Łukasz Lepak, 277324

TKOM 18L – projekt
Dokumentacja końcowa
Translator R – C++
prowadzący: dr inż. Piotr Gawkowski

1. Opis zadania

Celem zadania było stworzenie programu, który tłumaczy kod napisany w języku programowania R na język programowania C++. Translacja miała umożliwiać tłumaczenie pewnego zakresu funkcji języka R, m.in podstawowe operacje, instrukcje warunkowe czy pętle. Możliwe miało być także definiowanie wektorów oraz macierzy charakterystycznych dla języka R.

2. Zrealizowane funkcjonalności

a) definicje zmiennych oraz rozróżnienie ich typów (INTEGER, DOUBLE, STRING, BOOLEAN, VECTOR, MATRIX)
b) podstawowe operacje na zmiennych (+, -, *, /)
c) operacje na wektorach tej samej długości (dodawanie, odejmowanie, mnożenie element z elementem, długość wektora)
d) operacje na macierzach (dodawanie, odejmowanie, mnożenie przez stałą, dzielenie przez stałą, mnożenie element z elementem, mnożenie macierzowe, wyznacznik)
e) operacje i instrukcje warunkowe oraz logiczne
f) pętle while, repeat oraz for (w wersji „range for”)
g) mapowanie zmiennych zmieniających swój typ
h) podział widoczności zmiennych
i) wypisywanie zawartości
j) indeksowanie zawartości

Duży przykład pokazujący działanie funkcjonalności w praktyce będzie załączony wraz z plikiem *TKOM.R*. Zostaną dołączone również inne pliki (*hello.R, hello2.R, hello3.R*) z kodem w języku R, jednak będą one skupione na małych funkcjonalnościach.

3. Gramatyka

Gramatyka języka R została pobrana ze strony:
https://github.com/antlr/grammars-v4/blob/master/r/R.g4
Jest ona dostosowana do użycia wraz z programem ANTLR służącym do generowania lekserów oraz parserów po odpowiednim zdefiniowaniu języka. Właśnie za pomocą tego programu zostały wygenerowane klasy *RLexer* oraz *RParser* używane do analizy języka R. Plik R.g4 został rozszerzony także o akcje semantyczne, których zadaniem było stworzenie odpowiednich obiektów reprezentujących dane struktury językowe w języku R w celu ich przetłumaczenia.

4. Wymagania funkcjonalne i niefunkcjonalne, obsługa błędów

Wymagania funkcjonalne:
- program powinien rozpoznawać elementy gramatyki języka R
- program powinien móc przetłumaczyć zgodny z gramatyką języka R program na język C++
- program powinien wykryć nieobsługiwane w translatorze elementy składniowe języka R
- program powinien móc przyjąć pliki w formacie *.r* i odczytać je prawidłowo
- program powinien móc wyświetlić drzewo parsowania danego pliku wejściowego zgodnie z życzeniem użytkownika
- program powinien być w stanie kontynuować translację pomimo napotkania błędu (iść dalej, jakby dane wyrażenie w ogóle nie nastąpiło) zgodnie z życzeniem użytkownika
-program powinien dołączyć wszystkie potrzebne biblioteki w języku C++ tak, aby poprawny program w R po przetłumaczeniu kompilował się w C++ wybranym kompilatorem (np. MinGW)

Wymagania niefunkcjonalne:
- program powinien translować zarówno krótkie programy, jak i bardziej rozwinięte pliki języka R
- program powinien informować o błędach w trakcie translacji, jednak niekoniecznie ją przerywać

Obsługa błędów:
Program będzie wychwytywać błędy, które wydarzą się podczas translacji kodu języka R na C++. Pozwoli użytkownikowi kontynuować translację z pominięciem błędnego wyrażenia bądź zakończy działanie jak tylko wykryje błąd (w zależności od ustawionego parametru).

5. Tabela translacji

W tabeli poniżej są zawarte wszystkie większe translacje zrealizowane w translatorze. Więcej przykładów znajduje się w przykładowych plikach załączonych wraz z dokumentacją.

R	C++
var <- *value* *value* -> var var = *value*	auto var = *value;*
a=c(*values…*) // wektor	auto a = *arma::vec{values…};*
print(*var1 op var2*)	std::cout << *var1 op var2* << std::endl;

a <- seq(*number1*, *number2*, by = *delta*)
 	
b <- seq(*number1*, *number2*)
	auto a = regspace< vec>(*number1*, *delta*, *number2*);
	
	auto b = regspace< vec>(*number1*, *number2*);

mat <- matrix(c(*numbers*), ncol = *columns*, nrow = *rows*, byrow = TRUE) mat2 <- matrix(c(*numbers*), ncol = *columns*, nrow = *rows*)	auto mat = matrix(std::vector<double>{*numbers[0], …, numbers[columns – 1], …, numbers[columns * (rows – 1)], …, numbers[columns * (rows – 1) + columns – 1]}.data(), rows, columns);* auto mat2 = […]{*numbers[0], numbers[rows], numbers[2 * rows],…, …, {numbers[columns – 1, …}[…];*
vec[*index*]	vec(*index – 1);*

