Kacper Roszczyna

Projekt TKOM

Sprawozdanie

Temat: Język maszyny reguł do inwestowania na giełdach walutowych i akcyjnych.

I. Cele

* Głównym celem projektu jest napisanie kompilatora języka maszyny reguł umożliwiającej inwestycje giełdowe.

II. Opis Świata

Realizowana jest kompilacja języka służącego do tworzenia reguł inwestowania na giełdach akcyjnych i w waluty. Zakłada się, że istnieje jeden inwestor, a otaczający go świat się zmienia z dnia na dzień, na co reagują reguły, które wykonują pewne czynności na kapitale inwestora. Program wymaga w pełni zdefiniowanego i spójnego świata.

III. Opis Plików - Ogólny

1. Pliki konfiguracyjne definiujące dostępne produkty walutowe i produkty akcyjne.

2. Pliki konfiguracyjne zawierające listę zdarzeń, czyli zmian kursów walutowych i cen akcji.

3. Pliku startowego, który zawiera początkowy stan posiadania inwestora.

4. Pliku programu, który zawiera listę reguł złożonych z warunków wejścia oraz akcji oraz listę użytych plików konfiguracyjnych.

IV. Opis Plików - Szczegółowy

1. Składnia pliku konfiguracyjnego zawierającego produkty walutowe i akcyjne. Jest to plik .xml, który w ramach zadania będzie odczytywany poprzez gotową bibliotekę i w przypadku nie wystąpienia elementów obowiązkowych rzuci błąd. W przypadku wystąpienia elementów nadmiarowych zostaną one pominięte. Opis pliku:

* + Tag „*products”*: znacznik pierwszego poziomu dla pliku konfiguracyjnego. Jest obowiązkowy.
  + Tag „*currencies”:* znacznik drugiego poziomu, oznacza rozpoczęcie bloku definicji walut. Jego nieobecność w pliku wskazuje, że nie są definiowane żadne produkty walutowe. Obecność znacznika nie jest obowiązkowa. Blok może być pusty.
  + Tag „*currency”:* znacznik musi wystąpić wewnątrz bloku „*currencies”*. Gdy zostanie wykryty traktowane jest to jako początek definiowania nowego produktu walutowego.
    - Na definicję produktu walutowego składają się dwa obowiązkowe elementy nazwa i skrót, które oznaczone są:
      * „*name”:* znacznik, który zawiera pełną nazwę produktu. Jest obowiązkowy.
      * „*abbreviation”*: skrót nazwy produktu, który musi być unikatowy ze wszystkich produktów walutowych. W przypadku powtórzenia się skrótu nazwy kompilator rzuci błąd ze wskazaniem na zduplikowaną nazwę. Znacznik jest obowiązkowy.
  + Tag „*stocks”*: znacznik oznacza rozpoczęcie bloku definicji produktów akcyjnych. Znacznik nie jest obowiązkowy, jego brak oznacza, że plik konfiguracyjny nie definiuje żadnych produktów giełdowych. W przypadku pustej sekcji oznacza brak definicji produktów.
  + Tag „*stock”*: znacznik oznacza rozpoczęcie definicji produktu giełdowego. Znacznik ten musi znajdować się wewnątrz sekcji „*stocks*”.
    - Na definicję produktu akcyjnego składają się dwa elementy: nazwa i waluta, w której można kupować akcje.
      * „*name”:* nazwa produktu. Nazwa musi być unikatowa ze wszystkich nazw produktów akcyjnych. Znacznik jest obowiązkowy. W przypadku powtórzenia się nazwy kompilator rzuci błędem i wskaże zduplikowaną nazwę.
      * „*currency*”: skrót nazwy waluty, w której można kupić akcje. Jeżeli określony skrót nie zostanie zdefiniowany w żadnym pliku konfiguracyjnym poprzedzającym jego użycie kompilator rzuci błędem ze wskazaniem na ten skrót.

