Antoni Charchuła 283713

Projekt z TKOM

Język do generowania wizualizacji

Dokumentacja końcowa

CTML to imperatywny język programowania ułatwiający umieszczanie danych zawartych w pliku CSV do struktur HTML.

Jest to język typowany, który umożliwia m.in tworzenie funkcji, operacje arytmetyczne i logiczne, operacje na tablicach oraz operacje przy użyciu wbudowanych struktur typu *if, if else* oraz *while*. Jednak przede wszystkim zawiera wbudowane funkcje ułatwiające tworzenie struktur języka HTML.

Aby skorzystać z języka, wymagany jest interpreter, który przetworzy napisany program. Jego komunikacja z użytkownikiem polega na wywołaniu załączonego pliku Jar z dwoma argumentami – ścieżką do pliku źródłowego z napisanym kodem programu, który chcemy uruchomić, oraz docelowej ścieżki gdzie ma być zapisany finałowy plik HTML.

Wywołanie:

```
java -jar CTML-1.0.jar ścieżka_pliku_źródłowego ścieżka_pliku_docelowego
```

Program języka CTML umieszczany jest w pliku HTML. Jego kod wyodrębniany jest za pomocą znaczników *<? ?>*. Wszystko poza tymi znacznikami jest przekazywane bez zmian do pliku docelowego.

Przykładowy program zapisany przy użyciu języka CTML:

```
<div>
      <?
            func csv loadFile() {
                  csv file = load(„path/to/file/data.txt”);
                  return file;
            }
            {
                  head(„Hello World!”);
                  csv file = loadFile();
                  int a = 5;
                  while (a > 0) {
                        par(file[0][a]);
                        a = a – 1;
                  }
            }
      ?>
</div>
```

Architektura

Język napisany został przy pomocy języka Java. Kompilacja programu przebiega zgodnie z przedstawionym poniżej schematem:

1. Analiza leksykalna – Lekser

   Wyodrębnienie znaków oraz grupowanie ich w atomy leksykalne.

2. Analiza składniowa i semantyczna – Parser

   Grupowanie tokenów w struktury składniowe. Dodatkowo w trakcie parsowania tworzone są obiekty, na których będzie pracował program. Sprawdzane są czy struktury składniowe mają odpowiednie znaczenie w języku programowania.

3. Wykonanie programu – Program

   Użycie struktur wygenerowanych przez Parser, a następnie wykonanie zbudowanego programu oraz zwrócenie wyniku z preparowanymi danymi w postaci pliku HTML.

W trakcie każdego etapu generowane są odpowiednie wyjątki w przypadku wystąpienia błędu.

Opis składni języka

Tak jak już zostało to wspomniane, program musi zostać zapisany wewnątrz znaczników *<? ?>*. Każdy element umieszczony poza tymi znacznikami zostanie przekierowany do strumienia wyjściowego.

Wewnątrz znaczników można zdefiniować funkcje używając do tego struktury:

```
func TYPE ID( ARGUMENTS ) {
      BODY
}
```

Następnie możemy ją wykorzystać w polu wykonania programu, który jest wyszczególniony nawiasami *{ }*. Ważnym aspektem jest to, że pole wykonania programu musi być po definicjach funkcji.

Ilość struktur *<? .. ?>* w pliku HTML jest nieograniczona. Funkcje z poprzednich wywołań są widoczne w następnych, jednak zmienne już nie.

W polu wykonania programu lub w polu wykonania funkcji, można deklarować zmienne, przypisywać im wartości, używać operacji arytmetycznych, struktur if, if/else oraz while, a w nich operacji logicznych and - && i or ||.

Język CTML obsługuje tylko jednowymiarowe tablice dla typów int, float oraz string. Ich deklaracja polega na dodaniu nawiasów *[]* zaraz po typie np. *int[] a*. Zainicjalizować można je przy pomocy nawiasów oraz argumentów w nich umieszczonych np. *int[] a = {1, 2, 3}*. W trakcie wykonania programu można dodawać do tablicy dodatkowe wartości przy użyciu funkcji append np. *a.append(ARGUMENTY)*. Ilość argumentów jest nieograniczona.