for(*i* in *number1:number2)* {
…
}
	auto *i = number1;*
	for(*i*; *i <= number2; ++i)*
	{
	…
	}

while(*condition*) {
…
}
	while(*condition)*
	{
	…
	}

repeat {
…
}
	do
	{
	…
	} while(true);

length(*vec*)	*vec.n_elem;*
det(*mat*)	arma::det(*mat*);

if(*condition*) {
…
}
	if(*condition*) {
	…
	}

6. Opis klas i pakietów

Poza wszystkimi pakietami znajduje się klasa Main. Odpowiada ona za odczytanie parametrów uruchomienia programu, uruchomienie analizatora leksykalnego oraz składniowego, przeprowadzenie translacji oraz ewentualne wyświetlenie drzewa parsowania podanego pliku.

Pakiet **context:**
Jest to pakiet reprezentujący dane potrzebne do reprezentacji kontekstu oraz reprezentowania danych w nim.
Klasa *ContextHolder* jest główną klasą dla translatora. Zawiera ona informacje o obecnym kontekście, w jakim translator się znajduje. Posiada m. in. globalną tablicę symboli, lokalną tablicę mapowania zmiennych (potrzebna do zmian typów zmiennych), lokalną listę zmiennych, ilość obecnych wcięć, listy przechowujące tablicę mapowania zmiennych oraz listy zmiennych wyższych poziomów, a także writera, do którego zapisuje przetłumaczone rzeczy. Udostępnia ona metody statyczne umożliwiające zmianę kontekstu, przywrócenie go, wypisanie początku oraz końca pliku *.cpp* oraz gettery i settery na poszczególne kolekcje czy wartości. Jest ona klasą publiczną, zatem każdy obiekt ma do niej dostęp i może dowiedzieć się, w jakim stanie obecnie znajduje się przetłumaczony do tej pory program. W pakiecie context znajduje się również pole enumerowane *Type* do określania typów znalezionych zmiennych oraz definicja klasy *VariableData* służącej do trzymania informacji o danej zmiennej w tablicy symboli.

Pakiet **gen:**
Są to klasy wygenerowane przez ANTLR-a z plików R.g4 oraz RFilter.g4. Znajdują się tam m.in. klasy *RLexer* oraz *RParser*, służące odpowiednio do przeprowadzenia analizy leksykalnej i składowej. W folderze tym znajduje się również plik z biblioteką ANTLR-a potrzebną w programie oraz wyżej wspomnianymi plikami *g4*. Plik R.g4 posiada zapisane definicje akcji semantycznych po rozpoznaniu danych wyrażeń.

Pakiet **expression:**
Jest to główny pakiet zawierający definicje klas obiektów reprezentujących wyrażenia języka R. Każda klasa ma zdefiniowaną metodę *translate*, która pozwala przetłumaczyć dane wyrażenie zamknięte w obiekcie na język C++.
IExpression – interfejs, po którym dziedziczą wszystkie klasy z tego pakietu, definiuje metody *translate, type, print*
AddSubExpr –wyrażenie dodające do/odejmujące od siebie wartości dwóch wyrażeń
AssignmentExpr –wyrażenie przypisujące wartość do zmiennej
BoolExpr –wartość typu logicznego (prawda lub fałsz)

BracedExpr – wyrażenie zamknięte między nawiasami okrągłymi
BreakExpr – wyrażenie pozwalające wyjść z pętli
CallFunExpr – wyrażenie wywołujące funkcję (z argumentami bądź bez)
CompareExpr – wyrażenie porównujące dwie wartości pewną relacją
CompoundExpr – wyrażenie złożone, zamknięte między dwoma nawiasami klamrowymi
FloatExpr – wartości typu zmiennoprzecinkowego
ForExpr – wyrażenie pętli *for*
HexExpr – wartość całkowita zapisana szesnastkowo
IDExpr – wyrażenie reprezentujące zmienne bądź wywołania funkcji bez przypisania
IfElseExpr – wyrażenie bloku if/else (nie używane w tym projekcie)
IfExpr – wyrażenie bloku if z warunkiem
IndexExpr – wyrażenie indeksu (np. numer elementu w wektorze)
IntExpr – wartość typu całkowitego
LogicalExpr – wyrażenie ewaluowane do wartości logicznej
MulDivExpr – wyrażenie mnożące/dzielące dwa wyrażenia
NegationExpr – wyrażenie zaprzeczające danej wartości logicznej
NextExpr – wyrażenie pozwalające przejść do kolejnej iteracji pętli
PlusMinusExpr – wyrażenie z unarnym plusem/minusem
RangeExpr – wyrażenie wektora o zakresie od jednej wartości do drugiej
RepeatExpr – wyrażenie pętli bez warunku wyjścia, odpowiednik do..while(true)
StringExpr – wartość tekstowa
UserOpExpr – wyrażenie reprezentujące operatory zamknięte między dwoma znakami % (potrzebne np. do mnożenia macierzy nie element po elemencie)
WhileExpr – wyrażenie pętli iterującej dopóki jest spełniony dany warunek