Przykładowy plik:

<products>

<currencies>

<currency>

<name>yuan</name>

<abbreviation>YUA</abbreviation>

</currency>

<currency>

<name>yen</name>

<abbreviation>YEN</abbreviation>

</currency>

<currency>

<name>dollar USA</name>

<abbreviation>USD</abbreviation>

</currency>

</currencies>

<stocks>

<stock>

<name>CocaCola</name>

<currency>USD</currency>

</stock>

</stocks>

</products>

2. Składnia pliku zdarzeń zawierającego zmiany kursów walut i produktów akcyjnych. Jest to plik .xml, który w ramach zadania będzie odczytywany poprzez gotową bibliotekę i w przypadku nie wystąpienia elementów obowiązkowych rzuci błędem, zaś elementy nadmiarowe zignoruje. Opis pliku zdarzeń:

* + „*events”:* blok definiujący zdarzenia zachodzące w rzeczywistości, na skutek, których będą wykonywane różne reguły zdefiniowane w programie. Znacznik najwyższego rzędu, może być pusty.
    - „*event”:*  definicja pojedynczego zdarzenia, wszystkie pola są obowiązkowe.
      * „*type”:*  może przyjmować dwie wartości: STOCK lub CURRENCY. Które odpowiadają zmianie kursu akcji lub zmianie kursu waluty. Inne wartości spowodują błąd.
      * „*name”:* nazwa akcji lub skrót nazwy waluty. Nazwa ta musi występować w plikach konfiguracyjnych wcześniej wczytanych.
      * „*date”:* data nastąpienia zdarzenia. Jeżeli data zdarzenia jest ustawiona wcześniej niż data startu to kompilator wyrzuci ostrzeżenie. Format daty to yyyy.MM.dd. Inne formaty nie są przetwarzane i skutkują błędem.
      * „value*”:* kurs akcji lub waluty w dniu podanym w elemencie *date*.

<events>

<event>

<type>STOCK</type>

<name>CocaCola</type>

<date>2016.05.10</date>

<value>17.0</value>

</event>

<event>

<type>CURRENCY</type>

<name>YUA</type>

<date>2016.05.10</date>

<value>136</value>

</event>

</events>

3. Plik startowy zawiera startowy stan własności inwestora. Jest to plik XML, który opisuje ile jakiej waluty i akcji posiada inwestor. Plik jest parsowany za pomocą gotowej biblioteki i w przypadku nie wystąpienia elementów obowiązkowych rzuci błąd, zaś elementy nadmiarowe są ignorowane. Przykładowy plik:

* „*start”:* tag otwierający przetwarzanie, jest obowiązkowy.
  + „*currencies”:* blok wewnątrz, którego definiowane są posiadane waluty oraz ich ilość. Złożony z wielu elementów „*currency”.* Blok może być pusty, może też nie wystąpić.
    - „*currency”:* blok definiujący ilość posiadanej waluty przez inwestora, wszystkie elementy są obowiązkowe.
      * „*name”*: skrót nazwy waluty. Skrót musi zostać zdefiniowany w plikach konfiguracyjnych.
      * „*value”*: jest to wartość zmiennoprzecinkowa o dwóch miejscach znaczących, która oznacza ilość posiadanej waluty.
  + „*stocks*”: blok wewnątrz, którego definiowane są posiadane akcje. Złożony z wielu elementów „*stock”.* Może być pusty lub nie występować w poszczególnych plikach, ale wszystkie kursy początkowe muszą być zdefiniowane.
    - „*stock”*: blok definiujący początkową liczbę posiadanych akcji, wszystkie elementy obowiązkowe.
      * „*name”:* nazwa akcji, musi wystąpić we wcześniejszych plikach konfiguracyjnych.
      * „*amount*”: liczba posiadanych akcji.

