Interpreter skryptów dla platformy Android OS

Projekt wstępny

Piotr Jastrzębski
piotr.jastrzebski@gmail.com

1 Opis funkcjonalności

Projekt ma funkcjonować jako integralna część aplikacji, realizowanej na potrzeby pracy inżynierskiej. Będzie on działał pod kontrolą systemu Android, a napisany zostanie w Javie. W związku z koniecznością wielokrotnego testowania różnych funkcji aplikacji wektoryzacji obrazów bitmapowych z różnymi parametrami, interpreter powinien zapewnić taką możliwość, bez konieczności każdorazowej kompilacji. Wielokrotne wywołania mogą zostać wywołane skryptem z zastosowaniem pętli while, a parametry wywołań mogą być zależne od iteratora. Założeniem jest stworzenie interpretera obsługującego wczytywanie skryptów wywołań z poziomu interfejsu graficznego aplikacji. Skrypt powinien zostać przygotowany zgodnie z opisem i uwzględnieniem przeznaczenia zmiennych i wywołań podanymi w punkcie 1.1. Gramatyka wywołań została przedstawiona w punkcie 1.2.

1.1 Specyfikacja formalna

Plik skryptowy dla aplikacji powinien być zapisany w formacie tekstowym z zachowaniem rozszerzenia pliku `"txt"`, np. `"skrypt.txt"`. Obecność białych znaków nie wpływa na działanie aplikacji. Poprawny skrypt musi zawierać wczytanie obrazu wzorcowego i zapis do pliku wyniku w postaci wektorowej. Zmienne liczbowe oraz ścieżka zapisu mogą być zależne od wartości zmiennej iteratora. Wartości argumentów funkcji uzależnić można poprzez dodanie wyrażenia `+n` lub `-n`, po liczbie, a do ścieżki poprzez dodanie `+n+` przed rozszerzeniem pliku. (przyjmując, że `n` wybrano jako symbol zmiennej) Poniżej przedstawiono listę przeznaczenia poszczególnych zmiennych wszystkich funkcji skryptu.

```
load(
   path           //bezwzględna ścieżka obrazu
)

hough_c(
   dp,            //odwrócony współczynnik proporcjonalności
```

```
                //akumulatora
   minDist,     //minimalna odległość pomiędzy
                //środkami okręgów
   gaussSize,   //rozmiar maski filtra Gaussa
   gaussSigma   //współczynnik sigma filtra Gaussa
)

hough_l(
   rho,         //rozdzielczość akumulatora w pikselach
   theta,       //rozdzielczość akumulatora w radianach
   threshold,   //wartość progowa akumulatora
   minLineLength, //minimalna długość odcinka
   maxLineGap,  //maksymalna długość przerwy
   cannyT1,     //mniejsza wartość progowa detektora Cannyego
   cannyT2      //większa wartość progowa detektora Cannyego
)

harris(
   maxCorners,  //maksymalna liczba zwracanych wierzchołków
   qualityLevel, //minimalna "jakość" wierzchołka
   minDistance, //minimalna odległość między zwracanymi
                //wierzchołkami
   blockSize,   //rozmiar sąsiedztwa
   useHarris,   //korzysta z detektora Harrisa dla "true",
                //dla "false" z cornerMinEigenVal()
   k            //wolny parametr detektora Harrisa
)

save(
   path         //bezwzględna ścieżka rezultatu
)

while(
   var          //nazwa zmiennej (jedna litera)
   op           //relacja: <, >, <=, >=, ==, !=
   n            //wartość dziesiętna
)
{...}          //funkcje w pętli

ass(
   var,         //nazwa zmiennej (jedna litera)
   n            //przypisywana wartość (l. dziesiętna)
)
```

```
mod(
   var,           //nazwa zmiennej (jedna litera)
   n              //zmiana wartości zmiennej (l. dziesiętna)
)

progress          //znacznik postępu
```

