TKOM Projekt Dokumentacja Jakub Kordel

Przeznaczenie języka:

Przeznaczeniem języka jest możliwość prostego definiowania zasad gier rozgrywanych na prostokątnych planszach takich jak kółko i krzyżyk, warcaby, szachy czy GO. W języku można opisać zasady gry a następnie wykorzystując interpreter języka istnieje możliwość uruchomienia opisanej gry.

Założenia:

Poniżej znajduje się kilka założeń o grze, którą można w języku opisać:

- gra rozgrywana jest na planszy dwuwymiarowej (X na Y). Język pozwala na zdefiniowanie wymiarów planszy gry.
- pole na planszy (identyfikowane dwoma liczbami całkowitymi) może w dowolnym czasie rozgrywki być puste lub zawierać jeden obiekt zdefiniowany przez programistę.
- gra rozgrywana jest pomiędzy dwoma lub więcej graczami, gracze wykonują operacje na planszy podczas swojej tury. Tury graczy odbywają się cyklicznie (gracz 1 -> gracz 2 -> gracz 3). Interpreter języka pozwala na zdefiniowanie liczby graczy.
- gra posiada definiowalny stan początkowy. W języku można zdefiniować układ początkowy planszy.
- gracz w swojej turze może wykonać ruch opisany przez listę dowolnej długości składającej się z dwóch operacji:
    1. clear(x,y) - usunięcie obiektu na polu o współrzędnych x y
    2. place(x, y, obj) - umieszczenie obiektu obj na polu o współrzędnych x y

Funkcjonalność języka:

Język obsługuje komentarze. Komentarze należy umieszczać pomiędzy symbolami /* i */

Interpreter języka wymaga definicji dwóch funkcji:

- define isMoveLegal() -> bool {
  /* tutaj należy zdefiniować warunki tego czy wprowadzona przez gracza lista operacji na planszy może zostać uznana za poprawną */ }
- define end() -> bool { /* tutaj należy zdefiniować warunki zakończenia rozgrywki oraz ustawić zwycięzcę rozgrywki korzystając z wbudowanej w interpreter funkcji setWinner */ }

Język obsługuje typy:

- int – typ całkowity
- float – typ zmiennoprzecinkowy
- bool – typ logiczny prawda/fałsz
- obj – typ obiektów na planszy, jedną domyślną wartością wbudowaną tego typu jest wartość *empt.* Pozostałe wartości, które może przyjmować typ obj powinny być zdefiniowane przez programistę

Język pozwala na definicję własnych funkcji pomocniczych korzystając ze słowa kluczowego define.

Gramatyka

Poniżej (znaleźć można też w pliku gramatyka.txt) znajduje się gramatyka projektowanego języka zapisana w konwencji EBNF:

Symbole nieterminalne

```
Language =                  {definition};


Definition =
                            DefGameParameter
                            | DefStart
                            | DefObject
                            | DefFunction ;


DefGameParameter =          Word, SymbolColon, ParameterValue;


ParameterValue =            Word | Integer, [ SymbolX, Integer ];


DefStart =                  KeyWordStart, StartBlock;


StartBlock =
                            SymbolOpenCurlyBracket,
                            start_block_content,SymbolCloseCurlyBracket;


StartBlockContent =         [SqueresDefinition] ;


SqueresDefinition =         SquereDefinition, {SymbolComma, SquereDefinition};


SquereDefinition =
                            SymbolOpenRoundBracket,Integer,SymbolComma,Integer,
                            SymbolColon,DefinedObjectName, SymbolCloseRoundBracket;
```

```
DefObject =                   KeyWordObj, word;


DefFunction =
                              KeyWordDefine,FunctionName,SymbolOpenRoundBracket,
                        [ParametersList],SymbolCloseRoundBracket,[ReturnTypeDefinition],Block;


ReturnTypeDefinition =        SymbolArrow, TypeName;


ParametersList =
                              FunctionArgumentDeclaration,
                              {SymbolComma,FunctionArgumentDeclaration } ;


FunctionArgumentDeclaration = TypeName, VariableName;


Block =                       SymbolOpenCurlyBracket,BlockContent,SymbolCloseCurlyBracket;


BlockContent =                {Statement};


Statement =                    WhileLoop
                              | IfStatement
                              | LocalVariableDeclaration
                              | ReturnStatement
                              | ValueAssignmentOrFunctionCall ;


ReturnStatement =             KeyWordReturn, Expression, SymbolSemicolon;


ValueAssignmentOrFunctionCall=       Word,((SymbolAssign,Expression)|
                                            ArgumentsClause), SymbolSemicolon;
```

```
LocalVariableDeclaration =    TypeName, Word, SymbolAssign, Expression, SymbolSemicolon;

WhileLoop =                   KeyWordWhile,SymbolOpenRoundBracket,Expression,
                              SymbolCloseRoundBracket, Block;

IfStatement =                 IfHeader, Block, [ElseClause];

ElseClause =                  KeyWordElse, (Block | IfStatement);

IfHeader =                    KeyWordIf,SymbolOpenRoundBracket,Expression,
                              SymbolCloseRoundBracket;

ArgumentsClause =             SymbolOpenRoundBracket,[ArgumentsList],SymbolCloseRoundBracket;

ArgumentsList =               Expression , {SymbolComma, Expression };

Expression =                  AndExpression, { SymbolOr, AndExpression};

AndExpression =                       EqualityExpression,  {  SymbolAnd,  EqualityExpression};

EqualityExpression =                  RelationalExpression,
                                      [(SymbolEqual), RelationalExpression];

RelationalExpression =                AddSubstractExpression,
                                      [SymbolRelational, AddSubstractExpression];

AddSubstractExpression =              Term, {( SymbolAdd | SymbolSubstract ), Term };
Term =                                Primary, {(SymbolMultiply | SymbolDivide), Primary};
```

```
Primary =                       [SymbolSubstract | SymbolNegation], (  Integer
                                                                     | FloatNumber
                                                                     | KeyWordTrue
                                                                     | KeyWordFalse
                                                                     | ParenthExpression
                                                                     | Identificator);

ParenthExpression =             SymbolOpenRoundBracket,Expression,
                                SymbolCloseRoundBracket

Identificator =                 word, [ArgumentsClause | SquereBrackets];

SquereBracket =                 SymbolOpenSquereBracket,
                                Expression, SymbolCloseSquereBracket;


SquereBrackets =                SquereBracket, {SquereBracket}


Integer =                       NonZeroDigit, {Digit}
                                | Zero
                                ;

Word =                          Letter, { Letter | Digit };


Letter =                        LowerCaseLetter | UpperCaseLetter ;


Number =                        Integer | FloatNumber;


FloatNumber =                   Integer, SymbolDot, Digit, {Digit};
```

```
Digit =                          Zero | NonZeroDigit ;


SymbolRelational =               SymbolLess
                                 | SymbolGreater
                                 | SymbolLessEqual
                                 | SymbolGreaterEqual
                                 ;
TypeName =
                                 KeyWordInt
                                 | KeyWordFloat
                                 | KeyWordBool
                                 | KeyWordObj
Symbole terminalne
SymbolNegation =                 "!";



SymbolAnd =                      "&&" ;
SymbolOr =                       "||";

SymbolLess =                     "<";
SymbolGreater =                  ">";
SymbolLessEqual       =          "<=";
SymbolGreaterEqual =             ">=";
SymbolEqual    =                 "==";
SymbolNotEqual =                 "!=";


KeyWordInt =                     "int"
KeyWordFloat =                   "float"
KeyWordBool =                    „bool"
KeyWordObj =                     "obj"
KeyWordTrue =                    "true";
KeyWordFalse =                   "false";
KeyWordElse =                    "else";
KeyWordIf =                      "if";
KeyWordWhile =                   "while";
KeyWordDefine =                  "define";
```

```
KeyWordReturn =                     "return";
KeyWordObj =                        "obj";
KeyWordStart =                      "start";

SymbolOpenRoundBracket =            "(";
SymbolCloseRoundBracket =           ")";
SymbolOpenCurlyBracket =            "{";
SymbolCloseCurlyBracket =           "}";
SymbolOpenSquereBracket =           "[";
SymbolCloseSquereBracket=           "]";

SymbolSemicolon =                   ";";
SymbolArrow =                       "->";
SymbolDot =                         ".";
SymbolColon =                       ":";
SymbolComma =                       ",";

Zero =                              "0";

SymbolAdd =                         "+";
SymbolSubstract =                   "-";
SymbolMulptiply =                   "*";
SymbolDivide =                      "/";

SymbolAssign =                      "=";

SymbolX =                           "x";

LowerCaseLetter =                   "a" | "b" | "c" | ... | "z" ;
UpperCaseLetter =                   "A" | "B" | "C" | ... | "Z" ;
NonZeroDigit =                      "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

Przykład

Poniżej znajduje się przykład gry kółko i krzyżyk w pełni uruchamialnej przez interpreter języka. Przykład można też znaleźć w pliku *tictactoe.gdef.txt*.