<start>

<currencies>

<currency>

<name>USD</name>

<value>1.75</value>

</currency>

</currencies>

<stocks>

<stock>

<name>CocaCola</name>

<amount>300</amount>

</stock>

</stocks>

</start>

4. Plik programu

* Sekcje programu występują po sobie, gdy przejdzie się do kolejnej sekcji, nie wolno deklarować elementów z sekcji poprzedniej.
* Korzystanie z atrybutów modelu:
  + Inwestowanie i formułowanie reguł inwestycyjnych jest oczywiście zależne od naszego stanu posiadania oraz tego jakie są ceny akcji lub kursy walut. Model świata inwestycyjnego można podzielić między akcje (stock) i waluty (currency).
  + Liczby zmiennoprzecinkowe definiowane są z separatorem „ . ”.
  + *currency :* słowo kluczowe, które pozwala odnieść się do poszczególnych atrybutów. Atrybuty od słowa klucza oddzielone są przez „.” i skrótem nazwy waluty w nawiasach „[]”.
  + *have:* słowo kontekstowe wskazujące stan posiadania inwestora. Stosowane w kombinacjach z ze słowami *currency* i *stock*
    - *currency.[<skrót>].rate :* aktualny kurs waluty, czyli ile trzeba zapłacić tej waluty za jednostkę fikcyjnej waluty.
    - *currency.[<skrót>].have.amount :* aktualna ilość posiadanej waluty, liczba z dwoma miejscami po przecinku
    - Każdy z atrybutów przyjmuje dodatkowy argument, którym może być data lub liczba całkowita. Oznaczają one wartość z dnia podanego lub tyle dni wstecz. Podana data musi mieć format yyyy.MM.dd. I być otoczona znakami @. Argumenty otoczone są nawiasami okrągłymi.
      * Przykład: currency.[YUA].value(@2016.08.10@) – kurs YUA z dnia 10.08.2016.
      * Przykład 2: currency.[YUA].value(10) – kursu YUA 10 dni temu.
  + *stock:* słowo kluczowe pozwalające się odnieść do atrybutów akcji giełdowych. Atrybuty od słowa klucza oddzielone są przez „.” i nazwą akcji.
    - *stock.[<stock\_name>].value :* aktualna wartość pojedynczej akcji w walucie, w której występuje ten produkt na rynku
    - *stock.[<stock\_name>].have.amount:* liczba akcji posiadanych przez inwestora
    - Każdy z atrybutów przyjmuje dodatkowy argument, którym może być data lub liczba całkowita. Oznaczają one wartość z dnia podanego lub tyle dni wstecz. W przypadku podania wartości dodatniej minus jest automatycznie dodany. Podana data musi mieć format yyyy.MM.dd. I być otoczona znakami @. Jeżeli format daty się nie zgadza to jest rzucany błąd. Argumenty otoczone są nawiasami okrągłymi.
      * Przykład: stock.[CocaCola].value(@2016.08.10@) – kurs Coca Coli z dnia 10.08.2016.
      * Przykład 2: stock.[CocaCola].value(10) – kursu Coca Coli 10 dni temu.
* Sekcja konfiguracji:
  + *config –* słowo kluczowe poprzedzające nazwę pliku konfiguracyjnego, czyli takiego, który definiuje waluty i akcje. Plik musi znajdować się w tym samym katalogu co główny plik reguł. Brak pliku skutkuje ostrzeżeniem.
    - Przykład: *config [example\_config.xml]*
    - Blok konfiguracyjny: *config [<plik\_konfiguracji.xml>]*
    - Może wystąpić wiele bloków konfiguracyjnych ale musi się pojawić co najmniej 1.
  + *events –* słowo kluczowe poprzedzające nazwę pliku ze zdarzeniami, czyli takiego, który definiuje zdarzenia zaszłe w symulacji. Plik musi znajdować się w tym samym katalogu co główny plik reguł. Brak pliku skutkuje ostrzeżeniem.
    - Przykład: *events [example\_events.xml]*
    - Blok zdarzeń: *events [<plik\_zdarzen.xml>]*
    - Może wystąpić wiele bloków zdarzeń ale musi się pojawić co najmniej 1.
  + *start –* słowo kluczowe poprzedzające nazwę pliku z początkowym stanem posiadania inwestora. Plik musi znajdować się w tym samym katalogu co główny plik reguł. Brak pliku skutkuje ostrzeżeniem.
    - Przykład: start *[start\_config.xml]*
    - Blok startowy: start *[<plik\_startu.xml>]*
    - Może wystąpić wiele bloków startu, ale musi pojawić się co najmniej 1.
* Sekcja reguł:
  + Rozpoczyna ją pierwsza zadeklarowana reguła. Reguły można deklarować na dwa sposoby:
    - *rule [<plik\_reguly.invrl>],* gdzie plik\_reguly.invrl jest plikiem deklarującym regułę zgodnie z drugim typem deklaracji. Można zawrzeć dowolną liczbę takich deklaracji.
    - Pełną deklaracją reguły.
* Pełna deklaracja reguły.
  + Rozpoczęta słowem kluczowym: *rule.*
  + Ograniczona za pomocą nawiasów {}
  + Każde pole reguły jest zakończone „;”
  + Nazwy pól reguły od wartości oddzielone są za pomocą „:”
  + Tak więc ogólnie przyjmują postać: *nazwa\_pola: wartosc\_pola;* Pola dla reguł są obowiązkowe
    - Pierwszym polem każdej reguły jest *id*. Jest to wartość liczbowa, która jednoznacznie identyfikuje regułę. Nie mogą wystąpić dwie reguły o jednakowym id. Jeżeli tak się stanie to kompilacja zostanie przerwana z błędem.
      * Przykład: id: 10;
    - Drugim polem każdej reguły jest *priority.* Określa ona priorytet sprawdzenia warunków wejścia do reguły i wykonania jej akcji. Priorytet jest dodatnią liczbą całkowitą. W przypadku gdy ten warunek nie zostanie spełniony to kompilacja zostanie przerwana z błędem. Reguły sprawdzane są od najwyższego do najniższego priorytetu.
      * Przykład: priority: 55;
    - Trzecim polem każdej reguły jest *condition*. Pole to zawiera wyrażenie ewaluowane do wartości logicznej prawda/fałsz, która jest określana na podstawie warunków w nim zawartch. Puste pole jest traktowane jako prawda logiczna. Pole zakończone jest średnikiem
      * Warunek logiczny: Wyrażenie, które przyjmuje dwie wartości prawda lub fałsz. Powstaje dzięki operatorom relacji.
      * Wspierane operatory relacji: <, >, >=, <=. Operatory porównują dwie wartości i zwracają logiczną prawdę, jeżeli porównanie zachodzi lub fałsz gdy nie zachodzi. Porównywać można jedynie wartości liczbowe. Porównanie dwóch wartości, które nie są liczbami będzie skutkowało otrzymaniem logicznego fałszu.
        + Przykład: 5 < 3 (zwróci fałsz), 3< 5 (zwróci prawda)
      * Warunki logiczne można łączyć za pomocą operatorów logicznych. Wspierane operatory logiczne: || logiczny OR, && logiczny AND, ! Logiczny NOT.
        + Przykład 5 < 3 || 3 < 5 (zwróci prawda)
      * Warunki logiczne mogą być grupowane za pomocą nawiasów okrągłych „()”. Wtedy za całe wyrażenie w nawiasach wstawiana jest finalna wartość logiczna, która je prezentuje.
      * Specjalne mechanizmy:
        + Wykonanie reguły tylko, jeżeli inna reguła (musi mieć niższy priorytet została wykonana). Służy do tego następująca konstrukcja: *rule.<id\_reguly\_wykonanej>.executed.* Żeby uzyskać warunek przeciwny wystarczy wstawić „!” przed warunek: *!rule.<id\_reguly\_wykonanej>.executed.*
        + Obserwacja trendów jako warunek wykonania. Można obserwować tylko zmianę danych liczbowy. Do zdefiniowania trendu służy konstrukcja: *<inc/desc> <obser\_wartość> <by> <x> in <number\_of\_days>.* Aby pokazać przykład: *inc stock.CocaCola.value by < 3% in 4.* Interpretacja tego jest: Zwróć prawdę jeżeli w ciągu ostatnich 4 dni całkowita wartość wzrosła o nie więcej niż 3%, w przeciwnym wypadku zwróć fałsz.
    - Ostatnim polem każdej reguły jest pole *action.* Pole to zawiera spis czynności, które mają się wykonać jeżeli spełniony zostanie warunek wejścia, czyli jeżeli po ewaluacji pole *condition* będzie zawierać wartość logicznej prawdy. W przypadku, gdy zawartość pola *condition* zostanie określona jako prawda wykonane zostaną wszystkie możliwe czynności tej reguły.
      * Pole rozpoczyna się słowem *action* zakończonym „ : ”.
      * Pole zawiera listę akcji oddzielonych przecinakami.
      * Budowa pojedynczej akcji:
        + Akcje zaczynają się od czynności: *buy/sell* – oznacza to nabyj lub sprzedaj
        + Potem następuje identyfikacja obiektu do nabycia lub sprzedaży, wykonuje się ją po nazwie. Czyli w przypadku produktów giełdowych: *stock.[<stock\_name>],* natomiast w przypadków walutowych *currency.[*<*currency\_name>]*.
        + W przypadku sprzedaży następnie mogą nastąpić trzy różne konstrukcje:

*part <1-100>:* sprzedaj część swoich swoich odpowiadającą liczbie procentów wskazanej

*amount <liczba/ALL>:* sprzedaj określoną liczbę akcji lub wszystkie w przypadku użycia słowa ALL

*for <liczba dodatnia>:* sprzedaj tyle akcji aby uzyskać z nich co najmniej tyle waluty w jakiej była ta akcja co podana liczba

* + - * + W przypadku nabycia mogą nastąpić następujące konstrukcje:

dla walut:

*amount <liczba dodatnia całkowita/MAX> for <currency.[<currency\_name>]/ANY> :* nabyj określoną ilość waluty za inną walutę lub za dowolną walutę, brana jest wtedy losowo waluta, którą posiadamy, jeżeli jej nie wystarczy to losowana jest kolejna, aż do wyczerpania walut. Słowo kluczowe MAX oznacza nabycie takiej ilości jaka jest tylko możliwa.

dla produktów giełdowych:

*amount <liczba dodatnia całowita/MAX> for <currency.currency<name>/ANY/OWN>:* Kup wskazaną liczbę akcji, MAX - maksymalną możliwą liczbę akcji za wskazaną walutę, ANY – dowolnie wylosowaną walutę, która spełni tą akcję, OWN – walutę przypisaną do produktu akcyjnego.

* + - * + Reguły, dla których *condition* zostatnie spełniony próbują wykonać wszystkie swoje akcje. Reguła uznana jest za wykonaną, nawet, gdy żadna z akcji się nie powiedzie.
    - Dodatkowe uwagi:
      * Dozwolone są komentarze:
        + jednoliniowy rozpoczęty „//”

Przykładowy program:

config [config.xml]

events [events.xml]

start [start.xml]

rule [more\_rule.txt]

rule {

id: 1;

priority:10;

condition: currency.[USD].rate > 1.3;

actions: buy currency.[YUA] amount 10 for currency.[USD],

buy currency.[YEN] amount 1000 for ANY,

buy stock.[NukaCola] amount MAX for OWN;

}

rule {

id: 3;

priority:100;

condition: stock.[CocaCola].value > 150

|| stock.[CocaCola].have.amount > 15;

actions: sell stock.[CocaCola] part 20;

}

IV) Opis Formalny Gramatyki – EBNF

program = configurations, events, starts, rules ;

configurations = configuration | configuration, configurations ;

configuration = „config”, skipable, given\_name, newline;

events = event\_conf | event\_conf, events;

event\_conf = „events”, skipable, given\_name, newline;

starts = start | start, starts;

start = „start”, skipable, given\_name, newline;

rules = rule | rule, rules;

rule = „rule”, skipable, „{” , skipable, id, skipable, priority, skipable, condition, skipable, actions, skipable, „}”;

id = „id”, skipable, „:”, skipable, number, skipable, „;”;

priority = „priority” , skipable, „:”, skipable, number, skipable, „;”;

condition = „condition”, skipable, „:”, skipable, evaluated\_conditions, skipable, „;”;

evaluated\_conditions = [„!”] cond | [„!”] cond, skipable, condition\_join\_operator, skipable, evaluated\_conditions;

condition\_join\_operator = „||” | „&&”;