1.2 Składnia języka w notacji EBNF

```
expression   = loadEx | saveEx | houghCEx | houghLEx |
               harrisEx | whileEx | assEx | modEx | progressEx;
loadEx       = "load" , "("" , path , extensionIn , "")" ;
saveEx       = "save" , "("" , path , extensionOut , "")" ;
houghCEx     = "houghC" , "(" , d , n , n , n , ")" ;
houghLEx     = "houghL" , "(" , n , d , n , n ,
               n , n , n , ")" ;
harrisEx     = "harris" , "(" , n , d , n , n ,
               boolean , d , ")" ;
whileEx      = "while" , "(" , ( var | real ) , op ,
               ( var | real ) , ")" , "{" , {expression} , "}" ;
assEx        = "ass" , "(" , var , "," , real , ")" ;
modEx        = "mod" , "(" , var , "," , real , ")" ;
progressEx   = "progress" ;
path         = pathPart , { pathPart } ,
               [ "+" , var , +" ] , "." ;
pathPart     = "/" , ( character | digit ) , { character | digit } ;
extensionIn  = "jpg" | "jpeg" | "bmp" | "gif" | "png" ;
extensionOut = "svg" ;
n            = nat , varDep ;
d            = posReal , varDep ;
nat          = posDigit , {digit} | "0" ;
posReal      = nNum , [ "." , digit, {digit} ] ;
int          = [-] , nat ;
real         = [-] , posReal ;
varDep       = [ ( "-" | "+" ) , var ]
character    = var | "_" | "-" ;
digit        = "0" | posDigit ;
var          = "a" | "b" | "c" | "d" | "e" | "f" | "g" |
               "h" | "i" | "j" | "k" | "l" | "m" | "n" |
               "o" | "p" | "q" | "r" | "s" | "t" | "u" |
               "v" | "w" | "x" | "y" | "z" ;
posDigit     = "1" | "2" | "3" | "4" | "5" | "6" | "7" |
               "8" | "9" ;
op           = "<" | ">" | "<=" | ">=" | "==" | "!=" ;
```

```
boolean      = "true" | "false" ;
```

2 Wymagania funkcjonalne

- parsowanie skryptów zapisanych w plikach tekstowych
- umożliwienie wielokrotnego wykonywania wywołań zamkniętych w pętli while i zależnych od iteratora
- przestrzeganie logicznego porządku wczytanie-funkcje-zapis
- możliwość wywoływania funkcji wielokrotnie i w dowolnej kolejności
- kontrola poprawności wprowadzonych danych
- informowanie użytkownika, w którym miejscu skryptu wystąpił błąd

3 Wymagania niefunkcjonalne

Projekt powstaje jako integralna część programu wektoryzacji obrazów bitmapowych realizowanego na potrzeby pracy inżynierskiej. Konieczne jest poszerzenie interfejsu użytkownika o dodatkowy przycisk wywołujący parsowanie skryptu. Ścieżkę do skryptu definiuje się w aktualnie istniejącym polu tekstowym aplikacji. Zrzuty ekranu z działania aplikacji przedstawiono na rysunku 1.

4 Projekt realizacji

Aplikacja parsera składa się z kilku modułów: skanera, parsera i analizatora składniowego. Schemat przekazywania informacji pomiędzy modułami, a także między elementami parsera a aplikacją wektoryzującą przedstawiono na rysunku 2. Tablica symboli jest tablicą globalną i przy wystąpieniu w strukturze skryptu nowego symbolu umieszcza go w strukturze tablicy. Symbole te wykorzystywane są potem podczas generacji i wykonywania kodu.
Klasy programu:

- *StartActivity:* klasa UI (obsługa zdarzeń przycisków, przechwytywanie ścieżki skryptu, komunikacja z użytkownikiem)
- *FileUtil:* klasa obsługi pliku (obsługa plików, czytanie znaków ze strumienia)
- *ScanerUtil:* klasa skanera (rozbicie tekstu, pomijanie białych znaków, rozpoznawanie tokenów)

