Gear language

An interpreter written in Free Pascal using an external tree visitor pattern

J. de Haan
October 2018
(All quotes are from www.brainyquote.com)

Index

Chapter 1 – The basics – a calculator

This is a no-nonsense book about creating an interpreted programming language, named "Gear", using an external visitor pattern. Why Gear? No particular reason, I just like the sound of it…

In this book I won't describe a lot of theory, as there are plenty of good books out there. As a matter of fact, I do recommend reading a few of them, such as:

- Writing Compilers and Interpreters, A software emgineering approach (Ronald Mak)
- Language Implementation Patterns (Terence Parr)

Then, there are hands-on websites/blogs that describe and create an interpreter on-the-go:

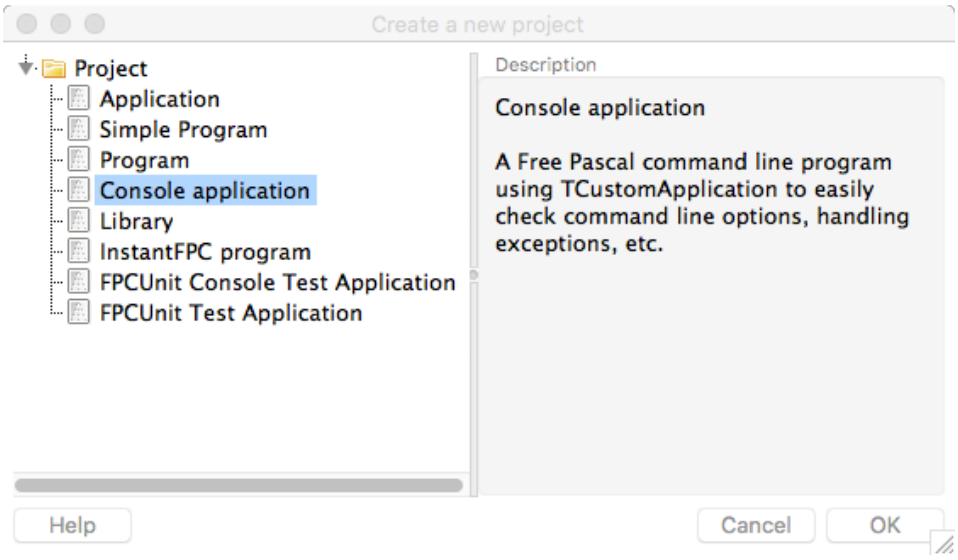
- Let's build a simple interpreter (https://ruslanspivak.com/lsbasi-part1/) (Ruslan Spivak)
- Crafting Interpreters (http://craftinginterpreters.com) (Bob Nystrom)

These books/blogs use Java or Python as programming language.

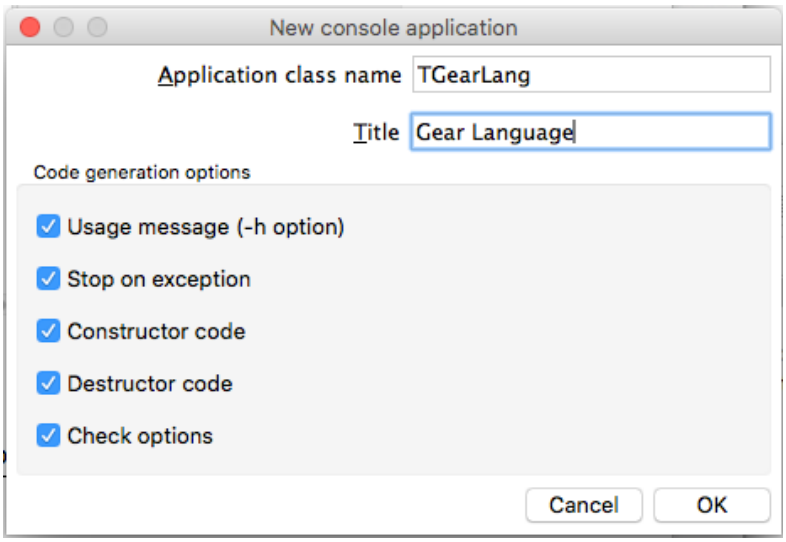
The code in this book is based on the programming language Object Pascal, a modern pascal version, and the free downloadable IDE 'Lazarus'. Lazarus/Free Pascal is available for Windows, Linux, OSX, and many other Operatig Systems. It's slogan is: 'Write once, compile anywhere'! The website and download links are available at: http://www.lazarus-ide.org/index.php. For this book I use version 1.8.4 (or later) of Lazarus and version 3.04 (or later) of Free Pascal.

Without further ado, let's start this project! By the end of the book you'll be an Object Pascal guru and have sound knowledge of Lazarus. Steps to take:

1. Download and install the latest Lazarus/Free Pascal version for your OS.
2. Start Lazarus and create a new project. Choose 'Console Application' and click 'Ok'.



In the next window change the Application class name to TGearLang and the title to 'Gear Language'.



Click on OK and a new project is created with one file with title 'project1.lpr' (lpr = Lazarus Project) and contents:

```
program Project1;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils, CustApp
  { you can add units after this };

type

  { TGearLang }

  TGearLang = class(TCustomApplication)
  protected
    procedure DoRun; override;
  public
    constructor Create(TheOwner: TComponent); override;
    destructor Destroy; override;
    procedure WriteHelp; virtual;
  end;

{ TGearLang }

procedure TGearLang.DoRun;
var
  ErrorMsg: String;
begin
  // quick check parameters
  ErrorMsg := CheckOptions('h', 'help');
  if ErrorMsg <> '' then begin
```

```
    ShowException(Exception.Create(ErrorMsg));
    Terminate;
    Exit;
  end;

  // parse parameters
  if HasOption('h', 'help') then begin
    WriteHelp;
    Terminate;
    Exit;
  end;

  { add your program here }

  // stop program loop
  Terminate;
end;

constructor TGearLang.Create(TheOwner: TComponent);
begin
  inherited Create(TheOwner);
  StopOnException := True;
end;

destructor TGearLang.Destroy;
begin
  inherited Destroy;
end;

procedure TGearLang.WriteHelp;
begin
  { add your help code here }
  writeln('Usage: ', ExeName, ' -h');
end;

var
  Application: TGearLang;
begin
  Application := TGearLang.Create(nil);
  Application.Title := 'Gear Language';
  Application.Run;
  Application.Free;
end.
```

3. Save the project in a new folder, e.g. ProjectGear/Chapter01. I named my project 'gear.lpr'.

The code should now read:

```
program Gear;

...
```

⇨ In Pascal it is common to make the first letter of an identifier a capital.

In this chapter we'll create a working calculator, which will be the basis for the next chapters. In fact, every new chapter builds upon the previous one. Usually, I work top-down, which means I first create the main program and from there I go deeper.

Type Gear to start

In this paragraph we'll set up the basics, for our calculator/interpreter. All that follows is more or less extending the code in the cradle we're building in this chapter.

> I have one speed, I have one gear: go!
> — Charlie Sheen

```
program Gear;

{$mode objfpc}{$H+}
```

If you don't know Free Pascal/Object Pascal, there's something to explain here. Free Pascal has different compiler modes, like 'delphi' and 'objfpc'. I tend to use mode objfpc, but of course you're free to choose your preferred mode. The compiler directive {H+} makes sure we use ansistrings, which can hold an unlimited number of characters (well in theory the computer memory is the limit). As we are processing the input program as a string later on, we need to work with large strings.

```
uses
   SysUtils, uLanguage;
```

Then, the socalled 'uses' clause, which lets you use functionality (functions and procedures, classes, constants and variables) from other units. A unit is a file with independent functionality that can be compiled separately. In this case unit SysUtils is one of Free Pascal's standard units. However, unit 'uLanguage' is one we are going to create ourselves shortly. Usually, I put a 'u' in front of unit names.

The only procedure we have to change in the main program (gear.lpr), is procedure procedure TGearLang.DoRun. I will go through it step by step.

```
procedure TGearLang.DoRun;
var
  ErrorMsg: String;
  InputFile: String='';
  FileNeeded: Boolean = False;
```

I added two variables, InputFile and FileNeeded. Variable 'InputFile' will contain the file that we wish to parse and execute, or it will contain the code we process at the Gear prompt.

Variable FileNeeded is a Boolean variable which starts of with value 'False', but will be true in case we wish to print the AST, or execute, or compile a file.

```
begin
  // quick check parameters
  ErrorMsg := CheckOptions('hxcaf:', 'help execute compile ast file:');
```

Function CheckOptions checks whether the parameter options passed are accepted options. If not an error message is generated. The error message is shown and the program is terminated.

```
  if ErrorMsg <> '' then begin
    ShowException(Exception.Create(ErrorMsg));
    Terminate;
    Exit;
  end;
```

Next, we start checking which options are used. The first one is to check for option '-h' or '--help'. If this option is used, the help text is printed and the program is terminated.

```
  // parse parameters
  if HasOption('h', 'help') then begin
    WriteHelp;
    Terminate;
    Exit;
  end;
```

Then, we check if we need to include a file. We require a file to be included if option -a, -x or -c are used. If we require a file, but it is not passed in the command line, we generate an error.

```
  { File and filename required for a, x, c }
  FileNeeded := HasOption('a', 'ast') or       // print AST
                HasOption('x', 'execute') or   // execute file
                HasOption('c', 'compile');     // compile file

  if FileNeeded and not HasOption('f', 'file') then begin
    ShowException(Exception.Create('Input file name is required.'));
    Terminate;
    Exit;
  end;
```

A call to the executable that requires a file to be included looks like the following:

- gear -x -f filename.gear

This executes (interprets) the file with the given name. Of course the file must exist, and the extension of the file must be '.gear'.

```
  if HasOption('f', 'file') then begin
    InputFile := GetOptionValue('f', 'file');
    if not FileExists(InputFile) then begin
      ShowException(Exception.Create('Input file does not exist.'));
      Terminate;
      Exit;
    end;
    if not (ExtractFileExt(InputFile) = '.gear') then begin
      ShowException(Exception.Create('File extension must be ".gear".'));
      Terminate;
      Exit;
    end;
  end;
```

Finally, we come to the processing of the input file. If the option -a (or --ast) is used we will parse the file and print its AST. I choose to do it in the console window, but of course you can write it to a file as well.

```
  { Execute a file }
  if HasOption('a', 'ast') then
    Language.ExecutePrintAST(InputFile)
```

If option -x (or --execute) is used we will interpret and execute the file. If -c (or --compile) is used the compiler will be used to create intermediary code. If none of -a, -x or -c are used we will start the REPL, in which you can type in commands directly.

```
  else if HasOption('x', 'execute') then
    Language.ExecuteFromFile(InputFile)
  else if HasOption('c', 'compile') then
    Language.CompileFromFile(InputFile)
  else  // gear REPL started
    Language.ExecuteFromPrompt;

  // stop program loop
  Terminate;
end;
```

If none of the options -a, -x, or -c are used, the -f option is ignored, and the REPL is started.
For completeness I show the WriteHelp procedure:

```
procedure TGearLang.WriteHelp;
begin
  { add your help code here }
  writeln('Usage: ', ExtractFileName(ExeName), ' -h -(a|x|c) -f filename.gear');
  writeln('Option:    -h --help               Show help');
  writeln('Option:    -a --ast                Print AST');
  writeln('Option:    -x --execute            Execute product');
  writeln('Option:    -c --compile            Compile product');
  writeln('Required:  -f --file= filename     Input product');
  writeln('No parameters: start REPL');
end;
```

There are two ways to run a Gear program, one is by typing in code directly at the prompt 'Gear>', and the other one is by calling Gear followed by a file name, or by a compiler directive and a file name. Here are some examples:

```
> gear <enter>
Gear REPL v0.1 - (c) J. de Haan 2018
Gear> 12*3^3
324
Gear> quit
>
```

File input.txt contains 1 expression: 12 * 3^3

```
> gear -x -f input.gear
Gear Interpreter v0.1 - (c) J. de Haan 2018

324
>
```

And, finally, for the same input file, but now with -a -f in the call:

```
> gear -a -f input.gear
Gear AST v0.1 - (c) J. de Haan 2018

    (*)
      12
      (^)
        3
        3
>
```

Execute the language

> In art, the hand can never execute anything higher than the heart can imagine.
> — Ralph Waldo Emerson

We continue with actually executing our language. That is to say, we show the code here ☺. It takes a while to be ready to start executing the real thing. But bear with me!

Create a new unit in the same folder and name it 'uLanguage.pas'. Again I'll go through it step by step and use the lines around the code to show the beginning, middle and end part.

```
unit uLanguage;

{$mode objfpc}{$H+}
{$modeswitch advancedrecords}

interface
uses
  Classes, SysUtils, uReader, uLexer, uParser, uError, uAST,
  uPrinter, uInterpreter;
```

A unit starts with the keyword 'unit'. Then, again we have to put the objfpc mode there, and introduce a new modeswitch command stating that we'll allow the record type to also include procedures and functions. The 'interface' section is what the outside world (other units or the main program) may use.

```
type
  Language = record
    private
      class procedure PrintAST(Tree: TExpr); static;
      class procedure Execute(const Source: String; InputType: TInputType); static;
    public
      class var Interpreter: TInterpreter;
      class procedure ExecuteFromFile(const Source: String); static;
      class procedure ExecuteFromPrompt; static;
      class procedure CompileFromFile(const Source: String); static;
      class procedure ExecutePrintAST(const Source: String); static;
  end;
```

We don't need to create an instance of record Language, hence we define the variable Interpreter and the procedures with the keyword 'class'. In the main program you saw we used

```
Language.ExecuteFromFile(SourcePath, Flags);
```

⇨ Usually defined types start with the capital 'T', as a coding convention. Contrary to languages like Java, in Pascal there is no difference between lower case and upper case letters. So, for example 'language' and 'Language' are exacly the same. That's why we use in Pascal a T in front of a type definition. By the way, Language is a special case, which doesn't need a 'T'.

Next, we define a constant GearVersion.

```
const
  GearVersion = 'v0.1';
```

The keyword that separates the outer from the inner world is 'implementation'. Everything defined and declared in the implementation section is not visible to the outside world.

```
implementation

class procedure Language.ExecuteFromFile(const Source: String);
begin
  WriteLn('Gear Interpreter ', GearVersion, ' - (c) J. de Haan 2018', LineEnding);
  Language.Execute(Source, itFile);
end;
```

Procedure ExecuteFromFile simply calls Language.Execute with parameters Source and the constant itFile (which stands for input type File). The other way of executing an input program is through the prompt:

```
class procedure Language.ExecuteFromPrompt;
var
  Source: String = '';
  Quit: Boolean = False;
begin
  WriteLn('Gear REPL ', GearVersion, ' - (c) J. de Haan 2018', LineEnding);
  while not Quit do begin
    Write('Gear> ');
    ReadLn(Source);
    Quit := LowerCase(Source) = 'quit';
    if not Quit then
      Language.Execute(Source, itPrompt);
    Errors.Reset;
  end;
end;
```

We continue to read input from the prompt until the user enters 'quit'. With the given input we call again Language.Execute, but now we don't provide any flags and the input type is itPrompt. This makes sure the interpreter processes the Source as the input code, instead of processing it as a file name.

There's one other Language operation, and that's 'compile'. Since we don't offer that yet, the given procedure is a stub for now:

```
class procedure Language.CompileFromFile(const Source: String);
begin
  WriteLn('Gear Compiler ', GearVersion, ' - (c) J. de Haan 2018', LineEnding);
  WriteLn('Not implemented yet.', LineEnding);
end;
```

Now, the real execution of the prompt and the interpreter takes place in Language.Execute. The below procedure is the actual workhorse, as it performs all the actions needed to read, lexically analyze, parse, print and interpret the input code. It looks simple, but it means we have a lot of code to create. On the other hand, once we have this, all other stuff will be merely additions to the cradle code.

We first define a variable Parser and a variable Tree. The parser takes as input a Lexer, which in turn has as input a Reader. The Reader takes a Source and an input type: file or prompt.

```
class procedure Language.Execute(const Source: String; InputType: TInputType);
var
  Parser: TParser;
  Tree: TExpr = Nil;
begin
  try
    Parser := TParser.Create(TLexer.Create(TReader.Create(Source, InputType)));
    Tree := Parser.Parse;
    if not Errors.isEmpty then
      WriteLn(Errors.toString)
    else
      Interpreter.Execute(Tree);
  finally
    if Assigned(Tree) then Tree.Free;
    Parser.Free;
  end;
end;
```

Then, we have a try…finally statement. The finally block is always executed, no matter what happens. I use it here for freeing the memory of objects, if they're assigned, meaning non Nil.

In the Try part, we first read the source file text into the Reader, which is then input to the lexical analyzer, or Lexer, which produces a list of Tokens. The Lexer is input to the Parser. We start simple and parse only a single expression. We'll extend this in the next chapter.

Then, we check for any parser errors, and if none, we continue interpreting the AST. If there were errors, we'll nicely print them.

```
class procedure Language.ExecutePrintAST(const Source: String);
var
  Parser: TParser;
  Tree: TExpr;
begin
  WriteLn('Gear AST ', GearVersion, ' - (c) J. de Haan 2018', LineEnding);
  try
    Parser := TParser.Create(TLexer.Create(TReader.Create(Source, itFile)));
    Tree := Parser.Parse;
    if not Errors.isEmpty then
      WriteLn(Errors.toString);
    PrintAST(Tree);

  finally
    Parser.Free;
    if Assigned(Tree) then Tree.Free;
  end;
end;
```

If we want to print the AST, we first have to parse it, similarly as when we interpret it. If any parse errors, we'll print them, and we print the AST, using below procedure.

Below routine prints the contents of the AST. Note that it contains a try…except statement inside a try…finally statement. If for any reason the tree cannot be printed, we show the reason why.

```
class procedure Language.PrintAST(Tree: TExpr);
var
  Printer: TPrinter = Nil;
begin
  try
    try
      Printer := TPrinter.Create(Tree);
      Printer.Print;
    except
      on E: Exception do begin
        Writeln('Unable to print the AST due to:');
        Writeln(E.Message);
      end;
    end;
  finally
    if Assigned(Printer) then Printer.Free;
  end;
end;
```

And, to finalyze this unit, we'll introduce two more sections in the unit: initialization and finalization.

```
initialization
  Language.Interpreter := TInterpreter.Create;

finalization
  Language.Interpreter.Free;

end.
```

On initialization we'll create and set up the interpreter object, and in the finalization (or end of the program) we clean up any memory allocation.

Read the input source

As you saw in procedure 'Language.Execute' it all starts with reading the input text, or rather the program that you wish to execute. Let's create a reader for that. Create a new unit uReader.pas, and add the following code.

> Creativity requires input, and that's what research is. You're gathering material with which to build.
> — Gene Luen Yang

```
unit uReader;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;

const
  {$IFDEF UNIX}
    FileEnding = ^D;
  {$ENDIF}
  {$IFDEF WINDOWS}
    FileEnding = ^Z (#26)
  {$ENDIF}
```

Note the constant FileEnding, which differs per operating system. On Unix for example it is ^D, while on Windows it's ^Z.

```
type

  TInputType = (itPrompt, itFile);

  TReader = class
    private
      FFileName: TFileName;
      FSource: string;
      FIndex: LongInt;
      FCount: LongInt;
      function getPeekChar: char;
    public
      property Count: LongInt read FCount;
      property Index: LongInt read FIndex;
      property FileName: TFileName read FFileName;
      property PeekChar: char read getPeekChar;
      constructor Create(Source: String; InputType: TInputType);
      function NextChar: char;
  end;
```

First, the enumeration type TInputType is defined, which contains itPrompt and itFile, in order to distinguish between input via the prompt or input from a file. Next, the TReader type, which is defined here as a class. However, we could've used a record instead as well.

⇨ Usually, the initials of the enum type name are prepended to the enum values. 'it' stands for InputType.

TReader's most important stuff for now are function NextChar and property PeekChar. NextChar delivers the next character to the Lexer, whereas PeekChar looks ahead one character. Property FileName might be of later use, when we discuss import of libraries.

The implementation is as follows:

```
implementation

constructor TReader.Create(Source: String; InputType: TInputType);
var
  SourceCode: TStringList = Nil;
begin
  FFileName := '';
  FIndex := 1;
  case InputType of
    itPrompt: FSource := Source;
    itFile:
      try
        FFileName := Source;
        SourceCode := TStringList.Create;
        SourceCode.LoadFromFile(FFileName);
        FSource := SourceCode.Text;
      finally
        if Assigned(SourceCode) then SourceCode.Free;
      end;
  end;
  FCount := FSource.Length;
end;
```

If the input type is itPrompt then variable Source contains the input text. On the other hand if the input type is itFile, variable Source will be a file name. We then create a string list SourceCode and use its standard routine LoadFromFile to read the contents from a folder. A TStringList variable contains a property Text, which contains the complete input text.

This means, in either case variable FSource contains the input text. Finally, we set variable FCount to be the length of the source.

```
function TReader.NextChar: char;
begin
  try
    Result := FSource[FIndex];
    Inc(FIndex);
  except
    Result := FileEnding;
  end;
end;

function TReader.getPeekChar: char;  //peek at next character, but don't process it
begin
  try
    Result := FSource[FIndex];
  except
    Result := FileEnding;
  end;
end;

end.
```

Function NextChar reads the character at location FIndex and then moves the index one character ahead. If any error occurs, or we try to read past the end of the input we return a FileEnding.

PeekChar on the other hand reads the character at location FIndex, however, doesn't move the pointer.

We have to go one further step before we can move to lexical analysis.

Tokens are the language

This is where we define the language in terms of keywords, operators, comments and punctuation marks. A token contains information about which token type is encountered, its lexeme, a location and a value if it is a number, string or boolean.

> What I value in books is lucidity. I want the language to be rich; I love lexical fireworks on the page, but I have to know what it means. I want to be surprised and delighted, not merely baffled.
>
> — Mal Peet

Create a new unit and name it uToken.pas.

```
unit uToken;
{$mode objfpc}{$H+}
{$modeswitch typehelpers}

interface

uses
  Classes, SysUtils, uCollections, Variants;
```

We introduce a new mode switch, called typehelpers. This mode switch makes it possible to extend types with additional functionality. Also note, in the uses clause, the unit 'collections'. This unit contains a few standard generic collection and dictionary types, and can be found in the appendix.

```
type

  TTokenTyp = (
    //Expressions - operators
    ttPlus, ttMin, ttMul, ttDiv, ttRem,
    ttPlusIs, ttMinIs, ttMulIs, ttDivIs, ttRemIs,
    ttOr, ttAnd, ttNot, ttXor,
    ttShl, ttShr, ttPow,
    ttEQ, ttNEQ, ttGT, ttGE, ttLT, ttLE,

    //Keywords declarations
    ttArray, ttClass, ttDictionary, ttEach, ttEnum, ttExtension, ttFunc,
    ttInit, ttLet, ttVal, ttVar, ttTrait,
    //Keywords statements and expressions
    ttIf, ttThen, ttElse, ttWhile, ttDo, ttRepeat, ttUntil,
    ttFor, ttIn, ttIs, ttReturn, ttEnd, ttMatch, ttWhere, ttSwitch, ttCase,
    ttEnsure, ttPrint, ttInherited, ttSelf, ttUse, ttBreak, ttOn,
    ttIdentifier,

    //Constant values
    ttFalse, ttTrue, ttNull, ttNumber, ttString, ttChar,

    //Symbols and punctuation marks
    ttComma, ttDot, ttDotDot, ttAssign, ttQuestion, ttArrow, ttColon,
    ttOpenParen, ttCloseParen, ttOpenBrace, ttCloseBrace,
    ttOpenBrack, ttCloseBrack, ttComment, ttEOF, ttNone
  );

  TTokenTypHelper = type helper for TTokenTyp
    function toString: string;
  end;

  TTokenTypSet = set of TTokenTyp;
```

Type TTokenTyp is an enumeration type. It is far less complex then the comparable types in Java or Swift. In fact, it can only contain the enumerated items and nothing else (well in fact you can assign numbers to them, but not strings for example). All items have 'tt' in front of them (from TokenTyp).

The token types are used thoughout the application. They are grouped in operators on expressions, keywords for declarations and for statements, constant values and other symbols. We will discuss all of them when we get to it. For now, this is the set we'll be using, but on the way in this journey we might add additional token types.

The expression operators are fairly standard. Operations on integer and float numbers are allowed, such as plus, minus, multiply and divide as the basic operators. Next to that it is allowed to use shift operators, << for left shift and >> for right shift. The remainder % operation is supported, as well as the power ^. And of course the boolean operators are there as well.

Our language supports a number of common statements, such as

- For .. do loops
- While … do loops
- Repeat … until loops
- Switch … case statements
- If … else conditional
- Match selection

Types will be discussed in more detail in another chapter, but I wish to support at least arrays, dictionaries, enumerations, ranges and classes. Classes are compound types that can have variables, properties, methods and initializers.

Next, you see type TokenTypHelper, which extends TTokenTyp with a helper method: toString, which indeed creates a string of a token type.

Also, there is a set type: TTokenTypSet. This set will be used for synchronizing the parser if we have to recover from a parser error.

Next we'll define the actual Token class. Remember, the Lexer will produce Tokens, which will be given to the Parser for processing.

```
TToken = class
  private
    FTyp: TTokenTyp;
    FLexeme: String;
    FValue: Variant;
    FLine, FCol: LongInt;
  public
    property Typ: TTokenTyp read FTyp;
    property Lexeme: String read FLexeme;
    property Value: Variant read FValue;
    property Line: LongInt read FLine;
    property Col: LongInt read FCol;
    constructor Create(ATyp: TTokenTyp; ALexeme: String;
      AValue: Variant; ALine, ACol: LongInt);
    function toString: String; override;
    function Copy: TToken;
end;
```

What do we see here? TToken is a class with private members and public members. The general agreement is to start names of private fields with an 'F'. They are bound to the properties in the public area. The constructor creates a new instance of a token, and sets the variables in the private area. These variables can only be read via the corresponding properties.

Each token consists of a type FTyp, which gets its value from the enum TTokenTyp. The value in FLexeme represents the actual string of characters read by the Lexer. FLine and FCol give the precise location in the source text. This is very handy for showing were errors happen. We have one field to go: FValue of type Variant. It represents a constant value of integer, float, boolean, string or Null. A Variant type thus can hold any value that we encounter.

Before we go into the implementation of this unit, there's one more type to discuss.

```
  TKeywords = specialize TDictionary<string, TTokenTyp>;

var
  Keywords: TKeywords;
```

This is a map or dictionary (uCollections) representing language keywords and their accompanying token types. It will be used for quickly searching the keyword in the map and retrieve its type. As an example consider the variable Typ of type TTokenTyp:

```
Typ := Keywords['while'];  // Typ will have the value ttWhile
```

The implementation of the unit uToken is described next.

```
constructor TToken.Create
  (ATyp: TTokenTyp; ALexeme: String; AValue: Variant; ALine, ACol: LongInt);
begin
  FTyp := ATyp;
  FLexeme := ALexeme;
  FValue := AValue;
  FLine := ALine;
  FCol := ACol;
end;

function TToken.toString: String;
var
  TypStr: String;
begin
  WriteStr(TypStr, FTyp);
  Result := TypStr.Substring(2) + ' ("' + FLexeme + '")';
  Result += ' ' + VarToStr(FValue);  // special function to print variant value
end;

function TToken.Copy: TToken;
begin
  Result := TToken.Create(FTyp, FLexeme, FValue, FLine, FCol);
end;
```

TToken has a constructor, which initializes the private fields as mentioned above, and a .toString function which returns the contents of the token as a string. Since all fields are private, we cannot address them directly, and we'll use the respective properties for that. So, if I want to get the value of FTyp in a variable Token, I have to use Token.Typ, which produces the value through the property. The value of FTyp can never be changed once the token is created.

The function Copy simply returns an exact copy of the token.

```
function TTokenTypHelper.toString: string;
begin
  case Self of
    ttPlus : Result := '+';
    ttMin  : Result := '-';
    ttMul  : Result := '*';
    ttDiv  : Result := '/';
    ttRem  : Result := '%';
    ttPlusIs : Result := '+=';
    ttMinIs  : Result := '-=';
    ttMulIs  : Result := '*=';
    ttDivIs  : Result := '/=';
    ttRemIs  : Result := '%=';
    ttOr : Result := '|';
    ttAnd : Result := '&';
    ttNot : Result := '!';
    ttXor : Result := '~';
    ttShl : Result := '<<';
    ttShr : Result := '>>';
    ttPow : Result := '^';
    ttEQ : Result := '=';
    ttNEQ : Result := '<>';
    ttGT : Result := '>';
    ttGE : Result := '>=';
    ttLT : Result := '<';
    ttLE : Result := '<=';
    ttComma: Result := ',';
    ttDot: Result := '.';
    ttDotDot: Result := '..';
    ttAssign: Result := ':=';
    ttQuestion: Result := '?';
    ttArrow: Result := '=>';
    ttColon: Result := ':';
    ttOpenParen: Result := '(';
    ttCloseParen: Result := ')';
    ttOpenBrace: Result := '{';
    ttCloseBrace: Result := '}';
    ttOpenBrack: Result := '[';
    ttCloseBrack: Result := ']';
    ttEOF: Result := 'End of file';
    else begin
      WriteStr(Result, Self);
      Result := Result.Substring(2);
    end;
  end;
end;
```

This will be used for pretty printing. In order to finalize this paragraph, we can also finalize the unit. This is done by creating an initialization section and a finalization section. In this case we set up the map or dictionary for our keywords.

```
initialization

  Keywords := TKeywords.Create;
  Keywords.Sorted := True;

  // the constant values
  Keywords['False'] := ttFalse;
  Keywords['Null'] := ttNull;
  Keywords['True'] := ttTrue;
```

```
  // the keywords
  Keywords['array'] := ttArray;
  Keywords['break'] := ttBreak;
  Keywords['case'] := ttCase;
  Keywords['class'] := ttClass;
  Keywords['dictionary'] := ttDictionary;
  Keywords['do'] := ttDo;
  Keywords['each'] := ttEach;
  Keywords['else'] := ttElse;
  Keywords['end'] := ttEnd;
  Keywords['ensure'] := ttEnsure;
  Keywords['enum'] := ttEnum;
  Keywords['extension'] := ttExtension;
  Keywords['for'] := ttFor;
  Keywords['func'] := ttFunc;
  Keywords['if'] := ttIf;
  Keywords['in'] := ttIn;
  Keywords['is'] := ttIs;
  Keywords['inherited'] := ttInherited;
  Keywords['init'] := ttInit;
  Keywords['let'] := ttLet;
  Keywords['match'] := ttMatch;
  Keywords['on'] := ttOn;
  Keywords['print'] := ttPrint;
  Keywords['repeat'] := ttRepeat;
  Keywords['return'] := ttReturn;
  Keywords['self'] := ttSelf;
  Keywords['switch'] := ttSwitch;
  Keywords['then'] := ttThen;
  Keywords['trait'] := ttTrait;
  Keywords['until'] := ttUntil;
  Keywords['use'] := ttUse;
  Keywords['val'] := ttVal;
  Keywords['var'] := ttVar;
  Keywords['where'] := ttWhere;
  Keywords['while'] := ttWhile;

finalization
  Keywords.Free;

end.
```

Upon initialization of the unit, the map will be created and filled with its values. It will be used every time the Lexer runs into a string of text which is either a keyword or an identifier. In the course of the next chapters we will extend the list of keywords if needed, but we do have a nice basic set right now!

Finally, we clean up the keywords variable, when we don’t need it anymore, in this case, after our compiler has stopped.

Lexical scanning, mining tokens

Wikipedia gives the following definition:

"In computer science, lexical analysis is the process of converting a sequence of characters into a sequence of tokens. A program that performs lexical analysis may be called a lexer, tokenizer, or scanner."

> Know then thyself, presume not God to scan; The proper study of mankind is man.
>
> — Alexander Pope

In our case we call it the Lexer. The Lexer produces tokens, which are input for the Parser. The Lexer communicates with the uReader unit and with the uToken unit. Create unit uLexer.pas.

```
unit uLexer;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, uReader, uToken, uError;
```

Next, we start by defining the letters of our alphabet, or rather the characters that our tokens are made of.

```
const
  Space  = #32;
  Tab    = #9;
  Quote1 = #39; // '
  Quote2 = #34; // "

  WhiteSpace   = [Tab, LineEnding, Space];  // LineEnding is predefined in FPC
  Underscore   = ['_'];
  LoCaseLetter = ['a'..'z'];
  UpCaseLetter = ['A'..'Z'];
  Letters      = UpCaseLetter + LoCaseLetter;
  AlphaChars   = UpCaseLetter + LoCaseLetter + Underscore;
  NumberChars  = ['0'..'9'];
  SpecialChar  = '#';
  IdentChars   = NumberChars + AlphaChars;
```

From the above we can conclude that there will be white space, which we don't need and can be ignored by the lexer. So, if we find a whitespace character, the Lexer will move on to the next character, without processing it. Also, there are words made of AlphaChars, numbers made of digits 0 to 9. Finally, identifiers may consist of alpha, underscores and numerical characters. There's even a special character '#', for possible future use.

Note that both single quotes and double quotes are defined. While strings are between single quotes, I'm also thinking about single characters. My idea is to prepend a single character with a double quote sign: "a means the character a. Next, we create the Lexer class.

```
type

  TLexer = class
    private
      FLook : char;              // next input character (still unprocessed)
      FLine, FCol : integer;     // line and column number of the input character
      FReader: TReader;          // contains the text to scan
      FTokens: TTokens;          // list of mined tokens
      EndOfFile: Boolean;
      function getChar: Char;
      procedure doKeywordOrIdentifier(const Line, Col: Integer);
      procedure doNumber(const Line, Col: Integer);
      procedure doString(const Line, Col: Integer);
      procedure doChar(const Line, Col: Integer);
      procedure ScanToken(const Line, Col: Integer);
      Procedure ScanTokens;
      Procedure SingleLineComment;
      Procedure MultiLineComment;
    public
      property Tokens: TTokens read FTokens;
      Constructor Create(Reader: TReader);
  end;
```

The variable FLook contains the last read character from the input stream. FLine and FCol keep track of its location in the source text. FReader contains the complete input source text. Variable FTokens contains all scanned tokens processed by procedure ScanTokens.

The respective routines are discussed below in the implementation.

```
implementation

constructor TLexer.Create(Reader: TReader);
begin
  FReader := Reader;
  FTokens := TTokens.Create();
  FLine := 1;
  FCol := 0;
  EndOfFile := false;
  FLook := getChar;   // get first character
  ScanTokens;  // scan all tokens
end;
```

The constructor has only one parameter, the Reader, which contains the complete input file. The list of tokens is created. The 'True' in the constructor means that when the Tokens memory is freed, automatically all tokens in the list will be freed as well. We initialize the line and column, get the first character into FLook as a starting point, and finally scan all tokens. This routine does it all!

Next, let's look at routine getChar:

```
function TLexer.getChar: Char;
begin
  Result := FReader.NextChar;
  Inc(FCol);
  if Result = LineEnding then begin
    Inc(FLine);
    FCol := 0;
  end;
end;
```

This function returns the next character from the input stream, and increases the column number. If the end of the line is reached we increase the line number and set the column to zero again. It's simple as that!

Then, we scan all tokens:

```
procedure TLexer.ScanTokens;
begin
  while not EndOfFile do begin
    while FLook in WhiteSpace do
      FLook := getChar;   // skip white space
    ScanToken(FLine, FCol);
  end;
  Tokens.Add(TToken.Create(ttEOF, 'End of file', Nil, FLine, FCol));
end;
```

Routine ScanTokens loops until the end of file is reached. Inside the loop all white spaces are ignored, and when no white space found we scan for a token, passing the line and column. If the end of the file was reached, we add one final token: the end-of-file itself.

Procedure ScanToken is the real workhorse:

```
procedure TLexer.ScanToken(const Line, Col: Integer);

  procedure AddToken(const Typ: TTokenTyp; const Lexeme: String);
  var Token: TToken;
  begin
    Token := TToken.Create(Typ, Lexeme, Null, Line, Col);
    Tokens.Add(Token);
    FLook := getChar;
  end;

begin
  case FLook of
    '+' : if FReader.PeekChar = '=' then begin
            FLook := getChar;
            AddToken(ttPlusIs, '+=');
          end else AddToken(ttPlus, '+');
    '-' : if FReader.PeekChar = '=' then begin
            FLook := getChar;
            AddToken(ttMinIs, '-=');
          end else AddToken(ttMin, '-');
    '/' : case FReader.PeekChar of
            '/': begin FLook := getChar; SingleLineComment end;
            '*': begin FLook := getChar; MultiLineComment end;
            '=': begin FLook := getChar; AddToken(ttDivIs, '/=') end;
            else AddToken(ttDiv, '/');
          end;
    '*' : if FReader.PeekChar = '=' then begin
            FLook := getChar;
            AddToken(ttMulIs, '*=');
          end else AddToken(ttMul, '*');
    '%' : if FReader.PeekChar = '=' then begin
            FLook := getChar;
            AddToken(ttRemIs, '%=');
          end else AddToken(ttRem, '%');
    ':' : if FReader.PeekChar = '=' then begin
            FLook := getChar;
            AddToken(ttAssign, ':=');
          end else AddToken(ttColon, ':');
    '&' : AddToken(ttAnd, '&');
    '|' : AddToken(ttOr, '|');
```

```
    '~' : AddToken(ttXor, '~');
    '!' : AddToken(ttNot, '!');
    '^' : AddToken(ttPow, '^');
    '(' : AddToken(ttOpenParen, '(');
    ')' : AddToken(ttCloseParen, ')');
    '{' : AddToken(ttOpenBrace, '{');
    '}' : AddToken(ttCloseBrace, '}');
    '[' : AddToken(ttOpenBrack, '[');
    ']' : AddToken(ttCloseBrack, ']');
    ',' : AddToken(ttComma, ',');
    '.' : if FReader.PeekChar = '.' then begin
            FLook := getChar;
            AddToken(ttDotDot, '..');
          end else AddToken(ttDot, '.');
    '=' : if FReader.PeekChar = '>' then begin
            FLook := getChar;
            AddToken(ttArrow, '=>');
          end else AddToken(ttEQ, '=');
    '<' : case FReader.PeekChar of
            '<' : begin FLook := getChar; AddToken(ttShl, '<<'); end;
            '=' : begin FLook := getChar; AddToken(ttLE, '<='); end;
            '>' : begin FLook := getChar; AddToken(ttNEQ, '<>'); end;
            else AddToken(ttLT, '<');
          end;
    '>' : case FReader.PeekChar of
            '>' : begin FLook := getChar; AddToken(ttShr, '>>'); end;
            '=' : begin FLook := getChar; AddToken(ttGE, '>='); end;
            else AddToken(ttGT, '>');
          end;
    '?' : AddToken(ttQuestion, '?');
    '0'..'9': doNumber(Line, Col);
    '_', 'A'..'Z', 'a'..'z': doKeywordOrIdentifier(Line, Col);
    Quote1: doString(Line, Col);
    Quote2: doChar(Line, Col);
    FileEnding: EndOfFile := true;
    else
      EndOfFile := true;
  end;
end;
```

First, we define a nested procedure AddToken, which adds the token to the list of tokens. The main part of ScanToken is actually one large case statement. For every character defined by the language we add the correspondent token. Is some cases we have operators that consist of multiple characters, like '>=' or '+='. In these cases we peek what the next character is and if it matches an expected character, we have a multi character token.

Important to note is the handling of comments // for single line comments or /* … */ for multi line comments. Also, how to handle strings, chars, numbers, identifiers and keywords, is described in below procedures.

```
procedure TLexer.MultiLineComment;
begin
  Repeat
    Repeat
      FLook := getChar;
    Until (FLook = '*') or (FLook = FileEnding);
    FLook := getChar;
  Until (FLook = '/') or (FLook = FileEnding);
  FLook := getChar;
end;
```

```
procedure TLexer.SingleLineComment;
begin
  Repeat
    FLook := getChar;
  until FLook = LineEnding;
  FLook := getChar;
end;
```

The procedures speak for itself, I think. The multi line comment is read across multiple lines until it finds consecutively a '*' and a '/'. The single line comment is read until the end of the line.

The next method is about how to determine if a lexeme is a keyword or an identifier. The procedure is called when variable FLook is one of '_', 'A'..'Z', 'a'..'z'. We are actually saying here that when a new word (token) starts with a character defined in ('A..Z', '_', 'a'..'z'), it must be a keyword or an identifier. So, valid identifiers for example are 'hello', 'Hello', '_hello', 'HEL_LO', but also '_123Hello'. It just cannot start with a number.

Let's look at the procedure.

```
procedure TLexer.doKeywordOrIdentifier(const Line, Col: Integer);
var
  Index: integer = -1;
  Lexeme: String = '';
  TokenTyp: TTokenTyp = ttIdentifier;
  Token: TToken;
begin
  Lexeme := FLook;
  FLook := getChar;
  while FLook in IdentChars do begin
    Lexeme += FLook;
    FLook := getChar;
  end;

  //Match the keyword and return its type, otherwise it's an identifier
  if Keywords.Contains(Lexeme, Index) then
    TokenTyp := Keywords.At(Index);

  Token := TToken.Create(TokenTyp, Lexeme, Null, Line, Col);
  Tokens.Add(Token);
end;
```

We know our first character FLook defined this word as a keyword or identifier, so we assign it to Lexeme. Next, we keep on reading and adding to Lexeme until the character is not part of the set IdentChars anymore, which was defined above. Now we have to determine its nature: keyword or identifier. We use standard functionality from the 'uCollections' unit.

In our Keywords (defined in unit uToken) list we look up the value of Lexeme. If it was found the Contains function returns its index. In this case it must be a keyword.

If not found it must be an identifier. Simple right? We create the token and add it to the list.

Next, let's look at numbers. If FLook is in the set 0..9 it must be a number. A number can either be an integer, a floating point or an exponential value:

- 314 is an integer
- 3.1415 is a float
- 3.14E2 is an exponential E notation, which means 3.14 * 10^2

```
procedure TLexer.doNumber(const Line, Col: Integer);
var
  Lexeme: String = '';
  Value: Double;
  Token: TToken;
  FoundDotDot: Boolean = False;
begin
  Lexeme := FLook;
  // first read integer part of number
  FLook := getChar;
  while FLook in NumberChars do begin
    Lexeme += FLook;
    FLook := getChar;
  end;

  // did we encounter a '..'
  FoundDotDot := (FLook = '.') and (FReader.PeekChar = '.');

  // read the fraction if any
  if (FLook = '.') and not FoundDotDot then begin
    Lexeme += FLook;
    FLook := getChar;
    while FLook in NumberChars do begin
      Lexeme += FLook;
      FLook := getChar;
    end;
  end;

  // is there an exponent?
  if (Upcase(FLook) = 'E') and not FoundDotDot then begin
    Lexeme += FLook;
    FLook := getChar;
    if FLook in ['+', '-'] then begin
      Lexeme += FLook;
      FLook := getChar;
    end;
    while FLook in NumberChars do begin
      Lexeme += FLook;
      FLook := getChar;
    end;
  end;

  Value := Lexeme.ToDouble;
  Token := TToken.Create(ttNumber, Lexeme, Value, Line, Col);
  Tokens.Add(Token);
end;
```

The number is parsed, firstly up to a non numeric character. If the next character is a dot '.', it may be that we have a floating point number, but…

Notice the use of the PeekChar function, which looks ahead one more character. This is needed to make sure it's a float, and not an integer, used in the start of a range. Check out the following.

```
A := 3.14
A in 3..14
```

The first one is a variable that is assigned a floating point value of pi (3.14), wheras the second is a check whether A is in the range of 3 to 14.

So, we can only continue with parsing the number if the token is not '..'. Next, we'll check for possible scientific notation, such as 6.23e23 (e or E can be used), and add that to the total Lexeme, if found. Finally, the lexeme is cast to a double, which is an 8-byte value, and for the time being enough size.

Then, let's look at quoted text. As mentioned before we have strings and we have characters. Strings are between single quotes, whereas characters have a double quote prepended.

S := 'Hello world'
C := "!

This must be possible in our language:
str := S + C //Variable str will be of type String.

```
procedure TLexer.doString(const Line, Col: Integer);
var
  Lexeme: string = '';
  Value: String;
  Token: TToken;
begin
  FLook := getChar;
  while (FLook <> Quote1) and (FLook <> LineEnding) do begin
    Lexeme += FLook;
    FLook := getChar;
  end;

  if FLook = LineEnding then
    Errors.Add(FLine, FCol, 'Lexer error: String exceeds line.');

  FLook := getChar;   // consume quote '
  Value := Lexeme;
  Lexeme := '''' + Lexeme + '''';  // including the quotes
  Token := TToken.Create(ttString, Lexeme, Value, Line, Col);
  Tokens.Add(Token);
end;
```

Note that we read until a LineEnding and if this is reached an error message is generated. If we don't do this, the string will be read until the end of file is reached.

For processing a character the following procedure is used.

```
procedure TLexer.doChar(const Line, Col: Integer);
var
  Value: Char;
  Token: TToken;
begin
  FLook := getChar;
  Value := FLook;
  Token := TToken.Create(ttChar, '"'+FLook, Value, Line, Col);
  Tokens.Add(Token);
  FLook := getChar;
end;
```

And we close of with an 'end;'.

```
end.
```

This closes off the Lexer for now.

Before we dive into the parser, we must first focus on the AST. It was already mentioned…

Abstract the syntax in a tree

> The longer you look at an object, the more abstract it becomes, and, ironically, the more real.
> — Lucian Freud

"An abstract syntax tree (AST), or just syntax tree, is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. The syntax is "abstract" in not representing every detail appearing in the real syntax. For instance, grouping parentheses are implicit in the tree structure, and a syntactic construct like an if-condition-then expression may be denoted by means of a single node with three branches."

And that's exactly what we're going to do in our parser. It defines the grammar of our language, and in the process of parsing it accepts tokens from the lexer, and generates an abstract syntax tree.

We start with constructing (part of) the AST. It sort of represents our language, and after it's created by the parser it will be "visited" more than one time, e.g. for printing, for variable resolving and for interpreting. Create a new unit and name it uAST.pas.

```
unit uAST;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, uToken;

type

  TNode = class
    private
      FToken: TToken;
    public
      property Token: TToken read FToken;
      constructor Create(AToken: TToken);
  end;
```

The root node of all other nodes is TNode, a class that contains just a Token.

For expressions we also have a root node, which has TNode as its parent class.

```
  TExpr = class(TNode)
    // Base node for expressions.
  end;
```

The base node for all expressions is TExpr, from which all expression types inherit. The expression tree structure more or less resembles the expressions in the language as shown in the corresponding EBNF:

```
EBNF Expressions
================
Expr = AddExpr [ RelOp AddExpr ] .
AddExpr = MulExpr { AddOp MulExpr } .
MulExpr = ExpExpr { MulOp ExpExpr } .
ShiftExpr = UnaryExpr { ShiftOp UnaryExpr } .
UnaryExpr = [ '+' | '-' | '!' ] UnaryExpr | Factor
Factor = True | False | Nil
       | Number | String | Char
       | '(' Expr ')' .
```

In this EBNF you may note that there are binary expressions, such as Expr, AddExpr, MulExpr and ShiftExpr, and Unary expressions. Finally, there's a Factor that takes care of constant values and parentheses. Let's look at the binary expressions first.

```
TBinaryExpr = class(TExpr)
   private
     FLeft: TExpr;
     FOp: TToken;
     FRight: TExpr;
   public
     property Left: TExpr read FLeft;
     property Op: TToken read FOp;
     property Right: TExpr read FRight;
     constructor Create(ALeft: TExpr; AOp: TToken; ARight: TExpr);
     destructor Destroy; override;
 end;
```

A binary expression always consists of a left hand side and a right hand side, and obviously an operator. For example, x^y is represented by the following tree structure:

```
  ^
 / \
x   y
```

The Factor is the basis for a few expression types, such as constant values and parentheses:

```
 TFactorExpr = class(TExpr)
   // Base node for parsing a factor
 end;
```

A unary expression is a single expression preceded by a '+', '-' or a not '!' operator.

```
 TUnaryExpr = class(TFactorExpr)
   private
     FOp: TToken;         //operator Not, Minus, Plus
     FExpr: TExpr;        //single expression
   public
     property Op: TToken read FOp;
     property Expr: TExpr read FExpr;
     constructor Create(AOp: TToken; AExpr: TExpr);
     destructor Destroy; override;
 end;
```

From the EBNF, you'll notice there's some kind of recursion involved. We take a look at how to handle that in the Parser.

```
  TConstExpr = class(TFactorExpr)
    private
      FValue: Variant;
    public
      property Value: Variant read FValue;
      constructor Create(Constant: Variant; AToken: TToken);
  end;
```

And a ConstExpr is a Number, a String, a Char, a Boolean or Null, all represented in a Variant.

The implementation is straightforward:

```
implementation

{ TNode }

constructor TNode.Create(AToken: TToken);
begin
  FToken := AToken;
end;

{ TBinaryExpr }

constructor TBinaryExpr.Create(ALeft: TExpr; AOp: TToken; ARight: TExpr);
begin
  Inherited Create(AOp);
  FLeft :=  ALeft;
  FOp := AOp;
  FRight := ARight;
end;

destructor TBinaryExpr.Destroy;
begin
  if Assigned(FLeft) then FLeft.Free;
  if Assigned(FRight) then FRight.Free;
  inherited Destroy;
end;

{ TUnaryExpr }

constructor TUnaryExpr.Create(AOp: TToken; AExpr: TExpr);
begin
  Inherited Create(AOp);
  FOp := AOp;
  FExpr :=  AExpr;
end;

destructor TUnaryExpr.Destroy;
begin
  if Assigned(FExpr) then FExpr.Free;
  inherited Destroy;
end;
```

```
{ TConstExpr }

constructor TConstExpr.Create(Constant: Variant; AToken: TToken);
begin
  Inherited Create(AToken);
  FValue := Constant;
end;


end.
```

Parsing to the tree

> When every word is parsed for ill intention, regardless of who is speaking or why, we become so afraid we'll offend that we stop trying to communicate with people we don't understand.
>
> — Marti Noxon

"A parser is a software component that takes input data and builds a data structure – often some kind of parse tree, abstract syntax tree or other hierarchical structure – giving a structural representation of the input, checking for correct syntax in the process. The parsing may be preceded or followed by other steps, or these may be combined into a single step. The parser is often preceded by a separate lexical analyser, which creates tokens from the sequence of input characters."

The parser thus, determines and checks the structure of the Gear language. The parser has some basic functionality such as retrieving the next token, error handling, and parsing each language construct, like declarations and statements. The parser will be build up piece by piece. In this chapter we'll focus on parsing calculations with constant values.

Let's explain the parser by example: 12 * 3^3

The binary operator expression, is constructed with a left and right sub expression, and an operator.

The printed result looks like this:

```
(*)
  12
  (^)
    3
    3
```

And more complicated expressions: 2+3-5 and 2+3*5

```
(-)                 (+)
  (+)                 2
    2                 (*)
    3                   3
  5                     5
```

Notice the difference? This has to do with the precedence of the operators. '*' has a higher precedence than '+'. This job is taken care of by the parser.

Let's explain this by writing the expressions for this chapter in EBNF form.

```
Expr       = AddExpr [ RelOp AddExpr ] .   // [] is zero or one
AddExpr    = MulExpr { AddOp MulExpr } .   // {} is zero or more
MulExpr    = ShiftExpr { MulOp ShiftExpr } .
ShiftExpr  = UnaryExpr { ShiftOp UnaryExpr } .
UnaryExpr  = [ '+' | '-' | '!' ] UnaryExpr | Factor  .
Factor     = True | False | Null
           | Number | String | Char
           | '(' Expr ')' .
RelOp      = '=' | '<>' | '>' | '>=' | '<' | '<='  .
AddOp      = '+' | '-' | '|' | '~'   .
MulOp      = '*' | '/' | '%' | '&'  .
ShiftOp    = '<<' | '>>' | '^'   .
```

An expression is made up from 1 or 2 additive expressions, separated by a relational operator. So, it will be perfectly legal to create expressions like a + b >= c. A relational operator can be =, <, >, <=, >=, <> for example. An additive expression consists of 1 or more multplicative expressions separated by addition operators (+, -, |), making it possible to write a * b + c / d. First a * b is calculated, then c / d, finally the results are added to each other. The multiplicative expressions contains 1 or more shift/power expressions, for example 2^3 * 2. '^' is the power operator.

For now the factor can be a any constant, including Null, start with a + (plus), - (minus) or | (or) sign, may use parentheses, or be a variable identifier. Function calls are not supported yet, this will follow later.

Create a new unit and name it uParser.pas.

```
unit uParser;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, uLexer, uToken, uAST, uError;

type

  TParser = class
    private
      Tokens: TTokens;
      Current: integer;
      function CurrentToken: TToken;
      function Peek: TToken;
      procedure Error(Token: TToken; Msg: string);
      procedure Expect(const TokenTyp: TTokenTyp);
      procedure Next;
      procedure Synchronize(Types: TTokenTypSet);
      function isLastToken: Boolean;
    public
      constructor Create(Lexer: TLexer);
      destructor Destroy; override;
      function Parse: TExpr;
    private
      // Expressions
      function ParseExpr: TExpr;
      function isRelOp: Boolean;
      function ParseAddExpr: TExpr;
      function isAddOp: Boolean;
      function ParseMulExpr: TExpr;
      function isMulOp: Boolean;
      function ParseShiftExpr: TExpr;
      function isShiftOp: Boolean;
      function ParseUnaryExpr: TExpr;
      function ParseFactor: TExpr;
  end;
```

We'll go through the implementation and describe all that happens. First, we start with a few helper routines.

```
implementation
```

```
function TParser.CurrentToken: TToken;
begin
  Result := Tokens[Current];
end;
```

This function returns the actual current token, currently being processed by the parser.

```
function TParser.Peek: TToken;
begin
  if not isLastToken then
    Result := Tokens[Current+1];
end;
```

Peek returns the next token in line without parsing it. We will need this later on.

```
procedure TParser.Error(Token: TToken; Msg: string);
begin
  Errors.Append(Token.Line, Token.Col, Msg);
end;
```

Procedure Error adds an error message to the list of errors.

```
procedure TParser.Expect(const TokenTyp: TTokenTyp);
const
  Msg = 'Syntax error, "%s" expected.';
begin
  if CurrentToken.Typ = TokenTyp then
    Next
  else
    Error(CurrentToken, Format(Msg, [TokenTyp.toString]));
end;
```

In some cases we expect a certain token. If that token doesn't appear, we generate an error. As an example consider the following:

2+(3*6

In this case a '))' was expected, so this produces an error:

```
@[1,7]: Syntax error, ")" expected.
```

Procedure Next moves the token pointer one notch up in the list of tokens.

```
procedure TParser.Next;
begin
  Current +=1;
end;
```

Here's the procedure I mentioned before: Synchronize.

```
procedure TParser.Synchronize(Types: TTokenTypSet);
begin
  while not (CurrentToken.Typ in Types) do
    Next;
end;
```

It is used after a syntax error, and will synchronize the parser such that parsing can continue. This way we don't have to stop after encountering a single error but we can finish the complete input text. It will be used after this chapter.

IsLastToken is to check whether we hit the last token in the tokens list.

```
function TParser.isLastToken: Boolean;
begin
  Result := Current = Tokens.Count-1;
end;
```

```
constructor TParser.Create(Lexer: TLexer);
begin
  Tokens := Lexer.Tokens;
  Current := 0;
end;
```

The constructor takes as the parameter the complete tokens list from the Lexer, and sets the current token to zero, the first one in the list. The destructor destroys the token list.

```
destructor TParser.Destroy;
begin
  if Assigned(Tokens) then Tokens.Free;
  inherited Destroy;
end;
```

```
function TParser.Parse: TExpr;
begin
  try
    Result := ParseExpr;
    Expect(ttEOF);
  except
    on E: EParseError do begin
      Result := Nil;
    end;
  end;
end;
```

Parse is the function that parses the complete input. For now it calls ParseExpr, as we deal with single line expressions here, but soon we'll extend it. If we find an error we return nil.

Now we move into the expressions themselves. Remember the EBNF above and note that we exactly follow the rules described there.

```
EBNF: Expr = AddExpr [ RelOp AddExpr ] .
```

```
function TParser.ParseExpr: TExpr;
var
  RelOp: TToken;
begin
  Result := ParseAddExpr;
  if isRelOp then begin
    RelOp := CurrentToken;
    Next;
    Result := TBinaryExpr.Create(Result, RelOp, ParseAddExpr);
  end;
end;
```

According to the EBNF, we should first parse an AddExpr, which we put in variable Result. Then there is an optional relational operator and again a additive expression. Function isRelOp checks if the current token type is a relational operator.

```
function TParser.isRelOp: Boolean;
begin
  Result := CurrentToken.Typ in [ttEQ, ttNEQ, ttGT, ttGE, ttLT, ttLE];
  //These tokens                  =     <>     >     >=    <     <=
end;
```

We set the current token to token RelOp, and move forward to the next token. Then, we create the binary tree.

Remember that it takes a left hand side expression, an operator and a right hand side expression as its parameters. The left hand side was already stored in variable Result. For the right hand side we call ParseAddExpr again.

The complete tree structure is then returned in variable Result.

Next, we take the multiplication expression, also a binary expression.

```
EBNF: AddExpr = MulExpr { AddOp MulExpr } .
```

```
function TParser.ParseAddExpr: TExpr;
var
  AddOp: TToken;
begin
  Result := ParseMulExpr;
  while isAddOp do begin
    AddOp := CurrentToken;
    Next;
    Result := TBinaryExpr.Create(Result, AddOp, ParseMulExpr);
  end;
end;
```

```
function TParser.isAddOp: Boolean;
begin
  Result := CurrentToken.Typ in [ttPlus, ttMin, ttOr, ttXOr];
  //                                 +      -      |     ~
end;
```

The difference with the previous EBNF rule is that we have curly braces {} in this one. That means that we can repeat that part of the rule zero or more times, as opposed to the block brackets [] that mean zero or one time exactly. The latter was solved with an if...then statement. In this rule however, we have to use a while...do statement. This rule for example solves expressions like 1+2+3+4+5+6+… As long as there's an AddOp it continues to build the binary tree.

```
EBNF: MulExpr = ShiftExpr { MulOp ShiftExpr } .
```

```
function TParser.ParseMulExpr: TExpr;
var
  MulOp: TToken;
begin
  Result := ParseShiftExpr;
  while isMulOp do begin
    MulOp := CurrentToken;
    Next;
    Result := TBinaryExpr.Create(Result, MulOp, ParseShiftExpr);
  end;
end;
```

```
function TParser.isMulOp: Boolean;
begin
  Result := CurrentToken.Typ in [ttMul, ttDiv, ttRem, ttAnd];
  //                              *      /      %      &
end;
```

Also here, we follow the EBNF rule exactly. As long as there are multiplicative operators we continue to parse and build the binary tree. It solves expressions such as 2*3*4/6.

By our choice for combining both boolean operators and multiplication operators (the same for the additive operators) we have to take care not to mix them in expressions. According our EBNF this is perfectly legal: 3 * 2 & 6, however we will create a runtime error when this happens. We have to catch this later on in the interpreter. Another solution could be to not use the & as a logical operator but as a bit-wise operator.

Next, the shift expression as I called it, which covers the operators '<<' for a left shift, '>>' for a right shift, and the power operator '^'. A left shift shifts the bits to the left side.

Take for example 2<<1. The number 2 in binary notation is 10. If you shift that 1 to the left you'll get 100, which is a decimal 4. The power operator '^' in 2^3 raises 2 to the power of 3, which is 8. So in comparison, 2<<2, 2^3 and 2*2*2 all result in 8.

```
EBNF: ShiftExp = UnaryExpr { ExpOp UnaryExpr } .
```

```
function TParser.ParseShiftExpr: TExpr;
var
  ShiftOp: TToken;
begin
  Result := ParseUnaryExpr;
  while isShiftOp do begin
    ShiftOp := CurrentToken;
    Next;
    Result := TBinaryExpr.Create(Result, ShiftOp, ParseUnaryExpr);
  end;
end;
```

```
function TParser.isShiftOp: Boolean;
begin
  Result := CurrentToken.Typ in [ttShl, ttShr, ttPow];
  //                              <<     >>      ^
end;
```

Next in line is the Unary expression. Does the expression have a '+', a '-' or a '!' in front of it?

```
function TParser.ParseUnaryExpr: TExpr;
var
  Op: TToken;
begin
  if CurrentToken.Typ in [ttPlus, ttMin, ttNot] then begin
    Op := CurrentToken;
    Next;
    Result := TUnaryExpr.Create(Op, ParseUnaryExpr());
  end
  else
    Result := ParseFactor;
end;
```

If a +, - or ! token was found, this Token is stored in variable 'Op'. When the unary expression is created, the second parameter calls again ParseUnaryExpr(). This is called a recursive call, and because of this you are able to calculate expressions, such as: ----7 or -+--++-6.

If the expression doesn't have one of the mentioned operators in front, we move on to the parsing of the factor.

The factor creates the leafs of the tree, so to say. This basic version (yes it will be extended with if-expressions, match-expressions and function calls) is able to parse this EBNF:

```
Factor = True | False
         | Null
         | Number | String | Char
         | '(' Expr ')' .
```

True and False are the two boolean constants. Null (sometimes called Nil) is a value that represents an empty or no value at all. Number can be integer or float values. Strings are between ` and ', whereas chars start with a ".

```
function TParser.ParseFactor: TExpr;
begin
  case CurrentToken.Typ of
    ttFalse, ttTrue: begin
      Result := TConstExpr.Create(CurrentToken.Typ = ttTrue, CurrentToken);
      Next;
    end;
    ttNull: begin
      Result := TConstExpr.Create(Null, CurrentToken); // Null is variant enum
      Next;
    end;
    ttNumber, ttString, ttChar: begin
      Result := TConstExpr.Create(CurrentToken.Value, CurrentToken);
      Next;
    end;
    ttOpenParen: begin
      Next; // skip '('
      Result := ParseExpr;
      Expect(ttCloseParen);
    end;
    else begin
      Result := TExpr.Create(CurrentToken);
      Error(CurrentToken,'Unexpected token: ' + CurrentToken.toString + '.');
    end;
  end;
end;

end. // end of unit file
```

If we detect a '(', we have to deal with a parenthesized expression. We skip the '(', parse the expression and finally expect a closing ')'.

Where's the bug report!

To keep track of errors, we need a way to store them during the parsing process. We don't want to stop at the first error, but continue parsing till the end. We'll store all errors in a list clas that we create ourselves. Create a new unit and name it uError.pas.

> A novelist can never be his own reader, except when he is ridding his manuscript of syntax errors, repetitions, or the occasional superfluous paragraph.
>
> — Patrick Modiano

```
unit uError;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, uToken, uCollections;

type

  TErrorItem = class
    Line, Col: Integer;
    Msg: String;
    constructor Create(const ALine, ACol: Integer; const AMsg: String);
    function toString: String; override;
  end;

  TErrors = class(specialize TArrayObj<TErrorItem>)
    procedure Append(const ALine, ACol: Integer; const AMsg: String);
    function isEmpty: Boolean;
    procedure Reset;
    function toString: String; override;
  end;

  EParseError = class(Exception);

  ERuntimeError = class(Exception)
    Token: TToken;
    constructor Create(AToken: TToken; AMessage: String);
  end;


procedure RuntimeError(E: ERuntimeError);
procedure RuntimeWarning(E: ERuntimeError);
```

Inside the TErrors class we define a type TErrorItem, which contains the error location and the message. We can append items, test for emptyness, print to string and reset the array to zero. The last one we need for the execution from the prompt. You want to reset the errors after each execution.

We also define a type ERuntimeError, which descends from class Exception. Additionally, we pass the token in order to have the Line and Column of the error location. Runtme errors are called via the convenience procedure RuntimeError, which takes an error of type ERuntimeError as the parameter.

There will also be warnings, which send a message but don't stop execution.

Finally, we define the variable Errors of type TErrors:

```
var
  Errors: TErrors;
```

Next, the implementation:

```
implementation

{ TErrorItem }

constructor TErrorItem.Create(const ALine, ACol: Integer; const AMsg: String);
begin
  Line := ALine;
  Col := ACol;
  Msg := AMsg;
end;

function TErrorItem.toString: String;
begin
  Result := Format('@[%d,%d]: %s', [Line, Col, Msg]);
end;
```

```
{ TErrors }

procedure TErrors.Append(const ALine, ACol: Integer; const AMsg: String);
begin
  Add(TErrorItem.Create(ALine, ACol, AMsg));
end;

function TErrors.isEmpty: Boolean;
begin
  Result := Count = 0;
end;

procedure TErrors.Reset;
begin
  Clear;
end;

function TErrors.toString: String;
var
  Item: TErrorItem;
begin
  Result := 'Errors:' + LineEnding;
  for Item in Self do
    Result += Item.toString + LineEnding;
end;
```

```
{ proc RuntimeError }

procedure RuntimeError(E: ERuntimeError);
begin
  WriteLn('@[' + IntToStr(E.Token.Line) + ',' + IntToStr(E.Token.Col) + '] ' +
    'Runtime error: ', E.Message);
  Exit;
end;

procedure RuntimeWarning(E: ERuntimeError);
begin
  WriteLn('@[' + IntToStr(E.Token.Line) + ',' + IntToStr(E.Token.Col) + '] ' +
    'Runtime warning: ', E.Message);
end;
```

```
{ ERuntimeError }

constructor ERuntimeError.Create(AToken: TToken; AMessage: String);
begin
  Token := AToken;
  inherited Create(AMessage);
end;

initialization
  Errors := TErrors.Create();

finalization
  Errors.Free;
end.
```

Note that we initialize variable Errors and also make sure that it is finalized again.

Plan your visit

There's lots of documentation to find on internet on how the visitor pattern theoretically works. It uses Visits and Accepts and is good to understand, however requires quite some work to implement. There's an easier way that uses the principle of reflection or realtime type information (RTTI) in Free Pascal.

> Santa Claus has the right idea - visit people only once a year.
>
> — Victor Borge

We are able to visit every tree node without adding code to the tree, and still be able to perform actions based on the tree contents. We can print the tree, or generate a filled symbol table, or generate code or interpret the tree directly. We just have to create a descendent from the the main visitor class that performs all actions. How cool is that?

Let's start by creating our visitor class. It requires knowledge of the RTTI system and inner workings of the Free Pascal compiler, but we will manage! Create a new unit called uVisitor.pas.

```
unit uVisitor;
{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, typinfo;
```

Important here is to include standard Free Pascal unit 'typinfo'.

```
type

  {$M+}

  TVisitor = class
    published
      function Visit(Node: TObject): Variant; virtual;
  end;
```

Here, you have to include the {$M+} directive. This makes it possible to use RTTI (RealTime Type Information). This simple class forms the basis for our visitor system. The class has only one published function Visit, that has a node of TObject as a parameter and which returns a Variant value. This only works for published methods! Now let's go to the implementation.

```
implementation

type
  TVisit = function(Node: TObject): Variant of object;
```

We define a private (to the unit) type TVisit, which more or less is a template copy of the previous mentioned Visit function. Since all visitor methods are inside classes, we add 'of object'.

Next, we'll implement function Visit of class TVisitor.

```
function TVisitor.Visit(Node: TObject): Variant;
var
  VisitName: string;
  VisitMethod: TMethod;
  doVisit: TVisit;
  SelfName: string = '';
begin
  // Build visitor name: e.g. VisitBinaryExpr from 'Visit' and TBinaryExpr
  VisitName := 'Visit' + String(Node.ClassName).Substring(1);  // remove 'T'
  SelfName := Self.ClassName;
  VisitMethod.Data := Self;
  VisitMethod.Code := Self.MethodAddress(VisitName);
  if Assigned(VisitMethod.Code) then begin
    doVisit := TVisit(VisitMethod);
    Result := doVisit(Node);
  end
  else
    Raise
      Exception.Create(Format('No %s.%s method found.', [SelfName, VisitName]));
end;


end.
```

Since Node is a descendant of TObject, it's possible to get its class name from Node.ClassName. We cast this to String and then remove the 'T' from the type name. Then, Self is the class that executes the visitor, e.g. TInterpreter, and we have to set the reference to the class to the Data field of the TMethod record. So, Data points to the visiting class. Next, we have to search for the actual visitor code (is method address) by calling Self.MethodAddress(VisitName).

If all goes well, variable VisitMethod now contains everything we need to execute it. We do this in two steps: first, cast VisitMethod to the TVisit template type and assign it to 'doVisit', followed by the actual execution 'Result := doVisit(Node);'.

If the respective method cannot be found, a runtime error is generated. This usually means you forgot to create the respective method in the visitor class.

This is it, basically!

All visitor classes that you create from here descend from TVisitor, and the visitor methods that you create must be published. For example:

```
TPrinter = class(TVisitor)
  published
    methods…
end;

TInterpreter = class(TVisitor)
  published
    methods…
end;
```

Print that tree

The first visitor class that we'll introduce is the printer class. Create a new unit and name it uPrinter.pas. It shows the simplicity and therefore beauty of our visitor solution.

> Too many trees are killed to print the words of people who may not have all that much to say, and authors and journalists are equally culpable in this regard.
>
> — Vikram Seth

```
unit uPrinter;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, Variants, uAST, uVisitor, uToken;

type

  TPrinter = class(TVisitor)
    private
      const Increase = 2;
      var
        Indent: string;
        Tree: TExpr;
      procedure IncIndent;
      procedure DecIndent;
    public
      constructor Create(ATree: TExpr);
      procedure Print;
    published
      procedure VisitNode(Node: TNode);
      procedure VisitBinaryExpr(BinaryExpr: TBinaryExpr);
      procedure VisitConstExpr(ConstExpr: TConstExpr);
      procedure VisitUnaryExpr(UnaryExpr: TUnaryExpr);
  end;
```

As already shown in a few examples, we create a simple tree printing structure. Instead of printing nice vertical trees, we create horizontal trees. For expressions: 2+3-5 and 2+3*5 the trees are:

But they will get printed as:

```
(-)                                   (+)
  (+)                                   2
    2                                   (*)
    3                                     3
  5                                       5
```

TPrinter inherits from TVisitor, so has access to the Visit method. Notice that for each AST node we define a visitor, such as 'VisitBinaryExpr'. This method will print the contents of a TBinaryExpr instance. So, all action takes place here, separate from the AST itself.

Upon creation the AST Tree is input, and since currently TExpr is the top most node, it contains the complete tree.

Also note we have two helper methods called 'IncIndent' and 'DecIndent', which are needed to create some readibility and true tree like structure when printing the AST. Now the implementation.

```
implementation

procedure TPrinter.IncIndent;
begin
  Indent := StringOfChar(' ', Length(Indent) + Increase);
end;

procedure TPrinter.DecIndent;
begin
  Indent := StringOfChar(' ', Length(Indent) - Increase);
end;

constructor TPrinter.Create(ATree: TExpr);
begin
  Indent := '  ';
  Tree := ATree;
end;

procedure TPrinter.Print;
begin
  Visit(Tree);
  Writeln;
end;
```

The private methods IncIndent and DecIndent assign a string of spaces to variable Indent, by taking the current length of Indent and increase or decrease it with the value in constant Increase (2). The constructor sets Indent to an initial value of 2 spaces and assigns the AST to variable Tree.

The public procedure Print does what it's name says it does. It prints the complete AST, by calling the Visit method derived from TVisitor.

Next follow the Visit*** methods, each printing the contents of their respective nodes.

```
procedure TPrinter.VisitNode(Node: TNode);
begin
  Writeln(Indent + String(Node.ClassName).Substring(1));
end;
```

Except for the VisitNode procedure, I added as a first action, a call to IncIndent, and concluded always with DecIndent. It's as simple as that.

```
procedure TPrinter.VisitBinaryExpr(BinaryExpr: TBinaryExpr);
begin
  IncIndent;
  WriteLn(Indent, '(', BinaryExpr.Op.Typ.toString, ')');
  Visit(BinaryExpr.Left);
  Visit(BinaryExpr.Right);
  DecIndent;
end;

procedure TPrinter.VisitConstExpr(ConstExpr: TConstExpr);
begin
  IncIndent;
  WriteLn(Indent, VarToStrDef(ConstExpr.Value, 'Null'));
  DecIndent;
end;
```

```
procedure TPrinter.VisitUnaryExpr(UnaryExpr: TUnaryExpr);
begin
  IncIndent;
  WriteLn(Indent, '(', UnaryExpr.Op.Typ.toString, ')');
  Visit(UnaryExpr.Expr);
  DecIndent;
end;

end.
```

Where's the benefit in all this? Well, firstly, we don't have to add difficult code to our AST nodes everytime we want to operate on it, and secondly, if we create a new AST node class, we only have to add one method to the visitor class. Thirdly, it keeps all functionality of an operation (such as printing, or interpreting) neatly together.

We have one more paragraph to go before we can actually run our own expression calculator.

Interpret your code

An interpreter is a computer program that directly executes, i.e. performs, instructions written in a programming or scripting language, without previously compiling them into a machine language program. An interpreter generally uses one of the following strategies for program execution:

> Language is a process of free creation; its laws and principles are fixed, but the manner in which the principles of generation are used is free and infinitely varied. Even the interpretation and use of words involves a process of free creation.
>
> — Noam Chomsky

1. parse the source code and perform its behavior directly.
2. translate source code into some efficient intermediate representation and immediately execute this.
3. explicitly execute stored precompiled code made by a compiler which is part of the interpreter system.

In our case, we already created an intermediate representation: the AST. This means we apply the 2nd strategy for program execution.

Again, we'll use the visitor pattern. Till now it has proven to be easy to understand an build. The big plus is that we don't need to change the AST for it. Create a new unit and call it uInterpreter.pas.

```
unit uInterpreter;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, uVisitor, uAST, uToken, uError, Variants, uMath;

type

  TInterpreter = class(TVisitor)
    public
      procedure Execute(Tree: TExpr);
    published
      function VisitBinaryExpr(BinaryExpr: TBinaryExpr): Variant;
      function VisitConstExpr(ConstExpr: TConstExpr): Variant;
      function VisitUnaryExpr(UnaryExpr: TUnaryExpr): Variant;
  end;
```

The nodes that we want to visit are grouped in the published area. From the outside world (the main pogram) we'll call the Execute method and in the private area we have a few helper methods.

Let's dive into the implementation.

When we execute, we always execute the complete tree. At this moment it's an expression, but later on we'll be able to execute complete programs. We use a try…except statement to execute the tree. The result of the tree visit is stored in variable Value. Using function VarToStrDef from Object Pascal unit Variants we are able to print the contents of any variant value. As default value we pass 'Null'.

```
procedure TInterpreter.Execute(Tree: TExpr);
var
  Value: Variant;
begin
  try
    Value := Visit(Tree);
    Writeln(VarToStrDef(Value, 'Null'));
  except
    on E: ERuntimeError do
      RuntimeError(E);
  end;
end;
```

If an exception of type ERuntimeError occurs, we call convenience procedure RuntimeError(E) to print the error, including location of the error.

We now introduce the heart of the binary expression. All binary calculations are done in this function. It uses functions _Add, _Sub, _Mul, etc from the uMath library, which you can find in the appendix.

```
function TInterpreter.VisitBinaryExpr(BinaryExpr: TBinaryExpr): Variant;
var
  Left, Right: Variant;
  Op: TToken;
begin
  Left := Visit(BinaryExpr.Left);
  Right := Visit(BinaryExpr.Right);
  Op := BinaryExpr.Op;
  case BinaryExpr.Op.Typ of
    ttPlus: Result := TMath._Add(Left, Right, Op);
    ttMin:  Result := TMath._Sub(Left, Right, Op);
    ttMul:  Result := TMath._Mul(Left, Right, Op);
    ttDiv:  Result := TMath._Div(Left, Right, Op);
    ttRem:  Result := TMath._Rem(Left, Right, Op);
    ttOr:   Result := TMath._Or(Left, Right, Op);
    ttAnd:  Result := TMath._And(Left, Right, Op);
    ttXor:  Result := TMath._Xor(Left, Right, Op);
    ttShl:  Result := TMath._Shl(Left, Right, Op);
    ttShr:  Result := TMath._Shr(Left, Right, Op);
    ttPow:  Result := TMath._Pow(Left, Right, Op);
    ttEQ:   Result := TMath._EQ(Left, Right, Op);
    ttNEQ:  Result := TMath._NEQ(Left, Right, Op);
    ttGT:   Result := TMath._GT(Left, Right, Op);
    ttGE:   Result := TMath._GE(Left, Right, Op);
    ttLT:   Result := TMath._LT(Left, Right, Op);
    ttLE:   Result := TMath._LE(Left, Right, Op);
  end;
end;
```

All expression based type checking is done in the interpreter for the moment. It means a lot of overhead, but also gives flexibility later on if we introduce dynamic typing. The function allows for a lot of functionality, for example the following additive expressions are allowed:

'abc' + 'def' => result is a string with value 'abcdef'

'abc' + "d => result is a string with value 'abcd', "d represents character d

'Number : ' + 42 => result is a string with value 'Number : 42'

Function VisitConstExpr simply returns the node's value:

```
function TInterpreter.VisitConstExpr(ConstExpr: TConstExpr): Variant;
begin
  Result := ConstExpr.Value;
end;
```

Next, function VisitUnaryExpr, which handles the '-', '!' and '+' opereators in front of an expression.

```
function TInterpreter.VisitUnaryExpr(UnaryExpr: TUnaryExpr): Variant;
var
  Expr: Variant;
begin
  Expr := Visit(UnaryExpr.Expr);
  case UnaryExpr.Op.Typ of
    ttNot: Result := TMath._Not(Expr, UnaryExpr.Op);
    ttMin: Result := TMath._Neg(Expr, UnaryExpr.Op);
    else Result := Expr;
  end;
end;

end.
```

With this, chapter 1 comes to an end. We have created a calculator that is capable of quite complex expressions. Compile and run the project, using both possible ways, directly from the prompt or by creating small files with single expressions, which also allows you to print the AST contents. Here are again the command line parameters.

> /path/gear -x -f /path/testfile.txt

Or

> /path/gear -a -f /path/testfile.txt
For printing the AST.

When you test the code, also try to make some deliberate errors.

Chapter 2 – Language basics

In the previous chapter we created a calculator. It could perform pretty complex calculations, handle Boolean expressions, String expressions, yes even the expression: 'The answer is: ' + (20*(1+3)+4)/2 is no problem, but in order to be a real language we need more, much more… For example identifiers such as variables that we need to store and find again.

A symbol table is a data structure used by a language translator such as a compiler or interpreter, where each identifier in a program's source code is associated with information relating to its declaration or appearance in the source.

Creating products

> By venturing into space, we improve life for everyone here on Earth - scientific advances and innovations that come from this kind of research create products we use in our daily lives.
>
> — Buzz Aldrin

Much has been written about variable declaration, and each programming language has it own philosophy on that. I don't know what the best way is, but I have some ideas. But what is a declaration exactly? The dictionary tells me it's 'an explicit, formal announcement, either oral or written'. I firmly believe that variables should be declared before they are used, and that they cannot change type once they're declared. This helps readibility of your program on one hand and debugging of errors on the other.

Usually, variables are declared to be of a certain type. For example you can declare the variable Number as an integer, or the variable Pi as a float. In the latter case you would probably want to assign a default value to Pi, namely 3.1415. Moreover, I would like to assign 3.145 to Pi without declaring the explicit type. Why? The value itself is a float, which means if I assign it to Pi, the compiler should infer that it is of type float. We're coming close to the definition of our variable declaration. So, to start, here are some examples.

```
var myName := 'Jeroen'
var pi := 3.1415926
var done := False
var object := Null
```

and then used, for example in assignments:

```
myName := myName + "!
done := True
```

However, the term variable implies that it is mutable. Do you want Pi to be mutable? I guess not, so we should think about how to make variables immutable. Immutable means constant: it cannot change. For constant values we introduce the keyword 'let'. Some examples:

```
let pi := 3.1415926
var radius := 2
var circleSquare := 2*pi*radius
```

By varying radius, circleSquare also varies, but pi will never change.

A couple of things can be noticed. First, we will not need semicolons to close or finish a declaration or statement. Secondly, we use ':=' for assignments, and not the single '=', which is only to be used in comparisons. So, for example a=b, means compare a with b and return True if they are equal, otherwise return False. Thirdly, the types of variables are inferred automatically from the expression. Types are discussed later, and our language will combine (in some cases) static and dynamic typing. For example, if a variable is declared and inferred as a String, it cannot change type anymore, which gives more safety to the programmer. Fourth, the constants False, True and Null start with a capital. Our language is case-sensitive! By the way, you can declare a variable with initial value Null, and later on assign it a different value.

In terms of statements in this chapter we'll discuss the print statement and the assignment.

The first chapter was quite long, in order to set up a full working cradle. From now on every paragraph creates a runnable version of the interpreter.

Product EBNF

A complete Gear product consists of declarations and statements. So far, only one single expression could be processed, so let's make our parser more future proof first. We wish to parse a complete product from now on. This means our EBNF will change. A Product is defined in EBNF as:

```
Product = Block .
Block = { Stmt | Decl } .         // zero or more declarations or statements
Stmt = PrintStmt | AssignStmt .     // either a print stmt or an assignment
Decl = VarDecl | LetDecl .          // either a var or let decl (for now…)
VarDecl = 'var' Ident ':=' Expr .
LetDecl = 'let' Ident ':=' Expr .
PrintStmt = 'print' '(' ExprList ')' .
AssignStmt = Variable ':=' Expr .
ExprList = Expr { ',' Expr }  .     // 1 or more expressions
```

The above new EBNF of a Product accepts zero or more of both declarations and/or statements. The use of Block seems redundant, however, it comes back later when we'll see blocks in use, for example in the while-statement: 'while' Condition 'do' Block 'end'

How are we going to represent that in a workable way, and then also parse the code? Best is to stick to the EBNF and create a new AST node for Product. Add it just above the keyword Implementation. We'll build it up top-down.

```
TProduct = class(TBlock)
end;
```

And right above this definition put the definition of a Block:

```
TNodeList = specialize TArrayObj<TNode>;
```

```
  TBlock = class(TNode)
    private
      FNodes: TNodeList;
    public
      property Nodes: TNodeList read FNodes;
      constructor Create(ANodes: TNodeList; AToken: TToken);
      destructor Destroy; override;
  end;
```

Since both declarations and statements are inherited from TNode, it's better to use a TNodeList. TNodeList is a specialization of the generic class TArrayObj<>. We'll use more classes from the uCollections unit later on (see appendix). This one maintains a list of objects/classes.

Private field FNodes contains all declarations and statements inside a block. By the way, also amend the uses clause at the top of the unit. It should include uCollections.

```
uses
  Classes, SysUtils, uToken, uCollections;
```

The implementation of TBlock:

```
constructor TBlock.Create(ANodes: TNodeList; AToken: TToken);
begin
  Inherited Create(AToken);
  FNodes := ANodes;
end;

destructor TBlock.Destroy;
begin
  if Assigned(FNodes) then FNodes.Free;
  inherited Destroy;
end;
```

In order not to parse only an expression we adjust the Language.Execute procedure in uLanguage.pas. Replace 'TExpr' with 'TProduct'. Also, in procedure PrintAST, replace the type of the parameter from TExpr to TProduct.

The class procedure header now looks like this:

```
class procedure Language.PrintAST(Tree: TProduct);
var
  ...
  Tree: TProduct = Nil;
begin
  ...
end;
```

The rest of the method stays the same, which is shown by the three dots '...'.

```
class procedure Language.Execute(const Source, Flags: String; InputType: TInputType);
var
  ...
  Tree: TProduct = Nil;
begin
  ...
end;
```

Parsing the product

A Gear product is a full working program, according to the EBNF, which consists of statements and declarations.

In this case we first look at parsing a block. We possibly can reuse some of this code when we parse the full product. We know that a block consists of zero or more statements or declarations, and that they are stored in a list of nodes. Our first try is this:

```
function TParser.ParseBlock: TBlock;
var
  Nodes: TNodeList;
  Token: TToken;
begin
  Token := CurrentToken;
  Nodes := TNodeList.Create();
  while CurrentToken.Typ in [ttLet, ttVar, ttPrint, ttIdentifier] do
    if CurrentToken.Typ in [ttLet, ttVar] then
      Nodes.Add(ParseDecl)
    else if CurrentToken.Typ in [ttPrint, ttIdentifier] then
      Nodes.Add(ParseStmt)
    else Error(CurrentToken, 'Unrecognized declaration or statement.');
  Result := TBlock.Create(Nodes, Token);
end;
```

However, this is not the most optimal way of parsing. We expect way more statement and declaration types, so it's better to create two constants, named StmtStartSet and DeclStartSet. Their names suggest they contain all token types that are the start of a statement or declaration. Let's define them. Put above function ParseBlock the following constants:

```
const
  StmtStartSet: TTokenTypSet = [ttPrint, ttIdentifier];
```

```
const
  DeclStartSet: TTokenTypSet = [ttLet, ttVar];
```

Define them under separate const sections.

Next, in uToken.pas we have defined TTokenTypSet as a type. Let's add a helper function contains() that checks whether a certain token type belongs to the set. So, in uToken, add:

```
  TTokenTypSetHelper = type helper for TTokenTypSet
    function Contains(TokenTyp: TTokenTyp): Boolean;
  end;


function TTokenTypSetHelper.Contains(TokenTyp: TTokenTyp): Boolean;
begin
  Result := TokenTyp in Self;
end;
```

Now we can rewrite the function ParseBlock such that it uses the above defined constants and helper function to create a readable function.

The new ParseBlock therefore is:

```
function TParser.ParseBlock: TBlock;
begin
  Result := TBlock.Create(TNodeList.Create(), CurrentToken);
  while (DeclStartSet + StmtStartSet).Contains(CurrentToken.Typ) do
    Result.Nodes.Add(ParseNode);
end;
```

The block is created and inside then constructor we create an empty NodeList. The condition in the while statement seems complex. In Pascal you concatenate two sets with a '+' operator, thus creating a new set. We group it together in a parenthesized expression and using dot (.) notation we call function Contains(), which returns True if the token type of CurrentToken is in the concatenated set.

The mentioned function ParseNode handles the parsing of the respective statements and declarations.

```
function TParser.ParseNode: TNode;
begin
  try
    if DeclStartSet.Contains(CurrentToken.Typ) then
      Result := ParseDecl
    else if StmtStartSet.Contains(CurrentToken.Typ) then
      Result := ParseStmt
    else
      Error(CurrentToken, ErrUnrecognizedDeclOrStmt);
  except
    Synchronize(DeclStartSet + StmtStartSet + [ttEOF]);
    Result := TNode.Create(CurrentToken);
  end;
end;
```

Function ParseNode is quite powerful. It not only parses statements and declarations and generates a parsing error, but also synchronizes the parser to a follow up token, so that it can continue parsing and possibly collect other errors.

Forn this to be possible, change procedure Error() such that it raises an exception:

```
procedure TParser.Error(Token: TToken; Msg: string);
begin
  Errors.Append(Token.Line, Token.Col, Msg);
  Raise EParseError.Create(Msg);
end;
```

Any exception raised within the try-except block is now caugth and will be synchronized.

