Techniki kompilacji

Język interpretowany do tworzenia skryptów testujących systemy OCR

# Opis projektu

Inspiracją dla stworzenia języka jest temat pracy inżynierskiej – stworzenie aplikacji tłumaczącej teksty odczytane z obrazu za pomocą OCR. Tworzony język miałby zapewnić wsparcie dla testowania mechanizmów OCR poprzez umożliwienie operacji na ciągach znaków, liczbach oraz obrazach i zestaw funkcji wbudowanych umożliwiających zbadanie jakości rezultatów działania OCR.

# Specyfikacja

Program w proponowanym języku składać się będzie z definicji funkcji, a definicje z instrukcji do wykonania. Dopuszczalne instrukcje to deklaracja zmiennych, przypisanie do zmiennej, instrukcja warunkowa, pętla while, instrukcja return oraz wywołanie funkcji. Zmienne występujące w skrypcie mogą mieć typ: Image, Text, Integer i Boolean. Funkcje mogą zwracać wszystkie typy zmiennych oraz Void – wtedy nie określa się w definicji wartości zwracanej.

## Opis gramatyki

script = (functionDefinition)\* ;  
functionDefinition = TYPE ID parameters instructionSet;  
parameters = OPEN\_ROUND\_BRACKETS [ TYPE ID {COMMA TYPE ID} ] CLOSE\_ROUND\_BRACKETS;  
instructionSet = OPEN\_CURLY\_BRACKETS {statement} CLOSE\_CURLY\_BRACKETS;  
statement = ifStatement | whileStatement | returnStatement SEMICOLON  
 | declarationStatement SEMICOLON | assignmentStatement SEMICOLON  
 | functionInvocation SEMICOLON;  
ifStatement = IF condition instructionSet [ { ELSEIF condition instructionSet } ELSE instructionSet] ;  
whileStatement = WHILE condition instructionSet;  
returnStatement = RETURN value;  
declarationStatement = TYPE ID [ ASSIGNMENT\_OP value ];  
assignmentStatement = ID ASSIGNMENT\_OP value;  
functionInvocation = ID arguments;  
arguments = OPEN\_ROUND\_BRACKETS [ value { COMMA value } ] CLOSE\_ROUND\_BRACKETS;

numericOperation = OPEN\_ROUND\_BRACKETS sum CLOSE\_ROUND\_BRACKETS;  
sum = multiplication { ( PLUS\_OP | MINUS\_OP ) multiplication };  
multiplication = singleOperation { ( MULTIPLICATION\_OP | DIVISION\_OP ) singleOperation };  
singleOperation = [ MINUS\_OP ] ( numericOperation | functionInvocation | idAtom | integerAtom   
 | textAtom);

condition = OPEN\_ROUND\_BRACKETS alternative CLOSE\_ROUND\_BRACKETS;  
alternative = cojunction { OR\_OP cojunction };  
cojunction = cojunctionOperand { AND\_OP cojunctionOperand };  
cojunctionOperand = comparison | singleCondition;

comparison = ( atom | functionInvocation | condition | sum ) COMPARISON\_OP ( atom   
 | functionInvocation | condition | sum );  
singleCondition = [NEGATION\_OP ] ( condition | functionInvocation | booleanAtom | idAtom);

value = atom | functionInvocation| sum | alternative;  
atom = integerAtom | booleanAtom | idAtom | textAtom;  
  
integerAtom = INTEGER\_VARIABLE;  
booleanAtom = TRUE | FALSE;  
textAtom = TEXT\_VARIABLE;  
idAtom = ID;

### Tokeny przetwarzane przez lexer

TYPE = 'Text' | 'Image' | 'Integer' | 'Void' | 'Boolean' ;  
OPEN\_ROUND\_BRACKETS = '(';  
CLOSE\_ROUND\_BRACKETS = ')';  
OPEN\_CURLY\_BRACKETS = '{';  
CLOSE\_CURLY\_BRACKETS = '}';  
SEMICOLON = ';';  
IF = 'if';  
ELSE = 'else';  
ELSEIF = 'elseif';  
WHILE = 'while';  
BREAK = 'break';  
CONTINUE = 'continue';  
RETURN = 'return';  
ASSIGNMENT\_OP = '=';  
OR\_OP = '||';  
AND\_OP = '&&';  
COMPARISON\_OP = '==' | '!=' | '<' | '>' | '<=' | '>=' ;  
NEGATION\_OP = '!';  
PLUS\_OP = '+';  
MINUS\_OP = '-';  
MULTIPLICATION\_OP = '\*';  
DIVISION\_OP = '/';  
QUOTATION = '"';  
ID = [a-zA-Z] {[a-zA-Z0-9]} ;  
INTEGER\_VARIABLE = [0-9] | [1-9] {[0-9]};  
TEXT\_VARIABLE : QUOTATION { ~('"') | ( '\\' ('"') ) } QUOTATION;  
TRUE : 'true';  
FALSE: 'false';  
COMMA : ',';  
COMMENT : '#' {~[\n\r]} ;

## Przykłady

Przykładowe skrypty w proponowanym języku zamieszczone będą w załącznikach.