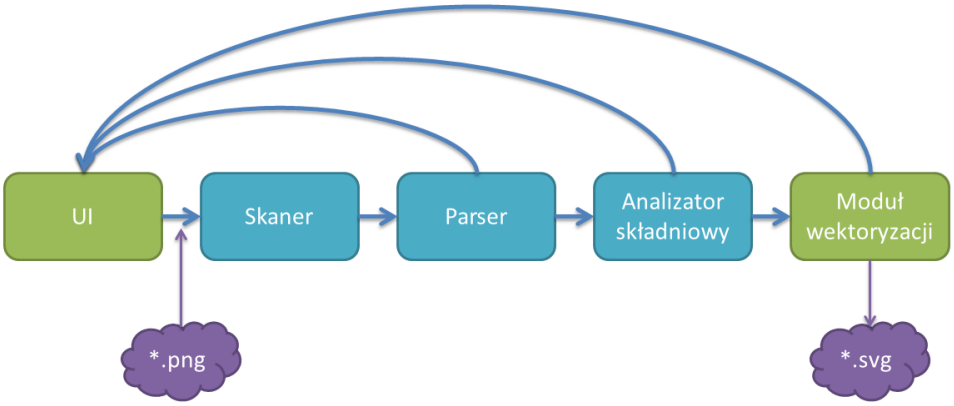
(a) Skrypt niepoprawny (b) Skrypt poprawny

Rysunek 1: Wygląd okna aplikacji w systemie Android

- *ParserUtil:* klasa parsera (tworzenie drzewa rozbioru, sprawdzanie zgodności z gramatyką)
- *SyntaxUtil:* klasa analizatora składniowego (sprawdzanie zgodności semantycznej skryptu z założeniami)
- *ProcessImage:* klasa przetwarzania obrazu (wykonuje funkcje graficzne na obrazie, zapisuje plik SVG)

4.1 Analiza leksykalna (skanowanie)

Polega na rozbiciu wczytanego z pliku tekstu na leksemy. Podczas skanowania ignorowane są wszelkie białe znaki. Następnie na podstawie leksemów zostają rozpoznane tokeny (odpowiednie przypasowanie do wzorca). Skaner rozpoznaje następujące tokeny:



Rysunek 2: Schemat zależności między modułami aplikacji.

Przykład leksemu	**Token**
load save houghC houghL harris while mod ass	wyróżnik funkcji
progress	znacznik postępu
(	początek funkcji
)	koniec funkcji
{	początek pętli
}	koniec pętli
-	minus
,	przecinek
x	zmienna
!=	operator
123	liczba naturalna
1.23	liczba dziesiętna
”/mnt/sdacrd/bsc/a.jpg”	napis
true	wartość logiczna

Dla przykładu:

```
load("/mnt/sdcard/bsc/1.jpg")
harris(99, 0.01, 58, 3, true, 0.04)
save("/mnt/sdcard/bsc/w1.svg")
```

analiza leksykalna wyglądałaby tak:

Leksem	**Token**	**Atrybut**
load	wyróżnik funkcji	load
(	początek funkcji	
”/mnt/sdcard/bsc/1.jpg”	napis	/mnt/sdcard/bsc/1.jpg
)	koniec funkcji	
harris	wyróżnik funkcji	harris
(	początek funkcji	
99	liczba naturalna	99
,	przecinek	
0.01	liczba dziesiętna	0.01
,	przecinek	
58	liczba naturalna	58
,	przecinek	
3	liczba naturalna	3
,	przecinek	
true	wartość logiczna	true
,	przecinek	
0.04	liczba dziesiętna	0.04
)	koniec funkcji	
save	wyróżnik funkcji	save
(	początek funkcji	
”/mnt/sdcard/bsc/w1.svg”	napis	/mnt/sdcard/bsc/w1.svg
)	koniec funkcji	

4.2 Analiza składniowa (parsowanie)

Analizator składniowy (parser) otrzymawszy od skanera ciąg symboli leksykalnych sprawdza czy może on zostać wygenerowany przez gramatykę. Tworzone jest drzewo składni na podstawie zapytania i sprawdzana są możliwości wykonania zapytania. W przypadku niemożliwości wygenerowania parser zgłasza błąd, o którym informuje użytkownika i przerywa interpretację.