Function ParseProduct reuses the functionality of ParseBloc:

```
function TParser.ParseProduct: TProduct;
var
  Token: TToken;
begin
  Token := CurrentToken;
  Result := TProduct.Create(ParseBlock.Nodes, Token);
end;
```

We can already add the printer visitors for Block and Product.
To the printer unit add the visitor for Block:

```
procedure TPrinter.VisitBlock(Block: TBlock);
var
  Node: TNode;
begin
  IncIndent;
  VisitNode(Block);
  for Node in Block.Nodes do
    Visit(Node);
  DecIndent;
end;
```

And the visitor for Product:

```
procedure TPrinter.VisitProduct(Product: TProduct);
var
  Node: TNode;
begin
  IncIndent;
  VisitNode(Product);
  for Node in Product.Nodes do
    Visit(Node);
  DecIndent;
end;
```

Both loop through all nodes and visit them. Don't forget to add the functions to the published section in the interface of TPrinter.
Also, replace TExpr by TProduct in the interface so that it looks like this:

```
  TPrinter = class(TVisitor)
    private
      const Increase = 2;
      var
        Indent: string;
        Tree: TProduct;
      procedure IncIndent;
      procedure DecIndent;
    public
      constructor Create(ATree: TProduct);
      procedure Print;
    published
      procedure VisitNode(Node: TNode);
      //expressions
      procedure VisitBinaryExpr(BinaryExpr: TBinaryExpr);
      procedure VisitConstExpr(ConstExpr: TConstExpr);
      procedure VisitUnaryExpr(UnaryExpr: TUnaryExpr);
      //statements
      //declarations
      //block
      procedure VisitBlock(Block: TBlock);
      procedure VisitProduct(Product: TProduct);
  end;
```

In the constructor change TExpr to TProduct.

Next, we go to the interpreter. For now both visitors of Block and Product are alike:

```
procedure TInterpreter.VisitBlock(Block: TBlock);
var
  Node: TNode;
begin
  for Node in Block.Nodes do
    Visit(Node);
end;

procedure TInterpreter.VisitProduct(Product: TProduct);
var
  Node: TNode;
begin
  for Node in Product.Nodes do
    Visit(Node);
end;
```

All nodes are visited, or rather executed. For the time being we'll not use the block, as it is needed later on in control statements and functions.

Finally, change the Parser.Parse function to the following in order to parse the complete product:

```
function TParser.Parse: TProduct;
begin
  Result := ParseProduct;
end;
```

You may have noticed that in function ParseNode() I used the constant 'ErrUnrecognizedDeclOrStmt' in call to Error(). From now on I will do this for all error messages, and the respective string constants are defined in the implementation section, like this:

```
implementation

const
  ErrSyntax = 'Syntax error, "%s" expected.';
  ErrUnexpectedToken = 'Unexpected token: %s.';
  ErrDuplicateTerminator = 'Duplicate terminator not allowed.';
  ErrUnexpectedAttribute = 'Unexpected attribute "%s:".';
  ErrInvalidAssignTarget = 'Invalid assignment target.';
  ErrExpectedAssignOp = 'Expected assignment operator.';
  ErrUnrecognizedDeclOrStmt = 'Unrecognized declaration or statement.';
```

In the respective functions and procedures change the text messages with the constants.

We cannot run nor compile the code yet. Let's move on to statements.

Make a statement

A statement is a syntactic unit of an imperative programming language that expresses some action to be carried out. A program written in such a language is formed by a sequence of one or more statements. A statement may have internal components (e.g., expressions).

> A statement is persuasive and credible either because it is directly self-evident or because it appears to be proved from other statements that are so.
>
> — Aristotle

Many imperative languages (e.g. Pascal, C) make a distinction between statements and definitions, with a statement only containing executable code and a definition instantiating an identifier, while an expression evaluates to a value only. A distinction can also be made between simple and compound statements; the latter may contain statements as components (WikiPedia).

We start by adjusting the AST, and add basic statement classes to it. In unit uAST.pas add the following code in the interface section, just above the TNodeList type.

We first define a base statement type that inherits from TNode:

```
  TStmt = class(TNode)
    // Base class for statements
  end;
```

All other statements, such as PrintStmt and AssignStmt inherit from this parent statement type.

The first statement that we are going to develop is the print statement. We need this in order to see any test results.

According the EBNF, a print statement takes a list of expressions between parenthesis and separated by comma's. Some examples:

```
print('Hello world!')
print('The answer is ', 42)
print('This line ends with a newline ', terminator: '\n')
print('This line does not ', terminator: '')
print() // default new line
```

From the examples a couple of things are noticed. The print statement takes expressions as arguments, separated by comma's and default it will always print a new line. It carries a parameter 'terminator', which accepts an expression. The default value of terminator = '\n', which is the newline character.

Before we move on we create a new type in the AST: TExprList that maintains a list of expressions. Besides the print statement it will be used in more places later on:

```
  TExprList = specialize TArrayObj<TExpr>;
```

Put it right under the definition of TExpr, to keep it together.

Next, the AST node of the print statement.

```
  TPrintStmt = class(TStmt)
    private
      FExprList: TExprList;
      FTerminator: TExpr;
    public
      property ExprList: TExprList read FExprList;
      property Terminator: TExpr read FTerminator;
      constructor Create(AExprList: TExprList;
        ATerminator: TExpr; AToken: TToken);
      destructor Destroy; override;
  end;
```

The implementation is as follows:

```
constructor TPrintStmt.Create
  (AExprList: TExprList; ATerminator: TExpr; AToken: TToken);
begin
  Inherited Create(AToken);
  FExprList := AExprList;
  FTerminator := ATerminator;
end;


destructor TPrintStmt.Destroy;
begin
  if Assigned(FExprList) then FExprList.Free;
  if Assigned(FTerminator) then FTerminator.Free;
  inherited Destroy;
end;
```

The printer visitor is straightforward:

```
procedure TPrinter.VisitPrintStmt(Node: TPrintStmt);
var
  Expr: TExpr;
begin
  IncIndent;
  VisitNode(Node);
  for Expr in Node.ExprList do
    Visit(Expr);
  DecIndent;
end;
```

On to the parser.The easy part will be the parsing of the expressions. The harder part will be the terminator. The terminator argument can only appear one time! Let's try. First, we'll write the pseudocode for the parser function.

Set Terminator to default new line

Expect an open parenthesis

Parse an item and check if it is an expression or terminator attribute

While the next token is a comma do

  Parse an item and check if it is an expression or terminator attribute

Expect a closing parenthesis

```
function TParser.ParsePrintStmt: TStmt;
var
  ExprList: TExprList;
  Token: TToken;
  Terminator: TExpr;
  SawTerminator: Boolean = False;

  procedure ParseItem;
  var
    PrintPretty: TIdent;
  begin
    if (CurrentToken.Typ = ttIdentifier) and (Peek.Typ = ttColon) then begin
      PrintPretty := ParseIdent;
      if PrintPretty.Text = 'terminator' then begin
        if SawTerminator then
          Error(PrintPretty.Token, ErrDuplicateTerminator);
        SawTerminator := True;
        Next; // skip :
        Terminator := ParseExpr;
      end
      else
        Error(PrintPretty.Token,
          Format(ErrUnexpectedAttribute, [PrintPretty.Text]));
    end
    else ExprList.Add(ParseExpr);
  end;

begin
  Terminator := TConstExpr.Create('\n', CurrentToken);
  Token := CurrentToken;
  Next; // skip print
  Expect(ttOpenParen);
  ExprList := TExprList.Create(false);
  if CurrentToken.Typ <> ttCloseParen then begin
    ParseItem;
    while CurrentToken.Typ = ttComma do begin
      Next; // skip ,
      ParseItem;
    end;
  end;
  Expect(ttCloseParen);
  Result := TPrintStmt.Create(ExprList, Terminator, Token);
end;
```

Try to grab what's happening. For each item to print, the parser checks if it is the terminator. This means it can be anywhere in the print statement list of expressions. But it can only appear once. We use the variable SawTerminator for this. That's pretty flexible!

We use a nested procedure ParseItem, which does the tough part of the parsing. It first checks if the current token is an identifier followed by a colon. If so, we expect it to be the terminator attribute, otherwise an error is generated. If there is no colon detected we'll treat it as an expression, which is added to the ExprList.

In the future we might need to add other attributes, for example, a separator or something else. This nested procedure ParseItem makes it possible.

I above function ParsePrintStmt, we use AST node TIdent and parse function ParseIdent. They will be used a lot in the next chapters, som let's define them.

The AST node for TIdent is:

```
  TIdent = class(TNode)
    private
      FText: String;
    public
      property Text: String read FText;
      constructor Create(AToken: TToken);
  end;


constructor TIdent.Create(AToken: TToken);
begin
  inherited Create(AToken);
  FText := AToken.Lexeme;
end;
```

Function ParseIdent can be defined as follows:

```
function TParser.ParseIdent: TIdent;
var
  Token: TToken;
begin
  Token := CurrentToken;
  Expect(ttIdentifier);
  Result := TIdent.Create(Token);
end;
```

The printer visitor is:

```
procedure TPrinter.VisitIdent(Ident: TIdent);
begin
  IncIndent;
  WriteLn(Indent + 'Ident: ' + Ident.Text);
  DecIndent;
end;
```

What needs to be done to execute the print statement? Let's create the visitor for the interpreter.

```
procedure TInterpreter.VisitPrintStmt(PrintStmt: TPrintStmt);
var
  Value: String='';
  Terminator: String;
  i: Integer;
begin
  for i := 0 to PrintStmt.ExprList.Count-1 do begin
    Value := VarToStrDef(Visit(PrintStmt.ExprList[i]), 'Null');
    Value := StringReplace(Value, '\n', LineEnding, [rfReplaceAll]);
    Value := StringReplace(Value, '\t', #9, [rfReplaceAll]);
    Write(Value);
  end;
  Terminator := VarToStr(Visit(PrintStmt.Terminator));
  Terminator := StringReplace(Terminator, '\n', LineEnding, [rfReplaceAll]);
  Terminator := StringReplace(Terminator, '\t', #9, [rfReplaceAll]);
  Write(Terminator);
end;
```

The main loop handles all expressions from the PrintStmt's ExprList. Function VarToStrDef() is a standard function from the Free Pascal Variants unit, and it returns a string from the variant value. Also, function StringReplace() is a standard function (sysutils unit).

We are far from complete, but we can do simple test, to see if the print statement works. In function ParseBlock(), replace the call to ParseNode by ParsePrintStmt, like this:

```
function TParser.ParseBlock: TBlock;
begin
  Result := TBlock.Create(TNodeList.Create(), CurrentToken);
  while (DeclStartSet + StmtStartSet).Contains(CurrentToken.Typ) do
    Result.Nodes.Add(ParsePrintStmt);
end;
```

Finally, a small change to visitor Execute in the interpreter:

```
procedure TInterpreter.Execute(Tree: TProduct);
begin
  try
    Visit(Tree);
  except
    on E: ERuntimeError do
      RuntimeError(E);
  end;
end;
```

Values are now printed using the print statement!

Now it's possible to compile and the run the interpreter. Test it with a few print statements:

```
Gear> print(8*8, terminator: '!!!!\n')
64!!!!

Gear> print('The answer is ', 42)
The answer is 42
```

Desperately need variables

> To be is to be the value of a variable.
> — Willard Van Orman Quine

A variable is a storage location paired with an associated symbolic name (an identifier), which contains some known or unknown quantity of information referred to as a value. The variable name is the usual way to reference the stored value; this separation of name and content allows the name to be used independently of the exact information it represents. The identifier in computer source code can be bound to a value during run time, and the value of the variable may thus change during the course of program execution (WikiPedia).

In Gear a variable is declared as either a variable or a constant:

```
EBNF:

VarDecl = ‘var’ Ident ‘:=’ Expr
LetDecl = ‘let’ Ident ‘:=’ Expr
```

Examples are:

```
var Planet := 'Neptune'
let pi := 3.1415926
var radius := 2
var area := pi * radius^2
var Ready := False
let A := "A
var initial := Null
```

Each variable declaration needs to be preceded by the keyword ‘var’ or ‘let’. You may have noticed I’m talking about a declaration, and not so much about a statement. Should I omit the keyword ‘var’, it would be an assignment statement! Also, note that we don’t use any type information. All variable types are inferred from the expression. So, under water, variable Planet is of type String, pi is a Number, as are radius and area, Ready is a Boolean and A is a Char, while initial is undefined.

Variables in declarations

Continuing with VarDecl and LetDecl, we see they require an identifier. Create a AST node class for TIdent as well. In unit uAST add the following code, right below the definition of TConstExpr:

```
  TIdent = class(TNode)
    private
      FText: String;
    public
      property Text: String read FText;
      constructor Create(AToken: TToken);
  end;
```

and its implementation:

```
constructor TIdent.Create(AToken: TToken);
begin
  inherited Create(AToken);
  FText := AToken.Lexeme;
end;
```

Next, just below the definition of the statements add the definitions for the declarations:

```
  TDecl = class(TNode)
    // Base class for declarations
    private
      FIdent: TIdent;
    public
      property Ident: TIdent read FIdent;
      constructor Create(AIdent: TIdent; AToken: TToken);
      destructor Destroy; override;
  end;
```

All declarations have an identifier, so we'll add that property to the base declaration node. The implementation is:

```
constructor TDecl.Create(AIdent: TIdent; AToken: TToken);
begin
  Inherited Create(AToken);
  FIdent := AIdent;
end;

destructor TDecl.Destroy;
begin
  if Assigned(FIdent) then FIdent.Free;
  inherited Destroy;
end;
```

From the above EBNF, you'll see that VarDecl and LetDecl are practically the same. Can we combine them into one declaration type? We can if we maintain a field that defines the mutability of the declaration, so that a variable is mutable, but a constant is not.

```
  TVarDecl = class(TDecl)
    private
      FExpr: TExpr;
      FMutable: Boolean;
    public
      property Expr: TExpr read FExpr;
      property Mutable: Boolean read FMutable;
      constructor Create(AIdent: TIdent; AExpr: TExpr; AToken: TToken;
        AMutable: Boolean=True);
      destructor Destroy; override;
  end;
```

The implementation for above class is here:

```
constructor TVarDecl.Create
  (AIdent: TIdent; AExpr: TExpr; AToken: TToken; AMutable: Boolean);
begin
  Inherited Create(AIdent, AToken);
  FExpr := AExpr;
  FMutable := AMutable;
end;

destructor TVarDecl.Destroy;
begin
  if Assigned(FExpr) then FExpr.Free;
  inherited Destroy;
end;
```

Next, let's create the parse functionality. Remember that so far we could only parse ttPrint token types. We have defined the two start sets for statements and declarations, and we're going to use them now. Right below the StmtStartSet, add the following ParseStmt function, like this:

```
const
  StmtStartSet: TTokenTypSet = [ttPrint, ttIdentifier];

function TParser.ParseStmt: TStmt;
begin
  case CurrentToken.Typ of
    ttPrint: Result := ParsePrintStmt;
    else
      Result := ParseAssignStmt;
  end;
end;
```

Function ParsePrintStmt was already defined previously, but ParseAssignment not yet. For now create a stub function:

```
function TParser.ParseAssignStmt: TStmt;
begin
  Result := TStmt.Create(CurrentToken);
end;
```

It doesn't do anything yet, but we can test the other code.

Like function ParseStmt, we also create a function ParseDecl. It is responsible for parsing all possible declaration types. Next to 'var' and 'let', there will be many more, such as 'class', 'func', 'val', etc. And inside classes we will reuse this functionality!

```
const
  DeclStartSet: TTokenTypSet = [ttLet, ttVar];

function TParser.ParseDecl: TDecl;
const Mutable = True;
begin
  case CurrentToken.Typ of
    ttLet: Result := ParseVarDecl(Mutable = False);
    ttVar: Result := ParseVarDecl(Mutable = True);
  end;
end;
```

To make clear that it's a VarDecl or LetDecl, the constant Mutable is passed to ParseVarDecl with a boolean test: True or False. The one thing that is used in above function but not yet defined is function ParseVarDecl. Define it just below ParseDecl.

```
function TParser.ParseVarDecl(Mutable: Boolean): TDecl;
var
  Ident: TIdent;
  Token: TToken;
  Expr: TExpr;
begin
  Token := CurrentToken;
  if Mutable then
    Expect(ttVar)
  else Expect(ttLet);
  Ident := ParseIdent;
  Expect(ttAssign);
  Expr := ParseExpr;
  Result := TVarDecl.Create(Ident, Expr, Token, Mutable);
end;
```

Remember the EBNF (('var' | 'let') Ident ':=' Expr). After skipping the 'var'/'let' keyword, the identifier is parsed. Then, we expect an assignment operator, followed by an expression.

```
procedure TPrinter.VisitVarDecl(VarDecl: TVarDecl);
begin
  IncIndent;
  VisitNode(VarDecl);
  Visit(VarDecl.Ident);
  Visit(VarDecl.Expr);
  DecIndent;
end;
```

As a last step in these list of changes we modify TInterpreter in unit uInterpreter. Add the following procedures to the published section:

```
procedure TInterpreter.VisitIdent(Ident: TIdent);
begin
  // do nothing
end;
```

Procedure VisitIdent requires no action, but it's better to keep visitors for all AST nodes, so we never run into a runtime error if it's accidentilly called.

```
procedure TInterpreter.VisitVarDecl(VarDecl: TVarDecl);
begin
  // do nothing yet!
end;
```

For the moment I keep VisitVarDecl empty, but we'll soon get to that! At least, we can now compile the complete program and test for any compile errors. There are none ☺!.

Variables in expressions

So, now we can declare variables, but there's no way to use them in expressions yet. You would like to do something like the following:

```
let a := 3
let b := 4
var c := (a^2 + b^2)^0.5 // calculation of Pythagoras; c = 5
```

Here's a summary recap of the expression EBNF, which now includes Variables:

```
    Expr       = AddExpr [ RelOp AddExpr ] .
    AddExpr    = MulExpr { AddOp MulExpr } .
    MulExpr    = ShiftExpr { MulOp ShiftExpr } .
    ShiftExpr  = UnaryExpr  { ExpOp UnaryExpr  } .
    UnaryExpr  = [ '+' | '-' | '!' ] UnaryExpr | Factor  .
    Factor     =  Constant
               | '(' Expr ')'
               | Variable .
    Variable   = Identifier .
```

We now implement that missing part of the Variable. Start with adding it to the AST, as a descendent of TFactorExpr. Put it below TIdent, but above TStmt.

```
  TVariable = class(TFactorExpr)
    private
      FIdent: TIdent;
    public
      property Ident: TIdent read FIdent;
      constructor Create(AIdent: TIdent);
      destructor Destroy; override;
  end;
```

Followed by its implementation:

```
constructor TVariable.Create(AIdent: TIdent);
begin
  Inherited Create(AIdent.Token);
  FIdent := AIdent;
end;

destructor TVariable.Destroy;
begin
  if Assigned(FIdent) then FIdent.Free;
  inherited Destroy;
end;
```

In uParser add to function ParseFactor the following line, as the first in the case statement (I try to put them in order of appearance in the TTokenTyp enum for compiler optimization):

```
function TParser.ParseFactor: TExpr;
begin
  case CurrentToken.Typ of
    ttIdentifier: Result := TVariable.Create(ParseIdent);
    ...
  end;
end;
```

If we find an identifier, then the parse result is a Variable that takes an identifier as its parameter. We already created function ParseIdent, so we reuse it here.

This leaves us with the creation of two more visitors, one in TPrinter and one in tInterpreter.

```
procedure TPrinter.VisitVariable(Variable: TVariable);
begin
  IncIndent;
  WriteLn(Indent, 'Var: ', Variable.Ident.Text);
  DecIndent;
end;
```

This one's easy and straightforward, as we just print the identifier's name.

```
function TInterpreter.VisitVariable(Variable: TVariable): Variant;
begin
  // do nothing yet
  Result := Null;
end;
```

The interpreter visitor we leave blank for now, as we need something to use variables with: a Memory model! We require the possibility to load, store and update variable values in order to create reusable variables.

Memory: Storing, loading and updating variables

At the start of this paragraph, in the WikiPedia text, there's mentioning of runtime binding of variables to values. We need to create a storage place for the values of declared variables, so that we can easily retrieve them again. Let's call this the memory space! Create a new unit uMemory.pas.

```
unit uMemory;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, uCollections, Variants, uAST, uToken, uError;

type

  TMemorySpace = class(specialize TDictionary<String, Variant>)
    public
      constructor Create;
      procedure Store(Name: String; Value: Variant);
      procedure Store(Ident: TIdent; Value: Variant);
      function Load(Name: String; Token: TToken): Variant;
      function Load(Ident: TIdent): Variant;
      procedure Update(Name: String; Value: Variant; Token: TToken);
      procedure Update(Ident: TIdent; Value: Variant);
  end;
```

The memory space is a TDictionary class, which we specialize to String for the keys and Variant for the data. Procedure Store, well, stores a value for a given identifier, whereas function Load retrieves the value of a given identifier. Procedure Update updates an existing variable.

The implementation is here:

```
implementation

const
  ErrVarUndefined = 'Variable "%s" is undefined.';
```

In the constructor set Sorted to True, so we have a faster lookup of variables.

```
constructor TMemorySpace.Create;
begin
  inherited Create;
  Sorted := True;
end;
```

```
procedure TMemorySpace.Store(Name: String; Value: Variant);
begin
  Self[Name] := Value;
end;

procedure TMemorySpace.Store(Ident: TIdent; Value: Variant);
begin
  Store(Ident.Text, Value);
end;
```

```
function TMemorySpace.Load(Name: String; Token: TToken): Variant;
var
  i: LongInt = -1;
begin
  if Find(Name, i) then     // Find is a standard FGL function to find a key
    Result := At(i)
  else
    Raise ERuntimeError.Create(Token, Format(ErrVarUndefined, [Name]));
end;

function TMemorySpace.Load(Ident: TIdent): Variant;
begin
  Result := Load(Ident.Text, Ident.Token);
end;

procedure TMemorySpace.Update(Name: String; Value: Variant; Token: TToken);
var
  i: LongInt = -1;
begin
  if Find(Name, i) then
    Data[i] := Value
  else
    Raise ERuntimeError.Create(Token, Format(ErrVarUndefined, [Name]));
end;

procedure TMemorySpace.Update(Ident: TIdent; Value: Variant);
begin
  Update(Ident.Text, Value, Ident.Token);
end;

end.
```

Noteworthy is that Store doesn't look if a variable exists or not. However, when we declare a variable, we have to make sure it doesn't exist yet. As mentioned, we won't allow redeclaration of variables, as it makes the code unclear. This is solved in the interpreter.

On the other hand, function Load reports an error if the variable doesn't exist. It tries to find the identifier first, and returns its value if found. But if not found, we have a programming error!

Procedure Update does check if a variable exists. If not, it's a programming error!

As a next step, we define the memory in the interpreter. Add a new private field 'CurrentSpace' to TInterpreter, and also add the unit uMemory to the uses clause.

```
  TInterpreter = class(TVisitor)
    private
      CurrentSpace: TMemorySpace;
    public
      constructor Create;
      destructor Destroy; override;
      procedure Execute(Tree: TProduct);
      ...
```

So far there was no create constructor necessary, but now we do, as we have to create the memory space now. Also add a destructor, in order to remove the memory when the program closes.

```
constructor TInterpreter.Create;
begin
  CurrentSpace:= TMemorySpace.Create;
end;
```

```
destructor TInterpreter.Destroy;
begin
  CurrentSpace.Free;
  inherited Destroy;
end;
```

Remember, we left the visitor for VarDecl empty so far. This was because we didn't have a memory space yet. Now we do, we can fulfil our promise to come back to it.

```
procedure TInterpreter.VisitVarDecl(VarDecl: TVarDecl);
begin
  CheckDuplicate(VarDecl.Ident, 'Variable');
  CurrentSpace.Store(VarDecl.Ident, Visit(VarDecl.Expr));
end;
```

Here, we first check if the variable already exists, and since we don't want redeclaration of variables, we only store it if there is no variable with the same name. This is handles by procedure CheckDuplicate(), which is declared as private:

```
procedure TInterpreter.CheckDuplicate(AIdent: TIdent; const TypeName: String);
begin
  if CurrentSpace.Contains(AIdent.Text) then
    Raise ERuntimeError.Create(AIdent.Token,
      Format(ErrDuplicateID, [TypeName, AIdent.Text]));
end;
```

At the top of the implementation section we'll add all error messages, defined as constants:

```
const
  ErrDuplicateID = '%s cannot be defined. Identifier "%s" is already declared.';
```

What's left is the loading of variables. The stub for VisitVariable() was already there. We are now actually going to load the data from the memory.

```
function TInterpreter.VisitVariable(Variable: TVariable): Variant;
begin
  Result := CurrentSpace.Load(Variable.Ident);
end;
```

Compile, and then run the Gear interpreter. As input file, use variable declarations and then print them, e.g.:

```
let a := 3
let b := 4
let c := (a^2 + b^2)^0.5
print(c)
```

This prints 5. How cool is that?

It works in the command mode (or REPL) as well:

```
Gear> var a := 3.1415926
Gear> print(a)
3.1415926
Gear>
```

And if we try to redeclare a variable in the REPL:

```
Gear> var a := 1
Gear> var a := 2
@[1,5] Runtime error: Variable "a" already declared.
```

Multiple declarations and calculations:

```
Gear> var a := 3
Gear> var b := 4
Gear> var c := (a^2+b^2)^0.5
Gear> print('Pythagoras of ', a, ' and ', b, ' is ', c)
Pythagoras of 3 and 4 is 5
Gear>
```

Here's your assignment

An assignment statement sets and/or re-sets the value stored in the storage location(s) denoted by a variable name; in other words, it copies a value into the variable. In most imperative programming languages, the assignment statement (or expression) is a fundamental construct.

> Great ability develops and reveals itself increasingly with every new assignment.
> — Baltasar Gracian

Today, the most commonly used notation for this basic operation has come to be 'x = expr' followed by 'x := expr', although there are many other notations in use. In some languages the symbol used is regarded as an operator (meaning that the assignment has a value) while others define the assignment as a statement (meaning that it cannot be used in an expression).

Assignments typically allow a variable to hold different values at different times during its life-span and scope (WikiPedia).

In this book I use the ':=' operator for plain assignments, and an assignment will be a statement, instead of an expression. We also defined in unit uToken the following assignment operators: '+=', '-=', '*=', '/=' and '%='. Examples of assignment statements are:

```
a := 1
b := a + 2

a := a + 1
which is the same as
a += 1
```

Parsing the assignment

So, an assignment consists of a variable, an operator and an expression, resulting in the EBNF:

```
AssignStmt = Variable AssignOp Expression .
AssignOp . = ':=' | '+=' | '-=' | '*=' | '/='| '%=' .
```

The variable must be an already declared variable. If not an error occurs.

The AST node for assignment statement is:

```
  TAssignStmt = class(TStmt)
    private
      FVariable: TVariable;
      FOp: TToken;
      FExpr: TExpr;
    public
      property Variable: TVariable read FVariable;
      property Op: TToken read FOp;
      property Expr: TExpr read FExpr;
      constructor Create(AVariable: TVariable; AOp: TToken; AExpr: TExpr);
      destructor Destroy; override;
  end;
```

Put it right below TPrintStmt. Note the use of type TVariable for a Variable. I could have taken TIdent as the type as it just concerns the name we want here, however, it will be usefull in some cases to retrieve the current value of the variable via the VisitVariable visitor in TInterpreter.

The implementation of TAssignStmt is:

```
constructor TAssignStmt.Create(AVariable: TVariable; AOp: TToken; AExpr: TExpr);
begin
  inherited Create(AOp);
  FVariable := AVariable;
  FOp := AOp;
  FExpr := AExpr;
end;
```

```
destructor TAssignStmt.Destroy;
begin
  if Assigned(FVariable) then FVariable.Free;
  if Assigned(FExpr) then FExpr.Free;
  inherited Destroy;
end;
```

Don’t free the FOp variable, as it is automatically freed! You know where?

In unit uParser, we change the function ParseAssignStmt stube to the following:

```
const
  AssignSet: TTokenTypSet =
    [ttPlusIs, ttMinIs, ttMulIs, ttDivIs, ttRemIs, ttAssign];

function TParser.ParseAssignStmt: TStmt;
var
  Token, Op: TToken;
  Left, Right: TExpr;
begin
  Token := CurrentToken;
  Left := ParseExpr;
  if CurrentToken.Typ in AssignSet then begin
    Op := CurrentToken;
    Next; // skip assign token
    Right := ParseExpr;
    if Left is TVariable then
      Result := TAssignStmt.Create(Left as TVariable, Op, Right)
    else
      Error(Token, ErrInvalidAssignTarget);
  end
  else
    Error(CurrentToken, ErrExpectedAssignOp);
end;
```

First, the variable is parsed as an expression, followed by checking if the next token is one of the assignment operators. If it is an assignment operator, the Right expression is parsed. If the Left expression is a variable expression, we have an assignment. If not an error is produced.

In TPrinter add the visitor to the published section, right below VisitPrintStmt:

```
procedure TPrinter.VisitAssignStmt(AssignStmt: TAssignStmt);
begin
  IncIndent;
  VisitNode(AssignStmt);
  WriteLn(Indent, '(', AssignStmt.Op.Typ.toString, ')');
  Visit(AssignStmt.Variable);
  Visit(AssignStmt.Expr);
  DecIndent;
end;
```

Interpreting assignment

And finally, the visitor for the interpreter assignment becomes:

```
procedure TInterpreter.VisitAssignStmt(AssignStmt: TAssignStmt);
var
  OldValue, NewValue, Value: Variant;
begin
  with AssignStmt do begin
    OldValue := Lookup(Variable);
    NewValue := Visit(Expr);
    Value := getAssignValue(OldValue, NewValue, Variable.Token, Op);
    Assign(Variable, Value);
  end;
end;
```

This seemingly small procedure contains a lot of logic. We first look up the old value of the variable, followed by calculating the new value. Lookup is a private function.

```
function TInterpreter.Lookup(Variable: TVariable): Variant;
begin
  Result := CurrentSpace.Load(Variable.Ident);
end;
```

In function getAssignValue() we do some checks, e.g. on type compatibility, and apply the correct assign operator. It is a local function in the implementation section just above visitor VisitAssignStmt().

Function TypeOf(), which is used in getAssignValue() returns the name of the type of the value:

```
function TypeOf(Value: Variant): String;
begin
  case VarType(Value) of
    varNull: Result := 'Null';
    varSingle, varDouble: Result := 'Number';
    varString: Result := 'String';
    varBoolean: Result := 'Boolean';
    varShortInt, varSmallInt, varInteger, varInt64,
    varByte, varWord, varLongWord, varQWord: Result := 'Number';
    else
      Result := 'Unknown';
  end;
end;
```

```
function getAssignValue(OldValue, NewValue: Variant; ID, Op: TToken): Variant;
var
  OldType, NewType: String;
begin
  OldType := TypeOf(OldValue);
  NewType := TypeOf(NewValue);
  if VarIsNull(OldValue) and (Op.Typ = ttAssign) then
    Exit(NewValue);
  if VarIsNull(NewValue) and (Op.Typ = ttAssign) then
    Exit(Null);

  if OldType <> NewType then
    Raise ERuntimeError.Create(ID,
      Format(ErrIncompatibleTypes, [OldType, NewType]));

  if not VarIsNull(OldValue) then begin
    if Op.Typ <> ttAssign then
    case Op.Typ of
      ttPlusIs: NewValue := TMath._Add(OldValue, NewValue, Op);
      ttMinIs:  NewValue := TMath._Sub(OldValue, NewValue, Op);
      ttMulIs:  NewValue := TMath._Mul(OldValue, NewValue, Op);
      ttDivIs:  NewValue := TMath._Div(OldValue, NewValue, Op);
      ttRemIs:  NewValue := TMath._Rem(OldValue, NewValue, Op);
    end;
    Exit(NewValue);
  end;

  Raise ERuntimeError.Create(ID,
    Format(ErrIncompatibleTypes, [OldType, NewType]));
end;
```

It looks complicated, let's explain what happens. First, we evaluate the OldValue of the Variable. Then we evaluate the expression, alias the NewValue. If the OldValue is Null and the normal assign operator is used we allow reinitialisation of the variable. This is allowed:

```
var n := Null
n := 100
n := True  // this is not allowed anymore
n := Null   // also this is not allowed anymore
```

So, you can initialize a variable with Null, and then assign a new value to it. From this moment on the variable has a type attached to it, and this type can no longer be changed. It's possible to set the value back to Null.

Then, if the old value is other than Null, we check if the types of the old and new values are the same.

Finally, we calculate the new value based on the operator. This is one of the operators +=, -=, *=, /= or %=. Note that regular assignment is automatically taken care of here. For example this is not allowed:

```
var n := 100
n := True  // this is not allowed, incompatible types in assignment
```

Since we use procedure Exit in a few cases, which in other languages is called Return, we can code the error handling right at the bottom of the method. It provides an error message, including the type names that do not match.

In this part we use some standard functions from unit Variants:

- VarIsNull(OldValue) returns True if the type of OldValue is Null.
- VarType(OldValue) returns the variant type of the value, eg. varNull.

Finally, the Assign() procedure, which simply updates the memory location of the variable:

```
procedure TInterpreter.Assign(Variable: TVariable; Value: Variant);
begin
  CurrentSpace.Update(Variable.Ident, Value);
end;
```

Regarding the Lookup function, we can also use this in the visitor for a variable expression:

```
function TInterpreter.VisitVariable(Variable: TVariable): Variant;
begin
  Result := Lookup(Variable);
end;
```

For example, this now works:

```
var x := Null
print(x)
x := 3>9
print(x)

var a := Null
a:=6
a+=5
print(a)

var b := 'b'
b+=a          // implicit conversion of a from number to string
print(b)
print(True|False)
```

(Im)mutable Let

You may have tested it already, or maybe not yet, but at this moment it is possible to assign new values to constants. And this should not be the case obviously. The problem however, is not solved easily, since there is no connection between a TVarDecl and a TVariable. This connection we have to create somehow. Usually this is done through a symbol table. In the next chapter we dive into the world of symbol tables, scopes, variable resolving, in order to prepare for the chapters thereafter that treat blocks in detail and functions.

Chapter 3 – Symbols, scope and resolving

In this chapter we'll create a new compiler pass, called semantic checking. Where the parser creates an AST and performs syntax checking, we now provide meaning through a new visitor class, called the Resolver. We'll also dive into the subject of scope, as variables can exist in a certain scope, but not outside this scope.

Blocks are in scope

> By looking at the questions the kids are asking, we learn the scope of what needs to be done.
> — Buffy Sainte-Marie

The scope of a name binding – an association of a name to an entity, such as a variable – is the region of a computer program where the binding is valid: where the name can be used to refer to the entity. Such a region is referred to as a scope block. In other parts of the program the name may refer to a different entity (it may have a different binding), or to nothing at all (it may be unbound) (WikiPedia).

A block is a group of declarations and statements, in which the declared variables exist during the lifetime of the block. After the block has finished, the declared variables are removed from memory.

Let's recall the EBNF of a block:

```
Product = Block .
Block = { Stmt | Decl } .          // zero or more declarations or statements
```

Concretely, this means that while we encounter a token from DeclStartSet or StmtStarSet we continue with parsing within the block, and as soon as we don't find a match from a token in the start sets, the block ends. Also note a Block contains the same nodes as the Product. In a block a variable can be defined, which was also defined in the global scope or in an enclosing scope, for example:

```
var a := 1
var b := 9
if a < 2 then
  // block start
  var a := 5
  print(a*b)
  // block end
end
print(a*b)
```

The variable declaration inside the if-statement is legal, and the first print statement should print 45, whereas the second one should print 9.

We have already created the AST node for a block as well as its parser and printer functions. Now we'll take it a step further.

We already have a memory space class that defines a global scope. All variables declared so far, remained in a global space scope. If a new scope is created, we must maintain a link to the enclosing space scope, in order to retrieve the value of variable b in above example code.

Let's add the necessary changes to TMemorySpace.

First, add a new private variable FEnclosingSpace of type TMemorySpace. Also, add the respective property EnclosingSpace. Then, change the constructor such that it accepts a parameter of type TMemorySpace with default value Nil. Finally, we change procedure Update and function Load, so they search in enclosing scopes.

```
  TMemorySpace = class(specialize TDictionary<String, Variant>)
    private
      FEnclosingSpace: TMemorySpace;
    public
      property EnclosingSpace: TMemorySpace read FEnclosingSpace;
      constructor Create(AEnclosingSpace: TMemorySpace = Nil);
      ...
  end;
```

```
constructor TMemorySpace.Create(AEnclosingSpace: TMemorySpace);
begin
  inherited Create;
  FEnclosingSpace := AEnclosingSpace;
end;
```

In procedure Load we start by looking for a variable in the current scope. If found we retrieve its value. If not found, we try to load it from the enclosing space. If the variable is not found in the complete chain of scopes, we return an error. Here's the complete function:

```
function TMemorySpace.Load(Name: String; Token: TToken): Variant;
var
  i: LongInt = -1;
begin
  if Contains(Name, i) then
    Result := At(i)
  else if Assigned(FEnclosingSpace) then
    Result := FEnclosingSpace.Load(Name, Token)
  else
    Raise ERuntimeError.Create(Token, Format(ErrVarUndefined, [Name]));
end;
```

The same concept is used for updating a value of a variable:

```
procedure TMemorySpace.Update(Name: String; Value: Variant; Token: TToken);
var
  i: LongInt = -1;
begin
  if Contains(Name, i) then
    Data[i] := Value
  else if Assigned(FEnclosingSpace) then
    FEnclosingSpace.Update(Name, Value, Token)
  else
    Raise ERuntimeError.Create(Token, Format(ErrVarUndefined, [Name]));
end;
```

Now we need to adjust the interpreter method for VisitBlock so that it can cope with scopes.

```
procedure TInterpreter.VisitBlock(Block: TBlock);
var
  Node: TNode;
  EnclosingSpace: TMemorySpace;
begin
  EnclosingSpace := CurrentSpace;
  try
    CurrentSpace := TMemorySpace.Create(EnclosingSpace);
    for Node in Block.Nodes do
      Visit(Node);
  finally
    CurrentSpace := EnclosingSpace;
  end;
end;
```

Since a Block creates a new scope, we basically create a new memory space which we use during the lifetime of the block. First, we have to save our current space. We set it to be the enclosing space. Then, in the try-finally statement, we create a new memory space and set it to be the current space, after we visit all block nodes. Finally, after we're done with the block, we return to the enclosing space and make it the current space again.

And this is all we need to change in the interpreter! Wow, that easy…

Now we move on to symbols.

Resolving identifiers

This paragraph is about resolving local identifiers, variable and function ID's in local scopes. Especially, the below program code is treated.

```
var x := 0

func def()
  var x := x
end

print(x)
```

> When a truth is necessary, the reason for it can be found by analysis, that is, by resolving it into simpler ideas and truths until the primary ones are reached.
> — Gottfried Leibniz

What should happen here? We'll treat it as an error!

Symbols in scope

We'll start of with setting up the global scope and setting up the symbol table.

Our symbol model is quite simple. A symbol just maintains an identifier and a notion of mutability. We'll use it for all types of symbols including functions and classes.

First, create a new unit uResolver.pas.

```
unit uResolver;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, uVisitor, uCollections, uAST, uToken, uError,
  Variants;

implementation

end.
```

Then, add the symbol class:

```
  TStatus = (Declared, Enabled);

  TSymbol = class
    Name: String;
    Status: TStatus;
    Mutable: Boolean;
    IsNull: Boolean;
    constructor Create(const AName: String; const AStatus: TStatus;
      const AMutable: Boolean=False);
  end;
```

A symbol has a name and a few properties, such as Status, for which Declared means the identifier is declared but not usable yet. Enabled means it's also defined and it may be referenced. Mutable, of course, means the variable can be changed, and isNull is True if the variable has a Null value.

The implementation of the constructor:

```
constructor TSymbol.Create(const AName: String; const AStatus: Boolean;
  const AMutable: Boolean);
begin
  Name := AName;
  Status := AStatus;
  Mutable := AMutable;
  isNull := False;
end;
```

The value of isNull is default set to False, for it can only get a value after being evaluated.

A symbol is always inside a scope. There are two types of scopes: local scope (e.g. inside a function) and global scope, which is the main program scope. For now we'll use the global scope. The scope is defined as:

```
  TScope = class(specialize TDictionary<String, TSymbol>)
    private
      FEnclosing: TScope;
    public
      property Enclosing: TScope read FEnclosing;
      constructor Create(AEnclosing: TScope=Nil);
      procedure Enter(ASymbol: TSymbol);
      function Lookup(AName: String): TSymbol;
  end;
```

Just like a memory space, a scope can have an enclosing scope. Furthermore it has a method to enter new symbols and a Lookup function to retrieve a symbol by its name. The implementation is as follows:

```
constructor TScope.Create(AEnclosing: TScope);
begin
  inherited Create;
  FEnclosing := AEnclosing;
end;

procedure TScope.Enter(ASymbol: TSymbol);
begin
  Self[ASymbol.Name] := ASymbol;
end;

function TScope.Lookup(AName: String): TSymbol;
var
  i: Integer;
begin
  i := IndexOf(AName);
  if i >= 0 then
    Result := Data[i]
  else if FEnclosing <> Nil then
    Result := FEnclosing.Lookup(AName)
  else
    Result := Nil;
end;
```

I used here FEnclosing <> Nil, instead of Assigned(FEnclosing). In fact, they are the same.

Resolving identifiers in the global scope

Now that we've defined symbols and a scope class, we put it to work in a resolver class. The resolver is a visitor class, so should inherit from TVisitor. Here's the interface, which you can put just under the TScope class:

```
  TResolver = class(TVisitor)
    public
      constructor Create;
      destructor Destroy; override;
      procedure Resolve(Tree: TProduct);
    published
      procedure VisitNode(Node: TNode);
      procedure VisitIdent(Ident: TIdent);
      // Expr
      procedure VisitBinaryExpr(BinaryExpr: TBinaryExpr);
      procedure VisitConstExpr(ConstExpr: TConstExpr);
      procedure VisitUnaryExpr(UnaryExpr: TUnaryExpr);
      procedure VisitVariable(Variable: TVariable);
      // Stmt
      procedure VisitPrintStmt(PrintStmt: TPrintStmt);
      procedure VisitAssignStmt(AssignStmt: TAssignStmt);
      // Decl
      procedure VisitVarDecl(VarDecl: TVarDecl);
      // Blocks
      procedure VisitBlock(Block: TBlock);
      procedure VisitProduct(Product: TProduct);
    private
      GlobalScope: TScope;
      CurrentScope: TScope;
      procedure Declare(Ident: TIdent; const isMutable: Boolean=False);
      procedure Enable(Ident: TIdent);
      function Retrieve(Ident: TIdent): TSymbol;
  end;
```

The published methods are exactly the same as in the Printer visitor class. There is a main method Resolve, which takes the AST as a whole as the parameter. Then, we have two TScope variables: GlobalScope and CurrentScope. The GlobalScope is the main program's scope (we'll have local scopes later on), but we'll always use the CurrentScope variable for performing any action.

Finally, there are some private methods. Declare() declares a variable, whereas Enable() enables it to be used. Finally, Retrieve() tries to read a symbol from the scope. Let's go through them.

In the implementation section start with the error constants:

```
const
  ErrCannotReadLocalVar = 'Cannot read local variable in its own declaration.';
  ErrCannotAssignToConstant = 'Cannot assign value to constant "%s".';
  ErrDuplicateIdInScope = 'Duplicate identifier "%s" in this scope.';
  ErrUndeclaredVar = 'Undeclared variable "%s".';
```

Then, the constructor and destructor:

```
constructor TResolver.Create;
begin
  GlobalScope := TScope.Create;
  GlobalScope.Sorted := True;
  CurrentScope := GlobalScope;
end;
```

```
destructor TResolver.Destroy;
begin
  GlobalScope.Free;
  inherited Destroy;
end;
```

The Resolve procedure starts the resolving process:

```
procedure TResolver.Resolve(Tree: TProduct);
begin
  Visit(Tree);
end;
```

The next procedures are straigtforward as well:

```
procedure TResolver.VisitNode(Node: TNode);
begin
  // do nothing
end;

procedure TResolver.VisitIdent(Ident: TIdent);
begin
  // do nothing
end;

procedure TResolver.VisitBinaryExpr(BinaryExpr: TBinaryExpr);
begin
  Visit(BinaryExpr.Left);
  Visit(BinaryExpr.Right);
end;

procedure TResolver.VisitConstExpr(ConstExpr: TConstExpr);
begin
  // do nothing
end;

procedure TResolver.VisitUnaryExpr(UnaryExpr: TUnaryExpr);
begin
  Visit(UnaryExpr.Expr);
end;
```

It starts to get more interesting with visitor for the variable:

```
procedure TResolver.VisitVariable(Variable: TVariable);
var
  Symbol: TSymbol;
begin
  Symbol := Retrieve(Variable.Ident);
  if Symbol <> Nil then begin
    if Symbol.Status = Declared then
      with Variable.Token do
        Errors.Append(Line, Col, ErrCannotReadLocalVar);
  end;
end;
```

We first try to retrieve the symbol. If it doesn't exist the Retrieve function generates an error. If it does exist we check if the status is Declared and if this is the case you are trying to something like this:

```
  var x := x
```

Which doesn't make sense, clearly.

The visitor for the print statement is:

```
procedure TResolver.VisitPrintStmt(PrintStmt: TPrintStmt);
var
  Expr: TExpr;
begin
  for Expr in PrintStmt.ExprList do
    Visit(Expr);
end;
```

If you print a variable, which doesn't exist this will generate an error.

The assignment statement is very interesting. We will do a check on mutability here, and if mutability is set to False, we also have to check if the value is Null.

```
procedure TResolver.VisitAssignStmt(AssignStmt: TAssignStmt);
var
  Symbol: TSymbol;
  Ident: TIdent;
begin
  Visit(AssignStmt.Expr);
  Ident := AssignStmt.Variable.Ident;
  Symbol := Retrieve(Ident);
  if Symbol <> Nil then begin
    if (not Symbol.Mutable) and (not Symbol.isNull) then
      Errors.Append(Ident.Token.Line, Ident.Token.Col,
        Format(ErrCannotAssignToConstant, [Ident.Text]))
    else if Symbol.isNull then
      Symbol.isNull := False;
  end;
end;
```

So, for example, this is allowed, except for the last assignment:

```
let a := Null
a := 99
print(a)
a := 0     // This is not allowed and generates an error
```

So, after the variable gets a value different than Null, it cannot be changed anymore, and it stays constant.

```
procedure TResolver.VisitVarDecl(VarDecl: TVarDecl);
var
  Symbol: TSymbol;
begin
  Declare(VarDecl.Ident, VarDecl.Mutable);
  Visit(VarDecl.Expr);
  Enable(VarDecl.Ident);

  if (VarDecl.Expr is TConstExpr) and
     ((VarDecl.Expr as TConstExpr).Token.Typ = ttNull) then
  begin
    Symbol := Retrieve(VarDecl.Ident);
    Symbol.isNull := True;
  end;
end;
```

If a variable is declared, the method Declare() adds the symbol to the scope with status Declared. Then, the VarDecl's expression is evaluated, and this is where we check that the same variable is not used in its own expression. If this passes, the variable is enabled and can be used and referenced to.

One other things that's done here, is the check on Null. The value Null is defined as a token type (ttNull), so if the expression is a constant expression, we can simply check the token type. In this case we set the symbol's isNull value to True.

Next, the methods for the Block and the Product.

```
procedure TResolver.VisitBlock(Block: TBlock);
var
  Node: TNode;
begin
  for Node in Block.Nodes do
    Visit(Node);
end;

procedure TResolver.VisitProduct(Product: TProduct);
var
  Node: TNode;
begin
  for Node in Product.Nodes do
    Visit(Node);
end;
```

For now, these are the visitors. Only, the private routines are left.

```
procedure TResolver.Declare(Ident: TIdent; const isMutable: Boolean);
begin
  if CurrentScope.Contains(Ident.Text) then
    Errors.Append(Ident.Token.Line, Ident.Token.Col, Format(
      ErrDuplicateIdInScope, [Ident.Text]));
  CurrentScope.Enter(TSymbol.Create(Ident.Text, Declared, isMutable));
end;
```

Function Declare() first checks if an identifier was already declared in the current scope and if so, generates an error. Otherwise the symbol is entered in the current scope.

Function Enable() searches the symbol in the scope and sets its status to Enabled:

```
procedure TResolver.Enable(Ident: TIdent);
var
  Symbol: TSymbol;
begin
  Symbol := Retrieve(Ident);
  if Symbol <> Nil then
    Symbol.Status := Enabled;
end;
```

And finally, the much use Retrieve() function, which first looks up the identifier in the current scope and which returns the accompanying symbol. If the symbol cannot be found, the result is Nil and an error message is generated.

```
function TResolver.Retrieve(Ident: TIdent): TSymbol;
begin
  Result := CurrentScope.Lookup(Ident.Text);

  if Result = Nil then
    Errors.Append(Ident.Token.Line, Ident.Token.Col, Format(
      ErrUndeclaredVar, [Ident.Text]));
end;
```

In unit uLanguage change class procedure Execute, so that it can cope with the Resolver:

```
class procedure Language.Execute(const Source: String; InputType: TInputType);
var
  ...
  Resolver: TResolver;
begin
  try
    ...
    Resolver := TResolver.Create;
    Resolver.Resolve(Tree);
    if not Errors.isEmpty then
      ...
  finally
    ...
    Resolver.Free;
  end;
end;
```

This ends the resolver for the global scope. You may test e.g.:

```
var a := a
print(b)
```

This results in two errors:

```
@[1,10]: Cannot read local variable in its own declaration.
@[2,7]: Undeclared variable "b".
```

Or:

```
let a := Null
a := 99
print(a)
a := 0
```

```
@[4,1]: Cannot assign value to constant "a".
```

Or:

```
let a := 3
let b := 4
var c := ((a*a)+(b*b))^0.5
print('a: ', a, terminator: ', ')
print('b: ', b, terminator: ', ')
print('c: ', c)
```

```
a: 3, b: 4, c: 5
```

Local scopes

Till now we only worked in the global scope of a program. In the next chapter we'll focus on actually using Blocks and thus creating local scopes. For example the if, while and for statements use blocks.
Then, we'll introduce a new field in the Variable AST node, called Distance, which allows for fast lookup during interpretation, instead of searching the variable through a tree of scopes.

But in order to build a scope stack, we first have to have a stack type. Luckily, this is available from the uCollections unit. It's purpose is to maintain a stack of TObject descendent types. You can push and pop objects to and from the stack, and you can have a peek at the top of the stack. There's also a function to check whether the stack is empty. We'll see it in use in the resolver.

First, create a new type right above TResolver:

```
  TScopes = specialize TStack<TScope>;
```

TScopes is a stack that maintains scopes with declared and enabled identifiers. Next, define a new private field Scopes in TResolver. Also, we need two new procedures: BeginScope and EndScope.

```
  TResolver = class(TVisitor)
    ...
    private
      Scopes: TScopes;
      GlobalScope: TScope;
      procedure BeginScope;
      procedure EndScope;
      ...
  end;
```

The implementation of BeginScope and EndScope is:

```
procedure TResolver.BeginScope;
begin
  Scopes.Push(TScope.Create(CurrentScope));
  CurrentScope := Scopes.Top;
end;

procedure TResolver.EndScope;
begin
  CurrentScope := Scopes.Top.Enclosing;
  Scopes.Pop;
end;
```

As we begin a new scope, it is created with the current scope as the enclosing scope. Then it's pushed on the stack, and the current scope becomes the top of the stack.

If we end a scope, we first set the enclosing scope to be the current scope again and then we pop the top of the stack, so that the current scope automatically becomes the top of stack.

Where do we use these functions? Well, for a start when we go into a block.

```
procedure TResolver.VisitBlock(Block: TBlock);
var
  Node: TNode;
begin
  BeginScope;
  for Node in Block.Nodes do
    Visit(Node);
  EndScope;
end;
```

Everything that happens in the nodes of a block happens in a new scope!

We have to initialize and finalize variable Scopes:

```
constructor TResolver.Create;
begin
  ...
  Scopes := TScopes.Create;
end;

destructor TResolver.Destroy;
begin
  ...
  Scopes.Free;
  inherited Destroy;
end;
```

This is all we need for now in terms of scopes.

Fast variable lookup

> I spent 20 years doing research on regular and irregular verbs, not because I'm an obsessive language lover but because it seemed to me that they tapped into a fundamental distinction in language processing, indeed in cognitive processing, between memory lookup and rule-driven computation.
>
> — Steven Pinker

I promised to create a fast lookup functionality for variables during interpretation. We saw in the previous paragraph that when local variables are declared they are stored in scopes. When a local variable gets resolved, we read from the scopes again. Scopes are stored on a scope stack. As we resolve a local variable, we can already determine the distance from the current scope to the scope where the local variable was declared. This information can be used in the interpreter later on to immediately hop to the right memory space, thus saving expensive search time. To start of with, create a new private procedure in the resolver class:

```
procedure TResolver.ResolveLocal(Variable: TVariable);
var
  i: LongInt;
begin
  for i := Scopes.Count-1 downto 0 do
    if Scopes[i].Contains(Variable.Ident.Text) then begin
      Variable.Distance := Scopes.Count-1-i;
      Exit;
    end;
end;
```

Starting from the top scope, we start searching backward in outer scopes, and when/if found we set the variable's distance to the found scope. If the variable was not found here it must be inside the global memory space.

Given the above, now change method VisitVariable to include the new function at the end:

```
procedure TResolver.VisitVariable(Variable: TVariable);
...
begin
...
  if Symbol <> Nil then begin
    ...
    ResolveLocal(Variable);
  end;
end;
```

With this, the variable and the information in which memory space to look it up is stored and will be used in the interpreter.

We will now do the same for assignments to local variables. Change method VisitAssignStmt also to include resolving the local variable.

```
procedure TResolver.VisitAssignStmt(AssignStmt: TAssignStmt);
...
begin
  ...
  if Symbol <> Nil then begin
    ...
    ResolveLocal(AssignStmt.Variable);
  end;
end;
```

So far so good, and only minimal changes to the resolver class. Let's move to the AST, in order to make it possible to store distance information. We need to make a small change to class TVariable:

```
  TVariable = class(TFactorExpr)
    private
      FIdent: TIdent;
      FDistance: Integer;
    public
      property Ident: TIdent read FIdent;
      property Distance: Integer read FDistance write FDistance;
      constructor Create(AIdent: TIdent);
      destructor Destroy; override;
  end;
```

and give the default value -1 (meaning it's not a local variable) to FDistance in the constructor:

```
constructor TVariable.Create(AIdent: TIdent);
begin
  Inherited Create(AIdent.Token);
  FIdent := AIdent;
  FDistance := -1;
end;
```

Now that's all stored in case of local variables, we need a way to look them up when needed, so that we don't have to search through all memory spaces in the interpreter.

First, we start by introducing a global memory space in the interpreter. Define the following field as a public property:

```
  TInterpreter = class(TVisitor)
    private
      CurrentSpace: TMemorySpace;
      FGlobals: TMemorySpace;
    public
      property Globals: TMemorySpace read FGlobals;
    ...
```

Then, change the constructor as follows:

```
constructor TInterpreter.Create;
begin
  FGlobals := TMemorySpace.Create();
  FGlobals.Sorted := True;
  CurrentSpace:= FGlobals;
end;
```

This seems redundant but it's much more readable, and later on we'll add a lot more stuff to the globals, such as standard functions and more.

We already defined a function Lookup in the interpreter. Now we change it a bit, so it can cope with local scopes as well as the global scope.

For this we will use the Distance variable and a couple of specific new methods in the memory model. But first Lookup:

```
function TInterpreter.Lookup(Variable: TVariable): Variant;
begin
  if Variable.Distance >= 0 then
    Result := CurrentSpace.LoadAt(Variable.Distance, Variable.Ident)
  else
    Result := Globals.Load(Variable.Ident);
end;
```

If it is a local variable it will search using the distance, otherwise we search in the global scope. In TMemory we create a new function LoadAt():

```
function TMemorySpace.LoadAt(Distance: Integer; Name: String): Variant;
var
  MemorySpace: TMemorySpace;
  Index: LongInt = -1;
begin
  MemorySpace := MemorySpaceAt(Distance);
  if MemorySpace.Find(Name, Index) then
    Result := MemorySpace.At(Index)
  else
    Result := Unassigned;
end;

function TMemorySpace.LoadAt(Distance: Integer; Ident: TIdent): Variant;
begin
  Result := LoadAt(Distance, Ident.Text);
end;
```

They look up the name of the identifier in the memory space at the given distance. The memory space at that specific distance is found through the private function MemorySpaceAt():

```
function TMemorySpace.MemorySpaceAt(Distance: Integer): TMemorySpace;
var
  MemorySpace: TMemorySpace;
  i: Integer;
begin
  MemorySpace := Self;
  for i := 1 to Distance do
    MemorySpace := MemorySpace.EnclosingSpace;
  Result := MemorySpace;
end;
```

Since we know the distance, we can quickly find the required memory space by hopping through the enclosing spaces.

The same should apply for assigning new values to variables. Procedure Assign in the interpreter becomes:

```
procedure TInterpreter.Assign(Variable: TVariable; Value: Variant);
begin
  if Variable.Distance >= 0 then
    CurrentSpace.UpdateAt(Variable.Distance, Variable.Ident, Value)
  else
    Globals.Update(Variable.Ident, Value);
end;
```

Which calls UpdateAt():

```
procedure TMemorySpace.UpdateAt(Distance: Integer; Ident: TIdent; Value: Variant);
begin
  MemorySpaceAt(Distance)[Ident.Text] := Value
end;
```

Ready! We finally can move on to using blocks and local scopes now this is in place. In the next chapter we'll work on control statements and expressions.

Chapter 4 – Getting in control

In this chapter we focus on control statements such as if-then, for-do, while-do, repeat-until and switch. This gives a lot of possibilities to our language. Also we look into control expressions such as the if-expression and the match-expression.

If Then For While Do Repeat Until

> While history never repeats itself, political patterns do.
> — Eric Alterman

It may become clear from the title we're going to explore a couple more statements, namely:

- `if ... then ... else ... end`
- `while ... do ... end`
- `repeat ... until`
- `for ... do ... end`
- `switch ... case ... end`

Every programming language always has some form of the above statements in place. We will present a form, based on clarity and readibilty. For each of the statements, we follow the same procedure:

1. Define EBNF
2. Create AST node
3. Create parser function
4. Adjust ParseStmt and add token to constant StmtStartSet
5. Create visitors for the Printer, Resolver and the Interpreter
6. Test the code

One common part that all above statements have, is the list of statements and declarations in their statement body, which in fact is a Block. Add the following line just below the definition of TStmt in the uAST unit.

```
  TBlock = class;
```

This is called a forward declaration, which is needed since we refer to it in below code.

If-then-else statement

Define the EBNF

The EBNF of a if-then-else statement is:

```
IfStmt = 'if' Condition 'then' Block [ 'else' Block ] 'end' .
Condition = Expr .
```

However, this is more like the traditional if-then statement. I want to enhance it a bit and create variable binding, like this:

```
if let x := add(a,b) where x > 10 then print(x) end
```

The EBNF for that is:

```
  IfStmt = ‘if’ [VarDecl ‘where’] Condition ‘then’ Block [ ‘else’ Block ] ‘end’ .
```

The identifier can be declared as constant (let) or as variable (var), and its scope will be the if-statement as a whole.

Create AST node

```
  TIfStmt = class(TStmt)
    private
      FVarDecl: TVarDecl;
      FCondition: TExpr;
      FThenPart: TBlock;
      FElsePart: TBlock;
    public
      property VarDecl: TVarDecl read FVarDecl;
      property Condition: TExpr read FCondition;
      property ThenPart: TBlock read FThenPart;
      property ElsePart: TBlock read FElsePart;
      constructor Create(AVarDecl: TVarDecl; ACondition: TExpr;
        AThenPart, AElsePart: TBlock; AToken: TToken);
      destructor Destroy; override;
  end;
```

The if-then statement contains a VarDecl, a Condition, a then-part, and and else-part. The else-part is optional, which is handled by the parser. The if-stmt is always closed by the keyword ‘end’.

```
if [VarDecl where] condition then
  block with declarations / statements
  optional else leg
end
```

Put this code right below the TAssignStmt declaration.

TVarDecl is only declared under the Decl section, so we need to create a forward declaration. Add right below the forward declaration of TBlock:

```
  TVarDecl = class;
```

The implementation is as follows:

```
constructor TIfStmt.Create(AVarDecl: TVarDecl; ACondition: TExpr;
  AThenPart, AElsePart: TBlock; AToken: TToken);
begin
  inherited Create(AToken);
  FVarDecl := AVarDecl;
  FCondition := ACondition;
  FThenPart := AThenPart;
  FElsePart := AElsePart;
end;
```

```
destructor TIfStmt.Destroy;
begin
  if Assigned(FCondition) then FCondition.Free;
  if Assigned(FThenPart) then FThenPart.Free;
  if Assigned(FElsePart) then FElsePart.Free;
  if Assigned(FVarDecl) then FVarDecl.Free;
  inherited Destroy;
end;
```

Create parser function

```
function TParser.ParseIfStmt: TStmt;
var
  Token: TToken;
  VarDecl: TVarDecl = Nil;
  Condition: TExpr;
  ThenPart: TBlock;
  ElsePart: TBlock = Nil;
begin
  Token := CurrentToken;
  Next; // skip if
  if CurrentToken.Typ in [ttVar, ttLet] then begin
    case CurrentToken.Typ of
      ttVar: VarDecl := ParseVarDecl(True) as TVarDecl;
      ttLet: VarDecl := ParseVarDecl(False) as TVarDecl;
    end;
    Expect(ttWhere);
  end;
  Condition := ParseExpr;
  Expect(ttThen);
  ThenPart := ParseBlock;
  if CurrentToken.Typ = ttElse then begin
    Next; // skip else
    ElsePart := ParseBlock;
  end;
  Expect(ttEnd);
  Result := TIfStmt.Create(VarDecl, Condition, ThenPart, ElsePart, Token);
end;
```

We save the current token and skip the if keyword. Second, we check if it has a VarDecl, with either Let or Var. Then, we parse the condition. Note that this can be any expression, as in the parser we don't check if it is boolean. We have to perform that check at runtime. The block ThenPart is parsed. If there's an 'else' keyword, we parse the ElsePart.

If there's no 'else' the ElsePart is Nil. During printing we have to take care to check for this, otherwise we get a runtime error.

Adjust ParseStmt and add token to constant StmtStartSet

```
StmtStartSet: TTokenTypSet = [ttIf, ttPrint, ttIdentifier];
```

```
function TParser.ParseStmt: TStmt;
begin
  case CurrentToken.Typ of
    ttIf: Result := ParseIfStmt;
    ttPrint: Result := ParsePrintStmt;
    else
      Result := ParseAssignStmt;
  end;
end;
```

Create visitors for the Printer, the Resolver and the Interpreter

```
procedure TPrinter.VisitIfStmt(IfStmt: TIfStmt);
begin
  IncIndent;
  VisitNode(IfStmt);
  if Assigned(IfStmt.VarDecl) then
    Visit(IfStmt.VarDecl);
  Visit(IfStmt.Condition);
  IncIndent;
  WriteLn(Indent, 'ThenPart:');
  Visit(IfStmt.ThenPart);
  if Assigned(IfStmt.ElsePart) then begin
    WriteLn(Indent, 'ElsePart:');
    Visit(IfStmt.ElsePart);
  end;
  DecIndent;
  DecIndent;
end;
```

We print the VarDecl it is there, then the condition, the block of the ThenPart, and only if there's an else part, we print its contents too.

As mentioned, if the if-statement has a VarDecl, then the variable is available throughout the lifetime of the statement. That means we begin a scope, and declare the variable. At the end we end the scope again.

If the VarDecl is done with a 'let' than the identifier can't be changed anymore, but if you use 'var' it is mutable and can change value.

```
procedure TResolver.VisitIfStmt(IfStmt: TIfStmt);
begin
  if Assigned(IfStmt.VarDecl) then begin
    BeginScope;
    Visit(IfStmt.VarDecl);
  end;
  Visit(IfStmt.Condition);
  Visit(IfStmt.ThenPart);
  if IfStmt.ElsePart <> Nil then
    Visit(IfStmt.ElsePart);
  if Assigned(IfStmt.VarDecl) then
    EndScope;
end;
```

The visitor for the interpreter is a bit more complex.

Just like for the resolver, if there is a VarDecl, then the declared variable needs to reside in an overarching memory space, which is live during the existence of the if-statement. Inside the then-part and/or else-part, this variable must be available. We do this by saving the current space and creating a new memory space, which temporarily becomes the current space. The saved current space becomes the enclosing space.

Next, we visit the condition, which must return a boolean value, otherwise a runtime error is generated.

```
procedure TInterpreter.VisitIfStmt(IfStmt: TIfStmt);
var
  Condition: Variant;
  SavedSpace: TMemorySpace;
begin
  try
    if Assigned(IfStmt.VarDecl) then begin
      SavedSpace := CurrentSpace;
      CurrentSpace := TMemorySpace.Create(SavedSpace);
      Visit(IfStmt.VarDecl);
    end;
    Condition := Visit(IfStmt.Condition);
    if VarIsBool(Condition) then begin
      if Condition then
        Visit(IfStmt.ThenPart)
      else
        if Assigned(IfStmt.ElsePart) then
          Visit(IfStmt.ElsePart)
    end
    else
      Raise ERuntimeError.Create(IfStmt.Token, 'Condition is not Boolean.');
  finally
    if Assigned(IfStmt.VarDecl) then begin
      CurrentSpace.Free;
      CurrentSpace := SavedSpace;
    end;
  end;
end;
```

If the condition is boolean, we determine if it's true or false. If the condition is true, we evaluate the block belonging to the ThenPart. If condition is false, we check if there's an ElsePart, and if so, execute that block. If no ElsePart, there's nothing to do.

Test it

```
var a := 1
var b := 9
if a < 2 then
  var a := 5
  print(a*b) // 45
end
print(a*b) // 9
```

var a := 3

```
var b := 0
if a-2 > 1 then
  a := 10
  b := 8
else
  a := 11
  b := 12
end
print(a*b) // 132
```

```
if let a := 8*9 where a > 100 then
  print(a)
else
  print('Its not correct! ', 'a=', a)
end
//print(a) // error undeclared variable “a”.
```

While-do statement

Define the EBNF

The standard EBNF of a while-do statement is:

```
  WhileStmt = ‘while’ Condition ‘do’ Block ‘end’ .
```

However, I would also like to introduce the possibility to use a loop variable, which is declared in the while-statement and only is in scope during the execution of the while-statement. Hereto, we use our Variable Declaration from earlier, but now only as a ‘var’, and not as a ‘let’. The reason is obvious: the loopvar changes after every iteration.

```
  WhileStmt = ‘while’ [VarDecl ‘where’ ] Condition ‘do’ Block ‘end’ .
```

As an example consider the following:

```
while var x:=1 where x<=10 do
  print('x^2= ', x^2)
  x+=1
end
```

Variable ‘x’ is here declared as a local variable inside the while loop, and doesn’t exist outside the loop anymore. Also note the condition where variable x has to adhere to.

Create AST node

```
  TWhileStmt = class(TStmt)
    private
      FVarDecl: TVarDecl;
      FCondition: TExpr;
      FBlock: TBlock;
    public
      property VarDecl: TVarDecl read FVarDecl;
      property Condition: TExpr read FCondition;
      property Block: TBlock read FBlock;
      constructor Create(AVarDecl: TVarDecl; ACondition: TExpr;
        ABlock: TBlock; AToken: TToken);
      destructor Destroy; override;
  end;
```

```
constructor TWhileStmt.Create(AVarDecl: TVarDecl; ACondition: TExpr;
  ABlock: TBlock; AToken: TToken);
begin
  inherited Create(AToken);
  FVarDecl := AVarDecl;
  FCondition := ACondition;
  FBlock := ABlock;
end;

destructor TWhileStmt.Destroy;
begin
  if Assigned(FVarDecl) then FVarDecl.Free;
  if Assigned(FCondition) then FCondition.Free;
  if Assigned(FBlock) then FBlock.Free;
  inherited Destroy;
end;
```

Create parser function

```
function TParser.ParseWhileStmt: TStmt;
var
  Token: TToken;
  VarDecl: TVarDecl = Nil;
  Condition: TExpr;
  Block: TBlock;
begin
  Token := CurrentToken;
  Next; // skip while
  if CurrentToken.Typ = ttVar then begin
    VarDecl := ParseVarDecl(True) as TVarDecl;
    Expect(ttWhere);
  end;
  Condition := ParseExpr;
  Expect(ttDo);
  Block := ParseBlock;
  Expect(ttEnd);
  Result := TWhileStmt.Create(VarDecl, Condition, Block, Token);
end;
```

Initially, the VarDecl is Nil, and only if we find a variable declaration, we'll parse it including the where keyword. Then, we parse the condition, the do-keyword, followed by parsing the block and the final 'end'.

Adjust ParseStmt and add token to constant StmtStartSet

Add ttWhile to the constant StmtStartSet.

```
const
  StmtStartSet: TTokenTypSet = [ttIf, ttWhile, ttPrint, ttIdentifier];
```

Add this line to the case statement in ParseStmt.

```
ttWhile: Result := ParseWhileStmt;
```

Create visitors for the Printer, the Resolver and the Interpreter

```
procedure TPrinter.VisitWhileStmt(WhileStmt: TWhileStmt);
begin
  IncIndent;
  VisitNode(WhileStmt);
  if Assigned(WhileStmt.VarDecl) then
    Visit(WhileStmt.VarDecl);
  Visit(WhileStmt.Condition);
  IncIndent;
  WriteLn(Indent, 'Loop:');
  Visit(WhileStmt.Block);
  DecIndent;
  DecIndent;
end;
```

```
procedure TResolver.VisitWhileStmt(WhileStmt: TWhileStmt);
begin
  if Assigned(WhileStmt.VarDecl) then begin
    BeginScope;
    Visit(WhileStmt.VarDecl);
  end;
  Visit(WhileStmt.Condition);
  Visit(WhileStmt.Block);
  if Assigned(WhileStmt.VarDecl) then
    EndScope;
end;
```

```
procedure TInterpreter.VisitWhileStmt(WhileStmt: TWhileStmt);
var
  Condition: Variant;
  SavedSpace: TMemorySpace;
begin
  try
    if Assigned(WhileStmt.VarDecl) then begin
      SavedSpace := CurrentSpace;
      CurrentSpace := TMemorySpace.Create(SavedSpace);
      Visit(WhileStmt.VarDecl);
    end;
    Condition := Visit(WhileStmt.Condition);
    if VarIsBool(Condition) then begin
      while Condition do begin
        Visit(WhileStmt.Block);
        Condition := Visit(WhileStmt.Condition);
      end;
    end
    else
      Raise ERuntimeError.Create(WhileStmt.Token, ErrConditionMustBeBool);
  finally
    if Assigned(WhileStmt.VarDecl) then begin
      CurrentSpace.Free;
      CurrentSpace := SavedSpace;
    end;
  end;
end;
```

Everything is fitted in a try-finally block. If there is a VarDecl we save the current space temporarily and create a new current space, like a scope. The VarDecl is visited and then starts the testing of the condition. Of course it should be a boolean expression, and unit Variants provides us the means through function VarIsBool(Condition). As long as the condition is true, we keep on visiting the node's block. Finally, after the test to condition is false, we skip to the end and in case there was a VarDecl, we first free up the current space, which we don't need anymore, and restore it back to the saved space.

Test it

```
var a := 5
var b := 1
while a>b do
  a-=1
  let c := 3.1415926
  print(a*b*c)
end

while var x := 1 where x <= 10 do
  print('x^2 = ', x^2)
  x+=1
end
print(x) // error x doesn't exist here
```

Repeat-until statement

Define the EBNF

This statement is more or less copied from the Pascal language. It is a great statement for its simplicity and starightforwardness.

The EBNF of a repeat-until statement is:

```
  RepeatStmt = ‘repeat’ Block ‘until’ Condition .
```

Create AST node

```
  TRepeatStmt = class(TStmt)
    private
      FCondition: TExpr;
      FBlock: TBlock;
    public
      property Condition: TExpr read FCondition;
      property Block: TBlock read FBlock;
      constructor Create(ACondition: TExpr; ABlock: TBlock; AToken: TToken);
      destructor Destroy; override;
  end;
```

```
constructor TRepeatStmt.Create(ACondition: TExpr; ABlock: TBlock; AToken: TToken);
begin
  inherited Create(AToken);
  FCondition := ACondition;
  FBlock := ABlock;
end;

destructor TRepeatStmt.Destroy;
begin
  if Assigned(FCondition) then FCondition.Free;
  if Assigned(FBlock) then FBlock.Free;
  inherited Destroy;
end;
```

Create parser function

```
function TParser.ParseRepeatStmt: TStmt;
var
  Token: TToken;
  Condition: TExpr;
  Block: TBlock;
begin
  Token := CurrentToken;
  Next; // skip repeat
  Block := ParseBlock;
  Expect(ttUntil);
  Condition := ParseExpr;
  Result := TRepeatStmt.Create(Condition, Block, Token);
end;
```

Adjust ParseStmt and add token to constant StmtStartSet

Add ttRepeat to the constant StmtStartSet.

```
  StmtStartSet: TTokenTypSet = [ttIf, ttWhile, ttRepeat, ttPrint, ttIdentifier];
```

Add this line to the case statement in ParseStmt.

```
 ttRepeat: Result := ParseRepeatStmt;
```

Create visitors for the Printer, the Resolver and the Interpreter

```
procedure TPrinter.VisitRepeatStmt(RepeatStmt: TRepeatStmt);
begin
  IncIndent;
  VisitNode(RepeatStmt);
  Visit(RepeatStmt.Condition);
  IncIndent;
  WriteLn(Indent, 'Loop:');
  Visit(RepeatStmt.Block);
  DecIndent;
  DecIndent;
end;
```

```
procedure TResolver.VisitRepeatStmt(RepeatStmt: TRepeatStmt);
begin
  Visit(RepeatStmt.Condition);
  Visit(RepeatStmt.Block);
end;
```

```
procedure TInterpreter.VisitRepeatStmt(RepeatStmt: TRepeatStmt);
var
  Condition: Variant;
begin
  Condition := Visit(RepeatStmt.Condition);
  if VarIsBool(Condition) then begin
    repeat
      Visit(RepeatStmt.Block);
      Condition := Visit(RepeatStmt.Condition);
    until Condition;
  end
  else
    Raise ERuntimeError.Create(RepeatStmt.Token, ErrConditionMustBeBool);
end;
```

The nice thing is that since it is also standard Pascal, we can 1 on 1 reuse the repeat statement here. Again first testing for Booleanness of the condition we then repeatedly visit the node's Block, until the condition is met. Don't make the mistake to put here 'until not Conditon', since the condition is already declared as the condition to stop, a sort of negative condition.

Test it

```
var a := 5
var b := 1
repeat
  a := a - 1
  let c := 3.1415926
  print(a*b*c)
until b>a


var n := 10
repeat
  print('n=', n)
  n-=1
until n<=0
```

For-do statement

Define the EBNF

Much has been written about the for-do statement, and there are many forms available. In fact it's actually another form of writing a while statement. In the Gear for-statements it is required to define a new loop variable that is only in scope during the for-loop. The declared variable adheres to a condition and is incremented or decremented in the iterator.

The EBNF of the for statement is thus:

```
  ForStmt = 'for' VarDecl 'where' Condition ',' Iterator 'do' Block 'end' .
```

Example:

```
for var i := 0 where i < 100, i+=1 do
  print(i)
end
```

This is actually the same as the following while loop:

```
while var i := 0 where i < 100 do
  print(i)
  i+=1
end
```

Just like in the while-statement it is again not allowed to use 'let' in the variable declaration, since we change the value at every iteration.

So, what better to do then to translate it into the existing AST node TWhileStmt?

Create AST node

So, there's no separate AST node for the for-do statement.

Create parser function

```
function TParser.ParseForStmt: TStmt;
var
  Token: TToken;
  VarDecl: TVarDecl;
  Condition: TExpr;
  Iterator: TStmt;
  Block: TBlock;
begin
  Token := CurrentToken;
  Next; // skip for
  VarDecl := ParseVarDecl(True) as TVarDecl;
  Expect(ttWhere);
  Condition := ParseExpr;
  Expect(ttComma);
  Iterator := ParseAssignStmt;
  Expect(ttDo);
  Block := ParseBlock;
  Block.Nodes.Add(Iterator);
  Expect(ttEnd);
  Result := TWhileStmt.Create(VarDecl, Condition, Block, Token);
end;
```

We first parse a variable declaration, followed by a Condition expression. The visitor for the while statemement in the interpreter checks whether this is a boolean expression.

Then, the iterator statement is parsed as an assignment. Finally, the block is parsed. Then, add the iterator statement as the last statement to the block. Now we have to turn this into a while-statement. We now have created a new while statement containing this structure:

```
while var i := 0 where i < 100 do
  print(i)
  i+=1
end
```

Adjust ParseStmt and add token to constant StmtStartSet

Add ttFor to the constant StmtStartSet.

```
  StmtStartSet: TTokenTypSet = [ttIf, ttWhile, ttRepeat, ttFor,
    ttPrint, ttIdentifier];
```

Add to the case statement in ParseStmt.

```
ttFor: Result := ParseForStmt;
```

Create visitors for the Printer and the Interpreter

Not necessary as we use the while-statement, which already has the respective visitors.

Test it

```
var i:= -1
for var i:=1 where i<=10, i+=1 do
  print(i)
end
print(i)
```

Result:

```
1^3 = 1
2^3 = 8
3^3 = 27
4^3 = 64
5^3 = 125
6^3 = 216
7^3 = 343
8^3 = 512
9^3 = 729
10^3 = 1000
-1
```

The 2nd print statement prints the original value of i, which proofs the variable i inside the loop is in a different scope, which is not valid anymore after the execution of the loop.

Ensure statement

> Good wishes alone will not ensure peace.
> — Alfred Nobel

Many languages have some form of assertion or guarding that certain conditions are met. If a condition is not met, usually an error is generated, or at least an early escape from a function is possible. Instead of 'assert' or 'guard', I prefer the keyword 'ensure'. We want to ensure that a certain condition is met. Take a look at the next example, which also gives a sneak preview of what lies ahead: functions and classes.

```
class Date
  var day := 1
  var month := 'Jan'
  var year := 1900
  init(day, month, year)
    self.day := day
    self.month := month
    self.year := year
  end
end

func getYear(birthDate)
  ensure var year := birthDate.year where year > 1900 else
    return 'The year ' + year + ' is a wrong year of birth!'
  end

  return 'You were born in ' + year
end

var birthDate := Date(29, 'Feb', 1978)
print(getYear(birthDate))
var yourDate := Date(29, 'Feb', 1899)
print(getYear(yourDate))
```

Within the 'ensure' statement you can declare a variable or a constant and add a where-clause. If the condition is met, basically nothing happens, and we continue the rest of the statements. If the condition fails, the statements in the 'else' block are executed. Though there is no prescription on what should be contained in the else-block, the most practical is to use a return-statement, so that early escape from the function is possible.

The variable that is declared inside the ensure statement is part of the surrounding scope; in the shown example, this is the function scope. The EBNF of the ensure statement is:

```
EnsureStmt = 'ensure' [VarDecl 'where'] Condition 'else' Block 'end' .
```

Note that the variable declaration and where-clause are optional. You can also just ensure a conditon, like this:

```
ensure (birthDate.day > 0) & (birthDate.day <= 31) else
  return 'Wrong day number!'
end
```

Ensure ;-) that the token type ttEnsure is added to the statement start set:

```
StmtStartSet: TTokenTypSet = [ttIf, ttWhile, ttRepeat, ttFor,
  ttEnsure, ttPrint, ttIdentifier];
```

Looking ahead to the parser, add the keyword to the function ParseStmt, like this:

```
function TParser.ParseStmt: TStmt;
begin
  case CurrentToken.Typ of
    ...
    ttEnsure: Result := ParseEnsureStmt;
    ttPrint: Result := ParsePrintStmt;
    else
      Result := ParseAssignStmt;
  end;
end;
```

We'll create the respective parser function in a moment. First, create a new AST node:

```
 TEnsureStmt = class(TStmt)
    private
      FVarDecl: TVarDecl;
      FCondition: TExpr;
      FElsePart: TBlock;
    public
      property VarDecl: TVarDecl read FVarDecl;
      property Condition: TExpr read FCondition;
      property ElsePart: TBlock read FElsePart;
      constructor Create(AVarDecl: TVarDecl; ACondition: TExpr;
        AElsePart: TBlock; AToken: TToken);
      destructor Destroy; override;
  end;
```

with its implementation:

```
constructor TEnsureStmt.Create(AVarDecl: TVarDecl; ACondition: TExpr;
  AElsePart: TBlock; AToken: TToken);
begin
  inherited Create(AToken);
  FVarDecl := AVarDecl;
  FCondition := ACondition;
  FElsePart := AElsePart;
end;

destructor TEnsureStmt.Destroy;
begin
  if Assigned(FVarDecl) then FVarDecl.Free;
  if Assigned(FCondition) then FCondition.Free;
  if Assigned(FElsePart) then FElsePart.Free;
  inherited Destroy;
end;
```

The visitor for the printer:

```
procedure TPrinter.VisitEnsureStmt(EnsureStmt: TEnsureStmt);
begin
  IncIndent;
  VisitNode(EnsureStmt);
  if Assigned(EnsureStmt.VarDecl) then
    Visit(EnsureStmt.VarDecl);
  Visit(EnsureStmt.Condition);
  IncIndent;
  WriteLn(Indent, 'ElsePart:');
  Visit(EnsureStmt.ElsePart);
  DecIndent;
  DecIndent;
end;
```

And for the resolver:

```
procedure TResolver.VisitEnsureStmt(EnsureStmt: TEnsureStmt);
begin
  if Assigned(EnsureStmt.VarDecl) then
    Visit(EnsureStmt.VarDecl);
  Visit(EnsureStmt.Condition);
  Visit(EnsureStmt.ElsePart);
end;
```

Now, let's look at the parser. It is in fact just a little different from parsing the if-statement. In this case we only have an else-block, and, this is important, in case a variable is declared, we don't put it in a different scope.

```
function TParser.ParseEnsureStmt: TStmt;
var
  Token: TToken;
  VarDecl: TVarDecl = Nil;
  Condition: TExpr;
  ElsePart: TBlock;
begin
  Token := CurrentToken;
  Next; // skip ensure
  if CurrentToken.Typ in [ttVar, ttLet] then begin
    case CurrentToken.Typ of
      ttVar: VarDecl := ParseVarDecl(True) as TVarDecl;
      ttLet: VarDecl := ParseVarDecl(False) as TVarDecl;
    end;
    Expect(ttWhere);
  end;
  Condition := ParseExpr;
  Expect(ttElse);
  ElsePart := ParseBlock;
  Expect(ttEnd);
  Result := TEnsureStmt.Create(VarDecl, Condition, ElsePart, Token);
end;
```

Last but not least, the interpreter visitor. If there was a variable declaration we visit it, which means that we store the name and value in the current scope. Then, we interpret the condition, which must be boolean, and execute the else-block if the condition is false. That's it.

```
procedure TInterpreter.VisitEnsureStmt(EnsureStmt: TEnsureStmt);
var
  Condition: Variant;
begin
  if Assigned(EnsureStmt.VarDecl) then
    Visit(EnsureStmt.VarDecl);
  Condition := Visit(EnsureStmt.Condition);
  if VarIsBool(Condition) then begin
    if not Condition then
      Visit(EnsureStmt.ElsePart)
  end
  else
    Raise ERuntimeError.Create(EnsureStmt.Token, ErrConditionMustBeBool);
end;
```

Pattern matching

Besides the expression types described earlier, we introduce a few complex expression types. These are if-expressions and match-expressions.

> The human brain is an incredible pattern-matching machine.
> — Jeff Bezos

If-expression

If-expressions are supported one way or the other in many languages. For example in c-like languages they appear as conditional expressions with a ternary operator.

```
C-like language: condition ? evaluated-when-true : evaluated-when-false
```

Example: `Y = X > 0 ? A : B`

Which means that if X is greater than zero than Y becomes A otherwise Y becomes B.

In order to create better readibility I personally prefer the following:

```
Y := if X > 0 then A else B
```

It's a matter of taste I guess. I will continue developing the If expression.

An if-expression is to be treated as a normal expression, which means you can assign it to variables. The EBNF of an if-expression is:

```
IfExpr = 'if' Condition 'then' TrueExpr 'else' FalseExpr .
TrueExpr = Expr .
FalseExpr = Expr .
```

The following will be possible to do:

```
let a := 7
let b := 13
print('max = ', if a>b then a else b)
```

If-expressions should be used wisely, and preferably not concatenated in endless if-expressions. For the latter case it's better to use the match-expression discussed hereafter.

The AST node for an if-expression is as follows:

```
  TIfExpr = class(TFactorExpr)
    private
      FCondition,
      FTrueExpr,
      FFalseExpr: TExpr;
    public
      property Condition: TExpr read FCondition;
      property TrueExpr: TExpr read FTrueExpr;
      property FalseExpr: TExpr read FFalseExpr;
      constructor Create(ACondition, ATrueExpr, AFalseExpr: TExpr; AToken: TToken);
      destructor Destroy; override;
  end;
```

And it's implementation:

```
constructor TIfExpr.Create
  (ACondition, ATrueExpr, AFalseExpr: TExpr; AToken: TToken);
begin
  inherited Create(AToken);
  FCondition := ACondition;
  FTrueExpr := ATrueExpr;
  FFalseExpr := AFalseExpr;
end;

destructor TIfExpr.Destroy;
begin
  if Assigned(FCondition) then FCondition.Free;
  if Assigned(FTrueExpr) then FTrueExpr.Free;
  if Assigned(FFalseExpr) then FFalseExpr.Free;
  inherited Destroy;
end;
```

Notice that TIfExpr descends from TFactorExpr. It will be recognized in the ParseFactor method of the expression parser.

The parser routine for the if-expression (also add it to the class interface!):

```
function TParser.ParseIfExpr: TExpr;
var
  Condition, TrueExpr, FalseExpr: TExpr;
  Token: TToken;
begin
  Token := CurrentToken;
  Next; // skip 'if'
  Condition := ParseExpr;
  Expect(ttThen);
  TrueExpr := ParseExpr;
  Expect(ttElse);
  FalseExpr := ParseExpr;
  Result := TIfExpr.Create(Condition, TrueExpr, FalseExpr, Token);
end;
```

Very straightforward, after the 'if' we parse the expression (it must be a condition, which is checked in the interpreter); we expect a 'then' followed by a 'truth' expression, followed by an 'else' and finally a false expression.

Add to function ParseFactor, just below the case for ttIdentifier, the case for ttIf:

```
    ...
  case CurrentToken.Typ of
    ttIf: Result := ParseIfExpr;
    ttIdentifier: Result := TVarExpr.Create(ParseIdent);
    ...
```

Of course, we have to create the AST independent accompanying methods for TPrinter, TResolver and TInterpreter.