Aby ułatwić pracę z typami, zmienne są automatycznie rzutowane do inicjalizowanej zmiennej. Działa to w przypadku: float -> int, int->float, int->string, float->string. Tablica csv od początku zawiera zmienne typu string, jednak to nie przeszkadza do używania zmiennych w operacjach arytmetycznych lub logicznych.

Zmienne typu csv inicjalizowane są przy pomocy funkcji *load*, która jako argument przyjmuje ścieżkę do pliku csv

```
csv file = load(„directory”);
```

Najważniejszym elementem języka są wbudowane funkcje tworzące struktury HTML. Jest to:

- head(arg) – tworzy heading, w którym umieszcza wartość argumentu

  <h1> Argument’s value </h1>

- par(arg) – tworzy paragraph, w którym umieszcza wartość argumentu

  <p> Argument’s value </h1>

- link(arg1 , arg2) – tworzy link, gdzie pierwszy argument to ścieżka do, której prowadzi link, a drugi treść jaka będzie wyświetlana jako pole do kliknięcia

  * Arg2’s value *

- img(arg) – tworzy image, który wyświetla zdjęcie, do którego prowadzi URL przekazany jako argument

  <img src= Argument’s value >

- List - tworzy listę oraz umieszcza argumenty jako jej elementy.

```
list {
      list_item(arg1);
      list_item(arg2);
}
```

```
<ul>
  <li> Arg1’s value </li>
  <li> Arg2’s value </li>
</ul>
```

- Table - tworzy tablicę umieszczając dane jako kolumnę lub dane kolumny.

```
table {
    row {
        column(arg1);
        column(arg2);
    }
    row {
        table_item(arg3);
        table_item(arg4);
    }
}
```

```
<table>
  <tr>
    <th>Arg1’s value</th>
    <th>Arg2’s value</th>
  </tr>
  <tr>
    <td> Arg3’s value</td>
    <td> Arg4’s value</td>
  </tr>
</table>
```

Gramatyka programu

```
<comment> = '//' all unicode characters ended with 'enter' - '\n';
<htmlContent> = all unicode characters ended with EOF or <ctmlProgram>;
<ctmlProgram> = '<?' [programContent] '?>';
<program> = { <ctmlProgram> | <htmlContent> };
<programContent> = {<functionDeclaration>} [<body>];

<functionDeclaration> = <functionHeader> <functionBody>;
<functionHeader> = 'func' <returnType> <identifier> '(' [<formalParameterList>] ')' ;
<formalParameterList> = <formalParameter> { ',' <formalParameter> };
<formalParameter> = <type> <variableDeclaratorId>;

<functionBody> = <body>;
<body> = '{' {<blockStatement>} '}';
<blockStatement> = <localVariableDeclaration> | <statement>;

<localVariableDeclaration> = <type> <variableDeclaratorId> [<variableInitializer>] ';';
<variableInitializer> = '=' <logicExpression> | <arrayInitializer>;
<arrayInitializer> = '{' [<argumentList>] '}';

<statement> = <ifStatement> | <whileStatement> | <returnStatement> | <ctmlStatement> | <methodOrAssignment>;
<ifStatement> = 'if' '(' <logicExpression> ')' <body> [ 'else' <body> ];
<whileStatement> = 'while' '(' <logicExpression> ')' <body>;
<addToArray> = '.append' '(' <argumentList> ')' ';';
<logicExpression> = <expression> {<logicOperator> <expression>} ;
<expression> = <simpleExpression> [<relationOperator> <simpleExpression>];
<simpleExpression> = <sign> <term> {<addOperator> <term>};
<term> = <factor> { <multOperator> <factor> };
<factor> = <variableOrMethod>| <constant> | '(' <expression> ')';
<relationOperator> = '==' | '<' | '>' | '<=' | '>=' | '!=';
<sign> = '+' | '-' | nothing ;
<addOperator> = '+' | '-';
<multOperator> = '*' | '/';
<logicOperator> = '&&' | '||';
<returnStatement> = 'return' (<logicExpression> | nothing) ';';
<methodOrAssignment> = <identifier> ( <variableInitializer> | <methodInvocation> | <addToArray> );
<methodInvocation> ='(' [<argumentList>] ')' ';';
<variableOrMethod> = <identifier> ( [<indexedVariable>] | <methodInvocation>) ;
<argumentList> = <argument> { ',' <argument>};
<argument> = <variable> | <constant>;

<ctmlStatement> = <simpleCtml> | <structuredCtml> | <link> ;
<simpleCtml> = <simpleCtmlFunction> '(' <argument> ')' ';';
<simpleCtmlFunction> = 'load' | 'head' | 'par' | 'img';
<link> = 'link' '(' <argument> ',' <argument> ')' ';';
<structuredCtml> = <list> | <table>;
<list> = 'List' '{' {<listItem>} '}';
<listItem> = 'list_item' '(' <argument> ')' ';';
<table> = 'Table' '{' { 'Row' '{' <row> ';' '}' } '}';
<row> = 'column' '(' <argument> ')' | 'table_item' '(' <argument> ')';

<type> = 'int' | 'float' | 'string' | 'csv';
<variableDeclaratorId> = <identifier> [ '[' ']' ];
<indexedVariable> = '[' <index> ']' [ '[' <index> ']' ];
<variable> = <identifier> [<indexedVariable>];
<returnType> = <type> | 'void';

<identifier> = letter {letter | <digit> | '_'} ;
<constant> = <numericConst> | <string>;
<string> = '\"' {<stringCharacter>} '\"';
<numericConst> = <integerConst> | <floatConst>;
<integerConst> = <digitWithoutZero> {<digit>};
<digitWithoutZero> = '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
<digit> = '0' | <digitWithoutZero>;
<floatConst> = <digitWithoutZero> {<digit>} '.' {<digit>} ;
<stringCharacter> = all Unicode characters except " and \;
<index> = <variable> | <constant>;
```