cond = comparison\_cond | trend\_cond | „(„, evaluated\_conditions, „)”;

comparison\_cond = currency\_numericals [date\_argument], comparison\_oper, number | stock\_numericals [date\_argument], comparison\_oper, number;

currency\_numericals = „currency”, „.”, given\_name, „.”, „rate” | „currency”, „.”, given\_name, „.”, „have”, „.”, „amount”;

stock\_numericals = „stock”, „.”, given\_name, „.”, „value” | „stock”, „.”, given\_name, „.”, „have”, „.”, „amount”;

comparison\_oper = „<” | „>” | „<=” | „>=”;

date\_argument = „(„, number, „)” | „(„, date, „)”;

trend\_cond = inc | desc, watchable\_value, „by”, number\_positive, „in”, number\_positive

actions = action | action, skipable, „,”, skipable, actions;

action = sell\_action | buy\_action;

sell\_action = „sell” , skipable, „stock”, „.”, given\_name, skipable, „part”, number\_le\_100 | „sell” , skipable, „stock”, „.”, given\_name, skipable, „amount”, skipable, number |

„sell” , skipable, „stock”, „.”, given\_name, skipable, amount, skipable, „ALL” |

„sell” , skipable, „stock”, „.”, given\_name, skipable, for, skipable, number\_positive;

buy\_action = buy\_currency\_action | buy\_stock\_action;

buy\_currency\_action = „buy”, skipable, „currency”, „.”, given\_name, skipable, „amount”, skipable, number, skipable, „for”, skipable, number | „buy”, skipable, „currency”, „.”, given\_name, skipable, „amount”, „MAX”, skipable, „for”, skipable, number | „buy”, skipable, „currency”, „.”, given\_name, skipable, „amount”, skipable, number, skipable, „for”, skipable, „ANY” | „buy”, skipable, „currency”, „.”, given\_name, skipable, „amount”, „MAX”, skipable, „for”, skipable, „ANY”;

buy\_stock\_action = buy\_stock\_pre, number, buy\_stock\_mid, „currency”, „.”, given\_name | buy\_stock\_pre, „MAX”, buy\_stock\_mid, „currency”, „.”, given\_name | buy\_stock\_pre, number, buy\_stock\_mid, „ANY” | buy\_stock\_pre, „MAX”, buy\_stock\_mid, „ANY” | buy\_stock\_pre, number, buy\_stock\_mid, „OWN” | buy\_stock\_pre, „MAX”, buy\_stock\_mid, „OWN”;

buy\_stock\_pre = „buy”, skipable, „stock”, „.”, given\_name, skipable, „amount”, skipable;

buy\_stock\_mid = skipable, „for”, skipable;

date = four\_digit\_num, „.”, two\_digit\_num, two\_digit\_num;

four\_digit\_num = digit, digit, digit, digit;

two\_digit\_num = digit, digit;

digit = „0” | „1” | „2” | „3” | „4” | „5” | „6” | „7” | „8” | „9”;

number = [„-”] number\_positive;

number\_positive = digit | digit, number;

newline = „\n”;

given\_name = „[„ , text, „]”;

letter = "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" | "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";

text = letter , text | number, text;

comment = „//”, text, newline;

whitespace = „\t” | „\n” | „ „;

skipable = comment | whitespace | comment, skipable | whitespace, skipable;

V. Realizacja

Zadanie obejmuje realizację dwóch elementów:

* parsera
* leksera

Realizacja zadania:

Projekt będzie zrealizowany z użyciem języka Python. Pliki konfiguracyjne, wydarzeń i startowe zostaną sparsowane za pomocą gotowych bibliotek. Parsowanie języka maszyny zasad jest celem projektu i zostaną zaimplementowane elementy wymienione powyżej.

Działanie programu:

1. Plik programu czytany jest przez lexer, który pobiera z niego kolejne tokeny, które zwraca w postaci typ plus wartość. Na podstawie zwracanych przez lekser tokenów działa parser.