Here's the TPrinter visitor, which nicely prints the condition and the True and False parts.

```
procedure TPrinter.VisitIfExpr(IfExpr: TIfExpr);
begin
  IncIndent;
  VisitNode(IfExpr);
  Visit(IfExpr.Condition);
  IncIndent;
  Writeln(Indent, 'True:');
  Visit(IfExpr.TrueExpr);
  Writeln(Indent, 'False:');
  Visit(IfExpr.FalseExpr);
  DecIndent;
  DecIndent;
end;
```

```
procedure TResolver.VisitIfExpr(IfExpr: TIfExpr);
begin
  Visit(IfExpr.Condition);
  Visit(IfExpr.TrueExpr);
  Visit(IfExpr.FalseExpr);
end;
```

```
function TInterpreter.VisitIfExpr(IfExpr: TIfExpr): Variant;
var
  Condition: Variant;
begin
  Condition := Visit(IfExpr.Condition);
  if VarIsBool(Condition) then begin
    if Condition then
      Result := Visit(IfExpr.TrueExpr)
    else
      Result := Visit(IfExpr.FalseExpr)
  end
  else
    Raise ERuntimeError.Create(IfExpr.Token, ErrConditionMustBeBool);
end;
```

We first evaluate the condition, and check if it's boolean. If it's a boolean, we check if it's true or false and execute that branch.

There you go, it works! Try it out. There's one side effect, which I'm not sure I will keep or not. It's the case that the TrueExpr has a different type than the FalseExpr, like this:

```
let b := 13
var d := if b <> 13 then 10 else 'too bad'
print(d)
```

If b is unequal to 13 the type of 'd' is Number, otherwise it's String. If we don't want this 'side-effect', we'll have to check that both types must be equal.

Match-expression

In this book (sort of) I start with the match expression instead of the case or switch statement. A match or case expression is rarely seen in programming languages, and that intrigues me. Because I have the idea it can be handy in many ways. E.g. consider the following function, which calculates the Fibonacci range: 1 1 2 3 5 8 13 21 34 55 etc. for a given number n.

```
func F(n)
  if n = 0 then
    return 0
  else
    if n = 1 then
      return 1
    else
      return F(n-1) + F(n-2)
    end
  end
end
```

We haven't discussed functions or methods yet, but this is already a sneak preview ☺.

But how nice would it be to do something like this:

```
func F(n) =>
  match n
    if 0 then 0
    if 1 then 1
    else F(n-1) + F(n-2)
```

In EBNF, the MatchExpr is defined like this:

```
MatchExpr = 'match' Expr 'if' Expr 'then' Expr {'if' Expr 'then' Expr} 'else' Expr .
```

The final 'else' is mandatory!

The AST node for a match-expression is:

```
  TMatchExpr = class(TFactorExpr)
    private
      type
        TIfLimb = class
          Value: TExpr;
          Expr: TExpr;
          constructor Create(AValue, AExpr: TExpr);
          destructor Destroy; override;
        end;
        TIfLimbs = specialize TArrayObj<TIfLimb>;
    private
      FExpr: TExpr;
      FIfLimbs: TIfLimbs;
      FElseLimb: TExpr;
    public
      property Expr: TExpr read FExpr;
      property IfLimbs: TIfLimbs read FIfLimbs;
      property ElseLimb: TExpr read FElseLimb write FElseLimb;
      constructor Create(aExpr: TExpr; AToken: TToken);
      destructor Destroy; override;
      procedure AddLimb(AValue, AExpr: TExpr);
  end;
```

Inside the class, we define a private type TIfLimb with fields Value and Expr. From the above example you see e.g. 'if 0 then 0' which translates to 'if Value then Expr'. Next to this a list type is defined, TIfLimbs, which holds all the defined if-limbs.

```
constructor TMatchExpr.TIfLimb.Create(AValue, AExpr: TExpr);
begin
  Value := AValue;
  Expr := AExpr;
end;

destructor TMatchExpr.TIfLimb.Destroy;
begin
  if Assigned(Value) then Value.Free;
  if Assigned(Expr) then Expr.Free;
  inherited Destroy;
end;

constructor TMatchExpr.Create(aExpr: TExpr; AToken: TToken);
begin
  Inherited Create(AToken);
  FExpr := AExpr;
  FElseLimb := Nil;
  FIfLimbs := TIfLimbs.Create();
end;

destructor TMatchExpr.Destroy;
begin
  if Assigned(FExpr) then FExpr.Free;
  if Assigned(FIfLimbs) then FIfLimbs.Free;
  if Assigned(FElseLimb) then FElseLimb.Free;
  inherited Destroy;
end;

procedure TMatchExpr.AddLimb(AValue, AExpr: TExpr);
begin
  FIfLimbs.Add(TIfLimb.Create(AValue, AExpr));
end;
```

Procedure AddLimb adds a combination of value and expression to the list of limbs.

In the parser add to function ParseFactor the case for ttMatch, like this:

```
function TParser.ParseFactor: TExpr;
begin
  case CurrentToken.Typ of
    ttIf: Result := ParseIfExpr;
    ttMatch: Result := ParseMatchExpr;
    ttIdentifier: Result := TVariable.Create(ParseIdent);
    ...
    else begin
      Result := TExpr.Create(CurrentToken);
      Error(CurrentToken, Format(ErrUnexpectedToken, [CurrentToken.toString]));
    end;
  end;
end;
```

Right above ParseFactor add the ParseMatchExpr function.

```
function TParser.ParseMatchExpr: TExpr;
var
  Token: TToken;
  Value, Expr: TExpr;
  MatchExpr: TMatchExpr;
begin
  Token := CurrentToken;
  Next; // skip 'match'
  Expr := ParseExpr;
  MatchExpr := TMatchExpr.Create(Expr, Token);
  Expect(ttIf); // one 'if' is mandatory
  Value := ParseExpr;
  Expect(ttThen);
  MatchExpr.AddLimb(Value, ParseExpr);
  while CurrentToken.Typ = ttIf do begin
    Next;  // skip if
    Value := ParseExpr;
    Expect(ttThen);
    MatchExpr.AddLimb(Value, ParseExpr);
  end;
  Expect(ttElse);  // else is mandatory
  MatchExpr.ElseLimb := ParseExpr;
  Result := MatchExpr;
end;
```

After the keyword 'match' we parse the expression to match. Together with the Token we already create the MatchExpr.

We have to manually add the if-limbs and the else expression. Since one 'if' is mandatory as is the 'else', we start with parsing an 'if-then'-construct and add it to the limb-list. Then, we continue parsing if-then's until we find another keyword. This must be 'else'. We add the else expression to the ElseLimb of MatchExpr.

Finally, we return the result.

Next, we look at the visitors, starting with the printer.

```
procedure TPrinter.VisitMatchExpr(MatchExpr: TMatchExpr);
var
  i: integer;
begin
  IncIndent;
  VisitNode(MatchExpr);
  Visit(MatchExpr.Expr);
  IncIndent;
  Writeln(Indent, 'If Limbs:');
  for i := 0 to MatchExpr.IfLimbs.Count-1 do begin
    IncIndent;
    WriteLn(Indent, 'IF:');
    Visit(MatchExpr.IfLimbs[i].Value);
    Visit(MatchExpr.IfLimbs[i].Expr);
    DecIndent;
  end;
  Writeln(Indent, 'Else:');
  Visit(MatchExpr.ElseLimb);
  DecIndent;
  DecIndent;
end;
```

The resolver:

```
procedure TResolver.VisitMatchExpr(MatchExpr: TMatchExpr);
var
  i: Integer;
begin
  Visit(MatchExpr.Expr);
  for i := 0 to MatchExpr.IfLimbs.Count-1 do begin
    Visit(MatchExpr.IfLimbs[i].Value);
    Visit(MatchExpr.IfLimbs[i].Expr);
  end;
  Visit(MatchExpr.ElseLimb);
end;
```

Finally, the visitor for the interpreter.

```
function TInterpreter.VisitMatchExpr(Node: TMatchExpr): Variant;
var
  MatchValue, IfValue: Variant;
  i: integer;
begin
  MatchValue := Visit(Node.Expr);
  for i := 0 to Node.IfLimbs.Count-1 do begin
    IfValue := Visit(Node.IfLimbs[i].Value);
    if MatchValue = IfValue then
      Exit(Visit(Node.IfLimbs[i].Expr));
  end;
  Result := Visit(Node.ElseLimb);
end;
```

First, we evaluate the expression to match. Then, we go through the list of if-limbs and evaluate the values, also getting their values. We compare the value to match with the value of the 'if', and if equal, we return (via the Exit() procedure) the resulting expression. If none of the values match, we return the expression in the else-limb.

That's it, try it out, for example I tested it with this input code:

```
var heads := 1
var tail := 2
var coin := heads
print(
  match coin
    if heads then 'Heads'
    if tail then 'Tail'
    else 'Nothing'
)
```

Result printed: Heads

The matching is done in a simple and not efficient way currently. We will extend the match expression later on to also include tests on conditions, and multiple values, like this:

```
var throw := 5
var diceRoll := match
  if 1,2,3 then 'Small throw'
  if >3 then 'Big number'
  else 'Something went wrong'
```

Enhanced pattern matching

In previous paragraph I introduced pattern matching via the match expression. Whilst being a powerful functionality, it is yet far from perfect. As an example consider the following code:

```
func Factorial(n) =>
  match n
    if 0 then 1
    if 1 then 1
    else n*Factorial(n-1)
```

The matching values can now only be concrete values, like a number or a string. But in this case you would like to be able to do this as well:

```
func Factorial(n) =>
  match n
    if 0,1 then 1
    else n*Factorial(n-1)
```

We need a few small changes to the AST node: Replace Value with Values and make it of type TExprList.

```
  TMatchExpr = class(TFactorExpr)
    private
      type
        TIfLimb = class
          Values: TExprList;
          ...
          constructor Create(AValues: TExprList; AExpr: TExpr);
          ...
        end;
          ...
    private
          ...
    public
          ...
      procedure AddLimb(AValues: TExprList; AExpr: TExpr);
  end;
```

Do this in the implementation as well:

```
constructor TMatchExpr.TIfLimb.Create(AValues: TExprList; AExpr: TExpr);
begin
  Values := AValues;
  ...
end;

destructor TMatchExpr.TIfLimb.Destroy;
begin
  if Assigned(Values) then Values.Free;
  ...
end;

procedure TMatchExpr.AddLimb(AValues: TExprList; AExpr: TExpr);
begin
  FIfLimbs.Add(TIfLimb.Create(AValues, AExpr));
end;
```

Let's enhance the parser function:

```
function TParser.ParseMatchExpr: TExpr;
var
  ...
  Values: TExprList;  // 'Value' becomes 'Values'
begin
  ...
  Values := ParseExprList;
  Expect(ttThen);
  MatchExpr.AddLimb(Values, ParseExpr);
  while CurrentToken.Typ = ttIf do begin
    Next;  // skip if
    Values := ParseExprList;
    Expect(ttThen);
    MatchExpr.AddLimb(Values, ParseExpr);
  end;
  ...
end;
```

Function ParseExprList is a private helper function that parses a list of expressions separated by comma's:

```
function TParser.ParseExprList: TExprList;
begin
  Result := TExprList.Create();
  Result.Add(ParseExpr);
  while CurrentToken.Typ = ttComma do begin
    Next;  // skip ,
    Result.Add(ParseExpr);
  end;
end;
```

Changes to the printer visitor and resolver visitor:

```
procedure TPrinter.VisitMatchExpr(MatchExpr: TMatchExpr);
var
  i, j: integer;
begin
  ...
  for i := 0 to MatchExpr.IfLimbs.Count-1 do begin
  ...
    for j := 0 to MatchExpr.IfLimbs[i].Values.Count-1 do
      Visit(MatchExpr.IfLimbs[i].Values[j]);
    Visit(MatchExpr.IfLimbs[i].Expr);
    DecIndent;
  end;
  ...
end;
```

```
procedure TResolver.VisitMatchExpr(MatchExpr: TMatchExpr);
var
  i, j: Integer;
begin
  ...
  for i := 0 to MatchExpr.IfLimbs.Count-1 do begin
    for j := 0 to MatchExpr.IfLimbs[i].Values.Count-1 do
      Visit(MatchExpr.IfLimbs[i].Values[j]);
    Visit(MatchExpr.IfLimbs[i].Expr);
  end;
  ...
end;
```

And here are the changes to the interpreter:

The same TMath._EQ check can be applied on the match expression:

```
function TInterpreter.VisitMatchExpr(MatchExpr: TMatchExpr): Variant;
var
  MatchValue, IfValue: Variant;
  i, j: integer;
begin
  MatchValue := Visit(MatchExpr.Expr);
  for i := 0 to MatchExpr.IfLimbs.Count-1 do begin
    for j := 0 to MatchExpr.IfLimbs[i].Values.Count-1 do begin
      IfValue := Visit(MatchExpr.IfLimbs[i].Values[j]);
      if TMath._EQ(MatchValue, IfValue, MatchExpr.IfLimbs[i].Values[j].Token) then
        Exit(Visit(MatchExpr.IfLimbs[i].Expr));
    end;
  end;
  Result := Visit(MatchExpr.ElseLimb);
end;
```

In the second loop (j) we compare each value of the expression list with the value to match. If the value was found we exit the loop and the function with Exit.

```
var someCharacter := 'u'

print(someCharacter, match someCharacter
  if 'a', 'e', 'i', 'o', 'u' then ' is a vowel'
  if 'b', 'c', 'd', 'f', 'g', 'h', 'j',
     'k', 'l', 'm', 'n', 'p', 'q', 'r',
     's', 't', 'v', 'w', 'x', 'y', 'z' then ' is a consonant'
  else ' is not a vowel nor a consonant') // Prints "u is a vowel"

someCharacter := 'x'
print(someCharacter, match someCharacter
  if 'a', 'e', 'i', 'o', 'u' then ' is a vowel'
  if 'b', 'c', 'd', 'f', 'g', 'h', 'j',
     'k', 'l', 'm', 'n', 'p', 'q', 'r',
     's', 't', 'v', 'w', 'x', 'y', 'z' then ' is a consonant'
  else ' is not a vowel nor a consonant') // Prints "x is a consonant"
```

Switch statement

What the match expression is to expressions (which can be used for pattern matching), is the switch statement to statements (which can be used for quick branching in the code. Almost every programming language has a switch statement in some form. In some languages it's called a case statement. Somehow, I see it as a branching switch, so I use that term for it.

Let's first define the EBNF for SwitchStmt:

> The key in mastering any kind of sales is switching statements about you and how great you are and what you do, to statements about them, and how great they are and how they will produce more and profit more from ownership of your product or service.
>
> — Jeffrey Gitomer

```
SwitchStmt = 'switch' Expr
               'case' Expr ':' Block
               [{ 'case' Expr ':' Block }]
               'else' Block
             'end' .
```

As a very quick example, in its basic form, have a look at the following:

```
var a := 3
switch a
  case 1: print(1)
  case 2: print(2)
  case 3:
    print(3)
    print('a')
  case 4: print(4)
  case 5: print(5)
  case 6: print(6)
  else print(0)
end
```

What we read from the EBNF is that at least 1 'case' limb is mandatory and also the 'else' limb is mandatory. This is to ensure we always cover all options.

After the case keyword an expression is expected. This can be any expression, as in this version there's no checking whether the switch expression has the same type as the case expression. We can build that in later.

Add the token ttSwitch to the statement start set. Do NOT add the ttCase token!

```
StmtStartSet: TTokenTypSet = [ttIf, ttWhile, ttRepeat, ttFor,
  ttSwitch, ttEnsure, ttPrint, ttIdentifier];
```

Also, add already the case for ttSwitch to the ParseStmt method in the parser:

```
function TParser.ParseStmt: TStmt;
begin
  case CurrentToken.Typ of
    ...
    ttSwitch: Result := ParseSwitchStmt;
    ttEnsure: Result := ParseEnsureStmt;
    ttPrint: Result := ParsePrintStmt;
    else
      Result := ParseAssignStmt;
  end;
end;
```

I use the order in which the token types are defined in unit uToken.

Now, let's explore the AST node for TSwitchStmt. In fact with some adjustments it's almost a copy from TMatchExpr, as you'll notice when you compare. The only differences are the fact that it's a statement type, that we have case-limbs, and that the limbs contain a Block instead of an expression. Even the method AddLimb is similar.

```
  TSwitchStmt = class(TStmt)
    private
      type
        TCaseLimb = class
          Value: TExpr;
          Block: TBlock;
          constructor Create(AValue: TExpr; ABlock: TBlock);
          destructor Destroy; override;
        end;
        TCaseLimbs = specialize TArrayObj<TCaseLimb>;
    private
      FExpr: TExpr;
      FCaseLimbs: TCaseLimbs;
      FElseLimb: TBlock;
    public
      property Expr: TExpr read FExpr;
      property CaseLimbs: TCaseLimbs read FCaseLimbs;
      property ElseLimb: TBlock read FElseLimb write FElseLimb;
      constructor Create(aExpr: TExpr; AToken: TToken);
      destructor Destroy; override;
      procedure AddLimb(AValue: TExpr; ABlock: TBlock);
  end;
```

```
constructor TSwitchStmt.TCaseLimb.Create(AValue: TExpr; ABlock: TBlock);
begin
  Value := AValue;
  Block := ABlock;
end;

destructor TSwitchStmt.TCaseLimb.Destroy;
begin
  if Assigned(Value) then Value.Free;
  if Assigned(Block) then Block.Free;
  inherited Destroy;
end;

constructor TSwitchStmt.Create(aExpr: TExpr; AToken: TToken);
begin
  Inherited Create(AToken);
  FExpr := AExpr;
  FElseLimb := Nil;
  FCaseLimbs := TCaseLimbs.Create(True);
end;

destructor TSwitchStmt.Destroy;
begin
  if Assigned(FExpr) then FExpr.Free;
  if Assigned(FElseLimb) then FElseLimb.Free;
  FCaseLimbs.Free;
  inherited Destroy;
end;

procedure TSwitchStmt.AddLimb(AValue: TExpr; ABlock: TBlock);
begin
  FCaseLimbs.Add(TCaseLimb.Create(AValue, ABlock));
end;
```

Next, we'll look at parsing the switch statement. Again many similarities with the ParseMatchExpr method.

```
function TParser.ParseSwitchStmt: TStmt;
var
  Token: TToken;
  Value: TExpr;
  SwitchStmt: TSwitchStmt;
begin
  Token := CurrentToken;
  Next; // skip 'switch'
  SwitchStmt := TSwitchStmt.Create(ParseExpr, Token);
  Expect(ttCase); // one 'case' is mandatory
  Value := ParseExpr;
  Expect(ttColon);
  SwitchStmt.AddLimb(Value, ParseBlock);
  while CurrentToken.Typ = ttCase do begin
    Next;  // skip case
    Value := ParseExpr;
    Expect(ttColon);
    SwitchStmt.AddLimb(Value, ParseBlock);
  end;
  Expect(ttElse);  // else is mandatory
  SwitchStmt.ElseLimb := ParseBlock;
  Expect(ttEnd);
  Result := SwitchStmt;
end;
```

First create the SwitchStmt, followed by parsing at least one case limb. While we encounter more 'case' limbs we parse them, until the last one is parsed. The else-limb is mandatory. We close of with expecting the 'end' keyword, and return the result, a fully parsed switch statement.

Move on to the printer:

```
procedure TPrinter.VisitSwitchStmt(SwitchStmt: TSwitchStmt);
var
  i: integer;
begin
  IncIndent;
  VisitNode(SwitchStmt);
  Visit(SwitchStmt.Expr);
  IncIndent;
  Writeln(Indent, 'Case Limbs:');
  for i := 0 to SwitchStmt.CaseLimbs.Count-1 do begin
    IncIndent;
    WriteLn(Indent, 'CASE:');
    Visit(SwitchStmt.CaseLimbs[i].Value);
    Visit(SwitchStmt.CaseLimbs[i].Block);
    DecIndent;
  end;
  Writeln(Indent, 'Else:');
  Visit(SwitchStmt.ElseLimb);
  DecIndent;
  DecIndent;
end;
```

And on to the resolver:

```
procedure TResolver.VisitSwitchStmt(SwitchStmt: TSwitchStmt);
var
  i: Integer;
begin
  Visit(SwitchStmt.Expr);
  for i := 0 to SwitchStmt.CaseLimbs.Count-1 do begin
    Visit(SwitchStmt.CaseLimbs[i].Value);
    Visit(SwitchStmt.CaseLimbs[i].Block);
  end;
  Visit(SwitchStmt.ElseLimb);
end;
```

We visit the expression, all limbs' values and blocks and finally the else-part.
So far, very similar to the MatchExpr. This also goes for the interpreter visitor.

```
procedure TInterpreter.VisitSwitchStmt(SwitchStmt: TSwitchStmt);
var
  SwitchValue, CaseValue: Variant;
  i: integer;
begin
  SwitchValue := Visit(SwitchStmt.Expr);
  for i := 0 to SwitchStmt.CaseLimbs.Count-1 do begin
    CaseValue := Visit(SwitchStmt.CaseLimbs[i].Value);
    if SwitchValue = CaseValue then begin
      Visit(SwitchStmt.CaseLimbs[i].Block);
      Exit;
    end;
  end;
  // if no match found => execute Else block
  Visit(SwitchStmt.ElseLimb);
end;
```

The expression and case values are compared using their variant values. For simple values, like strings, numbers, booleans and chars this is very easy. When the constant value is returned from the Lexer into the Token, this value is already calculated and thus readily available for the above comparison.

However, it is not possible to use class instances as the values of the case expressions. This will result in runtime errors.

For now there's one extra step to go for switch statements, and that's when a case line is to be tested on multiple values, like this:

```
var c := 'U'

switch c
  case 'a', 'A', 'e', 'E', 'i', 'I', 'u', 'U', 'o', 'O', 'y', 'Y':
    print('It is a vowel!')
  else
    print('It is a consonant')
end
```

// prints: It is a vowel!

First, make some small adjustments to the AST node:

```
   TSwitchStmt = class(TStmt)
     private
       type
         TCaseLimb = class
           Values: TExprList;
           ...
           constructor Create(AValues: TExprList; ABlock: TBlock);
           ...
         end;
           ...
     private
           ...
     public
       ...
       procedure AddLimb(AValues: TExprList; ABlock: TBlock);
   end;

constructor TSwitchStmt.TCaseLimb.Create(AValues: TExprList; ABlock: TBlock);
begin
  Values := AValues;
  ...
end;

destructor TSwitchStmt.TCaseLimb.Destroy;
begin
  if Assigned(Values) then Values.Free;
  ...
end;

procedure TSwitchStmt.AddLimb(AValues: TExprList; ABlock: TBlock);
begin
  FCaseLimbs.Add(TCaseLimb.Create(AValues, ABlock));
end;
```

Instead of a single value expression, a list of expressions is used. This is also the case for the AddLimb method. In the parser we have to change the parsing of one expression into the parsing of a list of expressions separated by comma's.

Previously, we introduced a private function 'ParseExprList' that returns a list of expressions. It continues parsing expressions while there are comma's.

Then, method ParseSwitchStmt changes from parsing just one value to parsing the list of values:

```
function TParser.ParseSwitchStmt: TStmt;
var
  ...
  Values: TExprList;  // replace 'Value' by 'Values'
begin
  ...
  Expect(ttCase); // one 'case' is mandatory
  Values := ParseExprList;
  Expect(ttColon);
  SwitchStmt.AddLimb(Values, ParseBlock);
  while CurrentToken.Typ = ttCase do begin
    Next;  // skip case
    Values := ParseExprList;
    Expect(ttColon);
    SwitchStmt.AddLimb(Values, ParseBlock);
  end;
  ...
end;
```

The structure is still the same as previously as shows. 'Value' has become 'Values'.

The visitor for the printer now has to visit each and every value in the expression list:

```
procedure TPrinter.VisitSwitchStmt(SwitchStmt: TSwitchStmt);
var
  i,j: integer;
begin
  IncIndent;
  VisitNode(SwitchStmt);
  Visit(SwitchStmt.Expr);
  IncIndent;
  Writeln(Indent, 'Case Limbs:');
  for i := 0 to SwitchStmt.CaseLimbs.Count-1 do begin
    IncIndent;
    WriteLn(Indent, 'CASE:');
    for j := 0 to SwitchStmt.CaseLimbs[i].Values.Count-1 do
      Visit(SwitchStmt.CaseLimbs[i].Values[j]);
    Visit(SwitchStmt.CaseLimbs[i].Block);
    DecIndent;
  end;
  Writeln(Indent, 'Else:');
  Visit(SwitchStmt.ElseLimb);
  DecIndent;
  DecIndent;
end;
```

The same goes for the resolver:

```
procedure TResolver.VisitSwitchStmt(SwitchStmt: TSwitchStmt);
var
  i, j: Integer;
begin
  Visit(SwitchStmt.Expr);
  for i := 0 to SwitchStmt.CaseLimbs.Count-1 do begin
    for j := 0 to SwitchStmt.CaseLimbs[i].Values.Count-1 do
      Visit(SwitchStmt.CaseLimbs[i].Values[j]);
    Visit(SwitchStmt.CaseLimbs[i].Block);
  end;
  Visit(SwitchStmt.ElseLimb);
end;
```

Finally, we change the interpreter. It now has to also visit each value in order to find a match. That means again we have to introduce a loop j inside the loop i.

```
procedure TInterpreter.VisitSwitchStmt(SwitchStmt: TSwitchStmt);
var
  SwitchValue, CaseValue: Variant;
  i, j: integer;
begin
  SwitchValue := Visit(SwitchStmt.Expr);
  for i := 0 to SwitchStmt.CaseLimbs.Count-1 do begin
    for j := 0 to SwitchStmt.CaseLimbs[i].Values.Count-1 do begin
      CaseValue := Visit(SwitchStmt.CaseLimbs[i].Values[j]);
      if TMath._EQ(SwitchValue,CaseValue, SwitchStmt.CaseLimbs[i].Values[j].Token) then
      begin
        Visit(SwitchStmt.CaseLimbs[i].Block);
        Exit;
      end;
    end;
  end;
  // if no match found => execute Else block
  Visit(SwitchStmt.ElseLimb);
end;
```

By using the TMath._EQ test we are sure that the value types also have to match, otherwise an error message is generated. The token is needed to find the correct error location.

And this is all for making it possible to use multiple values. Another example:

```
val someCharacter := 'u'
switch someCharacter
  case 'a', 'e', 'i', 'o', 'u':
    print(someCharacter, ' is a vowel')
  case 'b', 'c', 'd', 'f', 'g', 'h', 'j', 'k', 'l', 'm',
     'n', 'p', 'q', 'r', 's', 't', 'v', 'w', 'x', 'y', 'z':
    print(someCharacter, ' is a consonant')
  else
    print(someCharacter, ' is not a vowel nor a consonant')
end
// Prints "u is a vowel"
```

Break statement

When using a count-controlled loop to search through a table, it might be desirable to stop searching as soon as the required item is found. Some programming languages provide a statement such as break, which effect is to terminate the current loop immediately, and transfer control to the statement immediately after that loop.

> I am free because I know that I alone am morally responsible for everything I do. I am free, no matter what rules surround me. If I find them tolerable, I tolerate them; if I find them too obnoxious, I break them. I am free because I know that I alone am morally responsible for everything I do.
>
> — Robert A. Heinlein

Though there's always the possibility to use return from a statement, this in practice means you immediately return from the function. Also, since we support loop statements outside functions, a way to break early from a loop is welcome in some cases. Usually the break statement is part of an if-then statement, like in the following:

```
for var i := 0 where i<20, i+=1 do
  if i=10 then
    break
  end
  print(i)
end
```

However, a more convenient way to use break is making the condition part of the break statement, like this:

```
for var i := 0 where i<20, i+=1 do
  break on i=10
  print(i)
end
```

Both solutions are allowed in Gear. The first solution comes in handy if you need to add more complex actions in the if-then statement. The second option adds convenience.

The EBNF for BreakStmt is as follows:

```
BreakStmt = 'break' [ 'on' Condition ] .
Condition = (Boolean) Expr .
```

Based on this the AST node for BreakStmt is pretty simple:

```
  TBreakStmt = class(TStmt)
    private
      FCondition: TExpr;
    public
      property Condition: TExpr read FCondition;
      constructor Create(ACondition: TExpr; AToken: TToken);
      destructor Destroy; override;
  end;
```

The implementation:

```
constructor TBreakStmt.Create(ACondition: TExpr; AToken: TToken);
begin
  inherited Create(AToken);
  FCondition := ACondition;
end;

destructor TBreakStmt.Destroy;
begin
  if Assigned(FCondition) then FCondition.Free;
  inherited Destroy;
end;
```

In the parser add ttBreak to the constant StmtStartSet:

```
const
  StmtStartSet: TTokenTypSet = [ttIf, ttWhile, ttRepeat, ttFor,
    ttSwitch, ttEnsure, ttPrint, ttBreak, ttIdentifier];
```

Next, we are going to amend the Parser class a bit. Obviously, the usage of the Break statement should be restricted to while-loops and for-loops. I will leave the Repeat-loop as an exercise to the reader, but after this paragraph it shouldn't be too hard to build it in.

Create a new private field 'LoopDepth', which keeps track of the fact that we are inside a loop. A LoopDepth of 0 means we are not inside a loop, and thus the use of Break is not allowed. Of course you can have loops inside loops, in which case the LoopDepth counter increases.

```
  TParser = class
    ...
    private
      Tokens: TTokens;
      Current: Integer;
      LoopDepth: Integer;
    ...
  end;
```

In the Parser constructor set the initial LoopDepth to zero.

```
constructor TParser.Create(Lexer: TLexer);
begin
  ...
  LoopDepth := 0;
end;
```

Then, we parse the Break statement. First, add the case for ttBreak to ParseStmt:

```
function TParser.ParseStmt: TStmt;
begin
  case CurrentToken.Typ of
    ...
    ttBreak: Result := ParseBreakStmt;
    ...
  end;
end;
```

Function ParseBreakStmt first checks the LoopDepth. If it is zero, it means we are not in a while-loop or for-loop, so the use is illegal. If the keyword 'on' was detected we also parse the condition on which to break. Otherwise Condition will be Nil.

```
function TParser.ParseBreakStmt: TStmt;
var
  Token: TToken;
  Condition: TExpr=Nil;
begin
  if LoopDepth = 0 then
    Error(CurrentToken, 'Break can only be used from inside a loop.');
  Token := CurrentToken;
  Next; // skip Break
  if CurrentToken.Typ = ttOn then begin
    Next; // skip On
    Condition := ParseExpr;
  end;
  Result := TBreakStmt.Create(Condition, Token);
end;
```

The parse function for the for-statement and while-statement need to be changed so that we include the LoopDepth counter. If we enter either function we increase the depth and if we leave again we decrease the depth. We use a try-finally block for this so that we are sure the decrease takes place.

The amended for-statement:

```
function TParser.ParseForStmt: TStmt;
...
begin
  try
    Inc(LoopDepth);
    ...
  finally
    Dec(LoopDepth);
  end;
end;
```

The amended while-statement:

```
function TParser.ParseWhileStmt: TStmt;
...
begin
  try
    Inc(LoopDepth);
    ...
  finally
    Dec(LoopDepth);
  end;
end;
```

Here are the typical visitors for the printer and the resolver. Straightforward!

```
procedure TPrinter.VisitBreakStmt(BreakStmt: TBreakStmt);
begin
  IncIndent;
  VisitNode(BreakStmt);
  if Assigned(BreakStmt.Condition) then
    Visit(BreakStmt.Condition);
  DecIndent;
end;
```

And the resolver:

```
procedure TResolver.VisitBreakStmt(BreakStmt: TBreakStmt);
begin
  if Assigned(BreakStmt.Condition) then
    Visit(BreakStmt.Condition);
end;
```

in the uError.pas unit create a new exception class, called EBreakException, descending from class Exception. EBreakException is the exception that is thrown in the Break statement, and will be caught in the while statement.

```
  EBreakException = class(Exception);
```

In interpreter visitor VisitBreakStmt a local variable Condition initially is set to True, and if the break statement has a condition attached, may get a new value based on visiting that condition. Only if the Condition is True the exception is thrown. If the Condition, after visiting the Node.Condition, is not of type Boolean a runtime error is generated.

```
procedure TInterpreter.VisitBreakStmt(BreakStmt: TBreakStmt);
var
  Condition: Variant;
begin
  Condition := True;
  if Assigned(BreakStmt.Condition) then
    Condition := Visit(BreakStmt.Condition);

  if not VarIsBool(Condition) then
    Raise ERuntimeError.Create(BreakStmt.Token, ErrConditionMustBeBool);

  if Condition then
    raise EBreakException.Create('');
end;
```

In the interpreter's visitor for the while statement, inside the try...finally block, create a new try...except block. The actual looping is inside the try part, whereas in the except part we catch the EBreakException error, but we don't need to perform any further action on that. The loop will just stop and exit. The original finally part will be executed always!

```
procedure TInterpreter.VisitWhileStmt(Node: TWhileStmt);
...
begin
  try
    ...
    try
      Condition := Visit(Node.Condition);
      if VarIsBool(Condition) then begin
        while Condition do begin
          Visit(Node.Block);
          Condition := Visit(Node.Condition);
        end;
      end
      else
        Raise ERuntimeError.Create(Node.Token, 'Condition is not Boolean.');
    except
      on EBreakException do;
    end;
  finally
    ...
end;
```

Test it

```
while var i:=0 where i<10 do
  if i=5 then
    break
  end
  print(i)
  i+=1
end
print()

while var i:=0 where i<10 do
  break on i=5
  print(i)
  i+=1
end
print()

for var i := 0 where i<20, i+=1 do
  break on i>=9
  print(i)
end
```

Chapter 5 – Functions

In mathematics, a function is a relation between a set of inputs and a set of permissible outputs with the property that each input is related to exactly one output. The output of a function f corresponding to an input x is denoted by f(x). The input variable(s) are sometimes referred to as the argument(s) of the function. With this in mind, I will use the keyword 'func' for a function, and use parentheses () for containing parameters.

Make a copy of the chapter 4 code and paste it into a new folder Ch05.
Examples of Gear function definitions are:

```
func F(n)
  if n < 2 then
    return n
  else
    return F(n-1) + F(n-2)
  end
end

func sum(a, b)
  return a+b
end
```

or in a simpler way if only one expression is returned:

```
func sum(a, b) => a+b
```

Then, a function is called as (part of) an expression:

```
var a := sum(12, 30)
var f := F(10)
```

Next to this it will be possible to give alternative names to parameters, for example

```
func add(a, to b) => a+b
var a := add(17, to: 25)

func save(.filename)
  /.../
end
save(filename: 'data.txt')
```

But, we'll build it up step by step, and thus start as simple as possible.

Function declaration

> The chief function of the body is to carry the brain around.
> — Thomas A. Edison

We start with defining the EBNF of a function declaration. So far, the EBNF of Declaration only had a VarDecl (for both var and let) defined. Now it becomes:

```
Declaration = VarDecl | FuncDecl .
FuncDecl = ‘func’ Ident ‘(‘ [ Parameters ] ‘)’ Body ‘end’ .
Parameters = Ident { ‘,’ Ident } .
Body = Block .
```

This is the first version we’re going to build and use. The AST node for TFuncDecl becomes:

```
  TFuncDecl = class(TDecl)
    private
      type
        TParam = class
          Ident: TIdent;
          constructor Create(AIdent: TIdent);
          destructor Destroy; override;
        end;
        TParamList = specialize TArrayObj<TParam>;
      var
        FParams: TParamList;
        FBody: TBlock;
    public
      property Params: TParamList read FParams;
      property Body: TBlock read FBody write FBody;
      constructor Create(AIdent: TIdent; AToken: TToken);
      destructor Destroy; override;
      procedure AddParam(AIdent: TIdent);
  end;
```

There are a few things worth mentioning here. We define a private type TParamList, which holds the parameters (identifiers only for now). Procedure AddParam adds a parameter to this list. Next to this a function holds an identifier (handled in TDecl) and a body (of type TBlock)). The body will be added manually (meaning not in the Create constructor).

The implementation is straightforward:

```
constructor TFuncDecl.TParam.Create(AIdent: TIdent);
begin
  Ident := AIdent;
end;

destructor TFuncDecl.TParam.Destroy;
begin
  if Assigned(Ident) then Ident.Free;
  inherited Destroy;
end;

constructor TFuncDecl.Create(AIdent: TIdent; AToken: TToken);
begin
  Inherited Create(AIdent, AToken);
  FBody := Nil;
  FParams := TParamList.Create();
end;
```

```
destructor TFuncDecl.Destroy;
begin
  FParams.Free;
  if Assigned(FBody) then FBody.Free;
  inherited Destroy;
end;

procedure TFuncDecl.AddParam(AIdent: TIdent);
begin
  FParams.Add(TParam.Create(AIdent));
end;
```

Method AddParam adds the parameter while creating it.

We need to prepare the parser in order to parse function declarations. First, add the tokentype ttFunc to DeclStartSet:

```
const
  DeclStartSet: TTokenTypSet = [ttFunc, ttLet, ttVar];
```

Next, function ParseDecl needs to check for ttFunc:

```
function TParser.ParseDecl: TDecl;
const Mutable = True;
begin
  case CurrentToken.Typ of
    ttFunc: Result := ParseFuncDecl(ffFunction);
    ttLet: Result := ParseVarDecl(Mutable = False);
    ttVar: Result := ParseVarDecl(Mutable = True);
  end;
end;
```

You'll notice that method ParseFuncDecl takes a parameter: 'ffFunction'. This implies there will be more to come. The 'ff' stands for function form, and we'll need this later on to distinguish between different function forms. To this end create a new private type inside the class TParser:

```
  TParser = class
    private
      type
        TFuncForm = (ffFunction);
    Private
    ...
  end;
```

It does look a bit overdone with only 1 enum, but bear with me.

For parsing a function declaration, according to the EBNF and the AST node, we need to parse a function identifier, the parameters if there ar any and a function body.

We also have to prepare a bit for future extension of the different function forms, so you might think I'm going outside the lines…

```
function TParser.ParseFuncDecl(FuncForm: TFuncForm): TDecl;
var
  FuncDecl: TFuncDecl;
  Token: TToken;
  Name: TIdent = Nil;

  procedure ParseParameters;

    procedure ParseParam;
    begin
      FuncDecl.AddParam(ParseIdent);
    end;

  begin
    if CurrentToken.Typ <> ttCloseParen then begin
      ParseParam;
      while CurrentToken.Typ = ttComma do begin
        Next; // skip comma
        ParseParam;
      end;
    end;
  end;

begin
  Token := CurrentToken;
  case FuncForm of
    ffFunction: begin Next; Name := ParseIdent; end;
  end;
  FuncDecl := TFuncDecl.Create(Name, Token);
  Expect(ttOpenParen);
  ParseParameters;
  Expect(ttCloseParen);
  FuncDecl.Body := ParseBlock;
  Expect(ttEnd);
  Result := FuncDecl;
end;
```

After reading the FuncDecl token, we perform some action based on the FuncForm parameter. For enum ffFunction, which resembles a regular function, this is skipping the 'func' keyword and parsing the name of the function. We then create a function declaration – FuncDecl.

Always expect opening and closing parens. We also always parse the parameters. Procedure ParseParameters is a nested procedure as it's defined inside ParseFuncDecl. If, after the opening paren, we don't find a closing paren, there must be parameters. We parse and add the first parameter by calling yet another nested procedure, and while we find comma's, we continue to parse and add parameters. If no more comma's are found, we expect a closing paren.

This seems unnecessary, however, when we start parsing external parameters, it makes more sense, for readibility reasons.

Then, we parse and add the body as a block with statements and declarations to FuncDecl. Finally, we expect the 'end' keyword.

For sure the whole ParseFuncDecl method is ready for the moment but will get updated very soon.

Next, the visitor methods for the printer and the interpreter.

```
procedure TPrinter.VisitFuncDecl(FuncDecl: TFuncDecl);
var
  i: Integer;
begin
  IncIndent;
  VisitNode(FuncDecl);  // Print FuncDecl
  if Assigned(FuncDecl.Ident) then
    Visit(FuncDecl.Ident);
  IncIndent;
  WriteLn(Indent, 'Parameters:');
  for i := 0 to FuncDecl.Params.Count-1 do begin
    Visit(FuncDecl.Params[i].Ident);
  end;
  DecIndent;
  Visit(FuncDecl.Body);
  DecIndent;
end;
```

And the resolver, just declaring and enabling the function identifier, the rest follows later:

```
procedure TResolver.VisitFuncDecl(FuncDecl: TFuncDecl);
begin
  Declare(FuncDecl.Ident);
  Enable(FuncDecl.Ident);
end;
```

```
procedure TInterpreter.VisitFuncDecl(FuncDecl: TFuncDecl);
begin
  // do nothing yet
end;
```

We leave the interpreter visitor blank for now, as we'll focus on function calling first. However, compiling and printing the program with below input gives the AST we want:

```
var z:=0
func add(a,b,c)
  z:=a+b+c
end
```

```
Product
  VarDecl
    Ident: z
    0
  FuncDecl
    Ident: add
    Parameters:
      Ident: a
      Ident: b
      Ident: c
    Block
      AssignStmt
      (:=)
        Var: z
        (+)
          (+)
            Var: a
            Var: b
          Var: c
```

Function Calls

> Form follows function - that has been misunderstood. Form and function should be one, joined in a spiritual union.
>
> — Frank Lloyd Wright

A wide number of conventions for the coding of subroutines have been developed. Pertaining to their naming, many developers have adopted the approach that the name of a subroutine should be a verb when it does a certain task, an adjective when it makes some inquiry, and a noun when it is used to substitute variables.

Part of the expression EBNF so far is the UnaryExpr, which was defined as:

```
UnaryExpr  = [ '+' | '-' | '!' ] UnaryExpr | Factor  .
```

We enhance this part so that it can cope with function calls. The EBNF of calling a function (CallExpr) with arguments is as follows:

```
UnaryExpr  = [ '+' | '-' | '!' ] UnaryExpr | CallExpr  .
CallExpr   = Factor { '(' [ Arguments ] ')' } .
Arguments  = ExprList .
ExprList   = Expr { ',' Expr } .
Factor     = String | Char | Number | Nil
           | '(' Expr ')'
           | Ident
           | IfExpr | MatchExpr .
```

The EBNF changes such that a UnaryExpr now refers to a CallExpr, while a CallExpr now refers to Factor that may have multiple arguments and even multiple calls: Ident()()()…

With this, the AST node for a call expression is

```
TCallExpr = class(TExpr)
  private
    type
      TArg = class
        Expr: TExpr;
        constructor Create(AExpr: TExpr);
        destructor Destroy; override;
      end;
      TArgList = specialize TArrayObj<TArg>;
  private
    FCallee: TExpr;
    FArgs: TArgList;
  public
    property Callee: TExpr read FCallee;
    property Args: TArgList read FArgs;
    constructor Create(ACallee: TExpr; AToken: TToken);
    destructor Destroy; override;
    procedure AddArgument(Expr: TExpr);
end;
```

A private type TArg, containing the argument is defined as a class. It will be extended later on. The arguments come in a list. The callee is the function that's being called.

And the accompanying implementation is:

```
constructor TCallExpr.TArg.Create(AExpr: TExpr);
begin
  Expr := AExpr;
end;

destructor TCallExpr.TArg.Destroy;
begin
  if Assigned(Expr) then Expr.Free;
  inherited Destroy;
end;

constructor TCallExpr.Create(ACallee: TExpr; AToken: TToken);
begin
  inherited Create(AToken);
  FCallee := ACallee;
  FArgs := TArgList.Create();
end;

destructor TCallExpr.Destroy;
begin
  FArgs.Free;
  if Assigned(FCallee) then FCallee.Free;
  inherited Destroy;
end;

procedure TCallExpr.AddArgument(Expr: TExpr);
begin
  FArgs.Add(TArg.Create(Expr));
end;
```

We define a private type TArgList and a variable FArgs that maintain the list of arguments in the call. Public procedure AddArgument adds an argument expression to the list. The constructor Create initializes the list of arguments.

From this part of the EBNF:

```
  CallExpr   = Factor { '(' [ Arguments ] ')' } .
```

We note that we can have multiple parenthesized expression lists, like this:

method(a,b)(c,d)

This states that the result of the call to method(a,b) is used in a second call with arguments (c,d). This can be very helpful in object oriented coding. This is the parse function for it.

```
function TParser.ParseCallExpr: TExpr;
begin
  Result := ParseFactor;
  while CurrentToken.Typ = ttOpenParen do
    Result := ParseCallArgs(Result);
end;
```

Don't' forget to call ParseCallExpr in ParseUnaryExpr instead of ParseFactor.

```
function TParser.ParseUnaryExpr: TExpr;
...
  else
    Result := ParseCallExpr;
end;
```

ParseCallExpr exactly follows the EBNF, except for parsing the '(' and ')'. It first parser a factor, and if a left paren '(' is found it parses the call arguments, passing the factor as the argument.

If there is no left parenthesis '(' the result of the method will be the parsed Factor.

We use a helper function for parsing the arguments, which also takes care of handling the parentheses. The actual TCallExpr is created and filled in function ParseCallArgs.

Inside ParseCallArgs, the arguments are parsed in a nested procedure ParseArg. This seems like overweight, however it will make sense when we'll include parameter names later on. As an example, you can call function add(5, to: 7), where 'to' is a parameter name.

```
function TParser.ParseCallArgs(Callee: TExpr): TExpr;
var
  CallExpr: TCallExpr;
  Token: TToken;

  procedure ParseArg;
  var
    Expr: TExpr;
  begin
    Expr := ParseExpr;
    CallExpr.AddArgument(Expr);
  end;

begin
  Token := CurrentToken;
  Next;  // skip (
  CallExpr := TCallExpr.Create(Callee, Token);
  if CurrentToken.Typ <> ttCloseParen then begin
    ParseArg;
    while CurrentToken.Typ = ttComma do begin
      Next;  // skip ,
      ParseArg;
    end;
  end;
  Expect(ttCloseParen);
  Result := CallExpr;
end;
```

Remember the EBNF of Arguments from above: we parse a first expression, and then while there are commas, continue to parse the next expressions. Finally, we expect a closing parenthesis and return the result as a CallExpr.

The visitor for the printer is:

```
procedure TPrinter.VisitCallExpr(CallExpr: TCallExpr);
var
  i: Integer;
begin
  IncIndent;
  VisitNode(CallExpr);
  Visit(CallExpr.Callee);
  IncIndent;
  Writeln(Indent, 'Arguments:');
  for i := 0 to CallExpr.Args.Count-1 do
    Visit(CallExpr.Args[i].Expr);
  DecIndent;
  DecIndent;
end;
```

It first visits the Callee, and visits all arguments stored in the CallExpr.Args list. This is a list of TArg, which contains an Expr. The Expr is visited in this case.

The resolver visitor is as follows, which visits the callee and all arguments:

```
procedure TResolver.VisitCallExpr(CallExpr: TCallExpr);
var
  i: Integer;
begin
  Visit(CallExpr.Callee);
  for i := 0 to CallExpr.Args.Count-1 do
    Visit(CallExpr.Args[i].Expr);
end;
```

But now, more importantly, we focus on the visitor for the interpreter. This will be a complex adventure! We'll introduce Interfaces as Variants amongst others.

We go step by step top-down, first by showing the interpreter's VisitCallExpr method.

```
  ErrNotAFunction = '"%s" is not defined as function.';
```

Add the error message to the constant list of error messages in the parser.

```
function TInterpreter.VisitCallExpr(CallExpr: TCallExpr): Variant;
var
  Callee: Variant;
  Args: TArgList;
  i: Integer;
  Func: ICallable;
  Msg: String;
  CallArg: TCallArg;
begin
  Callee := Visit(CallExpr.Callee);
  if VarSupports(Callee, ICallable) then
    Func := ICallable(Callee)
  else begin
    Msg := Format(ErrNotAFunction, [CallExpr.Callee.Token.Lexeme]);
    Raise ERuntimeError.Create(CallExpr.Token, Msg);
  end;
  Args := TArgList.Create();
  for i := 0 to CallExpr.Args.Count-1 do begin
    CallArg := TCallArg.Create(Visit(CallExpr.Args[i].Expr));
    Args.Add(CallArg);
  end;
  Result := Func.Call(CallExpr.Token, Self, Args);
end;
```

We start by visiting the node's callee expression and return that into the variable Callee of type Variant. Then, we check if we are dealing with an actual function identifier. If not we report an error. So far so good. But how do we know it's a function? Later on we'll see that functions are stored in the current space as a TFunc type in a Variant, which conforms to the interface ICallable. Wow! In such a case we can perform the check on conformance to ICallable by asking the standard function VarSupports if the Callee conforms to ICallable. If this is the case, we can then set the variable Func to ICallable(Callee), or rather Callee is cast to ICallable…

If it is a function, we continue with the expression arguments. We visit and store them in the list Args, which is of type TArgList. We'll come to that later.

Finally, the function is called and its result is returned. In a nutshell…

Toe make this all work, let's make a side step into the world of callable functions. First, we'll define the interface ICallable, and then we move on to define TFunc, as both are necessary to get the above method to work.

The ICallable interface

Create a new unit uCallable.pas.

```
unit uCallable;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, uCollections, uInterpreter, uAST, uToken;

type
  TCallArg = class
    Value: Variant;
    constructor Create(AValue: Variant);
  end;

  TArgList = specialize TArrayObj<TCallArg>;

  ICallable = interface
    ['{EBA5469A-1CA1-FB54-0D6F-60889AFA4679}']
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
    function toString: String;
  end;

implementation

constructor TCallArg.Create(AValue: Variant);
begin
  Value := AValue;
end;

end.
```

We define an interface type ICallable to which every function should adhere, both declared and standard functions. We'll discuss standard functions later on. The call method accepts a token (for possible error location), the interpreter, which is sometimes required and also the list of argument objects. The argument list contains a class TCallArg, which has currently only the field Value of Variant type. Later on we'll extend the fields. The ICallable principle comes from Bob Nystrom's book.

By the way, the funny looking code ['{EBA5469A-1CA1-FB54-0D6F-60889AFA4679}'] is called a GUID, and can be created automatically by Lazarus via the menu "Source -> Insert General -> Insert a GUID". Make sure the cursor is already in the right position, as that's the location where it will be generated. The GUID is required for interfaces to work.

The TFunc function class

After this, we define the declared function class TFunc, which adopts the ICallable interface. This means the functions defined in ICallable must be implemented. For this, create a new unit uFunc.pas.

```
unit uFunc;

{$mode objfpc}{$H+}

// This code is based on the online book 'Crafting Interpreters',
// written by Bob Nystrom. http://craftinginterpreters.com
// The code is originally written in Java.

interface

uses
  Classes, SysUtils, uCallable, uInterpreter, uAst, uMemory, uError;

type

  TFunc = class(TInterfacedObject, ICallable)
    private
      FFuncDecl: TFuncDecl;
    public
      property FuncDecl: TFuncDecl read FFuncDecl;
      constructor Create(AFuncDecl: TFuncDecl);
      function Call(Token: TToken; Interpreter: TInterpreter;
        ArgList: TArgList): Variant;
      function toString: String; override;
      class procedure CheckArity(Token: TToken; NumArgs, NumParams: Integer); static;
  end;
```

The implementation requires some attention. The first attempt follows here.
The constructor is straightforward:

```
constructor TFunc.Create(AFuncDecl: TFuncDecl);
begin
  FFuncDecl := AFuncDecl;
end;
```

And the first version of the Call function. We will extend it a bit later.

```
function TFunc.Call(Token: TToken; Interpreter: TInterpreter;
  ArgList: TArgList): Variant;
var
  FuncSpace: TMemorySpace;
  i: Integer;
begin
  CheckArity(Token, ArgList.Count, FFuncDecl.Params.Count);
  Result := Null;
  FuncSpace := TMemorySpace.Create(Interpreter.Globals);
  for i := 0 to FFuncDecl.Params.Count-1 do
    FuncSpace.Store(FFuncDecl.Params[i].Ident, ArgList[i].Value);
  Interpreter.Execute(FFuncDecl.Body, FuncSpace);
end;
```

The first thing to do is checking the arity, meaning is the number of arguments the same as the number of parameters.

Since functions don't return values yet, we set the variable Result to Null. Then, we create a new memory space called FuncSpace for this specific function. Note that we set the enclosing scope to Interpreter.Globals.

Then, for all parameter identifiers, we store the argument value into the function space. After this we execute the function's body with the accompanying function space. Note that the Interpreter.Execute(body, space) is not defined yet.

The toString function returns the function's name:

```
function TFunc.toString: String;
begin
  Result := '<func ' + FFuncDecl.Ident.Text+ '>';
end;
```

Next, we perform a check on the number of arguments in the call compared to the number of parameters in the function declaration. In mathematics this is called the arity (the number of arguments or the number of operands a function or operation takes). An error is created if there's a mismatch.

```
class procedure TFunc.CheckArity(Token: TToken; NumArgs, NumParams: Integer);
begin
  if NumArgs <> NumParams then
    Raise ERuntimeError.Create(Token, Format(
      'Invalid number of arguments. Expected %d arguments.',
      [NumParams]));
end;
```

This is a so-called class procedure, which means it doesn't belong to an object/instance but to the class itself. It will be called by other functions later on. The call always has to be preceded by 'TFunc'.

Next, add the public procedure Execute to the interpreter, right below function Execute. Since the parameters differ, we can use the same name Execute.

```
procedure TInterpreter.Execute(Block: TBlock; MemorySpace: TMemorySpace);
var
  SavedSpace: TMemorySpace;
begin
  SavedSpace := CurrentSpace;
  try
    CurrentSpace := MemorySpace;
    Visit(Block);
  finally
    CurrentSpace := SavedSpace;
  end;
end;
```

The current space is temporarily saved in SavedSpace and the function's memory space becomes the current space. The block (function body is a TBlock) is executed (visited) and can handle any variable defined inside the block as well as in the global memory, since that is the enclosing memory space. Finally, after executing the block, the current space is restored.

We have got something to finish yet: the visitor for function declaration. Remember, we left it empty! We'll now finish it.

```
procedure TInterpreter.VisitFuncDecl(FuncDecl: TFuncDecl);
var
  Func: TFunc;
begin
  CheckDuplicate(FuncDecl.Ident, 'Func');
  Func := TFunc.Create(FuncDecl);
  CurrentSpace.Store(FuncDecl.Ident, ICallable(Func));
end;
```

First check if a function with that name doesn't exist yet, and if not, create the function, then store it in the current memory space as an ICallable.

In the interpreter add a uses clause right under the keyword implementation:

```
implementation
uses uCallable, uFunc;
```

Resolving a function

One more thing before we can run a small test. So far, we only declared and enabled the function identifier, but we didn't look at the parameters and scope inside the function.

Start with adding a new type above TResolver:

```
  TFuncKind = (fkNone, fkFunc);
```

Then, change VisitFuncDecl to include a call to ResolveFunction:

```
procedure TResolver.VisitFuncDecl(FuncDecl: TFuncDecl);
begin
  Declare(FuncDecl.Ident);
  Enable(FuncDecl.Ident);
  ResolveFunction(FuncDecl, fkFunc);
end;
```

The helper procedure ResolveFunction creates a new scope, enters the parameters, visits the block and finally leaves the scope again.

```
procedure TResolver.ResolveFunction(Func: TFuncDecl; FuncKind: TFuncKind);
var
  EnclosingFuncKind: TFuncKind;
  i: Integer;
begin
  EnclosingFuncKind := CurrentFuncKind;
  CurrentFuncKind := FuncKind;
  BeginScope;
  for i := 0 to Func.Params.Count-1 do begin
    Declare(Func.Params[i].Ident);
    Enable(Func.Params[i].Ident);
  end;
  Visit(Func.Body);
  EndScope;
  CurrentFuncKind := EnclosingFuncKind;
end;
```

In the TResolver class add a new private field CurrentFuncKind:

```
  TResolver = class(TVisitor)
    ...
    private
      Scopes: TScopes;
      GlobalScope: TScope;
      CurrentScope: TScope;
      CurrentFuncKind: TFuncKind;
    ...
  end;
```

and set it to fkNone in the constructor:

```
constructor TResolver.Create;
begin
    ...
  CurrentFuncKind := fkNone;
end;
```

With all this in place it's possible to run a small test, however still far from what we need, but here it goes.

```
func add(a,b,c)
  print(a+b+c)
end

var x := add(2,3, 4)
print(x)
```

Running it with gives as output:

```
9
Null
```

The AST looks as follows:

```
    Product
      FuncDecl
        Ident: add
        Parameters:
          Ident: a
          Ident: b
          Ident: c
        Block
          PrintStmt
            (+)
              (+)
                Var: a
                Var: b
              Var: c
      VarDecl
        Ident: x
        CallExpr
          Var: add
          Arguments:
            2
            3
            4
      PrintStmt
        Var: x
```

And that is conform expectation, because a function call now returns Null. Also, we cannot call functions as statements yet, for which we have to create a small adjustment.

But, the first thing we want is to be able to return a value from a function.

Return from a function

> Time travel used to be thought of as just science fiction, but Einstein's general theory of relativity allows for the possibility that we could warp space-time so much that you could go off in a rocket and return before you set out.
>
> — Stephen Hawking

According to WikiPedia, a return statement causes execution to leave the current subroutine and resume at the point in the code immediately after where the subroutine was called, known as its return address. The return address is saved, usually on the process's call stack, as part of the operation of making the subroutine call. Return statements in many languages allow a function to specify a return value to be passed back to the code that called the function.

In our case, while we create an interpreter, we're not really talking about addresses here, but under the hood our solution definitely uses addresses…

Examples of functions with return statements are:

```
func add(a, b)
  return a+b
end

func factorial(n)
  return match n
    if 0 then 0
    if 1 then 1
    else n*factorial(n-1)
end
```

As said, return is a statement, with the following EBNF:

```
  ReturnStmt = 'return' Expr .
```

This gives the AST node:

```
  TReturnStmt = class(TStmt)
    private
      FExpr: TExpr;
    public
      property Expr: TExpr read FExpr;
      constructor Create(AExpr: TExpr; AToken: TToken);
      destructor Destroy; override;
  end;
```

And the implementation:

```
constructor TReturnStmt.Create(AExpr: TExpr; AToken: TToken);
begin
  inherited Create(AToken);
  FExpr := AExpr;
end;

destructor TReturnStmt.Destroy;
begin
  if Assigned(FExpr) then FExpr.Free;
  inherited Destroy;
end;
```

Creating the parser for a return statement is thus, based on the EBNF:

```
function TParser.ParseReturnStmt: TStmt;
var
  Token: TToken;
  Expr: TExpr;
begin
  Token := CurrentToken;
  Next; // skip return
  Expr := ParseExpr;
  Result := TReturnStmt.Create(Expr, Token);
end;
```

Don't forget to add ttReturn to the StatementStartSet and add it to ParseStmt.

```
  StmtStartSet: TTokenTypSet = [ttIf, ttWhile, ttRepeat, ttFor, ttReturn,
    ttSwitch, ttEnsure, ttPrint, ttBreak, ttIdentifier];
```

```
function TParser.ParseStmt: TStmt;
begin
  case CurrentToken.Typ of
    ...
    ttFor: Result := ParseForStmt;
    ttReturn: Result := ParseReturnStmt;
    ...
end;
```

The accompanying visitor for the printer is:

```
procedure TPrinter.VisitReturnStmt(ReturnStmt: TReturnStmt);
begin
  IncIndent;
  VisitNode(ReturnStmt);
  Visit(ReturnStmt.Expr);
  DecIndent;
end;
```

Then, we come to the Resolver. We first check if we are in a non-function scope, and if not generate an error. Return statements are only allowed from functions.

Add to the error constants:

```
  ErrReturnFromFunc = 'Return can only be used from a function.';
```

```
procedure TResolver.VisitReturnStmt(ReturnStmt: TReturnStmt);
begin
  if CurrentFuncKind = fkNone then
    with ReturnStmt do
      Errors.Append(Token.Line, Token.Col, ErrReturnFromFunc);
  Visit(ReturnStmt.Expr);
end;
```

When a return statement is encountered, we need a way to return immediately from the function with the given value evaluated from the expression. Easiest way to handle this is through the use of exceptions. We create a special runtime error: EReturnFromFunc. Add the following to uError.pas:

```
  EReturnFromFunc = class(Exception)
    Value: Variant;
    constructor Create(AValue: Variant);
  end;
```

And the implementation:

```
constructor EReturnFromFunc.Create(AValue: Variant);
begin
  Value := AValue;
  inherited Create('return');
end;
```

Now, on to the interpreter for visitor VisitReturnStmt:

```
procedure TInterpreter.VisitReturnStmt(ReturnStmt: TReturnStmt);
begin
  raise EReturnFromFunc.Create(Visit(ReturnStmt.Expr));
end;
```

It just raises the EReturnFromFunc exception with the evaluated (visited) expression.

For our functions to work there's only one change to make. We have to catch the EReturnFromFunc exception. This is done in the 'call' function of class TFunc.

Here it is complete!

```
function TFunc.Call(Token: TToken; Interpreter: TInterpreter;
  ArgList: TArgList): Variant;
var
  FuncSpace: TMemorySpace;
  i: Integer;
begin
  CheckArity(Token, ArgList.Count, FFuncDecl.Params.Count);
  Result := Null;
  FuncSpace := TMemorySpace.Create(Interpreter.Globals);
  for i := 0 to FFuncDecl.Params.Count-1 do
    FuncSpace.Store(FFuncDecl.Params[i].Ident, ArgList[i].Value);
  try
    Interpreter.Execute(FFuncDecl.Body, FuncSpace);
  except
    On E: EReturnFromFunc do
      Result := E.Value;
  end;
end;
```

Let's try it out, for example with the following code:

```
func sum(a,b,c)
  return a+b+c
end

var x := sum(2,3,4)
print(x) // prints 9
```

Or this code:

```
func factorial(n)
  var f := 1
  for var i:=1 where i<=n, i+=1 do
    f *= i
```

```
  end
  return f
end

print(factorial(10))
```

Or as a recursive call:

```
func fac(n)
  if n<2 then
    return 1
  else
    return n * fac(n-1)
  end
end

print(fac(10))
```

Calling functions as statements

A function can also be called without returning a value. In fact the returned value would be Null. You just call the function to process some action. An example is:

> An iPod, a phone, an internet mobile communicator... these are NOT three separate devices! And we are calling it iPhone! Today Apple is going to reinvent the phone. And here it is.
>
> — Steve Jobs

```
func sum(a,b,c,d)
  print(a+b+c+d)
end

sum(187,327,123,9845)
```

For this to work we will use some existing code, but also have to create a new statement type. So far, the function ParseStmt is

```
function TParser.ParseStmt: TStmt;
begin
  case CurrentToken.Typ of
    ttIf: Result := ParseIfStmt;
    ttWhile: Result := ParseWhileStmt;
    ttRepeat: Result := ParseRepeatStmt;
    ttFor: Result := ParseForStmt;
    ttReturn: Result := ParseReturnStmt;
    ttSwitch: Result := ParseSwitchStmt;
    ttEnsure: Result := ParseEnsureStmt;
    ttPrint: Result := ParsePrintStmt;
    ttBreak: Result := ParseBreakStmt;
    else
      Result := ParseAssignStmt;
  end;
end;
```

A function call starts with an identifier, followed by arguments between left and right parens. The identifier is detected in the else clause in the ParseAssignStmt. We will use this function to also parse function calls. But first, the AST node for TCallExprStmt. Put it right below TAssignStmt.

```
  TCallExprStmt = class(TStmt)
    private
      FCallExpr: TCallExpr;
    public
      property CallExpr: TCallExpr read FCallExpr;
      constructor Create(ACallExpr: TCallExpr; AToken: TToken);
      destructor Destroy; override;
  end;

constructor TCallExprStmt.Create(ACallExpr: TCallExpr; AToken: TToken);
begin
  inherited Create(AToken);
  FCallExpr := ACallExpr;
end;

destructor TCallExprStmt.Destroy;
begin
  if Assigned(FCallExpr) then FCallExpr.Free;
  inherited Destroy;
end;
```

The field to record is actually a Call expression: Reuse!
The function ParseAssignStmt changes as follows:

```
function TParser.ParseAssignStmt: TStmt;
...
begin
  ...
  if CurrentToken.Typ in AssignSet then begin
    ...
  end
  else if Left is TCallExpr then
    Result := TCallExprStmt.Create(Left as TCallExpr, Token)
  else
    Error(Token, ErrExpectedAssignOpFunc);
end;
```

Above the line 'else error(…)' an additional if-statement was added. In case it is not an assignment, a check is done whether the left hand side is a Call expression, and if this is the case, the result is a TCallExprStmt. Change the error to the following:

```
  ErrExpectedAssignOpFunc = 'Expected assignment operator, or function call.';
```

To finalize, we create the visitors for the printer,

```
procedure TPrinter.VisitCallExprStmt(CallExprStmt: TCallExprStmt);
begin
  IncIndent;
  VisitNode(CallExprStmt);
  Visit(CallExprStmt.CallExpr);
  DecIndent;
end;
```

the Resolver,

```
procedure TResolver.VisitCallExprStmt(CallExprStmt: TCallExprStmt);
begin
  Visit(CallExprStmt.CallExpr);
end;
```

and the interpreter:

```
procedure TInterpreter.VisitCallExprStmt(CallExprStmt: TCallExprStmt);
begin
  Visit(CallExprStmt.CallExpr);
end;
```

And that's all folks! Test it and see the proof. This shows that both function call and assignment work.

```
func sum(a,b,c,d)
  print(a+b+c+d)
end
sum(1,2,3,4)     //10
var a := 1
var b := 2
var c := 3
var d := 4
var z := 0
sum(a,b,c,d)     //10
z := a+b+c+d
print(z)         //10
```

Standard functions

> We are all different. There is no such thing as a standard or run-of-the-mill human being, but we share the same human spirit.
>
> — Stephen Hawking

A language is only usable if it has a number of standard functions available that support the programmer in creating programs. We call this the standard library. This paragraph describes this library. First, we start with the code to add to the interpreter's constructor. Remember, we introduced the variable FGlobals. This was on purpose, especially for the standard routines, which are always available in the global memory space.

```
constructor TInterpreter.Create;
begin
  FGlobals := TMemorySpace.Create();
  FGlobals.Sorted := True;
  FGlobals.Store('abs', ICallable(TAbs.Create));
  FGlobals.Store('arctan', ICallable(TArctan.Create));
  FGlobals.Store('chr', ICallable(TChr.Create));
  FGlobals.Store('cos', ICallable(TCos.Create));
  FGlobals.Store('date', ICallable(TDate.Create));
  FGlobals.Store('exp', ICallable(TExp.Create));
  FGlobals.Store('frac', ICallable(TFrac.Create));
  FGlobals.Store('length', ICallable(TLength.Create));
  FGlobals.Store('ln', ICallable(TLn.Create));
  FGlobals.Store('milliseconds', ICallable(TMilliSeconds.Create));
  FGlobals.Store('now', ICallable(TNow.Create));
  FGlobals.Store('ord', ICallable(TOrd.Create));
  FGlobals.Store('pi', ICallable(TPi.Create));
  FGlobals.Store('random', ICallable(TRandom.Create));
  FGlobals.Store('randomLimit', ICallable(TRandomLimit.Create));
  FGlobals.Store('round', ICallable(TRound.Create));
  FGlobals.Store('sin', ICallable(TSin.Create));
  FGlobals.Store('sqr', ICallable(TSqr.Create));
  FGlobals.Store('sqrt', ICallable(TSqrt.Create));
  FGlobals.Store('time', ICallable(TTime.Create));
  FGlobals.Store('trunc', ICallable(TTrunc.Create));
  CurrentSpace:= FGlobals;
end;
```

We add for each standard function its name (in alphabetical order) and its class instance as an ICallable into the Globals memory space.

Let's move on the the implementation of the standard functions. I show a limited set, which can be extended upon need. Create a new unit uStandard.pas. The code can be found in the Appendix. As an example consider the standard function pi(), which returns the number 3.141592……

```
  TPi = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;
```

and it's implementation (function pi() has zero parameters):

```
function TPi.Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList
  ): Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 0);
  Result := pi;
end;
```

And here's a standard function with a parameter.

```
  TSqrt = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;


function TSqrt.Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList
  ): Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 1);
  Result := Sqrt(ArgList[0].Value);
end;
```

Add to the uses clause in the implementation of uInterpreter.pas the unit uStandard:

```
implementation
uses uCallable, uFunc, uStandard;
```

Then, in the Resolver we need to make sure the function names are declared and enabled. The following procedure takes care of this:

```
procedure TResolver.EnableStandardFunctions;
begin
  with GlobalScope do begin
    Enter(TSymbol.Create('abs', Enabled, False));
    Enter(TSymbol.Create('arctan', Enabled, False));
    Enter(TSymbol.Create('chr', Enabled, False));
    Enter(TSymbol.Create('cos', Enabled, False));
    Enter(TSymbol.Create('date', Enabled, False));
    Enter(TSymbol.Create('exp', Enabled, False));
    Enter(TSymbol.Create('frac', Enabled, False));
    Enter(TSymbol.Create('length', Enabled, False));
    Enter(TSymbol.Create('ln', Enabled, False));
    Enter(TSymbol.Create('milliseconds', Enabled, False));
    Enter(TSymbol.Create('now', Enabled, False));
    Enter(TSymbol.Create('ord', Enabled, False));
    Enter(TSymbol.Create('pi', Enabled, False));
    Enter(TSymbol.Create('random', Enabled, False));
    Enter(TSymbol.Create('randomLimit', Enabled, False));
    Enter(TSymbol.Create('round', Enabled, False));
    Enter(TSymbol.Create('sin', Enabled, False));
    Enter(TSymbol.Create('sqr', Enabled, False));
    Enter(TSymbol.Create('sqrt', Enabled, False));
    Enter(TSymbol.Create('time', Enabled, False));
    Enter(TSymbol.Create('trunc', Enabled, False));
  end;
end;
```

In the constructor of the Resolver call this procedure:

```
constructor TResolver.Create;
begin
  GlobalScope := TScope.Create();
  GlobalScope.Sorted := True;
  EnableStandardFunctions;
  ...
end;
```

This means that everytime you add a standard function, you have to make available in both the resolver and the interpreter.

I ran the following input, which results in this output:

```
print(milliseconds())  // milliseconds past midnight
print(sqrt(16))        // same as 16^0.5
print(sqr(16))         // same as 16^2
print(now())           // current time with millisecs
print(time())          // current time
print(date())          // current date
print(chr(65))         // char for ascii 65 = A
print(ord("A))         // ascii value of A
print(arctan(45))      // arctangent
print(frac(3.1415))    // fraction of 3.1415 = 0.1415
print(ln(1))           // natural log
print(exp(1))          // is number e
print(ln(exp(1)))      // natural log of e = 1
print(pi())            // 3.14159265...
print(sin(pi()/6))     // sinus
print(cos(pi()/4))     // cosinus
print(abs(-8))         // absolute value is positive
print(trunc(3.1415))   // chop of the fraction
print(round(5.78))     // round to nearest integer
print(random())        // random number between [0..1>
print(randomLimit(6))  // random integer from [0..5]
```

```
63286959
4
256
17:34:46.959
17:34:46
24-1-18
A
65
1.54857776146818
0.1415
0
2.71828182845905
1
3.14159265358979
0.5
0.707106781186548
8
3
6
```

Pretty neat right? It's all available to be used. Of course many more functions can be added as standard.

In the next paragraph we dive into the mechanics of external parameter names.

Name your parameter

At the start of chapter I mentioned a number of examples that function parsing and processing should adhere to. I covered a lot but still not everything. A few programming languages offer the possibility to optionally name parameters. The next examples show what we will cover in this paragraph.

> Freedom is not the absence of obligation or restraint, but the freedom of movement within healthy, chosen parameters.
> — Kristin Armstrong

```
function save(.fileName)
  /* … */
end

save(fileName: "output.txt")

function add(x, to y)
  return x + y
end

var x := add(7, to: 8)
```

In the first function declaration the parameter 'filename' contains a '.' in front of it, and in the second example parameter 'b' has the word 'to' in front of it.

By putting a '.' in front of a parameter declaration, it means that when the respective function is called from a statement or expression, the parameter name must be used in the call. In fact, the parameter name becomes the external name.

By putting an external name in the declaration of the parameter, it means the external name must be used in the call to the function. First, make the following changes to TFuncDecl:

```
  TFuncDecl = class(TDecl)
    private
      type
        TParam = class
          ...
          ExtIdent: TIdent;
          constructor Create(AIdent, AExtIdent: TIdent);
          ...
        end;
        ...
      var
        ...
    public
      ...
      procedure AddParam(AIdent, AExtIdent: TIdent);
  end;
```

Add a new variable ExtIdent (external identifier). Procedure AddParam adds both of them.

```
procedure TFuncDecl.AddParam(AIdent, AExtIdent: TIdent);
begin
  FParams.Add(TParam.Create(AIdent, AExtIdent));
end;
```

Instead of adding an Ident, it now adds both Ident and ExtIdent inside a TParam to the list of parameters.

The constructor for TParam has to handle ExtIdent:

```
constructor TFuncDecl.TParam.Create(AIdent, AExtIdent: TIdent);
begin
  ...
  ExtIdent := AExtIdent;
end;
```

And the destructor has to handle the case that Ident and ExtIdent could be the same, or different or ExtIdent can be Nil. If they are the same then only Ident.Free is required.

```
destructor TFuncDecl.TParam.Destroy;
begin
  if Assigned(Ident) then
    if Ident.Equals(ExtIdent) then
      Ident.Free
    else if Assigned(ExtIdent) then begin
      Ident.Free;
      ExtIdent.Free;
    end;
  inherited Destroy;
end;
```

Also, the parsing of parameters needs to change. We have three possibilities now:

- (name), which is the original form
- (.name), in which name also becomes the external name and thus required in calls
- (external_name name), in which we have an external name required in calls

```
function TParser.ParseFuncDecl(FuncForm: TFuncForm): TDecl;
...

  procedure ParseParameters;

    procedure ParseParam;
    var Ident: TIdent;
    begin
      case CurrentToken.Typ of
        ttDot: begin
          Next;
          Ident := ParseIdent;
          FuncDecl.AddParam(Ident, Ident);
        end;
        ttIdentifier: begin
          Ident := ParseIdent;
          if CurrentToken.Typ = ttIdentifier then
            FuncDecl.AddParam(ParseIdent, Ident)
          else
            FuncDecl.AddParam(Ident, Nil);
        end
        else Error(CurrentToken, 'Invalid parameter.');
      end;
    end;

  begin
    ...
  end;

begin
  ...
end;
```

We use the power of Pascal and its nested routines. The function ParseParam does the magic of parsing both the external parameter ident, if available, and the normal parameter ident. First, it checks if the current token is a dot '.' or an identifier. In case of a dot, the parsed identifier also becomes the external external parameter. In case of an identifier, the first parameter ident is parsed. If the next token is an identifier, there's an ExtIdent and we parse that. If the next token is not an identifier, we actually have the original way of using parameters, and we set the external ident to Nil.

The visitor for the printer changes to the following:

```
procedure TPrinter.VisitFuncDecl(Node: TFuncDecl);
...
begin
  ...
  for i := 0 to Node.Params.Count-1 do begin
    if Assigned(Node.Params[i].ExtIdent) then
      Write(Indent, 'ExtIdent: ', Node.Params[i].ExtIdent.Text);
    Visit(Node.Params[i].Ident);
  end;
  ...
end;
```

Only if the ExtIdent is not nil, it is printed.

The visitor for the Resolver and Interpreter doesn't change. Now, let's look at calling the functions.

The AST node for TCallExpr changes to this new code, now also taking care of calls with external idents.

```
  TCallExpr = class(TExpr)
    private
      type
        TArg = class
          ...
          Ident: TIdent;
          constructor Create(AExpr: TExpr; AIdent: TIdent);
          ...
        end;
          ...
    private
      ...
    public
      ...
      procedure AddArgument(Expr: TExpr; Ident: TIdent);
  end;
```

The constructor and destructor handle the identifier:

```
constructor TCallExpr.TArg.Create(AExpr: TExpr; AIdent: TIdent);
begin
  ...
  Ident := AIdent;
end;

destructor TCallExpr.TArg.Destroy;
begin
  if Assigned(Ident) then Ident.Free;
  ...
end;
```

Also procedure AddArgument has changed:

```
procedure TCallExpr.AddArgument(Expr: TExpr; Ident: TIdent);
begin
  FArgs.Add(TArg.Create(Expr, Ident));
end;
```

Go to the function ParseCallArgs. Change the nested routine ParseArg that parses the actual argument. It first parses an expression. Then, if the next token is a ':', it means we have just parsed an identifier, which must be the external parameter name. After skipping the colon, the actual expression is parsed. If there was no colon, it means we just parsed an actual expression and the external ident is Nil. Both Expr and Ident are added to the argument list.

```
function TParser.ParseCallArgs(Callee: TExpr): TExpr;
var
  ...

  procedure ParseArg;
  var
    ...
    Ident: TIdent = Nil;
  begin
    Expr := ParseExpr;
    if CurrentToken.Typ = ttColon then begin
      Ident := TVariable(Expr).Ident; // ExtIdent parsed as Expr
      Next; // skip :
      Expr := ParseExpr;
    end;
    CallExpr.AddArgument(Expr, Ident);
  end;

begin
  ...
end;
```

With this in place it is possible to parse function calls like this:

```
var c := add(7, to: 8)
save(filename: 'hello.txt')
```

There is one 'but' in all this. There's no checking whether the passed alternative name is correct. Is it the same name that was defined in the function declaration? During parsing we cannot know this, which means it has to be checked during interpretation.

For execution we only need the value of the expression and not the alternative identifier name. Again, no checking is done on correctness of the name. We'll do that in a moment.

Test and run what we have so far. For example, the following program:

```
func add(a, to b)
  return a+b
end

var c := add(7, to: 8)
```

```
print(c)

func save(.filename)
  print('Saved to ', filename)
end

save(filename: 'hello.txt')

print(sqr(12))
```

returns:

```
15
Saved to hello.txt
144
```

As promised we need to build in some checks to ensure parameter names are used correctly.

Checking alternative parameter identifiers

> True simplicity is, well, you just keep on going and going until you get to the point where you go, 'Yeah, well, of course.' Where there's no rational alternative.
>
> — Jonathan Ive

Up to this moment, while we can use alternative names for parameters, there's no checking on whether the alternative name is correct, or whether it's allowed or even if it must be there. In this paragraph we make the necessary changes to the interpreter and the function class.

Our example function for this paragraph is:

```
func add(x, to y) return x+y end
```

If we call the function correctly, like this:

```
var a := add(19, to: 23)
```

it will pass as OK. However, we'll look into the following cases:

```
var a := add(19, ot: 23)  // Alternative identifiers mismatch: expected "to:".
var a := add(19, 23)      // Expected alternative identifier: “to”.
var a := add(from: 19, to: 23)  // Did not expect alternative identifier.
```

First, start with unit uCallable. Add uAst, uToken to the uses clause. Then apply the following change to the TCallArg:

```
type
  TCallArg = class
    Value: Variant;
    Ident: TIdent;
    Token: TToken;
    constructor Create(AValue: Variant; AIdent: TIdent; AToken: TToken);
  end;
```

```
constructor TCallArg.Create(AValue: Variant; AIdent: TIdent; AToken: TToken);
begin
  Value := AValue;
  Ident := AIdent;
  Token := AToken;
end;
```

Then, add the following private helper function to class TFunc in unit uFunc. Also add unit uToken to the uses clause of unit uFunc.pas.

```
procedure TFunc.CheckIdents(Token: TToken; IdentCalled, IdentDefined: TIdent);
begin
  if (IdentCalled = Nil) and (IdentDefined = Nil) then Exit;
  if IdentDefined = Nil then
    Raise ERuntimeError.Create(Token, Format(
      ErrExpectedNoExtId, [IdentCalled.Text]));
  if IdentCalled = Nil then
    Raise ERuntimeError.Create(Token, Format(
      ErrExpectedExtId, [IdentDefined.Text]));
  if IdentCalled.Text <> IdentDefined.Text then
    Raise ERuntimeError.Create(Token, Format(
      ErrExtIdMismatch, [IdentDefined.Text]));
end;
```

I am aware of the extra burden on the interpreter but there's no alternative way of doing it without a proper symbol table.

Add the following error messages as constants to the implementation section of uFunc.pas:

```
implementation

const
  ErrInvalidNumberOfArgs = 'Invalid number of arguments. Expected %d arguments.';
  ErrExpectedNoExtId = 'Did not expect external identifier "%s:".';
  ErrExpectedExtId = 'Expected external identifier: "%s:".';
  ErrExtIdMismatch = 'External identifier mismatch: expected "%s:".';
```

Next, a small change to the function Call of TFunc.

```
function TFunc.Call(Interpreter: TInterpreter; ArgList: TArgList): TObject;
...
  for i := 0 to FuncDecl.Params.Count-1 do begin
    CheckIdents(ArgList[i].Token, ArgList[i].Ident, FFuncDecl.Params[i].ExtIdent);
    FuncSpace.Store(FFuncDecl.Params[i].Ident, ArgList[i].Value);
  end;
...
end;
```

We check the alternative identifiers by paasing them as arguments to the procedure CheckIdents.

Finally, in the interpreter change this code in VisitCallExpr.

```
function TInterpreter.VisitCallExpr(CallExpr: TCallExpr): Variant;
var
...
  Token: TToken;
begin
  ...
  Args := TArgList.Create();
  for i := 0 to Node.Args.Count-1 do begin
    if Assigned(CallExpr.Args[i].Ident) then
      Token := CallExpr.Args[i].Ident.Token
    else
      Token := CallExpr.Args[i].Expr.Token;
    CallArg := TCallArg.Create(Visit(CallExpr.Args[i].Expr),
      CallExpr.Args[i].Ident, Token);
    Args.Add(CallArg);
  end;
  Result := Func.Call(CallExpr.Token, Self, Args);
end;
```

Type TCallArg is defined in unit uCallable.pas. Create a new CallArg by calling its constructor. First, determine the token to save. We need this for error location information. If there's no external ident used in the call, the ident is Nil, in which case we save the token from the expression. This makes sure the error info always points to the right location. Then, we visit the expression and pass its resulting value into the CallArg. Lastly, we add the identifier itself, even if it is Nil.

As shown above, in the call to a function we perform a check on the parameters and their identifiers. Examples:

```
func add(x, to y) => x+y
var a := add(19,23)

@[2,17] Runtime error: Expected alternative identifier: "to:".
```

```
func add(x, to y) => x+y
var a := add(19, ot: 23)

@[2,18] Runtime error: Alternative identifiers mismatch: expected "to:".
```

```
func add(x, to y) => x+y
var a := add(from: 19, to: 23)

@[2,14] Runtime error: Did not expect alternative identifier "from:".
```

Functions get nested

> Earth is the nest, the cradle, and we'll move out of it.
> — Gene Roddenberry

A nested function is a function which is defined within another function, the enclosing function. Due to simple recursive scope rules, a nested function is itself invisible outside of its immediately enclosing function, but can see (access) all local objects (data, functions, types, etc.) of its immediately enclosing function as well as of any function(s) which, in turn, encloses that function. The nesting is theoretically possible to unlimited depth, although only a few levels are normally used in practical programs (WikiPedia).

As we've seen, Free Pascal supports nested functions and procedures. We'll offer it too in the Gear language. For this we introduce the term closure. A closure is a technique for implementing lexically scoped name binding in languages with first-class functions. Operationally, a closure stores a function together with an environment. A closure—unlike a plain function—allows the function to access those captured variables through the closure's copies of their values or references, even when the function is invoked outside their scope.

Let's move to it. In unit uFunc.pas change the class TFunc so that it includes a private field Closure, which is initialised in the constructor Create.

```
  TFunc = class(TInterfacedObject, ICallable)
    private
      ...
      FClosure: TMemorySpace;
      ...
    public
      ...
      constructor Create(AFuncDecl: TFuncDecl; AClosure: TMemorySpace);
      ...
  end;
```

```
constructor TFunc.Create(AFuncDecl: TFuncDecl; AClosure: TMemorySpace);
begin
  FFuncDecl := AFuncDecl;
  FClosure := AClosure;
end;
```

Then, in function Call, replace in the TMemorySpace.Create the passed Interpreter.Globals variable by Closure. You'll get:

```
function TFunc.Call(Interpreter: TInterpreter; ArgList: TArgList): Variant;
...
begin
  ...
  FuncSpace := TMemorySpace.Create(Closure);
  ...
end;
```

Finally, change the interpreter visitor for FuncDecl, so that the current memory space becomes the closure, when creating a TFunc:

```
procedure TInterpreter.VisitFuncDecl(Node: TFuncDecl);
...
begin
  ...
  Func := TFunc.Create(FuncDecl, CurrentSpace);
  ...
end;
```

This is it! It should work. Try it out!

A few examples:

```
func makeFunc()
  var name := 'Closure'
  func displayName()
    print(name)
  end
  return displayName
end

var runFunc := makeFunc()
runFunc() // prints Closure
```

Another one:

```
func startAt(x)
  func incrementBy(y)
    return x + y
  end
  return incrementBy
end

var closure1 := startAt(1)
var closure2 := startAt(5)

print(closure1(3))   // 4
print(closure2(3))   // 8
print(startAt(7)(9)) // 16
```

Or what about this one:

```
func makePoint(x, y)
  func closure(method)
    return match method
      if 'x' then x
      if 'y' then y
      else 'unknown method ' + method
  end

  return closure
end

var point := makePoint(2, 3)
print(point('x')) // 2
print(point('y')) // 3
print(point('z')) // unknown method z
```

Returning one expression

> There is nothing like returning to a place that remains unchanged to find the ways in which you yourself have altered.
> — Nelson Mandela

Besides the function declaration description in EBNF we have worked with so far, I would like to introduce a way that makes it easier to return values if there's only one expression to be returned. Let's recap the original EBNF of a function declaration.

```
  FuncDecl = 'func' Ident '(' [ Parameters ] ')' Block 'end' .
```

As an example, consider the function add:

```
func add(a,b)
  return a+b
end
```

In such cases it will be allowed to declare a function like this:

```
func add(a,b) => a+b
```

The EBNF changes because of this to the following:

```
  FuncDecl = 'func' Ident '(' [ Parameters ] ')' ('=>' Expr | Block 'end') .
```

The only thing we have to change for this is the parsing functionality. The AST node and visitors remain the same. Follow the EBNF and you'll get (only the changes are shown):

```
function TParser.ParseFuncDecl: TDecl;
  ...
  Expect(ttCloseParen);
  if CurrentToken.Typ = ttArrow then begin
    FuncDecl.Body := TBlock.Create(TNodeList.Create(), CurrentToken);
    FuncDecl.Body.Nodes.Add(ParseReturnStmt);
  end
  else begin
    FuncDecl.Body := ParseBlock;
    Expect(ttEnd);
  end;
  Result := FuncDecl;
end;
```

if a '=>' (ttArrow) token is found, we initialize the block of the function declaration with a new TNodeList and the CurrentToken. We then add a TReturnStatement via ParseReturnStmt to the Body's nodes. ParseReturnStmt also consumes the '=>' token. If no '=>' token, then the usual declaration follows.

Testing with this nice program gives four times the result 42.

```
func add(a,b) => a+b
func sub(a,b) => a-b
func mul(a,b) => a*b
func div(a,b) => a/b

print(add(35,7))   // 42
print(sub(50,8))   // 42
print(mul(21,2))   // 42
print(div(168,4))  // 42
```

```
func fact(n) =>
  match n
    if 0 then 0
    if 1 then 1
    else n*fact(n-1)

print(fact(10))
```

Call them anonymous

> Coincidence is God's way of remaining anonymous.
> — Albert Einstein

An anonymous function (function literal, lambda abstraction, or lambda expression) is a function definition that is not bound to an identifier. Anonymous functions are often:

- arguments being passed to higher-order functions, or
- used for constructing the result of a higher-order function that needs to return a function.