Kod źródłowy

Interpreter jest napisany w języku Java. Do uruchamiania testów oraz generowania pliku Jar wykorzystywany jest Maven. Aby to zrobić należy w folderze gdzie znajduje się plik pom.xml uruchomić polecenie:

```
mvn clean install
```

Kod podzielony jest na dwa główne foldery

- interpreter – umieszczony w nim jest interpreter oraz poszczególne elementy, które wywołuje, czyli Lexer, Parser oraz Program. To tutaj znajduje się „Core" programu.
- structures
    - token – pliki źródłowe definiujące Token i możliwe wartości jakie może przyjmować. W pliku TokenType znajduje się enum, który zawiera wszystkie typy tokenów jakie mogą wystąpić w programie, za to plik PredefinedTokens zawiera definicje tych typów
    - model – określa działanie wszystkich elementów stworzonych przez parser, które są wykonywane podczas egzekucji programu. Są tutaj elementy dziedziczące po interfejsie Executable oraz ReturnExecutable, które odpowiednio określają instrukcje, które nie zwracają wartości oraz te które ją zwracają w trakcie wykonania.

Testy automatyczne

Testy są odpowiednio stworzone dla leksera, parsera oraz programu. Testują podstawowe działanie interpretera oraz wyjątki jakie powinny zostać wyrzucone w przypadku wykonania niepoprawnej instrukcji lub zapisania programu w sposób niezgodny z gramatyką.

Aby uruchomić same testy bez tworzenia jar'a należy użyć komendy

```
mvn clean test
```

Dodatkowe uwagi

W katalogu projektu umieszczony jest plik example.html, który jest przykładowym programem wykorzystującym większość funkcjonalności języka, co może być pomocne w przypadku tworzenia nowego kodu. Nie zostały tam umieszczone podstawowe operacje arytmetyczne lub logiczne, ponieważ nie różnią się one od funkcjonalności znanych z innych języków programowania.

Język CTML z pewnością nie jest językiem idealnym – jego gramatyka nie przewiduje wywoływania funkcji w miejscu argumentu innej funkcji czy brak takich struktur jak *else if* lub *for*. Jednak przy użyciu aktualnych narzędzi na pewno da się osiągnąć większość (jak nie wszystko) co można zrobić przy użyciu brakujących możliwości.

Na pewno dużym plusem języka jest łatwa rozszerzalność o nowe struktury – aby dodać nową sturkturę htmlową wystarczy dodać element do modelu oraz zaimplementować jego wykonanie w funkcjach z interfejsu. Dodatkowo należy stworzyć odpowiednie Tokeny, które będą zwracane przez Lekser, a na koniec odpowiednio sparsować strukturę w Parserze w funkcji *parseStatement.*