* Lekser składa się z głównego leksera, który deleguje parsowanie konkretnych rodzajów tokenów do podlekserów:
  + DateLexer – lexer, którego zadaniem jest pobieranie tokenów typu datowego, oraz sprawdzenie czy data ma poprawny format. Data rozpoczyna się zawsze symbolem @ i tym samym symbolem się kończy, przy zachowaniu formatu yyyy.MM.dd.
  + GivenNameLexer – lexer pobierający nazwy nadane, czyli nazwy akcji lub skróty walut. Pobiera nazwę i ją zwraca. Nazwy nadane rozpoczynają się „[„ i kończą, „]”.
  + KeywordsLexer – lexer pobierający słowa kluczowe. Słowa kluczowe wyróżnia to, że zaczynają się od litery, zawierają wyłącznie litery. Wszelkie inne znaki powodują zakończenie pobierania.
  + NumberLexer – lexer pobierający tokeny typu liczb
  + OtherCharLexer – lexer pobierający znaki, które nie są białe, ani literami lub cyframi, oraz nie są to [, @.
  + WhitespaceLexer – lexer pobierający tokeny, które rozpoczyna znak biały oraz przetwarzające plik tak długo jak nie pojawi się inny znak.
* Przetwarzanie jest przekazywane do lekserów w zależności od tego jaki znak rozpoczyna rozpoznawanie tokena.
* Gdy lekser dojdzie do końca pliku programu zwraca specjalny token eof.

2. Parser główny dostaje wstrzyknięty Lexer oraz RealityController, czyli obiekt odpowiadający za tworzenie świata na podstawie plików konfiguracyjnych oraz reguł na podstawie programu. Parser działa na podobnej zasadzie co lekser, czyli na podstawie otrzymanego słowa kluczowego kieruje dalszą obsługę do adekwatnych podparserów. W programie wykorzystane są parsery:

* Parser konfiguracji – przejmuje sterowanie, gdy odczytany zostanie token „config”. Korzysta z gotowej biblioteki do parsowania plików XML celem odczytania informacji zawartych w plikach konfiguracyjnych, zapisuje odpowiednie symbole w tabeli symboli oraz wywołuje metody RealityControllera celem budowania świata, w którym będą się odbywać inwestycje.
* Parser zdarzeń – EventsParser – jest to parser, który korzysta z gotowego parsera XML celem pobrania z pliku konfiguracji zdarzeń i umieszczenia ich w świecie poprzez RealityController. Warunkiem wejścia do niego jest znlaezienie tokenu „events”.
* Parser startu – StartParser – jest to parser, który korzysta z gotowego parsera XML celem pobrania z pliku konfiguracji startu odpowiednich wartości startowego stanu majątkowego inwestora. Warunkiem wejścia do niego jest znalezienie tokenu „start”.
* Parser reguł – RuleParser – zajmuje się przetwarzaniem reguł, jest ostatnim parserem, korzysta z pomocniczego parsera warunków (ConditionParser). Parser reguł parsuje kolejne pola budując obiekt reguły, który zawiera identyfikator, priorytet, obiekt warunku i listę obiektów akcji.

3. Silnik

Po zakończeniu parsowania następuje wywołanie silnika. Silnik na podstawie zbudowanego przez parsery świata wykonuje następujące czynności:

1. Wykonuje wszystkie reguły uszeregowane priorytetem.
2. Inkrementuje znacznik dnia i wraca do kroku 1.

Silnik działa od dnia wywołania podanego w argumencie do dnia zakończenia podanego w argumencie.

4. Wykonanie programu

Program wykonywany jest z linii poleceń. Napisany jest w pythonie 3.4. Tak więc aby go wywołać na Ubuntu 14.04 należy wykonać następujące polecenie:

python3 iNvest.py <program\_file> <start\_date\_yyyy.MM.dd> <end\_date\_yyyy.MM.dd>

Pliki programu należy umieścić w tym samym katalogu co plik iNvest.py. Przykładowy program jest już umieszczony w katalogu, aby go wywołać należy użyć komendy:

python3 iNvest.py program.txt 2016.05.10 2016.05.15.

Wynikiem działania jest wydruk, krótkie raportu o stanie końcowym posiadania.