(WikiPedia)

Function as expression

We use function declaration expression (FuncDeclExpr) hereafter. The definition in EBNF of a FuncDeclExpr expression is based on the function declaration, which was:

```
FuncDecl = ‘func’ Ident ‘(‘ [ Parameters ] ‘)’ (‘=>’ Expr | Block ‘end’) .
```

A function declaration, apart from the keyword func and the function identifier, is basically a group of parameters and an execution block. To make this look good in real code, consider the following examples.

```
func times10(f)
  for var i:=1 where i<=10, i+=1 do
    print(f(i))
  end
end

times10( func(x) return x^2 end )
times10( func(x) => x^2 )
```

The FuncDeclExpr expression becomes (note it lacks the Ident!):

```
FuncDeclExpr = ‘func’ ‘(‘ [ Parameters ] ‘)’ (‘=>’ Expr | Block ‘end’) .
```

Besides all this, the fact that it is called an expression means it has to be parsed as a sort of inline function ‘as an expression’. First, let’s define the new AST node:

```
  TFuncDeclExpr = class(TFactorExpr)
    private
      FFuncDecl: TFuncDecl;
    public
      property FuncDecl: TFuncDecl read FFuncDecl;
      constructor Create(AFuncDecl: TFuncDecl);
      destructor Destroy; override;
  end;
```

Define it right under the definition of TFuncDecl. The implementation is:

```
constructor TFuncDeclExpr.Create(AFuncDecl: TFuncDecl);
begin
  inherited Create(AFuncDecl.Token);
  FFuncDecl := AFuncDecl;
end;
```

```
destructor TFuncDeclExpr.Destroy;
begin
  if Assigned(FFuncDecl) then FFuncDecl.Free;
  inherited Destroy;
end;
```

In the parser we need to make a few small changes. To start with change the function ParseFactor to include the case for ttFunc, like this

```
function TParser.ParseFactor: TExpr;
begin
  case CurrentToken.Typ of
    ttFunc: Result := TFuncDeclExpr.Create(ParseFuncDecl(ffAnonym) as TFuncDecl);
    ttIf: Result := ParseIfExpr;
  ...
  end;
end;
```

We call function ParseFuncDecl with the value 'ffAnonym'. Together with 'ffFunction', these are the possible function forms. Later we'll extend it further. In the class declaration of TParser change the private type:

```
  TParser = class
    private
      type
        TFuncForm = (ffFunction, ffAnonym);
      ...
  end;
```

Add the case for ttAnonym to the case statement in ParseFuncDecl:

```
function TParser.ParseFuncDecl(FuncForm: TFuncForm): TDecl;
var
  ...
begin
  Token := CurrentToken;
  case FuncForm of
    ffNormal: begin Next; Name := ParseIdent; end;
    ffAnonym: Next;
  end;
...
end;
```

In the above call to ParseFuncDecl in ParseFactor we set FuncForm to ffAnonym, which results in the function Name to be Nil. If it is a normal function, which is the default, the function identifier is parsed as normal. The rest stays the same.

Then add the visitor for FuncDeclExpr in the printer:

```
procedure TPrinter.VisitFuncDeclExpr(FuncDeclExpr: TFuncDeclExpr);
begin
  IncIndent;
  VisitNode(FuncDeclExpr);
  Visit(FuncDeclExpr.FuncDecl);
  DecIndent;
end;
```

And the visitor for the Resolver:

```
procedure TResolver.VisitFuncDeclExpr(FuncDeclExpr: TFuncDeclExpr);
begin
  Visit(FuncDeclExpr.FuncDecl);
end;
```

Also, change the Resolver visitor for VisitFuncDecl() so that it can handle function ID's that are Nil:

```
procedure TResolver.VisitFuncDecl(FuncDecl: TFuncDecl);
begin
  if FuncDecl.Ident <> Nil then begin
    Declare(FuncDecl.Ident);
    Enable(FuncDecl.Ident);
  end;
  ResolveFunction(FuncDecl, fkFunc);
end;
```

Now there's only one thing left to do and that's the implementation of the visitor for the interpreter. You'll notice the simplicity of the solution, and I could hardly believe it myself when I designed it, but it really works.

Since it is an expression, the function returns a value. The value in fact is a function!

```
function TInterpreter.VisitFuncDeclExpr(FuncDeclExpr: TFuncDeclExpr): Variant;
begin
  Result := ICallable(TFunc.Create(FuncDeclExpr.FuncDecl, CurrentSpace));
end;
```

These are all the changes needed to make this to work. Try it out, for example with this code:

```
func calc(times n, f)
  for var i:=1 where i<=n, i+=1 do
    print(f(i))
  end
end

calc(times: 5, func(x) return x^2 end )  // 1 4 9 16 25
```

Alternatively, and shorter, you can call it like this, if only one expression is returned:

```
calc(times: 5, func(x) => x^2)  // 1 4 9 16 25
calc(times: 5, func(x) => x^3)  // 1 8 27 64 125
calc(times: 5, func(x) => x^4)  // 1 16 81 256 625
```

Another use, that immediately comes available is declaring a variable with a function value:

```
var xs := func(x) => x^2

print(xs(16))  // 256

var hello := func()
  print('Hello world!')
end

hello()  // Hello World!
```

And with multiple parameters:

```
func calc(times n, f)
  for var i:=1 where i<=n, i+=1 do
    print(f(i,n))
  end
end

calc(times: 5, func (x,y) => x*y )  // 5 10 15 20 25
```

Even this is allowed:

```
var x := func ()
  print('Hello')
end()   // prints immediately Hello

var y := func (a)
  return a*a*a
end(7)

print(y) // 343

print(func (a) return a*a*a end(8))  // 512


var z := (func(x)=>2^x)(9)  // parentheses are required here!
print(z)  // 512
```

Lambda functions and currying

Finally, let's see if we can make things a bit simpler by removing the need for the 'func' keyword. Have a look at the following series of calculations:

```
func times10(f)
  for var i:=1 where i<=10, i+=1 do
    print(f(i))
  end
end

times10(func(x) return x^2 end)
times10(func(x) => x^2)
times10(x=>x^2)
```

They are all the same. The last one not only is more concise, but also more readable.

Lambda calculus uses functions of 1 input. An ordinary function that requires two inputs, for instance the addition function x+y can be altered in such a way that it accepts 1 input and as output creates another function, that in turn accepts a single input. As an example, consider:

```
func(x,y)=> x+y, which can be rewritten as x=>y=>x+y
```

This method, known as currying, transforms a function that takes multiple arguments into a chain of functions each with a single argument. See example:

```
var add := x=>y=>x+y
print(add(7)(9))
```

In function ParseFactor change the following line:

```
  ttIdentifier: Result := TVarExpr.Create(ParseIdent);
```

into:

```
  ttIdentifier: Result := ParseIdentifierExpr;
```

Add this private function just above ParseFactor:

```
function TParser.ParseIdentifierExpr: TExpr;
var
  Ident: TIdent;
  FuncDecl: TFuncDecl;
begin
  Ident := ParseIdent;
  if CurrentToken.Typ = ttArrow then begin
    FuncDecl := TFuncDecl.Create(Nil, CurrentToken);
    FuncDecl.AddParam(Ident, Nil);
    FuncDecl.Body := TBlock.Create(TNodeList.Create(), CurrentToken);
    FuncDecl.Body.Nodes.Add(ParseReturnStmt);
    Result := TFuncDeclExpr.Create(FuncDecl);
  end
  else Result := TVariable.Create(Ident);
end;
```

First, an ident is parsed. If the current token is an arrow '=>' we are actually defining an anonymous function or lambda expression with 1 parameter. We create a function declaration and return that as an expression.

Otherwise, the ident is a regular variable expression.

However, we wish to do this for multiple parameters as well. How nice would it be to write something like:

```
var add := (x,y) => x+y
print(add(7,9))
```

However, we already parse parenthesized expressions in ParseFactor:

```
    ttOpenParen: begin
      Next; // skip '('
      Result := ParseExpr;
      Expect(ttCloseParen);
    end;
```

We need to add more functionality here. Change this to:

```
ttOpenParen: Result := ParseParenExpr;
```

Add the following private function:

```
function TParser.ParseParenExpr: TExpr;
type TIdentList = specialize TArrayObj<TIdent>;
var
  Ident: TIdent;
  IdentList: TIdentList;
  FuncDecl: TFuncDecl;
begin
  Expect(ttOpenParen);
  if Peek.Typ = ttComma then begin  // check next token
    // we have a parameter list
    IdentList := TIdentList.Create(false);
    IdentList.Add(ParseIdent);
    while CurrentToken.Typ = ttComma do begin
      Next; // skip ,
      IdentList.Add(ParseIdent);
    end;
    FuncDecl := TFuncDecl.Create(Nil, CurrentToken);
    for Ident in IdentList do
      FuncDecl.AddParam(Ident, Nil);
    Expect(ttCloseParen);
    if CurrentToken.Typ <> ttArrow then
      Errors.Append(CurrentToken.Line, CurrentToken.Col, ErrExpectedArrow);
    FuncDecl.Body := TBlock.Create(TNodeList.Create(), CurrentToken);
    FuncDecl.Body.Nodes.Add(ParseReturnStmt);
    Result := TFuncDeclExpr.Create(FuncDecl);
  end
  else if (CurrentToken.Typ = ttCloseParen) and
          (Peek.Typ = ttArrow) then begin // no parameters
    FuncDecl := TFuncDecl.Create(Nil, CurrentToken);
    Expect(ttCloseParen);
    FuncDecl.Body := TBlock.Create(TNodeList.Create(), CurrentToken);
    FuncDecl.Body.Nodes.Add(ParseReturnStmt);
    Result := TFuncDeclExpr.Create(FuncDecl);
  end
  else begin
    Result := ParseExpr;
    Expect(ttCloseParen);
  end;
end;
```

Add unit uCollections to the uses clause in the implementation:

```
implementation
uses uCollections;
```

and add a new error message:

```
  ErrExpectedArrow = 'Expected arrow "=>".';
```

More examples. Here are the examples from the nested function paragraph but now as anonymous function.

```
func startAt(x) => y => x + y

var closure1 := startAt(1)
var closure2 := startAt(5)

print(closure1(3))    // 4
print(closure2(3))    // 8
print(startAt(7)(9))  // 16



func calc(times n, f)
  for var i:=1 where i<=n, i+=1 do
    print(f(i,n))
  end
end

calc(times: 5, func(x,y) => x*y )  // 5 10 15 20 25
calc(times: 5, (x,y) => x*y)

calc(times: 5, func(x,y) => x/y )  // 0.2 0.4 0.6 0.8 1
calc(times: 5, (x,y) => x/y)


var s := () => 'Hello world!'
print(s())
```

Chapter 6 – Classes

In object-oriented programming, a class is an extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions or methods). In many languages, the class name is used as the name for the class (the template itself), the name for the default constructor of the class (a subroutine that creates objects). When an object is created by a constructor of the class, the resulting object is called an instance of the class, and the member variables specific to the object are called instance variables, to contrast with the class variables shared across the class.

Class declaration

So, a class has a name, some fields and methods, and at least a constructor. The EBNF for a class declaration is:

> I paint objects as I think them, not as I see them.
> — Pablo Picasso

```
ClassDecl = 'class' Ident { Decl } 'end'  .
```

Add the class declaration just above the TBlock declaration.

```
  TDeclList = specialize TArrayObj<TDecl>;

  TClassDecl = class(TDecl)
    private
      FDeclList: TDeclList;
    public
      property DeclList: TDeclList read FDeclList;
      constructor Create(AIdent: TIdent; ADeclList: TDeclList; AToken: TToken);
      destructor Destroy; override;
  end;


constructor TClassDecl.Create
  (AIdent: TIdent; ADeclList: TDeclList; AToken: TToken);
begin
  Inherited Create(AIdent, AToken);
  FDeclList := ADeclList;
end;

destructor TClassDecl.Destroy;
begin
  if Assigned(FDeclList) then FDeclList.Free;
  inherited Destroy;
end;
```

A class declaration can be done like this:

```
class Car
  var name := 'Car'
  func show()
    return self.name
  end
  init(name)
    self.name := name
  end
end

var car := Car('Volvo')
print(car.name)   // “Volvo”
print(car.show()) // “Volvo”
```

I like it when all fields are known at compile time, as it becomes very clear to the programmer what’s in the class and saves him/her from errors in this respect. However, we start from a dynamic viewpoint, where it is possible to add fields on the fly. If you want to stop there… it’s ok, but I’ll also show the more strict version.

Parsing a class declaration according the EBNF is not too difficult:

```
function TParser.ParseClassDecl: TDecl;
var
  Ident: TIdent;
  DeclList: TDeclList;
  Token: TToken;
begin
  Token := CurrentToken;
  Next;  // skip class
  Ident := ParseIdent;
  DeclList := TDeclList.Create(false);
  while CurrentToken.Typ in DeclStartSet do
    DeclList.Add(ParseDecl);
  Result := TClassDecl.Create(Ident, DeclList, Token);
  Expect(ttEnd);
end;
```

Just like the EBNF tells us, after the keyword ‘class’, we parse the class identifier. Then, while we find declarations we parse them and add them to the Declaration list. Lastly, we expect the ‘end’ keyword.

Furthermore, in the parser, add the token to the declaration’s start set:

```
const
  DeclStartSet: TTokenTypSet = [ttClass, ttFunc, ttLet, ttVar];
```

Don’t forget to add the case for ttClass to the function ParseDecl:

```
function TParser.ParseDecl: TDecl;
const Mutable = True;
begin
  case CurrentToken.Typ of
    ttClass: Result := ParseClassDecl;
    ttFunc: Result := ParseFuncDecl(ffFunction);
    ttLet: Result := ParseVarDecl(Mutable = False);
    ttVar: Result := ParseVarDecl(Mutable = True);
  end;
end;
```

The visitor for the printer is straightforward:

```
procedure TPrinter.VisitClassDecl(ClassDecl: TClassDecl);
var
  Decl: TDecl;
begin
  IncIndent;
  VisitNode(ClassDecl);
  Visit(ClassDecl.Ident);
  for Decl in ClassDecl.DeclList do
    Visit(Decl);
  DecIndent;
end;
```

In the resolver we just simply declare and define the new class declaration (as a first step):

```
procedure TResolver.VisitClassDecl(ClassDecl: TClassDecl);
begin
  Declare(ClassDecl.Ident);
  Enable(ClassDecl.Ident);
end;
```

The first version of the interpreter visitor is:

```
procedure TInterpreter.VisitClassDecl(ClassDecl: TClassDecl);
var
  GearClass: TGearClass;
begin
  CheckDuplicate(ClassDecl.Ident, 'Class');
  CurrentSpace.Store(ClassDecl.Ident, Nil);
  GearClass := TGearClass.Create(ClassDecl.Ident);
  CurrentSpace.Update(ClassDecl.Ident, IClassable(GearClass));
end;
```

First, store the class identifier with a Nil value. Next, the class is created, followed by updating the current space. Note that GearClass is cast to interface IClassable, which is inheriting from ICallable. This is because we can instantiate a class by calling it like a function. It's discussed in more detail later.

Create a new unit uClassIntf:

```
unit uClassIntf;
{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, uCallable, uAST;

type

  IClassable = interface(ICallable)
    ['{230E841B-D57B-F5F8-4DD3-224B74645A17}']
  end;

implementation

end.
```

A class that implements IClassable must also implement ICallable. The implementation of ICallable requires that functions Call and toString are implemented.

Now, let's look at the TGearClass. Create a new unit uClass.pas.

```
unit uClass;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, uAST, uCallable, uClassIntf, uInterpreter, uToken;

type

  TGearClass = class(TInterfacedObject, IClassable)
    private
      Ident: TIdent;
    public
      constructor Create(AIdent: TIdent);
      function Call(Token: TToken; Interpreter: TInterpreter;
        ArgList: TArgList): Variant;
      function toString: String; override;
  end;

implementation

{ TGearClass }

constructor TGearClass.Create(AIdent: TIdent);
begin
  Ident := AIdent;
end;

function TGearClass.Call(Token: TToken; Interpreter: TInterpreter;
  ArgList: TArgList): Variant;
begin
  Result := Null;
end;

function TGearClass.toString: String;
begin
  Result := Ident.Text;
end;

end.
```

In the interpreter make sure to adjust the uses clause in the implementation to the following:

```
implementation
uses uCallable, uFunc, uStandard, uClassIntf, uClass;
```

It doesn't do a lot yet. The only thing you can do right now is print the value of an instantiated class. Like this:

```
class Home
  var type := 'none'
  func toString()
```

```
    return type
  end
end

var home := Home()
print(home) // prints Null
```

It prints Null, since that's what is returned in the TGearClass.Call() method. However, in this case we would prefer that it would've printed 'Home instance' for example.

Also, printing the class itself results in an error, instead of printing the class name:

```
print(Home) // error, invalid variant type cast
```

That's not quite possible now, since the IClassable interface doesn't know how to get printed yet. We need a little adjustment to how variants get printed.

Create a new unit that adds some support for Variants.

```
unit uVariantSupport;

{$mode objfpc}{$H+}
{$modeswitch typehelpers}

interface

uses
  Classes, SysUtils, Variants;

type

  TVariantHelper = type helper for Variant
    function toString: string;
  end;

function VarSupportsIntf(Value: Variant; Intf: array of TGUID): Boolean;

implementation

function TVariantHelper.toString: string;
begin
  if VarSupports(Self, IUnknown) then
    Result := (IUnknown(Self) as TInterfacedObject).toString
  else
    Result := VarToStrDef(Self, 'Null');
end;

function VarSupportsIntf(Value: Variant; Intf: array of TGUID): Boolean;
var
  i: Integer;
begin
  Result := False;
  for i := 0 to High(Intf) do
    if VarSupports(Value, Intf[i]) then
      Exit(True);
end;

end.
```

In short, add {$modeswitch typehelpers} above the interface keyword. Then, add unit Variants to the uses clause. Next, add a new type extension for type Variant: TVariantHelper with only one function: toString. IUnknown is a base interface that all other interfaces inherit from, so if we use that here, all printing from variant values is taken care of.

In the implementation of this function we use standard Free Pascal function VarSupports to check whether the variant itself supports interface ICallable or IClassable, and if so the result will be ICallable(Self).toString or IClassable(Self).toString.

In the interpreter change visitor VisitCallExpr so that it uses VarSupportsIntf. All descend interfaces of ICallable will be coded there.

```
function TInterpreter.VisitCallExpr(CallExpr: TCallExpr): Variant;
...
begin
  Callee := Visit(CallExpr.Callee);
  if VarSupportsIntf(Callee, [ICallable, IClassable]) then
  ...
end;
```

Finally, in the interpreter, in method VisitPrintStmt change the line:

```
    Write(VarToStrDef(Value, 'Null'));
```

To:

```
    Write(Value.toString);
```

So that it looks like this:

```
procedure TInterpreter.VisitPrintStmt(PrintStmt: TPrintStmt);
...
begin
  for i := 0 to PrintStmt.ExprList.Count-1 do begin
    Value := (Visit(PrintStmt.ExprList[i])).toString;
    ...
  end;
  Terminator := (Visit(PrintStmt.Terminator)).toString;
    ...
end;
```

Latest, add uVariantSupport to the uses clause in the implementation:

```
implementation
uses uCallable, uFunc, uStandard, uClassIntf, uClass, uVariantSupport;
```

With this in place printing the class, now prints its name!

Class Instances

> There is no instance of a nation benefitting from prolonged warfare.
> — Sun Tzu

In class-based programming, objects are created from classes by subroutines called constructors, and destroyed by destructors. An object is an instance of a class, and may be called a class instance or class object; instantiation is then also known as construction (WikiPedia).

Goal in this part is to create an instance of a class by using the following structure:

```
var instance := Class() // so far this returned Null
```

Notice that it looks like a function call. In fact we'll use what we designed in the functions: ICallable. In order to use that, TGearClass implements IClassable, which descends from ICallable.

Let's work out function Call() in TGearClass. Replace "Result := Null;":

```
function TGearClass.Call(Token: TToken; Interpreter: TInterpreter;
  ArgList: TArgList): Variant;
var
  Instance: IGearInstance;
begin
  Instance := IGearInstance(TGearInstance.Create(Self));  // cast to IGearInstance
  Result := Instance;
end;
```

In function Call an instance of a TGearClass is created: TGearInstance, and returned as the function result. Note that when you call a class, an instance is returned! Also note that the result of the call is not so much the instance, but the Instance as IGearInstance, yet another interface. Create this interface in unit uClassIntf:

```
  IGearInstance = Interface
    ['{6E9625B1-8DAD-EC3B-57E5-80D360CC7B67}']
    function toString: String;
  end;
```

Next, we define class TGearInstance in uClass, which is the runtime representation of a TGearClass.

```
  TGearInstance = class(TInterfacedObject, IGearInstance)
    private
      GearClass: TGearClass;
    public
      constructor Create(AGearClass: TGearClass);
      function toString: String; override;
  end;
```

And its implementation:

```
{ TGearInstance }

constructor TGearInstance.Create(AGearClass: TGearClass);
begin
  GearClass := AGearClass;
end;
```

```
function TGearInstance.toString: String;
begin
  Result := GearClass.Ident.Text + ' instance';
end;
```

The constructor accepts a parameter of type TGearClass, which creates the binding between the instance and the class.

Furthermore, there's a toString function that returns the name and the fact that it's an instance.

Try and run the code from the previous paragraph.

It now nicely prints the class and instance names.

Class Instance Fields

> The proper method for inquiring after the properties of things is to deduce them from experiments.
>
> — Isaac Newton

A class contains data field descriptions (or properties, fields, data members, or attributes). These are usually field types and names that will be associated with state variables at program run time; these state variables either belong to the class or specific instances of the class (WikiPedia).

In this paragraph we work on instance fields. In this first exercise the focus is on reading and dynamic adding of fields. This is usually done through getters and setters. See the example:

```
class Car

end

var car := Car()
car.name := 'Volvo'
print(car.name) // prints 'Volvo'
```

Notice the use of the '.' or dot syntax. In car.name, car is the object and name is the field/property. The EBNF of call expressions was:

```
  CallExpr   = Factor { '(' [ Arguments] ')' } .
```

But now with the dot operator it becomes:

```
  CallExpr = Factor { '(' [Arguments] ')' | '.' Ident } .
```

This allows for repeated property calls, like object1.object2.object3.field.

Let's start with the getter first. The AST node is:

```
  TGetExpr = class(TExpr)
    private
      FInstance: TExpr;
      FIdent: TIdent;
    public
      property Instance: TExpr read FInstance;
      property Ident: TIdent read FIdent;
      constructor Create(AInstance: TExpr; AIdent: TIdent);
      destructor Destroy; override;
  end;


constructor TGetExpr.Create(AInstance: TExpr; AIdent: TIdent);
begin
  Inherited Create(AIdent.Token);
  FInstance := AInstance;
  FIdent := AIdent;
end;

destructor TGetExpr.Destroy;
begin
  if Assigned(FInstance) then FInstance.Free;
  if Assigned(FIdent) then FIdent.Free;
  inherited Destroy;
end;
```

Following the new EBNF for CallExpr, the parser function changes to:

```
function TParser.ParseCallExpr: TExpr;
begin
  Result := ParseFactor;
  while CurrentToken.Typ in [ttDot, ttOpenParen] do
    case CurrentToken.Typ of
      ttOpenParen: Result := ParseCallArgs(Result);
      ttDot: begin
        Next; // skip '.'
        Result := TGetExpr.Create(Result, ParseIdent);
      end;
    end;
end;
```

The visitor for the printer is:

```
procedure TPrinter.VisitGetExpr(GetExpr: TGetExpr);
begin
  IncIndent;
  VisitNode(GetExpr);
  Visit(GetExpr.Instance);
  Visit(GetExpr.Ident);
  DecIndent;
end;
```

The visitor for the resolver is:

```
procedure TResolver.VisitGetExpr(GetExpr: TGetExpr);
begin
  Visit(GetExpr.Instance);
end;
```

The visitor for the interpreter is more complicated:

```
  ErrExpectedClassInstance = 'Expected class instance.';
```

```
function TInterpreter.VisitGetExpr(GetExpr: TGetExpr): Variant;
var
  Instance: Variant;
begin
  Instance := Visit(GetExpr.Instance);
  if VarSupports(Instance, IGearInstance) then
    Result := IGearInstance(Instance).GetMember(GetExpr.Ident)
  else
    Raise ERuntimeError.Create(GetExpr.Ident.Token, ErrExpectedClassInstance);
end;
```

First, the expression that should be a class instance is visited. If it is a class instance, the field's value is retrieved. If it isn't a class instance, an error is generated. Function GetMember in TGearInstance is not yet defined, let's take care of that.

Let's dive into the world of class members.

Every field, method, or constant value declared inside a class is called a member of that class. But this will later on also apply (partly) to enums, arrays and dictionaries, so it's better to create separate functionality for this. So, create a new class uMembers.pas:

```
unit uMembers;

{$mode objfpc}{$H+}
{$modeswitch typehelpers}
{$modeswitch advancedrecords}

interface

uses
  Classes, SysUtils, uCollections, variants;

type

  TMemberTable = specialize TDictionary<String, Variant>;
  TFieldTable = TMemberTable;
  TConstTable = TMemberTable;
  TMethodTable = TMemberTable;

  TMembers = record
    Fields: TFieldTable;
    Constants: TConstTable;
    Methods: TMethodTable;
    constructor Create(AFields: TFieldTable; AConstants: TConstTable;
      AMethods: TMethodTable);
    procedure Init;
  end;

implementation

{ TMembers }

constructor TMembers.Create
  (AFields: TFieldTable; AConstants: TConstTable; AMethods: TMethodTable);
begin
  Fields := AFields;
  Constants := AConstants;
  Methods := AMethods;
end;

function TMembers.Init: TMembers;
begin
  Self.Fields := TFieldTable.Create;
  Self.Constants := TConstTable.Create;
  Self.Methods := TMethodTable.Create;
end;

end.
```

Later we'll also add calculated properties.

A variable of type TFieldTable holds the fields and their values.

In uClass.pas add unit uMembers to the uses class at the top of the unit, so that it looks like this:

```
uses
  Classes, SysUtils, uAST, uCallable, uClassIntf, uInterpreter, uToken, uError,
  uMembers;
```

Then, in the interface IGearInstance add the following function:

```
  IGearInstance = Interface
    ...
    function GetMember(Ident: TIdent): Variant;
  end;
```

This must be implemented in class TGearInstance. Also add the variable Fields:

```
  TGearInstance = class(TInterfacedObject, IGearInstance)
    private
      ...
      InstanceFields: TFieldTable;
    public
      ...
      function GetMember(Ident: TIdent): Variant;
  end;
```

We add the creation of a Fields list to the Create constructor:

```
constructor TGearInstance.Create(AGearClass: TGearClass);
begin
  GearClass := AGearClass;
  InstanceFields := TFieldTable.Create();
  InstanceFields.Sorted := True;
  InstanceFields['className'] := GearClass.Ident.Text;
end;
```

Make sure the FieldTable is sorted for quick search. I added a standard field in the table, className, which holds the name of the class the instance belongs to.
And implement function GetMember:

```
function TGearInstance.GetMember(Ident: TIdent): Variant;
var
  i: LongInt = -1;
begin
  if InstanceFields.Contains(Ident.Text, i) then
    Exit(InstanceFields.At(i));

  Raise ERuntimeError.Create(Ident.Token,
    'Undefined class member "' + Ident.Text + '".');
end;
```

If we want to read the value of a field, it must exist, so we first try to find its index. If found we return the value for this index. If not found, an error is raised.

It doesn't do much yet, as there is only one field in the class. New fields will be added by the setter.
In a setter an expression or value is assigned to a field, like in this example:

```
car.name := 'Volvo'
```

This means we have to make changes to assignments. First, the 'old' EBNF:

```
AssignStmt = Identifier ( '(' Arguments ')' |  ':=' Expression ) .
```

Now with the dotted syntax added:

```
AssignStmt = Identifier [ '.' Identifier ] ( '(' Arguments ')' |  ':=' Expression ) .
```

The Setter thus is a statement, and its AST node becomes:

```
  TSetStmt = class(TStmt)
    private
      FInstance: TExpr;
      FIdent: TIdent;
      FExpr: TExpr;
    public
      property Instance: TExpr read FInstance;
      property Ident: TIdent read FIdent;
      property Expr: TExpr read FExpr;
      constructor Create(AInstance: TExpr; AIdent: TIdent; AExpr: TExpr);
      destructor Destroy; override;
  end;
```

It looks like a Getter, but we added the expression that is assigned to the dotted variable.

```
constructor TSetStmt.Create(AInstance: TExpr; AIdent: TIdent; AExpr: TExpr);
begin
  Inherited Create(AIdent.Token);
  FInstance := AInstance;
  FIdent := AIdent;
  FExpr := AExpr;
end;

destructor TSetStmt.Destroy;
begin
  if Assigned(FInstance) then FInstance.Free;
  if Assigned(FIdent) then FIdent.Free;
  if Assigned(FExpr) then FExpr.Free;
  inherited Destroy;
end;
```

The visitor for the printer:

```
procedure TPrinter.VisitSetStmt(SetStmt: TSetStmt);
begin
  IncIndent;
  VisitNode(SetStmt);
  Visit(SetStmt.Instance);
  Visit(SetStmt.Ident);
  Visit(SetStmt.Expr);
  DecIndent;
end;
```

And the visitor for the resolver:

```
procedure TResolver.VisitSetStmt(SetStmt: TSetStmt);
begin
  Visit(SetStmt.Instance);
  Visit(SetStmt.Expr);
end;
```

For the parser function ParseAssignStmt, we first read the identifier part as an expression. Then based on the tokens that come after, we determine what type of expression it was.

Because of its complexity here's the complete ParseAssignStmt again:

```
function TParser.ParseAssignStmt: TStmt;
var
  Token, Op: TToken;
  Left, Right: TExpr;
begin
  Token := CurrentToken;
  Left := ParseExpr;
  if CurrentToken.Typ in AssignSet then begin
    Op := CurrentToken;
    Next; // skip assign token
    Right := ParseExpr;
    if Left is TVariable then
      Result := TAssignStmt.Create(Left as TVariable, Op, Right)
    else if Left is TGetExpr then
      with Left as TGetExpr do
        Result := TSetStmt.Create(Instance, Ident, Right)
    else
      Error(Token, ErrInvalidAssignTarget);
  end
  else if Left is TCallExpr then
    Result := TCallExprStmt.Create(Left as TCallExpr, Token)
  else
    Error(CurrentToken, ErrExpectedAssignOpFunc);
end;
```

First, the left side is parsed as an expression. If we encounter an assignment token, we move on and parse the right side as an expression. If the left side is a regular variable we create and return a normal assignment statement. If the left side is a getter expression, so with one or more dots, we create and return a setter statement!

If in first instance we didn't find an assignment token, it must be a call statement.

And finally, the interpreter visitor.

```
procedure TInterpreter.VisitSetStmt(SetStmt: TSetStmt);
var
  Instance, Value: Variant;
begin
  Instance := Visit(SetStmt.Instance);
  if not VarSupports(Instance, IGearInstance) then
    Raise ERuntimeError.Create(SetStmt.Token, ErrExpectedClassInstance);
  Value := Visit(SetStmt.Expr);
  IGearInstance(Instance).SetField(SetStmt.Ident, Value);
end;
```

First, the instance variable is visited and the result is stored in 'Instance'. We then check if Instance is a IGearInstance and if not raise an error stating that a class instance was expected.

If we passed that, the expression is visited, which returns a value. Next, since we now know that Instance is a IGearInstance, we typecast it and call the procedure SetField to store the field identifier and its value in the local Fields list.

We only need to create public procedure SetField in IGearInstance and implement it in TGearInstance.

```
  IGearInstance = Interface
    ...
    procedure SetField(Ident: TIdent; Value: Variant);
  end;
```

```
procedure TGearInstance.SetField(Ident: TIdent; Value: Variant);
begin
  InstanceFields[Ident.Text] := Value;
end;
```

And now this works:

```
class Car
end

var car := Car()
car.brand := 'Volvo'
car.type := 'V60'
print('Car brand: ', car.brand, ' and type: ', car.type)
print(car.className)
```

It prints:

'Car brand: Volvo and type: V60'

'Car'

Methods

> He who seeks for methods without having a definite problem in mind seeks in the most part in vain.
> — David Hilbert

We now have fields in our class system that can be created on the fly and then used, but other than printing them, we cannot do a lot with them. We need behavior, in the form of methods.

A method is nothing more than a function that operates on class instance fields. So for instance, an example class with a method could be:

```
class Volvo
  func printName()
    print('Volvo')
  end
end
var volvo := Volvo()
volvo.printName()
```

In orde to get basic methods to work we'll make a few changes. First, start with the resolver visitor VisitClassDecl. Call the new method ResolveClass:

```
procedure TResolver.VisitClassDecl(ClassDecl: TClassDecl);
begin
  Declare(ClassDecl.Ident);
  Enable(ClassDecl.Ident);
  ResolveClass(ClassDecl);
end;
```

Add the enum fkMethod to the type TFuncKind:

```
  TFuncKind = (fkNone, fkFunc, fkMethod);
```

Then, we will resolve all methods in the private method ResolveClass:

```
procedure TResolver.ResolveClass(ClassDecl: TClassDecl);
var
  Decl: TDecl;
  FuncKind: TFuncKind;
begin
  for Decl in ClassDecl.DeclList do
    if Decl is TFuncDecl then begin
      FuncKind := fkMethod;
      ResolveFunction(Decl as TFuncDecl, FuncKind);
    end;
end;
```

In the function ParseClassDecl, we allow more kinds of declaration to be added, such as VarDecl. For now we focus on methods only so hence the check: 'if Decl is TFuncDecl'. We then call ResolveFunction with the respective function kind.

In order to prepare for interpreting the methods, we'll use the uMembers type TMethodTable and TMembers.

This will hold the methods as ICallable interfaces in TGearClass.

Add the field as a private field to the fields in TGearClass:

```
  TGearClass = class(TInterfacedObject, IClassable)
    private
      Ident: TIdent;
      Methods: TMethodTable;
    public
      constructor Create(AIdent: TIdent; Members: TMembers);
      function Call(Token: TToken; Interpreter: TInterpreter;
        ArgList: TArgList): Variant;
      function toString: String; override;
  end;
```

The constructor of TGearClass becomes:

```
constructor TGearClass.Create(AIdent: TIdent; Members: TMembers);
begin
  Ident := AIdent;
  Methods := Members.Methods;
end;
```

Now we can update the visitor VisitClassDecl in the interpreter.

```
procedure TInterpreter.VisitClassDecl(ClassDecl: TClassDecl);
var
  GearClass: TGearClass;
  Decl: TDecl;
  FuncDecl: TFuncDecl;
  Func: TFunc;
  Members: TMembers;
begin
  Members.Init;
  CheckDuplicate(ClassDecl.Ident, 'Class');
  CurrentSpace.Store(ClassDecl.Ident, Nil);
  for Decl in ClassDecl.DeclList do begin
    if Decl is TFuncDecl then begin
      FuncDecl := Decl as TFuncDecl;
      Func := TFunc.Create(FuncDecl, CurrentSpace);
      Members.Methods[FuncDecl.Ident.Text] := ICallable(Func);
    end
  end;
  GearClass := TGearClass.Create(ClassDecl.Ident, Members);
  CurrentSpace.Update(ClassDecl.Ident, IClassable(GearClass));
end;
```

New is that we create a Methods table, and we walk through the list of declarations. If we encounter a function declaration, we create a TFunc and pass the current space as the closure. Then, we add the function under its identifier name to the methods table. When we now create a TGearClass, we pass in the members with the methods table.

The methods need to be accessed from the instance. We extend the GetMember function for that:

```
function TGearInstance.GetMember(Ident: TIdent): TObject;
var
  ...
  Method: ICallable;
begin
  ...

  Method := GearClass.FindMethod(Ident.Text);
  if Method <> Nil then
    Exit(Method);

  Raise ERuntimeError.Create(Ident.Token,
    'Undefined class member "' + Ident.Text + '".');
end;
```

If the getter is not a field, we try to find a method. If the method was found we return it as the result.

Public function FindMethod in TGearClass is as follows:

```
function TGearClass.FindMethod(const AName: String): ICallable;
var
  Index: integer = -1;
begin
  Result := Nil;
  if Methods.Contains(AName, Index) then
    Result := ICallable(Methods.At(Index));
end;
```

It simply looks up the index of the method name, and if found returns the accompanying method instance.

Try it out. It should be possible to do this now:

```
class Volvo
  func printName()
    print('Volvo')
  end
end

var car := Volvo()
car.printName()    // prints 'Volvo'

Volvo().printName()  // prints 'Volvo'
```

Self

We have fields and we have methods, however it's not possible to refer to the fields from the methods yet. So, for example the following is what we wish to accomplish:

> There is only one corner of the universe you can be certain of improving, and that's your own self.
> — Aldous Huxley

```
class Car
  func printBrand()
    print(self.name + ' ' + self.type)
  end
end

var car := Car()
car.name := 'Volvo'

car.type := 'V60'
car.printBrand()

car.type := 'S90'
car.printBrand()
```

'self' is a keyword, recognized through the token ttSelf. It is also part of a factor expression, and is parsed in function ParseFactorExpr. But, first we need a new AST node for Self.

```
  TSelfExpr = class(TFactorExpr)
    private
      FVariable: TVariable;
    public
      property Variable: TVariable read FVariable;
      constructor Create(AVariable: TVariable);
      destructor Destroy; override;
  end;
```

The 'self' text is parsed as if it were a variable expression. The accompanying constructor and destructor are:

```
constructor TSelfExpr.Create(AVariable: TVariable);
begin
  inherited Create(AVariable.Ident.Token);
  FVariable := AVariable;
end;

destructor TSelfExpr.Destroy;
begin
  if Assigned(FVariable) then FVariable.Free;
  inherited Destroy;
end;
```

In ParseFactor add the case for ttSelf:

```
function TParser.ParseFactor: TExpr;
begin
  case CurrentToken.Typ of
    ...
    ttSelf: Result := ParseSelfExpr;
    ttIdentifier: Result := ParseIdentifierExpr;
    ...
end;
```

And also add the following parsing method for parsing the 'self' expression.

```
function TParser.ParseSelfExpr: TExpr;
var
  Variable: TVariable;
begin
  Variable := TVariable.Create(TIdent.Create(CurrentToken));
  Next;  // skip self
  Result := TSelfExpr.Create(Variable);
end;
```

The visitors for the printer and the resolver:

```
procedure TPrinter.VisitSelfExpr(SelfExpr: TSelfExpr);
begin
  IncIndent;
  VisitNode(SelfExpr);
  Visit(SelfExpr.Variable);
  DecIndent;
end;
```

In method VisitSelfExpr we resolve the locality of self as it were a normal variable.

```
procedure TResolver.VisitSelfExpr(SelfExpr: TSelfExpr);
begin
  ResolveLocal(SelfExpr.Variable);
end;
```

And of course we have to add it as a variable in VisitClassDecl. Also, it should be in a class scope.

```
procedure TResolver.ResolveClass(ClassDecl: TClassDecl);
var
  Decl: TDecl;
  FuncKind: TFuncKind;
begin
  BeginScope;
  Scopes.Top.Enter(TSymbol.Create('self', Enabled));
  for Decl in ClassDecl.DeclList do
    if Decl is TFuncDecl then begin
      FuncKind := fkMethod;
      ResolveFunction(Decl as TFuncDecl, FuncKind);
    end;
  EndScope;
end;
```

After the class is declared and defined, we begin a new scope. We add 'self' as a local variable to the scope, so that it can be referred to from the methods. Then, all methods are resolved, and finally, we end the scope again.

The visitor for the interpreter for Self is:

```
function TInterpreter.VisitSelfExpr(SelfExpr: TSelfExpr): Variant;
begin
  Result := Lookup(SelfExpr.Variable);
end;
```

This result should be a IGearInstance. What we have to do is bind 'self' to the instance it belongs to.

```
function TFunc.Bind(Instance: Variant): Variant;
var
  MemorySpace: TMemorySpace;
begin
  MemorySpace := TMemorySpace.Create(FClosure);
  MemorySpace.Store('self', Instance);
  Result := ICallable(TFunc.Create(FuncDecl, MemorySpace));
end;
```

A new memory space is created with the method's own closure as the enclosing memory space. The variable 'self' is declared within this memory space, and bound to the Instance.

And the bind function is called from TGearClass' FindMethod. Change the header to include a parameter Instance of type IGearInmstance. Change the last line in order to bind it to the instance. Note that we first have to cast to TFunc in order to call Bind.

```
function TGearClass.FindMethod(Instance: IGearInstance;
  const AName: String): ICallable;
var
  Index: integer = -1;
begin
  Result := Nil;
  if Methods.Contains(AName, Index) then
    Result := (ICallable(Methods.At(Index)) as TFunc).Bind(Instance);
end;
```

Add Self as an argument to the call to FindMethod in GetMember:

```
function TGearInstance.GetMember(Ident: TIdent): Variant;
...
begin
  ...
  Method := GearClass.FindMethod(Self, Ident.Text);
  ...
end;
```

And finally, add to the uses clause in uClass in the implementation section:

```
implementation
uses uFunc;
```

Try it out, this is now possible.

```
class Car
  func printBrand()
    print(self.name + ' ' + self.type)
  end
end

var car := Car()
car.name := 'Volvo'

car.type := 'V60'
car.printBrand()

car.type := 'S90'
car.printBrand()
```

You can build in error generation in the resolver for misuse of the 'self' name. Now if you use self outside a class a runtime error is created. But to make it perfect we'll build the checking mechanism.

Create a new type inside the resolver right under TFuncKind:

```
  TClassKind = (ckNone, ckClass);
```

Then, add a new local variable in the resolver class:

```
  CurrentClassKind: TClassKind;
```

In the constructor initialize it to ckNone:

```
constructor TResolver.Create;
begin
  GlobalScope := TScope.Create();
  GlobalScope.Sorted := True;
  EnableStandardFunctions;
  CurrentScope := GlobalScope;
  Scopes := TScopes.Create;
  CurrentFuncKind := fkNone;
  CurrentClassKind := ckNone;
end;
```

In method VisitClassDecl add a new variable EnclosingClassKind, which before resolving the full class is set to the CurrentClassKind. After resolving the class it's value is set back to the EnclosingClassKind again.

```
procedure TResolver.VisitClassDecl(ClassDecl: TClassDecl);
var
  EnclosingClassKind: TClassKind;
begin
  Declare(ClassDecl.Ident);
  Enable(ClassDecl.Ident);
  EnclosingClassKind := CurrentClassKind;
  CurrentClassKind := ckClass;
  ResolveClass(ClassDecl);
  CurrentClassKind := EnclosingClassKind;
end;
```

In method VisitSelfExpr we now check if 'self' is used in the global scope, or outside a class, in which case CurrentClassKind is set to ckNone.

```
ErrSelfOutsideClass = 'Cannot use "self" outside a class.';
```

```
procedure TResolver.VisitSelfExpr(SelfExpr: TSelfExpr);
begin
  if CurrentClassKind = ckNone then
    Errors.Append(SelfExpr.Token.Line, SelfExpr.Token.Col, ErrSelfOutsideClass);
  ResolveLocal(SelfExpr.Variable);
end;
```

And now for 'self' there's one minor thing to solve. Currently, it's not possible to assign to a self expression. So, inside a method you cannot do: "self.name := newName" for instance. Luckily, this is solved extremely easy.

In uParser.pas, add the constant ttSelf to the statement start set:

```
const
  StmtStartSet: TTokenTypSet = [ttIf, ttWhile, ttRepeat, ttFor, ttReturn,
    ttSwitch, ttEnsure, ttPrint, ttSelf, ttBreak, ttIdentifier];
```

Run the program, which now accepts this code:

```
class Car
  func printBrand()
    print(self.name + ' ' + self.type)
  end
  func setName(to newName)
    self.name := newName
  end
end

var car := Car()
car.name := 'Volvo'

car.type := 'V60'
car.printBrand()              // prints Volvo V60

car.setName(to: 'New Volvo')
car.type := 'S90'
car.printBrand()              // prints New Volvo S90
```

Class construction with init

We can already construct a class by calling Class(), as if it were a function. In this paragraph we introduce class construction with an initializer. As an example consider the following code snippet.

> The least initial deviation from the truth is multiplied later a thousandfold.
> — Aristotle

```
class Car
  func show()
    print(self.name)
  end
  init(.name)
    self.name := name
  end
end

var car := Car(name: 'Volvo')
print(car.name)
car.show()
```

The init call returns an instance, and if any parameters, automatically creates the field(s). 'init' is defined as a function without the 'func' keyword. First, let's add the init keyword/token to the declaration start set.

```
const
  DeclStartSet: TTokenTypSet = [ttClass, ttFunc, ttInit, ttLet, ttVar];
```

In the parser we already defined the private type TFuncForm, which could be normal or anonymous. Add the ffInit constant, as init is also just a form of a function:

```
  TParser = class
    private
      type TFuncForm = (ffNormal, ffAnonym, ffInit);
    ...
    ...
```

Then, in function ParseFuncDecl add the new case for ffInit:

```
function TParser.ParseFuncDecl(FuncForm: TFuncForm): TDecl;
...
...
begin
  Token := CurrentToken;
  case FuncForm of
    ffNormal: begin Next; Name := ParseIdent; end;
    ffAnonym: Next;
    ffInit:   begin Name := TIdent.Create(Token); Next; end;
  end;
...
...
end;
```

When Normal, we skip the func keyword and parse the function name, when Anonymous, we skip the func keyword and the function name is Nil. When an Init, then Init is the name of the function, and the Next makes sure we move to the next token.

Finally, in the parser, add the case for ttInit to function ParseDecl, like this:

```
function TParser.ParseDecl: TDecl;
const Mutable = True;
begin
  case CurrentToken.Typ of
    ttClass: Result := ParseClassDecl;
    ttFunc: Result := ParseFuncDecl(ffFunction);
    ttInit: Result := ParseFuncDecl(ffInit);
    ttLet: Result := ParseVarDecl(Mutable = False);
    ttVar: Result := ParseVarDecl(Mutable = True);
  end;
end;
```

In TGearClass we make a few changes to function Call:

```
function TGearClass.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Instance: IGearInstance;
  Init: TFunc;
  Index: LongInt = -1;
begin
  Instance := IGearInstance(TGearInstance.Create(Self));  // cast to IGearInstance

  if Methods.Contains('init', Index) then begin
    Init := ICallable(Methods.At(Index)) as TFunc;
    Arity := Init.FuncDecl.Params.Count; // Get arity of init(params)

    if (ArgList.Count = 0) and (Arity > 0) then  // Is init called or not
      Exit(Instance)  // init is not called, but Class() directly
    else
      ICallable(Init.Bind(Instance)).Call(Token, Interpreter, ArgList);
  end;
  Result := Instance;
end;
```

If there is an init method, it must be called. We bind it to the instance and call the method with the given arguments. All argument checking is in place. If there is no init method, arguments are not allowed (arity must be zero).

Since a class can only have 0 or 1 init(), we first determine the index of the init. Then we retrieve the init's arity, or how many parameters does it have. And now comes the trick. If the number of arguments in the constructor is zero, but the number of parameters (arity) is greater than zero, we are actually creating an instance without calling init!

If on the other hand the number of arguments is greater than zero, we are calling init. In the call there's an automatic check on arity: the number of arguments must match the number of parameters.

And with this, you can try and run the code from the beginning of this paragraph.

Note that you can't call something like car.init(), meaning init() as a method of an instance. Init is a reserved word and cannot be used as an identifier.

Finally, we have to look at the return statement again. It should not be possible to use return from an init(), as it will always return an instance automatically.

In the Resolver unit add the enum fkInit to the type TFuncKind:

```
  TFuncKind = (fkNone, fkFunc, fkInit, fkMethod);
```

Then adjust function ResolveClass such that we build in a check for 'init', in which case the FuncKind becomes fkInit.

```
procedure TResolver.ResolveClass(ClassDecl: TClassDecl);
...
begin
  ...
  for Decl in ClassDecl.DeclList do
    if Decl is TFuncDecl then begin
      FuncKind := fkMethod;
      if (Decl as TFuncDecl).Ident.Text = 'init' then
        FuncKind := fkInit;
  ...
end;
```

Finally, change the VisitReturnStmt function as follows:

```
procedure TResolver.VisitReturnStmt(ReturnStmt: TReturnStmt);
begin
  if CurrentFuncKind in [fkNone, fkInit] then
    with ReturnStmt do
      Errors.Append(Token.Line, Token.Col, ErrReturnFromFunc);
  Visit(ReturnStmt.Expr);
end;
```

And this is it for the basic class functionality! Try it out.
Code like this is now possible:

```
class Car
  func show()
    print(self.name)
  end
  init(.name)
    self.name := name
  end
end

var car := Car(name: 'Volvo')
print(car.name)
car.show()

var other := Car()
other.name := 'Merc'
print(other.name)
other.show()


class Me
  init()
    print(self)
  end
end

var me := Me()
print(me)
```

Variable declarations

> To be is to be the value of a variable.
> — Willard Van Orman Quine

As mentioned in the beginning of this chapter, I like it when variable fields inside a class are already declared at compile time. It becomes clear to the programmer which fields can be used, and reduces the risk on errors.

In this paragraph, we'll do the following:

- introduce variable declarations inside a class
- stop allowing dynamically adding fields
- allow initialization without using the init() constructor

This paragraph is not mandatory, and can be skipped if you want to keep the freedom of adding fields dynamically!

At the end of this section, the following can be compiled and run:

```
class Car
  var name := 'Car'
  func show()
    print(self.name)
  end
  init(.name)
    self.name := name
  end
end

var car := Car()
print(car.name)          // prints 'Car'
car.show()               // prints 'Car'

var volvo := Car(name: 'Volvo')
print(volvo.name)        // prints 'Volvo'
volvo.show()             // prints 'Volvo'

car.show()               // prints 'Car'
```

However, assignments to non-declared fields will not be allowed, as demonstrated in the following:

```
car.brand := 'Ferrari'    // Undefined class field "brand" error.
```

First, we are going to create a smarter way than checking classes with 'is' TClassType. We have to allow for a lot more types later on. Go to the AST and add a declaration type to class TDecl:

```
  TDecl = class(TNode)
    private
       type TDeclKind = (dkVar, dkFunc, dkClass);
    private
       FKind: TDeclKind;
       FIdent: TIdent;
    public
      property Ident: TIdent read FIdent;
      property Kind: TDeclKind read FKind;
      constructor Create(AIdent: TIdent; AKind: TDeclKind; AToken: TToken);
      destructor Destroy; override;
  end;
```

The constructor becomes:

```
constructor TDecl.Create(AIdent: TIdent; AKind: TDeclKind; AToken: TToken);
begin
  Inherited Create(AToken);
  FIdent := AIdent;
  FKind := AKind;
end;
```

Now, in the constructors of VarDecl, FuncDecl and ClassDecl change the inherited call:
In TVarDecl.Create:

```
  inherited Create(AIdent, dkVar, AToken);
```

In TFuncDecl.Create:

```
  inherited Create(AIdent, dkFunc, AToken);
```

In TClassDecl.Create:

```
  inherited Create(AIdent, dkClass, AToken);
```

Next, let's make sure the variable declarations inside a class are correctly resolved and interpreted, using the above declaration kind:

```
procedure TResolver.ResolveClass(ClassDecl: TClassDecl);
var
  Decl: TDecl;
  FuncKind: TFuncKind;
begin
  BeginScope;
  Scopes.Top.Enter(TSymbol.Create('self', Enabled));
  for Decl in ClassDecl.DeclList do begin
    case Decl.Kind of
      dkFunc: begin
        FuncKind := fkMethod;
        if Decl.Ident.Text = 'init' then
          FuncKind := fkInit;
        ResolveFunction(Decl as TFuncDecl, FuncKind);
      end;
      dkVar: Visit(Decl as TVarDecl);
    end;
  end;
  EndScope;
end;
```

We coninue with the class fields, the variable declarations. The fields in a class can be variable or constant. If a constant is declared with a Null value, it can receive a one-time new value. This is especially handy in class initializers, for example:

```
class Car
  let brand := Null
  init(brand)
    self.brand := brand  // brand has a value now and is immutable
  end
end
car := Car('Volvo') // brand has value 'Volvo'
car.brand := 'Merc'  // this is not allowed
```

The problem here is that when a setter is called, such as 'test.x := 7', we are outside the original scope, and we just directly call upon the field belonging to the class instance. This field doesn't contain any information on mutability. That's what we must build in!

In fact, there's no way currently, in the Resolver, to find the original class from a class instance. And that's what you really need in visitor VisitSetStmt(), where the new value is assigned to a class field. We have to do it in runtime, and build it into the interpreter, as here we do know the class and its fields.

The approach is, just like we did for fields, to create a separate dictionary of constants.

Let's look at the visitor VisitClassDecl, and also apply the DeclKind to switch between different declaration types.

Here's the complete class visitor:

```
procedure TInterpreter.VisitClassDecl(ClassDecl: TClassDecl);
var
  GearClass: TGearClass;
  Decl: TDecl;
  Func: TFunc;
  Members: TMembers;
begin
  Members.Init;
  CheckDuplicate(ClassDecl.Ident, 'Class');
  CurrentSpace.Store(ClassDecl.Ident, Nil);
  for Decl in ClassDecl.DeclList do begin
    case Decl.Kind of
      dkFunc: begin
        Func := TFunc.Create(Decl as TFuncDecl, CurrentSpace);
        Members.Methods[Decl.Ident.Text] := ICallable(Func);
      end;
      dkVar:
        if (Decl as TVarDecl).Mutable then
          Members.Fields[Decl.Ident.Text] := Visit((Decl as TVarDecl).Expr)
        else
          Members.Constants[Decl.Ident.Text] := Visit((Decl as TVarDecl).Expr);
    end;
  end;
  GearClass := TGearClass.Create(ClassDecl.Ident, Members);
  CurrentSpace.Update(ClassDecl.Ident, IClassable(GearClass));
end;
```

The changes comprise adding the fields and constants and their interpreted expressions (so real values now) to the constructor of the GearClass. We use the earlier defined TMember record for this.

TGearClass will get new private variables Fields and Constants:

```
  TGearClass = class(TInterfacedObject, ICallable)
    private
      ...
      Fields: TFieldTable;
      Constants: TConstTable;
    public
      ...
  end;
```

In the constructor we retrieve their values from the Member record.

```
constructor TGearClass.Create(AIdent: TIdent; Members: TMembers);
begin
  Ident := AIdent;
  Methods := Members.Methods;
  Fields := Members.Fields;
  Constants := Members.Constants;
end;
```

We made the fields and their initial values (the interpreted expressions) part of the class. However, what we really would like is for them to become part of an instance of that particular class. So, we need to find a way to copy the fields and their values to the instance.

This is easily done in the constructor of the instance. First add field InstanceConsts to the TGearInstance class:

```
  TGearInstance = class(TInterfacedObject, IGearInstance)
    private
      GearClass: TGearClass;
      InstanceFields: TFieldTable;
      InstanceConsts: TConstTable;
    public
      constructor Create(AGearClass: TGearClass);
      function toString: String; override;
      function GetMember(Ident: TIdent): Variant;
      procedure SetField(Ident: TIdent; Value: Variant);
  end;
```

```
constructor TGearInstance.Create(AGearClass: TGearClass);
var
  i: Integer;
begin
  GearClass := AGearClass;
  InstanceFields := TFieldTable.Create();
  InstanceFields.Sorted := True;
  for i := 0 to GearClass.Fields.Count-1 do
    InstanceFields.Add(GearClass.Fields.Keys[i], GearClass.Fields.Data[i]);
  InstanceConsts := TConstTable.Create();
  InstanceConsts.Sorted := True;
  for i := 0 to GearClass.Constants.Count-1 do
    InstanceConsts.Add(GearClass.Constants.Keys[i], GearClass.Constants.Data[i]);
  InstanceConsts['className'] := GearClass.Ident.Text;
end;
```

We just add the fields that were passed to the class, as a copy, to the instance.

Finally, adjust GetMember so that it also searches in the constants:

```
function TGearInstance.GetMember(Ident: TIdent): Variant;
...
begin
  ...

  if InstanceConsts.Contains(Ident.Text, i) then
    Exit(InstanceConsts.At(i));
  ...
end;
```

For example, have a look at this declaration:

```
class Date
  var day := 1
  var month := 'January'
  var year := 1980
  init(.day, .month, .year)
    self.day := day
    self.month := month
    self.year := year
  end
  func toString()
    return 'Date: ' + self.day+'-'+self.month+'-'+self.year
  end
end

class Person
  var name := 'Person'
  var birthDate := Date()
  init(name, .birthDate)
    self.name := name
    self.birthDate := birthDate
  end
  func toString()
    return self.name + ': ' + self.birthDate.toString()
  end
end

var Harry := Person('Harry', birthDate: Date(day: 23, month: 'February', year: 1987))
var Sally := Person('Sally', birthDate: Date(day: 18, month: 'July', year: 1992))

print(Harry.toString())
print(Sally.toString())
print(Person().toString())
```

If you accidentilly forget to use 'self' before a variable in a method or val, there's currently no error or warning, despite the fact that it compiles and runs. So, we want to make sure that when a variable, which is declared in a class, is used in either a func or a val, has 'self.' in front of it. If we don't than the variable is in fact unassigned or empty. Let's create a check on this:

```
  ErrUndefVarOrSelfMissing = 'Variable is undefined or missing "self".';
```

```
function TInterpreter.VisitVariable(Variable: TVariable): Variant;
begin
  Result := Lookup(Variable);
  if VarIsEmpty(Result) then
    Raise ERuntimeError.Create(Variable.Token, ErrUndefVarOrSelfMissing);
end;
```

And that's all! Test it if you like.

The above example gives the following result:

```
Harry: Date: 23-February-1987
Sally: Date: 18-July-1992
Person: Date: 1-January-1980
```

But how do we handle it such that constants can not be assigned to anymore after they've received their value after initialization.

In order to find out if a member is a constant we need to add the function isConstant to TGearInstance. In the VisitSetStmt visitor we will use this function.

```
  IGearInstance = Interface
    ...
    function isConstant(Ident: TIdent): Boolean;
  end;
```

Add the function to TGearInstance as well:

```
  TGearInstance = class(TInterfacedObject, IGearInstance)
    public
      ...
      function isConstant(Ident: TIdent): Boolean;
  end;
```

And then implement the function.

```
function TGearInstance.isConstant(Ident: TIdent): Boolean;
begin
  Result := InstanceConsts.Contains(Ident.Text);
end;
```

It just checks whether an identifier is part of the list of constants. It returns true or false depending on the search result.

Finally, we change the inerpreter's VisitSetStmt visitor. We check if the variable is a constant and not Null. In this case we throw an error, since we cannot mutate it. The whole procedure is shown. Quite some stuff has changed. Also, a field must now exist, as we try to get the old value.

```
procedure TInterpreter.VisitSetStmt(SetStmt: TSetStmt);
var
  Instance, OldValue, NewValue, Value: Variant;
begin
  Instance := Visit(SetStmt.Instance);
  if not VarSupports(Instance, IGearInstance) then
    Raise ERuntimeError.Create(SetStmt.Token, ErrExpectedClassInstance);

  OldValue := IGearInstance(Instance).GetMember(SetStmt.Ident);

  if IGearInstance(Instance).isConstant(SetStmt.Ident) and
     (not VarIsNull(OldValue)) then
    Raise ERuntimeError.Create(SetStmt.Ident.Token, Format(
      ErrClassMemberImmutable, [SetStmt.Ident.Text]));

  NewValue := Visit(SetStmt.Expr);
  Value := getAssignValue(OldValue, NewValue, SetStmt.Ident.Token, SetStmt.Op);
  IGearInstance(Instance).SetField(SetStmt.Ident, Value);
end;
```

For this to work we change the AST node for SetStmt to include an operator:

```
  TSetStmt = class(TStmt)
    private
      ...
      FOp: TToken;
    public
      ...
      property Op: TToken read FOp;
      constructor Create(AInstance: TExpr; AIdent: TIdent; AOp: TToken;
        AExpr: TExpr);
      ...
  end;
```

And the constructor:

```
constructor TSetStmt.Create
  (AInstance: TExpr; AIdent: TIdent; AOp: TToken; AExpr: TExpr);
begin
  ...
  FOp := AOp;
end;
```

Also, change the function ParseAssignStmt, so that the SetStmt gets the extra parameter as well:

```
function TParser.ParseAssignStmt: TStmt;
...
begin
  ...
    else if Left is TGetExpr then
      with Left as TGetExpr do
        Result := TSetStmt.Create(Instance, Ident, Op, Right)
  ...
end;
```

So, in conclusion, unfortunately, we had to create some extra code to check during runtime whether a class variable is mutable or not. However, it works as can be seen with the below test code, for which constant variables are declared. Moreover, it only works correctly if the initial value is Null. And this is exactly what we want.

```
class Base
  let it := Null
  init(it)
    self.it := it
  end
end

let base := Base('Good')

print(base.it)
base.it := 'allo' // error
```

Though for example you can assign a class instance to a constant, it is still possible to mutate fields of this class, like in this example.

```
class Car
  var name := ''
  init(.name)
    self.name := name
  end
  func toString() => self.name
end

let volvo := Car(name: 'Volvo')
print(volvo.toString())
```

```
volvo.name := 'Volvo V60'
print(volvo.toString())
```

Parsing class members

So far we've been easy on parsing the type of member inside a class. We've even allowed parsing init() outside a class (!). We have to make a statement on the first one and rectify the second one.

But to start of with we have to change the way the Block is parsed. ParseBlock now only accepts correct declarations and statements, and will just ignore any undefined behavior. So, change function ParseBlock in the parser to the following, including the constant BlockEndSet:

```
const
  BlockEndSet: TTokenTypSet = [ttElse, ttUntil, ttEnd, ttEOF];

function TParser.ParseBlock: TBlock;
begin
  Result := TBlock.Create(TNodeList.Create(), CurrentToken);
  while not BlockEndSet.Contains(CurrentToken.Typ) do
    Result.Nodes.Add(ParseNode);
end;
```

This will continue parsing block nodes until a token from the BlockEndSet is found. Now function ParseNode will report incorrect behavior.

Next, we have defined a constant DeclStartSet, which contains the token types [ttClass, ttFunc, ttInit, ttLet, ttVar]. However, typically ttInit is only used inside a class, so shouldn't be in this place. Remove ttStatic so we have:

```
const
  DeclStartSet: TTokenTypSet = [ttClass, ttFunc, ttLet, ttVar];
```

Also, remove the case for ttInit from function ParseDecl, so we end up with the next function:

```
function TParser.ParseDecl: TDecl;
const Mutable = True;
begin
  case CurrentToken.Typ of
    ttClass: Result := ParseClassDecl;
    ttFunc: Result := ParseFuncDecl(ffFunction);
    ttLet: Result := ParseVarDecl(Mutable = False);
    ttVar: Result := ParseVarDecl(Mutable = True);
  end;
end;
```

So, how are we going to parse Init() then? First, we define two new constant sets. Put them right above ParseClassDecl:

```
const
  ClassStartSet: TTokenTypSet = [ttFunc, ttInit, ttLet, ttVar];
```

It determines which declarations can be done inside a class. You'll notice I won't support a class declaration inside a class (yet).

Then, we need a way to parse to parse the class declarations. Change ParseClassDecl as follows:

```
function TParser.ParseClassDecl: TDecl;
var
  Ident: TIdent;
  DeclList: TDeclList;
  Token: TToken;
begin
  Token := CurrentToken;
  Next;  // skip class
  Ident := ParseIdent;
  DeclList := TDeclList.Create(false);
  while ClassStartSet.Contains(CurrentToken.Typ) do begin
    case CurrentToken.Typ of
      ttFunc: DeclList.Add(ParseFuncDecl(ffFunction));
      ttInit: DeclList.Add(ParseFuncDecl(ffInit));
      ttLet: DeclList.Add(ParseVarDecl(False));
      ttVar: DeclList.Add(ParseVarDecl(True));
    end;
  end;
  Result := TClassDecl.Create(Ident, DeclList, Token);
  Expect(ttEnd);
end;
```

It will only parse functions, initializers, variables and constants for now.

Finally, in the interpreter I want to make the VisitClassDecl visitor a bit more readable, so I introduce a small helper function that gets the members into a TMember record, and use that in VisitClassDecl:

```
function TInterpreter.getMembers(DeclList: TDeclList): TMembers;
var
  Decl: TDecl;
  Func: TFunc;
begin
  Result.Init;
  for Decl in DeclList do begin
    case Decl.Kind of
      dkFunc: begin
        Func := TFunc.Create(Decl as TFuncDecl, CurrentSpace);
        Result.Methods[Decl.Ident.Text] := ICallable(Func);
      end;
      dkVar:
        if (Decl as TVarDecl).Mutable then
          Result.Fields[Decl.Ident.Text] := Visit((Decl as TVarDecl).Expr)
        else
          Result.Constants[Decl.Ident.Text] := Visit((Decl as TVarDecl).Expr);
    end;
  end;
end;
```

New member types can easily be added to the case list.

VisitClassDecl now becomes a lot more understandable and readable:

```
procedure TInterpreter.VisitClassDecl(ClassDecl: TClassDecl);
var
  GearClass: TGearClass;
```

```
  Members: TMembers;
begin
  CheckDuplicate(ClassDecl.Ident, 'Class');
  CurrentSpace.Store(ClassDecl.Ident, Nil);
  Members := getMembers(ClassDecl.DeclList);
  GearClass := TGearClass.Create(ClassDecl.Ident, Members);
  CurrentSpace.Update(ClassDecl.Ident, IClassable(GearClass));
end;
```

Chapter 7 – Class inheritance

Inheritance in most class-based object-oriented languages is a mechanism in which one object acquires all the properties and behaviours of the parent object. Inheritance allows programmers to create classes that are built upon existing classes, to specify a new implementation to maintain the same behaviour (realizing an interface), to reuse code and to independently extend original software via public classes and interfaces. An inherited class is called a subclass of its parent class (or super class).

Consider the following class and inheritance example.

```
class Vehicle
  ...
end

class Car (Vehicle)
  ...
end

class Train (Vehicle)
  ...
end
```

This means that both class Car and Train inherit from (or is subclass of) class Vehicle. Vehicle is the parent class.
Remember our EBNF for class declaration:

```
ClassDecl = 'class' Ident { Decl } 'end'  .
```

Let's add support for parent classes to this:

```
ClassDecl = 'class' Ident [ '(' Ident ')' ] { Decl } 'end'  .
```

Note that the parent class definition is between [], which means that it's optional.
The definition for TClassDecl in unit uAst.pas needs to be enhanced to capture the parent.

```
TClassDecl = class(TDecl)
  private
    ...
    FParent: TVariable;
  public
    ...
    property Parent: TVariable read FParent;
    constructor Create(AIdent: TIdent; AParent: TVariable;
      ADeclList: TDeclList; AToken: TToken);
    ...
end;
```

And the implementation of the constructor will add the value of the parent:

```
constructor TClassDecl.Create
  (AIdent: TIdent; AParent: TVariable; ADeclList: TDeclList; AToken: TToken);
begin
  ...
  FParent := AParent;
end;
```

```
destructor TClassDecl.Destroy;
begin
  ...
  if Assigned(FParent) then FParent.Free;
  ...
end;
```

The parent is stored here as a TVariable. This helps us in resolving and interpreting later on, since the parent will be treated as a variable expression.

Next, we'll change the parser for class declaration:

```
function TParser.ParseClassDecl: TDecl;
var
  ...
  Parent: TVariable = Nil;
begin
  ...
  Ident := ParseIdent;
  if CurrentToken.Typ = ttOpenParen then begin
    Next;  // skip (
    Parent := TVariable.Create(ParseIdent);
    Expect(ttCloseParen);
  end;
  DeclList := TDeclList.Create(false);
  ...
  Result := TClassDecl.Create(Ident, Parent, DeclList, Token);
  Expect(ttEnd);
end;
```

Variable Parent is declared with initial value Nil. Only if we find a '(Ident)' Parent will get a value other than Nil. The Parent identifier is parsed and stored inside a TVariable.

We also have to make small changes to the Printer and the Resolver:

```
procedure TPrinter.VisitClassDecl(ClassDecl: TClassDecl);
  ...
begin
  ...
  Visit(ClassDecl.Ident);
  if Assigned(ClassDecl.Parent) then
    Visit(ClassDecl.Parent);
  ...
end;
```

The visitor for the resolver needs more work though. When we made the choice to use the variant type to hold all kinds of values, we automatically were restricted to non class types. In other words it is impossible to test variants on class membership, e.g. 'value is classtype' doesn't work for variants. In the case of resolving whether a parent is a class type, we cannot perform that check. However, we don't just want that any variable can be used as a parent. E.g. this is wrong:

```
var x := 2
class One(x)
end
```

Whereas this is right:

```
class One
end

class Two(One)
end
```

The best place to perform this check is in the resolver, as we don't want to do it at runtime, as this costs precious time. Let's start with the visitor VisitClassDecl:

```
procedure TResolver.VisitClassDecl(ClassDecl: TClassDecl);
var
  EnclosingClassKind: TClassKind;
begin
  Declare(ClassDecl.Ident);
  Enable(ClassDecl.Ident);
  if Assigned(ClassDecl.Parent) then begin
    Visit(ClassDecl.Parent);
    if not ClassList.Contains(ClassDecl.Parent.Ident.Text) then
      with ClassDecl.Parent.Ident do
        Errors.Append(Token.Line, Token.Col, Format(ErrParentNoClass, [Text]));
  end;
  ClassList.Add(ClassDecl.Ident.Text);  // add this class to list
  EnclosingClassKind := CurrentClassKind;
  CurrentClassKind := ckClass;
  ResolveClass(ClassDecl);
  CurrentClassKind := EnclosingClassKind;
end;
```

If the Parent is filled, we first visit it, and then we look in a new to be created list of classes if it was already declared. This also means classes must be declared in the right order. You cannot declare the parent class after the child class.

Also, it must not be possible that a parent class is the same as the class that's just being declared, like this:

```
class One(One)
end
```

That's why the current class node ident is saved in the class list after the check if a parent is found.

Define a new type above TResolver:

```
  TClassList = specialize TArray<String>;
```

The list of declared classes is captured in a private field in class TResolver. Add this line:

```
      ClassList: TClassList;
```

In the constructor and destructor make the following changes:

```
constructor TResolver.Create;
begin
  ...
  ClassList := TClassList.Create;
end;

destructor TResolver.Destroy;
begin
  ...
  ClassList.Free;
  inherited Destroy;
end;
```

This resolves parent classes.

Then, moving on to the interpreter. Change VisitClassDecl as follows:

```
procedure TInterpreter.VisitClassDecl(ClassDecl: TClassDecl);
var
  ...
  Parent: Variant;
begin
  ...
  CurrentSpace.Store(ClassDecl.Ident, Nil);
  if Assigned(ClassDecl.Parent) then
    Parent := Visit(ClassDecl.Parent)
  else Parent := Null;
  ...
  GearClass := TGearClass.Create(ClassDecl.Ident, Parent, Members);
  ...
end;
```

Define a new variable Parent of Variant. If the ClassDecl node has a Parent class than this node is visited and the resulting value is stored in variable Parent.

Finally, we pass the Parent class to the constructor of TGearClass, which stores the parent class in a field Parent.

```
  TGearClass = class(TInterfacedObject, ICallable)
    private
      ...
      Parent: Variant;
    public
      ...
      constructor Create(AIdent: TIdent; AParent: Variant; Members: TMembers);
      ...
  end;


constructor TGearClass.Create
  (AIdent: TIdent; AParent: Variant; Members: TMembers);
begin
  ...
  Parent := AParent;
  ...
end;
```

Now that's in place, we need to see how to make the parent class practical.

Finding methods in the parent class

With inheritance in its simplest form, the methods at parent class level become available at child class level. Or in other words, the child class can use the methods of the parent class. For this to work we only need to make a small change to the function FindMethod of TGearClass. If the respective method can't be found, we'll try to find it in the parent class.

> The story of the Web starts in 1980, when Berners-Lee, a young consulting physicist at the CERN physics laboratory near Geneva, grew frustrated with existing methods for finding and transferring information.
>
> — Katie Hafner

```
function TGearClass.FindMethod(Instance: IGearInstance;
  const AName: String): ICallable;
var
  Index: integer = -1;
begin
  Result := Nil;
  if Methods.Contains(AName, Index) then
    Result := (ICallable(Methods.At(Index)) as TFunc).Bind(Instance)
  else if not VarIsNull(Parent) then
    Result := (ICallable(Parent) as TGearClass).FindMethod(Instance, AName);
end;
```

If the parent class does exist (not Null), we recursively search the method in Parent. Note that we first have to cast Parent to ICallable and then to a TGearClass in order to call FindMethod. This way you can chain multiple parent classes. Compile and run for example below input code:

```
class First
  func toString()
    return 'First'
  end
end

class Second(First)
end

class Third(Second)
  func toString()
    return 'Third'
  end
end

class Fourth(Third)
end

var second := Second()
print(second.toString())    // prints 'First'

var third := Third()
print(third.toString())     // prints 'Third'

var fourth := Fourth()
print(fourth.toString())    // prints 'Third'
```

What you see here is that the second definition of toString() completely overrides the first definition. There is no way to call the first toString() method from the third or fourth class. We need a way to call the method of the parent class directly, should we wish to.

Calling inherited methods

If you look at the unit uToken.pas you'll notice it already contains the basics for parsing the keyword 'inherited'. Especially in init() methods it can be helpful to call the parent's init(). You do this for example by using:

> Your job is not just to do what your parents say, what your teachers say, what society says, but to figure out what your heart calling is and to be led by that.
> — Oprah Winfrey

```
inherited init()
```

The referral to 'inherited' is a so-called variable access, and thus parsed in method Factor. The EBNF of Factor now becomes:

```
    Factor     = String | Char | Number
               | Nil | True | False
               | '(' Expr ')'
               | 'func' '(' Parameters ')' ('=>' Expr | Block 'end')
               | IfExpr | MatchExpr
               | VarExpr
               | 'self'
               | 'inherited' Ident .
```

Putting this into practice, we create a new AST node for the inherited method call, like this:

```
  TInheritedExpr = class(TFactorExpr)
    private
      FVariable: TVariable;
      FMethod: TIdent;
    public
      property Variable: TVariable read FVariable;
      property Method: TIdent read FMethod;
      constructor Create(AVariable: TVariable; AMethod: TIdent);
      destructor Destroy; override;
  end;
```

```
constructor TInheritedExpr.Create(AVariable: TVariable; AMethod: TIdent);
begin
  inherited Create(AVariable.Ident.Token);
  FVariable := AVariable;
  FMethod := AMethod;
end;

destructor TInheritedExpr.Destroy;
begin
  if Assigned(FVariable) then FVariable.Free;
  if Assigned(FMethod) then FMethod.Free;
  inherited Destroy;
end;
```

Note that the keyword 'inherited' is parsed as a TVariable. This makes resolving and interpreting it later on easier.

Now on to the parsing of 'inherited' itself. Create a new parse function for this:

```
function TParser.ParseInheritedExpr: TExpr;
var
  Variable: TVariable;
begin
  Variable := TVariable.Create(TIdent.Create(CurrentToken));
  Next;  // skip inherited
  Result := TInheritedExpr.Create(Variable, ParseIdent);
end;
```

and add the case for ttInherited to function ParseFactor, right before 'ttSelf:' .

```
    ttInherited: Result := ParseInheritedExpr;
```

Also, add the token ttInherited to the statement start set:

```
const
  StmtStartSet: TTokenTypSet = [ttIf, ttWhile, ttRepeat, ttFor, ttReturn,
    ttSwitch, ttEnsure, ttPrint, ttInherited, ttSelf, ttBreak, ttIdentifier];
```

As usual, we start with the printer visitor:

```
procedure TPrinter.VisitInheritedExpr(InheritedExpr: TInheritedExpr);
begin
  IncIndent;
  VisitNode(InheritedExpr);
  Visit(InheritedExpr.Variable);
  Visit(InheritedExpr.Method);
  DecIndent;
end;
```

Next, we resolve 'inherited' up the chain by starting a new scope when a parent class is found. In this new scope we add 'inherited' as a variable. Also, end the scope in case of a Parent class.

```
procedure TResolver.VisitClassDecl(ClassDecl: TClassDecl);
...
begin
  ...
  if Assigned(ClassDecl.Parent) then begin
    ...
    BeginScope;
    Scopes.Top.Enter(TSymbol.Create('inherited', Enabled));
  end;
  ...
  if Assigned(ClassDecl.Parent) then EndScope;
  CurrentClassKind := EnclosingClassKind;
end;
```

Now add the visitor for the inherited expression itself, including the new error message:

```
  ErrInheritedInClass = 'Can only use "inherited" inside a class.';
```

```
procedure TResolver.VisitInheritedExpr(InheritedExpr: TInheritedExpr);
begin
  if CurrentClassKind = ckNone then
    with InheritedExpr do
      Errors.Append(Token.Line, Token.Col, ErrInheritedInClass);
  ResolveLocal(InheritedExpr.Variable);
end;
```

It is to make sure we don't use 'inherited' outside a class, much the same as for 'self'.
Jusr like we did with 'self', we also Resolve the inherited keyword as a local variable.

In the visitor for the interpreter, we do more or less the same with the memory spaces. We create a new memory space where we store the 'inherited' variable.

```
procedure TInterpreter.VisitClassDecl(ClassDecl: TClassDecl);
var
  ...
begin
  ...
  if Assigned(ClassDecl.Parent) then begin
    Parent := Visit(ClassDecl.Parent);
    CurrentSpace := TMemorySpace.Create(CurrentSpace);
    CurrentSpace.Store('inherited', Parent);
  end
  ...
  if Assigned(ClassDecl.Parent) then
    CurrentSpace := CurrentSpace.EnclosingSpace;
  GearClass := TGearClass.Create(ClassDecl.Ident, Parent, Members);
  ...
end;
```

And just like in the Resolver, return to the original Space by setting CurrentSpace to its EnclosingSpace.

Moving on to the interpreter visitor for the inherited expression. Remember that in the Resolver, for each variable expression (used variable), we determined the location where it was declared and compare that to where it's used and called it 'distance'. Finally, we can reap the benefits of that for 'inherited'. At 'distance' we load the class connected to 'inherited'. Then, we load at the distance minus 1 the instance connected to 'self'. Finally, we find the respective method as the result.

Add the error message to the list of constants:

```
  ErrUndefinedMethod = 'Undefined method "%s".';
```

```
function TInterpreter.VisitInheritedExpr(InheritedExpr: TInheritedExpr): Variant;
var
  Distance: Integer;
  Parent, Method: ICallable;
  Instance: IGearInstance;
begin
  Distance := InheritedExpr.Variable.Distance;
  Parent := ICallable(CurrentSpace.LoadAt(Distance, 'inherited'));
  Instance := IGearInstance(CurrentSpace.LoadAt(Distance-1, 'self'));
  Method := (Parent as TGearClass).FindMethod(Instance, InheritedExpr.Method.Text);
  if Method = Nil then
    Raise ERuntimeError.Create(InheritedExpr.Method.Token, Format(
      ErrUndefinedMethod, [InheritedExpr.Method.Text]));
  Result := Method;
end;
```

If the method wasn't found we report a nice error message.

You can try and run for example below code:

```
class ApplePie
  func serve()
    print('Serve warm apple pie', terminator: ', ')
  end
end

class Apfelstrudel(ApplePie)
  func serve()
    inherited serve()
    print('on a hot plate with vanilla saus.')
  end
end

Apfelstrudel().serve()
```

Results in:

```
Serve warm apple pie, on a hot plate with vanilla saus.
```

However, if you make an error in calling an inherited method…

```
class ApplePie
  func serve()
    print('Serve warm apple pie,')
  end
end

class Apfelstrudel(ApplePie)
  func serve()
    inherited service()
    print('on a hot plate with vanilla saus.')
  end
end

Apfelstrudel().serve()
```

Results in:

```
@[9,15] Runtime error: Undefined method "service".
```

Inherited initialization

Now that we are able to call inherited methods we need to go one step further, and that's allowing to call 'inherited init()'.

We want to be able to do the following:

> Our object in the construction of the state is the greatest happiness of the whole, and not that of any one class.
> — Plato

```
class Circle
  var radius := 1
  func area() => pi() * self.radius^2
  init(.radius)
    self.radius := radius
  end
end

class Cylinder (Circle)
  var height := 0
  func volume() => self.area() * self.height
  init(.radius, .height)
    inherited(radius: radius)
    self.height := height
  end
end

var circle := Circle(radius: 1)
var cylinder := Cylinder(radius: 2, height: 10)

print(circle.radius)     // prints 1
print(circle.area())       // prints 3.14159265358979
print(cylinder.radius)   // prints 2
print(cylinder.area())     // prints 12.5663706143592
print(cylinder.volume())   // prints 125.663706143592
```

In this example we introduce a couple of new things that are not possible yet:

- using inherited init
- using variables in parent classes

In this paragraph we address these topics and solve them one by one.

Let's start with inherited init(). Given below example, there are two ways of calling inherited init:

```
class A
  init()
    print('A')
  end
end

class B(A)
  init()
    inherited init()
    print('B')
  end
end
```

```
class C(B)
  init()
    inherited()
    print('B')
  end
end
```

The two ways are with or without the keyword 'init':

- inherited init(params) or
- inherited(params), which is the short version.

Both are allowed and it only applies to init(). Inheriting other functions always need to present the name of the function to inherit. So, in the ParseInheritExpr method we have to cater for three situations, being:

- inherited <identifier>
- inherited (
- inherited init

```
  ErrIncorrectInherit = 'Incorrect inheritance expression.';
```

```
function TParser.ParseInheritedExpr: TExpr;
var
  Variable: TVariable;
  Token: TToken;
begin
  Variable := TVariable.Create(TIdent.Create(CurrentToken));
  Next;  // skip inherited
  if CurrentToken.Typ = ttIdentifier then
    Result := TInheritedExpr.Create(Variable, ParseIdent)
  else if CurrentToken.Typ in [ttInit, ttOpenParen] then begin
    if CurrentToken.Typ = ttOpenParen then
      with CurrentToken do
        Token := TToken.Create(ttIdentifier, 'init', Null, Line, Col)
    else begin
      Token := CurrentToken;
      Next;
    end;
    Result := TInheritedExpr.Create(Variable, TIdent.Create(Token));
  end
  else Error(CurrentToken, ErrIncorrectInherit);
end;
```

First, create a new TVariable from the current token 'inherited', which we'll treat as a variable. If the next token is an identifier, we create the TInheritedExpr with parameters Variable and parsing the identifier inside the call.

If, on the other hand, the token is either an 'init' or a '(', we have to deal with an inerited constructor. If the token is a '(' we create an 'init' token manually, otherwise the current token contains 'init'. Again, create the TInheritedExpr. It can be parsed in a smarter way, but I wanted to keep the normal identifier separated from the init.

Next we take up the reference to or usage of variables declared in a parent class from a subclass. So far, it's not possible to refer to a variable declared in the parent class.

For example, the following is not possible yet:

```
class Circle
  var radius := 1
  init(.radius)
    self.radius := radius
  end
end

class Sphere (Circle)
  func volume() => 4/3 * self.radius^3
  init(.radius)
    inherited(radius: radius)
  end
end
var sphere := Sphere(radius: 3)
print(sphere.volume())
```

An error message is generated stating that variable radius is unknown. It means class 'Sphere', or rather, instance 'sphere' does not have access to variable 'radius'. How to arrange that?

By far the easiest way is to copy the variables from the parent class to the subclass instance. This means not only their names, but also their initial values. Hey, we've done that before!

So, in order to arrange this we need access to a parent class and to its fields. We find all this nicely in the constructor of TGearClass.

```
constructor TGearClass.Create(AIdent: TIdent; AParent: Variant; Members: TMembers);
var
  ParentClass: TGearClass;
  i: Integer;
begin
  Ident := AIdent;
  Methods := Members.Methods;
  Fields := Members.Fields;
  Constants := Members.Constants;
  Parent := AParent;
  if not VarIsNull(Parent) then begin
    ParentClass := ICallable(Parent) as TGearClass;
    for i := 0 to ParentClass.Fields.Count-1 do
      Fields[ParentClass.Fields.Keys[i]] := ParentClass.Fields.Data[i];
    for i := 0 to ParentClass.Constants.Count-1 do
      Constants [ParentClass.Constants.Keys[i]] := ParentClass.Constants.Data[i];
  end;
end;
```

If the parent is assigned, we will copy one by one its fields and corresponding values to the fields of the subclass, thereby overwriting the newer declarations. Hm, do we want that? Or should we allow overriding variables in a subclass. If we allow overriding then change the code to the following:

```
...
  if not VarIsNull(Parent) then begin
    ParentClass := ICallable(Parent) as TGearClass;
    for i := 0 to ParentClass.Fields.Count-1 do begin
      if not Fields.Contains(ParentClass.Fields.Keys[i]) then
        Fields[ParentClass.Fields.Keys[i]] := ParentClass.Fields.Data[i];
    end;
    for i := 0 to ParentClass.Constants.Count-1 do begin
      if not Constants.Contains(ParentClass.Constants.Keys[i]) then
        Constants[ParentClass.Constants.Keys[i]] := ParentClass.Constants.Data[i];
    end;
  end;
...
```

This means that only if the variable doesn't exist in the subclass, we add the parent variable to the list of fields. We will not overwrite the subclass variable with the value of the variable in the parent class.

With this the above example now works. And even variables can be reintroduced in subclasses if needed.

```
class One
  var one := 1
  init(one)
    self.one := one
  end
end

class Two(One)
  var two := 2
  init(two)
    inherited (two)
    self.two := two
  end
end

var two := Two(3)
print(two.one) //3
print(two.two) //3
```

Invalid use of inherit

Final words on inherit.
Inherit can only be used in a subclass or child class. If a class doesn't have a parent then the inherited keyword should not be used. We don't test for that right now. Let's change that.

In the resolver add to type TClassKind the value ckSubClass:

```
  TClassKind = (ckNone, ckClass, ckSubClass);
```

Then, in visitor VisitClassDecl make the following small change:

```
procedure TResolver.VisitClassDecl(ClassDecl: TClassDecl);
var
  EnclosingClassKind: TClassKind;
begin
  Declare(ClassDecl.Ident);
  Enable(ClassDecl.Ident);
  EnclosingClassKind := CurrentClassKind;
  CurrentClassKind := ckClass;
  if Assigned(ClassDecl.Parent) then begin
    Visit(ClassDecl.Parent);
    CurrentClassKind := ckSubClass;
    if not ClassList.Contains(ClassDecl.Parent.Ident.Text) then
      with ClassDecl.Parent.Ident do
        Errors.Append(Token.Line, Token.Col, Format(ErrParentNoClass, [Text]));
    BeginScope;
    Scopes.Top.Enter(TSymbol.Create('inherited', Enabled));
  end;
  ClassList.Add(ClassDecl.Ident.Text);  // add this class to list
  ResolveClass(ClassDecl);
  if Assigned(ClassDecl.Parent) then EndScope;
  CurrentClassKind := EnclosingClassKind;
end;
```

And in visitor VisitInheritedExpr also add the check on ckSubClass:

```
  ErrInheritedNotInSubClass = 'Cannot use "inherited" in a class without parent class.';
```

```
procedure TResolver.VisitInheritedExpr(InheritedExpr: TInheritedExpr);
begin
  if CurrentClassKind = ckNone then
    with InheritedExpr do
      Errors.Append(Token.Line, Token.Col, ErrInheritedInClass)
  else if CurrentClassKind <> ckSubClass then
    with InheritedExpr do
      Errors.Append(Token.Line, Token.Col, ErrInheritedNotInSubClass);
  ResolveLocal(InheritedExpr.Variable);
end;
```

Ans this ends the chapter on inheritance.
Next, we focus on a new type: Values as calculated properties.