```
name: tictactoe
players: 2
size: 3 x 3
```

```
	obj X

	obj O


	/*definicja ukladu poczatkowego */

	start {

		/* definiuje uklad poczatkowy planszy */

		(0,0: empty), (0,1: empty), (0,2: empty),

		(1,0: empty), (1,1: empty), (1,2: empty),

		(2,0: empty), (2,2: empty)


		/*pola tutaj niezdefiniowane bedą miały domyslna zawartosc empty np. pole o wsp
2,1 */

	}


	define isMoveLegal() ->bool {

		if (operationsNum() != 1 || !isPlaceOperation(0)){  /* argumentem funkcji
wbudowanej isPlaceOperation jest numer operacji z listy operacji ktore wykonal gracz w swojej
kolejce*/

			return false;

		}

		int xCord = OperationList[0][0];

		int yCord = OperationList[0][1];

		obj placedObj = OperationList[0][2];


		if (CurrentPosition[xCord][yCord] != empty || onTurn() == 0 && placedObj != X
|| onTurn() == 1 && placedObj != O){

			return false;

		}


		return true;

	}


	define end() ->bool{

		/* definiuje funkcje sprawdzajaca czy pozycja gry jest terminalna */

		if ( isXWinner() ){


			setWinner(0);


			return true;

		} else if ( isOWinner() ){


			setWinner(1);
```

```
                return true;
        } else {
                return false;
        }
}


define isWinner(obj object) -> bool {
        int collumn = 0;
        int row = 0;
        bool flag = true;

        while(collumn < 3){
                while(row < 3){
                        if(CurrentPosition[row][collumn] != object ){
                                flag = false;
                        }
                        row = row + 1;
                }

                if(flag == true){
                        return true;
                }
                flag = true;
                collumn = collumn + 1;
                row = 0;
        }

        collumn = 0;

        while(row < 3){
                while(collumn < 3){
                        if(CurrentPosition[row][collumn] != object ){
                                flag = false;
                        }
                        collumn = collumn + 1;
                }
                if(flag){
                        return true;
                }
```

```
                        flag = true;
                        row = row + 1;
                        collumn = 0;
                }


                if (CurrentPosition[0][0] == object && CurrentPosition[1][1] == object &&
CurrentPosition[2][2] == object ){
                        return true;
                } else
                if (CurrentPosition[0][2] == object && CurrentPosition[1][1] == object &&
CurrentPosition[2][0] == object ){
                        return true;
                }
                else {
                        return false;
                }
        }