Lista produkcji:

```
S         = wyróżnik funkcji + początek funkcji + ARGUMENTY
            + koniec funkcji
S         = wyróżnik funkcji + początek funkcji + WARUNEK
            + koniec funkcji + początek pętli + FUNKCJE
            + koniec pętli
S         = znacznik postępu
```

```
FUNKCJE   = S
FUNKCJE   = S + FUNKCJE
WARUNEK   = zmienna + operator + l. dziesiętna
WARUNEK   = zmienna + operator + minus + l. dziesiętna
ARGUMENTY = zmienna + przecinek + l. dziesiętna
ARGUMENTY = zmienna + przecinek + minus + l. dziesiętna
ARGUMENTY = napis
ARGUMENTY = l.dziesiętna + l.naturalna + l.naturalna + l.naturalna
ARGUMENTY = l.naturalna + l.dziesiętna +l.naturalna + l.naturalna
            + l.naturalna + l.naturalna + l.naturalna
ARGUMENTY = l.naturalna + l.dziesiętna + l.naturalna + l.naturalna
            + w.logiczna + l.naturalna
```

Każdy symbol nieterminalny gramatyki odpowiada jednemu węzłowi w drzewie rozbioru. Każdy taki węzeł ma swoją metodę *execute()*, która daje wynik po analizie wywołań funkcji *execute()* węzłów podrzędnych itd.

Struktury danych:

- *SAbstractC:* klasa abstrakcyjna węzła
- *SWhileC:* klasa pętli while
- *SFunC:* klasa funkcji innych niż pętla while
- *FunsC:* klasa pozwalająca na hierarchiczne wywoływanie funkcji
- *ArgsC:* klasa nadrzędna zbiorów argumentów
- *CondC:* klasa warunku logicznego
- *NatC:* klasa l. naturalnej
- *PosRealC:* klasa l. rzeczywistej nieujemnej
- *BoolC:* klasa w. logicznej
- *StringC:* klasa napisu

4.3 Analiza semantyczna

Po zakończeniu faz analizy leksykalnej i analizy składniowej następuje analiza semantyczna. Zadaniem tej fazy jest sprawdzenie programu źródłowego pod względem semantycznej zgodności z definicją języka źródłowego np. czy typy operandów są zgodne z wymaganiami.

5 Przykłady testowe

5.1 Pozytywne

Powinny zrealizować wczytanie, zrealizowanie funkcji przetwarzania obrazów, a następnie zapisanie wyników wykrywania tych elementów w sposób wektorowy do pliku SVG:

- Wykrywanie okręgów, odcinków, wierzchołków.

```
load("/mnt/sdcard/bsc/1.jpg")
houghC(1.150, 58, 9, 2)
houghL(1, 0.0174532925, 10, 10, 10, 50, 200)
harris(99, 0.01, 58, 3, true, 0.04)
save("/mnt/sdcard/bsc/wynik_skrypt1.svg")
```

- Wykrywanie odcinków.

```
load("/mnt/sdcard/camera/DCIM1209.jpeg")
houghL(1, 0.02, 10, 5, 20, 75, 100)
save("/mnt/sdcard/test/test.svg")
```

5.2 Negatywne

- Parser powinien zwrócić błąd dla:

```
load("/mnt/sdcard/bsc/1.jpg)     //brak domknięcia cudzysłowu
houghC(1.150, 58, 9)             //zbyt mała liczba argumentów
ass(xy, -12.3)                   //nieprawidłowa zmienna
xyz(99, 0.01, 58, 3, true, 0.04) //nieznana funkcja
save("/mnt/sdcard+x+/wynik.svg") //uzależnienie od zmiennej
                                 //w złym miejscu
```