Chapter 8 – Value properties

A property, in some object-oriented programming languages, is a special sort of class member, intermediate in functionality between a field and a method. The field-like syntax is easier to read and write than lots of method calls, yet the interposition of method calls "under the hood" allows for data validation, active updating (e.g., of GUI elements), or implementation of what may be called "read-only fields".

Calculated values

Next to variable fields, there will be a new type of field, which I call calculated value fields. The difference with a variable (field) is, that you cannot reassign values to value fields, they can only return a value, and thus are read-only. They work as if they are constants. As an example consider the following:

> An expert is a man who has made all the mistakes which can be made, in a narrow field.
>
> — Niels Bohr

```
var radius := 2
val area
  return pi() * radius * radius
end
print(area)   // prints 12.56637…
```

In case we deal with only 1 simple return statement inside the value code, we can omit the 'return' and 'end' keywords, like this:

```
var radius := 2
val area := pi() * radius^2
print(area)   // prints 12.56637…
```

However, this is a forbidden assignment:

```
var radius := 2
val area := pi * radius * radius
area := 12  // error, assignment to value field not allowed.
```

The EBNF of a value declaration is:

```
ValDecl = 'val' Ident ( ':=' Expr | Block 'end' ) .
```

Note that in fact it is a parameterless function !

Let's look at the AST node of a value declaration:

```
  TValDecl = class(TDecl)
    private
      FFuncDecl: TFuncDecl;
    public
      property FuncDecl: TFuncDecl read FFuncDecl;
      constructor Create(AIdent: TIdent; AFuncDecl: TFuncDecl; AToken: TToken);
  end;
```

```
constructor TValDecl.Create(AIdent: TIdent; AFuncDecl: TFuncDecl;
  AToken: TToken);
begin
  Inherited Create(AIdent, dkVal, AToken);
  FFuncDecl := AFuncDecl;
end;
```

Here you see the functional character of a calculated value. Besides its Ident, it takes a function declaration. In fact, we create a function, which is called when a 'val' is used. We don't have a separate destructor, as function declaration already has one.

In TDecl add the declaration kind dkVal to the enum TDeclKind:

```
  TDecl = class(TNode)
    private
       type TDeclKind = (dkClass, dkFunc, dkVal, dkVar);
  ...
```

Then, moving on to the parser. First add ttVal to the declaration start set:

```
const
  DeclStartSet: TTokenTypSet = [ttClass, ttFunc, ttLet, ttVal, ttVar];
```

Next, add the case for ttVal to function ParseDecl:

```
function TParser.ParseDecl: TDecl;
const Mutable = True;
begin
  case CurrentToken.Typ of
    ttClass: Result := ParseClassDecl;
    ttFunc: Result := ParseFuncDecl(ffFunction);
    ttLet: Result := ParseVarDecl(Mutable = False);
    ttVal: Result := ParseValDecl;
    ttVar: Result := ParseVarDecl(Mutable = True);
  end;
end;
```

Note that I always put the token types in order of definition in uToken.pas.

Function ParseValDecl is as follows:

First, the identifier of the value declaration is parsed. It is then used to create a function declaration. The name of the function is the identifier of the value. If we encounter a ':=' token, we have to deal with a single return expression, and we manually create a block and insert a return statement. Otherwise, the body is parsed, after which an 'end' is expected. Finally, the TValDecl is returned.

```
function TParser.ParseValDecl: TDecl;
var
  Ident: TIdent;
  FuncDecl: TFuncDecl;
  Token: TToken;
begin
  Token := CurrentToken;
  Next; // skip val
  Ident := ParseIdent;
  FuncDecl := TFuncDecl.Create(Ident, Ident.Token);
  if CurrentToken.Typ = ttAssign then begin
    FuncDecl.Body := TBlock.Create(TNodeList.Create(), CurrentToken);
    FuncDecl.Body.Nodes.Add(ParseReturnStmt);
  end
  else begin
    FuncDecl.Body := ParseBlock;
    Expect(ttEnd);
  end;
  Result := TValDecl.Create(Ident, FuncDecl, Token);
end;
```

It's time for the visitors. Starting with the printer visitor for value declaration:

```
procedure TPrinter.VisitValDecl(ValDecl: TValDecl);
begin
  IncIndent;
  VisitNode(ValDecl);
  Visit(ValDecl.Ident);
  Visit(ValDecl.FuncDecl);
  DecIndent;
end;
```

In the Resolver we only call the node's function declaration:

```
procedure TResolver.VisitValDecl(ValDecl: TValDecl);
begin
  Visit(ValDecl.FuncDecl);
end;
```

Then, in the Interpreter, we need to add the visitor for VisitValDecl, but also change the VisitVariable visitor.

Method VisitValDecl needs to create a function declaration. First, we check for the name of the value already existing. We then create a new TVal with the node's FuncDecl into the current space, and finally store this function into the current space.

```
procedure TInterpreter.VisitValDecl(ValDecl: TValDecl);
var
  Value: TVal;
begin
  CheckDuplicate(ValDecl.Ident, 'Val');
  Value := TVal.Create(ValDecl.FuncDecl, CurrentSpace);
  CurrentSpace.Store(ValDecl.Ident, IValuable(Value));
end;
```

TVal is a new type, which is directly derived from TFunc, and which doesn't add any additional functionality. Add it just below the definition of TFunc:

```
  TVal = class(TFunc, IValuable)
    // this is empty
  end;
```

Also, a new interface is declared for this purpose: IValuable, which derives directly from ICallable.

```
  IValuable = interface(ICallable)
    ['{C5B7F7F5-3E31-AE0F-C7E2-FC4B56B77224}']
  end;
```

It's main purpose is to use it as an instance tester, like this: if VarSupports(Item, IValuable) then <do something>. Other than that, items of TVal will be part of a separate value list instead of the method table in class definitions. First, however, we work on the global 'val' declarations, not part of classes.

Currently, the variable's value is loaded from the current space, however in the case of a Value, we need to check if the current space's identifier looks up a TVal, and if so, execute (or call) the value's accompanying function.

We give an empty argument list as parameter, as a Value doesn't have any parameters.

```
function TInterpreter.VisitVariable(Variable: TVariable): Variant;
begin
  Result := Lookup(Variable);
  if VarSupports(Result, IValuable) then
    Result := IValuable(Result).Call(Variable.Token, Self, TArgList.Create());
  if VarIsEmpty(Result) then
    Raise ERuntimeError.Create(Variable.Token, ErrUndefVarOrSelfMissing);
end;
```

Compile and run this program. We are now able to create Value properties in the main program.

As an example consider the following input.

```
val Pi := pi()
```

This simply returns result of the function pi(). Building on this:

```
var radius := 2
val circleArea
  return Pi*radius^2
end
print(circleArea)   // prints 12.5663706143592
radius := 3
print(circleArea)   // prints 28.2743338823081
```

But also the following, now using the short ':=' notation:

```
var height := 8
val cylinderVolume := circleArea * height
print(cylinderVolume)   // prints 226.194671058465
height := 10
print(cylinderVolume)   //  282.743338823081
```

So, a 'val' can be used as a constant value, however keep in mind it always makes a function call. It's better to use let for that purpose. Val is like a dynamic constant.

A last one for now:

```
val diceRoll
  let rnd := randomLimit(6)+1
  let result := match rnd
    if 1 then 'One'
    if 2 then 'Two'
    if 3 then 'Three'
    if 4 then 'Four'
    if 5 then 'Five'
    if 6 then 'Six'
    else 'Wrong dice roll'

  return result
end

for var i := 0 where i < 10, i+=1 do
  print(diceRoll, terminator: ' | ')
end
```

it prints:

```
One | Five | Two | Five | Four | Two | Five | Three | Two | Six |
```

For this to be possible I used one of the standard functions, dealing with random numbers. This one accepts an integer parameter and returns an integer between 0 and the parameter, whereby the parameter value is not included, hence the use: `randomLimit(6)+1`

This makes sure a number from 1 to 6 is returned.

Since we took care of automatically putting the field Mutable to False, it is impossible to assign any value to a ‘val’ variable. It now shows an error: Cannot assign value to constant "diceRoll".

Class values

> The aim of education is the knowledge, not of facts, but of values.
> — William S. Burroughs

Next, we'll focus on Value fields inside classes. As an example consider the following:

```
val Pi := pi()
class Circle
  var radius := 1
  val area := Pi * self.radius^2
  init(.radius)
    self.radius := radius
  end
end

var circle := Circle(radius: 5)
print(circle.area)   // prints 78.5398163397448
```

Note that the Value Pi is defined here as a (sort of) constant value, however, be aware of the fact that when Pi is used as a reference variable, the full calculation is executed, meaning 1 function call for the inside call of Pi and 1 call to the standard function pi().

Recall that when a class declaration is parsed, it parses methods, fields and inits. Nested classes are not supported (yet). But let's add values to this:

```
const
  ClassStartSet: TTokenTypSet = [ttFunc, ttInit, ttLet, ttVal, ttVar];
```

```
function TParser.ParseClassDecl: TDecl;
var
  ...
begin
  ...
  while ClassStartSet.Contains(CurrentToken.Typ) do begin
    case CurrentToken.Typ of
      ...
      ttVal: DeclList.Add(ParseValDecl);
      ttVar: DeclList.Add(ParseVarDecl(True));
    end;
  end;
    ...
end;
```

Before we interpret the Class declaration, we first need to resolve its internals. So, let's start with extending VisitClassDecl in the Resolver:

```
procedure TResolver.ResolveClass(ClassDecl: TClassDecl);
...
begin
  ...
  for Decl in ClassDecl.DeclList do begin
    case Decl.Kind of
      dkFunc: begin
        ...
      end;
      dkVal: ResolveFunction((Decl as TValDecl).FuncDecl, fkMethod);
      dkVar: ...
  ...
end;
```

I only show the for-loop details as the rest remains unchanged. In fact, the only thing added was the case for 'Decl.Kind of dkVal'. As you see we call ResolveFunction for the function declaration part of a value declaration, and then treat it as a fkMethod.

Since a Value declaration doesn't have any parameters, the only actions in ResolveFunction needed are creating a new scope, resolve the function's body, and then close the scope again.

Next up is the interpreter's VisitClassDecl method, as we need to take care for Value declarations inside a class there as well. So far, we've already cleared normal methods and inits.

Before we go there, we need to add Values and its ValueTable to the TMember record in uMembers:

Add as a new type, and then add it to the record, the constructor and the init procedure:

```
  TValueTable = TMemberTable;

  TMembers = record
    ...
    Values: TValueTable;
    constructor Create(AFields: TFieldTable; AConstants: TConstTable;
      AMethods: TMethodTable; AValues: TValueTable);
    ...
  end;

constructor TMembers.Create(AFields: TFieldTable; AConstants: TConstTable;
  AMethods: TMethodTable; AValues: TValueTable);
begin
    ...
  Values := AValues;
end;

procedure TMembers.Init;
begin
    ...
  Self.Values := TValueTable.Create;
end;
```

With this we can process it in the interpreter's getMembers method:

```
function TInterpreter.getMembers(DeclList: TDeclList): TMembers;
var
  ...
  Value: TVal;
begin
  ...
  for Decl in DeclList do begin
    case Decl.Kind of
      dkFunc: begin
        ...
      end;
      dkVal: begin
        Value := TVal.Create((Decl as TValDecl).FuncDecl, CurrentSpace);
        Result.Values[Decl.Ident.Text] := IValuable(Value);
      end;
      dkVar:
        ...
    end;
  end;
end;
```

New is the Value variable. The case clause now contains the instance comparison for dkVal. If a value declaration is found, we extract it's function declaration, and from that create a new TVal in the current space, much the same as for the functions above. However, instead of adding it to the method table we add them to the Values table.

Finally, we add the Values variable in the call to the constructor of TGearClass. This means we have to change this constructor in unit uClass.pas.

```
  TGearClass = class(TInterfacedObject, ICallable)
    private
      ...
      Values: TValueTable;
    public
      ...
  end;
```

And the implementation of the constructor now assigns the values.

```
constructor TGearClass.Create(AIdent: TIdent; AParent: Variant; Members: TMembers);
begin
  ...
  Values := Members.Values;
  ...
end;
```

Finally, we need to make a subtile adjustment in the Interpreter to method VisitGetExpr.

```
function TInterpreter.VisitGetExpr(GetExpr: TGetExpr): Variant;
var
  Instance: Variant;
begin
  Instance := Visit(GetExpr.Instance);
  if VarSupports(Instance, IGearInstance) then begin
    Result := IGearInstance(Instance).GetMember(GetExpr.Ident);
    if VarSupports(Result, IValuable) then
      Result := ICallable((IValuable(Result) as TVal)
                          .Bind(Instance))
                          .Call(GetExpr.Token, Self, TArgList.Create());
  end
  else
    Raise ERuntimeError.Create(GetExpr.Ident.Token, ErrExpectedClassInstance);
end;
```

As we read the value of a field, we now need to check if it is a value (if VarSupports(Result, IValuable) then…). If it is, we actually have to call it's underlying function. That means first binding it to the instance, followed by a call with an empty argument list. Remember, Values have no arguments.

Last, but not least, we need a small change to the method GetMember of TGearInstance, in order to find the right Value when used.

Put it just below the method search:

```
function TGearInstance.GetMember(Ident: TIdent): Variant;
...
begin
  ...
  if GearClass.Values.Contains(Ident.Text, Index) then
    Exit(GearClass.Values.At(Index));

  Raise ERuntimeError.Create(Ident.Token,
    'Undefined class member "' + Ident.Text + '".');
end;
```

This is it! Test it and see how it works ☺

For example, I tried the below input program.

```
class Date
  var day := 1
  var month := 'Jan'
  var year := 1980
  init(.day, .month, .year)
    self.day := day
    self.month := month
    self.year := year
  end
  func toString()
    return '' + self.day + '-' + self.month + '-' + self.year
  end
end

var currentDate := Date(day: 01, month: 'April', year: 2018)

class Person
  var name := ''
  var birthDate := Date()
  init(name, .birthDate)
    self.name := name
    self.birthDate := birthDate
  end
  val age := currentDate.year - self.birthDate.year
  func toString()
    return self.name + ': ' + self.birthDate.toString()
  end
end

val Harry := Person('Harry',
               birthDate: Date(day: 23, month: 'February', year: 1987))
print(Harry.toString())
print(Harry.age)
```

This results in the following output:

```
Harry: 23-February-1987
31
```

The last item of this paragraph is the reference to and usage of value properties declared in parent classes. For this we only need two minor changes, looking a lot like things we already did before.

First, have a look at the way how we found methods in parent classes, using the FindMethod function.

We will have a similar approach to finding value properties, as you'll remember, they are similar to functions.

```
function TGearInstance.GetMember(Ident: TIdent): Variant;
var
  ...
  Value: IValuable;
begin
  ...
  Value := GearClass.FindValue(Ident.Text);
  if Value <> Nil then
    Exit(Value);

  Raise ...
  ...
end;
```

Finally, add the following public function to TGearClass:

```
function TGearClass.FindValue(const AName: String): IValuable;
var
  Index: LongInt = -1;
begin
  Result := Nil;
  if Values.Contains(AName, Index) then
    Result := Values.At(Index)
  else if not VarIsNull(Parent) then
    Result := (ICallable(Parent) as TGearClass).FindValue(AName);
end;
```

It searches the tree of parents to find a value property, and if found returns the correct (first found) property. It is not possible to use 'inherited' with value properties!

Test it, and see how good it works ☺

```
class Circle
  var radius := 1
  val area := pi() * self.radius^2
  init(.radius)
    self.radius := radius
  end
end

class Cylinder(Circle)
  var height := 0
  val volume := self.area * self.height
  init(.radius, .height)
    inherited(radius: radius)
    self.height := height
  end
end

var circle := Circle(radius: 1)
var cylinder := Cylinder(radius: 2, height: 10)
```

```
print(circle.area)
print(cylinder.area)
print(cylinder.height)
print(cylinder.volume)
print(cylinder.radius)
```
Resulting in:

```
3.14159265358979
12.5663706143592
10
125.663706143592
2
```

Final epic is to make sure we cannot re-assign to value fields in classes. Hereto we change method VisitSetStmt in the Interpreter. Also, add the error message.

```
  ErrAssignToValue = 'Assignment to Value property not allowed.';
```

Here's thew complete VisitSetStmt visitor:

```
procedure TInterpreter.VisitSetStmt(SetStmt: TSetStmt);
var
  Instance, OldValue, NewValue, Value: Variant;
begin
  Instance := Visit(SetStmt.Instance);
  if not VarSupports(Instance, IGearInstance) then
    Raise ERuntimeError.Create(SetStmt.Token, ErrExpectedClassInstance);

  OldValue := IGearInstance(Instance).GetMember(SetStmt.Ident);
  if VarSupports(OldValue, IValuable) then
    Raise ERuntimeError.Create(SetStmt.Token, ErrAssignToValue);

  if IGearInstance(Instance).isConstant(SetStmt.Ident) and
     (not VarIsNull(OldValue)) then
    Raise ERuntimeError.Create(SetStmt.Ident.Token, Format(
      ErrClassMemberImmutable, [SetStmt.Ident.Text]));

  NewValue := Visit(SetStmt.Expr);
  Value := getAssignValue(OldValue, NewValue, SetStmt.Ident.Token, SetStmt.Op);
  IGearInstance(Instance).SetField(SetStmt.Ident, Value);
end;
```

After the check on the instance, we also check if the old value is of type IValuable. If so, then we try to assign to a value property, which is not allowed and we raise a runtime error.

This is it for value properties.

Chapter 9 – Extensions

In computing, a plug-in (or plugin, add-in, addin, add-on, addon, or extension) is a software component that adds a specific feature to an existing computer program. When a program supports plug-ins, it enables customization.

Applications support plug-ins for many reasons. Some of the main reasons include (WikiPedia):

- to enable third-party developers to create abilities which extend an application
- to support easily adding new features
- to reduce the size of an application
- to separate source code from an application because of incompatible software licenses.

In Gear, we call the plug-in an 'extension'. An extension is defined using the keyword 'extension', followed by the name of the type that is extended. For example:

```
extension Array
  func toString() => self.toString()
  val count := length(self)
end
```

Functionality defined in an extension immediately becomes available for all variables that were declared of the respective type, e.g. Array. Only func's and val's can be used in an extension. All user defined types can have extensions.

A func or val defined in an extension overwrites earlier defined ones.

> There is a wide, yawning black infinity. In every direction, the extension is endless; the sensation of depth is overwhelming. And the darkness is immortal. Where light exists, it is pure, blazing, fierce; but light exists almost nowhere, and the blackness itself is also pure and blazing and fierce.
>
> — Carl Sagan

Let's look at the AST node:

```
  TExtensionDecl = class(TDecl)
    private
      FDeclList: TDeclList;
    public
      property DeclList: TDeclList read FDeclList;
      constructor Create(AIdent: TIdent; ADeclList: TDeclList; AToken: TToken);
      destructor Destroy; override;
  end;
```

An extension has an identifier (part of TDecl) and a set of declarations. In TDecl, add the enum dkExtension:

```
  TDecl = class(TNode)
    private
       type TDeclKind = (dkClass, dkExtension, dkFunc, dkVal, dkVar);
  ...
```

The implementation of TExtensionDecl:

```
constructor TExtensionDecl.Create
  (AIdent: TIdent; ADeclList: TDeclList; AToken: TToken);
begin
  inherited Create(AIdent, dkExtension, AToken);
  FDeclList := ADeclList;
end;

destructor TExtensionDecl.Destroy;
begin
  if Assigned(FDeclList) then FDeclList.Free;
  inherited Destroy;
end;
```

The parser function is quite straightforward:

```
function TParser.ParseExtensionDecl: TDecl;
var
  Token: TToken;
  Extension: TExtensionDecl;
begin
  Token := CurrentToken;
  Next;  // skip extension
  Extension := TExtensionDecl.Create(ParseIdent, TDeclList.Create(false), Token);
  while CurrentToken.Typ in DeclStartSet do
    if CurrentToken.Typ in [ttFunc, ttVal] then
      Extension.DeclList.Add(ParseDecl)
    else
      Error(CurrentToken, Format(ErrUnallowedDeclIn, ['extension']));
  Expect(ttEnd);
  Result := Extension;
end;
```

As mentioned an extension can only have function and/or value declarations. If any other declaration type is defined, an error is generated. If you want to add a new field for example, the usual way is to create a subclass from the original class and then add the field.

Also, add the new error message:

```
  ErrUnallowedDeclIn = 'Unallowed declaration in "%s".';
```

Add ttExtension to the DeclStartSet and the parse function to ParseDecl:

```
const
  DeclStartSet: TTokenTypSet = [ttClass, ttExtension, ttFunc,
    ttLet, ttVal, ttVar];

function TParser.ParseDecl: TDecl;
const Mutable = True;
begin
  case CurrentToken.Typ of
    ttClass: Result := ParseClassDecl;
    ttExtension: Result := ParseExtensionDecl;
    ...
  end;
end;
```

The printer visitor:

```
procedure TPrinter.VisitExtensionDecl(ExtensionDecl: TExtensionDecl);
var
  Decl: TDecl;
begin
  IncIndent;
  VisitNode(ExtensionDecl);
  Visit(ExtensionDecl.Ident);
  for Decl in ExtensionDecl.DeclList do
    Visit(Decl);
  DecIndent;
end;
```

The resolver method uses a new class kind: ckExtension:

```
  TClassKind = (ckNone, ckClass, ckSubClass, ckExtension);
```

Since the extended types create new scopes for their internal declarations, we do the same:

```
procedure TResolver.VisitExtensionDecl(ExtensionDecl: TExtensionDecl);
var
  Decl: TDecl;
  EnclosingClassKind: TClassKind;
begin
  EnclosingClassKind := CurrentClassKind;
  CurrentClassKind := ckExtension;
  BeginScope;
  Scopes.Top.Enter(TSymbol.Create('self', Enabled));
  for Decl in ExtensionDecl.DeclList do begin
    case Decl.Kind of
      dkFunc: ResolveFunction(Decl as TFuncDecl, fkMethod);
      dkVal: ResolveFunction((Decl as TValDecl).FuncDecl, fkMethod);
    end;
  end;
  EndScope;
  CurrentClassKind := EnclosingClassKind;
end;
```

The interpreter visitor is a bit more complicated. We first need to retrieve the declared type that the extension belongs to. So, we load it from the current space and if it supports one of the types, it's a valid identifier. After this we store all func's and val's in member tables.

```
procedure TInterpreter.VisitExtensionDecl(ExtensionDecl: TExtensionDecl);
var
  TypeDecl: Variant;
  Members: TMembers;
begin
  TypeDecl := CurrentSpace.Load(ExtensionDecl.Ident);  // get actual type
  if not VarSupportsIntf(TypeDecl, [IClassable]) then
    Raise ERuntimeError.Create(ExtensionDecl.Ident.Token,
      Format(ErrExtIdNoType, [ExtensionDecl.Ident.Text]));
  Members := getMembers(ExtensionDecl.DeclList);
  if VarSupports(TypeDecl, IClassable) then
    IClassable(TypeDecl).ExtendWith(Members);
end;
```

Finally, based on the type that is supported by the interface, we call the respective extend method. This in fact means we add those methods to the existing types, moreover we overwrite any

methods or values with the same name. This is important to note! In an extension you are allowed to overwrite an existing method!

As a side note, in the future when we add Arrays, Dictionaries, Enums, etc, this visitor must be updated, so for example, suppose we Enums, then the call to VarSupportsIntf() must include IEnumable and finally add something like this:

```
  if VarSupports(TypeDecl, IEnumable) then
    IEnumable(TypeDecl).ExtendWith(Members);
```

And this must be done for all new types!

To get this to work we have to add the mentioned function ExtendWith to the interfaces and classes.

Add uMembers to the uses clause in uClassIntf.pas:

```
uses
  Classes, SysUtils, uCallable, uAST, uMembers;
```

Add the method to the respective interface:

```
  IClassable = interface(ICallable)
    ['{230E841B-D57B-F5F8-4DD3-224B74645A17}']
    procedure ExtendWith(Members: TMembers);
  end;
```

Next, implement the 'public' method in TGearClass:

```
procedure TGearClass.ExtendWith(Members: TMembers);
var
  i: Integer;
begin
  for i := 0 to Members.Methods.Count-1 do
    Methods[Members.Methods.Keys[i]] := Members.Methods.Data[i];
  for i := 0 to Members.Values.Count-1 do
    Values[Members.Values.Keys[i]] := Members.Values.Data[i];
end;
```

Remember, for each and every type this has to be done. However, it's just a copy and paste action, which is not really a big deal.

Chapter 10 – Traits

A trait is a concept used in object-oriented programming, which represents a set of methods that can be used to extend the functionality of a class. Traits both provide a set of methods that implement behaviour to a class, and require that the class implement a set of methods that parameterize the provided behaviour. For inter-object communication, traits are somewhat between an object-oriented protocol (interface) and a mixin. An interface may define one or more behaviors via method signatures, while a trait defines behaviors via full method definitions: i.e., it includes the body of the methods. In contrast, mixins include full method definitions and may also carry state through member variable, while traits usually don't (WikiPedia).

Parsing and resolving traits

The traits in Gear are as described above, and they follow these set of rules:

- they have reusable functions,
- a class may contain zero or many traits,
- traits can use other traits, so that the functions defined in a trait become part of the new trait,
- redefined functions result in a collission.

> The moment a person forms a theory, his imagination sees in every object only the traits which favor that theory.
> — Thomas Jefferson

Given the above the EBNF of a trait declaration is:

```
TraitDecl = 'trait' Ident [':' Traits ] [ DeclList ] 'end'  .
Traits = ExprList .
ExprList = Expr {',' Expr} .
DeclList = Decl {',' Decl} .
```

Based on this, create a declaration type in the AST. Remember the Ident is already in TDecl.

```
  TTraitDecl = class(TDecl)
    private
      FTraits: TExprList;
      FDeclList: TDeclList;
    public
      property Traits: TExprList read FTraits;
      property DeclList: TDeclList read FDeclList;
      constructor Create(AIdent: TIdent; ATraits: TExprList;
        ADeclList: TDeclList; AToken: TToken);
      destructor Destroy; override;
  end;
```

Like we did for other type declarations, we include a property DeclList, which will only hold functions. Traits don't carry state, so we'll not use variables.

Here's the implementation:

```
constructor TTraitDecl.Create
  (AIdent: TIdent; ATraits: TExprList; ADeclList: TDeclList; AToken: TToken);
begin
  inherited Create(AIdent, dkTrait, AToken);
  FTraits := ATraits;
  FDeclList := ADeclList;
end;

destructor TTraitDecl.Destroy;
begin
  if Assigned(FDeclList) then FDeclList.Free;
  if Assigned(FTraits) then FTraits.Free;
  inherited Destroy;
end;
```

Make sure to add dkTrait to the enum TDeclKind in class TDecl:

```
  TDecl = class(TNode)
    private
       type TDeclKind =
         (dkClass, dkExtension, dkFunc, dkTrait, dkVal, dkVar);
```

Next, we have to amend the class declaration, so that it can handle traits. The new class EBNF becomes:

```
ClassDecl = 'class' Ident ['('Parent')'] [':' Traits] { Decl } 'end'.
```

For example:

```
class Car(Vehicle): Stringable, Drivable
end
```

```
  TClassDecl = class(TDecl)
    private
      ...
      FTraits: TExprList;
    public
      ...
      property Traits: TExprList read FTraits;
      constructor Create(AIdent: TIdent; AParent: TVariable; ATraits: TExprList;
        ADeclList: TDeclList; AToken: TToken);
      ...
  end;
```

Also amend the constructor and destructor.

```
constructor TClassDecl.Create(AIdent: TIdent; AParent: TVariable;
  ATraits: TExprList; ADeclList: TDeclList; AToken: TToken);
begin
  ...
  FTraits := ATraits;
end;
```

```
destructor TClassDecl.Destroy;
begin
  ...
  if Assigned(FTraits) then FTraits.Free;
  inherited Destroy;
end;
```

We'll now have a look at the parser functions ParseTraitDecl and also at the changes needed for ParseClassDecl.

```
function TParser.ParseTraitDecl: TDecl;
var
  Token: TToken;
  Ident: TIdent;
  Traits: TExprList;
  DeclList: TDeclList;
begin
  Token := CurrentToken;
  Next; // skip trait
  Ident := ParseIdent;
  Traits := TExprList.Create();
  if CurrentToken.Typ = ttColon then begin
    Next; // skip :
    Traits := ParseExprList;
  end;
  DeclList := TDeclList.Create();
  while CurrentToken.Typ in DeclStartSet do begin
    if CurrentToken.Typ = ttFunc then
      DeclList.Add(ParseDecl)
    else
      Error(CurrentToken, Format(ErrUnallowedDeclIn, ['trait']));
  end;
  Expect(ttEnd);
  Result := TTraitDecl.Create(Ident, Traits, DeclList, Token);
end;
```

Note that only functions and value properties are allowed. Any other declaration type will result in a parse error.

Add token ttTrait to the declaration start set and add the parse function to ParseDecl.

```
const
  DeclStartSet: TTokenTypSet = [ttClass, ttExtension, ttFunc,
    ttLet, ttVal, ttVar, ttTrait];

function TParser.ParseDecl: TDecl;
...
begin
  case CurrentToken.Typ of
    ...
    ttTrait: Result := ParseTraitDecl;
  end;
end;
```

Next, we change the parse function for class declaration, so that it also can include traits.

```
function TParser.ParseClassDecl: TDecl;
var
  ...
  Traits: TExprList;
begin
  ...
  end;
  Traits := TExprList.Create();
  if CurrentToken.Typ = ttColon then begin
    Next; // skip :
    Traits := ParseExprList;
  end;
  DeclList := TDeclList.Create(false);
  ...
  Result := TClassDecl.Create(Ident, Parent, Traits, DeclList, Token);
  Expect(ttEnd);
end;
```

In ParseClassDecl, we parse the traits the same way as we did for ParseTraitDecl. Add the traits to the class declaration constructor.

The routines for the printer for Traits and changed Class:

```
procedure TPrinter.VisitTraitDecl(TraitDecl: TTraitDecl);
var
  Decl: TDecl;
  Trait: TExpr;
begin
  IncIndent;
  VisitNode(TraitDecl);
  Visit(TraitDecl.Ident);
  WriteLn(Indent + 'Traits:');
  for Trait in TraitDecl.Traits do
    Visit(Trait);
  WriteLn(Indent + 'Declarations:');
  for Decl in TraitDecl.DeclList do
    Visit(Decl);
  DecIndent;
end;
```

```
procedure TPrinter.VisitClassDecl(ClassDecl: TClassDecl);
var
  Decl: TDecl;
  Trait: TExpr;
begin
  IncIndent;
  VisitNode(ClassDecl);
  Visit(ClassDecl.Ident);
  if Assigned(ClassDecl.Parent) then
    Visit(ClassDecl.Parent);
  WriteLn(Indent + 'Traits:');
  for Trait in ClassDecl.Traits do
    Visit(Trait);
  WriteLn(Indent + 'Declarations:');
  for Decl in ClassDecl.DeclList do
    Visit(Decl);
  DecIndent;
end;
```

Moving on to the resolver, first add ckTrait to the enum TClassKind:

```
  TClassKind = (ckNone, ckClass, ckSubClass, ckEnum, ckExtension, ckTrait);
```

Then, create the resolver visitor, much the same as the one for extensions.

```
procedure TResolver.VisitTraitDecl(TraitDecl: TTraitDecl);
var
  Decl: TDecl;
  EnclosingClassKind: TClassKind;
  Trait: TExpr;
begin
  Declare(TraitDecl.Ident);
  Enable(TraitDecl.Ident);
  EnclosingClassKind := CurrentClassKind;
  CurrentClassKind := ckTrait;
  for Trait in TraitDecl.Traits do
    Visit(Trait);
  BeginScope;
  Scopes.Top.Enter(TSymbol.Create('self', Enabled));
  for Decl in TraitDecl.DeclList do begin
    if Decl.Kind = dkFunc then
      ResolveFunction(Decl as TFuncDecl, fkMethod);
  end;
  EndScope;
  CurrentClassKind := EnclosingClassKind;
end;
```

In function ResolveClass, we add just before BeginScope, the line to visit all traits:

```
procedure TResolver.ResolveClass(ClassDecl: TClassDecl);
var
  ...
  Trait: TExpr;
begin
  for Trait in ClassDecl.Traits do
    Visit(Trait);
  BeginScope;
  ...
  EndScope;
end;
```

Last bit to solve in the resolver is to disallow to call inherited from a trait:

```
procedure TResolver.VisitInheritedExpr(Node: TInheritedExpr);
begin
  if CurrentClassKind in [ckNone, ckTrait] then
    with InheritedExpr do
      Errors.Append(Token.Line, Token.Col, ErrInheritedInClass)
  ...
end;
```

Interpreting traits

The final step to take is to interpret the traits. For this we create a new interface ITraitable in uClassIntf.pas and a new class TGearTrait in uClass.

> I think the most important trait for an entrepreneur is persistence. When you try to do something new and difficult, you are more likely to fail than to succeed.
>
> — Trip Adler

In uCallable add the new interface:

```
  ITraitable = interface
    ['{4CEE3AA1-2D73-E3C0-DA25-6C6E52BCD1D1}']
    function getIdent: TIdent;
    function getMethods: TMethodTable;
    property Ident: TIdent read getIdent;
    property Methods: TMethodTable read getMethods;
  end;
```

Then, in uClass, add the following new class:

```
  TGearTrait = class(TInterfacedObject, ITraitable)
    private
      FIdent: TIdent;
      FMethods: TMethodTable;
      function getIdent: TIdent;
      function getMethods: TMethodTable;
    public
      property Ident: TIdent read getIdent;
      property Methods: TMethodTable read getMethods;
      constructor Create(AIdent: TIdent; Members: TMembers);
      function toString: String; override;
  end;
```

```
{ TGearTrait }

function TGearTrait.getIdent: TIdent;
begin
  Result := FIdent;
end;

function TGearTrait.getMethods: TMethodTable;
begin
  Result := FMethods;
end;

constructor TGearTrait.Create(AIdent: TIdent; Members: TMembers);
begin
  FIdent := AIdent;
  FMethods := Members.Methods;
end;

function TGearTrait.toString: String;
begin
  Result := FIdent.Text;
end;
```

Again, we use a TMembers record as the vehicle to transport methods and values. It's sort of a dressed down copy of TGearClass.

In the visitor for the interpreter we first store the identifier, after checking for duplication. Then function ApplyTraits returns the list of all traits used by this trait. Also, a trait can have one or more traits attached to it. ApplyTraits check for any collissions (i.e. duplications) and reports an error if this is the case. Next, the trait members are combined with the new trait members (methods and values), to produce one set of functionality available in the trait.

```
procedure TInterpreter.VisitTraitDecl(TraitDecl: TTraitDecl);
var
  Members, Traits: TMembers;
  Trait: TGearTrait;
begin
  CheckDuplicate(TraitDecl.Ident, 'Trait');
  CurrentSpace.Store(TraitDecl.Ident, Nil);
  Traits := ApplyTraits(TraitDecl.Traits);
  Members := CombineMembers(Traits, TraitDecl.DeclList);
  Trait := TGearTrait.Create(TraitDecl.Ident, Members);
  CurrentSpace.Update(TraitDecl.Ident, ITraitable(Trait));
end;
```

```
function TInterpreter.ApplyTraits(Traits: TExprList): TMembers;
var
  Trait: TExpr;
  TraitValue: Variant;
  GearTrait: ITraitable;
  i: Integer;
  Name: String;
begin
  Result.Init;
  for Trait in Traits do begin
    TraitValue := Visit(Trait);
    if not VarSupports(TraitValue, ITraitable) then
      Raise ERuntimeError.Create(Trait.Token,
        Format(ErrNotDeclaredAsTrait, [Trait.Token.Lexeme]));
    GearTrait := ITraitable(TraitValue);
    for i := 0 to GearTrait.Methods.Count-1 do begin
      Name := GearTrait.Methods.Keys[i];
      if Result.Methods.Contains(Name) then
        Raise ERuntimeError.Create(GearTrait.Ident.Token,
          Format(ErrTraitIsDeclared, ['function', Name]));
      Result.Methods[Name] := GearTrait.Methods[Name];
    end;
  end;
end;
```

Function ApplyTraits walks through all traits declared in the traits list. It first checks if the identifier is actually a trait (varsupports ITraitable), and if not raises a runtime error. Next, we cast the trait to a ITraitable interface object and go through both methods and values to check for any collission. If it doesn't collide, the method or value is added to the member list.

After we've succesfully retrieved a list of all trait functions in a TMembers record, we need to combine it with the new list of functions defined in the new trait. Function CombineMembers takes care of that.

Since we need to apply the same functionality to a class with its members, we also include the combination of class members such as Values, Fields and Constants.

If any of the class or trait declarations collide with an existing identifier in the list of traits a runtime error is generated.

```
function TInterpreter.CombineMembers(Traits: TMembers; DeclList: TDeclList): TMembers;
var
  Decl: TDecl;
  Func: TFunc;
  Value: TVal;
begin
  Result := Traits;
  for Decl in DeclList do begin
    if Traits.Methods.Contains(Decl.Ident.Text) then
      Raise ERuntimeError.Create(Decl.Ident.Token,
        Format(ErrTraitDeclared, [Decl.Ident.Text]));
    case Decl.Kind of
      dkFunc: begin
        Func := TFunc.Create(Decl as TFuncDecl, CurrentSpace);
        Result.Methods[Decl.Ident.Text] := ICallable(Func);
      end;
      dkVal: begin
        Value := TVal.Create((Decl as TValDecl).FuncDecl, CurrentSpace);
        Result.Values[Decl.Ident.Text] := IValuable(Value);
      end;
      dkVar:
        if (Decl as TVarDecl).Mutable then
          Result.Fields[Decl.Ident.Text] := Visit((Decl as TVarDecl).Expr)
        else
          Result.Constants[Decl.Ident.Text] := Visit((Decl as TVarDecl).Expr);
    end;
  end;
end;
```

I chose here to only allow functions in traits and not values, though that should in theory be allowed as well. It just makes the checking on collissions a bit easier right now and in principle functions can cover all functionality needed.

Finally, we apply the traits and combine them with the existing class members in the visitor for the class declaration, much the same as for the trait declaration.

```
procedure TInterpreter.VisitClassDecl(ClassDecl: TClassDecl);
var
  GearClass: TGearClass;
  Members, Traits: TMembers;
  Parent: Variant;
begin
  CheckDuplicate(ClassDecl.Ident, 'Class');
  CurrentSpace.Store(ClassDecl.Ident, Nil);
  if Assigned(ClassDecl.Parent) then begin
    Parent := Visit(ClassDecl.Parent);
    CurrentSpace := TMemorySpace.Create(CurrentSpace);
    CurrentSpace.Store('inherited', Parent);
  end
  else Parent := Null;
  Traits := ApplyTraits(ClassDecl.Traits);
  Members := CombineMembers(Traits, ClassDecl.DeclList);
  if Assigned(ClassDecl.Parent) then
    CurrentSpace := CurrentSpace.EnclosingSpace;
  GearClass := TGearClass.Create(ClassDecl.Ident, Parent, Members);
  CurrentSpace.Update(ClassDecl.Ident, IClassable(GearClass));
end;
```

Instance is class

> Scientific thought, then, is not momentary; it is not a static instance; it is a process.
> — Jean Piaget

In many languages you can test whether an instance belongs to a certain class, for example in Java, the function instanceOf() returns True if such is the case. In Object Pascal the operator 'is' is used for this.

We will use the keyword 'is' as well, for example:

```
trait Instance
  func instanceOf(Class)
    return self is Class
  end
end

class Number: Instance
  var code := 0
  init(code)
    self.code := code
  end
  func write()
    print(self.code)
  end
end

var number := Number(10)

if number.instanceOf(Number) then
  print(True)
end
```

This example also shows how a trait can be used. For it to be reusable, it should have few, if none, dependencies with the classes that are using it. You can use 'self' in a trait, which always points to the class instance that implements the trait.

Let's see how we can implement 'is'. The token type and keyword are already declared in uToken.pas. In the parser amend function isRelOp() so that it also includes 'is'.

```
function TParser.isRelOp: Boolean;
begin
  Result := CurrentToken.Typ in [ttEQ, ttNEQ, ttGT, ttGE, ttLT, ttLE, ttIs];
  //                              =     <>    >     >=     <    <=    is
end;
```

Then, in the interpreter, change visitor VisitBinaryExpr to include the case for ttIs:

```
function TInterpreter.VisitBinaryExpr(BinaryExpr: TBinaryExpr): Variant;
...
begin
  ...
  case BinaryExpr.Op.Typ of
    ...
    ttIs:   Result := TMath._Is(Left, Right, Op);
  end;
end;
```

Add uClassIntf to the uses clause in uMath.pas:

```
uses
  Classes, SysUtils, uToken, uError, math, Variants, uClassIntf;
```

Next, in uMath.pas, add the function _Is:

```
class function TMath._Is(const Left, Right: Variant; Op: TToken): Variant;
begin
  if oneOfBothNull(Left, Right) then
    Exit(False);
  if VarSupports(Left, IGearInstance) and VarSupports(Right, IClassable) then
    Exit(IGearInstance(Left).ClassName = IClassable(Right).Name);
  Raise ERuntimeError.Create(Op, Format(ErrIncompatibleOperands, ['is']));
end;
```

As you can see, the left argument must be a IGearInstance, while the right argument must be a IClassable interface. If this is the case we cast the left argument to IGearInstance and retrieve its class name. The right argument is cast to IClassable and we retrieve the (class) name. If these are the same we have a match, and we can say the instance belongs to the class.

To finalize we extend the IGearInstance and IClassable interfaces with the functionality to retrieve the names:

```
  IClassable = interface(ICallable)
    ...
    function getName: String;
    property Name: String read getName;
  end;

  IGearInstance = Interface
    ...
    function getClassName: String;
    property ClassName: String read getClassName;
  end;
```

These functions have to be implemented:

Function getName for TGearClass:

```
function TGearClass.getName: String;
begin
  Result := Ident.Text;
end;
```

Function getClassName for TGearInstance:

```
function TGearInstance.getClassName: String;
begin
  Result := GearClass.Ident.Text;
end;
```

And this is enough to make it happen. Try and test it!

Chapter 11 – Arrays

Wikipedia says: an array is a systematic arrangement of similar objects, usually in rows and columns. An array is a data structure consisting of a collection of elements (values or variables), each identified by one array index. The simplest type of data structure is a linear array, called one-dimensional array.

In Gear, an Array is extremely flexible and it's not limited to just one element type. So, as an example, an array can be:

```
['+', '-', '*', '/', '%']
[1,2,3,4,5]
['Hello', 'world', "!, 3.1415]
```

In fact you can put anything in an array, as long as it's an expression. That means, even this is an array:

```
[x=>x+x, x=>x-x, x=>x*x, x=>x/x, x=>x%x]
```

Array definition

An array can be defined in several ways. An array can be defined like we did for class definition, creating an array-type so to speak. We can also declare an array variable directly. And for both ways, we can enforce type consistency, if we want to create an array of a specific defined type.

> We face a wide array of threats, which means we have to have a wide array of capabilities.
>
> — Mac Thornberry

Array type declaration

The EBNF for a basic array type declaration is as follows:

```
ArrayDecl = 'array' Ident '[' [ ExprList ] ']' [ DeclList ] 'end' .
```

As you can see, when you define an array type, it's also possible to define declarations such as functions and values. Variables and other declarations such as classes are not allowed.

Arrays come with a few standard instance fields, for instance 'count', which returns the number of elements in the array.

We create an array class and an array instance class in a new unit.

But first, let's work on the AST node for TArrayDecl.

```
  TArrayDecl = class(TDecl)
    private
      FElements: TExprList;
      FDeclList: TDeclList;
    public
      property Elements: TExprList read FElements;
      property DeclList: TDeclList read FDeclList;
      constructor Create(AIdent: TIdent; ADeclList: TDeclList; AToken: TToken);
      destructor Destroy; override;
      procedure AddElement(Expr: TExpr);
  end;
```

In TDecl, add the new enum dkArray to TDeclKind:

```
  TDecl = class(TNode)
    private
       type TDeclKind =
         (dkArray, dkClass, dkExtension, dkFunc, dkTrait, dkVal, dkVar);
```

Here's the implementation for TArrayDecl:

```
constructor TArrayDecl.Create
  (AIdent: TIdent; ADeclList: TDeclList; AToken: TToken);
begin
  inherited Create(AIdent, dkArray, AToken);
  FElements := TExprList.Create();
  FDeclList := ADeclList;
end;

destructor TArrayDecl.Destroy;
begin
  if Assigned(FDeclList) then FDeclList.Free;
  FElements.Free;
  inherited Destroy;
end;

procedure TArrayDecl.AddElement(Expr: TExpr);
begin
  FElements.Add(Expr);
end;
```

We will add type enforcement in a later stage.

Now add ttArray to the DeclStartSet and ParseDecl:

```
const
  DeclStartSet: TTokenTypSet = [ttArray, ttClass, ttExtension, ttFunc,
    ttLet, ttVal, ttVar, ttTrait];

function TParser.ParseDecl: TDecl;
const Mutable = True;
begin
  case CurrentToken.Typ of
    ttArray: Result := ParseArrayDecl;
    ttClass: Result := ParseClassDecl;
...
```

The parse function is as follows:

```
function TParser.ParseArrayDecl: TDecl;
var
  Token: TToken;
  ArrayDecl: TArrayDecl;
begin
  Token := CurrentToken;
  Next;  // skip array
  ArrayDecl := TArrayDecl.Create(ParseIdent, TDeclList.Create(false), Token);
  Expect(ttOpenBrack);
  if CurrentToken.Typ <> ttCloseBrack then
    ArrayDecl.Elements.Assign(ParseExprList);
  Expect(ttCloseBrack);
  // other declarations: val and func
  while CurrentToken.Typ in DeclStartSet do
    if CurrentToken.Typ in [ttFunc, ttVal] then
      ArrayDecl.DeclList.Add(ParseDecl)
    else
      Error(CurrentToken, Format(ErrUnallowedDeclIn, ['array']));
  Expect(ttEnd);
  Result := ArrayDecl;
end;
```

Again, see how it follows the EBNF. First, the array keyword, after which we create the array declaration. The ident is parsed directly as an argument to the constructor. Next, we expect the mandatory [and], and in between them we parse any expression. In case there are func and val definitions we parse them and close off with the 'end'. Note that the array can be empty!

The printer visitor is as follows:

```
procedure TPrinter.VisitArrayDecl(ArrayDecl: TArrayDecl);
var
  Expr: TExpr;
  Decl: TDecl;
begin
  IncIndent;
  VisitNode(ArrayDecl);
  Visit(ArrayDecl.Ident);
  WriteLn(Indent + 'Elements:');
  IncIndent;
  for Expr in ArrayDecl.Elements do
    Visit(Expr);
  for Decl in ArrayDecl.DeclList do
    Visit(Decl);
  DecIndent;
  DecIndent;
end;
```

For the resolver visitor, first add a new class kind 'ckArray' to TClassKind:

```
  TClassKind = (ckNone, ckClass, ckSubClass, ckExtension, ckTrait, ckArray);
```

Then, the resolver visitor becomes:

```
procedure TResolver.VisitArrayDecl(ArrayDecl: TArrayDecl);
var
  Expr: TExpr;
  Decl: TDecl;
  EnclosingClassKind: TClassKind;
begin
  Declare(ArrayDecl.Ident);
  Enable(ArrayDecl.Ident);
  for Expr in ArrayDecl.Elements do
    Visit(Expr);
  EnclosingClassKind := CurrentClassKind;
  CurrentClassKind := ckArray;
  BeginScope;
  Scopes.Top.Enter(TSymbol.Create('self', Enabled));
  for Decl in ArrayDecl.DeclList do begin
    case Decl.Kind of
      dkFunc: ResolveFunction(Decl as TFuncDecl, fkMethod);
      dkVal: ResolveFunction((Decl as TValDecl).FuncDecl, fkMethod);
    end;
  end;
  EndScope;
  CurrentClassKind := EnclosingClassKind;
end;
```

We declare and enable the array identifier, visit the elements, then set the classkind to ckArray. Next open a new scope, insert 'self' and visit any func or val. Finally, close the scope and set the class kind back to the original.

To make sure we don't use inherited in a function inside an array declaration, we need a small adjustment to VisitInheritedExpr:

```
procedure TResolver.VisitInheritedExpr(InheritedExpr: TInheritedExpr);
begin
  if CurrentClassKind in [ckNone, ckArray] then
  ...
end;
```

Moving on to the interpreter, we have to consider a number of things. First, we have to take care of visiting the array expressions. Since they are expressions, they might contain anything, including variables, calculations, function declarations, class instances, etc. We store these in a variable of type TArrayElements (defined in a few moments…).

Next, we have to deal with the func and val declarations, and copy them over to the ArrayClass declaration.

We also take the array elements to the ArrayClass.

Finally, we store the ArrayClass as an IArrayable interface into the current space.

```
procedure TInterpreter.VisitArrayDecl(ArrayDecl: TArrayDecl);
var
  ArrayClass: TArrayClass;
  Expr: TExpr;
  Members: TMembers;
  Elements: TArrayElements;
begin
  CheckDuplicate(ArrayDecl.Ident, 'Array');
  Elements := TArrayElements.Create;
  for Expr in ArrayDecl.Elements do
    Elements.Add(Visit(Expr));
  Members := getMembers(ArrayDecl.DeclList);
  ArrayClass := TArrayClass.Create(
    ArrayDecl.Ident, Elements, Members);
  CurrentSpace.Store(ArrayDecl.Ident, IArrayable(ArrayClass));
end;
```

We are not done yet with the interpreter, as we'll have to change VisitCallExpr and VisitGetExpr as well later on. First, we create two new interfaces and a new unit, and work on the internals of TArrayClass.

Create and add the following interfaces to new unit uArrayIntf:

```
unit uArrayIntf;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, uCallable, uAST, Variants, uMembers, uCollections;

type

  IArrayable = Interface(ICallable)
    ['{1A6EC506-9318-F69B-0563-830506C351F7}']
    procedure ExtendWith(Members: TMembers);
  end;

implementation

end.
```

IArrayable inherits from ICallable, because an instance of an array is called like a function:

```
array Numbers
  [0,1,2,3,4,5,6,7,8,9]

  func write()
    print(self)
  end
end
var numbers := Numbers()
```

Create a new unit uArray.pas:

```
unit uArray;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, uAST, uError, uToken, Variants, uClass, uInterpreter,
  uCallable, uArrayIntf, uMembers;

type

  TArrayClass = class(TInterfacedObject, IArrayable)
    private
      Ident: TIdent;
      Elements: TArrayElements;
      Methods: TMethodTable;
      Values: TValueTable;
      function FindMethod(Instance: IArrayInstance; const AName: String): ICallable;
      function FindValuable(const AName: String): IValuable;
    public
      constructor Create(AIdent: TIdent; AElements: TArrayElements;
        Members: TMembers);
      function Call(Token: TToken; Interpreter: TInterpreter;
        ArgList: TArgList): Variant;
      function toString: string; override;
      procedure ExtendWith(Members: TMembers);
  end;

implementation
uses uFunc , uVariantSupport;

function TArrayClass.FindMethod(Instance: IArrayInstance; const AName: String
  ): ICallable;
var
  Index: integer = -1;
begin
  Result := Nil;
  if Methods.Contains(AName, Index) then
    Result := (ICallable(Methods.At(Index)) as TFunc).Bind(Instance)
end;

function TArrayClass.FindValuable(const AName: String): IValuable;
var
  Index: LongInt = -1;
begin
  Result := Nil;
  if Values.Contains(AName, Index) then
    Result := Values.At(Index);
end;

constructor TArrayClass.Create
  (AIdent: TIdent; AElements: TExprList; Members: TMembers);
begin
  Ident := AIdent;
  Elements := AElements;
  Methods := Members.Methods;
  Values := Members.Values;
end;
```

So far so different from the TGearClass situation. Now we get to the more interesting part, the Call function:

```
function TArrayClass.Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList
  ): Variant;
var
  Instance: IArrayInstance;
begin
  Instance := IArrayInstance(TArrayInstance.Create(Self));
  for i := 0 to Self.Elements.Count-1 do
    Instance.Elements.Add(Self.Elements[i]);
  Result := Instance;
end;
```

In the call we create a new instance of the array via IArrayInstance. Then all elements from the array type are copied to the instance, which now owns a copy and can do anything with them. Finally, the instance is returned.

```
function TArrayClass.toString: string;
begin
  Result := Ident.Text;
end;

procedure TArrayClass.ExtendWith(Members: TMembers);
var
  i: Integer;
begin
  for i := 0 to Members.Methods.Count-1 do
    Methods[Members.Methods.Keys[i]] := Members.Methods.Data[i];
  for i := 0 to Members.Values.Count-1 do
    Values[Members.Values.Keys[i]] := Members.Values.Data[i];
end;

end.
```

It is all comparable to the way a class was created. After this, we'll insert the Array instance class and implementation. Add TArrayInstance:

```
  TArrayInstance = class(TInterfacedObject, IArrayInstance)
    private
      ArrayClass: TArrayClass;
      FElements: TArrayElements;
      function getCount: LongInt;
      function getElements: TArrayElements;
      function getItem(i: integer): Variant;
      procedure setItem(i: integer; AValue: Variant);
    public
      property Elements: TArrayElements read getElements;
      property Count: LongInt read getCount;
      property Items[i:integer]: Variant read getItem write setItem; default;
      constructor Create(AnArrayClass: TArrayClass);
      destructor Destroy; override;
      function toString: string; override;
      function GetMember(Ident: TIdent): Variant;
  end;
```

Note the usage of the new elements type TArrayElements. This and some functions and properties are now part of the interface IArrayInstance. Add to uArrayIntf.pas:

```
  TArrayElements = specialize TArray<Variant>;
  IArrayInstance = Interface
    ['{5C2DF1A9-88BC-8FFE-B6F2-6C3DCF535601}']
    function GetMember(Ident: TIdent): Variant;
    function getElements: TArrayElements;
    function getCount: LongInt;
    property Elements: TArrayElements read getElements;
    property Count: LongInt read getCount;
    function getItem(i: integer): Variant;
    procedure setItem(i: integer; AValue: Variant);
    property Items[i:integer]: Variant read getItem write setItem; default;
  end;
```

The array TArrayElements contains the elements after they are visited in the interpreter. Remember that array Elements contained the pure expressions. The new array contains the visited expressions.

And the implementation of TArrayInstance:

```
{ TArrayInstance }

function TArrayInstance.getCount: LongInt;
begin
  Result := FElements.Count;
end;

function TArrayInstance.getElements: TArrayElements;
begin
  Result := FElements;
end;

function TArrayInstance.getItem(i: integer): Variant;
begin
  Result := FElements[i];
end;

procedure TArrayInstance.setItem(i: integer; AValue: Variant);
begin
  FElements[i] := AValue;
end;

constructor TArrayInstance.Create(AnArrayClass: TArrayClass);
begin
  ArrayClass := AnArrayClass;
  FElements := TArrayElements.Create;
end;

destructor TArrayInstance.Destroy;
begin
  FElements.Free;
  inherited Destroy;
end;
```

The GetMember function is comparable with the one used in TGearInstance, except for the fields and constants, which are not allowed in Arrays.

```
function TArrayInstance.GetMember(Ident: TIdent): Variant;
var
  Method: ICallable;
  Valuable: IValuable;
begin
  Method := ArrayClass.FindMethod(IArrayInstance(Self), Ident.Text);
  if Method <> Nil then
    Exit(Method);

  Valuable := ArrayClass.FindValuable(Ident.Text);
  if Valuable <> Nil then
    Exit(Valuable);

  Raise ERuntimeError.Create(Ident.Token,
    'Undefined array member "' + Ident.Text + '".');
end;
```

The toString() function should print the contents of the array (note that this requires unit uVariantSupport to be included in the uses clause of uArray):

```
implementation
uses uFunc, uVariantSupport;
```

```
function TArrayInstance.toString: string;
var
  i: Integer;
begin
  Result := '[';
  if FElements.Count > 0 then begin
    for i := 0 to FElements.Count-2 do
      Result += FElements[i].toString + ', ';
    Result += FElements[FElements.Count-1].toString;
  end;
  Result += ']';
end;
```

The last element is stringyfied separately, in order not to get an additional comma behind it!
Don't forget to add the uses clause 'uArray' and 'uArrayIntf' to the implementation of uInterpreter:

```
implementation
uses uCallable, uFunc, uStandard, uClassIntf, uClass, uArrayIntf, uArray,
  uMath, uVariantSupport;
```

Note: uMath moved from top of unit to implementation section.
Next, the promised few changes to the interpreter's VisitCallExpr and VisitGetExpr.

```
function TInterpreter.VisitCallExpr(Node: TCallExpr): Variant;
...
begin
  ...
  if VarSupportsIntf(Callee, [ICallable, IClassable, IArrayable]) then
    Func := ICallable(Callee)
  else begin
  ...
end;
```

We need to check if Callee is IArrayable, and if so convert it to ICallable. This is possible since IArrayable is descendent of ICallable.

Next, we add the IArrayInstance to VisitGetExpr to be able to have access to some val's belonging to to the array instance.

```
function TInterpreter.VisitGetExpr(GetExpr: TGetExpr): Variant;
var
  Instance: Variant;
begin
  Instance := Visit(GetExpr.Instance);
  if VarSupportsIntf(Instance, [IGearInstance, IArrayInstance]) then begin
    if VarSupports(Instance, IGearInstance) then
      Result := IGearInstance(Instance).GetMember(GetExpr.Ident)
    else if VarSupports(Instance, IArrayInstance) then
      Result := IArrayInstance(Instance).GetMember(GetExpr.Ident);

    if VarSupports(Result, IValuable) then
      Result := ICallable((IValuable(Result) as TVal)
                          .Bind(Instance))
                          .Call(GetExpr.Token, Self, TArgList.Create());
  end
  else
    Raise ERuntimeError.Create(GetExpr.Ident.Token, ErrExpectedInstance);
end;
```

I changed the GetExpr visitor such that we can check for multiple interfaces and then retrieve the right member into Result. Next, we check if it is a Valuable and call that if so.

I changed error message ErrExpectedClassInstance into a more generic ErrExpectedInstance:

```
  ErrExpectedInstance = 'Expected declared type instance.';
```

The length function

Obviously, it's important to know the length of an array, or count the number of elements. For this purpose we'll extend the standard function, to return the number of elements of an array. Next to this it also returns the length of a string.

In unit uStandard add unit uArrayIntf to the uses clause in the implementation section so that you get:

```
implementation
uses uArrayIntf;
```

Then, change the class TLength Call function:

```
function TLength.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Value: Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 1);
  Value := ArgList[0].Value;
  if VarSupports(Value, IArrayInstance) then
    Result := IArrayInstance(Value).Count
  else if VarIsStr(Value) then
    Result := Length(Value)
  else
    Raise ERuntimeError.Create(Token,
      'Length function not possible for this type.');
end;
```

It accepts only 1 argument, which is either an array or a string value.

If it is an array instance, we retrieve the count of the number of elements in the list, and if it is a string we just return it's length. If it is any other value, it returns an error.

Now it's possible to create below code:

```
var s := '123456789'
var l := length(s)
print(l)

array Chars
  ["a, "b, "c]
  val count := length(self)
end
var chars := Chars()
print(chars.count)
```

Array declaration by expression

Of course it's nice to be able to create array types first and then declare a variable for this type. However, most of the time you just want to create a variable as an array instance on the fly, like this:

```
var a := [0,1,2,3,4,5,6,7,8,9]
var c := ["a, "b, "c, "d, "e, "f, "g]
```

In this case the array is declared as an anonymous array instance. The accompanying AST node is:

```
  TArrayDeclExpr = class(TFactorExpr)
    private
      FArrayDecl: TArrayDecl;
    public
      property ArrayDecl: TArrayDecl read FArrayDecl;
      constructor Create(AnArrayDecl: TArrayDecl);
      destructor Destroy; override;
  end;


constructor TArrayDeclExpr.Create(AnArrayDecl: TArrayDecl);
begin
  inherited Create(AnArrayDecl.Token);
  FArrayDecl := AnArrayDecl;
end;

destructor TArrayDeclExpr.Destroy;
begin
  if Assigned(FArrayDecl) then FArrayDecl.Free;
  inherited Destroy;
end;
```

In the parser, create a new parse function to parse the array expression. The expression list is parsed through function ParseExprList and then assigned in one go to the elements of the array. Function Assign is a standard function in the unit FGL, part of Free Pascal.

```
function TParser.ParseArrayDeclExpr: TExpr;
var
  ArrayDecl: TArrayDecl;
begin
  ArrayDecl := TArrayDecl.Create(Nil, TDeclList.Create(false), CurrentToken);
  Next; // skip [
  if CurrentToken.Typ <> ttCloseBrack then
    ArrayDecl.Elements.Assign(ParseExprList);
  Expect(ttCloseBrack);
  Result := TArrayDeclExpr.Create(ArrayDecl);
end;
```

Since it's an anonymous declaration, the first parameter of TArrayDecl, a TIdent, is Nil. We create trhe ArrayDecl with an empty list of declarations.

Next, we skip the opening '[', after which we parse all expressions. Finally, we expect a closing ']'.

In function ParseFactor add the case for ttOpenBrack:

```
function TParser.ParseFactor: TExpr;
begin
  case CurrentToken.Typ of
    ...
    ttOpenBrack: Result := ParseArrayDeclExpr;
    else
      Error(CurrentToken,'Unexpected token: ' + CurrentToken.toString + '.');
  end;
end;
```

In the printer first create a visitor for TArrayDeclExpr:

```
procedure TPrinter.VisitArrayDeclExpr(ArrayDeclExpr: TArrayDeclExpr);
begin
  IncIndent;
  VisitNode(ArrayDeclExpr);
  Visit(ArrayDeclExpr.ArrayDecl);
  DecIndent;
end;
```

and a small change to VisitArrayDecl, in order to check for the array ident:

```
procedure TPrinter.VisitArrayDecl(ArrayDecl: TArrayDecl);
...
begin
  ...
  if Assigned(ArrayDecl.Ident) then
    Visit(ArrayDecl.Ident);
  ...
end;
```

The resolver changes a bit more, as we don’t have to care about func and val declarations:

```
procedure TResolver.VisitArrayDecl(ArrayDecl: TArrayDecl);
var
  Expr: TExpr;
  Decl: TDecl;
  EnclosingClassKind: TClassKind;
begin
  for Expr in ArrayDecl.Elements do
    Visit(Expr);
  if Assigned(ArrayDecl.Ident) then begin
    Declare(ArrayDecl.Ident);
    Enable(ArrayDecl.Ident);
    EnclosingClassKind := CurrentClassKind;
    CurrentClassKind := ckArray;
    BeginScope;
    Scopes.Top.Enter(TSymbol.Create('self', Enabled));
    for Decl in ArrayDecl.DeclList do begin
      case Decl.Kind of
        dkFunc: ResolveFunction(Decl as TFuncDecl, fkMethod);
        dkVal: ResolveFunction((Decl as TValDecl).FuncDecl, fkMethod);
      end;
    end;
    EndScope;
    CurrentClassKind := EnclosingClassKind;
  end;
end;
```

And then add:

```
procedure TResolver.VisitArrayDeclExpr(ArrayDeclExpr: TArrayDeclExpr);
begin
  Visit(ArrayDeclExpr.ArrayDecl);
end;
```

In the interpreter we have to create the anonymous array class AND create the instance in one go. That's done by creating the class and call it directly like this:

```
function TInterpreter.VisitArrayDeclExpr(ArrayDeclExpr: TArrayDeclExpr): Variant;
var
  ArrayExpr: IArrayable;
  Members: TMembers;
  Elements: TArrayElements;
  Expr: TExpr;
begin
  Members.Init;
  Elements := TArrayElements.Create;
  for Expr in ArrayDeclExpr.ArrayDecl.Elements do
    Elements.Add(Visit(Expr));
  ArrayExpr := IArrayable(TArrayClass.Create(Nil, Elements, Members));
  Result := ArrayExpr.Call(ArrayDeclExpr.Token, Self, TArgList.Create());
end;
```

What happens here? We first create an empty Members record, followed by visiting all expressions and storing them in an array of TArrayElements.

Then we create an anonymous array class. Only the array's elements and empty members are passed. The arrayclass is cast to the IArrayable interface. From the IArrayable ArrayExpr we do an immediate call. Of course the arguments are empty, as we don't have any parameters. As this is always part of a variable or value declaration, the memory location where the array instance is stored can be retrieved via the variable name.

With this we can now use it as follows:

```
var numbers := [0,1,2,3,4,5,6,7,8,9]
print(numbers)

var tokens := ['+', '-', '*', '/', '%']
print(tokens)
```

Alternative creation of an array

With a simple adjustment to the Call method we can have alternative ways of creating and filling an array. Change method Call to the following:

```
function TArrayClass.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Instance: IArrayInstance;
  i: Integer;
begin
  Instance := IArrayInstance(TArrayInstance.Create(Self));
  if ArgList.Count = 0 then
    for i := 0 to Self.Elements.Count-1 do
      Instance.Elements.Add(Self.Elements[i])
  else if ArgList.Count > 1 then
    for i := 0 to ArgList.Count-1 do
      Instance.Elements.Add(ArgList[i].Value)
  else if VarSupports(ArgList[0].Value, IArrayInstance) then
    Instance.Elements.Assign(IArrayInstance(ArgList[0].Value).Elements)
  else
    Raise ERuntimeError.Create(Token,
      'Array member of this type not allowed.');

  Result := Instance;
end;
```

It states that when the argument list in the call is empty, we take the elements as defined in the array definition. However, if there are any arguments, we'll take these as the contents for the array, and we discard the defined ones.

Then, there is a difference between 1 and more arguments. If there's exactly one argument than this must be an array expression. If there are more, it will take all values and create an array expression. Examples:

Create a generic type Array:

```
array Array
  []  // empty, no elements
  val count := length(self)
end

var a := Array([1,2,3,4,5]) // call with array expression
print(a)
print(a.count)

var b := Array(["a, "b, "c]) // array expression
print(b)
print(b.count)
var c := Array(b)  // create a copy

var d := Array(pi(), 'Hello', "z, 42)  // call with multiple arguments
print(d)
print(d.count)

var e := Array(["a, "b, "c], [1 ,2, 3, [4, 5]], pi())    // multiple nested arguments
print(e)
print(e.count)
```

This way you can define your own master array type and include all functionality you want.

Math on arrays

> To solve math problems, you need to know the basic mathematics before you can start applying it.
> — Catherine Asaro

Arrays wouldn't be useful if you couldn't do calculations with them. For example, the following operations I would like to add:

- concatenation, e.g. [a, b, c] >< [d, e, f] results in [a, b, c, d, e, f]. Here we introduce a new operator '><'. In some languages the '+' operator is used for this, however that's not the right way.
- addition, e.g. [1, 2, 3] + 1 results in [2, 3, 4], and [1, 2, 3] + [3, 2, 1] results in [4, 4, 4].
- multiplication, e.g. [1, 2, 3] * 10 results in [10, 20, 30].
- subtraction and negation
- division
- comparison of arrays
- check if an element is in an array

The nice thing about our array system is that you can create nested arrays, or multiple dimension arrays known as matrices. For example it's possible to create the following matrices:

```
var m := [[1,2,3],
          [4,5,6],
          [7,8,9]]

var n := [[3,2,1],
          [6,5,4],
          [9,8,7]]
```

After this paragraph it will be possible to do all kinds of math on arrays and matrices, such as:

```
m><n // concatenation of arrays
m+n or m+value  // addition of 2 equal sized array or with a value
m*n or m*value  // multiplication of 2 equal sized arrays or with a value
m-n or m-value  // subtraction of 2 equal sized arrays or from a value
m/n or m/value  // division of 2 equal sized arrays of by a value
-m              // negate an array
m=n or m<>n     // compare 2 equally sized arrays on contents
m::n            // dot product
```

As an example, consider the following array calculations on above defined arrays:

```
m><n:
[[1,2,3],[4,5,6],[7,8,9],[3,2,1],[6,5,4],[9,8,7]]
m+n:
[[4,4,4],[10,10,10],[16,16,16]]
m*n:
[[3,4,3],[24,25,24],[63,64,63]]
m*4:
[[4,8,12],[16,20,24],[28,32,36]]
```

Another example, multiplication of a matrix and a vector:

```
var A := [[1,2,3],
          [4,5,6],
          [7,8,9]]
var V := [10,20,30]
print(A*V)
```

results in:

```
[[10,20,30],[80,100,120],[210,240,270]]
```

It's important two know if two variables/expressions are an array. This can be checked with the following class function in TMath in unit uMath:

```
class function TMath.areBothArray(const Left, Right: Variant): Boolean;
begin
  Result := VarSupports(Left, IArrayInstance) and
            VarSupports(Right, IArrayInstance);
end;
```

Add uArrayIntf to the uses clause in uMath.pas:

```
uses
  Classes, SysUtils, uToken, uError, math, Variants, uClassIntf, uArrayIntf;
```

In the next paragraphs, we'll discuss how to implement the math for above mentioned calculations.

Concatenation

For concatenation I want to introduce a new token type: '><', which seems a logical symbol for it. It results in an expression where the left and right operand are brought together. Add the token to the list of token types, like this:

```
  TTokenTyp = (
    //Expressions - operators
    ...
    ttShl, ttShr, ttPow, ttConcat,
    ...
  );

function TTokenTypHelper.toString: string;
begin
  case Self of
    ...
    ttPow : Result := '^';
    ttConcat: Result := '><';
    ...
  end;
end;
```

In the Lexer, we must read the token and return ttConcat:

```
procedure TLexer.ScanToken(const Line, Col: Integer);
...
begin
  case FLook of
    ...
    '>' : case FReader.PeekChar of
            ...
            '<' : begin FLook := getChar; AddToken(ttConcat, '><'); end;
            else AddToken(ttGT, '>');
          end;
    ...
  end;
end;
```

Next, in the parser, add the token to the checking method isAddOp. We treat it as an addition operator in terms of precedence. This means you even can do calculations like this:

```
[1,2,3]*2 >< [12,8,4]/2 resulting in: [2,4,6,6,4,2]
```

```
function TParser.isAddOp: Boolean;
begin
  Result := CurrentToken.Typ in [ttPlus, ttMin, ttOr, ttXOr, ttConcat];
  //                              +       -      |     ~       ><
end;
```

In the interpreter, we'll add the case for ttConcat to the visitor for binary expressions:

```
function TInterpreter.VisitBinaryExpr(BinaryExpr: TBinaryExpr): Variant;
...
begin
  ...
  case BinaryExpr.Op.Typ of
    ...
    ttPow:  Result := TMath._Pow(Left, Right, Op);
    ttConcat: Result := TMath._Concat(Left, Right, Op);
    ...
  end;
end;
```

Finally, the method for the actual concatenation in unit uMath is:

```
class function TMath._Concat(const Left, Right: Variant; Op: TToken): Variant;
var
  newArray, leftArray, rightArray: IArrayInstance;
begin
  if not areBothArray(Left, Right) then
    Raise ERuntimeError.Create(Op, ErrConcatNotAllowed);

  leftArray := IArrayInstance(Left);
  rightArray := IArrayInstance(Right);
  if not sameArrayTypes(leftArray, rightArray) then
    Raise ERuntimeError.Create(Op, ErrArrayWrongTypes);

  newArray := makeArray(leftArray.TypeName, Op);
  newArray.Elements.Assign(leftArray.Elements);
  newArray.Elements.Concat(rightArray.Elements);
  Result := newArray;
end;
```

First check if both variables are an array. If so, next we have to check whether the two arrays belong to the same array type. Then create a new array variable, based on Left's array type, using helper function makeArray. Sequentially add all elements of the left and right array to the new array and return that value.

The accompanying error messages:

```
      ErrConcatNotAllowed = 'Both types must be array for concat operation.';
      ErrArrayWrongTypes = 'Both variables must be array of same type.';
```

In function _Concat we use a standard function of unit FGL: .Assign, which assigns the elements of one array to another. The Concat method is not defined in FGL so, we have to create it ourselves. It is available in uCollections.pas:

```
procedure TArray.Concat(withArray: TArray);
var
  i: Integer;
begin
  for i := 0 to withArray.Count-1 do
    Self.Add(withArray[i]);
end;
```

The above mentioned helper function makeArray takes as argument the type name of the left array, to retrieve the array's type. For this we need to add the function TypeName to IArrayIndstance:

```
  IArrayInstance = Interface
    ...
    function TypeName: String;
  end;
```

Next, implement it in TArrayInstance:

```
function TArrayInstance.TypeName: String;
begin
  Result := ArrayClass.Ident.Text;
end;
```

Finally, the function makeArray can be added as either a private function to TMath or just be defined as independent function in the implementation section (which I did):

```
function makeArray(TypeName: string; Token: TToken): IArrayInstance;
var
  ArrayType: IArrayable;
begin
  ArrayType := IArrayable(
    Language.Interpreter.Memory.Load(TypeName, Token));
  Result := IArrayInstance(TArrayInstance.Create(ArrayType as TArrayClass));
end;
```

The function loads the value of the array type from the interpreter's Memory (CurrentSpace), and then creates a new array instance where the array type is cast to TArrayClass.

You'll need to add unit uLanguage to the uses clause in the implementation section of uMath.

```
implementation
uses uArray, uLanguage;
```

Also, in the interpreter, add a new property Memory, as follows:

```
  TInterpreter = class(TVisitor)
    private
      CurrentSpace: TMemorySpace;
    public
      Globals: TMemorySpace;
      property Memory: TMemorySpace read CurrentSpace;
      ...
  end;
```

Now the function sameArrayTypes(), which simply compares the type names:

```
class function TMath.sameArrayTypes(const Left, Right: IArrayInstance): Boolean;
begin
  Result := Left.TypeName = Right.TypeName;
end;
```

By the way, now that we have the type name of an array, and we print the contents of an array, we should know of which type it is. This can be easilly done by amending the TArrayInstance.toString method, like this:

```
function TArrayInstance.toString: string;
var
  i: Integer;
begin
  Result := TypeName + ' [';
  for i := 0 to FElements.Count-2 do
    Result += FElements[i].toString + ', ';
  Result += FElements[FElements.Count-1].toString;
  Result += ']';
end;
```

The following code shows the result:

```
array Numbers [1,2,3,4,5] end
var n := Numbers()
print(n)  // prints Numbers [1, 2, 3, 4, 5]
var a := [1,2,3,4,5]
print(a)  // prints Array [1, 2, 3, 4, 5]
```

However, I chose not to implement this in my version of Gear!

While we're at it, we might as well introduce the ConcatIs, or '><=' operator. It makes appending to an array much easier. Instead of doing:

```
list := list >< [item]
```

It's easier to do:

```
list ><= [item]
```

Add the token ttConcatIs to uToken, right after ttConcat. Also add it to TTokenTypHelper.toString:

```
    ttConcatIs: Result := '><=';
```

In the Lexer change procedure ScanToken, so that the case for '>' becomes:

```
    '>' : case FReader.PeekChar of
            '>' : begin FLook := getChar; AddToken(ttShr, '>>'); end;
            '=' : begin FLook := getChar; AddToken(ttGE, '>='); end;
            '<' : begin
                    FLook := getChar;
                    if FReader.PeekChar = '=' then begin
                      FLook := getChar; AddToken(ttConcatIs, '><=');
                    end
                    else AddToken(ttConcat, '><');
                  end
            else AddToken(ttGT, '>');
          end;
```

Then, in the parser, add ttConcatIs to the constant AssignSet:

```
const
  AssignSet: TTokenTypSet =
    [ttPlusIs, ttMinIs, ttMulIs, ttDivIs, ttRemIs, ttConcatIs, ttAssign];
```

Finally, in th interpreter, amend function getAssignValue, so that it includes TMath,_Concat:

```
  if not VarIsNull(OldValue) then begin
    if Op.Typ <> ttAssign then
    case Op.Typ of
      ttPlusIs:   NewValue := TMath._Add(OldValue, NewValue, Op);
      ttMinIs:    NewValue := TMath._Sub(OldValue, NewValue, Op);
      ttMulIs:    NewValue := TMath._Mul(OldValue, NewValue, Op);
      ttDivIs:    NewValue := TMath._Div(OldValue, NewValue, Op);
      ttRemIs:    NewValue := TMath._Rem(OldValue, NewValue, Op);
      ttConcatIs: NewValue := TMath._Concat(OldValue, NewValue, Op);
    end;
    Exit(NewValue);
  end;
```

That's it.

Examples:

```
array Number [] end

var a := Number(1,2,3)
var b := Number([4,5,6])
print(a><b) // [1, 2, 3, 4, 5, 6]

var c := Number(a)
c ><= b
print(c) // [1, 2, 3, 4, 5, 6]
```

Note, that it doesn't work (yet) for anonymous arrays. But we'll solve that later, when they become sort of anonymous :-)

Addition

There are some different types of addition with regards to arrays. Arrays, if they have the same size can be added to each other, a vector can be added to a matrix array, and a number or string can be added. In case of strings the normals calculation rules apply. Strings can be added to strings, and numbers can be added to strings (but not the other way around). So, these expressions are legal:

```
[1,2,3] + [4,5,6]
['a','b','c'] + 4
[[1,2,3],[2,3,4],[3,4,5]] + [4,5,6]
```

The implementation of addition constitutes two new addition functions and a change to function _Add.

Working top to bottom, we first change _Add:

```
class function TMath._Add(const Left, Right: Variant; Op: TToken): Variant;
begin
  ...
  if areBothArray(Left, Right) then
    Exit(addTwoArrays(Left, Right, Op));
  if VarSupports(Left, IArrayInstance) then
    Exit(addToArray(Left, Right, Op));
  Raise ERuntimeError.Create(Op, Format(ErrWrongTypes, ['+']));
end;
```

We check if Left and Right ar both of type array, and if so, execute the new function 'addTwoArrays'. If not both Left and Right are arrays we check if Left is an array, and if so execute function 'addToArray'.

```
class function TMath.addTwoArrays(const Left, Right: IArrayInstance; Op: TToken):Variant;
var
  i: Integer;
  newArray: IArrayInstance;
begin
  if not sameArrayTypes(Left, Right) then
    Raise ERuntimeError.Create(Op, ErrArrayWrongTypes);
  if Left.Count <> Right.Count then
    Raise ERuntimeError.Create(Op, ErrArrayMismatchElem);

  newArray := makeArray(Left.TypeName, Op);
  for i := 0 to Left.Count-1 do
    newArray.Elements.Add(_Add(Left[i], Right[i], Op));

  Result := newArray;
end;
```

The items of Right are added to the items of Left. Note that we call _Add() for this. This is a recursive call, so that it's possible to include nested or multidimensional arrays. Of course it's only allowed to add two arrays together if they are from the same type. So, for example:

```
array Numbers [1,2,3,4,5] end
var n := Numbers()
var a := [1,2,3,4,5]
print(n=a)    // prints "False"
```

Arrays 'n' and 'a' are not the same!

This one adds a number or string to an array. Note that the array must be on the left side of the expression and the number or string to add on the right side. So, it is allowed to do ["a, "b] + "c (which is ['ac', 'bc']), but not allowed to do: "c + ["a, "b].

```
class function TMath.addToArray(
  const Left: IArrayInstance; Right: Variant; Op: TToken): Variant;
var
  i: Integer;
  newArray: IArrayInstance;
begin
  newArray := makeArray(Left.TypeName, Op);
  for i := 0 to Left.Count-1 do
    newArray.Elements.Add(_Add(Left[i], Right, Op));

  Result := newArray;
end;
```

First create a new empty array of type Left.TypeName. We walk trough all elements of the input array and while doing the addition, add the result to the new array. Finally, the new array is returned as the result.

Do add the following error message to the list of constant messages.

```
    ErrArrayMismatchElem = 'Mismatch in number of array elements in array operation.';
```

Add the functions to the interface:

```
    class function addTwoArrays(
      const Left, Right: IArrayInstance; Op: TToken):Variant; static;
    class function addToArray(
      const Left: IArrayInstance; Right: Variant; Op: TToken): Variant; static;
```

Example:

```
array Number [] end

var a := Number(1,2,3)
var b := Number([4,5,6])
print(a+b)  // [5, 7, 9]

var c := Number(a)
c += b
print(c)  // [5, 7, 9]
print(c+10)  // [15, 17, 19]
```

Subtraction

In fact, for subtraction, similar routines apply, so without explanation they follow here.

```
class function TMath._Sub(const Left, Right: Variant; Op: TToken): Variant;
begin
  ...
  if areBothArray(Left, Right) then
    Exit(subTwoArrays(Left, Right, Op));
  if VarSupports(Left, IArrayInstance) then
    Exit(subNumberFromArray(Left, Right, Op));
  Raise ERuntimeError.Create(Op, Format(ErrBothNumber, ['-']));
end;
```

```
class function TMath.subNumberFromArray(
  const Left: IArrayInstance; Right: Variant; Op: TToken): Variant;
var
  i: Integer;
  newArray: IArrayInstance;
begin
  newArray := makeArray(Left.TypeName, Op);
  for i := 0 to Left.Count-1 do
    newArray.Elements.Add(_Sub(Left[i], Right, Op));

  Result := newArray;
end;
```

The items of Right are subtracted from the items of Left. Note that we call _Sub() for this. This is a recursive call, so that it's possible to include nested or multidimensional arrays.

Then, subtracting two arrays fromeach other:

```
class function TMath.subTwoArrays(
  const Left, Right: IArrayInstance; Op: TToken): Variant;
var
  i: Integer;
  newArray: IArrayInstance;
begin
  if not sameArrayTypes(Left, Right) then
    Raise ERuntimeError.Create(Op, ErrArrayWrongTypes);
  if Left.Count <> Right.Count then
    Raise ERuntimeError.Create(Op, ErrArrayMismatchElem);

  newArray := makeArray(Left.TypeName, Op);
  for i := 0 to Left.Count-1 do
    newArray.Elements.Add(_Sub(Left[i], Right[i], Op));

  Result := newArray;
end;
```

Add the functions to the interface:

```
    class function subNumberFromArray(
      const Left: IArrayInstance; Right: Variant; Op: TToken): Variant; static;
    class function subTwoArrays(const Left, Right: IArrayInstance; Op: TToken
      ): Variant; static;
```

Examples:

```
array Number [] end

var a := Number(1,2,3)
var b := Number([4,5,6])
print(b-a)  // [3, 3, 3]

var c := Number(b)
c -= a
print(c)  // [3, 3, 3]

print(c - 5) // [-2, -2, -2]
```

Multiplication

And the same for multiplication.

```
class function TMath._Mul(const Left, Right: Variant; Op: TToken): Variant;
begin
  ...
  if areBothArray(Left, Right) then
    Exit(mulTwoArrays(Left, Right, Op));
  if VarSupports(Left, IArrayInstance) then
    Exit(mulNumberToArray(Left, Right, Op));
  Raise ERuntimeError.Create(Op, Format(ErrBothNumber, ['*']));
end;
```

```
class function TMath.mulNumberToArray(
  const Left: IArrayInstance; Right: Variant; Op: TToken): Variant;
var
  i: Integer;
  newArray: IArrayInstance;
begin
  newArray := makeArray(Left.TypeName, Op);
  for i := 0 to Left.Count-1 do
    newArray.Elements.Add(_Mul(Left[i], Right, Op));

  Result := newArray;
end;
```

In this case we only allow numeric values to be multiplied to an array. The _Mul() function checks whether the array elements are also numeric. This happens recursively.

```
class function TMath.mulTwoArrays(
  const Left, Right: IArrayInstance; Op: TToken): Variant;
var
  i: Integer;
  newArray: IArrayInstance;
begin
  if not sameArrayTypes(Left, Right) then
    Raise ERuntimeError.Create(Op, ErrArrayWrongTypes);
  if Left.Count <> Right.Count then
    Raise ERuntimeError.Create(Op, ErrArrayMismatchElem);

  newArray := makeArray(Left.TypeName, Op);
  for i := 0 to Left.Count-1 do
    newArray.Elements.Add(_Mul(Left[i], Right[i], Op));

  Result := newArray;
end;
```

Here again the recursively called _Add() function makes sure every multplication is numeric.
Add the class functions to the interface.

```
    class function mulNumberToArray(
      const Left: IArrayInstance; Right: Variant; Op: TToken): Variant; static;
    class function mulTwoArrays(const Left, Right: IArrayInstance; Op: TToken
      ): Variant; static;
```

Examples:

```
array Number [] end

var a := Number(1,2,3)
var b := Number([4,5,6])
print(a*b)    // [4, 10, 18]

var c := Number(a)
c *= b
print(c)     // [4, 10, 18]

print(c * 5)     // [20, 50, 90]
```

Division

What counts for multiplication also counts for division:

```
class function TMath._Div(const Left, Right: Variant; Op: TToken): Variant;
begin
  ...
  if areBothArray(Left, Right) then
    Exit(divTwoArrays(Left, Right, Op));
  if VarSupports(Left, IArrayInstance) then
    Exit(divArrayByNumber(Left, Right, Op));
  Raise ERuntimeError.Create(Op, Format(ErrBothNumber, ['/']));
end;
```

```
class function TMath.divArrayByNumber(
  const Left: IArrayInstance; Right: Variant; Op: TToken): Variant;
var
  i: Integer;
  newArray: IArrayInstance;
begin
  newArray := makeArray(Left.TypeName, Op);
  for i := 0 to Left.Count-1 do
    newArray.Elements.Add(_Div(Left[i], Right, Op));

  Result := newArray;
end;

class function TMath.divTwoArrays(
  const Left, Right: IArrayInstance; Op: TToken): Variant;
var
  i: Integer;
  newArray: IArrayInstance;
begin
  if not sameArrayTypes(Left, Right) then
    Raise ERuntimeError.Create(Op, ErrArrayWrongTypes);
  if Left.Count <> Right.Count then
    Raise ERuntimeError.Create(Op, ErrArrayMismatchElem);

  newArray := makeArray(Left.TypeName, Op);
  for i := 0 to Left.Count-1 do
    newArray.Elements.Add(_Div(Left[i], Right[i], Op));

  Result := newArray;
end;
```

And, add to the interface:

```
    class function divArrayByNumber(
      const Left: IArrayInstance; Right: Variant; Op: TToken): Variant; static;
    class function divTwoArrays(const Left, Right: IArrayInstance; Op: TToken
      ): Variant; static;
```

```
array Number [] end
var a := Number(1,2,3)
var b := Number([4,5,6])
print(a/b)       // [0.25, 0.4, 0.5]

var c := Number(a)
c /= b
print(c)        // [0.25, 0.4, 0.5]
print(c / 5)    // [0.05, 0.08, 0.1]
```

Negation

With negation in this case we mean to multiply the entire array with -1, so that every element is negated.

```
class function TMath._Neg(const Value: Variant; Op: TToken): Variant;
begin
  ...
  if VarSupports(Value, IArrayInstance) then
    Exit(negArray(Value, Op));
  Raise ERuntimeError.Create(Op, Format(ErrNumber, ['-']));
end;
```

```
class function TMath.negArray(const Value: IArrayInstance; Op: TToken): Variant;
var
  i: Integer;
  newArray: IArrayInstance;
begin
  newArray := makeArray(Value.TypeName, Op);
  for i := 0 to Value.Count-1 do
    newArray.Elements.Add(_Neg(Value[i], Op));

  Result := newArray;
end;
```

And add to the interface:

```
    class function negArray(const Value: IArrayInstance; Op: TToken): Variant; static;
```

With this you'll be able to do for example:

```
var m := [[1,2,3],
          [4,5,6],
          [7,8,9]]
m := -m
print(m)
```

This results in [[-1,-2,-3],[-4,-5,-6],[-7,-8,-9]]

Equality

You also would want to be able to compare two arrays with each other:

```
class function TMath._EQ(const Left, Right: Variant; Op: TToken): Variant;
begin
  ...
  if areBothArray(Left, Right) then
    Exit(eqTwoArrays(Left, Right, Op));
  Raise ERuntimeError.Create(Op, Format(ErrWrongTypes, ['equal']));
end;
```

```
class function TMath.eqTwoArrays(const Left, Right: IArrayInstance; Op: TToken
  ): Variant;
var
  i: Integer;
begin
  if not sameArrayTypes(Left, Right) then Exit(False);
  if Left.Count <> Right.Count then Exit(False);

  Result := True;
  for i := 0 to Left.Count-1 do
    if _NEQ(Left[i], Right[i], Op) then
      Exit(False);
end;
```

If the number of elements do not match, the result is false. Then, every element of the Left array is compared with the same element of the right array. If one of the items doesn't match, the result is immediately returned as false.