Pakiet **argument:**
Jest to pakiet reprezentujący wartości, które mogą być argumentami funkcji. Tak samo jak klasy z **expression**, posiadają zdefiniowaną funkcję translate.
IArgument – interfejs argumentów, definiuje metody *translate, type, print*
ExprArgument – argument ewaluowany do wyrażenia, najczęściej przy wywołaniach funkcji
IDArgument – argument nazywający parametr funkcji, najczęściej przy definiowaniu nowych funkcji

Pakiet **exceptions:**
Zawiera definicje wyjątków rzucanych przez program.
TranslationException – wyjątek rzucany, kiedy translator napotka błąd uniemożliwiający mu przetłumaczenie danego wyrażenia

Pakiet **utilities:**
Klasy pomocnicze.
RandomString – klasa generująca losowe ciągi znaków, przydatna przy definiowaniu mapowania nazw nowych zmiennych

7. Realizacja

Translator przechowuje informacje o swoim stanie w statycznych polach klasy *ContextHolder.* Znajduje się tam tablica symboli, która jest mapą kluczowaną nazwami zmiennych, a jako wartości trzyma obiekty klasy *VariableData*. Pozwalają one zachować wymaganą do translacji informację o danych zmiennych. Znajduje się tam także mapa, która w lokalnym obszarze łączy daną nazwę zmiennej z inna. Jest to używane do symulowania zmiany typu danej zmiennej. W klasie tej znajduje się także lista z lokalnymi nazwami zmiennych. W klasie *ContextHolder* znajdują się także listy przechowujące mapy łączące nazwy oraz listy zmiennych wyższych poziomów, dzięki czemu informacje z wyższej części nie są utracone. Klasa ta udostępnia także metody *changeContext, restoreContext* umożliwiające zmiany kontekstu. Klasa przechowuje również informacje o ilości wcięć wymaganej przy danym poziomie. Metoda *restoreContext* pozwala dodatkowo także na realizację „globalności" wszystkich zmiennych dzięki dodawaniu zmiennych z opuszczanego kontekstu do obecnego.

Obiekty rozpoznane przez parser są przekazywane do klas reprezentujących dane wyrażenia bardziej konkretnie. Następnie na obiektach tychże klas wywoływana jest metoda *translate*, tłumacząca dany obiekt względem danego kontekstu. Niestety, definicja gramatyki R sprawiła, że w wielu miejscach potrzebne było rzutowanie typów oraz nadpisywanie/podmienianie/wyciąganie wartości wygenerowanych przez parser.

Translator zapisuje przetłumaczone wyrażenia do pliku bądź na standardowe wyjście (jeśli ścieżka do pliku nie została podana). Ścieżkę do pliku z programem można podać jako pierwszy parametr wywołania bądź (jeśli go nie podamy) można pisać program języka R bezpośrednio w konsoli.

8. Kompilacja, sposób uruchomienia, parametry

Program kompilujemy znajdując się w katalogu src za pomocą instrukcji:
javac -cp „main/gen/antlr-4.7.1-complete.jar;.” main/Main.java
Program możemy uruchomić za pomocą instrukcji:
java -cp „main/gen/antlr-4.7.1-complete.jar;.” main/Main
Program możemy wywołać z następującymi parametrami:
java -cp „main/gen/antlr-4.7.1-complete.jar;.” main/Main in
java -cp „main/gen/antlr-4.7.1-complete.jar;.” main/Main in out
java -cp „main/gen/antlr-4.7.1-complete.jar;.” main/Main in out cont
java -cp „main/gen/antlr-4.7.1-complete.jar;.” main/Main in out cont tree
java -cp „main/gen/antlr-4.7.1-complete.jar;.” main/Main help
Opis parametrów:
in – plik wejściowy języka R do translowania
out – plik wyjściowy z przetłumaczonym kodem w C++
cont – *true* jeśli chcemy kontynuować translację po błędzie (ignorując błędnę wyrażenie), w przeciwnym wypadku *false*
tree – *true* jeśli chcemy na końcu wyświetlić drzewo parsowania pliku, w przeciwnym wypadku *false*

9. Sposób testowania

Do programu zostało dołączonych 25 testów jednostkowych testujących zdolność translacji pojedynczych wyrażeń. Są to proste testy, sprawdzają one tylko, czy dany obiekt jest dobrze tłumaczony w danej trywialnej sytuacji. Bardziej rozbudowanymi testami były testy przeprowadzane na przykładach dołączonych do dokumentacji. Wszystkie testy jednostkowe wykazały poprawność sprawdzanych rezultatów, a „większy” test przeprowadzony na przykładowym pliku dołączonym do dokumentacji wykazał, że skrypt w języku R i jego tłumaczenie w języku C++ wypisały tę samą zawartość na konsolę.