text | romanNumber;
funCallStatement = ‘(‘, funName, {argForFunCalling} , ‘)’;
argForFunCalling = literal | variableName | funCallStatement|arithmeticStatement|concatStatement|conditionStatement

arithmeticStatement = ‘(‘, operation, arithmeticArg, arithmeticArg, ‘)’
operation = ‘+’ | ‘-’ | ‘*’ | ‘/’;
arithmeticArg = number | realNumber | romanNumber variableName|funCallStatement|arithmeticStatement|conditionStatement

varAssignmentStatement = ‘(‘, ‘=’, literal | variableName | funCallStatement | arithmeticStatement|concatStatement|conditionStatement, ‘)’;
concatStatement = ‘(‘, ‘.’, text | variableName | funCallStatement|concatStatement, text | variableName | funCallStatement|concatStatement, ‘)’;

condExpression = (condBOperator, condArg, condArg)
| (condUOperator, condArg) | (‘or’|’and’, condArg, condArg) | literal | variableName | funCallStatement | arithmeticStatement | conditionStatement

condBOperator = ‘<’ | ‘>’ | ‘<=’ | ‘>=’ | ‘==’ | ‘!=’
condUOperator = ‘!’

condArg = literal | variableName | funCallStatement | arithmeticStatement | conditionStatement’

romanNumber =[“-”], romanDigit , {romanDigit }
romanDigit = ‘I’, ‘V’, ‘X’, ‘L’, ‘D’, ‘M’, ‘C’

zeroDigit = “0”;
digitWithoutZero = “1” .. “9”;
digit = zeroDigit | digitWithoutZero;
positiveNumber = digitWithoutZero, {digit};
negativeNumber = “-”, positiveNumber;
number = zeroDigit | positiveNumber | negativeNumber;

realNumber = number, ‘.’, digit, { digit };

variableName = correctName;

6 Informacje techniczne

6.1 Środowisko

Projekt jest realizowany w języku Python w wersji 3.6. Do projektu został dołączony plik z wymaganymi do zainstalowania bibliotekami.

6.2 Obsługa programu

Program jest prostą aplikacją konsolową uruchamianą poprzez wywołanie wraz z parametrem uruchomieniowym reprezentującym ścieżkę do pliku źródłowego.

Wynik poszczególnych etapow analizy pliku oraz samego wyniku interpretacji końcowej i wykonania jest wyświetlany na standardowym wyjściu. W zależności od wyniku analizy, na standardowe wyjście mogą być zgłaszane: błędy analizy pliku, błędy składniowe, błędy semantyczne lub wynik wykonania (wraz z możliwymi błędami czasu wykonania). Aplikacja konsolowa nie przewiduje zapisywania wyników do pliku (można to zrobić przekierowując wyjście bezpośrednio do pliku).

Do projektu został dołączony plik requirement.txt z wymaganymi do zainstalowania bibliotekami.
Przykładowa instalacja:

```
python3.6 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

6.2.1 Sposób uruchomienia programu

Po zainstalowaniu potrzebnych bibliotek:

```
python3.6 main.py <plik wejściowy z kodem do zinterpretowania>
```

6.2.2 Sposób testowania programu:

- `python3.6 lexer_tests.py`
- `python3.6 parser_tests.py`
- `python3.6 interpreter_tests.py`

Odpowiednio dla testowania leksera, parsera, interpretera.

6.3 Struktura projektu:

- plik lexer.py - implementacja leksera
- plik parser.py - implementacja parsera
- plik interpreter.py - implementacja interpretera

- pliki statement.py, elements.py - obiekty przy pomocy których parser tworzy hierarchię
- pliki enviroment.py, env_elements.py - implementacja klas pomocniczych dla interpretera
- pliki inode.py, inodevisitor.py - klasy pomocnicze dla implementacji interpretera
- pliki error.py, errors.py - implementacja obsługi błedów
- pliki lexer_tests.py, parser_tests.py, interpreter_tests.py - implementacja testów jednostkowych
- plik limits.py -wspólne dla wielu klas zmienne
- plik my_token.py - implementacja tokena
- plik program.py - reprezentacja obiektowa programu
- plik Source.py - implementacja interfejsów wejściowych dla programu
- plik main.py - punkt wejściaa