Add to the interface:

```
    class function eqTwoArrays(const Left, Right: IArrayInstance; Op: TToken
      ): Variant; static;
```

Since TMath.NEQ() uses EQ(), this function also works for unequality.

```
array Number [] end

var m := Number(
  Number(1,2,3),
  Number(4,5,6),
  Number(7,8,9))

var n := Number(
  Number(3,2,1),
  Number(6,5,4),
  Number(9,8,7))

print(m=n)   // False
print(m<>n)  // True
```

The dot-product of two arrays

The dot product of two array A.B multiplies all array items with each other and results in 1 number. For example:

[1,2,3] . [3,2,1] = 10 // 1x3 + 2x2 + 3x1

Since the dot '.' Is already used for access to fields and functions of classes, it's very difficult to also use it for array product calculation.

We can also introduce a new array operator for a dot-product, e.g. '::'. It is similar in importance as multiplication and division, and only applicable to arrays. And I like it as the symbol that multiplicates all elements of two arrays…

In unit uToken.pas create the new token type 'ttColons', and add it rigt after ttColon. In method TTokenTypHelper.toString add the line:

```
    ttColons: Result := '::';
```

In the Lexer (uLexer.pas) modify the case for ':' in procedure ScanToken to:

```
    ':' : case FReader.PeekChar of
            '=' : begin FLook := getChar; AddToken(ttAssign, ':='); end;
            ':' : begin FLook := getChar; AddToken(ttColons, '::'); end;
           else AddToken(ttColon, ':');
          end;
```

Add the colons :: operator to parser function isMulOp:

```
function TParser.isMulOp: Boolean;
begin
  Result := CurrentToken.Typ in [ttMul, ttDiv, ttRem, ttAnd, ttColons];
  //                              *      /      %      &       ::
end;
```

Then, amend the interpreter's VisitBinaryExpr method to include ttColon:

```
function TInterpreter.VisitBinaryExpr(BinaryExpr: TBinaryExpr): Variant;
...
begin
  ...
  case BinaryExpr.Op.Typ of
    ...
    ttColons: Result := TMath._Dot(Left, Right, Op);
  end;
end;
```

Finally, add this new class function to unit uMath:

```
class function TMath._Dot(const Left, Right: Variant; Op: TToken): Variant;
var
  temp: IArrayInstance;
  i: Integer;
begin
  if areBothArray(Left, Right) then begin
    temp := _Mul(Left, Right, Op);
    Result := 0;
    for i := 0 to temp.Count-1 do
      Result += temp[i];
  end
  else
    Raise ERuntimeError.Create(Op, Format(ErrArrayWrongTypes, ['::']));
end;
```

Add the following class function to the interface:

```
    class function _Dot(const Left, Right: Variant; Op: TToken): Variant; static;
```

Now it's possible to do the following:

```
array Number [] end

var A := Number(1,2,3)
var B := Number(3,2,1)

print(A::B)  // 10
print(A::A)  // 14
print(B::B)  // 14
```

Note that those are very simple array calculations, and for example the dot-product only works on 1-dimensional arrays.

Element in array

For this we simply create a math function for the 'in' operator in uMath.pas:

```
class function TMath._In(const Left, Right: Variant; Op: TToken): Variant;
begin
  if oneOfBothNull(Left, Right) then
    Exit(False);
  if VarSupports(Right, IArrayInstance) then
    Exit(IArrayInstance(Right).Elements.Contains(Left));
  Raise ERuntimeError.Create(Op, Format(ErrMustBeBoolean, ['in']));
end;
```

Add ttIn to the isRelOp function:

```
function TParser.isRelOp: Boolean;
begin
  Result := CurrentToken.Typ in [ttEQ, ttNEQ, ttGT, ttGE, ttLT, ttLE, ttIn, ttIs];
  //                              =     <>    >     >=     <    <=     in   is
end;
```

And add the case for ttIn to VisitBinaryExpr:

```
function TInterpreter.VisitBinaryExpr(BinaryExpr: TBinaryExpr): Variant;
...
begin
...
  case BinaryExpr.Op.Typ of
    ...
    ttIn:   Result := TMath._In(Left, Right, Op);
    ttIs:   Result := TMath._Is(Left, Right, Op);
    ...
  end;
end;
```

With this it is possible to check whether an element belongs to the array, like this:

```
array Number [] end

var a := Number(1,2,3,4,5,6,7,8,9)
var b := 0
if b in a then
  print('included')
end
```

Standard Array type

So far, you could define your own array type and create new array expressions based on such types. However, we would need to have a standard array type with standard added functionality, which is also availabe on standard declared array expressions. For example, we can create a standard type:

```
array Array [] end
```

and make expressions from this type:

```
var a := [1,2,3,4,5]
```

to be of type Array.

How can we arrange this? By defining a standard array type in the Globals memory:

```
constructor TInterpreter.Create;
begin
  ...

  // store base Array type
  Globals.Store('Array', IArrayable(
    TArrayClass.Create(
      TIdent.Create(TToken.Create(ttIdentifier, 'Array', Null, 0, 0)),
      TArrayElements.Create(),
      TMembers.Create(TFieldTable.Create, TConstTable.Create,
        TMethodTable.Create, TValueTable.Create))));

  CurrentSpace := Globals;
end;
```

Next, we have to change the visitor for ArrayDeclExpr. We want array expressions to be of type Array.

```
function TInterpreter.VisitArrayDeclExpr(ArrayDeclExpr: TArrayDeclExpr): Variant;
var
  ArrayType: IArrayable;
  Instance: IArrayInstance;
  i: Integer;
begin
  ArrayType := IArrayable(Globals['Array']);

  Instance := IArrayInstance(TArrayInstance.Create(ArrayType as TArrayClass));
  for i := 0 to ArrayDeclExpr.ArrayDecl.Elements.Count-1 do
    Instance.Elements.Add(Visit(ArrayDeclExpr.ArrayDecl.Elements[i]));

  Result := Instance;
end;
```

We retrieve the array type from the Globals memory and then create an instance of the Array type. Next, we visit all elements and store them in the instance's elements. Finally, the instance is returned as the function result.

We have to add 'Array' to the resolver's EnableStandardFunctions procedure:

```
procedure TResolver.EnableStandardFunctions;
begin
  with GlobalScope do begin
    …
    Enter(TSymbol.Create('Array', Enabled, False));
  end;
end;
```

Next, in the interpreter visitor VisitExtensionDecl() add the cases for IClassable:

```
procedure TInterpreter.VisitExtensionDecl(ExtensionDecl: TExtensionDecl);
...
begin
  ...
  if not VarSupportsIntf(TypeDecl, [IClassable, IArrayable]) then
  ...
  if VarSupports(TypeDecl, IClassable) then
  ...
  else if VarSupports(TypeDecl, IArrayable) then
    IClassable(TypeDecl).ExtendWith(Members);
end;
```

Code example:

```
// test extension

extension Array

  val count := length(self)

end

var a := [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]

print(a.count)

var b := Array(a)
b := b >< [16]
print(b)

print(b.count)
```

Accessing array items

> A capacity, and taste, for reading gives access to whatever has already been discovered by others.
>
> — Abraham Lincoln

Like in any other programmig language, in Gear an array item is accessed through its index. Given the array a := [1,2,3,4,5], then its first index is 0, whereas the last index is 4.

So a[0] returns the value 1, a[1] returns 2, etc.

Items in multi dimensional arrays can be found by repeated indexes, for eample in array:

m := [[1,2],[3,4],[5,6]], then m[0][0] returns 1; m[1][0] returns 3; m[2][1] returns 6.

Parsing indexed expressions

Let's create a new AST node for an indexed expression. It simply contains a variable of type TExpr and an index of type TExpr. Later on we have to make sure that the index complies with being an integer number.

```
  TIndexedExpr = class(TFactorExpr)
    private
      FVariable: TExpr;
      FIndex: TExpr;
    public
      property Variable: TExpr read FVariable;
      property Index: TExpr read FIndex;
      constructor Create(AVariable: TExpr; AIndex: TExpr);
      destructor Destroy; override;
  end;
```

And the implementation:

```
constructor TIndexedExpr.Create(AVariable: TExpr; AIndex: TExpr);
begin
  inherited Create(AVariable.Token);
  FVariable := AVariable;
  FIndex := AIndex;
end;

destructor TIndexedExpr.Destroy;
begin
  if Assigned(FVariable) then FVariable.Free;
  if Assigned(FIndex) then FIndex.Free;
  inherited Destroy;
end;
```

The EBNF of an indexed expression is as follows:

```
IndexedExpr = Variable '[' Expr ']' .
```

We place it inside a Call expression, so that the new EBNF for CallExpr is:

```
  CallExpr = Factor { '(' [Arguments] ')' | '.' Ident | '[' Expr ']' } .
```

Change function ParseCallExpr so that it accepts the '[' token:

```
function TParser.ParseCallExpr: TExpr;
begin
  Result := ParseFactor;
  while CurrentToken.Typ in [ttDot, ttOpenParen, ttOpenBrack] do
    case CurrentToken.Typ of
      ttDot: begin
        Next; // skip '.'
        Result := TGetExpr.Create(Result, ParseIdent);
      end;
      ttOpenParen: Result := ParseCallArgs(Result);
      ttOpenBrack: begin
        Next; // skip [
        Result := TIndexedExpr.Create(Result, ParseExpr);
        Expect(ttCloseBrack);
      end;
    end;
end;
```

If it finds a '[' then a TIndexedExpr is created, and finally a closing ']' is expected. Note that this can be repeated to cover for mutliple [][][]… cases.

Here are the usual printer and resolver functions:

```
procedure TPrinter.VisitIndexedExpr(IndexedExpr: TIndexedExpr);
begin
  IncIndent;
  VisitNode(IndexedExpr);
  Visit(IndexedExpr.Variable);
  Writeln(Indent, 'Index: ');
  IncIndent;
  Visit(IndexedExpr.Index);
  DecIndent;
  DecIndent;
end;
```

```
procedure TResolver.VisitIndexedExpr(IndexedExpr: TIndexedExpr);
begin
  Visit(IndexedExpr.Variable);
  Visit(IndexedExpr.Index);
end;
```

Next, we'll look at interpreting an indexed expression.

Interpreting an indexed expression

The visitor for the interpreter first checks if the variable expression is an array instance, and if not, checks if it is a string variable, otherwise raises an error. This error is also raised if you try to access an element in an non-existent dimension, like this:

```
var a := [1,2,3]
print(a[1][0]) // error Variable or value must be of an array type.
```

This is because the expression a[1] in itself is not an array!

```
function TInterpreter.VisitIndexedExpr(IndexedExpr: TIndexedExpr): Variant;
var
  Instance: Variant;
  Index: Variant;
begin
  Instance := Visit(IndexedExpr.Variable);
  Index := Visit(IndexedExpr.Index);
  if VarSupports(Instance, IArrayInstance) then begin
    CheckNumericIndex(Index, IndexedExpr.Index.Token);
    Result := IArrayInstance(Instance).get(Index, IndexedExpr.Index.Token);
  end
  else if VarIsStr(Instance) then begin
    CheckNumericIndex(Index, IndexedExpr.Index.Token);
    Result := String(Instance)[Index+1];  // strings start at index 1
  end
  else
    Raise ERuntimeError.Create(IndexedExpr.Variable.Token, ErrMustBeArrayType);
end;
```

We use a small private helper procedure that checks whether the index is numeric. Preferrably, we check it in the resolver, but that's not possible, due to the fact that the expression needs to be evaluated to get its value, and this happens in the interpreter.

```
  ErrArrayTypeExpected = 'Array or string type expected.';
```

```
procedure TInterpreter.CheckNumericIndex(Value: Variant; Token: TToken);
begin
  if not VarIsNumeric(Value) then
    Raise ERuntimeError.Create(Token, ErrArrayTypeExpected);
end;
```

We then retrieve the index, and call function 'get' to load the value in the array. Function 'get' is not yet defined in the interface so let's add that.

```
  IArrayInstance = Interface
    ...
    function Get(i: Integer; Token: TToken): Variant;
  end;
```

Add to the uses clause of uArrayIntf 'uToken':

```
uses
  Classes, SysUtils, uCallable, uAST, Variants, uMembers, uCollections, uToken;
```

And the implementation of function 'get':

```
function TArrayInstance.Get(i: Integer; Token: TToken): Variant;
begin
  if (i >= 0) and (i < getCount) then
    Result := FElements[i]
  else
    Raise ERuntimeError.Create(Token,
      'Index ('+ IntToStr(i) + ') out of range ('+ '0,' + IntToStr(getCount-1) + ').');
end;
```

First a range check is performed, and if outside the range an error is created. If in range, the value of the i-th element is fetched.

Now this works:

```
var a := [1,2,3]
print(a[0], a[1], a[2]) // 123

var b := [[1,2],[3,4],[5,6]]
print(b[1][1])  // 4
print(b[2][0])  // 5


var s := 'Hello world!'
print(s[0])             // H
print(s[length(s)-1])   // !
```

Accessing an index out of range provides the following error message:

```
print(a[3])   // error: Index (3) out of range (0,2)
```

For strings note that a Pascal string starts at index 1, so add one to the index. Also, the variant value has to be cast to string first.

There is one small side effect. Since internally we use the type Number, which is based on float, and always cast to a variant value, you can even enter a float value as index. Like a[1.5] for example. When retrieving the value in the get(I, Token) function we explicitly ask for an Integer, which means the variant is cast to Integer, using standard rounding. So, 1.5 will be rounded off to 2, whereas 1.4 becomes 1.

```
var x := a[1.4]
print(x)   // 2
x := a[1.5]
print(x)   // 3
```

Of course you can add a check in the get() function to make sure the index is an integer value. How would you do this?

Assigning to an indexed expression

We also want to be able to assign values to array expressions. For example:

```
var a := [1,2,3]
print(a) // [1,2,3]
a[1] := 6
print(a) // [1,6,3]
```

For this we create a new AST node that takes care of statements with indexed expressions:

```
  TIndexedExprStmt = class(TStmt)
    private
      FIndexedExpr: TIndexedExpr;
      FOp: TToken;
      FExpr: TExpr;
    public
      property IndexedExpr: TIndexedExpr read FIndexedExpr;
      property Expr: TExpr read FExpr;
      property Op: TToken read FOp;
      constructor Create(AIndexedExpr: TIndexedExpr; AOp: TToken; AExpr: TExpr);
      destructor Destroy; override;
  end;


constructor TIndexedExprStmt.Create
  (AIndexedExpr: TIndexedExpr; AOp: TToken; AExpr: TExpr);
begin
  Inherited Create(AOp);
  FIndexedExpr := AIndexedExpr;
  FOp := AOp;
  FExpr := AExpr;
end;

destructor TIndexedExprStmt.Destroy;
begin
  if Assigned(FIndexedExpr) then FIndexedExpr.Free;
  if Assigned(FExpr) then FExpr.Free;
  inherited Destroy;
end;
```

The regular printer and resolver visitors:

```
procedure TPrinter.VisitIndexedExprStmt(IndexedExprStmt: TIndexedExprStmt);
begin
  IncIndent;
  VisitNode(IndexedExprStmt);
  Visit(IndexedExprStmt.IndexedExpr);
  Visit(IndexedExprStmt.Expr);
  DecIndent;
end;
```

```
procedure TResolver.VisitIndexedExprStmt(IndexedExprStmt: TIndexedExprStmt);
begin
  Visit(IndexedExprStmt.IndexedExpr);
  Visit(IndexedExprStmt.Expr);
end;
```

In the parser we change ParseAssignStmt such that it can handle indexed expression assignments:

```
function TParser.ParseAssignStmt: TStmt;
var
  ...
  IndexedExpr: TIndexedExpr;
begin
  ...
  if CurrentToken.Typ in AssignSet then begin
    ...
    else if Left is TIndexedExpr then begin
      IndexedExpr := Left as TIndexedExpr;
      Result := TIndexedExprStmt.Create(IndexedExpr, Op, Right);
    end
    else
      Error(Token, ErrInvalidAssignTarget);
  end
  ...
end;
```

The interpreter visitor for indexed expression statements is as follows:

```
procedure TInterpreter.VisitIndexedExprStmt(IndexedExprStmt: TIndexedExprStmt);
var
  Variable, OldValue, NewValue, Value: Variant;
  Index: Variant;
  VarAsStr: String;
begin
  Variable := Visit(IndexedExprStmt.IndexedExpr.Variable);
  Index := Visit(IndexedExprStmt.IndexedExpr.Index);
  NewValue := Visit(IndexedExprStmt.Expr);

  with IndexedExprStmt do
  if VarSupports(Variable, IArrayInstance) then begin
    CheckNumericIndex(Index, IndexedExpr.Index.Token);
    OldValue := IArrayInstance(Variable).get(Index, IndexedExpr.Index.Token);
    Value := getAssignValue(OldValue, NewValue, IndexedExpr.Variable.Token, Op);
    IArrayInstance(Variable).Put(Index, Value, IndexedExpr.Index.Token);
  end
  else if VarIsStr(Variable) then begin
    CheckNumericIndex(Index, IndexedExpr.Index.Token);
    VarAsStr := Variable;
    OldValue := VarAsStr[Index+1];
    VarAsStr[Index+1] := getAssignValue(OldValue, NewValue,
      IndexedExpr.Variable.Token, Op);
    Assign(IndexedExpr.Variable as TVariable, VarAsStr);
  end
  else
    Raise ERuntimeError.Create(IndexedExpr.Variable.Token, ErrArrayTypeExpected);
end;
```

First, retrieve (visit) the used variable in the indexed expression. This must be an array instance or a string!

Then, retrieve the index, which must be a numeric value. If numeric, the old value of the indexed expression is retrieved. Next, visit the new value, and apply both in the getAssignValue function. Finally, the new value is put in the memory location of the indexed variable.

In order to put a new value into an array index, we create a new procedure Put in the interface of IArrayInstance:

```
  IArrayInstance = Interface
    ...
    procedure Put(i: Integer; Value: Variant; Token: TToken);
  end;
```

The accompanying implementation in TArrayInstance is a public function:

```
procedure TArrayInstance.Put(i: Integer; Value: Variant; Token: TToken);
begin
  if (i >= 0) and (i < getCount) then
    FElements[i] := Value
  else
    Raise ERuntimeError.Create(Token,
      'Index ('+ IntToStr(i) + ') out of range ('+ '0,' + IntToStr(getCount-1) + ').');
end;
```

It checks the range of the index and if outside the required range an error is generated.

Extending Array

Now that we have created an Array system, with built-in functionality, such as indexed access, array type definitions and array expressions, it is time to combine this all with extensions. We can extend the standard Array type with nice functionality like map, filter and reduce functions, for instance. Also functions for adding, inserting, deleting, searching, etc. can be added, so that this will always be available to the user of the Gear language.

> The reasons why images are so primal and people immediately relate to it is that we are exquisitely engineered to interpret information that is arrayed in two dimensions. That's our eyesight. That's how our eye-brain system works. So it immediately feels to us when we look at an image like we have extended our senses.
>
> — Carolyn Porco

Standard functions

An array wouldn't be very useful if it didn't have some standard functions available to the programmer in order to manipulate its contents.

Specifically, you would like to add, insert and delete items, as well as search on contents and even retrieve items, given a certain key. So, let's define the following set of standard functions that will be available to arrays.

- add, which adds a value to an array,
- insert, which inserts a value at a certain position,
- delete, which deletes an items from an array,
- indexOf, which gets the index of an item,
- contains, which returns True if an item is in the list, and
- retrieve, which gets the value of an index.

Updating the interface and instance

All these functions operate on the instance of an array, so let's add them to the interface of array, IArrayInstance.

```
IArrayInstance = Interface
  ...
  procedure Add(const AValue: Variant);
  procedure Insert(const i: Integer; const AValue: Variant; Token: TToken);
  procedure Delete(const AValue: Variant; Token: TToken);
  function Contains(const AValue: Variant): Boolean;
  function IndexOf(const AValue: Variant): Variant;
  function Retrieve(const AIndex: Integer; Token: TToken): Variant;
end;
```

We'll now implement them in TArrayInstance:

```
procedure TArrayInstance.Add(const AValue: Variant);
begin
  FElements.Add(AValue);
end;
```

```
procedure TArrayInstance.Insert(const i: Integer;
  const AValue: Variant; Token: TToken);
begin
  if (i >= 0) and (i <= getCount) then
    FElements.Insert(i, AValue)
  else
    Raise ERuntimeError.Create(Token,
      'Index ('+ IntToStr(i) + ') out of range ('+ '0,' +
      IntToStr(getCount) + ') during insert action.');
end;
```

```
procedure TArrayInstance.Delete(const AValue: Variant; Token: TToken);
var
  i: Integer;
begin
  i := FElements.Remove(AValue);
  if i<0 then
    Raise ERuntimeError.Create(Token,
      'Value "' + AValue.toString + '" not found in delete action.');
end;
```

```
function TArrayInstance.Contains(const AValue: Variant): Boolean;
begin
  Result := FElements.Contains(AValue);
end;
```

```
function TArrayInstance.IndexOf(const AValue: Variant): Variant;
begin
  Result := FElements.IndexOf(AValue);
  if Result < 0 then
    Result := Unassigned;
end;
```

```
function TArrayInstance.Retrieve(const AIndex: Integer; Token: TToken): Variant;
begin
  if (AIndex >= 0) and (AIndex < getCount) then
    Result := FElements[AIndex]
  else
    Result := Unassigned;
end;
```

Add functions to the runtime library

It's time to set the functions to use. We do that by adding the following classes with call methods to unit uStandard.pas. We've done this before, so it shouldn't be too hard…

```
  TListAdd = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

  TListInsert = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;
```

```
  TListDelete = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

  TListContains = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

  TListIndexOf = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

  TListRetrieve = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;
```

```
  TListFirst = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

  TListLast = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;
```

All function classes start with 'List', and not with Array. You'll see shortly that I use the functions for both arrays and dictionaries.

Each call function takes as first parameter the value of 'self', which is either an array instance (or a dictionary instance later on). If it is none of both a runtime error is generated, like this:

```
  Instance := ArgList[0].Value;                          // self as value
  if VarSupports(Instance, IArrayInstance) then begin
    // action on array
  end
  else if VarSupports(Instance, IDictInstance) then begin
    // action on dictionary
  end
  else
    Raise ERuntimeError.Create(Token, Message);
```

This principle is used for all List functions below.

Add

The add function takes 2 parameters for an array (self and value). It then calls the function available through the interface with the given arguments.

```
function TListAdd.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Instance: Variant;
begin
  Instance := ArgList[0].Value;
  if VarSupports(Instance, IArrayInstance) then begin
    TFunc.CheckArity(Token, ArgList.Count, 2);
    IArrayInstance(Instance).Add(ArgList[1].Value);
    Result := IArrayInstance(Instance);
  end
    Raise ERuntimeError.Create(Token,
      'listAdd function not possible for this type.');
end;
```

Insert

Function Insert has an extra parameter compared to add: the index where to insert the item. In case of arrays the index and the value.

```
function TListInsert.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Instance: Variant;
begin
  Instance := ArgList[0].Value;
  if VarSupports(Instance, IArrayInstance) then begin
    TFunc.CheckArity(Token, ArgList.Count, 3);
    IArrayInstance(Instance).Insert(ArgList[1].Value, ArgList[2].Value);
    Result := IArrayInstance(Instance);
  end
  else
    Raise ERuntimeError.Create(Token,
      'listInsert function not possible for this type.');
end;
```

Delete

Function Delete calls the instance's delete function with the given value for an array.

```
function TListDelete.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Instance: Variant;
begin
  Instance := ArgList[0].Value;
  if VarSupports(Instance, IArrayInstance) then begin
    TFunc.CheckArity(Token, ArgList.Count, 2);
    IArrayInstance(Instance).Delete(ArgList[1].Value);
    Result := IArrayInstance(Instance);
  end
  else
    Raise ERuntimeError.Create(Token,
      'listDelete function not possible for this type.');
end;
```

Contains

Function Contains gets its result from the call to the instance's Contains function.

```
function TListContains.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Instance: Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 2);
  Instance := ArgList[0].Value;
  if VarSupports(Instance, IArrayInstance) then
    Result := IArrayInstance(Instance).Contains(ArgList[1].Value)
  else
    Raise ERuntimeError.Create(Token,
      'listContains function not possible for this type.');
end;
```

IndexOf

Function IndexOf gets the index of a value from an array.

```
function TListIndexOf.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Instance: Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 2);
  Instance := ArgList[0].Value;
  if VarSupports(Instance, IArrayInstance) then
    Result := IArrayInstance(Instance).IndexOf(ArgList[1].Value)
  else
    Raise ERuntimeError.Create(Token,
      'listIndexOf function not possible for this type.');
end;
```

Retrieve

Lastly, the retrieve function, which takes an index in case of an array.

```
function TListRetrieve.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Instance: Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 2);
  Instance := ArgList[0].Value;
  if VarSupports(Instance, IArrayInstance) then
    Result := IArrayInstance(Instance).Retrieve(ArgList[1].Value)
  else
    Raise ERuntimeError.Create(Token,
      'listRetrieve function not possible for this type.');
end;
```

First and last item

We use the standard FGL functions instead of creating our own.

The first item of a list is retrieved via:

```
function TListFirst.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Instance: Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 1);
  Instance := ArgList[0].Value;
  if VarSupports(Instance, IArrayInstance) then begin
    if IArrayInstance(Instance).Count > 0 then
      Result := IArrayInstance(Instance).Elements.First
    else Result := Unassigned;
  end
  else
    Raise ERuntimeError.Create(Token,
      'listFirst function not possible for this type.');
end;
```

And the last item of a list via:

```
function TListLast.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Instance: Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 1);
  Instance := ArgList[0].Value;
  if VarSupports(Instance, IArrayInstance) then begin
    if IArrayInstance(Instance).Count > 0 then
      Result := IArrayInstance(Instance).Elements.Last
    else Result := Unassigned;
  end
  else
    Raise ERuntimeError.Create(Token,
      'listLast function not possible for this type.');
end;
```

Last but not least, add the new standard functions to the Globals store:

```
constructor TInterpreter.Create;
begin
  ...
  Globals.Store('listAdd', ICallable(TListAdd.Create));
  Globals.Store('listContains', ICallable(TListContains.Create));
  Globals.Store('listDelete', ICallable(TListDelete.Create));
  Globals.Store('listIndexOf', ICallable(TListIndexOf.Create));
  Globals.Store('listInsert', ICallable(TListInsert.Create));
  Globals.Store('listRetrieve', ICallable(TListRetrieve.Create));
  Globals.Store('listFirst', ICallable(TListFirst.Create));
  Globals.Store('listLast', ICallable(TListLast.Create));
  ...
end;
```

Finally, also add them to the Resolver's EnableStandardFunctions method:

```
    Enter(TSymbol.Create('listAdd', Enabled, False));
    Enter(TSymbol.Create('listContains', Enabled, False));
    Enter(TSymbol.Create('listDelete', Enabled, False));
    Enter(TSymbol.Create('listIndexOf', Enabled, False));
    Enter(TSymbol.Create('listInsert', Enabled, False));
    Enter(TSymbol.Create('listRetrieve', Enabled, False));
    Enter(TSymbol.Create('listFirst', Enabled, False));
    Enter(TSymbol.Create('listLast', Enabled, False));
```

Let's try to use the above in some nice code that shows the capabilities of Gear's arrays:

```
extension Array
  val count := length(self)
  val first := listFirst(self)
  val last := listLast(self)
  func add(.value) => listAdd(self, value)
  func insert(at index, .value) => listInsert(self, index, value)
  func delete(.value) => listDelete(self, value)
  func contains(.value) => listContains(self, value)
  func index(of value) => listIndexOf(self, value)
  func retrieve(.index) => listRetrieve(self, index)
end
```

```
extension Array

  func reduce(initialValue, nextValue)
    var result := initialValue

    for var i := 0 where i < length(self), i+=1 do
      result := nextValue(result, self[i])
    end
    return result
  end

  func filter(includeElement)
    var result := []
    for var i := 0 where i < length(self), i+=1 do
      if includeElement(self[i]) then
        result.add(value: self[i])
      end
    end
    return result
  end

  func map(transform)
    var result := []
    for var i := 0 where i < length(self), i+=1 do
      result.add(value: transform(self[i]))
    end
    return result
  end

  func forEach(function)
    for var i:=0 where i<length(self), i+=1 do
      function(self[i])
    end
  end

end

var a := [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]

print(a.count)
print(a.first)
print(a.last)

print(a.map(x=>x^2))
print(a.map(x=>x^3))

print(a.reduce(0, (x,y)=>x+y))

print(a.filter(x=>x%2=0))

var b := Array(a)
b := b >< [16]
print(b)

print(b.count)
print(b.first)
print(b.last)

var c := a.filter(x=>x%2=0)
print(c)
print(b.filter(x=>x%2=0))

print(a.reduce(1, (x,y)=>x*y)) //fac(15)

a.forEach(func(n) print(n, terminator: ' | ') end)
```

The language is becoming powerful already. Later we'll add dictionary as a type, and enums too. But next we'll focus on embedding external files and libraries, because it's not so practical to copy above code everytime in a new test file.

Chapter 12 – Using external files

Standard libraries typically include definitions for commonly used algorithms, data structures, and mechanisms for input and output. Depending on the constructs made available by the host language, a standard library may include: subroutines, global variables, type definitions and extensions.

Most standard libraries include definitions for at least the following commonly used facilities: algorithms (such as sorting algorithms), data structures (such as lists, trees, and hash tables), interaction with the host platform, including input/output and operating system calls (WikiPedia).

Parsing the used file

In the last chapter we created arrays with extensions, for example for the standard Array type. However, it is a pain to always copy the code for the array extension into the code we're currently working on. Preferrably we would like to create the extension code once and if needed we make a reference to it, for example like this:

> Don't set pen to paper until you know your main characters inside out. Create files detailing their appearances, likes, dislikes, and personal background. You may not use all the information, but it is a crucial step in planning your story.
>
> — Jojo Moyes

```
use arrays

var a := [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15].filter(x=>x%2=0)
```

The filter function then is declared in the extension to array in unit arrays.gear.

There are two ways of doing this:

1. copy the file arrays.gear into the current file and interpret it,
2. interpret and create a fully generated AST, which can be imported into the current program.

The second option is preferred since we don't have to interpret the standard functionality time and again. So, how to do this?

We need a way to write the AST to a file and read it back again when we encounter the keyword 'use'. However, in our case, the burden of creating an output file in JSON format for example and then reading and parsing it back takes as much, if not more, time, then just importing the contents of the library into the current file and building the AST in one go.

The strategy is as follows: the clause 'use filename' is treated as a statement. It can be anywhere in the code. The functionality of the file becomes available directly after the use-statement.

Our first solution will only look in a standard library location, called 'gearlib', which is a subfolder in the gear application folder.

The AST node for the use statement is as follows:

```
  TUseStmt = class(TStmt)
    private
      FFileName: String;
    public
      property FileName: String read FFileName;
      constructor Create(AFileName: String; AToken: TToken);
  end;


constructor TUseStmt.Create(AFileName: String; AToken: TToken);
begin
  Inherited Create(AToken);
  FFileName := AFileName;
end;
```

It simply stores the file name of the used file.
In the parser, add the ttUse token type to the statement start set:

```
const
  StmtStartSet: TTokenTypSet = [ttIf, ttWhile, ttRepeat, ttFor, ttReturn,
    ttSwitch, ttEnsure, ttPrint, ttInherited, ttSelf, ttUse, ttBreak,
    ttIdentifier];
```

Then add the case for ttUse to method ParseStmt:

```
function TParser.ParseStmt: TStmt;
begin
  case CurrentToken.Typ of
    ...
    ttUse: Result := ParseUseStmt;
    ttBreak: Result := ParseBreakStmt;
    else
      Result := ParseAssignStmt;
  end;
end;
```

Next, write the printer and resolver visitors:

```
procedure TPrinter.VisitUseStmt(UseStmt: TUseStmt);
begin
  IncIndent;
  VisitNode(UseStmt);
  WriteLn(Indent + 'Filename: ' + UseStmt.FileName);
  DecIndent;
end;
```

The printer visitor prints the file name and nothing more.

The resolver visitor does nothing. It's not necessary since the code inside the used file will become part of the AST and gets resolved there.

```
procedure TResolver.VisitUseStmt(UseStmt: TUseStmt);
begin
  // do nothing
end;
```

Function ParseUseStmt is as follows:

```
function TParser.ParseUseStmt: TStmt;
const Ext = '.gear';
var
  Token: TToken;
  Ident: TIdent;
  Index, i: LongInt;
  FileName, UsedFile: String;
  Lexer: TLexer = Nil;
  Reader: TReader = Nil;
begin
  Token := CurrentToken;
  Next; // skip use
  Ident := ParseIdent;
  FileName := Ident.Text;
  Result := TUseStmt.Create(FileName, Token);
  if FileExists(GearLibFolder + FileName + Ext) then
    UsedFile := GearLibFolder + FileName + Ext
  else
    Error(Ident.Token, Format(ErrNotExistsUseFile, [FileName]));
  if not FileNameArray.Contains(UsedFile) then begin
    try
      Reader := TReader.Create(UsedFile, itFile);
    except
      Error(Ident.Token, Format(ErrIncorrectUseFile, [FileName]));
    end;
    Lexer := TLexer.Create(Reader);
    //Insert the new tokens in the main tokens list
    Index := Current;
    for i := 0 to Lexer.Tokens.Count-2 do
      Tokens.Insert(Index+i, Lexer.Tokens[i]);
  end;
end;
```

To the errors add:

```
  ErrNotExistsUseFile = 'Used file "%s" does not exist.';
  ErrIncorrectUseFile = 'Used file "%s" is incorrect or corrupt.';
```

First, we create the use-statement, which simply returns the name of the file.

Next, from the file name a complete folder and file path is created in variable UsedFile. In the Try..Except statement we try to create a TReader instance. If this fails, it means the used filename is illegal or it doesn't exist, and an error is generated.

If the file exists, a new TLexer instance is created. Recall that in the creation process already all tokens are read and stored in a TTokens array.

Finally, we have to insert all new tokens into the main program's token list. We use a helper index, which copies the current parser index, to keep track of the position in the tokens array. One by one the new tokens are inserted. Note that we don't copy the last item of the inserted file (Count-2 instead of Count-1), since that is an end of file token, and we don't want to have that in the middle of our code.

Add to the uses clause in the parser's implementation uReader and uLanguage:

```
implementation
uses uCollections, uLanguage, uReader;
```

The visitor for the interpreter is empty, just as the resolver's visitor:

```
procedure TInterpreter.VisitUseStmt(UseStmt: TUseStmt);
begin
  //do nothing
end;
```

By inserting all new tokens into the main token list, they will automatically get interpreted.
The mentioned variable GearLibFolder in ParseUseStmt is newly defined in unit uLanguage:

```
...
var
  AppFolder: String = '';
  GearLibFolder: String = '';

implementation
```

The value of these variables is set in the main program: gear.lpr.

```
...
begin
  //get the application directory from the executable
  AppFolder := ExtractFilePath(ExcludeTrailingPathDelimiter(ParamStr(0)));
  GearLibFolder := AppFolder + 'gearlib/';

  Application := TGearLang.Create(nil);
  ...
```

Free Pascal array ParamStr contains all commandline parameters, and as such ParamStr(0) is the name and folder of the application itself. We extract the clean folder name and append 'gearlib/' to it to create the GearLib folder, which should contain all library files. By the way, note that the files to be used should have the file extension '.gear'.

In principle the above changes should work already after compilation and running with for example the following code:

```
print('Hello world!')

use arrays

var a := [0,1,2,3,4,5,6,7,8,9].filter(x=>x%2=0)
print(a)  // Array [0, 2, 4, 6, 8]
```

However, what happens in case an error is detected in one of the used files?

Error detection in used files

By using library files into your program, it may be possible that an error occurs, not in your main program, but in one of the used files. If such error happens, then of course you would want to know where exactly it occurs, meaning in which file and location inside that file.

> Any man can make mistakes, but only an idiot persists in his error.
> — Marcus Tullius Cicero

The best location to store location information still is the Token, and we could store the filename inside the token. However, this means we have to store a possibly long string in each token. Better would be to store a number, which takes much less space. The number could be an index pointer to an array of used file names. Since the file reading starts in unit uReader, let's also start with adding the functionality for managing a list of used file names. Add the following to unit uReader:

```
  TFileNameArray = specialize TArray<TFileName>;

var
  FileNameArray: TFileNameArray;
```

Make sure to add unit uCollections to the uses clause. Note: TFileName is a standard Free Pascal type of String.

At the end of the unit create an initialization and finalization section:

```
initialization
  FileNameArray := TFileNameArray.Create;
finalization
  FileNameArray.Free;
end.
```

Next, change class TReader to include a file index. Each reader will have its own file index.

```
  TReader = class
    private
      FFileName: TFileName;
      FFileIndex: Integer;
      ...
    public
      ...
      property FileIndex: Integer read FFileIndex;
      ...
  end;
```

Finally, change the constructor to include the file index:

```
constructor TReader.Create(Source: String; InputType: TInputType);
...
begin
  ...
  FFileIndex := -1;
  case InputType of
```

```
    ...
    itFile:
      try
        FFileName := Source;
        FFileIndex := FileNameArray.Add(FFileName);
        ...
      end;
  end;
  ...
end;
```

As the file name is added to the file name array, it automatically provides the index where it was added to FileIndex.

Next step is to include the file information in the tokens. Simply add the field FFileIndex, create a property and make sure to give it a value through the constructor:

```
  TToken = class
    private
      ...
      FFileIndex: Integer;
    public
      ...
      property FileIndex: Integer read FFileIndex;
      constructor Create(ATyp: TTokenTyp; ALexeme: String;
        AValue: Variant; ALine, ACol: LongInt; AFileIndex: Integer = 0);
      ...
  end;
```

The default value 0 for AFileIndex means it is the main program. The implementation of the constructor is:

```
constructor TToken.Create(ATyp: TTokenTyp; ALexeme: String; AValue: Variant;
  ALine, ACol: LongInt; AFileIndex: Integer);
begin
  ...
  FFileIndex := AFileIndex;
end;
```

Now we want to use the file info/index so that when an error is generated, we see in which file it appears. This means changing uError.pas. Below the changed/added lines in gray.

```
  TErrorItem = class
    Line, Col: Integer;
    FileIndex: Integer;
    Msg: String;
    constructor Create(const ALine, ACol: Integer; const AMsg: String);
    constructor Create(AToken: TToken; const AMsg: String);
    function toString: String; override;
  end;

  TErrors = class(specialize TArrayObj<TErrorItem>)
    procedure Append(const ALine, ACol: Integer; const AMsg: String);
    procedure Append(AToken: TToken; const AMsg: String);
    function isEmpty: Boolean;
    procedure Reset;
    function toString: String; override;
  end;
```

We now add a new error by using the token as a whole.

```
constructor TErrorItem.Create(AToken: TToken; const AMsg: String);
begin
  Line := AToken.Line;
  Col := AToken.Col;
  Msg := AMsg;
  FileIndex := AToken.FileIndex;
end;
```

```
procedure TErrors.Append(AToken: TToken; const AMsg: String);
begin
  Add(TErrorItem.Create(AToken, AMsg));
end;
```

We also have to change the way the errors are printed.

```
function TErrorItem.toString: String;
begin
  Result := Format('@[%d,%d] in %s: %s',
    [Line, Col, ExtractFileName(FileNameArray[FileIndex]), Msg]);
end;
```

The function ExtractFileName() extracts all folder info, so that you only see the unit name and extension.
Let's also change the runtime error generation:

```
procedure RuntimeError(E: ERuntimeError);
const
  ErrString = '@[%d,%d] in %s - Runtime error: %s';
begin
  with E.Token do
    Writeln(Format(ErrString,
      [Line, Col, ExtractFileName(FileNameArray[FileIndex]), E.Message]));
  Exit;
end;
```

Add uReader to the uses clause in in the implementation of uError.pas.

```
implementation
uses uReader;
```

An example runtime error message could then be:

```
@[8,51] in testuse.gear - Runtime error: Variable "z" is undefined.
```

It is clear that the error occurred in file 'testuse.gear'.

Since that we've changed TToken to include the file index, we need to make sure to add the right file index to each created token. We took care to create a default value of zero for file index, so our interpreter will be compiled without errors. However, we need to make some clean up changes, starting in unit uLexer.pas.

All changes will be shown here (add FReader.FileIndex as the last parameter to the call to TToken.Create):

```
procedure TLexer.ScanTokens;
begin
  ...
  Tokens.Add(TToken.Create(ttEOF, '', Nil, FLine, FCol, FReader.FileIndex));
end;
```

```
procedure TLexer.ScanToken(const Line, Col: Integer);

  procedure AddToken(const Typ: TTokenTyp; const Lexeme: String);
  ...
  begin
    Token := TToken.Create(Typ, Lexeme, Null, Line, Col, FReader.FileIndex);
  ...
  end;
```

```
procedure TLexer.doKeywordOrIdentifier(const Line, Col: Integer);
...
begin
  ...
  Token := TToken.Create(TokenTyp, Lexeme, Null, Line, Col, FReader.FileIndex);
  ...
end;

procedure TLexer.doNumber(const Line, Col: Integer);
...
begin
  ...
  if FLook = LineEnding then
    Errors.Append(TToken.Create(ttNone, '', Null, Line, Col, FReader.FileIndex),
      'Lexer error: String exceeds line.');
  ...
  Token := TToken.Create(ttNumber, Lexeme, Value, Line, Col, FReader.FileIndex);
  ...
end;

procedure TLexer.doString(const Line, Col: Integer);
...
begin
  ...
  Token := TToken.Create(ttString, Lexeme, Value, Line, Col, FReader.FileIndex);
  ...
end;

procedure TLexer.doChar(const Line, Col: Integer);
...
begin
  ...
  Token := TToken.Create(ttChar, '"'+FLook, Value, Line, Col, FReader.FileIndex);
  ...
end;
```

Also, a small change to uParser.pas:

```
function TParser.ParseInheritedExpr: TExpr;
...
begin
  ...
  else if CurrentToken.Typ in [ttInit, ttOpenParen] then begin
    if CurrentToken.Typ = ttOpenParen then
      with CurrentToken do
        Token := TToken.Create(ttIdentifier, 'init', Null, Line, Col, FileIndex)
    else begin
  ...
end;
```

Finally, we need to amend the Error() procedure in the parser:

```
procedure TParser.Error(Token: TToken; Msg: string);
begin
  Errors.Append(Token, Msg);
  Raise EParseError.Create(Msg);
end;
```

This is it! Let's create a separate file arrays.gear and add the following code in there (make sure to save the file in ../gearlib/).

```
extension Array
  val count := length(self)
  val first := listFirst(self)
  val last := listLast(self)
  func add(.value) => listAdd(self, value)
  func insert(at index, .value) => listInsert(self, index, value)
  func delete(.value) => listDelete(self, value)
  func contains(.value) => listContains(self, value)
  func index(of value) => listIndexOf(self, value)
  func retrieve(.index) => listRetrieve(self, index)
end

extension Array

  func reduce(initialValue, nextValue)
    var result := initialValue

    for var i := 0 where i < length(self), i+=1 do
      result := nextValue(result, self[i])
    end
    return result
  end

  func filter(includeElement)
    var result := []
    for var i := 0 where i < length(self), i+=1 do
      if includeElement(self[i]) then
        result.add(value: self[i])
      end
    end
    return result
  end

  func map(transform)
    var result := []
    for var i := 0 where i < length(self), i+=1 do
      result.add(value: transform(self[i]))
    end
    return result
  end

  func forEach(function)
    for var i:=0 where i<length(self), i+=1 do
      function(self[i])
    end
  end

  func sum()
    return self.reduce(0, (x,y)=>x+y)
  end

end
```

Now, in a separate file, put for example the following code:

```
print('Hello world!')
```

```
use arrays

var a := [0,1,2,3,4,5,6,7,8,9].filter(x=>x%2=0)
print(a)  // Array [0, 2, 4, 6, 8]

var b := [0,1,2,3,4,5,6,7,8,9].reduce(0, (x,y)=>x+z)
print(b)  // 45

a.forEach(func(x) print(x) end)  // 0 2 4 6 8
```

A more complex example of what Gear can do is here:

```
// complex example

use arrays

class Person
  var name := ''
  var address := ''
  var age := 0
  var income := 0
  var cars := []

  init(.name, .address, .age, .income, .cars)
    self.name := name
    self.address := address
    self.age := age
    self.income := income
    self.cars := cars
  end

end

var people := [
  Person(name: 'Jerry', address: 'Milano, Italy', age: 39, income: 73000,
    cars: ['Opel Astra', 'Citroen C1']),
  Person(name: 'Cathy', address: 'Berlin, Germany', age: 34, income: 75000,
    cars: ['Audi A3']),
  Person(name: 'Bill', address: 'Brussel, Belgium', age: 48, income: 89000,
    cars: ['Volco XC60', 'Mercedes B', 'Smart 4x4']),
  Person(name: 'Francois', address: 'Lille, France', age: 56, income: 112000,
    cars: ['Volco XC90', 'Jaguar X-type']),
  Person(name: 'Joshua', address: 'Madrid, Spain', age: 43, income: 42000,
    cars: [])
  ]

val names := people.map(person=>person.name)
print(names) // Array [Jerry, Cathy, Bill, Francois, Joshua]

val totalIncome := people.map(person=>person.income).reduce(0, (x,y)=>x+y)
print(totalIncome)  // 391000

val cars := people.map(person=>person.cars)
print(cars)
// prints an array of arrays
// Array [Array [Opel Astra, Citroen C1], Array [Audi A3], Array [Volco XC60, Mercedes B, Smart 4x4],
Array [Volco XC90, Jaguar X-type], Array []]

val flatcars := people.map(person=>person.cars).reduce([],(x,y)=>x><y)
print(flatcars)
// prints a flattened array of cars
// Array [Opel Astra, Citroen C1, Audi A3, Volco XC60, Mercedes B, Smart 4x4, Volco XC90, Jaguar X-
type]

val carList := people.flatMap(person=>person.cars)
print(carList)
// prints a flattened array of cars
// Array [Opel Astra, Citroen C1, Audi A3, Volco XC60, Mercedes B, Smart 4x4, Volco XC90, Jaguar X-
type]
```

Note that flattening an array of arrays also removes the empty arrays!

Using use smart

So far, 'use' is working properly. But what if you accidentilly use the same file twice or more? Or you use for example arrays in two different files of your project? You only need it once in your token list. Luckily, there's an easy fix for this. We just check if the used file was already loaded.

And if this is the case we don't need to reload it again.

> Life solves its problems with well-adapted designs, life-friendly chemistry and smart material and energy use.
>
> — Janine Benyus

```
function TParser.ParseUseStmt: TStmt;
...
begin
  ...
  UsedFile := GearLibFolder + FileName + '.gear';
  if not FileNameArray.Contains(UsedFile) then begin
    try
      Reader := TReader.Create(UsedFile, itFile);
    except
      Error(Token, Format(ErrUseFileError, [FileName]));
    end;
    Lexer := TLexer.Create(Reader);
    //Insert the new tokens in the main tokens list
    Index := Current;
    for i := 0 to Lexer.Tokens.Count-2 do
      Tokens.Insert(Index+i, Lexer.Tokens[i]);
  end;
end;
```

With this in place the exact same file cannot be reloaded more than once. So the following code only parses the arrays exactly once.

```
use arrays
use arrays
var a := [1,2,3,4,5,6,7,8,9].filter(x=>x%2=0)
```

Searching files in the current folder

It's good to have a place for library files, such as arrays.gear in folder /gearlib/, but of course it would be great to also search for a file in the current folder. The current folder is the folder of the main Gear program, or in Gear terminology, the Gear product folder.

> If you do not expect the unexpected you will not find it, for it is not to be reached by search or trail.
>
> — Heraclitus

Therefore, in unit uLanguage define a new global variable GearProdFolder:

```
var
  AppFolder: String = '';
  GearLibFolder: String = '';
  GearProdFolder: String = '';
```

In the main pascal program add the following line to procedure DoRun:

```
procedure TGearLang.DoRun;
...
  if HasOption('f', 'file') then begin
    ...
    GearProdFolder := ExtractFilePath(ExcludeTrailingPathDelimiter(InputFile));
  end;
  ...
end;
```

Variable GearProdFolder now has the path of the folder of the main Gear product.

In the parser function ParseUseStmt first checks for the use file in the current product folder and then in the /gearlib/ folder.

```
function TParser.ParseUseStmt: TStmt;
const Ext = '.gear';
var
  ...
  FileName: String;
  UseFile: String = '';
  ...
begin
  ...
  Result := TUseStmt.Create(FileName, Token);
  if FileExists(GearProdFolder + FileName + Ext) then
    UseFile := GearProdFolder + FileName + Ext
  else if FileExists(GearLibFolder + FileName + Ext) then
    UseFile := GearLibFolder + FileName + Ext
  else Error(Ident.Token, Format(ErrNotExistsUseFile, [FileName]));

  if not FileNameArray.Contains(UseFile) then begin
    try
  ...
  end
end;
```

In the Resolver replace all calls to Error.Append(Token.Line, Token.Col, Message) with Error.Append(Token, Message). For instance:

```
procedure TResolver.VisitVariable(Variable: TVariable);
var
  Symbol: TSymbol;
begin
  Symbol := Retrieve(Variable.Ident);
  if Symbol <> Nil then begin
    if Symbol.Status = Declared then
      Errors.Append(Variable.Token, ErrCannotReadLocalVar);
    ResolveLocal(Variable);
  end;
end;
```

For all appearances of Errors.Append use the token instead of the line and col. Token now contains the file index information which must be passed in order to correctly find and print the error.

Replace in functions VisitSelfExpr(), VisitInheritedExpr(), VisitAssignStmt(), VisitReturnStmt(), VisitClassDecl(), Declare(), Retrieve().

Test it. In file person.gear I have the following:

```
class Person
  var name := ''
  var address := ''
  var age := 0
  var income := 0
  var cars := []

  init(.name, .address, .age, .income, .cars)
    self.name := name
    self.address := address
    self.age := age
    self.income := income
    self.cars := cars
  end
end
```

Then in the main program I can now use this file:

```
use person
use arrays

var people := [
  Person(name: 'Jerry', address: 'Milano, Italy', age: 39, income: 73000,
    cars: ['Opel Astra', 'Citroen C1']),
  Person(name: 'Cathy', address: 'Berlin, Germany', age: 34, income: 75000,
    cars: ['Audi A3']),
  Person(name: 'Bill', address: 'Brussel, Belgium', age: 48, income: 89000,
    cars: ['Volco XC60', 'Mercedes B', 'Smart 4x4']),
  Person(name: 'Francois', address: 'Lille, France', age: 56, income: 112000,
    cars: ['Volco XC90', 'Jaguar X-type']),
  Person(name: 'Joshua', address: 'Madrid, Spain', age: 43, income: 42000,
    cars: [])
  ]


let names := people.map(person=>person.name)
print(names) //  [Jerry, Cathy, Bill, Francois, Joshua]
```

Chapter 13 – Dictionaries

An associative array, map, symbol table, or dictionary (WikiPedia) is an abstract data type composed of a collection of (key, value) pairs, such that each possible key appears at most once in the collection. Operations associated with this data type allow:

- the addition of a pair to the collection
- the removal of a pair from the collection
- the modification of an existing pair
- the lookup of a value associated with a particular key

Dictionary definition

Like an array, a dictionary can also be defined in two ways: as a type instance or as an expression. In the latter case we'll define a standard type Dictionary, for which we'll create a extension with some standard functionality. Here are a few examples:

> Dictionary - opinion expressed as truth in alphabetical order.
>
> — John Ralston Saul

```
dictionary Words
  [:]
end
var words := Words()
words['pair'] := 'couple'
words['collection'] := 'group'
var newDict := [:]
var family := ['Harry': 37, 'Susan': 29, 'John': 8, 'Mary': 5]
```

In the latter two cases, the predefined type Dictionary will be used.

Dictionary type declaration

The EBNF for a basic dictionary type declaration is as follows:

```
DictDecl = 'dictionary' Ident DictElements [ DeclList ] 'end' .
DictElements = '[' ( KeyValuePair { ‘,’ KeyValuePair }  | ':' ) ']' .
KeyValuePair = Expr ‘:’ Expr .
DeclList = Decl { ‘,’ Decl }  .
DictExpr = DictElements .
```

As you can see, when you define a dictionary type, it's also possible to define declarations such as functions and values. Variables and other declarations such as enums and classes are not allowed. The dictionary elements can either be empty [:] or filled with expressions [expr:exper, expr:expr]. A dictionary expression will always be of type Dictionary, like we had type Array for array expressions.

Let's define the AST node for TDictionaryDecl.
First, create a class TKeyValuePair, which holds the key and value parts as expressions.

```
  TKeyValuePair = class
    Key, Value: TExpr;
    constructor Create(AKey, AValue: TExpr);
    destructor Destroy; override;
  end;
```

The key-value pairs are stored in an array of objects, TKeyValueList:

```
  TKeyValueList = specialize TArrayObj<TKeyValuePair>;
```

Next, the TDictDecl class, which is almost the same as the TArrayDecl class.

```
  TDictDecl = class(TDecl)
    private
      FElements: TKeyValueList;
      FDeclList: TDeclList;
    public
      property Elements: TKeyValueList read FElements write FElements;
      property DeclList: TDeclList read FDeclList;
      constructor Create(AIdent: TIdent; ADeclList: TDeclList; AToken: TToken);
      destructor Destroy; override;
      procedure AddElement(KeyExpr, ValueExpr: TExpr);
  end;
```

Then, to the definition of TDecl add a new declaration kind: TDict:

```
  TDecl = class(TNode)
    private
       type TDeclKind =
         (dkArray, dkClass, dkDict, dkExtension, dkFunc, dkTrait, dkVal, dkVar);
```

The implementation of TDictElement and TDictDecl is straightforward:

```
constructor TKeyValuePair.Create(AKey, AValue: TExpr);
begin
  Key := AKey;
  Value := AValue;
end;

destructor TKeyValuePair.Destroy;
begin
  if Assigned(Key) then Key.Free;
  if Assigned(Value) then Value.Free;
  inherited Destroy;
end;
```

```
constructor TDictDecl.Create
  (AIdent: TIdent; ADeclList: TDeclList; AToken: TToken);
begin
  inherited Create(AIdent, dkDict, AToken);
  FElements := TKeyValueList.Create;
  FDeclList := ADeclList;
end;
```

The elements are created as an empty list in the constructor, and not passed as argument.

```
destructor TDictDecl.Destroy;
begin
  if Assigned(FDeclList) then FDeclList.Free;
  FElements.Free;
  inherited Destroy;
end;

procedure TDictDecl.AddElement(KeyExpr, ValueExpr: TExpr);
begin
  FElements.Add(TKeyValuePair.Create(KeyExpr, ValueExpr));
end;
```

Procedure AddElement() adds a key-value pair to the list of dictionary elements. The reason we work with a separate list of key-value pairs and not directly use the FGL map classes for this is that the latter doesn't allow class objects as the key, only regular keys such as integers and strings can be used. We need a much more flexible approach for Gear.

Parsing a dictionary declaration

Much like we did for parsing a list of expressions, we create a parse function to parse a list of key-value pairs, and add that to the parser. We start with a function to parse a key-value pair.

```
function TParser.ParseKeyValuePair: TKeyValuePair;
var
  Key, Value: TExpr;
begin
  Key := ParseExpr;
  Expect(ttColon);
  Value := ParseExpr;
  Result := TKeyValuePair.Create(Key, Value);
end;
```

In fact, the variable 'value' can be omitted and parsed directly in the constructor as a ParseExpr, however, sometimes readability is preferred over less code.

```
function TParser.ParseKeyValueList: TKeyValueList;
begin
  Result := TKeyValueList.Create();
  Result.Add(ParseKeyValuePair);
  while CurrentToken.Typ = ttComma do begin
    Next;  // skip ,
    Result.Add(ParseKeyValuePair);
  end;
end;
```

Moving on to the parse function. We have to cope with different situations. Either dictionary elements are empty or they are filled with key-value pairs, separated by comma's. In the case that the dictionary elements are empty, we expect to have a colon between two brackets [:].

This means we have to test for a colon after the opening bracket.

There's one other thing, now that we are creating small helper routines, and which we need to clean up. Also, for the declarations in the different types such as extensions and arrays, we constantly add the same code, e.g.:

```
  while CurrentToken.Typ in DeclStartSet do
    if CurrentToken.Typ in [ttFunc, ttVal] then
      Result.Add(ParseDecl)
    else
      Error(CurrentToken, Format(ErrUnallowedDeclIn, ['array']));
```

This can be done a lot smarter, much the same as for parsing a list of expressions or key:value pairs. Create a new private function ParseDeclList. Put it right below function ParseDecl.

```
function TParser.ParseDeclList
  (DeclSet: TTokenTypSet; const TypeName: String): TDeclList;
begin
  Result := TDeclList.Create();
  while CurrentToken.Typ in DeclStartSet do
    if CurrentToken.Typ in DeclSet then
      Result.Add(ParseDecl)
    else
      Error(CurrentToken, Format(ErrUnallowedDeclIn, [TypeName]));
end;
```

It accepts as input a set of allowed declaration types such as ttFunc or ttVal, and a type name of the type we are declaring, for instance 'array', or 'extension'. The decl list is then asigned to the declaration type's decl list. Now the parse function of TDictDecl becomes as follows:

```
function TParser.ParseDictDecl: TDecl;
var
  Token: TToken;
  DictDecl: TDictDecl;
begin
  Token := CurrentToken;
  Next;  // skip dictionary
  DictDecl := TDictDecl.Create(ParseIdent, TDeclList.Create(false), Token);
  Expect(ttOpenBrack);
  case CurrentToken.Typ of
    ttColon: Next;
    ttCloseBrack: Expect(ttColon);
    else DictDecl.Elements.Assign(ParseKeyValueList);
  end;
  Expect(ttCloseBrack);
  // other declarations: val and func
  DictDecl.DeclList.Assign(ParseDeclList([ttFunc, ttVal], 'dictionary'));
  Expect(ttEnd);
  Result := DictDecl;
end;
```

Again, see how it follows the EBNF. First, the dictionary keyword, after which we create the dictionary declaration. The ident is parsed directly as an argument to the constructor. Next, we expect the mandatory [and], and in between them we parse any key:value pair, or the empty case, just a colon ':'. In case there are func and val definitions we parse them and close off with the 'end'. Note that the dictionary can be empty!

```
dictionary Empty [] end
```

Of course, you can now replace the same functionality in functions ParseExtensionDecl, ParseTraitDecl, ParseArrayDecl. Here they are for completeness:

```
function TParser.ParseExtensionDecl: TDecl;
var
  Token: TToken;
  Extension: TExtensionDecl;
begin
  Token := CurrentToken;
  Next;  // skip extension
  Extension := TExtensionDecl.Create(ParseIdent, TDeclList.Create(false), Token);
  Extension.DeclList.Assign(ParseDeclList([ttFunc, ttVal], 'extension'));
  Expect(ttEnd);
  Result := Extension;
end;
```

```
function TParser.ParseTraitDecl: TDecl;
var
  Token: TToken;
  Ident: TIdent;
  Traits: TExprList;
  DeclList: TDeclList;
begin
  Token := CurrentToken;
  Next; // skip trait
  Ident := ParseIdent;
  Traits := TExprList.Create();
  if CurrentToken.Typ = ttColon then begin
    Next; // skip :
    Traits := ParseExprList;
  end;
  DeclList := TDeclList.Create();
  DeclList.Assign(ParseDeclList([ttFunc], 'trait'));
  Expect(ttEnd);
  Result := TTraitDecl.Create(Ident, Traits, DeclList, Token);
end;
```

In traits we only allow functions and not calculated values (yet).

```
function TParser.ParseArrayDecl: TDecl;
var
  Token: TToken;
  ArrayDecl: TArrayDecl;
begin
  Token := CurrentToken;
  Next;  // skip array
  ArrayDecl := TArrayDecl.Create(ParseIdent, TDeclList.Create(false), Token);
  Expect(ttOpenBrack);
  if CurrentToken.Typ <> ttCloseBrack then
    ArrayDecl.Elements.Assign(ParseExprList);
  Expect(ttCloseBrack);
  ArrayDecl.DeclList.Assign(ParseDeclList([ttFunc, ttVal], 'array'));
  Expect(ttEnd);
  Result := ArrayDecl;
end;
```

In the parser add the token to the constant DeclStartSet:

```
const
  DeclStartSet: TTokenTypSet = [ttArray, ttClass, ttDictionary,
    ttExtension, ttFunc, ttLet, ttVal, ttVar, ttTrait];
```

In function ParseDecl, add the case for parsing dictionary declarations (use the same order as in the DeclStartSet):

```
function TParser.ParseDecl: TDecl;
begin
  case CurrentToken.Typ of
    ...
    ttDictionary: Result := ParseDictDecl;
    ...
  end;
end;
```

The printer visitor is as follows:

```
procedure TPrinter.VisitDictDecl(DictDecl: TDictDecl);
var
  Element: TKeyValuePair;
  Decl: TDecl;
begin
  IncIndent;
  VisitNode(DictDecl);
  Visit(DictDecl.Ident);
  WriteLn(Indent + 'Elements:');
  IncIndent;
  for Element in DictDecl.Elements do begin
    Write(Indent + 'Key: '); Visit(Element.Key);
    Write(Indent + 'Val: '); Visit(Element.Value);
  end;
  WriteLn(Indent + 'Declarations:');
  for Decl in DictDecl.DeclList do
    Visit(Decl);
  DecIndent;
  DecIndent;
end;
```

For the resolver visitor, first add a new class kind ‘ckDictionary’ to TClassKind:

```
  TClassKind = (ckNone, ckClass, ckSubClass, ckExtension, ckTrait,
    ckArray, ckDictionary);
```

To make sure we don’t use inherited in a function inside an dictionary declaration, we need a small adjustment to VisitInheritedExpr:

```
procedure TResolver.VisitInheritedExpr(InheritedExpr: TInheritedExpr);
begin
  if CurrentClassKind in [ckNone, ckTrait, ckArray, ckDictionary] then
    Errors.Append(InheritedExpr.Token, ErrInheritedInClass)
  else if CurrentClassKind <> ckSubClass then
    Errors.Append(InheritedExpr.Token, ErrInheritedNotInSubClass);
  ResolveLocal(InheritedExpr.Variable);
end;
```

Then, the resolver visitor becomes:

```
procedure TResolver.VisitDictDecl(DictDecl: TDictDecl);
var
  Element: TKeyValuePair;
  Decl: TDecl;
  EnclosingClassKind: TClassKind;
begin
  for Element in DictDecl.Elements do begin
    Visit(Element.Key);
    Visit(Element.Value);
  end;
  if Assigned(DictDecl.Ident) then begin
    Declare(DictDecl.Ident);
    Enable(DictDecl.Ident);
    EnclosingClassKind := CurrentClassKind;
    CurrentClassKind := ckDictionary;
    BeginScope;
    Scopes.Top.Enter(TSymbol.Create('self', Enabled));
    for Decl in DictDecl.DeclList do begin
      case Decl.Kind of
        dkFunc: ResolveFunction(Decl as TFuncDecl, fkMethod);
        dkVal: ResolveFunction((Decl as TValDecl).FuncDecl, fkMethod);
      end;
    end;
    EndScope;
    CurrentClassKind := EnclosingClassKind;
  end;
end;
```

We visit the elements' key and value pairs and declare and enable the dictionary identifier if assigned, then set the classkind to ckDictionary. Next open a new scope, insert 'self' and visit any func or val. Finally, close the scope and set the class kind back to the original.

Note that if the identifier is not assigned, we skip resolving any internal declaration. In such case there's no need. If we later on create the standard Dictionary type we'll add declarations through extensions.

Interpreting dictionaries

The same practice applies as we used for interpreting arrays. We start high level and first create the interpreter's visitor for dictionary declarations, and as usual, we create a new unit uDictionary, which defines a dictionary class and dito instance.

Also, we need two more interfaces: IDictionable and IDictInstance. Create them in a new unit file uDictIntf.pas.

```
unit uDictIntf;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, uCallable, uAST, Variants, uMembers, uCollections, uToken;
```

```
type

  IDictionable = interface(ICallable)
    ['{F461B1D0-B563-695D-455C-A3606F6DE48F}']
    procedure ExtendWith(Members: TMembers);
  end;

  TDictElements = specialize TDictionary<Variant, Variant>;

  IDictInstance = Interface
    ['{A467BC62-AC8F-F5F2-6691-E2586512EE21}']
    function GetMember(Ident: TIdent): Variant;
  end;

implementation

end.
```

First the basics, with procedure ExtendWith for IDictionable and GetMember for IDictInstance, which we know we'll ne needing anyway, but we'll add more functionalty to the interfaces soon. Also, just like we did for arrays, add a new type TDictElements which holds the visited keys and values.

Next, the visitor method, which is roughly a copy from the array declaration visitor.

```
procedure TInterpreter.VisitDictDecl(DictDecl: TDictDecl);
var
  DictClass: TDictClass;
  Element: TKeyValuePair;
  Elements: TDictElements;
  Members: TMembers;
begin
  CheckDuplicate(DictDecl.Ident, 'Dictionary');
  Elements := TDictElements.Create;
  for Element in DictDecl.Elements do
    Elements[Visit(Element.Key)] := Visit(Element.Value);
  Members := getMembers(DictDecl.DeclList);
  DictClass := TDictClass.Create(DictDecl.Ident, Elements, Members);
  CurrentSpace.Store(DictDecl.Ident, IDictionable(DictClass));
end;
```

The difference with the array visitor is the visiting of the elements' key-value pairs.

We are not done yet with the interpreter, as we'll have to change VisitCallExpr and VisitGetExpr as well later on. First, we create a new unit, and work on the internals of TDictClass.

Dictionary class

Create a new unit uDictionary.pas:

```
unit uDict;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, uAST, uError, uToken, Variants, uInterpreter,
  uCallable, uDictIntf, uMembers;
```

Then, we'll create a new type TDictClass, which implements interface IDictionable. The DictClass holds the initial elements, the methods and value properties. It has already support for adding new methods and properties through extensions.

```
type

  TDictClass = class(TInterfacedObject, IDictionable)
    private
      Ident: TIdent;
      Elements: TDictElements;
      Methods: TMethodTable;
      Values: TValueTable;
      function FindMethod(Instance: IDictInstance; const AName: String): ICallable;
      function FindValuable(const AName: String): IValuable;
    public
      constructor Create(AIdent: TIdent; AElements: TDictElements;
        Members: TMembers);
      function Call(Token: TToken; Interpreter: TInterpreter;
        ArgList: TArgList): Variant;
      function toString: string; override;
      procedure ExtendWith(Members: TMembers);
  end;
```

It is close to the TArrayClass definition. Note that FElements is of type TDictElements, which is defined in the uDictIntf unit.

In the implementation section add the following uses clause. uVariantSupport is needed for printing variants.

```
implementation
uses uFunc, uVariantSupport;
```

The constructor is straightforward and initializes the fields. The members contain the respective methods and values.

```
constructor TDictClass.Create
  (AIdent: TIdent; AElements: TDictElements; Members: TMembers);
begin
  Ident := AIdent;
  Elements := AElements;
  Methods := Members.Methods;
  Values := Members.Values;
end;
```

In this first version of Call, we don't accept any arguments yet, to start of simple:

```
function TDictClass.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Instance: IDictInstance;
begin
  Instance := IDictInstance(TDictInstance.Create(Self));
  Instance.Elements.Assign(Self.Elements)
  Result := Instance;
end;
```

We do add the elements that are defined in the dictionary definition, like this:

```
dictionary Words
  ['pair':'couple', 'collection':'group', 'hello':'world']
end
var words := Words()
```

Variabe 'words' now contains: ['pair': 'couple', 'collection': 'group', 'hello': 'world'], and print(words) will print the contents of the dictionary.

Function toString just creates a string of the name of the dictionary:

```
function TDictClass.toString: string;
begin
  Result := Ident.Text;
end;
```

Next, we'll look at the functions FindMethod and FindValue, which are mere copies of the ones introduced in uArray.

```
function TDictClass.FindMethod(Instance: IDictInstance; const AName: String
  ): ICallable;
var
  Index: integer = -1;
begin
  Result := Nil;
  if Methods.Contains(AName, Index) then
    Result := (ICallable(Methods.At(Index)) as TFunc).Bind(Instance)
end;

function TDictClass.FindValuable(const AName: String): IValuable;
var
  Index: LongInt = -1;
begin
  Result := Nil;
  if Values.Contains(AName, Index) then
    Result := Values.At(Index);
end;
```

These will find the methods and/or values defined in the dictionary definition, or in a dictionary extension. Now we're talking about extensions, here is the procedure used to add the methods and values to the dictionary as an extension.

```
procedure TDictClass.ExtendWith(Members: TMembers);
var
  i: Integer;
begin
  for i := 0 to Members.Methods.Count-1 do
    Methods[Members.Methods.Keys[i]] := Members.Methods.Data[i];
  for i := 0 to Members.Values.Count-1 do
    Values[Members.Values.Keys[i]] := Members.Values.Data[i];
end;
```

Now let's move on to the dictionary instance.

Dictionary instance

Just like we did for arrays we also create an instance of a dictionary.

```
  TDictInstance = class(TInterfacedObject, IDictInstance)
    private
      DictClass: TDictClass;
      FElements: TDictElements;
      function getCount: LongInt;
      function getElements: TDictElements;
      function getItem(i: integer): Variant;
      procedure setItem(i: integer; AValue: Variant);
    public
      property Elements: TDictElements read getElements;
      property Count: LongInt read getCount;
      property Items[i:integer]: Variant read getItem write setItem; default;
      constructor Create(ADictClass: TDictClass);
      destructor Destroy; override;
      function toString: string; override;
      function GetMember(Ident: TIdent): Variant;
      function TypeName: String;
  end;
```

The privately defined methods are:

```
function TDictInstance.getCount: LongInt;
begin
  Result := FElements.Count;
end;

function TDictInstance.getElements: TDictElements;
begin
  Result := FElements;
end;

function TDictInstance.getItem(i: integer): Variant;
begin
  Result := FElements[i];
end;

procedure TDictInstance.setItem(i: integer; AValue: Variant);
begin
  FElements[i] := AValue;
end;
```

The constructor takes a dictionary class as parameter, so that we know to which dictionary type.

```
constructor TDictInstance.Create(ADictClass: TDictClass);
begin
  DictClass := ADictClass;
  FElements := TDictElements.Create;
end;
```

The destructor frees the elements.

```
destructor TDictInstance.Destroy;
begin
  FElements.Free;
  inherited Destroy;
end;
```

The toString function gets a special treatment here. In order to show that elements of a dictionary (both key and value) are of a string type, we need to put single quotes around them. The current way of printing strings does not do that. If you want to print quoted strings you have to do that yourself. Hence, for dictionaries we need a little help.

```
function TDictInstance.toString: string;
var
  i: Integer;

  function getStr(Value: Variant): String;
  begin
    Result := Value.toString;
    if VarType(Value) = varString then
      Result := QuotedStr(Result);
  end;

begin
  Result := ' [';
  if FElements.Count > 0 then begin
    for i := 0 to FElements.Count-2 do
      Result += getStr(FElements.Keys[i]) + ': ' +
                getStr(FElements.Data[i]) + ', ';
    Result += getStr(FElements.Keys[FElements.Count-1]) + ': ' +
              getStr(FElements.Data[FElements.Count-1]);
  end
  else Result += ':';
  Result += ']';
end;
```

The nested function getStr() first creates a string of the value, and then determines if the value's type itself is String, and if so puts single quotes, using the standard Free Pascal function QuotedStr().

An example of such printed dictionary instance is:

```
['pair': 'couple', 'collection': 'group', 'hello': 'world']
```

The last element is stringyfied separately, in order not to get an additional comma behind it!

The function getMember tries to find the right method or value property:

```
function TDictInstance.GetMember(Ident: TIdent): Variant;
var
  Method: ICallable;
  Valuable: IValuable;
begin
  Method := DictClass.FindMethod(IDictInstance(Self), Ident.Text);
  if Method <> Nil then
    Exit(Method);

  Valuable := DictClass.FindValuable(Ident.Text);
  if Valuable <> Nil then
    Exit(Valuable);

  Raise ERuntimeError.Create(Ident.Token,
    'Undefined dictionary member "' + Ident.Text + '".');
end;
```

Function typeName returns the type name of a dictionary instance:

```
function TDictInstance.TypeName: String;
begin
  Result := DictClass.Ident.Text;
end;
```

Next, the promised few changes to the interpreter's VisitCallExpr and VisitGetExpr.
In the interpreter's VisitCallExpr add the interface IDictionable to the VarSupportsIntf function:

```
function TInterpreter.VisitCallExpr(CallExpr: TCallExpr): Variant;
...
begin
  Callee := Visit(CallExpr.Callee);
  if VarSupportsIntf(Callee,
       [ICallable, IClassable, IArrayable, IDictionable]) then
    Func := ICallable(Callee)
  else begin
  ...
end;
```

And in VisitGetExpr add a new if-then statement:

```
function TInterpreter.VisitGetExpr(GetExpr: TGetExpr): Variant;
...
begin
  ...
  if VarSupportsIntf(Instance,
       [IGearInstance, IArrayInstance, IDictInstance]) then
  begin
    ...
    else if VarSupports(Instance, IDictInstance) then
      Result := IDictInstance(Instance).GetMember(GetExpr.Ident);

    ...
  end
  else
    ...
end;
```

Finally, the interface IDictInstance should now look like this:

```
  TDictElements = specialize TDictionary<Variant, Variant>;

  IDictInstance = Interface
    ['{A467BC62-AC8F-F5F2-6691-E2586512EE21}']
    function getCount: LongInt;
    function getElements: TDictElements;
    property Elements: TDictElements read getElements;
    property Count: LongInt read getCount;
    function GetMember(Ident: TIdent): Variant;
    function TypeName: String;
  end;
```

In the interpreter add to the uses clause in the implementation the units uDictIntf and uDict:

```
implementation
uses uCallable, uFunc, uStandard, uClassIntf, uClass, uArrayIntf, uArray,
  uDictIntf, uDict, uMath, uVariantSupport;
```

Testing…

```
dictionary Words
  ['pair':'couple', 'collection':'group']

  func write()
    print(self)
  end
end

var words := Words()
print(words)           // ['pair': 'couple', 'collection': 'group']

words.write()          // ['pair': 'couple', 'collection': 'group']
```

Adding extensions

To make it possible to add extensions to dictionary types, change method VisitExtensionDecl to the following:

```
procedure TInterpreter.VisitExtensionDecl(ExtensionDecl: TExtensionDecl);
var
  TypeDecl: Variant;
  Members: TMembers;
begin
  TypeDecl := CurrentSpace.Load(ExtensionDecl.Ident);  // get actual type
  if not VarSupportsIntf(TypeDecl, [IClassable, IArrayable, IDictionable]) then
    Raise ERuntimeError.Create(ExtensionDecl.Ident.Token,
      Format(ErrExtIdNoType, [ExtensionDecl.Ident.Text]));
  Members := getMembers(ExtensionDecl.DeclList);
  if VarSupports(TypeDecl, IClassable) then
    IClassable(TypeDecl).ExtendWith(Members)
  else if VarSupports(TypeDecl, IArrayable) then
    IClassable(TypeDecl).ExtendWith(Members)
  else if VarSupports(TypeDecl, IDictionable) then
    IDictionable(TypeDecl).ExtendWith(Members);
end;
```

Yes, given our previous preparations, it's that simple to add extensions for a dictionary type.

Again testing this:

```
dictionary Words
  ['pair':'couple', 'collection':'group']
end

var words := Words()
print(words)

extension Words
  func write()
    print(self)
  end

  val asString
    return self
  end
end

words.write()
print(words.asString)
```

Which prints:

```
 ['pair': 'couple', 'collection': 'group']
 ['pair': 'couple', 'collection': 'group']
 ['pair': 'couple', 'collection': 'group']
```

Standard Dictionary type and expression

So far, you could define your own dictionary type and create new dictionary expressions based on such types. However, we would need to have a standard dictionary type with standard added functionality, which is also availabe on standard declared dictionary expressions. For example, we can create a standard type:

```
  dictionary Dictionary [:] end
```

and make expressions from this type:

```
  var a := [1:'one', 2:'two', 3:'three', 4:'four', 5:'five']
```

to be of type Dictionary.

How can we arrange this? By defining a standard dictionary type in the Globals memory. We apply the same trick as we used for arrays in this respect.

```
constructor TInterpreter.Create;
begin
  ...

  // store base Dictionary type
  Globals.Store('Dictionary', IDictionable(
    TDictClass.Create(
      TIdent.Create(TToken.Create(ttIdentifier, 'Dictionary', Null, 0, 0)),
      TDictElements.Create(),
      TMembers.Create(TFieldTable.Create, TConstTable.Create,
        TMethodTable.Create, TValueTable.Create
      )
    ))
  );
  CurrentSpace:= FGlobals;
end;
```

Next, we have to create functionality for Dictionary expressions. First the AST node for dictionary expressions, the same way as for array expressions.

```
  TDictDeclExpr = class(TFactorExpr)
    private
      FDictDecl: TDictDecl;
    public
      property DictDecl: TDictDecl read FDictDecl;
      constructor Create(ADictDecl: TDictDecl);
      destructor Destroy; override;
  end;


constructor TDictDeclExpr.Create(ADictDecl: TDictDecl);
begin
  inherited Create(ADictDecl.Token);
  FDictDecl := ADictDecl;
end;

destructor TDictDeclExpr.Destroy;
begin
  if Assigned(FDictDecl) then FDictDecl.Free;
  inherited Destroy;
end;
```

The printer visitor:

```
procedure TPrinter.VisitDictDecExprl(DictDeclExpr: TDictDeclExpr);
begin
  IncIndent;
  VisitNode(DictDeclExpr);
  Visit(DictDeclExpr.DictDecl);
  DecIndent;
end;
```

and the Resolver visitor:

```
procedure TResolver.VisitDictDeclExpr(DictDeclExpr: TDictDeclExpr);
begin
  Visit(DictDeclExpr.DictDecl);
end;
```

Then, move on to the parser. We cannot just parse a dictionary expression based on an opening bracket '[', because we don't know at this moment if it is an array or a dictionary. That's why we create a new parser function ParseBracketExpr, which will handle the choices. It must handle empty arrays [] or dictionaries [:], and filled ones as well of course. Let's try.

```
function TParser.ParseBracketExpr: TExpr;
var
  Expr: TExpr;
  ArrayDecl: TArrayDecl;
  DictDecl: TDictDecl;
begin
  Next; // skip [
  case CurrentToken.Typ of
    ttCloseBrack: begin    // an empty array
      ArrayDecl := TArrayDecl.Create(Nil, TDeclList.Create(false), CurrentToken);
      Result := TArrayDeclExpr.Create(ArrayDecl);
    end;
    ttColon: begin        // an empty dictionary
      DictDecl := TDictDecl.Create(Nil,TDeclList.Create(false), CurrentToken);
      Result := TDictDeclExpr.Create(DictDecl);
      Next;  // skip :
    end
    else                 // a filled array or dictionary
      begin
        Expr := ParseExpr;       // parse the first expr
        if CurrentToken.Typ = ttColon then    // then check for ':'
          Result := ParseDictDeclExpr(Expr)   // a dictionary; pass the first expr
        else
          Result := ParseArrayDeclExpr(Expr); // an array; pass the first expr
      end;
  end;
  Expect(ttCloseBrack);   // final closing bracket
end;
```

This parses as we expect, empty arrays, empty dictionaries and filled ones. We do have to change the current ParseFactor and current ParseArrayDeclExpr. The latter one now takes an expression as argument. In ParseFactor change the following line:

```
    ttOpenBrack: Result := ParseArrayDeclExpr;
```

with:

```
    ttOpenBrack: Result := ParseBracketExpr; // [ array or dictionary
```

The parser for array expression changes to:

```
function TParser.ParseArrayDeclExpr(FirstExpr: TExpr): TExpr;
var
  ArrayDecl: TArrayDecl;
begin
  ArrayDecl := TArrayDecl.Create(Nil, TDeclList.Create(false), CurrentToken);
  ArrayDecl.AddElement(FirstExpr);
  while CurrentToken.Typ = ttComma do begin
    Next;  // skip ,
    ArrayDecl.AddElement(ParseExpr);
  end;
  Result := TArrayDeclExpr.Create(ArrayDecl);
end;
```

The first expression is added to the list and then we parse while there are comma's.

For dictionary expressions the parse function is:

```
function TParser.ParseDictDeclExpr(KeyExpr: TExpr): TExpr;
var
  DictDecl: TDictDecl;
  Key: TExpr;
begin
  DictDecl := TDictDecl.Create(Nil, TDeclList.Create(false), CurrentToken);
  Expect(ttColon);
  DictDecl.AddElement(KeyExpr, ParseExpr);
  while CurrentToken.Typ = ttComma do begin
    Next;  // skip ,
    Key := ParseExpr;
    Expect(ttColon);
    DictDecl.AddElement(Key, ParseExpr);
  end;
  Result := TDictDeclExpr.Create(DictDecl);
end;
```

The same idea, but now we parse key:value pairs.

There are a couple of things left to do, amongst others the inerpreter visitor for DictDeclExpr. We want dictionary expressions to be of type Dictionary.

```
function TInterpreter.VisitDictDeclExpr(DictDeclExpr: TDictDeclExpr): Variant;
var
  DictType: IDictionable;
  Instance: IDictInstance;
  i: Integer;
begin
  DictType := IDictionable(Globals['Dictionary']);

  Instance := IDictInstance(TDictInstance.Create(DictType as TDictClass));
  for i := 0 to DictDeclExpr.DictDecl.Elements.Count-1 do
    Instance.Elements.Add(
      Visit(DictDeclExpr.DictDecl.Elements[i].Key),
      Visit(DictDeclExpr.DictDecl.Elements[i].Value));

  Result := Instance;
end;
```

We retrieve the dictionary type from the Globals memory and then create an instance of the Dict type. Next, we visit all elements and store them in the instance's elements. Finally, the instance is returned as the function result.

We have to add 'Dictionary' to the resolver's EnableStandardFunctions procedure:

```
procedure TResolver.EnableStandardFunctions;
begin
  with GlobalScope do begin
    ...
    Enter(TSymbol.Create('Dictionary', Enabled, False));
  end;
end;
```

Now this works:

```
var a := ['pair':'couple', 'collection':'group']
print(a)
```

Standard functions for dictionary

A dictionary wouldn't be very useful if it didn't have some standard functions available to the programmer in order to manipulate the contents.

> The standard library saves programmers from having to reinvent the wheel.
>
> — Bjarne Stroustrup

Specifically, you would like to add, insert and delete items, as well as search on contents and even retrieve items, given a certain key. So, let's define the following set of standard functions that will be available to dictionaries.

- add, which adds a key-value pair to a dictionary,
- insert, which inserts a key:value at a certain position,
- delete, which deletes an item from a dictionary,
- indexOf, which gets the index of an item,
- contains, which returns True if an item is in the list, and
- retrieve, which gets the value of a key.

Updating the interface and instance

All these functions operate on the instance of a dictionary, so let's add them to the interface.

```
  IDictInstance = Interface
    ...
    procedure Add(const AKey, AValue: Variant);
    procedure Insert(const i: Integer; const AKey, AValue: Variant; Token: TToken);
    procedure Delete(const AValue: Variant; Token: TToken);
    function Contains(const AKey: Variant): Boolean;
    function IndexOf(const AKey: Variant): Variant;
    function Retrieve(const AKey: Variant; Token: TToken): Variant;
  end;
```

Of course this means we have to implement the methods in TDictInstance in uDict.pas. The simplest one is add, which just calls FElement.Add():

```
procedure TDictInstance.Add(const AKey, AValue: Variant);
begin
  FElements.Add(AKey, AValue);
end;
```

Insert is a bit more difficult, as you can only insert items within the current range of items. If you try to insert outside the range an error is generated. If we would not catch the error ourselves, the underlying pascal code will generate a runtime error, without further information than 'list out of bounds'.

```
procedure TDictInstance.Insert(const i: Integer;
  const AKey, AValue: Variant; Token: TToken);
begin
  if (i >= 0) and (i <= getCount) then
    FElements.InsertKeyData(i, AKey, AValue)
  else
```

```
    Raise ERuntimeError.Create(Token,
      'Index ('+ IntToStr(i) + ') out of range ('+ '0,' +
      IntToStr(getCount-1) + ') during insert action.');
end;
```

For the delete function I was doubting whether to generate an error when an item cannot be deleted if it doesn't exist, or do nothing at all. But if we do nothing, and don't show an error, it will be very difficult to detect what went wrong, so I chose to present an error message:

```
procedure TDictInstance.Delete(const AValue: Variant; Token: TToken);
var
  i: Integer;
begin
  i := FElements.Remove(AValue);
  if i<0 then
    Raise ERuntimeError.Create(Token,
      'Value "' + AValue.toString + '" not found in delete action.');
end;
```

Function 'contains' is again straightforward and returns true if a dictionary contains the respective key:

```
function TDictInstance.Contains(const AKey: Variant): Boolean;
begin
  Result := FElements.Contains(AKey);
end;
```

The function IndexOf returns the index of a given key, or returns Null if it was not found.

```
function TDictInstance.IndexOf(const AKey: Variant): Variant;
begin
  Result := FElements.IndexOf(AKey);
  if Result < 0 then
    Result := Unassigned;
end;
```

Finally, the function retrieve, which retrieves the value of a dictionary element with a given key:

```
function TDictInstance.Retrieve(const AKey: Variant; Token: TToken): Variant;
begin
  if FElements.Contains(AKey) then
    Result := FElements[AKey]
  else
    Result := Unassigned;
end;
```

If the key doesn't exist, the value Unassigned is returned. This makes it possible to create code like this:

```
ensure var m := numbers.retrieve(key: 66) where Assigned(m) else
  print('Alas')
end
print(m)
```

Modify functions in the runtime library

It's time to set the functions to use for dictionaries as well.