        define isXWinner() -> bool {
                return isWinner(X);
        }


        define isOWinner() -> bool {
                return isWinner(O);
        }
```

Działanie języka

- Język oferuje typy podstawowe: int, float, bool, obj (definiowalne obiekty na planszy),
- Język obsługuje podstawowe działania arytmetyczne (+, -, *, /) na danych liczbowych typu int oraz float. W tym uwzględnia priorytety nawiasowania, oraz wyższości * i / nad + i -,
- Język pozwala na umieszczenie komentarzy przy użyciu symbolu otwierającego komentarz /* oraz zamykającego */. Komentarze nie mogą znajdować się w obrębi nazw zmiennych, funkcji, typów itd.,
- Język umożliwia deklaracje własnych zmiennych lokalnych. Zadeklarowane zmienne lokalne widoczne są tylko w obrębie funkcji, w której zostały zadeklarowane. Istnieje możliwość przypisywania do nich wartości,
- typowanie dynamiczne, nadawanie typów zmiennym podczas interpretacji języka,
- wartości zmiennych można nadpisywać wielokrotnie,
- Język oferuje standardową pętlę while oraz konstrukcję warunkową if.

Opis wbudowanej funkcjonalności oferowanej przez interpreter

Funkcje wbudowane:

- operationsNum() – zwraca liczbę operacji w aktualnie analizowanym ruchu gracza,

- isPlaceOperation(*numer operacji*) – zwraca prawdę jeżeli operacja o numerze podanym w argumencie jest operacją place, a w przeciwnym przypadku (operacja o podanym numerze jest operacją clear) zwróci fałsz, zazwyczaj należy poprzedzić sprawdzeniem liczby operacji przy użyciu operationsNum(),
- onTurn() – zwraca numer gracza, do którego aktualnie należy kolejka, gracze numerowani od zera,
- setWinner(*numer gracza*) – ustawia gracza o numerze podanym w argumencie jako zwycięzcę rozgrywki,

Wbudowane wyrażenia tablicowe:

- CurrentPosition[*x*][*y*] – zwraca obiekt (wartość typu obj) aktualnie umieszczony na polu o współrzędnych x i y.
- OperationList[*numer operacji w liscie ruchu*][*numer parametru operacji*] – najłatwiej wyjaśnić na przykładzie: załóżmy, że gracz wykonał w swoim ruchu dwie operacje: operację clear(0, 0) oraz operację place(7, 7, Queen). Aby w programie odwołać się do obiektu umieszczonym w drugiej operacji należy napisać konstrukcję: OperationList[1][2], w pierwszym nawiasie jest 1, ponieważ odwołujemy się do drugiej operacji (numeracja jest od zera) , w drugim nawiasie jest liczba 2 ponieważ odwołujemy się do trzeciego argumentu funkcji place. Odwoływanie się do tych parametrów należy zwykle poprzedzić sprawdzeniem isPlaceOperation(*numer operacji*).

Budowanie interpretera

Projekt został wykonany w języku c++.

Budowanie interpretera przy pomocy narzędzia CMake.

- System Windows:

W głównym katalogu projektu należy wywołać polecenia:

```
cmake -B build
cmake --build build --config Release
```

- System linux:

W głównym katalogu:

```
cmake -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build
```

Interpreter o nazwie rungame.exe zostanie utworzony w katalogu build/Release

Produkt końcowy

Program otrzymuje na wejściu plik z definicją gry w zaprojektowanym języku i na jego podstawie rysuje w konsoli pozycje początkową gry. Następnie program wczytuje kolejne ruchy graczy (pojedynczym ruchem gracza jest lista operacji clear i place) i na podstawie pliku opisującego zasady gry ocenia czy ruch jest poprawny. Jeżeli ruch jest poprawny to pozycja gry na ekranie jest aktualizowana, jeżeli nie to program wyświetla odpowiedni komunikat i pyta o ponowne wczytanie sekwencji operacji. Po każdym ruchu program sprawdza czy spełniony jest warunek zakończenia rozgrywki, który również jest opisany przy pomocy zaprojektowanego języka w pliku wejściowym. Pliki wejściowe z kodem w zaprojektowanym języku posiadać będą rozszerzenie .gdef. Np. plik z definicją gry w kółko i krzyżyk, może widnieć pod nazwą tictactoe.gdef Przykład polecenia

uruchamiającego program: ./rungame tictactoe.gdef Zdefiniowane funkcje pomocnicze mogą znajdować się w różnych plikach z rozszerzeniem gdef. W przypadku użycia kilku plików uruchomienie może wyglądać tak: ./rungame tictactoe.gdef funpom1.gdef funpom2.gdef