## Funkcje wbudowane

Język wyposażony będzie w funkcje wbudowane umożliwiające podstawowe operacje na obrazach i tekstach.

Dla obrazów:

* Image loadImage(Text path) - funkcja wczytująca obraz z pliku do zmiennej
* Text ocr(path) – funkcja poddająca obraz przetworzeniu przez OCR

Dla tekstów:

* Text loadText(Text path) - wczytanie tekstu z pliku,
* Integer countWords(Text text) - policzenie słów,
* Integer countParagraphs(Text text) - policzenie akapitów,
* Integer compare(Text text1, Text text2) - porównanie dwóch ciągów znakowych zwracające ich procentową zgodność.

Dodatkowo:

* print(Text text), print(Integer integer), print(Boolean boolean) - funkcja drukująca wskazane wartości w konsoli.
* toText(Integer integer), toText(Boolean boolean) – funkcja pozwalająca na przetworzenie typów Integer i Boolean na ciąg znaków

## Operatory

Język obsługiwać będzie kilka operatorów realizujących 4 podstawowe działania na liczbach całkowitych z nawiasowaniem, konkatenację tekstów oraz podstawowe działania na wartościach logicznych (koniunkcja, alternatywa, negacja, porównanie dwóch zmiennych) z uwzględnieniem nawiasowania. Zrealizowany ma być także operator przypisania.

# Wymagania

## Wymagania funkcjonalne

* Odczyt skryptu z pliku i utworzenie drzewa rozbioru
* Przejście po drzewie rozbioru i poprawne wykonanie skryptu
* Kontrola poprawności – analiza leksykalna i składniowa przy tworzeniu drzewa oraz analiza semantyczna przy wykonywaniu skryptu. Błędy obsługiwane są poprzez mechanizm wyjątków.
* Możliwość definiowania własnych funkcji i ich wywoływania
* Przeprowadzanie operacji na tekstach, obrazach, liczbach całkowitych i wartościach logicznych
* Możliwość wykorzystania pętli i instrukcji warunkowej

## Wymagania niefunkcjonalne

* Interpreter w przejrzysty sposób informuje użytkownika o rodzaju i miejscu wystąpienia błędu w skrypcie
* Uruchomienie interpretera w niepoprawny sposób powinno skutkować poinformowaniem użytkownika o błędzie i prawidłowej metodzie uruchomienia

# Architektura interpretera

Interpreter będzie aplikacją napisaną w języku Java z wykorzystaniem narzędzia ANTLR w wersji 4. ANTRL generować będzie klasy lexera i parsera tworzącego drzewo wyprowadzenia na podstawie pliku z opisem gramatyki i wykonującego zadany kod.

Logikę interpretera oraz obsługę błędów realizować będzie pakiet zwierający wygenerowany przez narzędzie ANTLR interfejs *Visitor* umożliwiający przejście po drzewie wyprowadzenia z wykonaniem wyznaczonych przez programistę instrukcji. Za obsługę błędów odpowiadać będzie mechanizm wyjątków wykorzystujący zdefiniowane przez programistę wyjątki typu *RuntimeException*, zawierające stosowne powiadomienie o błędzie.

Parser tworzyć będzie dwie struktury danych – mapę w której będzie przechowywał referencje do definicji funkcji (klucz stanowi nazwa funkcji) oraz stos map zawierający zmienne wykorzystywane przez program. Każda z map na stosie stanowić będzie zakres widzialności zmiennych dla funkcji. Przy wywołaniu funkcji na stos położona zostanie nowa mapa, następnie odłożone zostaną do niej argumenty wywołania funkcji. W czasie jej wykonania, funkcja będzie mogła korzystać tylko ze zmiennych znajdujących się aktualnie na szczycie stosu. Po wykonaniu funkcji mapa zostanie zdjęta ze stosu i program będzie mógł powrócić do wykonywania instrukcji w tym samym zakresie zmiennych co przed wywołaniem.

Parser rozpocznie pracę od analizy węzła typu *script* i stopniowo schodzić będzie w kolejne gałęzie drzewa wyprowadzenia. Wybór kolejnej gałęzi zależeć będzie od metod zdefiniowanych w klasie implementującej *Visitor*.

## Testowanie

Jako że w projekcie wykorzystane będzie zewnętrzne narzędzie, testowania wymagać będzie głównie gramatyka oraz implementacja metod z interfejsu Visitor, zatem do interpretera dołączony zostanie zestaw skryptów testujących wszystkie możliwe funkcjonalności języka wraz z oczekiwanym rezultatem ich wykonania.

# Sposób uruchomienia

Interpreter skompilowany do postaci pliku .jar powinien być uruchamiany z konsoli poprzez java -jar interpreter.jar wraz z dodatkowym parametrem zawierającym nazwę pliku ze skryptem do wykonania. Skrypt powinien być zapisany w pliku tekstowym.