Each call function takes as first parameter the value of 'self', which is either an array instance or a dictionary instance. If it is none of both a runtime error is generated, like this:

```
  Instance := ArgList[0].Value;                          // self as value
  if VarSupports(Instance, IArrayInstance) then begin
    // action on array
  end
  else if VarSupports(Instance, IDictInstance) then begin
    // action on dictionary
  end
  else
    Raise ERuntimeError.Create(Token, Message);
```

This principle is used for all List functions below.

Add uDictIntf to the uses clause of uStandard:

```
implementation
uses uArrayIntf, uDictIntf;
```

Add

The add function takes 2 parameters for an array (self and value), and 3 for a dictionary (self and key, value). It then calls the function available through the interface with the given arguments.

```
function TListAdd.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
...
begin
  ...
  else if VarSupports(Instance, IDictInstance) then begin
    TFunc.CheckArity(Token, ArgList.Count, 3);
    IDictInstance(Instance).Add(ArgList[1].Value, ArgList[2].Value);
    Result := IDictInstance(Instance);
  end
  else
    ...
end;
```

Insert

Function Insert has an extra parameter compared to add: the index where to insert the item. In case of arrays the index and the value, and in case of a dictionary, the index and key, value pair.

```
function TListInsert.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
...
begin
  ...
  else if VarSupports(Instance, IDictInstance) then begin
    TFunc.CheckArity(Token, ArgList.Count, 4);
    IDictInstance(Instance).Insert(
      ArgList[1].Value, ArgList[2].Value, ArgList[3].Value);
    Result := IDictInstance(Instance);
  end
  else
    ...
end;
```

Delete

Function Delete calls the instance's delete function with the given value for an array or key for a dictionary.

```
function TListDelete.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
...
begin
  ...
  else if VarSupports(Instance, IDictInstance) then begin
    TFunc.CheckArity(Token, ArgList.Count, 2);
    IDictInstance(Instance).Delete(ArgList[1].Value);
    Result := IDictInstance(Instance);
  end
  else
    ...
end;
```

Contains

Function Contains gets its result from the call to the instance's Contains function.

```
function TListContains.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
...
begin
  ...
  else if VarSupports(Instance, IDictInstance) then
    Result := IDictInstance(Instance).Contains(ArgList[1].Value)
  else
    ...
end;
```

IndexOf

Function IndexOf gets the index of a value from an array or the index of a key from a dictionary.

```
function TListIndexOf.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
...
begin
  ...
  else if VarSupports(Instance, IDictInstance) then
    Result := IDictInstance(Instance).IndexOf(ArgList[1].Value)
  else
    ...
end;
```

Retrieve

Lastly, the retrieve function, which takes an index in case of an array and a key in case of a dictionary.

```
function TListRetrieve.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
...
begin
  ...
  else if VarSupports(Instance, IDictInstance) then
    Result := IDictInstance(Instance).Retrieve(ArgList[1].Value)
  else
    ...
end;
```

Trying the new code

First, try to add the following extension for Array to file arrays.gear:

Then, it will be possible to do the following for arrays:

```
use arrays

var numbers := [1,2,3,4,5,6,7,8,9]
print(numbers)

numbers.insert(at:0, value:0)
print(numbers)

numbers.add(value: 10)
print(numbers)
print(numbers.first)
print(numbers.last)

print(numbers.index(of:7))
numbers.insert(at:3, value:3)
print(numbers)

numbers.delete(value:3)
print(numbers)
print(numbers.contains(value: 3))
print(numbers.retrieve(index: 3))
```

And for dictionaries:

```
extension Dictionary
  func toString() => self

  func add(.key, .value) => listAdd(self, key, value)
  func insert(at index, .key, .value) => listInsert(self, index, key, value)
  func delete(.key) => listDelete(self, key)
  func contains(.key) => listContains(self, key)
  func index(of key) => listIndexOf(self, key)
  func retrieve(.key) => listRetrieve(self, key)
end

var numbers := [:]

numbers.add(key: 0, value: 'Zero')
numbers.add(key: 1, value: 'One')
numbers.add(key: 2, value: 'Two')
numbers.add(key: 3, value: 'Three')
numbers.add(key: 4, value: 'Four')
numbers.add(key: 5, value: 'Five')
numbers.add(key: 6, value: 'Six')
numbers.add(key: 7, value: 'Seven')
numbers.add(key: 8, value: 'Eight')
numbers.add(key: 9, value: 'Nine')
numbers.add(key: 10, value: 'Ten')

print(numbers)

print(numbers.index(of: 7))
print(numbers.retrieve(key: 8))

print(numbers.retrieve(key: numbers.index(of: 8)))
numbers.insert(at:5, key: 11, value: 'eleven')
print(numbers)
print(numbers.retrieve(key: numbers.index(of: 8)))

print(numbers[5])

numbers[11] := 'not eleven'
print(numbers)
print(numbers[11])

//print(numbers.retrieve(key: 15)) error
print(numbers.index(of:18))

ensure var m := numbers.retrieve(key: 66) where m <> Null else
  print('Alas')
end
print(m)
```

Accessing dictionary elements

Though we have defined a function dictionaryType.retrieve(key:), it should also be possible to just do dictionaryType[key] to get the value belonging to that particular key.

For this we need to rewrite the interpreter visitor for IndexedExpr. Next to recognizing indexes for arrays and strings, it should also recognize dictionary indexes, which may or may not be an integer index.

```
function TInterpreter.VisitIndexedExpr(IndexedExpr: TIndexedExpr): Variant;
var
  Instance: Variant;
  Index: Variant;
begin
  Instance := Visit(IndexedExpr.Variable);
  Index := Visit(IndexedExpr.Index);
  if VarSupports(Instance, IArrayInstance) then begin
    CheckNumericIndex(Index, IndexedExpr.Index.Token);
    Result := IArrayInstance(Instance).get(Index, IndexedExpr.Index.Token);
  end
  else if VarSupports(Instance, IDictInstance) then
    Result := IDictInstance(Instance).get(Index, IndexedExpr.Index.Token)
  else if VarIsStr(Instance) then begin
    CheckNumericIndex(Index, IndexedExpr.Index.Token);
    Result := String(Instance)[Index+1];  // strings start at index 1
  end
  else
    Raise ERuntimeError.Create(IndexedExpr.Variable.Token, ErrArrayTypeExpected);
end;
```

For array and string indexing we need to make sure that the index is an integer. In order to check this we use procedure CheckNumericIndex(). The type of Index is a variant and can be anything, which is required for the dictionary.

Setting an array or dictionary index happens in VisitIndexedExprStmt(). So far, there is no solution for assigning new values to strings with index. Let's include that here as well.

For both arrays and strings we require an integer number as the index, so we'll check for that through procedure CheckNumericIndex().

For dictionaries this can be any key, thus not specifically numbers.

In all cases we first get the old value, followed by determining the new value. Then we set the new value in memory.

```
procedure TInterpreter.VisitIndexedExprStmt(IndexedExprStmt: TIndexedExprStmt);
var
  Variable, OldValue, NewValue, Value: Variant;
  Index: Variant;
  VarAsStr: String;
begin
  Variable := Visit(IndexedExprStmt.IndexedExpr.Variable);
  Index := Visit(IndexedExprStmt.IndexedExpr.Index);
  NewValue := Visit(IndexedExprStmt.Expr);

  with IndexedExprStmt do
  if VarSupports(Variable, IArrayInstance) then begin
    CheckNumericIndex(Index, IndexedExpr.Index.Token);
    OldValue := IArrayInstance(Variable).get(Index, IndexedExpr.Index.Token);
    Value := getAssignValue(OldValue, NewValue, IndexedExpr.Variable.Token, Op);
    IArrayInstance(Variable).Put(Index, Value, IndexedExpr.Index.Token);
  end
  else if VarSupports(Variable, IDictInstance) then begin
    OldValue := IDictInstance(Variable).get(Index, IndexedExpr.Index.Token);
    Value := getAssignValue(OldValue, NewValue, IndexedExpr.Variable.Token, Op);
    IDictInstance(Variable).Put(Index, Value, IndexedExpr.Index.Token);
  end
  else if VarIsStr(Variable) then begin
    CheckNumericIndex(Index, IndexedExpr.Index.Token);
    VarAsStr := Variable;
    OldValue := VarAsStr[Index+1];
    VarAsStr[Index+1] := getAssignValue(OldValue, NewValue,
      IndexedExpr.Variable.Token, Op);
    Assign(IndexedExpr.Variable as TVariable, VarAsStr);
  end
  else
    Raise ERuntimeError.Create(IndexedExpr.Variable.Token, ErrArrayTypeExpected);
end;
```

For the dictionary instance to be able to process this, we need to add functions get and put to the interface and instance:

```
  IDictInstance = Interface
    ['{A467BC62-AC8F-F5F2-6691-E2586512EE21}']
    ...
    function Get(Key: Variant; Token: TToken): Variant;
    procedure Put(Key: Variant; Value: Variant; Token: TToken);
  end;
```

```
function TDictInstance.Get(Key: Variant; Token: TToken): Variant;
begin
  if FElements.Contains(Key) then
    Result := FElements[Key]
  else
    Raise ERuntimeError.Create(Token,
      'Key ('+ Key.toString + ') not found.');
end;
```

```
procedure TDictInstance.Put(Key: Variant; Value: Variant; Token: TToken);
begin
  try
    FElements[Key] := Value
  except
    Raise ERuntimeError.Create(Token,
      'Key:Value ('+ Key.toString + ', ' + Value.toString + ') wrong.');
  end;
end;
```

And now code like this is possible:

```
dictionary Words
  ['pair':'couple', 'oma':'grandmother', 'man':'male', 'woman':'female']
end

var words := Words()
print(words)

words['oma'] := 'grootmoeder'
print(words)
```

Result:

```
['pair': 'couple', 'oma': 'grandmother', 'man': 'male', 'woman': 'female']
['pair': 'couple', 'oma': 'grootmoeder', 'man': 'male', 'woman': 'female']
```

Assigned and ?

If you want to know the first item of an array, by asking for variable.first, the array tries to get variable[0], which results in a value if the array is filled, however produces a runtime error if the array is empty. This may possibly be an unwanted situation, if it crashes your application.
Let's look at the case of the first item. In uStandard.pas we have:

```
function TListFirst.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Instance: Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 1);
  Instance := ArgList[0].Value;
  if VarSupports(Instance, IArrayInstance) then begin
    if IArrayInstance(Instance).Count > 0 then
      Result := IArrayInstance(Instance)[0]
    else Result := Unassigned;
  end
  else if VarSupports(Instance, IDictInstance) then begin
    if IDictInstance(Instance).Count > 0 then
      Result := IDictInstance(Instance).getElements.Keys[0]
    else Result := Unassigned;
  end
  else
    Raise ERuntimeError.Create(Token,
      'listFirst function not possible for this type.');
end;
```

Why do we return Unassigned? Think about it, what would happen if we return Null. Null is just a value, which could even be added as the first item in a list. Then there would be no difference between an empty list and the first list item, when testing for emptyness.

Function listFirst takes as an argument and array or a dictionary. If the number of items in the list is greater than zero (>0) the first item is returned of an array or the first key is returned of a dictionary. If the list is empty, the value Unassigned is returned. This value however is unknown as yet in Gear. So how will we test for a value that is unassigned? Or rather test if it is assigned…
Create a new standard function class, which returns True if the value is assigned:

```
  TAssigned = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;
```

```
function TAssigned.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
begin
  Result := ArgList[0].Value <> Unassigned;
end;
```

Then, add the following line to the interpreter's constructor:

```
  Globals.Store('assigned', ICallable(TAssigned.Create));
```

Add the name also to the resolver:

```
    Enter(TSymbol.Create('assigned', Enabled, False));
```

Now it will be possible to run the following test:

```
use arrays
var a := []
if assigned(a.first) then
  print('assigned')
end
ensure assigned(a.first) else
  print('not assigned')
end
```

Since we're actually questioning the variable's first item whether it is assigned, we could also intoduce an operator for this. Behold the question mark '?'.

```
if ?a.first then
  print('assigned')
else
  print('unassigned')
end

ensure ?a.first else
  print('it is not assigned')
end
```

This is actually very simple to build in. In the parser add the token type ttQuestion to if-statement in ParseUnaryExpr:

```
function TParser.ParseUnaryExpr: TExpr;
...
begin
  if CurrentToken.Typ in [ttPlus, ttNot, ttMin, ttQuestion] then begin
...
end;
```

Then, in the inerpreter add the case for ttQuestion to VisitUnaryExpr:

```
function TInterpreter.VisitUnaryExpr(Node: TUnaryExpr): Variant;
...
begin
  ...
  case Node.Op.Typ of
    ...
    ttQuestion: Result := Expr <> Unassigned;
    else Result := Expr;
  end;
end;
```

Note that I check for Expr <> Unassigned. This is because we only have a Pascal available function for Unassigned. Above code now works! Cool huh?

This can really catch a lot of runtime errors, e.g. ?a[0].

The Array instance's get() function now returns a runtime out of range error if the array you're trying to access is empty. This can easily be solved by checking for the number of items in the array, and if this is zero, return the Unassigned value.

```
function TArrayInstance.Get(i: Integer; Token: TToken): Variant;
begin
  if getCount = 0 then
    Result := Unassigned
  else if (i >= 0) and (i < getCount) then
    Result := FElements[i]
  else
    Raise ERuntimeError.Create(Token,
      'Index ('+ IntToStr(i) + ') out of range ('+ '0,' + IntToStr(getCount-1) + ').');
end;
```

Create a file dictionaries.gear, and save it in /gearlib/ folder.

```
extension Dictionary

  val count := length(self)
  val first := listFirst(self)
  val last := listLast(self)

  func toString() => self

  func add(.key, .value) => listAdd(self, key, value)
  func insert(at index, .key, .value) => listInsert(self, index, key, value)
  func delete(.key) => listDelete(self, key)
  func contains(.key) => listContains(self, key)
  func index(of key) => listIndexOf(self, key)
  func retrieve(.key) => listRetrieve(self, key)
end
```

Then add use dictionaries to system.gear:

```
use arrays
use ranges
use dictionaries
```

In arrays.gear change the val first and val last to:

```
  val first := listFirst(self)
  val last := listLast(self)
```

Then it’s possible now to do things like this:

```
use system

var a := [1,2,3,4,5]
print(a)

var d := [1:'one', 2:'two', 3:'three']
print(d)

a.add(value:6)

d.add(key:4, value:'four')

print(a)
print(d)

var aa := []
var dd := [:]

print(aa)
print(dd)
print(a.first)
print(a.last)
```

```
print(d.first)  // first key
print(d.last)   // last key
```

Resulting in:

```
 [1, 2, 3, 4, 5]
 [1: 'one', 2: 'two', 3: 'three']
 [1, 2, 3, 4, 5, 6]
 [1: 'one', 2: 'two', 3: 'three', 4: 'four']
 []
 [:]
 1
 6
 1
 4
```

And testing the 'assignedness' of array/dictionary entries:

```
ensure ?dd.first else
  print('dd is unassigned')
end
print(dd.first)

var l := []
if assigned(l.first) then
  print(l.first)
else
  print('not assigned')
end

ensure assigned(l.first) else
  print('not assigned')
end

if ?l.first then
  print('assigned')
else
  print('unassigned')
end

ensure ?l.first else
  print('it is not assigned')
end

ensure ?l[1] else
  print('it is not assigned')
end
```

And if you switch on range checking automatically errors are generated at runtime.

gear execute -r <filename>

Another example of Gear's dictionary capabilities:

```
var operators :=
  ['+': (x,y)=>x+y,
   '-': (x,y)=>x-y,
   '*': (x,y)=>x*y,
   '/': (x,y)=>x/y,
   '%': (x,y)=>x%y]

var o := '*'

var x := operators[o](7,8)
print(x)
```

```
print(operators['+'](8,7))
print(operators['-'](8,7))
print(operators['*'](8,7))
print(operators['/'](8,7))
print(operators['%'](8,7))
```

Chapter 14 – Iterators and list comprehension

An iterator is a class (instance) that enables traversing a container, in particular lists. Iteration can be done implicitly by using e.g. for each constructs, or explicitly by creating an explicit iterator.

The Array iterator

> Design is an iterative process. One idea often builds on another.
> — Mark Parker

An example of implicit iteration is:

```
var list := [0,1,2,3,4,5,6,7,8,9]
for each item in list do
  print(item)
end
```

The biggest advantage of an implicit iteration is its readability.

In principle a for each statement can be translated to a normal while loop, and you'll get an explicit iteration, like this:

```
var list := [0,1,2,3,4,5,6,7,8,9]
while var index := 0 where index < length(list) do
  var item := list[index]
  print(item)
  index += 1
end
```

Though a clear and good solution, it doesn't fit in all cases. E.g. try to create a similar solution for traversing enums or other types of collections. Another solution to this is to add iterator classes and an iterator function to collection classes. An example of such an explicit iterator is for example:

```
var list := [0,1,2,3,4,5,6,7,8,9]
while var iterator := list.iterator where iterator.hasNext do
  var item := iterator.next
  print(item)
end
```

For this to properly work we have to create an iterator object and an extension to Array. So, if you do that for all collection types in the same way, you have a generic solution.

```
class ArrayIterator
  code
end

extension Array
  val iterator := ArrayIterator(self)
end
```

That seems like a lot of work for just creating an explicit iterator, but the trick is to use the implicit iterator of course, but in such a generic way, that it can be used for all collection types that adhere to the iterator protocol.

This means for each collection type you create an iterator class, that can be initialized and contains a hasNext and a next 'val'. Then, create an extension to the collection type that contains the iterator value, which returns the iterator object.

For example the same iterator can also be used on an array of string:

```
var animals := ['Giraffe', 'Lion', 'Elephant', 'Monkey', 'Dolphin']

for each animal in animals do
  print(animal)
end
```

This becomes:

```
while var iterator := animals.iterator where iterator.hasNext do
  var animal := iterator.next
  print(animal)
end
```

So all we have to do is translate a for-each statement with implicit iterator to a while statement with explicit iterator. Piece of cake, right?

Creating the iterator object

An iterator object is always written in pure Gear code, and should adhere to the following rules:

- it must have an init(list/range_type) that takes e.g. an array or other type as parameter;
- it must have a 'hasNext' value property that returns a boolean value;
- it must have a 'next' value property, which returns the next item in the list.

It is important that the cases of 'hasNext' and 'next' match exactly!

Then, the (extension to the) type must have a value property 'iterator' that returns an initialized iterator class. In the case of type Arrays, this is:

```
class ArrayIterator
  var list := Null
  var index := 0

  init(list)
    self.list := list
  end

  val hasNext := self.index < length(self.list)

  val next
    var nextItem := self.list [self.index]
    self.index +=1
    return nextItem
  end
end

extension Array
  val iterator := ArrayIterator(self)
end
```

In the code 'hasNext' checks whether there is still a next element by checking the index value against the length of the array. If the index is greater that the array length, a boolean False is returned.

Value property 'next' returnes the array item with current index. The index is increased after the assignment to nextItem. Finally, nextItem is returned as value.

The extension to Array is very simple. A value property 'iterator' gets its value by calling/initializing the array iterator with parameter 'self'.

Copy the above code for ArrayIterator and extension Array to the bottom of file 'arrays.gear', located in folder /gearlib/.

Translate from implicit to explicit iterator

Recall the implicit Gear code for an iterator loop:

```
for each LoopIdent in ListExpr do
  block using Ident
end
```

The keyword 'each' is introduced to distinct from a normal 'for var item :=' statement. Each has the same meaning as Var in this case: to introduce a new variable. So after 'each' do not use 'var'! It can only be used in this construct.

The above has to be translated into:

```
while var iterator := ListExpr.iterator where iterator.hasNext do
  var LoopIdent := iterator.next
  block
end
```

As a first step, change TParser.ParseForStmt, so that it also parses the 'each' keyword.

```
function TParser.ParseForStmt: TStmt;
var
  Token: TToken;
  VarDecl: TVarDecl;
  Condition: TExpr;
  Iterator: TStmt;
  Block: TBlock;
begin
  Token := CurrentToken;
  Next; // skip for
  if CurrentToken.Typ = ttEach then
    Result := ParseForEachStmt
  else begin
    VarDecl := ParseVarDecl as TVarDecl;
    Expect(ttWhere);
    Condition := ParseExpr;
    Expect(ttComma);
    Iterator := ParseAssignStmt;
    Expect(ttDo);
    Block := ParseBlock;
    Block.Nodes.Add(Iterator);
    Result := TWhileStmt.Create(VarDecl, Condition, Block, Token);
  end;
  Expect(ttEnd);
end;
```

After skipping the 'for' keyword, we check for 'each'. If found we parse the ForEachStmt. If not, we continue with the original 'for var Ident :=' loop. Finally, we expect the 'end' keyword.

The new parsing function ParseForEachStmt must be added to TParser's interface and it's implementation can be right below ParseForStmt.

First, create a small private helper function to create identifiers:

```
function TParser.MakeID(Name: String): TIdent;
begin
  with CurrentToken do
    Result := TIdent.Create(TToken.Create(
      ttIdentifier, Name, Null, Line, Col, FileIndex));
end;
```

```
function TParser.ParseForEachStmt: TStmt;
var
  Condition: TGetExpr;
  IterDecl, VarDecl: TVarDecl;
  Block: TBlock;
  LoopIdent: TIdent;
  ListExpr: TExpr;
begin
  Next; // skip each
  // parse the for-loop variable
  LoopIdent := ParseIdent;
  Expect(ttIn);
  // parse list type expression
  ListExpr := ParseExpr;
  Expect(ttDo);

  // create initial iterator vardecl: var iterator := ListExpr.iterator
  IterDecl := TVarDecl.Create(MakeID('iterator'),
    TGetExpr.Create(ListExpr, MakeID('iterator')), Nil);

  // build condition: iterator.hasNext
  Condition := TGetExpr.Create(
    TVariable.Create(MakeID('iterator')), MakeID('hasNext'));

  // build var LoopIdent := iterator.next
  VarDecl := TVarDecl.Create(LoopIdent,
    TGetExpr.Create(TVariable.Create(MakeID('iterator')), MakeID('next')), Nil);

  // parse the block
  Block := ParseBlock;
  Block.Nodes.Insert(0, VarDecl);   // insert VarDecl at start of block

  Result := TWhileStmt.Create(IterDecl, Condition, Block, Nil);
end;
```

It has a small helper function MakeID(name) to create TIdent variables on the fly, just by passing a name.

After skipping 'each', we parse the loop ident, then expect 'in', followed by parsing the list expression, followed by 'do'. Now we have all items parsed except the block. The translation part starts here.

First: var iterator := ListExpr.iterator, which is stored in variable IterDecl, which is a TVarDecl taking three arguments: a TIdent based on 'iterator', and a expression 'ListExpr.iterator'. The latter is a TGetExpr. The third argument is a Token, which we pass as 'Nil' here.

In the same manner we create the boolean Condition of the while-stmt: iterator.hasNext.

The next variable declaration is for the LoopIdent, which is initialized to 'iterator.next'.

Then, we parse the block. The VarDecl (of the LoopIdent) however must be the first node in the parsed block. So, we insert it at position zero in the block's nodes.

Finally, we create the WhileStmt and return that as the function's result.

Since it only concerns a parsing action, a translation to the existing TWhileStmt, we don't have to change anything further in our code. But, it only works if the iterator class and extension to Array is added to arrays.gear. If not the code will not run.

Compile and run it:

```
use arrays

var animals := ['Giraffe', 'Lion', 'Elephant', 'Monkey', 'Dolphin']

for each animal in animals do
  print(animal)
end


for each i in ["a, 123, 3.1415, pi()] do
  print(i)
end


var list := [0,1,2,3,4,5,6,7,8,9,10]

for each item in list do
  print(item)
end


for each item in Array(1,2,3,4,5,6,7,8,9) do
  print(item)
end
```

Range class and iterator

> My range is limited.
> — Bob Dylan

To show you that the above is a powerful concept, which doesn't only apply to arrays, you can find an implementation of some class Range, which is completely coded in Gear and needs no adaptation in the underlying code.

A range usually runs 'from' a starting number 'to' an ending number, like in Range(1,10), where it runs from 1 to 10, including 10.

First step, is to define an Iterator class:

```
class RangeIterator
  var index := 0
  var range := Null

  init(aRange)
    self.range := aRange
    self.index := aRange.from
  end

  val hasNext := self.index <= self.range.to

  val next
    var result := self.index
    self.index +=1
    return result
  end
end
```

The variable 'index' keeps track of the position, an is initialized to 0. Variable range is first set to Null. This is important as the final type will only be available in the init(), since the Range class is defined after the RangeIterator class.

In the init(), the index is set to range.from.

In the value property hasNext we check if index is less than or equal to range.to, and returns True if it does, otherwise False.

In the 'next' property, we put the index to the result variable and then increase index by 1. The result is returned.

Now we define the Range class:

```
class Range
  var from := Null
  var to := Null

  init(from, to)
    self.from := from
    self.to := to
  end

  val iterator := RangeIterator(self)
end
```

We have the 'from' and 'to' variables, which get their values in the init(). Then, we have the mandatory iterator property, which returns an instance of RangeIterator.

Now we can do this:

```
for each number in Range(3,8) do
  print(number^2)
end
```
And this is pure Gear code!!!

It's also possible to transform the range to an array, very easy:

Create an extension to class Range, like this:

```
extension Range
  func toArray()
    var result := []
    for each item in self do
      result := result >< [item]
    end
    return result
  end

  func filter(includeElement)
    var result := []
    for each item in self where includeElement(item) do
      result := result >< [item]
    end
    return result
  end
end
```

You can combine the range functionality into a file called ranges.gear and save it to the /gearlib/ folder. All you have to do is 'use ranges'.

Now you can use it like this:

```
use arrays
use ranges

for each number in Range(1,20).filter(x=>x%2=0) do
  print(number)
end
```

It prints :
```
2
4
6
etc
```

Adding a where clause

While the latter construct of Range(1,20).filter(x=>x%2=0) is working, it has a few drawbacks:

- it is complex, thus hard to read and understand
- it is cumbersome

> Do not go where the path may lead, go instead where there is no path and leave a trail.
>
> — Ralph Waldo Emerson

It would be nicer if we could add a where clause to the 'for each' statement, like this:

```
for each item in Range(1,20) where item%2=0 do
  print(item)
end
```

This should only print the even numbers in the range. What we have to do is translate this to the under the hood iterator construct, so that we get:

```
while var iterator := Range(1,20).iterator where iterator.hasNext do
  var item := iterator.next
  if item%2=0 then
    print(item)
  end
end
```

So, all we have to do is turn the where-clause into an if-stmt that takes the statement block as its then-part. It turns out this is not so hard to do. Here are the parser changes:

```
function TParser.ParseForEachStmt: TStmt;
var
  ...
  WhereExpr: TExpr = Nil;
  IfStmt: TIfStmt = Nil;
begin
  ...
  if CurrentToken.Typ = ttWhere then begin
    Next;
    WhereExpr := ParseExpr;
  end;
  Expect(ttDo);

  ...

  if Assigned(WhereExpr) then begin
    IfStmt := TIfStmt.Create(Nil, WhereExpr,
      TBlock.Create(TNodeList.Create(), Nil), Nil, Nil);
    IfStmt.ThenPart := ParseBlock;
    Block := TBlock.Create(TNodeList.Create(), Nil);
    Block.Nodes.Add(VarDecl);
    Block.Nodes.Add(IfStmt);
  end else begin
    Block := ParseBlock;
    Block.Nodes.Insert(0, VarDecl);   // insert VarDecl at start of block
  end;
  Result := TWhileStmt.Create(IterDecl, Condition, Block, Nil);
end;
```

If the keyword 'where' is encountered the WhereExpr is parsed. Later on where the block is created we test if WhereExpr is assigned to, and if so, create an if-statement, with a lot of Nil values. Those values are not important here. The then-part of the if-statement is a block also, and it should contain the full block code in it. For this to work, you have to make a small change to the AST node for TIfStmt:

```
  TIfStmt = class(TStmt)
      ...
    public
      ...
      property ThenPart: TBlock read FThenPart write FThenPart;
      ...
  end;
```

You have to make the ThenPart writable.

Next, an empty Block is created, and we add the VarDecl and IfStmt consecutively.

If the 'where' keyword was not encountered, we do it like before.

Compile and test.

```
use arrays
use ranges

var list := [0,1,2,3,4,5,6,7,8,9,10]

for each item in list where item%2=0 do
  print(item)
end

for each item in Range(1,20) where item%2=0 do
  print(item)
end

for each item in ['ape', 'dog', 'cat', 'cow'] do
  print(item)
end
```

Another test with animals as smileys ☺:

```
use arrays

var smileys := ['🐶', '🐔', '🐵', '🦁', '😺', '🐭', '🐱', '🐷', '🐽']

for each smiley in smileys do
  print(smiley)
end
```

With this in place the for-each statement has become a powerful iterator.

List comprehension

A list comprehension is a syntactic construct available in some programming languages for creating a list based on existing lists. It follows the form of the mathematical set-builder notation (set comprehension) as distinct from the use of map and filter functions (WikiPedia).

> Comprehension follows perception.
> — Philip K. Dick

Given an array filled with integers from 1 to 15, you can apply a filter and map action on this:

```
inputSet = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

Using the set builder notation:

```
result = { 2x | x ∈ inputSet, x² > 3 }
```

Which gives for result and array:

```
[4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30]
```

In terms of Gear code this could be done using filter and map:

```
var result := inputSet.filter(x=>x^2>3).map(x=>2*x)

print(result) // Array [4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30]
```

It is possible to create a function that simulates a list comprehension:

```
use arrays

func listComp(apply transform, on input, when predicate)
  return input.filter(predicate).map(transform)
end

var list := [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]

print(listComp(apply: x=>2*x, on: list, when: x=>x^2>3))
```

Alternatively, one can use the flatMap func (given the same list):

```
var evens := list.flatMap(n => if n%2=0 then [n] else [])
print(evens)

var evensSquared := list.flatMap(n => if n%2=0 then [n*n] else [])
print(evensSquared)
```

But, we'll try to implement a form of the set builder. Instead of using '|' as separator we use the keyword 'for', and for '∈' we use 'in' and for the comma, we'll use 'where'. Like this:

```
var result := { 2*x for x in inputSet where x^2>3 }
```

or rather:

```
var result := { OutputFuncExpr for Ident in InputSet where PredicateFuncExpr }
```

In Gear language, I created library 'system.gear', which contains now two simple use clauses:

```
use arrays
use ranges
```

To this we'll add a function that simulates the set builder structure. Though I call it listBuilder, simply because it always returns a list. The Gear code for this function is:

```
func listBuilder(transform, input, include)
  var result := []
  if include <> Null then
    for each item in input where include(item) do
      listAdd(result, transform(item))
    end
  else
    for each item in input do
      listAdd(result, transform(item))
    end
  end
  return result
end
```

This small func is all you need to create a list of a certain input (array or Range), whereby the items are transformed, only if they are included. If the where-clause is empty (include=Null) we use the for each without where clause. For example:

```
var x := listBuilder(x=>x^2, [1,2,3,4,5,6], x=>x%2=0)
// x is [4, 16, 36]
```

So, all we have to do is translate

```
var x := { x^2 for x in [1,2,3,4,5,6] where x%2=0 }
```

into a call to func listBuilder.

The good thing is that we only have to create a parser function for this. Next, step by step, we go through the code. The first step is to parse the items of the list builder structure.

```
function TParser.ParseSetBuilderExpr: TExpr;
var
  OutputExpr, InputSet: TExpr;
  PredicateExpr: TExpr = Nil;
  Ident: TIdent;
  OutputFuncExpr, PredicateFuncExpr: TFuncDeclExpr;
  ListBuilder: TCallExpr;
begin
  // first step: parse all elements of set builder expression
  Next;                                    // '{'
  OutputExpr := ParseExpr;                 // x^2
  Expect(ttFor);                           // 'for'
  Ident := ParseIdent;                     // x
  Expect(ttIn);                            // 'in'
  InputSet := ParseExpr;                   // [1,2,3,4,5,6]
  if CurrentToken.Typ = ttWhere then begin
    Next; // skip where                    // 'where'
    PredicateExpr := ParseExpr;            // x%2=0
  end;
  Expect(ttCloseBrace);                    // '}'
```

In this step we parsed all elements of the construct and stored them in the respective variables. Note that the where-clause is optional. The above shown func listBuilder tests for the optional being of func variable 'include'.

In the second step we create the respective functions x=>x^2 and x=>x%2=0.

They are created as anonymous functions, so the Ident is Nil. The found identifier in the above (the 'x' in our example) is added as a parameter to both functions. The expressions are transformed into return statements and added to the body of the functions.

```
  // second step: create the functions
  OutputFuncExpr := TFuncDeclExpr.Create(TFuncDecl.Create(Nil, Nil));
  with OutputFuncExpr do begin
    FuncDecl.AddParam(Ident, Nil);
    FuncDecl.Body := TBlock.Create(TNodeList.Create(), Nil);
    FuncDecl.Body.Nodes.Add(TReturnStmt.Create(OutputExpr, Nil));
  end;

  if Assigned(PredicateExpr) then begin
    PredicateFuncExpr := TFuncDeclExpr.Create(TFuncDecl.Create(Nil, Nil));
    with PredicateFuncExpr do begin
      FuncDecl.AddParam(Ident, Nil);
      FuncDecl.Body := TBlock.Create(TNodeList.Create(), Nil);
      FuncDecl.Body.Nodes.Add(TReturnStmt.Create(PredicateExpr, Nil));
    end;
  end;
```

Note that where normally the Token has a value, I now assigned it Nil. This is allowed and also means we don't have to free the memory later on. Only the tokens added via the Lexer are freed automatically.

In the third step we create the call expression that calls func listBuilder. We use function MakeID to create the Ident listBuilder. Next, we add the above created functions and the InputSet expression as arguments to the listBuilder call expression. Make sure that in case the where-clause (PredicateExpr) is empty, we add the constant Null for the predicate.

```
  // third step: create call to func listBuilder(transform, input, include)
  ListBuilder := TCallExpr.Create(TVariable.Create(MakeID('listBuilder')), Nil);
  ListBuilder.AddArgument(OutputFuncExpr, Nil);
  ListBuilder.AddArgument(InputSet, Nil);
  if Assigned(PredicateExpr) then
    ListBuilder.AddArgument(PredicateFuncExpr, Nil)
  else
    ListBuilder.AddArgument(TConstExpr.Create(Null, Nil), Nil);
  Result := ListBuilder;
end;
```

This should do the trick. As said, create a new library 'system.gear', with the above code.

Now you can test the code with e.g.:

```
use system

var list := { x^2 for x in [1,2,3,4,5,6] where x%2=0 }
print(list)

list := { x^2 for x in [1,2,3,4,5,6] }
```

```
print(list)

var newList := { x^2 for x in Range(1,10) where x%2=0 }
print(newList)


print({x^3 for x in Range(1,10)})
```

Result:

```
[4, 16, 36]
[1, 4, 9, 16, 25, 36]
[4, 16, 36, 64, 100]
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

This ends the chapter on iterators, which gave us iterators on arrays and ranges, as well as the 'list builder' expression that provides a convenient way of creating a list.

Chapter 15 – Enumeration

An enumerated type (also called enumeration or enum,) is a data type consisting of a set of named values called elements, members, or enumerators of the type. The enumerator names are usually identifiers that behave as constants in the language. A variable that has been declared as having an enumerated type can be assigned any of the enumerators as a value. In other words, an enumerated type has values that are different from each other, and that can be compared and assigned.

In our language Gear an enumeration is declared as starting with the keyword 'enum'. A few examples of its declaration are shown below.

```
enum Color
  (Red, Blue, Yellow, Green, Orange, Purple)
end
```

This is the simplest version whereby a simple enumeration of elements is given. A variable that uses this enum is declared like this:

```
var myColor := Color.Blue
print(myColor.name)   // prints 'Blue'
print(Color.count)    // prints '6'
```

As you see an enum will have a default field called 'name' that holds the string representation of an enum plus a field that holds the number of items: count.

You can also define an emum with values, for example if you want to print different values than the name, like this:

```
enum TokenType
  (Plus='+', Min='-', Mul='*', Div='/', Rem='%')
end
var tokenType := TokenType.Mul
print(tokenType.name)   // prints 'Mul'
print(tokenType.value)  // prints '*'
```

In the next phase we'll introduce the keyword case followed by the set-name, like this:

```
enum Color
  (None
   case Primary: Red, Blue, Yellow
   case Secondary: Green, Orange, Purple)
end

var myColor := Color.Blue
if myColor in Color.Primary then
  print(myColor.name, ' is a primary color.')  // "Blue is a primary color."
end
```

The enum basics

An enum thus holds a list of names and optional values and is supported with functions and values as members. Variables and constants are not allowed. Let's put together the EBNF of an enum declaration.

EBNF of EnumDecl:

> Happy indeed is the scientist who not only has the pleasures which I have enumerated, but who also wins the recognition of fellow scientists and of the mankind which ultimately benefits from his endeavors.
>
> — Irving Langmuir

```
EnumDecl = 'enum' Ident '(' EnumElements ')' [ DeclList ] end .
EnumElements = EnumElement { EnumElement } .
EnumElement = Name [ '=' Value ] .
Name = String constant .
Value = String constant | Character constant | Number constant .
```

We apply this in the parser and create the following AST node for EnumDecl:

```
  TEnumElements = specialize TDictionary<String, Variant>;

  TEnumDecl = class(TDecl)
    private
      FElements: TEnumElements;
      FDeclList: TDeclList;
    public
      property Elements: TEnumElements read FElements;
      property DeclList: TDeclList read FDeclList;
      constructor Create(AIdent: TIdent; ADeclList: TDeclList; AToken: TToken);
      destructor Destroy; override;
      procedure AddElement(const Name: String; Value: Variant);
  end;
```

```
constructor TEnumDecl.Create
  (AIdent: TIdent; ADeclList: TDeclList; AToken: TToken);
begin
  inherited Create(AIdent, dkEnum, AToken);
  FElements := TEnumElements.Create;
  FDeclList := ADeclList;
end;

destructor TEnumDecl.Destroy;
begin
  if Assigned(FDeclList) then FDeclList.Free;
  FElements.Free;
  inherited Destroy;
end;

procedure TEnumDecl.AddElement(const Name: String; Value: Variant);
begin
  FElements.Add(Name, Value);
end;
```

The property Elements holds the names and values of the enum, whereby it is associated with a Dictionary of String and Variant name:value pairs.

In the definition of TDecl add dkEnum to type TDeclKind.

```
  TDecl = class(TNode)
    private
       type TDeclKind = (dkArray, dkClass, dkDict, dkEnum, dkExtension,
         dkFunc, dkTrait, dkVal, dkVar);
```

Now let's continue with the parser. First add the token type ttEnum to the declaration start set:

```
const
  DeclStartSet: TTokenTypSet = [ttArray, ttClass, ttDictionary, ttEnum,
    ttExtension, ttFunc, ttLet, ttVal, ttVar, ttTrait];
```

Then, add the parse function to method ParseDecl:

```
function TParser.ParseDecl: TDecl;
...
begin
  case CurrentToken.Typ of
    ...
    ttEnum: Result := ParseEnumDecl;
    ...
  end;
end;
```

Now create the parse function. In its simplest form the method is as follows:

```
function TParser.ParseEnumDecl: TDecl;
var
  EnumDecl: TEnumDecl;
  Token: TToken;
begin
  Token := CurrentToken;
  Next;  // skip enum
  EnumDecl := TEnumDecl.Create(ParseIdent, Token);
  Expect(ttOpenParen);
  if CurrentToken.Typ <> ttCloseParen then begin
    EnumDecl.AddElement(ParseIdent.Text, 0);  // at least expect 1 enumeration
    while CurrentToken.Typ = ttComma do begin
      Next;  // skip ,
      EnumDecl.AddElement(ParseIdent.Text, 0);
    end;
  end
  else Error(CurrentToken, ErrAtLeastOneEnum);
  Expect(ttCloseParen);
  // other declarations: val and func
  EnumDecl.DeclList.Assign(ParseDeclList([ttFunc, ttVal], 'enum'));
  Expect(ttEnd);
  Result := EnumDecl;
end;
```

It creates an enum declaration and parses the identifier of the enum. Then an '(' token is expected. Technically, this is not required, but esthetically it's better. Compare these:

```
enum Color Blue, Red, Yellow end
enum Color (Blue, Red, Yellow) end
```

At least one enum is required. If an empty list is detected, and error is generated. Add to the errors:

```
  ErrAtLeastOneEnum = 'At least one enum is required in declaration.';
```

Then, while commas are found, we continue to parse enums. At the end we expect a closing parenthesis, and the keyword end. The full enum declaration finally is returned as the function result. For now all values are set to 0, but we'll change that soon.

At this moment the 'end' keyword is not yet required, however when we start adding 'func's and 'val's we need a closing keyword.

Next, the printer and resolver visitors.

```
procedure TPrinter.VisitEnumDecl(EnumDecl: TEnumDecl);
var
  i: Integer;
  Decl: TDecl;
begin
  IncIndent;
  VisitNode(EnumDecl);
  Visit(EnumDecl.Ident);
  IncIndent;
  WriteLn(Indent + 'Elements:');
  IncIndent;
  for i := 0 to EnumDecl.Elements.Count-1 do begin
    Writeln(Indent, 'Name: ', EnumDecl.Elements.Keys[i], ', Value: ',
            EnumDecl.Elements.Data[i]);
  end;
  DecIndent;
  WriteLn(Indent + 'Declarations:');
  for Decl in EnumDecl.DeclList do
    Visit(Decl);
  DecIndent;
  DecIndent;
end;
```

It simply prints the elements, including their values, line by line.

In the Resolver, first add ckEnum to the type class kind:

```
  TClassKind = (ckNone, ckClass, ckSubClass, ckExtension, ckTrait,
    ckArray, ckDictionary, ckEnum);
```

The visitor for the resolver:

```
procedure TResolver.VisitEnumDecl(EnumDecl: TEnumDecl);
var
  Decl: TDecl;
  EnclosingClassKind: TClassKind;
begin
  Declare(EnumDecl.Ident);
  Enable(EnumDecl.Ident);
  EnclosingClassKind := CurrentClassKind;
  CurrentClassKind := ckEnum;
  BeginScope;
  Scopes.Top.Enter(TSymbol.Create('self', Enabled));
  for Decl in EnumDecl.DeclList do begin
    case Decl.Kind of
      dkFunc: ResolveFunction(Decl as TFuncDecl, fkMethod);
      dkVal: ResolveFunction((Decl as TValDecl).FuncDecl, fkMethod);
    end;
  end;
  EndScope;
  CurrentClassKind := EnclosingClassKind;
end;
```

Make sure to adjust the visitor for inherited expression, so that 'inherited' cannot be called from enums.

```
procedure TResolver.VisitInheritedExpr(InheritedExpr: TInheritedExpr);
begin
  if CurrentClassKind in [ckNone, ckTrait, ckArray, ckDictionary, ckEnum] then
    Errors.Append(InheritedExpr.Token, ErrInheritedInClass)
  ...
end;
```

And the interpreter is empty for now:

```
procedure TInterpreter.VisitEnumDecl(EnumDecl: TEnumDecl);
begin
  // do nothing
end;
```

Adding raw values

We can test the simplest form now, but let's go one step further to enable the use of raw values with the enums. And also note, it's now allowed to repeat names in enums. Of course this should not be the case. So, the new parse function will be:

```
function TParser.ParseEnumDecl: TDecl;
var
  EnumDecl: TEnumDecl;
  Token: TToken;

  procedure AddNameValue;
  var
    Name: TIdent;
    Value: Variant;
  begin
    Name := ParseIdent;
    if EnumDecl.Elements.Contains(Name.Text) then
      Error(Name.Token, Format(ErrDuplicateEnum, [Name.Text]));
    Value := Null;
    if CurrentToken.Typ = ttEQ then begin
      Next;  // skip =
      if CurrentToken.Typ in [ttNumber, ttString, ttChar] then begin
        Value := CurrentToken.Value;
        Next;
      end
      else
        Error(CurrentToken, Format(ErrNotAllowedInEnum, [CurrentToken.Lexeme]));
    end;
    EnumDecl.AddElement(Name.Text, Value);
  end;

begin
  ...
  if CurrentToken.Typ <> ttCloseParen then begin
    AddNameValue;  // at least expect 1 enumeration
    while CurrentToken.Typ = ttComma do begin
      Next;  // skip ,
      AddNameValue;
    end;
  end
  ...
end;
```

The nested procedure takes care of reading the name and the value. And it checks for unallowed constants and duplicate enum names. Add the following error messages:

```
  ErrDuplicateEnum = 'Duplicate enum name "%s".';
  ErrNotAllowedInEnum = 'Constant value "%s" not allowed in enum declaration.';
```

Now we are ready to create an interpreter visitor for EnumDecl.

Interpreting enums

Let's move back to the Interpreter. We'll fill in the method VisitEnumDecl. We need to do a couple of things. Most important is to store the Enum in the memory space. We copy the elements from the AST node into the memory space as well.

```
procedure TInterpreter.VisitEnumDecl(EnumDecl: TEnumDecl);
var
  Enum: TEnumClass;
  Members: TMembers;
begin
  CheckDuplicate(EnumDecl.Ident, 'Enum');
  Members := getMembers(EnumDecl.DeclList);
  Enum := TEnumClass.Create(EnumDecl.Ident, EnumDecl.Elements.Copy, Members);
  CurrentSpace.Store(EnumDecl.Ident, IEnumable(Enum));
end;
```

We create an instance of TEnum (which has yet to be defined), and pass the enum's Ident as well as a true copy of the elements. Copy is a TDictionary function defined in the uCollections unit. We also pass the members (functions and values). The enum instance is stored as an interface in the memory space.

Next, create unit uEnumIntf.pas and add two new interfaces:

```
unit uEnumIntf;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, uAST, Variants, uMembers, uCollections, uToken;

type

  IEnumable = Interface
    ['{0C787B09-3EB8-C197-D539-47B40762797D}']
    procedure ExtendWith(Members: TMembers);
  end;

  IEnumInstance = Interface
    ['{1F7D0C58-2866-4A85-7FA2-E36590303912}']
    function GetMember(Ident: TIdent): Variant;
  end;

implementation

end.
```

We start with almost empty interfaces. We already know we need the procedure ExtendWith(), and getMember(), so I added those.

We continue with creating a new unit: uEnum.pas.

```
unit uEnum;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, uAST, uError, uToken, Variants, uCallable,
  uEnumIntf, uMembers;

type

  TEnumClass = class(TInterfacedObject, IEnumable)
    private
      Ident: TIdent;
      Elements: TEnumElements;
      Methods: TMethodTable;
      Values: TValueTable;
      function FindMethod(Instance: IEnumInstance; const AName: String): ICallable;
      function FindValuable(const AName: String): IValuable;
    public
      constructor Create(AIdent: TIdent; AElements: TEnumElements; Members: TMembers);
      destructor Destroy; override;
      function toString: string; override;
      procedure ExtendWith(Members: TMembers);
  end;
```

Create a class TEnum, which derives from TInterfacedObject and that implements IEnumable. It has fields for Ident, the name of the enum type and Elements, which holds the enum elements. Next to that of course the methods and values from the extensions.

```
implementation
uses uFunc, uVariantSupport;

{ TEnum }

function TEnumClass.FindMethod(Instance: IEnumInstance; const AName: String
  ): ICallable;
var
  Index: integer = -1;
begin
  Result := Nil;
  if Methods.Contains(AName, Index) then
    Result := (ICallable(Methods.At(Index)) as TFunc).Bind(Instance)
end;

function TEnumClass.FindValuable(const AName: String): IValuable;
var
  Index: LongInt = -1;
begin
  Result := Nil;
  if Values.Contains(AName, Index) then
    Result := Values.At(Index);
end;
```

Our standard find method and find value functions are available.

```
constructor TEnumClass.Create
  (AIdent: TIdent; AElements: TEnumElements; Members: TMembers);
begin
  Ident := AIdent;
  Elements := AElements;
  Methods := Members.Methods;
  Values := Members.Values;
end;

destructor TEnumClass.Destroy;
begin
  if Assigned(Elements) then Elements.Free;
  inherited Destroy;
end;
```

The constructor provides values for the Ident, the Elements and the members.

Finally, the toString method and the standard ExtendWith() method:

```
function TEnumClass.toString: string;
begin
  Result := Ident.Text;
end;

procedure TEnumClass.ExtendWith(Members: TMembers);
var
  i: Integer;
begin
  for i := 0 to Members.Methods.Count-1 do
    Methods[Members.Methods.Keys[i]] := Members.Methods.Data[i];
  for i := 0 to Members.Values.Count-1 do
    Values[Members.Values.Keys[i]] := Members.Values.Data[i];
end;

end.
```

TEnumClass takes care of defining an enum and put the enum as a class in the memory, or current space. If you want to assign an enum element to a variable, like for instance "color := Color.Red" then in fact you create an instance of the enum Color with a specific element name and value.

For this we create the Enum instance, right below the TEnumClass class:

```
  TEnumInstance = class(TInterfacedObject, IEnumInstance)
    private
      EnumClass: TEnumClass;
      Name: String;
      Value: Variant;
    public
      constructor Create(AEnumClass: TEnumClass; Ident: TIdent);
      function toString: string; override;
      function GetMember(Ident: TIdent): Variant;
  end;
```

It has three fields: the EnumClass it belongs to, the name of the element and the possible value of the element.

It's implementation is as follows:

```
constructor TEnumInstance.Create(AEnumClass: TEnumClass; Ident: TIdent);
var
  Index: LongInt=-1;
begin
  EnumClass := AEnumClass;
  Name := '';
  if EnumClass.Elements.Contains(Ident.Text, Index) then begin
    Name := Ident.Text;
    Value := EnumClass.Elements.At(Index);
  end
  else
    Raise ERuntimeError.Create(Ident.Token,
      'Undefined enum element "' + Ident.Text + '".');
end;

function TEnumInstance.toString: string;
begin
  Result := Name;
end;
```

We also provide function GetMember:

```
function TEnumInstance.GetMember(Ident: TIdent): Variant;
var
  Method: ICallable;
  Valuable: IValuable;
begin
  Method := EnumClass.FindMethod(IEnumInstance(Self), Ident.Text);
  if Method <> Nil then
    Exit(Method);

  Valuable := EnumClass.FindValuable(Ident.Text);
  if Valuable <> Nil then
    Exit(Valuable);

  Raise ERuntimeError.Create(Ident.Token,
    'Undefined enum member "' + Ident.Text + '".');
end;
```

In the interpreter, add unit uEnum to the uses clause in the implementation section:

```
implementation
uses uCallable, uFunc, uStandard, uClassIntf, uClass, uArrayIntf, uArray,
  uDictIntf, uDict, uEnumIntf, uEnum, uMath, uVariantSupport;
```

Finally, for now, change method VisitGetExpr to include a check on IEnumable.

```
function TInterpreter.VisitGetExpr(Node: TGetExpr): Variant;
...
begin
  ...
  if VarSupportsIntf(Instance,
       [IGearInstance, IArrayInstance, IDictInstance]) then
    ...
  end
  else if VarSupports(Instance, IEnumable) then begin
    Result := IEnumInstance(TEnumInstance.Create(
                IEnumable(Instance) as TEnumClass, GetExpr.Ident));
  end
  else
    ...
end;
```

Since an enum is assigned as "EnumName.ElementName" the instance in above method could also be the enum name. The enum is stored as an interface IEnumable, so we check for that. If this is the case we create an instance of TEnumInstance(TEnumClass, TIdent) and cast that to IEnumInstance, which we return as the function result.

Compile, run and test it, e.g. with the following input:

```
enum Color
  (Blue, Red, Yellow)
end
var myColor := Color.Red
print(Color)    // prints 'Color'
print(myColor)  // prints 'Red'

enum Token
  (Add='+', Min='-', Mul='*', Div='/', Rem='%')
end
var myToken := Token.Mul
print(myToken)       // prints 'Mul'
var hisToken := myToken
print(hisToken)      // prints 'Mul'
```

Assignment consistency

We also want to make sure you cannot assign a enum value of one type to another, like this:

```
hisToken := Color.Yellow
```

Variable hisToken is of type Token, and conform our wish you are not allowed to reassign it a different type. So, we have to adjust the interpreter's assignment to disallow such statements.

```
function getAssignValue(OldValue, NewValue: Variant; ID, Op: TToken): Variant;
...
begin
  ...
  if OldType <> NewType then
    Raise ERuntimeError.Create(ID,
      Format(ErrIncompatibleTypes, [OldType, NewType]));

  if VarSupports(OldValue, IEnumInstance) then begin
    if not TMath.sameEnumTypes(OldValue, NewValue) then
      Raise ERuntimeError.Create(ID,
        Format('Incompatible enum types in assignment: %s vs. %s.',
          [IEnumInstance(OldValue).EnumName, IEnumInstance(NewValue).EnumName]));
     if Op.Typ = ttAssign then
       Exit(NewValue)
     else
       Raise ERuntimeError.Create(Op, 'Illegal assignment operator, ":=" expected.');
  end;

  if not VarIsNull(OldValue) then begin
  ...
end;
```

If the types are the same we check if it is an instance of an enum. We compare the old and new enum types and if not the same we generate an error. For this we need to add a new function to the uMath unit:

```
class function TMath.sameEnumTypes(const Left, Right: Variant): Boolean;
begin
  Result := IEnumInstance(Left).EnumName = IEnumInstance(Right).EnumName;
end;
```

In the interface define the class function with the 'static' keyword.
The function uses a new property in the IEnumInstance interface. Add function EnumName:

```
  IEnumInstance = Interface
    ['{1F7D0C58-2866-4A85-7FA2-E36590303912}']
    ...
    function EnumName: String;
  end;
```

and implement it in the class TEnumInstance as a public function:

```
function TEnumInstance.EnumName: String;
begin
  Result := EnumClass.Ident.Text;
end;
```

Now, the following is not possible anymore:

```
enum Color
  (Blue, Red, Yellow)
end
enum Token
  (Add='+', Min='-', Mul='*', Div='/', Rem='%')
end
var myColor := Color.Red
var myToken := Token.Mul
myToken := myColor    // error: Incompatible enum types in assignment: Token vs. Color
```

Comparing enums

It must be possible to compare two enums with each other using the normal '=' equality operator. So if we have defined an enum and two variables are of the same enum type than it must be possible to write 'if enumvar1 = enumvar2 then…end'.

First, we enhance the IEnumInstance interface, in unit uEnumIntf:

```
  IEnumInstance = Interface
    ['{1F7D0C58-2866-4A85-7FA2-E36590303912}']
    ...
    function ElemName: String;
    function ElemValue: Variant;
  end;
```

We are going to compare element names in the comparison. It's a string comparison, which is not the fastest way, but currently it is the simplest way.

I also added function ElemValue, for completeness.

Implement the two new functions in TEnumInstance in unit uEnum.pas. They must be public.

```
function TEnumInstance.ElemName: String;
begin
  Result := Name;
end;

function TEnumInstance.ElemValue: Variant;
begin
  Result := Value;
end;
```

Next, we will make a few changes to unit uMath.pas. Note that this book contains the final version of uMath.pas, however the code chapters may not be in sync yet!

First, add uEnumIntf to the uses clause in uMath:

```
uses
  Classes, SysUtils, uToken, uError, math, Variants,
  uClassIntf, uArrayIntf, uEnumIntf;
```

Then, add the following new class function: areBothEnum().

```
class function TMath.areBothEnum(const Left, Right: Variant): Boolean;
begin
  Result := VarSupports(Left, IEnumInstance) and VarSupports(Right, IEnumInstance);
end;
```

Function sameEnumTypes checks whether both enums are of the same enum type.

And change function _EQ to include enums:

```
class function TMath._EQ(const Left, Right: Variant; Op: TToken): Variant;
begin
  ...
  if areBothEnum(Left, Right) and sameEnumTypes(Left, Right) then
    Exit(IEnumInstance(Left).ElemName = IEnumInstance(Right).ElemName);
  Raise ERuntimeError.Create(Op, Format(ErrWrongTypes, ['equal']));
end;
```

Run and test, for example with the following input:

```
enum Token
  (Add='+', Min='-', Mul='*', Div='/', Rem='%')
end

var myToken := Token.Mul
var hisToken := myToken

print(hisToken = myToken)   // prints 'True'
print(hisToken <> myToken)  // prints 'False'

hisToken := Token.Div

print(hisToken = myToken)   // prints 'False'
print(hisToken <> myToken)  // prints 'True'
```

However, adding a new enum and comparing an instance with a Token instance produces an error.

```
enum Color
  (Blue, Red, Yellow)
end

var color := Color.Blue
print(color = myToken) // error: incompatible operand types for “equal” operation
```

Access to name and value

As mentioned in the beginning of the chapter we want to have access to the enum's name and value. For this we create two standard fields 'name' and 'value' that are always available, so you can do the following:

```
enum Token
  (Add='+', Min='-', Mul='*', Div='/', Rem='%')
end

var myToken := Token.Div

print(myToken.name)       // prints 'Div'
print(myToken.value)      // prints '/'

var hisToken := myToken

print(hisToken.name)      // prints 'Div'
print(hisToken.value)     // prints '/'
```

Change TEnumInstance to include a variable Fields and methods Destroy:

```
  TEnumInstance = class(TInterfacedObject, IEnumInstance)
    private
      ...
      Fields: TFieldTable;
    public
      ...
      destructor Destroy; override;
  end;
```

In constructor Create we fill the Fields variable:

```
constructor TEnumInstance.Create(AEnum: TEnum; Ident: TIdent);
...
begin
  ...
  Fields := TFieldTable.Create;
  if Enum.Elements.Contains(Ident.Text, Index) then begin
      ...
    Fields['name'] := Name;
    Fields['value'] := Value;
  end
  else
    ...
end;
```

In destructor Destroy we we have to free the Fields variable again:

```
destructor TEnumInstance.Destroy;
begin
  Fields.Free;
  inherited Destroy;
end;
```

In function GetMember add the following line so that we can search for fields:

```
function TEnumInstance.GetMember(Ident: TIdent): Variant;
var
  ...
  Index: LongInt = -1;
begin
  if Fields.Contains(Ident.Text, Index) then
    Exit(Fields.At(Index));
  ...
end;
```

Finally, we have to change the interpreter's method VisitGetExpr, in order to retrieve the actual fields:

```
function TInterpreter.VisitGetExpr(GetExpr: TGetExpr): Variant;
...
begin
...
  if VarSupportsIntf(Instance,
       [IGearInstance, IArrayInstance, IDictInstance, IEnumInstance]) then
  begin
    ...
    else if VarSupports(Instance, IEnumInstance) then
      Result := IEnumInstance(Instance).GetMember(GetExpr.Ident);

    if VarSupports(Result, IValuable) then
    ...
end;
```

And this is it. Test it!

```
enum Token
  (Add='+', Min='-', Mul='*', Div='/', Rem='%')

  val asString := self.name + '<' + self.value + '>'
end

var myToken := Token.Div

print(myToken.name)       // prints 'Div'
print(myToken.value)      // prints '/'

var hisToken := myToken

print(hisToken.name)      // prints 'Div'
print(hisToken.value)     // prints '/'

print(hisToken.asString)  // Div </>
hisToken := Token.Rem
print(hisToken.asString)  // Rem <%>
```

A case for sets

In this paragraph we'll discuss one more addition to enums: sets. After this we're finished extending the enum type.

A group of enum elements within an enum can be referred to as a subset, like this:

> He who knows only his own side of the case knows little of that.
>
> — John Stuart Mill

```
enum Color
  (None
   case Primary: Red, Yellow, Blue
   case Secondary: Orange, Green, Brown)
end

func randomColor() =>
  match randomLimit(7)+1
    if 1 then Color.Red
    if 2 then Color.Yellow
    if 3 then Color.Blue
    if 4 then Color.Orange
    if 5 then Color.Green
    if 6 then Color.Brown
    else Color.None

var myColor := randomColor()

if myColor in Color.Primary then
  print(myColor, ' is a primary color.')
else
  if myColor in Color.Secondary then
    print(myColor, ' is a secondary color.')
  else
    print(myColor)
  end
end
```

The set name

Using the keyword 'in' (already declared and used for arrays) we can test if an enum is part of a case set. Note that the 'case' is not required! Though we use the keyword case, we call it a set. The set name is the identifier after the case keyword. So far, the enum element could hold a value, but now it can be part of a set as well. This means we've got to extend the meaning of an element.

Create a new type TEnumElement, consisting of a value and a setName. Also, the copy function creates a copy of the element. We also add a copy function to TEnumElements. We need this later on.

```
  TEnumElement = class
    Value: Variant;
    SetName: String;
    constructor Create(AValue: Variant; ASetName: String);
    function Copy: TEnumElement;
  end;

  TEnumElements = specialize TDictionaryObj<String, TEnumElement>;
  TEnumElementsHelper = class helper for TEnumElements
    function Copy: TEnumElements;
  end;
```

```
constructor TEnumElement.Create(AValue: Variant; ASetName: String);
begin
  Value := AValue;
  SetName := ASetName;
end;
```

```
function TEnumElement.Copy: TEnumElement;
begin
  Result := TEnumElement.Create(Value, SetName);
end;
```

The helper function copy creates a true copy of all enum elements:

```
function TEnumElementsHelper.Copy: TEnumElements;
var
  i: Integer;
begin
  Result := TEnumElements.Create();
  for i := 0 to self.Count-1 do
    Result.Add(self.Keys[i], TEnumElement(self.Data[i]).Copy);
end;
```

In AST node TEnumDecl the procedure AddElement changes, so that we also add the setName.

```
  TEnumDecl = class(TDecl)
      ...
    public
      ...
      procedure AddElement(const SetName, Name: String; Value: Variant);
  end;
```

And the new implementation of it:

```
procedure TEnumDecl.AddElement(const SetName, Name: String; Value: Variant);
begin
  FElements.Add(Name, TEnumElement.Create(Value, SetName));
end;
```

In the parser, we now build in the parsing of the case sets. For every identifier we have to check whether it's the case keyword or not. If we find the case keyword, we then parse the identifier following 'case'. This is the name of an enum set. The set name must be followed by a colon ':'. After the colon, the enum identifiers with their respective value are parsed, until we don't receive a comma. In that case either we get a new case set or the closing paren.

We continue parsing until we find a close paren ')'.

We have to change the inner function AddNameValue so that it accepts the name of the found set, if any, as an argument, which can be passed to method AddElement.

Let's look at the result:

```
function TParser.ParseEnumDecl: TDecl;
var
  ...
  SetName: String = '';

  procedure AddNameValue(const ASetName: String);
    ...
  begin
    ...
    EnumDecl.AddElement(ASetName, Name.Text, Value);
  end;

begin
  ...
  Expect(ttOpenParen);
  if CurrentToken.Typ <> ttCloseParen then begin
    while CurrentToken.Typ <> ttCloseParen do begin
      if CurrentToken.Typ = ttCase then begin
        Next;
        SetName := ParseIdent.Text;
        Expect(ttColon);
      end;
      AddNameValue(SetName);   // at least expect 1 enumeration
      while CurrentToken.Typ = ttComma do begin
        Next;  // skip ,
        AddNameValue(SetName);
      end;
      SetName := '';
    end;
  end
  else Error(CurrentToken, ErrAtLeastOneEnum);
  Expect(ttCloseParen);
  ...
end;
```

A small change in the printer is required, if we want to continue printing the enum values:

```
procedure TPrinter.VisitEnumDecl(EnumDecl: TEnumDecl);
...
begin
...
  for i := 0 to EnumDecl.Elements.Count-1 do begin
    Writeln(Indent, 'Name: ', EnumDecl.Elements.Keys[i], ', Value: ',
            EnumDecl.Elements.Data[i].Value);
  end;
...
end;
```

The following helper function simply provides the setname of an enum. This is needed later on.

```
  IEnumInstance = Interface
    ...
    function ElemSetName: String;
  end;
```

Ans implement it in TEnumInstance:

```
function TEnumInstance.ElemSetName: String;
begin
  Result := EnumClass.Elements[Name].SetName;
end;
```

Next, add field 'SetName' to the enum instance:

```
  TEnumInstance = class(TInterfacedObject, IEnumInstance)
    private
      ...
      SetName: String;
      ...
  end;
```

Then, we change TEnumInstance.Create, so that we create a field name for set. Also, adjust the retrieval of the value:

```
constructor TEnumInstance.Create(AEnum: TEnum; Ident: TIdent);
  ...
begin
  ...
  Name := '';
  SetName := '';
  Fields := TFieldTable.Create;
  if EnumClass.Elements.Contains(Ident.Text, Index) then begin
    Name := Ident.Text;
    Value := EnumClass.Elements.At(Index).Value;
    SetName := EnumClass.Elements.At(Index).SetName;
    Fields['name'] := Name;
    Fields['value'] := Value;
    Fields['set'] := SetName;
  end
  ...
end;
```

For example in (use randomColor() from previous example):

```
enum Color
  (None
   case Primary: Red, Yellow, Blue
   case Secondary: Orange, Green, Brown
  )
  func from(.set) => set = self.set
end

var myColor := randomColor()
print(myColor)
print(myColor.set)
print(myColor.from(set: 'Primary'))
```

The 'in' keyword

Though the 'in' keyword was defined, so far it was used only to check if a value belonged to an array. So, let's add the check on set membership in enums to that.

```
if myColor in Color.Primary then ... end
```

'myColor in Color.Primary' is a boolean expression, which either returns True or False.

In unit uMath change the _In function to include enums:

```
class function TMath._In(const Left, Right: Variant; Op: TToken): Variant;
begin
  ...
  if VarSupports(Left, IEnumInstance) then
    Exit(IEnumInstance(Left).ElemSetName = Right);
  Raise ERuntimeError.Create(Op, Format(ErrMustBeBoolean, ['in']));
end;
```

Since we have the set name available in an enum instance the easiest way is to compare it with the right hand side of the 'in' expression.

If we wish to test whether an enum element is part of a case set, we use 'myColor in Color.Primary', as if Primary is a field of Color. Since it is not an actual field, we need another way to retrieve the right information on the set. In VisitGetExpr we introduce method 'isCase()':

```
function TInterpreter.VisitGetExpr(GetExpr: TGetExpr): Variant;
    ...
begin
    ...
  else if VarSupports(Instance, IEnumable) then begin
    if IEnumable(Instance).isCase(GetExpr.Ident.Text) then
      Result := GetExpr.Ident.Text
    else
      Result := IEnumInstance(TEnumInstance.Create(
        IEnumable(Instance) as TEnumClass, GetExpr.Ident));
  end
    ...
end;
```

If it is a case set, we just return the name of the case set. The rest of VisitGetExpr stays the same. For this, we'll add a new function to interface IEnumable:

```
  IEnumable = Interface
    ...
    function isCase(Name: String): Boolean;
  end;
```

And implement that in class TEnum:

```
function TEnumClass.isCase(Name: String): Boolean;
begin
  Result := CaseTable.Contains(Name);
end;
```

We don't have a case table yet. There are more ways to create such a table, and in first instance I created a solution that works when creating an enum instance at runtime, but I figured it puts again pressure on performance during running, so I decided to make it part of the parsing process.

In the AST add just above TEnumDecl a new type TCaseTable, which is an array of strings. Then add it as a property to TEnumDecl:

```
  TCaseTable = specialize TArray<String>;
```

```
  TEnumDecl = class(TDecl)
    private
    ...
      FCaseTable: TCaseTable;
    public
      ...
      property CaseTable: TCaseTable read FCaseTable;
      constructor Create(AIdent: TIdent; ADeclarations: TDeclList;
        ACaseTable: TCaseTable; AToken: TToken);
      ...
  end;
```

Add it to the constructor and destructor as well:

```
constructor TEnumDecl.Create(AIdent: TIdent; ADeclarations: TDeclList;
  ACaseTable: TCaseTable; AToken: TToken);
begin
  ...
  FCaseTable := TCaseTable.Create;
end;

destructor TEnumDecl.Destroy;
begin
  ...
  FCaseTable.Free;
  inherited Destroy;
end;
```

Next, add the parsing functionality to ParseEnumDecl:

```
function TParser.ParseEnumDecl: TDecl;
  ...

begin
  ...
    while CurrentToken.Typ <> ttCloseParen do begin
      if CurrentToken.Typ = ttCase then begin
        Next; // skip case
        SetName := ParseIdent.Text;
        if EnumDecl.CaseTable.Contains(SetName) then
          Error(CurrentToken, Format(ErrDuplicateSetName, [SetName]))
        else EnumDecl.CaseTable.Add(SetName);
        Expect(ttColon);
      end;
    ...
  end;
  ...
end;
```

When we encounter a case keyword, we add the set identifier as a string to the case table.
If we encounter a duplicate we report an error:

```
  ErrDuplicateSetName = 'Duplicate enum set name "%s".';
```

Add it to the list of errors in the parser.
In the interpreter, make sure that the case table is sent to the TEnum constructor:

```
procedure TInterpreter.VisitEnumDecl(EnumDecl: TEnumDecl);
  ...
begin
  ...
  Enum := TEnumClass.Create(EnumDecl.Ident, EnumDecl.Elements.Copy,
    EnumDecl.CaseTable, Members);
  ...
end;
```

Finally, add a private field CaseTable to TEnum and amend the constructor.

```
  TEnum = class(TInterfacedObject, IEnumable)
    private
      ...
      CaseTable: TCaseTable;
      ...
    public
      constructor Create(AIdent: TIdent; AElements: TEnumElements;
        ACaseTable: TCaseTable; Members: TMembers);
      ...
      function isCase(Name: String): Boolean;
  end;


constructor TEnumClass.Create(AIdent: TIdent; AElements: TEnumElements;
  ACaseTable: TCaseTable; Members: TMembers);
begin
  ...
  CaseTable := ACaseTable;
end;
```

And we're ready to run! Try it out.

```
enum Color
  (None
   case Primary: Red, Yellow, Blue
   case Secondary: Orange, Green, Brown
  )
  func from(.set) => set = self.set
  val primary := self in Color.Primary
end

func randomColor() =>
  match randomLimit(7)+1
    if 1 then Color.Red
    if 2 then Color.Yellow
    if 3 then Color.Blue
    if 4 then Color.Orange
    if 5 then Color.Green
    if 6 then Color.Brown
    else Color.None

var myColor := randomColor()
print(myColor)
print(myColor.set)
print(myColor.from(set: 'Primary'))

if myColor in Color.Primary then
  print('Yes, a hit!')
end

if myColor.primary then
  print('wow again a hit.')
end
```

Other example:

```
enum Dice
  (case Even: Two, Four, Six
   case Odd: One, Three, Five
   None)

  func toString() => self.name + '<' + self.set + '>'

  func roll()
    var rnd := randomLimit(6)+1
    var result := match rnd
      if 1 then Dice.One
      if 2 then Dice.Two
      if 3 then Dice.Three
      if 4 then Dice.Four
      if 5 then Dice.Five
      if 6 then Dice.Six
      else Dice.None
    return result.toString()
  end
end

var dice := Dice.None

func diceRoll(.n, f)
  for var i := 1 where i<=n, i+=1 do
    print(f())
  end
end

var f := diceRoll(n:10, () => dice.roll())
```

Prints something like this (depends on random result):

```
Six<Even>
Six<Even>
Five<Odd>
Three<Odd>
Two<Even>
Two<Even>
Six<Even>
Five<Odd>
Two<Even>
Four<Even>
```

Iterating over enums

In some cases you want to iterate over a set, or all defined enums. Consider the following example:

> Iteration, like friction, is likely to generate heat instead of progress.
> — George Eliot

```
use system

enum Color
  (None
   case Primary: Red, Yellow, Blue
   case Secondary: Orange, Green, Brown)
end

for each color in Color.Elements where color in Color.Primary do
  print(color)
end
```

This prints the colors Red, Yellow and Blue.

Against my principle, this is where we'll use a bit hard coding. You'll see…

In uEnumIntf add function ElementList to IEnumable:

```
  IEnumable = Interface
    ...
    function ElementList: IArrayInstance;
  end;
```

Then implement this function in TEnumClass as a public function:

```
function TEnumClass.ElementList: IArrayInstance;
var
  ArrayType: IArrayable;
  i: Integer;
  EnumIdent: TIdent;
begin
  // create a array/list of all items
  ArrayType := IArrayable(Language.Interpreter.Globals['Array']);
  Result := IArrayInstance(TArrayInstance.Create(ArrayType as TArrayClass));
  for i := 0 to Elements.Count-1 do begin
    EnumIdent := TIdent.Create(TToken.Create(
      ttIdentifier, Elements.Keys[i], Null, 0, 0));
    Result.Elements.Add(IEnumInstance(TEnumInstance.Create(Self, EnumIdent)));
  end;
end;
```

What do we do here? We create a new array instance of type Array. By doing so, we can use the functionality defined in library arrays.gear, or system.gear. This contains, amongst others the iterators, which we need for the 'for-each' statement. Inside the array we store all the enum instances of the defined enum. We create a TIdent with the name of the enum element, and from this we create an instance that we add to the array.

Next, we have to amend VisitGetExpr in the interpreter.

```
function TInterpreter.VisitGetExpr(GetExpr: TGetExpr): Variant;
...
begin
  ...
  else if VarSupports(Instance, IEnumable) then begin
    if IEnumable(Instance).isCase(GetExpr.Ident.Text) then
      Result := GetExpr.Ident.Text
    else if GetExpr.Ident.Text = 'Elements' then
      Result := IEnumable(Instance).ElementList
    else
      Result := IEnumInstance(TEnumInstance.Create(
        IEnumable(Instance) as TEnumClass, GetExpr.Ident));
  end
  ...
end;
```

Chapter 16 – Enhancements

This chapter discusses some items I forgot or that newly popped up.

Array of characters from a string

Sometimes you want to easily create an array of characters from a string input, like in this example:

```
var input := 'while value < 10 do'
```

Suppose you want create a Lexer in Gear than it would help if we could create an array of char from this input. Luckily this is very easy. In uArray.pas we only need to change the call function in order to allow a construct like this:

```
var chars := Array(input)
```

In function TArrayClass.call add the following line:

```
function TArrayClass.Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList
  ): Variant;
...
begin
  ...
  else if VarIsStr(ArgList[0].Value) then
    for i := 1 to Length(String(ArgList[0].Value)) do
      Instance.Elements.Add(String(ArgList[0].Value)[i])
  else
    Raise ERuntimeError.Create(Token,
  ...
end;
```

```
var a := Array('Hello world!')
print(a)
```

results in:

```
[H, e, l, l, o,  , w, o, r, l, d, !]
```

Or you can do things like looping over an array of char:

```
for each char in Array('Hello world, my answer is 42') do
  print(char)
end
```

Reading from standard input

So far we can print to the standard output, but we cannot read from it. The print statement was created as a statement routine, but we will create a reading function in the standard functions.

Add a class TReadLn to uStandard.pas and implement it using the standard Pascal routine readln.

```
  TReadLn = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

function TReadLn.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Input: String;
begin
  Readln(input);
  Result := input;
end;
```

Pascal's readln(input) routine reads the input into the variable input, which is a socalled var parameter. However, in Gear we will treat it as a function that provide a return result. Examples of usage:

```
var a := readln()
```

Note that this works on the commandline only.

Next, add the function to the globals in the interpreter:

```
  FGlobals.Store('readln', ICallable(TReadLn.Create));
```

And to the symbol table in the Resolver:

```
  Enter(TSymbol.Create('readln', Enabled, False));
```

Conversions

Often you need conversions from string to number or vice versa. Also you want conversions to succeed, so should take care of any errors that might pop up.
We'll introduce two new standard functions:

- toStr(value), which turns any variable into a string value.
- toNum(value), which turns a string value into a number if this potential value is a valid number. Otherwise the result will be Null.

Let's add them to uStandard:

```
  TToNum =class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

  TToStr =class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;
```

and their implementations:

```
function TToNum.Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList
  ): Variant;
var
  Success: Boolean = False;
  Value: Double;
begin
  try
    Success := TryStrToFloat(ArgList[0].Value, Value);
    if Success then
      Result := Value
    else
      Result := Null;
  except
    Result := Null;
  end;
end;
```

```
function TToStr.Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList
  ): Variant;
begin
  try
    Result := VarToStr(ArgList[0].Value);
  except
    Result := Null;
  end;
end;
```

Store them in the Interpreter:

```
  FGlobals.Store('toNum', ICallable(TToNum.Create));
  FGlobals.Store('toStr', ICallable(TToStr.Create));
```

And the Resolver:

```
  Enter(TSymbol.Create('toNum', Enabled, False));
  Enter(TSymbol.Create('toStr', Enabled, False));
```

Examples:

```
var a := '3.1415'
print(toNum(a))

var b := 3.1415
print(toStr(b))

print(toNum(a) = b)

print('a' = 'a')
print(a = toStr(b))

print(toStr(False))
print(toStr(pi()))
print(toStr(b=pi()))
print(toStr(111))
print(toStr("a))

print(toNum(True))
```

Resulting in:

```
3.1415
3.1415
True
True
True
False
3.14159265358979
False
111
a
Null
```

Since we cannot convert the value True to a number it results in Null being printed.

Continue loop

We have created a break-statement, which is handy if you want to leave a loop early, but sometimes you want to stop the current iteration and continue with the following, like in this example, a number guessing game:

```
var number := randomLimit(100) + 1

while True do
  print('Guess a number: ', terminator: '')

  ensure var guess := toNum(readln()) where guess <> Null else
    print('That`s not a number!')
    continue
  end

  if guess < number then
    print('Too low.')
  else
    if guess = number then
      print('You win!')
      break
    else
      print('Too high.')
    end
  end

end
```

The program loops until the number is guessed.

The ensure statement checks whether the input is a number. If not guess is Null and we don't want to execute the second part where we are comparing numbers. That would end up in a crash!

So, using continue stops the iteration and continues the next one.

If finally the number is guessed we use break to leave the loop and thus the program.

The token ttContinue doesn't exist yet, so add it to TTokenType in unit uToken, and create a keyword for it as well:

```
  Keywords['continue'] := ttContinue;
```

We also need an AST node for TContinueStmt:

```
   TContinueStmt = class(TStmt)
     //nothing in here
   end;
```

Next, create a parser function, and add ttContinue to the Statement start set:

```
const
  StmtStartSet: TTokenTypSet = [ttIf, ttWhile, ttRepeat, ttFor, ttReturn,
    ttSwitch, ttEnsure, ttPrint, ttInherited, ttSelf, ttUse, ttBreak,
    ttContinue, ttIdentifier];
```

```
function TParser.ParseStmt: TStmt;
begin
  case CurrentToken.Typ of
    ...
    ttContinue: Result := ParseContinueStmt;
    else
      Result := ParseAssignStmt;
  end;
end;
```

```
function TParser.ParseContinueStmt: TStmt;
begin
  Result := TContinueStmt.Create(CurrentToken);
  Next; // skip continue
end;
```

The printer and resolver:

```
procedure TPrinter.VisitContinueStmt(ContinueStmt: TContinueStmt);
begin
  IncIndent;
  VisitNode(ContinueStmt);
  DecIndent;
end;
```

```
procedure TResolver.VisitContinueStmt(ContinueStmt: TContinueStmt);
begin
  // nothing to do
end;
```

All very simple and straightforward. The challenge is to put it in our loop statements. However, first create a error exception type in uError.pas:

```
  EContinueException = class(Exception);
```

Finally, we are ready to implement it in the inerpreter. Start with the continue statement itself:

```
procedure TInterpreter.VisitContinueStmt(ContinueStmt: TContinueStmt);
begin
  raise EContinueException.Create('');
end;
```

It just raises an EContinueException, nothing else. But now the magic!

```
procedure TInterpreter.VisitWhileStmt(WhileStmt: TWhileStmt);
...
begin
  ...
    try
      Condition := Visit(WhileStmt.Condition);
      if VarIsBool(Condition) then begin
        while Condition do begin
          try
            Visit(WhileStmt.Block);
          except
            on EContinueException do;
          end;
          Condition := Visit(WhileStmt.Condition);
        end;
      end
  ...
end;
```

Inside the while loop we create a new try…except statement, which catches the EContinueException error, and does…nothing. But see that the statement where the condition is checked is right below that, so in fact we just continue the process. This is all that is needed!

We do the same for the repeat-statement:

```
procedure TInterpreter.VisitRepeatStmt(RepeatStmt: TRepeatStmt);
...
begin
  try
    ...
    if VarIsBool(Condition) then begin
      repeat
        try
          Visit(RepeatStmt.Block);
        except
          on EContinueException do;
        end;
        Condition := Visit(RepeatStmt.Condition);
      until Condition;
  ...
end;
```

If elseif where there

In the last example there was some nesting of if-else statements, which looks a bit difficult to read and may be the source of errors, for example by forgetting an 'end' keyword:

```
let number := 100
var guess := 1000

if guess < number then
  print('Too low.')
else
  if guess = number then
    print('You win!')
  else
    print('Too high.')
  end
end
```

Preferrably, you would want something like this:

```
if guess < number then
  print('Too low.')
elseif guess = number then
  print('You win!')
else
  print('Too high.')
end
```

Let's first solve this for the case of only 1 elseif first.
Start by adding the token type and keyword to uToken.pas:

```
  Keywords['elseif'] := ttElseif;
```

In the AST change the IfStmt node:

```
  TIfStmt = class(TStmt)
    private
      ...
      FElseIfExpr: TExpr;
      FElseIfPart: TBlock;
    public
      ...
      property ElseIfExpr: TExpr read FElseIfExpr;
      property ElseIfPart: TBlock read FElseIfPart;
      constructor Create(AVarDecl: TVarDecl; ACondition, AElseIfExpr: TExpr;
        AThenPart, AElseIfPart, AElsePart: TBlock; AToken: TToken);
      ...
  end;
```

Change the Printer:

```
procedure TPrinter.VisitIfStmt(IfStmt: TIfStmt);
begin
  ...
  Visit(IfStmt.ThenPart);
  if Assigned(IfStmt.ElseIfPart) then begin
    WriteLn(Indent, 'IfElsePart:');
    Visit(IfStmt.ElseIfExpr);
    Visit(IfStmt.ElseIfPart);
  end;
  ...
end;
```

And the Resolver:

```
procedure TResolver.VisitIfStmt(IfStmt: TIfStmt);
begin
  ...
  Visit(IfStmt.ThenPart);
  if Assigned(IfStmt.ElseIfPart) then begin
    Visit(IfStmt.ElseIfExpr);
    Visit(IfStmt.ElseIfPart);
  end;
  ...
end;
```

We change the Parser to also parse the elseif part if it is there. If not there both the expression and the block are Nil. Don't forget to initialize both with Nil!

```
function TParser.ParseIfStmt: TStmt;
var
  ...
  ElseIfExpr: TExpr = Nil;
  ElseIfPart: TBlock = Nil;
begin
  ...
  if CurrentToken.Typ = ttElseif then begin
    Next;  // skip elseif
    ElseIfExpr := ParseExpr;
    Expect(ttThen);
    ElseIfPart := ParseBlock;
  end;
  if CurrentToken.Typ = ttElse then begin
  ...
  Result := TIfStmt.Create(VarDecl, Condition, ElseIfExpr,
    ThenPart, ElseIfPart, ElsePart, Token);
end;
```

Finally, return the result with the new elements in the in the IfStmt constructor.

In ParseForEachStmt modify the creation of the IfStmt to:

```
  if Assigned(WhereExpr) then begin
    IfStmt := TIfStmt.Create(Nil, WhereExpr, Nil,
      TBlock.Create(TNodeList.Create(), Nil), Nil, Nil, Nil);
```

The interpreter gets most of the changes. An inner function isBooleanAndTrue is introduced, which perfoms the following tasks: It visits the condition expression, and then checks if it results in a Boolean value. If not and error is generated. If it is a Boolean value we return this as the result of the evaluation. The VarDecl part doesn't change.

```
procedure TInterpreter.VisitIfStmt(IfStmt: TIfStmt);
var
  SavedSpace: TMemorySpace;

  function isBooleanAndTrue(Condition: TExpr): Boolean;
  var Value: Variant;
  begin
    Value := Visit(Condition);
    if VarIsBool(Value) then
      Result := Boolean(Value)
    else
      Raise ERuntimeError.Create(IfStmt.Token, ErrConditionMustBeBool);
  end;
```

```
begin
  try
    if Assigned(IfStmt.VarDecl) then begin
      SavedSpace := CurrentSpace;
      CurrentSpace := TMemorySpace.Create(SavedSpace);
      Visit(IfStmt.VarDecl);
    end;

    if isBooleanAndTrue(IfStmt.Condition) then
      Visit(IfStmt.ThenPart)
    else if Assigned(IfStmt.ElseIfPart) then begin
      if isBooleanAndTrue(IfStmt.ElseIfExpr) then
        Visit(IfStmt.ElseIfPart)
      else if Assigned(IfStmt.ElsePart) then
        Visit(IfStmt.ElsePart);
    end
    else if Assigned(IfStmt.ElsePart) then
      Visit(IfStmt.ElsePart);

  finally
    if Assigned(IfStmt.VarDecl) then begin
      CurrentSpace.Free;
      CurrentSpace := SavedSpace;
    end;
  end;
end;
```

The IfStmt's first condition is evaluated and if True it's block is executed. If not True we test for existence of the ElseIf part. If so, we evaluate the expression and if True we execute this block. If not True, we check if an Else block is there and execute if so. Note that we have to do this also for the outer if statement in case there is no ElseIf part.

Test for example with the following program:

```
let max := 100
var number := randomLimit(max) + 1

while True do
  print('Guess a number (1 to ', max, '): ', terminator: '')

  ensure var guess := toNum(readln()) where guess <> Null else
    print('That`s not a number!')
    continue
  end

  ensure (guess > 0) & (guess <= max) else
    print('That number is not in range 1 to ', max)
    continue
  end

  if guess < number then
    print('Too low.')
  elseif guess = number then
    print('You win!')
    break
  else
    print('Too high.')
  end

end
```

Note that only 1 elseif is allowed!

Let's now try to create a solution for multiple elseif branches. This means multiple elseif expressions and multiple blocks as well. In the AST node we replace the single occurrances with lists of multiple.

```
  TBlockList = specialize TArrayObj<TBlock>;

  TIfStmt = class(TStmt)
    private
      ...
      FElseIfs: TExprList;
      FElseIfParts: TBlockList;
    public
      ...
      property ElseIfs: TExprList read FElseIfs;
      property ElseIfParts: TBlockList read FElseIfParts;
      constructor Create(AVarDecl: TVarDecl; ACondition: TExpr;
        AElseIfs: TExprList; AThenPart, AElsePart: TBlock;
        AElseIfParts: TBlockList; AToken: TToken);
      ...
  end;
```

```
constructor TIfStmt.Create(AVarDecl: TVarDecl; ACondition: TExpr;
  AElseIfs: TExprList; AThenPart, AElsePart: TBlock;
  AElseIfParts: TBlockList; AToken: TToken);
begin
  ...
  FElseIfs := AElseIfs;
  FElseIfParts := AElseIfParts;
end;

destructor TIfStmt.Destroy;
begin
  ...
  if Assigned(FElseIfs) then FElseIfs.Free;
  if Assigned(FElseIfParts) then FElseIfParts.Free;
  inherited Destroy;
end;
```

Printing the elseif parts now requires a loop, so add variable i: Integer, and loop over all the parts in the list of elseifs.

```
procedure TPrinter.VisitIfStmt(IfStmt: TIfStmt);
var
  i: Integer;
begin
  ...
  if Assigned(IfStmt.ElseIfs) then begin
    WriteLn(Indent, 'IfElseParts:');
    for i := 0 to IfStmt.ElseIfs.Count-1 do begin;
      Visit(IfStmt.ElseIfs[i]);
      Visit(IfStmt.ElseIfParts[i]);
    end;
  end;
  if Assigned(IfStmt.ElsePart) then begin
  ...
end;
```

The same should happen in the Resolver.

```
procedure TResolver.VisitIfStmt(IfStmt: TIfStmt);
var
  i: Integer;
begin
  ...
  if Assigned(IfStmt.ElseIfs) then begin
    for i := 0 to IfStmt.ElseIfs.Count-1 do begin;
      Visit(IfStmt.ElseIfs[i]);
      Visit(IfStmt.ElseIfParts[i]);
    end;
  end;
  if IfStmt.ElsePart <> Nil then
  ...
end;
```

Now the Parser. Replace variable ElseIfExpr with ConditionList of type TExprList, and replace ElseifPart with ElseIfParts of type TBlockList.
If we determine a ttElseIf, we know we have 1 or more elseif parts to parse. So, here's where we create the lists (otherwise they remain Nil).

```
function TParser.ParseIfStmt: TStmt;
var
  ...
  ConditionList: TExprList = Nil;
  ElseIfParts: TBlockList = Nil;
begin
  ...
  if CurrentToken.Typ = ttElseif then begin
    ConditionList := TExprList.Create();
    ElseIfParts := TBlockList.Create();
    repeat
      Next;  // skip elseif
      ConditionList.Add(ParseExpr);
      Expect(ttThen);
      ElseIfParts.Add(ParseBlock);
    until CurrentToken.Typ <> ttElseIf;
  end;
  if CurrentToken.Typ = ttElse then begin
  ...
  Result := TIfStmt.Create(VarDecl, Condition, ConditionList,
    ThenPart, ElsePart, ElseIfParts, Token);
end;
```

The nice thing of the Pascal repeat statement is that it repeats at least one block. In this case the first elseif token is skipped, followed by parsing the expression and adding it to the list of ElseIf expressions (conditions), followed by a 'then' keyword, and finally parsing the block and adding it to the ElseIfParts (list of blocks). We keep repeating this until we don't find a ttElseIf.

The order of parameters was a bit changed in the constructor, and some are now lists, so the new constructor line needs to incorporate that.

Next, we enhance the Interpreter. Since we have multiple elseif branches we have to walk through them one by one, either till we find a condition that is True, or until we hit the 'else' keyword, or a completely new statement.

```
procedure TInterpreter.VisitIfStmt(IfStmt: TIfStmt);
var
  ...
  i: Integer;
  ElseIfExecuted: Boolean = False;

  function isBooleanAndTrue(Condition: TExpr): Boolean;
  ...
  end;

begin
  try
    ...
    if isBooleanAndTrue(IfStmt.Condition) then
      Visit(IfStmt.ThenPart)
    else if Assigned(IfStmt.ElseIfs) then begin
      for i := 0 to IfStmt.ElseIfs.Count-1 do begin
        if isBooleanAndTrue(IfStmt.ElseIfs[i]) then begin
          Visit(IfStmt.ElseIfParts[i]);
          ElseIfExecuted := True;
          Break;
        end;
      end;
      if not ElseIfExecuted then
        if Assigned(IfStmt.ElsePart) then
          Visit(IfStmt.ElsePart);
    end
    else if Assigned(IfStmt.ElsePart) then
      Visit(IfStmt.ElsePart);

  finally
    ...
end;
```

Add the two new variables i as the loop counter and ElseIfExecuted, which determines if an elseif was actually executed. Because if an elseif was executed, we should not also execute the 'else' part. If none of the elseifs were executed because the condition was not met, we are allowed to check for and execute the else-part. Now with this in place we can do things like this:

```
class Root
  var name := 'Root'
end

class First (Root)
end

class Second (Root)
end

class Third (Second)
end

var sec := Second()
if sec is Root then
  print('Root')
elseif sec is First then
  print('First')
elseif sec is Second then
  print('Second')
elseif sec is Third then
  print('Third')
else
  print('None of the above')
end
```

Which prints: 'Second'.

Switch and instance of a class

In the previous chapter you saw an if-then-elseif statement where multiple class instances could be compared with a value. It would be great if our switch statement could be used for this as well, so that you could do this:

```
class Root
  var name := 'Root'
end

class First (Root)
end

class Second (Root)
end

class Third (Second)
end
class Fourth
end

var sec := Second()

switch sec
  case is Root: print('Root')
  case is First: print('First')
  case is Second: print('Second')
  case is Third: print('Third')
  else
    print('None of the above')
end
```

And it would print 'Second'.

Change the AST node for TSwitchStmt, so that it includes an indication of the 'is' case. I used IsObj for this, a boolean variable.

```
   TSwitchStmt = class(TStmt)
     private
       type
         TCaseLimb = class
           ...
           IsObj: Boolean;
           constructor Create(AValues: TExprList; AIsObj: Boolean; ABlock: TBlock);
         end;
         ...
     public
       ...
       procedure AddLimb(AValues: TExprList; AIsObj: Boolean; ABlock: TBlock);
   end;
```

Add it to the constructor of TCaseLimb:

```
constructor TSwitchStmt.TCaseLimb.Create
  (AValues: TExprList; AIsObj: Boolean; ABlock: TBlock);
begin
  ...
  IsObj := AIsObj;
end;
```

And change AddLimb accordingly:

```
procedure TSwitchStmt.AddLimb
  (AValues: TExprList; AIsObj: Boolean; ABlock: TBlock);
begin
  FCaseLimbs.Add(TCaseLimb.Create(AValues, AIsObj, ABlock));
end;
```

Then, in the parser, add ttCase to the constant BlockEndSet. This should already be there since chapter 4 by the way…

```
const
  BlockEndSet: TTokenTypSet = [ttElse, ttElseif, ttUntil, ttCase, ttEnd, ttEOF];
```

In the Parser we'll check for the 'is' keyword, and if so switch the variable IsObj to True. We do the same in the respective loop, while we find ttCase. Note, we have to set IsObj first to False again.

```
function TParser.ParseSwitchStmt: TStmt;
var
  ...
  IsObj: Boolean = False;
begin
  ...
  Expect(ttCase); // one 'case' is mandatory
  if CurrentToken.Typ = ttIs then begin
    Next; // skip is
    IsObj := True;
  end;
  Values := ParseExprList;
  Expect(ttColon);
  SwitchStmt.AddLimb(Values, IsObj, ParseBlock);
  while CurrentToken.Typ = ttCase do begin
    Next;  // skip case
    IsObj := False;
    if CurrentToken.Typ = ttIs then begin
      Next; // skip is
      IsObj := True;
    end;
    Values := ParseExprList;
    Expect(ttColon);
    SwitchStmt.AddLimb(Values, IsObj, ParseBlock);
  end;
  ...
end;
```

Finally, a small change in the Interpreter:

```
procedure TInterpreter.VisitSwitchStmt(SwitchStmt: TSwitchStmt);
var
  ...
  IsObjValue: Boolean;
begin
  ...
      CaseValue := Visit(SwitchStmt.CaseLimbs[i].Values[j]);
      IsObjValue := SwitchStmt.CaseLimbs[i].IsObj;
      if IsObjValue then begin
        if TMath._Is(SwitchValue, CaseValue,
          SwitchStmt.CaseLimbs[i].Values[j].Token) then
        begin
          Visit(SwitchStmt.CaseLimbs[i].Block);
          Exit;
        end;
      end
      else if TMath._EQ(SwitchValue, CaseValue,
        SwitchStmt.CaseLimbs[i].Values[j].Token) then
      begin
        Visit(SwitchStmt.CaseLimbs[i].Block);
        Exit;
      end;
    end;
  end;
  ...
end;
```

We get the value for the IsObj variable. If it is True, we have to deal with an 'case is' situation, and we use the TMath's _Is function to check if the switch value is an instance of the case value. Otherwise we apply the normal TMath._EQ function to check the respective values.

More on ranges

So far, we have created a range type in Gear library 'ranges.gear'. It comes with an iterator. Here's that code for a reminder.

```
class RangeIterator
  var index := 0
  var range := Null

  init(range)
    self.range := range
    self.index := range.from
  end

  val hasNext := self.index <= self.range.to

  val next
    var result := self.index
    self.index +=1
    return result
  end
end

class Range
  var from := Null
  var to := Null

  init(from, to)
    self.from := from
    self.to := to
  end

  val iterator := RangeIterator(self)
end
```

With this it is possible to do for-each loops, like this:

```
for each i in Range(1,10) do
  print(i)
end
```

However, wouldn't it be nice, to use instead of Range(a,b) something more appropriate like:

```
for each i in 1..10 do
  /…./
end
for each n in -5..5 do
  /…./
end
```

The good thing is that this can be solved easily right in the Parser. We don't have to create a new AST node, nor do we need to change something in the Interpreter. The only thing that is required is to use either 'ranges.gear', or 'system.gear'. Here goes!

First, let's see how the expression n..m fits in the EBNF. This is what we have so far:

```
  ShiftExpr = UnaryExpr { ['<<' | '>>' | '^'] UnaryEpr } .
  UnaryExpr  = [ '+' | '-' | '!' | '?' ] UnaryExpr | CallExpr  .
  CallExpr = Factor { '(' [Arguments] ')' | '.' Ident | '[' Expr ']' } .
```

As you can see from above examples, a range can start with a negative sign, so with a unary expression in fact. Ans also the upper bound of a range is a unary expression.

In the RangeExpr we have to see if it contains a token type '..', or ttDotDot, which is already defined in uToken. So, we can create the EBNF as follows:

```
  ShiftExpr = RangeExpr { ['<<' | '>>' | '^'] UnaryEpr } .
  RangeExpr = UnaryExpr [ '..' UnaryExpr ] .
  UnaryExpr  = [ '+' | '-' | '!' | '?' ] UnaryExpr | CallExpr .
  CallExpr = Factor { '(' [Arguments] ')' | '.' Ident | '[' Expr ']' } .
```

Maybe, there are even smarter ways of putting the range expression in the parser, but this seems to work.

This changes function ParseShiftExpr to the following:

```
function TParser.ParseShiftExpr: TExpr;
var
  ShiftOp: TToken;
begin
  Result := ParseRangeExpr;
  while isShiftOp do begin
    ShiftOp := CurrentToken;
    Next;
    Result := TBinaryExpr.Create(Result, ShiftOp, ParseUnaryExpr);
  end;
end;
```

Function ParseUnaryExpr doesn't change. We have to create a new function ParseRangeExpr. Add this initial version to the TParser interface as well of course.

```
function TParser.ParseRangeExpr: TExpr;
var
  UpperBound: TExpr;
  Range: TCallExpr;
begin
  Result := ParseUnaryExpr;
  if CurrentToken.Typ = ttDotDot then begin
    Next; // skip ..
    UpperBound := ParseUnaryExpr;
    // build call to Range(Lower, Upper)
    Range := TCallExpr.Create(TVariable.Create(MakeID('Range')), Nil);
    Range.AddArgument(Result, Nil);
    Range.AddArgument(UpperBound, Nil);
    Result := Range;
  end;
end;
```

First, we parse a UnaryExpr. Then we check if the next token is a dotdot '..' and if so parse the upper bound of the range expression. We then create a new TCallExpr. Actually, Range is a class, but remember that a class construction is just like a function call, which is why we can use a TCallExpr here. We build it up by creating a TIdent with name 'Range'. Next, we add the two arguments Result (which is the lower bound) and upper bound and return this as the result.

If we didn't see a dotdot, it means there's no range expression and we parse the regular UnaryExpr.

Test and run it. In the following Gear code the two loops are equal, but the second one looks much nicer:

```
use system

for each i in Range(1,10) do
  print(i)
end

for each i in 1..10 do
  print(i)
end
```

It can also handle negative values, such as -5..-1. However, since a UnaryExpr can also have the '!' and '?' signs, this will give problems is used by the programmer. So, we must detect if the right token types are used in a range expression. Let's add that check.

```
function TParser.ParseRangeExpr: TExpr;
var
  ...

  procedure CheckUnaryOp(Expr: TExpr);
  begin
    if Expr is TUnaryExpr then
      if (Expr as TUnaryExpr).Op.Typ in [ttNot, ttQuestion] then
        Error((Expr as TUnaryExpr).Op,
          Format(ErrNotAllowedInRange, [(Expr as TUnaryExpr).Op.Typ.toString]));
  end;

begin
  ...
  if CurrentToken.Typ = ttDotDot then begin
    CheckUnaryOp(Result);
    ...
    UpperBound := ParseUnaryExpr;
    CheckUnaryOp(UpperBound);
  ...
end;
```

If the unary expression as part of the range expression contains a '!' or a '?' an error is generated.

Sometimes you need to loop over the contents of an array, where you need the index to be part of the loop, like in this example:

```
for each i in 0..elements.count-1 do ... end
```

where elements is an array.
It would be nicer to use the following, more readable expression:

```
for each i in 0..<elements.count do ... end
```

The token '..<' states that the right side is not included in the range. So, 1..<10 means actually 1..9.
We will build this in, and it requires only minor changes in above function ParseRangeExpr.

```
function TParser.ParseRangeExpr: TExpr;
var
  ...
  Exclude: Boolean=False;
  Minus: TToken;
  One: TConstExpr;
  ...

begin
  ...
  if CurrentToken.Typ = ttDotDot then begin
    ...
    if CurrentToken.Typ = ttLT then begin  // ..<
      Exclude := True;
      Next;  // skip <
    end;
    UpperBound := ParseUnaryExpr;
    CheckUnaryOp(UpperBound);
    if Exclude then begin
      Minus := TToken.Create(ttMin, '-', Null, 0, 0);
      One := TConstExpr.Create(1, TToken.Create(ttNumber, '1', 1, 0, 0));
      UpperBound := TBinaryExpr.Create(UpperBound, Minus, One);
    end;
    // build call to Range(Lower, Upper)
  ...
end;
```

We check for the less than '<' token and if found set the Exclude variable to True. Next, we parse the upper bound expression, and then we have to subtract 1 from this. This means we have to create a Binary expression from the UpperBound expression, the Minus token and the constant expression 1. We feed that back into the UpperBound expression, and then continue as is. This is it. Test and run it.

```
use system

let list := [0,1,2,3,4,5,6,7,8,9]

for each i in Range(0,list.count-1) do
  print(i)
end

for each i in 0..<list.count do
  print(i)
end
```

One more thing re ranges is to add more standard functionality to the file ranges.gear as extensions. Add the following code to the file:

```
extension Range
  func toArray()
    var result := []
    for each item in self do
      listAdd(result, item)
    end
    return result
  end

  func reduce(initialValue, nextValue)
    var result := initialValue
    for each item in self do
      result := nextValue(result, item)
    end
    return result
  end
```

```
  func filter(includeElement)
    var result := []
    for each item in self where includeElement(item) do
      listAdd(result, item)
    end
    return result
  end

  func map(transform)
    var result := []
    for each item in self do
      listAdd(result, transform(item))
    end
    return result
  end

  func forEach(function)
    for each item in self do
      function(item)
    end
  end

end
```

With this ranges get similar functionality as arrays in terms of funcitonal programming.

```
var a := 1..10
a.forEach(func(n) print(n) end)
print(a.filter(x=>x%2=0))
print(a.map(x=>x^2))
print(a.reduce(0, (x,y)=>x+y))
var listOfEvens := (1..100).filter(x=>x%2=0)
```

Note that filter and map return an array!

```
var factorial := n => (1..n).reduce(1, (x,y)=>x*y)
print(factorial(10))
```

As usual reduce returns a number. The above is yet another version of the factorial function, now captured in a variable.

Sometimes you want to use a range with steps, like in 0..20 you would like to use the numbers 0, 2,4,6,8, etc. It would be helpful to create a functional solution for this.

```
var range := (0..20).step(by:2)
for each i in range do
  print(i)
end

for each i in (0..20).step(by:2) do
  print(i)
end
```

This can be solved completely in Gear code. We only have to adjust the gearlib ranges.gear. Create the following extension:

```
extension Range
  func step(.by)
    self.stepBy := by
    return self
  end
end
```

And then change classes RangeIterator and Range so that they capture the stepBy variable and the index increase is now by stepBy insetad of 1.

```
class RangeIterator
  var index := 0
  var range := Null
  var stepBy := 1

  init(range)
    self.range := range
    self.index := range.from
    self.stepBy := range.stepBy
  end

  val hasNext := self.index <= self.range.to

  val next
    var result := self.index
    self.index += self.stepBy
    return result
  end
end

class Range
  var from := Null
  var to := Null
  var stepBy := 1

  init(from, to)
    self.from := from
    self.to := to
  end

  val iterator := RangeIterator(self)
end
```

As an example consider the calculation of the prime numbers up to the number 1000. This can be done in a fantastic one-liner now, though it is not the fastest solution…

```
let primes :=
  {n for n in (3..<1000).step(by:2) where {m for m in (1..sqrt(n)).step(by:2) where n%m=0}.count=1}
```

or

```
let primes :=
  (3..<1000).step(by:2).filter(n=>{m for m in (1..sqrt(n)) .step(by:2) where n%m=0}.count=1)
```

By using step(by:2) we are skipping the even numbers, which are not prime be default. The list comprehension creates a list of divisors per number, and the filter checks if there's only one: itself.

By the way a much faster solution would be:

```
use system

func isPrime(number)
  let sqrtNum := sqrt(number)
  for var i:=3 where i<= sqrtNum, i+=2 do
    if number % i = 0 then
      return False
    end
  end
  return True
end
```

```
var primes := (3..<10000).step(by:2).filter(n=>isPrime(n))
```

Or alternatively, to solve it as an anonymous function:

```
var primes := (3..<10000).step(by:2).filter(func(number)
  let sqrtNum := sqrt(number)
  for var i:=3 where i<= sqrtNum, i+=2 do
    if number % i = 0 then
      return False
    end
  end
  return True
end)
```

Iterating over a dictionary

We have iterators for arrays and ranges, and you may be able to create your own iterator. What we don't have yet is an iterator on dictionaries. Of course this is something we have to build in. Luckily, it's not so hard, but we need to take a few steps. First, we need a function to read all keys from the dictionary. This is possible through standard FGL TFPGMap property Keys and property Data. Define the new functions in the interface IDictInstance in uDictInf.pas:

```
  IDictInstance = Interface
    ...
    function Keys: Variant;
    function Values: Variant;
  end;
```

Then implement it as a public function in TDictInstance:

```
function TDictInstance.Keys: Variant;
var
  ArrayType: IArrayable;
  Instance: IArrayInstance;
  i: Integer;
begin
  ArrayType := IArrayable(Language.Interpreter.Globals['Array']);

  Instance := IArrayInstance(TArrayInstance.Create(ArrayType as TArrayClass));
  for i := 0 to Elements.Count-1 do
    Instance.Elements.Add(Elements.Keys[i]);

  Result := Instance;
end;
```

```
function TDictInstance.Values: Variant;
var
  ArrayType: IArrayable;
  Instance: IArrayInstance;
  i: Integer;
begin
  ArrayType := IArrayable(Language.Interpreter.Globals['Array']);

  Instance := IArrayInstance(TArrayInstance.Create(ArrayType as TArrayClass));
  for i := 0 to Elements.Count-1 do
    Instance.Elements.Add(Elements.Data[i]);
  Result := Instance;
end;
```

```
implementation
uses uFunc, uVariantSupport, uArrayIntf, uArray, uLanguage;
```

```
  Globals.Store('listKeys', ICallable(TListKeys.Create));
  Globals.Store('listValues', ICallable(TListValues.Create));
```

```
    Enter(TSymbol.Create('listKeys', Enabled, False));
    Enter(TSymbol.Create('listValues', Enabled, False));
```

```
function TListKeys.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Instance: Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 1);
  Instance := ArgList[0].Value;
  if VarSupports(Instance, IDictInstance) then begin
    Result := IDictInstance(Instance).Keys;
  end
  else
    Raise ERuntimeError.Create(Token,
      'listKeys function not possible for this type.');
end;
```

```
function TListValues.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Instance: Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 1);
  Instance := ArgList[0].Value;
  if VarSupports(Instance, IDictInstance) then begin
    Result := IDictInstance(Instance).Values;
  end
  else
    Raise ERuntimeError.Create(Token,
      'listValues function not possible for this type.');
end;
```

Add the following code to dictionaries.gear in the library:

```
class DictionaryIterator
  let keys := Null
  let values := Null
  var index := 0

  init(list)
    self.keys := list.keys
    self.values := list.values
  end

  func makeItem(key, value)
    class Item
      let key := Null
      let value := Null
      init(key, value)
        self.key := key
        self.value := value
      end
    end
    return Item(key, value)
  end

  val hasNext := self.index <  length(self.keys)

  val next
    let nextItem := self.makeItem(
      self.keys[self.index], self.values[self.index])
    self.index +=1
    return nextItem
  end
end
```

```
extension Dictionary
  val iterator := DictionaryIterator(self)
end
```

And now it's possible to iterate over dictionaries, and not only that… it's easy to use the key and value data from the dictionary, see for example:

```
use system

var dict := [
  0: 'Zero',
  1: 'One',
  2: 'Two',
  3: 'Three',
  4: 'Four',
  5: 'Five',
  6: 'Six',
  7: 'Seven',
  8: 'Eight',
  9: 'Nine',
  10: 'Ten']

for each item in dict do
  print(item.key, ': ', item.value)
end
```

We can do this because the iterator returns a class object with the two fields key and value! Here's another example of what Gear is capable of:

```
use system

let table := [
  '+': (x,y)=>x+y,
  '-': (x,y)=>x-y,
  '*': (x,y)=>x*y,
  '/': (x,y)=>x/y,
  '%': (x,y)=>x%y]

var result := table['*'](7,9)
print(result)

func calc(x,y)
  for each item in table do
    print(x, item.key, y, ':\t', item.value(x,y))
  end
end

calc(16,12)
```

This prints:

```
63
16+12:  28
16-12:  4
16*12:  192
16/12:  1.33333333333333
16%12:  4
```

Quotes in strings

So far our string handling is pretty basic. Yes, we can include a few escape characters (\t for tabs and \n for line endings), but that's it. For example we cannot handle a string like this:

```
'that's a nice one.'
```

This string includes in itself an apostrophe '. In out Lexer this means the end of a string is reached and currently an error is generated. Let's first improve on that. What we want to achieve is that inside a quoted string, a double single quote '' will end up in one single quote being visible in the output. So for example the following input string:

```
var s := 'that''s a nice one.'
print(s)
```

Results in:

```
that's a nice one.
```

For this to happen we have to improve our current Lexer procedure DoString().

```
procedure TLexer.doString(const Line, Col: Integer);
var
  Lexeme: string = '';
  Value: String;
  Token: TToken;
  Typ: TTokenTyp;
begin
  Typ := ttString;

  while True do begin
    FLook := getChar;
    case FLook of
      Quote1:
        if FReader.PeekChar = Quote1 then
          FLook := getChar
        else begin
          FLook := getChar;   // consume quote '
          Break;
        end;
      LineEnding:
        begin
          Error(TToken.Create(ttNone, '', Null, Line, Col, FReader.FileIndex),
            'Lexer error: String exceeds line.');
          Break;
        end;
    end;
    Lexeme += FLook;
  end;

  Value := Lexeme;
  Lexeme := '''' + Lexeme + '''';   // including the quotes
  Token := TToken.Create(Typ, Lexeme, Value, Line, Col, FReader.FileIndex);
  Tokens.Add(Token);
end;
```

First of all, we have changed the loop structure a bit. We continue while True and Break out on certain conditions met, such as finding the end quote or a line ending.

If we detect a single quote and the next peeked char is also a quote, we skip one quote and add it to the lexeme, so it becomes part of the string. If we don't find a second quote, it means it is the end of the string and we break out, comsuming the end quote.

Next, I want to move a small functionality from the Interpreter to the Lexer. Currently in Interpreter visitor VisitPrintStmt, we use StringReplace functions to replace \t and \n for the required values… at runtime! This means we loose precious time for operations we could also do when lexing the input. Step one is to add these StringReplace functions to the Lexer:

```
procedure TLexer.doString(const Line, Col: Integer);
...
begin
  ...

  Lexeme := StringReplace(Lexeme, '\n', LineEnding, [rfReplaceAll]);
  Lexeme := StringReplace(Lexeme, '\t', Tab, [rfReplaceAll]);

  Value := Lexeme;
  Lexeme := '''' + Lexeme + '''';   // including the quotes
  Token := TToken.Create(Typ, Lexeme, Value, Line, Col, FReader.FileIndex);
  Tokens.Add(Token);
end;
```

Add the StringReplace functions right below the loop. By the way the constant Tab is defined as:

```
const Tab = #9;
```

at the top of the Lexer.
Also in the Parser we need a minor change in function ParsePrintStmt. Replace in the line stating:

```
  Terminator := TConstExpr.Create('\n', CurrentToken);
```

the '\n' with LineEnding, so that you get:

```
function TParser.ParsePrintStmt: TStmt;
...
begin
  Terminator := TConstExpr.Create(LineEnding, CurrentToken);
  Token := CurrentToken;
...
```

Finally, in the Interpreter, remove the StringReplace functions, since they are not needed anymore. It now becomes:

```
procedure TInterpreter.VisitPrintStmt(PrintStmt: TPrintStmt);
var
  i: Integer;
begin
  for i := 0 to PrintStmt.ExprList.Count-1 do
    Write((Visit(PrintStmt.ExprList[i])).toString);
  Write((Visit(PrintStmt.Terminator)).toString);
end;
```

String Interpolation

Many modern languages support interpolated strings, which means you can use variables, or even calculations right in your string. Here are some examples:

JavaScript

```
var apples = 4;
var bananas = 3;
console.log(`I have ${apples} apples`);
console.log(`I have ${apples + bananas} fruit`);
```

Kotlin

```
val quality = "superhero"
val apples = 4
val bananas = 3
val sentence = "A developer is a $quality. I have ${apples + bananas} fruit"
println(sentence)
```

Ruby

```
apples = 4
puts "I have #{apples} apples"
```

Swift

```
let apples = 4
print("I have \(apples) apples")
```

There are more, but this is to get an idea of the variations out there.

I don't want to introduce a new token, so to make it easy we'll use the Swift solution to build our own interpolation, meaning a '\ follwed by '(expression)': \(expression).

I also thought about nesting of expressions and interpolations, but that becomes unreadable in your code, so I decided not to allow that (currently).

Let's start. In TLexer define a new field 'OpenParen' of type Boolean:

```
  TLexer = class
    private
      ...
      OpenParen: Boolean;
      ...
  end;
```

This variable keeps track of whether we are inside an interpolated string, and it's value is True if we are. In the constructor set it's value initially to False.

```
constructor TLexer.Create(Reader: TReader);
begin
  ...
  OpenParen := False;
  FLook := getChar;   // get first character
  ScanTokens;  // scan all tokens
end;
```

The we move to procedure doString(). Add the following to the case statement:

```
procedure TLexer.doString(const Line, Col: Integer);
...
begin
  ...
  while True do begin
    ...
    case FLook of
      ...
      '\':
        if FReader.PeekChar = '(' then begin
          FLook := getChar;
          Typ := ttInterpolated;
          OpenParen := True;
          FLook := getChar;
          Break;
        end;
    end;
    Lexeme += FLook;
  end;

  ...
end;
```

If it finds a backslash '\' then we peek if the next character is an open paren '('. If not than nothing happens and the '\' is added to the string normally, but if a combined '\(' is found than we do have an interpolated string. Note that a space between \ and (is not allowed.

We read the next char, which is the '(' and do a couple of things: set the token type to ttInterpolated, set OpenParen to True, read the next char and break out of the string loop. This means we don't read the next part of the string in this loop, which is the part of the expression. Note that the '(' is not part of the expression; it just gets skipped.

In TTokenTyp in uToken.pas, add ttInterpolated to the constant section:

```
    //Constant values
    ttFalse, ttTrue, ttNull, ttNumber, ttString, ttChar, ttInterpolated,
```

Next, we look at procedure ScanToken in the Lexer. We now have an interpolated string and an open paren, so we need to close that, in order to finalize the expression part.

```
procedure TLexer.ScanToken(const Line, Col: Integer);
...
begin
  case FLook of
    ...

    ')' : if OpenParen then begin
            OpenParen := False;
            doString(Line, Col);
          end
          else AddToken(ttCloseParen, ')');
    ...
end
```

If variable OpenParen is True, we are in an interpolated expression, and we need to close the expression (OpenParen := False;) and read the last part of the string, which at minimum is the final single quote.

If OpenParen was False, we are not inside an interpolated expression and we just treat the closing paren as usual.

Here is an example of how this gets Lexed:

Input:

```
let apples := 4
print('I have \(apples) apples.')
```

This produces the following tokens:

```
Let ("let")
Identifier ("apples")
Assign (":=")
Number ("4") 4
Print ("print")
OpenParen ("(")
Interpolated ("'I have '") I have
Identifier ("apples")
String ("' apples.'")  apples.
CloseParen (")")
EOF ("End of file")
```

You'll note that the '\' and '(' ')' are not part of the tokens anymore. Also, note that after the token Interpolated, two more tokens follow that belong to the same string. These are the identifier 'apples' and the string 'apples.'. Somehow, we have to bring them together again as one output.

We get to the middle part: building up the interpolated expression. Create a new AST node:

```
  TInterpolatedExpr = class(TFactorExpr)
    private
      FExprList: TExprList;
    public
      property ExprList: TExprList read FExprList;
      constructor Create(AExprList: TExprList; AToken: TToken);
      destructor Destroy; override;
  end;

constructor TInterpolatedExpr.Create(AExprList: TExprList; AToken: TToken);
begin
  inherited Create(AToken);
  FExprList := AExprList;
end;

destructor TInterpolatedExpr.Destroy;
begin
  if Assigned(FExprList) then FExprList.Free;
  inherited Destroy;
end;
```

We need to parse the interpolated string in such a way that its parts are saved in an expression list.

In ParseFactor in the Parser add the case for ttInterpolated:

```
function TParser.ParseFactor: TExpr;
begin
  case CurrentToken.Typ of
    ...
    ttInterpolated: Result := ParseInterpolatedExpr;
    ttOpenParen: Result := ParseParenExpr;
    ...
end;
```

The function ParseInterpolatedExpr becomes as follows.

```
function TParser.ParseInterpolatedExpr: TExpr;
var
  ExprList: TExprList;
  Token: TToken;
begin
  Token := CurrentToken;
  ExprList := TExprList.Create();
  while CurrentToken.Typ = ttInterpolated do begin
    ExprList.Add(TConstExpr.Create(CurrentToken.Value, CurrentToken)); // Opening string
    Next;
    ExprList.Add(ParseExpr);          // Interpolated expression
  end;
  if CurrentToken.Typ = ttString then
    ExprList.Add(ParseFactor)
  else
    Error(CurrentToken, 'Expected end of string interpolation.');
  Result := TInterpolatedExpr.Create(ExprList, Token);
end;
```

We create an empty expression list that will hold all parts of the interpolated string. Next, we have a while loop that continues to parse interpolated strings. If there are no more in interpolated strings we get out of the loop and require a final string to close the task. Have a look at this example:

```
let apples := 4
let bananas := 5
print('I have \(apples) apples and \(bananas) bananas.')
```

Which produces the following token list:

```
Let ("let")
Identifier ("apples")
Assign (":=")
Number ("4") 4
Let ("let")
Identifier ("bananas")
Assign (":=")
Number ("5") 5
Print ("print")
OpenParen ("(")
Interpolated ("'I have '") I have
Identifier ("apples")
Interpolated ("' apples and '")  apples and
Identifier ("bananas")
String ("' bananas.'")  bananas.
CloseParen (")")
EOF ("End of file")
```

In the example string are two interpolated expressions, and you see that neatly appear in the token list as well. Actually, the second interpolated expression is part of the final string part of the first expression. Remember, in the Lexer, when we close the paren ‘)’, we call doString() to finalize the string part. In this case this string part also contains an interpolated expression and the process recursively processes this. That’s why the while loop above works.

The opening string part is added as a pure constant value, whereas the expression part is parsed as an expression.

Of course for every AST node we add the Printer and Resolver visitors:

```
procedure TPrinter.VisitInterpolatedExpr(InterpolatedExpr: TInterpolatedExpr);
var
  Expr: TExpr;
begin
  IncIndent;
  VisitNode(InterpolatedExpr);
  for Expr in InterpolatedExpr.ExprList do
    Visit(Expr);
  DecIndent;
end;
```

```
procedure TResolver.VisitInterpolatedExpr(InterpolatedExpr: TInterpolatedExpr);
var
  Expr: TExpr;
begin
  for Expr in InterpolatedExpr.ExprList do
    Visit(Expr);
end;
```

For the final part we move to the Interpreter. All the prework done should end up here in a fully processed string, presenting the end result.

We have stored all parts of the interpolated string in an expression list, and we should now visit all parts and concatenate the result. That's not so hard to do:

```
function TInterpreter.VisitInterpolatedExpr(InterpolatedExpr: TInterpolatedExpr
  ): Variant;
var
  Expr: TExpr;
begin
  Result := '';
  for Expr in InterpolatedExpr.ExprList do
    Result := TMath._Add(Result, Visit(Expr), Expr.Token);
end;
```

We start with an empty string, and then use the TMath._Add function to add the next evaluated expression to the left side string, which is allowed, and thus concatenating the results. While the loop continues the result is fed back to the addition.

We only need to make a minor change to TMath._Add, and this is the order of actions. Checking for a string should be moved to the first line:

```
class function TMath._Add(const Left, Right: Variant; Op: TToken): Variant;
begin
  if VarIsStr(Left) then
    Exit(Left + VarToStr(Right));
  if VarIsBool(Left) then
    Raise ERuntimeError.Create(Op, Format(ErrIncompatibleOperands, ['+']));
  if areBothNumber(Left, Right) then
    Exit(Left + Right);
  if areBothArray(Left, Right) then
    Exit(addTwoArrays(Left, Right, Op));
  if VarSupports(Left, IArrayInstance) then
    Exit(addToArray(Left, Right, Op));
  Raise ERuntimeError.Create(Op, Format(ErrIncompatibleOperands, ['+']));
end;
```

And this marks milestone, you can now do string interpolation!

```
let apples := 4
let bananas := 5
print('I have \(apples) apples and \(bananas) bananas, so \(apples+bananas) pieces of fruit.')
```

But...

Yes, there's a but. You can now do simple calculations inside the interpolated string, however try to add a function,

```
func add(a,b) => a+b
print('I have \(add(apples,bananas)) pieces of fruit.')
```

and you'll see that an error is generated. Reason is that inside the interpolated string part additional parens are used, and currently the Lexer can not handle this. Let's do something about it. We are going to count the parens.

Replace the variable OpenParen of type Boolean with the following variable:

```
      NumParens: Integer;
```

And initialize it to zero in the constructor.

```
constructor TLexer.Create(Reader: TReader);
begin
  ...
  NumParens := 0;
  FLook := getChar;   // get first character
  ...
end;
```

In procedure doString() change the part for the interpolated string:

```
procedure TLexer.doString(const Line, Col: Integer);
...
begin
  ...
  while True do begin
    ...
    case FLook of
      ...
      '\':
        if FReader.PeekChar = '(' then begin
          ...
          NumParens := 1;
          FLook := getChar;
          Break;
        end;
    end;
    Lexeme += FLook;
  end;

  ...
end;
```

And in procedure ScanToken() change the cases for '(' and ')'.

```
procedure TLexer.ScanToken(const Line, Col: Integer);
...
begin
  ...
    '(' : begin
            if NumParens>0 then NumParens +=1;
            AddToken(ttOpenParen, '(');
          end;
    ')' : if NumParens>0 then begin
            NumParens -= 1;
            if NumParens = 0 then doString(Line, Col)
            else AddToken(ttCloseParen, ')');
          end
          else AddToken(ttCloseParen, ')');
  ...
end;
```

In procedure doString(), we set NumParens to 1 if and only if we encounter the start of an interpolated string.

Then, if another '(' pops up we add 1 to the number of parens. If, on the other hand, a ')' pops up we do some checks on variable NumParens. First, if it NumParens > 0 then we decrease with 1. If NumParens is then equal to zero, we are actually at the end of the interpolated string, and we process the rest of the string. If not zero, we are actually in an expression with parens and we add the token the list.

And if NumParens is anyway zero, we treat it is a normal expression paren and add the token.

With this you can now true interpolation, like this:

```
let apples := 4
let bananas := 5
print('I have \(apples) apples and \(bananas) bananas, so \(apples+bananas) pieces of fruit.')

let truth := True
print('Is the truth \(truth | False)?')

print('Hello ' + 'world' + "!)

let c := '🤪'
print('That''s a nice smiley: \(c)')
print('Number (' + 42 + ')')

func add(a,b) => a+b
print('I have \(add(apples,bananas)) pieces of fruit.')
```

Result:

```
I have 4 apples and 5 bananas, so 9 pieces of fruit.
Is the truth True?
Hello world!
That's a nice smiley: 🤪
Number (42)
I have 9 pieces of fruit.
```

One more thing to do in this area, and that's to improve error handling. What happens if you accidentilly forget a closing paren in an interpolated expression. Yes, an rubbish message, because

it tries to read beyond the line. We can easily solve that by checking if the number of parens is greater than zero if we find a line ending, like this:

```
procedure TLexer.doString(const Line, Col: Integer);
...
begin
  ...
      LineEnding:
        begin
          Token := TToken.Create(ttNone, '', Null, Line, Col, FReader.FileIndex);
          if NumParens > 0 then
            Error(Token, 'Lexer error: Expected closing paren ")" in expression.')
          else
            Error(Token, 'Lexer error: String exceeds line.');
          Break;
        end;
  ...
end;
```

And with this we have interpolated strings including function calls. Try and test it.

Boolean calculation extended

```
class function TMath._Or(const Left, Right: Variant; Op: TToken): Variant;
begin
  if areBothBoolean(Left, Right) then begin
    if Left then
      Result := Left
    else
      Result := Right;
  end
  else if oneOfBothBoolean(Left, Right) then
    Raise ERuntimeError.Create(Op, ErrMustBeBothBoolean)
  else if areBothNumber(Left, Right) then
    Result := Left or Right
  else
    Raise ERuntimeError.Create(Op, ErrMustBeBothBoolean);
end;

class function TMath._And(const Left, Right: Variant; Op: TToken): Variant;
begin
  if areBothBoolean(Left, Right) then begin
    if not Left then
      Result := Left
    else
      Result := Right;
  end
  else if oneOfBothBoolean(Left, Right) then
    Raise ERuntimeError.Create(Op, ErrMustBeBothBoolean)
  else if areBothNumber(Left, Right) then
    Result := Left and Right
  else
    Raise ERuntimeError.Create(Op, ErrMustBeBothBoolean);
end;

class function TMath._XOr(const Left, Right: Variant; Op: TToken): Variant;
begin
  if areBothBoolean(Left, Right) then
    Result := Left xor Right
  else if oneOfBothBoolean(Left, Right) then
    Raise ERuntimeError.Create(Op, ErrMustBeBothBoolean)
  else if areBothNumber(Left, Right) then
    Result := Left xor Right
  else
    Raise ERuntimeError.Create(Op, ErrMustBeBothBoolean);
end;


class function TMath._Not(const Value: Variant; Op: TToken): Variant;
begin
  if (VarType(Value) = varBoolean) or VarIsNumeric(Value) then
    Result := not Value
  else
    Raise ERuntimeError.Create(Op, Format(ErrMustBeBoolean, ['!']));
end;
```

The Gear language

Gear is a so-called multi-paradigm programming language. It supports, imperative, procedural programming, as well as functional progamming and is also object oriented. It is a partial dynamic language that supports type inference and in fact doesn't have a static type system, but still is type-safe. Gear is case sensitive!

As a sort of tradition, every language's first program prints 'Hello world!'. In Gear, this program can be written in a single line:

```
print('Hello world!')
```

This is a full program; there's no need for a main() function or some other entry point. A program can be a simple statement.

Gear doesn't look like C-type programs, meaning there are no curly braces to create blocks of code. In the distance it looks like a Wirthian language (Pascal, Oberon), but also has similarities with Swift and Javascript. Semicolons are not used to separate statements from each other.

Variables and constants

In Gear a variable is declared as either a variable or a constant. Variables are declared using the keyword 'var', and constants are declared using 'let'.

Examples are:

```
var Planet := 'Neptune'
let Pi := 3.1415926
var radius := 2
var area := Pi * radius^2
var Ready := False
let A := "A
var initial := Null
```

Each variable declaration needs to be preceded by the keyword 'var' or 'let'. You may have noticed I'm talking about a declaration, and not so much about a statement. Should I omit the keyword 'var', it would be an assignment statement! Also, note that we don't use any type information. All variable types are inferred from the expression. So, under water, variable Planet is of type String, pi is a Number, as are radius and area, Ready is a Boolean and A is a Char, while variable initial is undefined.

After a constant gets a value other than Null, it becomes immutable and cannot be changed anymore. Variables can be changed, however, after their initialization (other than Null) they they cannot change type anymore. So, if a variable is initialized with a string value, it's type remains string type.

Next to constants and variables, there's a third variable type called "value", and the associated keyword is 'val'. A value variable is always calculated at the moment it is used. The difference with a normal variable is that the latter gets a value which remains the same until you explicitly give it a different value. The example makes things more clear. Given the following:

```
let Pi := 3.1415926
var radius := 2
var area := Pi * radius^2
print(area)  // prints 12.5663704
radius := 1
print(area)  // prints 12.5663704
```

If I now change the value of radius, the variable area will not change because of this. However, if we define the same code but now with a value, like this:

```
let Pi := 3.1415926
var radius := 2
val area := Pi * radius^2
print(area)  // prints 12.5663704
radius := 1
print(area)  // prints 3.1415926
```

Variable area is automatically recalculated , and now gets a new value.

Changing the value of variable radius is called an assignment. An assignment looks like a variable declaration, but doesn't have the keywords **let**, **var** or **val** in front of it. Assignments assign new values to variables.

Gear supports the regular constant types number, string, char and boolean. The types are not explicitly available as such; they are internal to the interpreter. In principal values are not implicitly converted to another type, with one exception. You can add any value to a string, whereby the added value is converted to string.

Strings and characters

A string is text between single quotes. Here are some examples:

```
let helloWorld := 'Hello world!'
var code := '123'
```

If you want the string to contain a single quote, like in the string 'it's me', then you need to repeat the single quote, like this 'it''s me'.

```
let label := 'it''s me'
print(label)        // prints: it's me.
```

Strings can be added to each other.

```
let hi :=  'Hello' + 'world' + '!' // 'Hello world!'
```

Numbers and Booleans can be added to strings.

```
let hiNumber :=  'Hello ' + 42 // 'Hello 42'
let truth :=  'It is ' + True // 'It is True'
```

Next to strings there are characters, which are preceded by a double quote, like this:
"a is the character a
"1 is the character 1
Characters can also be added to strings:

```
let helloWorld := 'Hello world' + "!    // 'Hello world!'
```

Sometimes adding other value types to a string can become messy and less readable.

```
let bananas := 4
let strawberries := 7
let fruits := 'There are ‘ + bananas + ' bananas and ' + strawberries + ' strawberries.'
```

This can be done easier by using socalled string interpolation.

```
let fruits := 'There are \(bananas) bananas and \(strawberries) strawberries.'
```

The values must appear in parentheses and preceded by a backslash. You can even make calculations within the parentheses.

```
let bowl := 'The bowl contains \(bananas + strawberries) pieces of fruit.'
```

Multiple interpolations can appear in a string however nested interpolation is not allowed.
Use of functions in an interpolated strings is also allowed.

```
use system
let chars := Array('0123456789abcdefghijklmnopqrstuvwxyz')

for each char in chars do
  print('Char \(char) has ascii code \(ord(char)).')
end
```

The function ord() belongs to the standard available functions of Gear. It returns the ordinal value of a character. The system library provides functionality on arrays and iteration, for instance.

Unicode strings and characters are also supported though you need to use single quotes, like for instance in this example:

```
let c := '🤪'
print(c)
```

Strings may contain ‘\n’ and ‘\t’ for newlines and tabs.

Numbers

The number type in Gear is a double precision floating point, which also handles all required integers. Examples of Gear numbers are:

1	integer
999	integer
3.1415	floating point
2.1e-2 or 2.1E-2	scientific

Booleans

There are of course two Boolean values True and False. They start with a capital.

Null

Null is a value. You can assign Null to variables, with a capital N. By assigning Null to a variable, e.g. in cases that you don't know the initial value and thus it's type, you provide a way to use the variable as a forward declared variable. Null cannot be used in calculations. This will result in a Gear runtime error.

Unassigned

Comparable to Nil in some languages. It's usually the result of a variable that doesn't exist or a non initialized variable. You can test variables if they are assigned through function assigned(variable) or simpler with ?variable.

Expressions

Basic arithmetic

Gear features the basic arithmetic operators you know from other languages:

Addition: **a + b**

```
Number + Number -> Number         // 8+9->17
String + Number -> String         // 'Hello ' + 42 -> 'Hello 42'
String + String -> String         // 'Hello ' + 'world' -> 'Hello world'
String + Char -> String           // 'Hello world' + "! -> 'Hello world!'
String + Boolean -> String        // 'It''s '  + True -> 'It's True'
```

Subtraction: **a - b**

```
Number - Number -> Number  // 8-9 -> -1
```

Multiplication: **a * b**

```
Number * Number -> Number  // 8*9 -> 72
```

Division: **a / b**

```
Number / Number -> Number  // 72/9 -> 8
```

Remainder: **a % b**

```
Number % Number -> Number  // 72%7 -> 2
```

But also Gear supports, only for numbers:

Negation: -a // -4
Shift left: a << b // 4<<2 -> 16
Shift right: a >> b // 36>>2 -> 9
Power: a ^ b // 4^3 -> 64

And special array operators:

Concatenation: **a >< b**

```
Array1 >< Array2 -> Array1Array2 // [1,2,3] >< [4,5,6] -> [1,2,3,4,5,6]
```

Dot product: **a :: b**

```
Array1 :: Array2 -> Number       // [1,2,3] :: [4,5,6] -> 32
```

More detailed info on array operations in the chapter Arrays.

Logical operators

The logical operators are

```
not: !       !True -> False,  !!True -> True
and: &       True & False -> False
or:  |       True | False -> True
xor: ~       True ~ False -> True
```

Relational operators

=	a = b	a equals b
<>	a <> b	a is not equal to b
>=	a >= b	a is greater than or equal to b
>	a > b	a is greater than b
<=	a <= b	a is less than or equal b
<	a < b	a is less than b
in	a in b	a is element of b
is	a is b	a is instance of b

If-expression

If-expressions are supported one way or the other in many languages. For example in c-like languages they appear as conditional expressions with a ternary operator.

```
C-like language: condition ? evaluated-when-true : evaluated-when-false
```

Example: `Y = X > 0 ? A : B`

Which means that if X is greater than zero than Y becomes A otherwise Y becomes B.

In order to create better readibility Gear supports the If-Expression:

```
Y := if X > 0 then A else B
```

The 'then' and the 'else' part are required.

An if-expression is to be treated as a normal expression, which means you can assign it to variables.

```
let a := 7
let b := 13
let max := if a>b then a else b
print('Maximum = \(max)')
```

If-expressions should be used wisely, and preferably not concatenated in endless if-expressions. For the latter case it's better to use the match-expression discussed hereafter.

Match-expression

A match or case expression is rarely seen in programming languages. E.g. consider the following function, which calculates the Fibonacci range: 1 1 2 3 5 8 13 21 34 55 etc. for a given number n.

```
func Fibonacci(n)
  if n = 0 then
    return 0
  elseif n = 1 then
    return 1
  else
    return Fibonacci(n-1) + Fibonacci(n-2)
  end
end
```

This can be rewritten using a match expression, which is not only shorter in code, but also more clear to read.

```
func Fibonacci(n) =>
  match n
    if 0,1 then n
    else Fibonacci(n-1) + Fibonacci(n-2)

func Factorial(n) =>
  match n
    if 0,1 then 1
    else n*Factorial(n-1)
```

A match expression consists of 1 or more if-then parts and a mandatory final else-clause.
Here's an example with more if-then cases:

```
var someCharacter := 'u'

print(someCharacter, match someCharacter
  if 'a', 'e', 'i', 'o', 'u' then ' is a vowel'
  if 'b', 'c', 'd', 'f', 'g', 'h', 'j',
     'k', 'l', 'm', 'n', 'p', 'q', 'r',
     's', 't', 'v', 'w', 'x', 'y', 'z' then ' is a consonant'
  else ' is not a vowel nor a consonant') // Prints "u is a vowel"
```

There is no limit on the number of if-then cases.

Arrays and dictionaries

Both arrays and dictionaries are created using square brackets: [] for arrays and [:] for dictionaries. Elements of both are accessed by using an index (arrays) or a key (dictionary) between the square brackets.

```
var groceries := ['Potatoes', 'Cucumber', 'Tomatoes', 'Olive-oil', 'Peanuts']
print(groceries[3])     // 'Olive-oil'
groceries[1] := 'Salad'
```

The first index of an array is 0. By using the system library additional functionality is available, such as the first and the last item.

```
use system
print(groceries.first)    // Potatoes
print(groceries.last)     // Peanuts
```

There are two ways of adding new items to an array. One is by using the concat '><' operator, and the other is by using the .add(value:) function, which comes with the system library.

```
groceries := groceries >< ['French fries']
groceries.add(value: 'Mushrooms')
```

Since concatenation works with arrays, the 'French fries' must be between brackets, so it is seen as an array of 1 item. The concatenation can also be written as:

```
groceries ><= ['French fries']
```

The add() function requires the use of the external parameter name 'value:'.

An empty array is created in either of the following ways:

```
var emptyArray := Array()
var emptyArray := []
```

An empty dictionary is created in a similar way:

```
var emptyDict := Dictionary()
var emptyDict := [:]
```

Filling a dictionary is almost similar, except that the add() function now requires a key: and value: pair.

```
use dictionaries
var smileys := ['smile': '😀', 'saint': '😇']
smileys.add(key: 'lol', value: '🤣')
smileys.add(key: 'cool', value: '😎')
smileys.add(key: 'cry', value: '😭')
print(smileys)
```

If you want to know the number of items in an array or dictionary, you use the field count.

```
let numberOfItems := smileys.count
```

If you insert an item at a specific index take care that for arrays the first index is 0.

```
groceries.insert(at: 2, value: 'Apples')
smileys.insert(at: smileys.index(of: 'lol'), key: 'kiss', value: '😘')
```

To see if a list contains a certain item, you use the .contains() function.

```
let tomatoesAvailable := groceries.contains(value: ‘Tomatoes')    // True
let smileyAvailable := smileys.contains(key: ‘lol')               // True
```

Removing items from either list is done through the delete() function.

```
groceries.delete(value: 'Cucumber')
smileys.delete(key: 'lol')
```

With the **use** statement you more or less import the code from another file. The use statement should be used before the usage of the imported code. You can use multiple files and there’s no fixed location required, meaning you can put the use statement anywhere in the code, as long as it’s before code that uses it. As an example consider:

```
print('Hello world!')

use arrays

var a := [0,1,2,3,4,5,6,7,8,9].filter(x=>x%2=0)
print(a)  // Array [0, 2, 4, 6, 8]
```

In file arrays.gear the functionality for the use of the standard Array type is defined. For example the functions filter, map and reduce are defined in this file.

The parser looks in two locations for files: first it searches in the current folder, and if not found it searches in the folder /gearlib/. If also not found there and error is generated.

Flow of control

Control flow (or flow of control) is the order in which individual statements, instructions or function calls of a program are executed or evaluated. For Boolean expression evaluation Gear offers the if-then, ensure and switch statements. Loops can be created using while, repeat and for statements. A special case is the for-each loop, which is based on an iterator pattern. All loops have in common that the statement block starts with ‘do’ and ends with ‘end’.

The while-statement is used to repeat one or more statements while a certain condition in True. The simple while loop prints the squares of numbers 1 to 10.

```
var x:=1
while x<=10 do
  print('x^2= ', x^2)
  x+=1
end
print(x) // 11
```

Note that after the loop has finished, the variable 'x' is still available and has value 11. Alternatively, you can declare the variable 'x' as part of the loop, so that it's scope ends when the loop ends.

```
while var x:=1 where x<=10 do
  print('x^2= ', x^2)
  x+=1
end
print(x) // error variable "x" unknown
```

Variable 'x' is here declared as a local variable inside the while loop, and doesn't exist outside the loop anymore. Also note the condition where variable x has to adhere to. In this case the 'where' keyword is required.

Much has been written about the for-do statement, and there are many forms available. In fact it's actually another form of writing a while statement. In the Gear for-statements it is required to define a new loop variable that is only in scope during the for-loop. The declared variable adheres to a condition and is incremented or decremented in the iterator.

The for-statement is recognizable for C-type language programmers, though without the curly braces.

```
for var i := 0 where i < 100, i+=1 do
  print(i^3)
end
```

This is actually the same as the following while loop:

```
while var i := 0 where i < 100 do
  print(i^3)
  i+=1
end
```

The repeat loop is more or less copied from the Pascal language. It is a great statement for its simplicity and straightforwardness.

```
var a := 5
var b := 1
let c := 3.1415926
repeat
  a := a - 1
  print(a*b*c)
until b>a
```

The repeat loop repeats the statements until a condition is met.

```
var n := 10
repeat
  print('n=', n)
  n-=1
until n<=0
```

Contrary to the while and for loops, the repeat loop is executed at least once.

As mentioned, a for-each loop is a special loop as it is based on iterators. For arrays, dictionaries and ranges these iterators are already predefined in libraries arrays.gear, dictionaries.gear and ranges.gear. Using the system library is sufficient to make all of them available at once.

```
use system
let scores := [8.3, 7.7, 9.0, 5.4]
var sum := 0
for each score in scores do
  sum += score
end
let average := sum / scores.count

if average > 6.0 then
  print('You passed!')
elseif (average>5.0) & (average<=6.0) then
  print('You need to work harder!')
else
  print('You failed!')
end
```

The variable 'score' is declared inplicitly after the 'each' keyword and is of the same type as an element of the 'scores' array.

The if-statement has a Boolean condition, followed by the 'then' keyword, zero or more elseif clauses, and an optional else clause. The 'end' keyword is mandatory.

It is also possible to declare the variable 'average' as part of the if-statement. The scope of the variable will be the if-statement. The 'where' clause is mandatory in such a case.

```
if let average := sum / scores.count where average > 6.0 then
  print('You passed!')
elseif (average>5.0) & (average<=6.0) then
  print('You need to work harder!')
else
  print('You failed!')
end
```

Iterating over a dictionary works more or less the same. The score variable contains both the key and value of the dictionary element. The key is available via score.key and the value is available via score.value.

```
use system
let scores := ['Math': 8.3, 'Science': 7.7, 'English': 9.0, 'Economy': 5.4]
var sum := 0
for each score in scores do
  sum += score.value
  print('You''re \(score.key) result was \(score.value).')
end
let average := sum / scores.count
print('The average is \(average).')
```

```
You're Math result was 8.3.
You're Science result was 7.7.
You're English result was 9.
You're Economy result was 5.4.
The average is 7.6.
```

You iterate over a range by using either the Range(from, to) function or by using the dotted notation.

```
for each n in 1..10 do  // or use Range(1,10)
  print(n^2)
end
```
Or if you don't want to include the right side of the range, use the '..<' symbol.

```
use system

let list := [0,1,2,3,4,5,6,7,8,9]

for each n in 0..<list.count do
  print(n^2)
end
```

Value list.count is not included in the range.

What the match expression is to expressions (which can be used for pattern matching), is the **switch** statement to statements, which can be used for quick branching in the code. Almost every programming language has a switch statement in some form. In some languages it's called a case statement.

A switch statement can have multiple cases and requires a final else clause.

```
let char := 'r'
switch char
  case 'a', 'e', 'i', 'o', 'u':
    print('\(char) is a vowel with ascii value \(ord(char)).')
  case 'b', 'c', 'd', 'f', 'g', 'h', 'j', 'k', 'l', 'm',
     'n', 'p', 'q', 'r', 's', 't', 'v', 'w', 'x', 'y', 'z':
    print('\(char) is a consonant with ascii value \(ord(char)).')
  else
    print('\(char) is not a vowel nor a consonant. Ascii value: \(ord(char)).')
end
// r is a consonant with ascii value 114.
```

The switch can be used to test against any value. You can even test if an instance of a class is of a certain class type, by using the 'case is' construct.

Note that after a case clause is executed, the switch statement is done automatically.

Gear's **print** statement lets you print all kinds of values, separated by comma's.

```
print('Hello world!')
print('The answer is ', 42)
print('This line ends with a newline ', terminator: '\n')
print('This line does not ', terminator: '')
print() // default new line
print(8*8, terminator: '!!!!\n')
```

The print statement takes expressions as arguments, separated by comma's and default it will always print a new line. It carries a parameter 'terminator', which accepts an expression. The default value of terminator = '\n', which is the newline character.

An **assignment** statement sets and/or re-sets the value stored in the storage location(s) denoted by a variable name; in other words, it copies a value into the variable.

Gear uses the ':=' operator for plain assignments, and an assignment will be a statement, instead of an expression. An assignment can only be done to an existing variable.

```
a := 1
b := a + 2

a := a + 1, which is the same as a += 1
```

Gear supports the following assignment operators:

:=	a := b			Normal assignment
+=	a += b	->	a := a + b	Addition
-+	a -= b	->	a := a – b	Subtraction
*=	a *= b	->	a := a * b	Multiplication
/=	a /= b	->	a := a / b	Division
%=	a %= b	->	a := a % b	Remainder
>><=	a >><= b	->	a := a >< b	Concatenation

Many languages have some form of assertion or guarding that certain conditions are met. If a condition is not met, usually an error is generated, or at least an early escape from a function is possible. We want to ensure that a certain condition is met with the **ensure** statement.

```
let max := 100
var number := randomLimit(max) + 1

while True do
  print('Guess a number (1 to ', max, '): ', terminator: '')

  ensure var guess := toNum(readln()) where guess <> Null else
    print('That`s not a number!')
    continue
  end

  ensure (guess > 0) & (guess <= max) else
    print('That number is not in range 1 to ', max)
    continue
  end

  if guess < number then
    print('Too low.')
  elseif guess = number then
    print('You win!')
    break
  else
    print('Too high.')
  end

end
```

Within the ‘ensure’ statement you can declare a variable or a constant and add a where-clause. If the condition is met, basically nothing happens, and we continue the rest of the statements. If the condition fails, the statements in the ‘else’ block are executed. Though there is no prescription on what should be contained in the else-block, the most practical is to use a return-statement, so that early escape from the function is possible, or like in the above example a ‘**continue**’ statement to stop this iteration and continue with the next one. The ‘**break**’ statement can be used to leave a loop completely.

The variable that is declared inside the ensure statement is part of the surrounding scope; in the shown example, this is the ‘while’ scope.

Note that the variable declaration and where-clause are optional. You can also just ensure a conditon, as shown in the second ensure statement.

The readln() function reads a string input from the command line. This can be converted to a number by using the toNum() function. If it cannot be converted the result will be Null.

When using a count-controlled loop to search through a table, it might be desirable to stop searching as soon as the required item is found. Some programming languages provide a statement such as break, which effect is to terminate the current loop immediately, and transfer control to the statement immediately after that loop.

Though there's always the possibility to use return from a statement, this in practice means you immediately return from the function. Also, since we support loop statements outside functions, a way to break early from a loop is welcome in some cases. Usually the break statement is part of an if-then statement, like in the following:

```
for var i := 0 where i<20, i+=1 do
  if i=10 then
    break
  end
  print(i)
end
```

However, a more convenient way to use break is making the condition part of the break statement, like this:

```
for var i := 0 where i<20, i+=1 do
  break on i=10
  print(i)
end
```

Both solutions are allowed in Gear. The first solution comes in handy if you need to add more complex actions in the if-then statement. The second option adds convenience.

Functions

A function is declared with the 'func' keyword. It has a name and optional parameters (the parentheses are required). Then follow statements and a final 'end'.

```
func hello(person, day)
  print('Hello \(person), it''s \(day) today.')
end
```

The function is called by its name and the required arguments between parentheses.

```
hello('Jerry', today())
// Hello Jerry, it's Monday today.
```

Sometimes it more readable if you use parameter labels in the call. You can provide a new label or use the parameter name as the label if it has a dot (.) in front of it.

```
func hello(name person, .day)
  print('Hello \(person), it''s \(day) today.')
```

```
end
```

Then, the call to hello() requires the labels followed by a colon.

```
hello(name: 'Jerry', day: today())
```

Functions can be recursive. This means inside the function it can call itself. The following function solves the famous Tower of Hanoi problem.

```
func tower(diskNumbers, source, auxilary, destination)
  if diskNumbers = 1 then
    print('\(source) \t-> \(destination)')
  else
    tower(diskNumbers-1, source, destination, auxilary)
    print('\(source) \t-> \(destination)')
    tower(diskNumbers-1, auxilary, source, destination)
  end
end

tower(3, 'src', 'aux', 'dest')
```


```
src     -> dest
src     -> aux
dest    -> aux
src     -> dest
aux     -> src
aux     -> dest
src     -> dest
```

All disks have to be moved from source to destination, but you may use the auxilary as intermediate.

A function can return any declared type. If you wish to return multiple values, it's possible to 'pack' them in a class. But you can also return dictionaries, arrays, enums and all simple types such as strings, numbers and Booleans.

```
use system

func calculate(.scores)
  class Statistics
    var min := 0
    var max := 0
    var sum := 0
  end
  var statistics := Statistics()

  for each score in scores do
    if score > statistics.max then
      statistics.max := score
    elseif score < statistics.max then
      statistics.min := score
    end
    statistics.sum += score
  end

  return statistics
end

var statistics := calculate(scores: [6, 7, 8, 9, 5, 4, 10, 3])

print(statistics.min)
```

```
print(statistics.max)
print(statistics.sum)
```

A function can be nested inside another function. The inner function then has access to variables and constants declared in the outer function.

```
func greet(name)

  var result := name
  func makeGreeting()
    result := 'Hello ' + name
  end

  makeGreeting()

  return result
end

print(greet('Emanuelle'))
```

A function can return another function as its value. This implies that a Gear function is a first-class type.

```
func startAt(x)
  func incrementBy(y)
    return x + y
  end
  return incrementBy
end

var adder1 := startAt(1)
var adder2 := startAt(5)

print(adder1(3))   // 4
print(adder2(3))   // 8
print(startAt(7)(9)) // 16
```

Functions that return only a single expression can be written in a simpler manner.

```
func add(a,b)
  return a+b
end
```

can also be formulated like an arrow function.

```
func add(a,b) => a+b
func sub(a,b) => a-b
func mul(a,b) => a*b
func div(a,b) => a/b

print(add(35,7))   // 42
print(sub(50,8))   // 42
print(mul(21,2))   // 42
print(div(168,4))  // 42
```

You can pass functions as arguments to other functions.

```
func add2Numbers(a,b) => a+b
func mul2Numbers(a,b) => a*b

func calc(x, y, function)
  return function(x,y)
end

var sum := calc(20, 22, add2Numbers)
var product := calc(2, 21, mul2Numbers)

print('Sum is: \(sum) and product is: \(product).')

// Sum is: 42 and product is: 42.
```

Gear supports anonymous functions. An anonymous function (function literal, lambda abstraction, or lambda expression) is a function definition that is not bound to an identifier. Anonymous functions are often:

- arguments being passed to higher-order functions, or
- used for constructing the result of a higher-order function that needs to return a function.

Given this the above example can be rewritten as:

```
func calc(x, y, function)
  return function(x,y)
end

var sum := calc(20, 22, func(a,b) => a+b)
var product := calc(2, 21, func(a,b) => a*b)
```

In fact, the keyword 'func' is not even required in such cases, so that you can simplify further.

```
var sum := calc(20, 22, (a,b) => a+b)
var product := calc(2, 21, (a,b) => a*b)
```

Where there's only 1 parameter in the anonymous function, the parentheses can be omitted.

```
use system
func calc(times n, function)
  for each k in 1..n  do
    print(function(k), terminator: ' ')
  end
  print()
end

calc(times: 5, x=>x^2)  // 1 4 9 16 25
calc(times: 5, x=>x^3)  // 1 8 27 64 125
calc(times: 5, x=>x^4)  // 1 16 81 256 625
```

Using anonymous functions can be handy in situations where you need to perform multiple repetitive calculations for instance. You don't have to create the function to calculate first, but you can immediately declare it in the calling function.

The following function calculates the integral over a mathematical function.

```
func integral(f, from a, to b, steps n)
  var sum := 0
  let dt := (b-a)/n
  for var i := 0 where i<n, i+=1 do
    sum += f(a + (i + 0.5) * dt)
  end
  return sum*dt
end
```

Let us now calculate the integral over the distance 0..1 in 10,000 steps for the following functions:

1) f(x) = x^2 - 2x + 4
2) f(x) = x^3
3) f(x) = x^2 + 4*x - 21

```
print(integral(func(x)=>x^2-2*x+4, from: 0, to: 1, steps: 10000))
print(integral(func(x)=>x^3, from: 0, to: 1, steps: 10000))
print(integral(func(x)=>x^2 + 4*x - 21, from: 0, to: 1, steps: 10000))
```

```
3.3333333325
0.24999999875
-18.6666666675
```

Higher order functions

The standard array type Array supports higher order functions, such as forEach, map, filter and reduce. The code of these functions can be examined in library 'arrays.gear'.

The forEach function applies an anonymous function or closure to each item in an array.

```
use system

var numbers := [1,2,3,4,5,6,7,8,9,10]

numbers.forEach(
  func(x)
    let squared := x^2
    print('Number \(x) squared is: \(squared).')
  end)
```

The higher order map function transforms all items in an array and returns a new array with the transformed items.

```
let squaredNumbers := numbers.map(x=>x^2)
print(squaredNumbers)
// [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

The filter function returns a new array with items that have passed the filter.

```
let evenNumbers := numbers.filter(x=>x%2=0)
print(evenNumbers)
// [2, 4, 6, 8, 10]
```

Each number when divided by 2 and has a remainder of 0 is an even number.

The reduce function accepts an initial value (start value), and applies a two-parameter function on the array items, so that exactly one value is returned. Internally, the items of the array are processed recursively, and accumulated after each step.

```
let sum := numbers.reduce(0, (x,y)=>x+y)
print(sum) // 55

let factorial := numbers.reduce(1, (x,y)=>x*y)
print(factorial) // 3628800
```

In Gear variables can be declared as a function.

```
var square := x => x^2
print(square(16)) // 256


var helloWorld := func()
  print('Hello world!')
end
helloWorld()  // Hello World!

var z := (x=>2^x)(9)  // parentheses are required here!
print(z)  // 512
```

Lambda calculus uses functions of 1 input. An ordinary function that requires two inputs, for instance the addition function x+y can be altered in such a way that it accepts 1 input and as output creates another function, that in turn accepts a single input. As an example, consider:

```
 (x,y)=> x+y, which can be rewritten as x=>y=>x+y
```

This method, known as currying, transforms a function that takes multiple arguments into a chain of functions each with a single argument. See example:

```
var add := x=>y=>x+y
print(add(7)(9)) // 16
```

And here is the example from the nested function paragraph but now as anonymous curried function.

```
func startAt(x) => y => x + y

var adder1 := startAt(1)
var adder2 := startAt(5)

print(adder1(3))   // 4
print(adder2(3))   // 8
print(startAt(7)(9)) // 16
```

Gear has predefined a number of standard functions.

Standard functions

pi()	the constant Pi
sqrt(x)	square root of x
sqr(x)	square of x, the same as x^2
trunc(x)	truncate a floating point value, return the integer part of x

round(x)	round floating point value to nearest integer number
abs(x)	the absolute value
arctan(x)	the inverse tangent of x
sin(x)	the sine of angle x
cos(x)	the co sine of angle x
exp(x)	the exponent of X, i.e. the number e to the power X
ln(x)	the natural logarithm of X. X must be positive
frac(x)	the fractional part of floating point value, the non integer part
ceil(x)	the lowest integer number greater than or equal to x
floor(x)	the largest integer smaller than or equal to x
ord(x)	the Ordinal value of a ordinal-type variable X
chr(x)	the character which has ASCII value X
milliseconds()	number of milliseconds since midnight 0:00
date()	string with current date
time()	string with current time in hh:mm:ss
now()	string with current time in hh:mm:ss:ms
today()	string with current day
random()	random number between 0 and 1, 1 not included
randomLimit(n)	random integer number between 0 and the limit, limit not included
toNum(s)	tries to convert a string to a number, Null if not succesful
toStr(n)	tries to convert a number to a string, Null if not succesful
readln()	returns input from the keyboard, Null if not succesful
assigned(v); ?v	return True if v is not Unassigned

length(string); length(array); length(dictionary)
listAdd(array, value); listAdd(dictionary, key, value)
listInsert(array, index, value); listInsert(dictionary, index, key, value)
listDelete(array, value); listDelete(dictionary, key)
listContains(array, value); listContains(dictionary, key)
listIndexOf(array, value); listIndexOf(dictionary, key)
listRetrieve(array, index); listRetrieve(dictionary, key)
listFirst(array); listFirst(dictionary)
listLast(array); listLast(dictionary)

Classes

A class has a name, some fields and methods, and a constructor. A class declaration starts with the keyword 'class', followed by the class name.

```
class Car
  var name := ''
  func show()
    print(self.name)
  end
  init(name)
    self.name := name
  end
end

var car := Car('Volvo')
print(car.name)   // "Volvo"
car.show()        // "Volvo"
```

A class can have the following members:

- a constructor init() // currently only 1 init is allowed
- methods / functions
- variables
- constants
- calculated properties or values

Every field (variable or constant), method, or value declared inside a class is called a member of that class. All members must be known at compile time, as it becomes very clear to the programmer what's in the class and saves him/her from errors in this respect.

In order to use members inside other members it is required to use the keyword 'self'.

In class-based programming, objects are created from classes by subroutines called constructors. In Gear this is performed through the init() method. An object is an instance of a class, and may be called a class instance or class object; instantiation is then also known as construction.

```
class Class
  var field := Null
  init(field)
    self.field := field
  end
end
```

An instance of a class is created by using the following structure.

```
var instance := Class()           // the value of 'field' is Null
var instance := Class('Hello')    // the value of 'field' is 'Hello'
```

Notice that it looks like a function call.

The fields in a class can be variable or constant. If a constant is declared with a Null value, it can receive a one-time new value. This is especially handy in class initializers, for example:

```
class Car
  let brand := Null
  init(brand)
    self.brand := brand  // brand has a value now and is immutable
  end
end
car := Car('Volvo') // brand has value 'Volvo'
car.brand := 'Merc'  // this is not allowed
```

Fields of classes can be classes themselves.

```
class Date
  var day := 1
  var month := 'January'
  var year := 1980
  init(.day, .month, .year)
    self.day := day
    self.month := month
    self.year := year
  end
  func toString()
    return 'Date: \(self.day)-\(self.month)-\(self.year)'
  end
end

class Person
  var name := 'Person'
  var birthDate := Date()
  init(name, .birthDate)
    self.name := name
    self.birthDate := birthDate
  end
  func toString()
    return self.name + ': ' + self.birthDate.toString()
  end
end

var Harry := Person('Harry', birthDate: Date(day: 23, month: 'February', year: 1987))
var Sally := Person('Sally', birthDate: Date(day: 18, month: 'July', year: 1992))

print(Harry.toString())
print(Sally.toString())
print(Person().toString())
```

Classes can inherit from each other. You can declare a base class, which can be the parent of children classes. The parent class is written between parentheses after the class name. A class can have at most one parent.

```
class Vehicle
  ...
end

class Car (Vehicle)
  ...
end

class Train (Vehicle)
  ...
end
```

This means that both class Car and Train inherit from (or are subclass of) class Vehicle. Vehicle is the parent class.

With inheritance in its simplest form, the methods at parent class level become available at child class level. Or in other words, the child class can use the methods of the parent class. If the respective called method can't be found in the child class, we'll try to find it in the parent class.

```
class Parent
  func method()
    print('Parent Method')
  end
end

class Child(Parent)
end

var child := Child()
child.method()        // "Parent Method"
```

If you define a method with the same name in a child class, it overrides the method of the parent class automatically. You can reuse the method of the parent class by using the 'inherited' statement.

```
class Child(Parent)
  func method()
    inherited method()
    print('Child Method')
  end
end

var child := Child()
child.method()

// Parent Method
// Child Method
```

Especially in init() methods it can be helpful to call the parent's init(). You do this by using:

```
inherited init()
or easier:
inherited()


class Circle
  var radius := 1
  val area := pi() * self.radius^2
  init(.radius)
    self.radius := radius
  end
end

class Cylinder (Circle)
  var height := 0
  val volume := self.area * self.height
  init(.radius, .height)
    inherited(radius: radius)
    self.height := height
  end
end
```

```
var circle := Circle(radius: 1)
var cylinder := Cylinder(radius: 2, height: 10)

print(circle.radius)     // prints 1
print(circle.area)       // prints 3.14159265358979
print(cylinder.radius)   // prints 2
print(cylinder.area)     // prints 12.5663706143592
print(cylinder.volume)   // prints 125.663706143592
```

The two ways of using inherited initialization are with or without the keyword 'init':

- inherited init(params) or
- inherited(params), which is the short version.

Both are allowed and it only applies to init(). Inheriting other functions always require the name of the function to inherit.

You can store class instances in an array. Class Person is defined in file person.gear in your 'current' folder.

```
class Person
  var name := ''
  var address := ''
  var age := 0
  var income := 0
  var cars := []

  init(.name, .address, .age, .income, .cars)
    self.name := name
    self.address := address
    self.age := age
    self.income := income
    self.cars := cars
  end
end
```

The main program makes use of this file and system.gear.

```
use person
use arrays

var people := [
  Person(name: 'Jerry', address: 'Milano, Italy', age: 39, income: 73000,
    cars: ['Opel Astra', 'Citroen C1']),
  Person(name: 'Cathy', address: 'Berlin, Germany', age: 34, income: 75000,
    cars: ['Audi A3']),
  Person(name: 'Bill', address: 'Brussels, Belgium', age: 48, income: 89000,
    cars: ['Volco XC60', 'Mercedes B', 'Smart 4x4']),
  Person(name: 'Francois', address: 'Lille, France', age: 56, income: 112000,
    cars: ['Volco XC90', 'Jaguar X-type']),
  Person(name: 'Joshua', address: 'Madrid, Spain', age: 43, income: 42000,
    cars: [])
  ]

let names := people.map(person=>person.name)
let totalIncome := people.map(person=>person.income).reduce(0, (x,y)=>x+y)
let allCars := people.flatMap(person=>person.cars)
```

The last one creates a flat list of all available cars.

Extensions

Extensions can add functionality to existing types. You can create extensions for all user declared types. An extension is defined using the keyword 'extension', followed by the name of the type that is extended.

```
extension Array
  func toString() => self.toString()
  val count := length(self)
end
```

Functionality defined in an extension immediately becomes available for all variables that were declared of the respective type, e.g. Array. Only func's and val's can be used in an extension.
A func or val defined in an extension overwrites earlier defined ones.

As mentioned an extension can only have function and/or value declarations. If any other declaration type is defined, an error is generated. If you want to add a new field for example, the usual way is to create a subclass from the original class and then add the field.

Look in arrays.gear and dictionaries.gear for extensions on both types.

Traits

A trait is a concept used in object-oriented programming, which represents a set of methods that can be used to extend the functionality of a class. Traits both provide a set of methods that implement behaviour to a class, and require that the class implement a set of methods that parameterize the provided behaviour. For inter-object communication, traits are somewhat between an object-oriented protocol (interface) and a mixin. An interface may define one or more behaviors via method signatures, while a trait defines behaviors via full method definitions: i.e., it includes the body of the methods. In contrast, mixins include full method definitions and may also carry state through member variable, while traits usually don't.

The traits in Gear are as described above, and they follow these set of rules:

- they have reusable functions,
- a class may contain zero or many traits,
- traits can use other traits, so that the functions defined in a trait become part of the new trait,
- redefined functions result in a collission.

In a class traits are defined right after the class header. Use a colon ':' to start the trait definitions.

```
class Car(Vehicle): Stringable, Drivable
end
```

In many languages you can test whether an instance belongs to a certain class, for example in Java, the function instanceOf() returns True if such is the case. In Object Pascal the operator 'is' is used for this.

We will use the keyword 'is' as well, for example:

```
trait Instance
  func instanceOf(Class)
    return self is Class
  end
end

class Number: Instance
  var code := 0
  init(code)
    self.code := code
  end
  func write()
    print(self.code)
  end
end

var number := Number(10)

if number.instanceOf(Number) then
  print(True)
end
```

This example also shows how a trait can be used. For it to be reusable, it should have few, if none, dependencies with the classes that are using it. You can use 'self' in a trait, which always points to the class instance that implements the trait.

Traits themselves as well as classes can have multiple traits, separated by commas.

```
trait One
end

trait Two
end

trait Three: One, Two
end
```

Trait Three will receive all functions defined in the other traits. Functions defined in traits that use or refer to each other must be preceded by 'self'.

The following example shows the use of traits in a class that checks if the user name and password contains correct characters.

```
use system

trait ValidatesUsername

  func validUNChars()
    var result := []
    for each ascii in Range(48, 57) do
      result.add(value: chr(ascii))
    end
    for each ascii in Range(65, 90) do
      result.add(value: chr(ascii))
    end
    for each ascii in Range(97, 122) do
      result.add(value: chr(ascii))
    end
    return result
  end

  func isUsernameValid(username)
    let chars := Array(username)
    for each char in chars do
      ensure self.validUNChars().contains(value: char) else
        return False
      end
    end
    return if chars.count < 8 then False else True
  end

end

trait ValidatesPassword

  func validPWChars()
    var result := []
    for each char in Range(34, 125) do
      result.add(value: chr(char))
    end
    return result
  end

  func isPasswordValid(password)
    let chars := Array(password)
    for each char in chars do
      ensure self.validPWChars().contains(value: char) else
        return False
      end
    end
    return if chars.count < 8 then False else True
  end

end
```

```
class Login: ValidatesUsername, ValidatesPassword
  var userName := ''
  var passWord := ''

  init(userName, passWord)
    self.userName := userName
    self.passWord := passWord
  end

  func loginEntered()
    ensure self.isUsernameValid(self.userName) else
      print('Wrong user name')
      return False
    end
    ensure self.isPasswordValid(self.passWord) else
      print('Wrong password')
      return False
    end
    return True
  end

end

print('user name: ', terminator: '')
let userName := readln()
print('password : ', terminator: '')
let passWord := readln()

var login := Login(userName, passWord)

print(if login.loginEntered() then 'Login correct' else 'Login incorrect')
```

Arrays

In Gear, an Array is extremely flexible and it's not limited to just one element type. So, as an example, an array can be:

```
['+', '-', '*', '/', '%']
[1,2,3,4,5]
['Hello', 'world', "!, 3.1415]
```

In fact you can put anything in an array, as long as it's an expression. That means, even this is an array:

```
[x=>x+x, x=>x-x, x=>x*x, x=>x/x, x=>x%x] or
[(x,y)=>x+y, (x,y)=>x-y, (x,y)=>x*y, (x,y)=>x/y, (x,y)=>x%y]
```

An array can be defined in several ways. An array can be defined like we did for class definition, creating an array-type so to speak. We can also declare an array variable directly. And for both ways, we can enforce type consistency, if we want to create an array of a specific defined type.

When you define an array type, it's also possible to define declarations such as functions and value properties. Variables, constants and other declarations such as classes are not allowed.
An instance of an array is called like a function:

```
array Numbers
  [0,1,2,3,4,5,6,7,8,9]

  func write()
    print(self)
  end
end
var numbers := Numbers()
numbers.write()
```

Gear has a default array type, called 'Array'. This standard available array comes with a few standard functions. The following table shows the standard function and the way it is implemented in the default type Array as an extension. See arrays.gear for details.

Default function	Function in type Array	Example
length(array)	.count	numbers.count
listAdd(array, value)	.add(value:)	numbers.add(value: 10)
listInsert(array, index, value)	.insert(at:, value:)	numbers.insert(at: 1, value: 9)
listDelete(array, value)	.delete(value:)	numbers.delete(value: 10)
listContains(array, value)	.contains(value:)	numbers.contains(value: 9)
listIndexOf(array, value)	.index(of:)	numbers.index(of: 9)
listRetrieve(array, index)	.retrieve(index:)	numbers.retrieve(index: 0)
listFirst(array)	.first	numbers.first
listLast(array)	.last	numbers.last

The length function also applies to strings.

```
var s := '123456789'
var l := length(s)
print(l)

array Chars
  ["a, "b, "c]
  val count := length(self)
end
var chars := Chars()
print(chars.count)
```

When the argument list in the call to create a new array is empty, the elements as defined in the array definition become the elements of the instance. However, if there are any arguments, these are taken as the contents for the array, and the predefined ones are discarded.

Then, there is a difference between 1 and more arguments. If there's exactly one argument than this must be an array expression. If there are more, it will take all values and create an array expression.

```
array myArray
  []  // empty, no elements
  val count := length(self)
end

var a := myArray([1,2,3,4,5]) // call with array expression
print(a)         // [1,2,3,4,5]
print(a.count)   // 5

var b := myArray(["a, "b, "c]) // array expression
print(b)         // ['a', 'b', 'c']
print(b.count)   // 3
var c := myArray(b)  // create a copy

var d := myArray(pi(), 'Hello', "z, 42)  // call with multiple arguments

var e := myArray(["a, "b, "c], [1 ,2, 3, [4, 5]], pi())    // multiple nested arguments

var f := myArray('Pretty cool!') // f is an array of characters
```

This way you can define your own master array type and include all functionality you want.

Arrays come with built-in mathematics. Math on arrays support the following standard operations.

Note that the types of the arrays must be the same.

- concatenation, e.g. [a, b, c] >< [d, e, f] results in [a, b, c, d, e, f]. Here we introduce a new operator '><'. In some languages the '+' operator is used for this, however that's not the right way.
- addition, e.g. [1, 2, 3] + 1 results in [2, 3, 4], and [1, 2, 3] + [3, 2, 1] results in [4, 4, 4].
- multiplication, e.g. [1, 2, 3] * 10 results in [10, 20, 30].
- subtraction and negation
- division
- dot product
- comparison of arrays
- check if an element is in an array

The Gear array system supports the creation of nested arrays, or multiple dimension arrays known as matrices.

```
var m := [[1,2,3],
          [4,5,6],
          [7,8,9]]

var n := [[3,2,1],
          [6,5,4],
          [9,8,7]]

var x := 2      // a simple value
```

It is possible to do all kinds of math on arrays and matrices.
Array concatenation is defined by the >< operator.

```
m >< n = [[1,2,3],[4,5,6],[7,8,9],[3,2,1],[6,5,4],[9,8,7]]
```

When two arrays or matrices are added, subracted, multiplied or divided together, their sizes and types must be equal.

```
m + n = [[4,4,4],[10,10,10],[16,16,16]]

m + x = [[3,4,5],[6,7,8],[9,10,11]]

m ∗ n = [[3,4,3],[24,25,24],[63,64,63]]

m ∗ x = [[2,4,6],[8,10,12],[14,16,18]]

m − n = [[−2,0,2],[−2,0,2],[−2,0,2]]

m − x = [[−1,0,1],[2,3,4],[5,6,7]]
```

Division works the same way. Other operation that are allowed are negation, test for equal and the dot-product.

```
−m = [[−1,−2,−3],[−4,−5,−6],[−7,−8,−9]]

m = n = False
m <> n = True

m :: n = not allowed! Only non−nested arrays are supported.

[1,2,3,4,5,6,7,8,9] :: [3,2,1,6,5,4,9,8,7] = 273
```

By equal sizes, we mean the number of array elements in both arrays are the same. Given the following matrix and vector, the mentioned operations are allowed.

```
var A := [[1,2,3],
          [4,5,6],
          [7,8,9]]
var V := [10,20,30]
print(A∗V)
```

results in:

```
[[10,20,30],[80,100,120],[210,240,270]]
```

Internally, the Interpreter has defined a standard array type called 'Array', like this:

```
array Array [] end
```

It is an empty shell without any functionality. That has to be added. All array expressions use this type. So in for example the following variable declaration:

```
var a := [1,2,3,4,5]
```

variable 'a' is of type Array.

Of course you can create your own array type, and add funtionality to it. For example:

```
array Numbers [] B
var numbers := Numbers([1,2,3,4,5])
```

But be careful if you do this since all operations between arrays of type Number must be the same, meaning, e.g. that

```
numbers = [1,2,3,4,5]
```

returns a Boolean False, since [1,2,3,4,5] is of type Array and not of Numbers. Even concatenation of these two will not work, because they are not the same.

Like in any other programmig language, in Gear an array item is accessed through its index. Given the array a := [1,2,3,4,5], then its first index is 0, whereas the last index is 4.

So a[0] returns the value 1, a[1] returns 2, etc.

Items in multi dimensional arrays can be found by repeated indexes, for eample in array:

m := [[1,2],[3,4],[5,6]], then m[0][0] returns 1; m[1][0] returns 3; m[2][1] returns 6.

You can also assign values to array expressions. For example:

```
var a := [1,2,3]
print(a) // [1,2,3]
a[1] := 6
print(a) // [1,6,3]
```

List comprehension

Gear supports list comprehension for single variables. The system library must be used for this to be available. The official mathematical notation for building a set though the set builder notation is:

```
result = { 2x | x ∈ N, x² > 3 }
```

The result is the set (or list) of all numbers 2 times x for which x is an element of the natural numbers and where the predicate x squared is greater than 3 must be satisfied.

In Gear the input set must be a closed set, e.g. 1..100, or [1,2,3,4,5,6].

The representation of the set builder notation in Gear is more based on existing keywords. InputSet can be an array or a range.

```
var result := { 2*x for x in 1..100 where x^2>3 }
```

The result is an array filled with numbers that satisfy the conditions.

In terms of Gear code this could be done using filter and map as well.

```
let list := [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
var result := list.filter(x=>x^2>3).map(x=>2*x)

print(result) // Array [4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30]
```

It is possible to create a function that simulates a list comprehension:

```
use arrays

func listComp(apply transform, on input, when predicate)
  return input.filter(predicate).map(transform)
end

let list := [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]

print(listComp(apply: x=>2*x, on: list, when: x=>x^2>3))
```

Alternatively, one can use the flatMap func (given the same list):

```
var evens := list.flatMap(n => if n%2=0 then [n] else [])
print(evens)

var evensSquared := list.flatMap(n => if n%2=0 then [n*n] else [])
print(evensSquared)
```

But, we'll try to implement a form of the set builder. Instead of using '|' as separator we use the keyword 'for', and for '∈' we use 'in' and for the comma, we'll use 'where'. Like this:

```
var result := { 2*x for x in inputSet where x^2>3 }
```

As input set you can use arrays and ranges, e.g.:

```
var result := { 2*x for x in [1,2,3,4,5,6,7,8,9,10] where x^2>3 }
var result := { 2*x for x in Range(1,10) where x^2>3 }
var result := { 2*x for x in 1..10 where x^2>3 }
```

In all cases the result is the same: an array.

You can use a dictionary as an input set, for example:

```
use system

let dict := [1:'One', 2:'Two', 3:'Three', 4:'Four']

var x := { x.value for x in dict where x.key%2=0}
print(x)  // [Two, Four]
```

Remember that a dictionary has two standard fields: 'key' and 'value'. Another one:

```
let words := [
  'pair':'couple',
  'oma':'grandmother',
  'man':'male',
  'woman':'female']

let w := {word.value for word in words where word.key > 'oma' }
print(w)  // [couple, female]
```

Ranges

Two examples of the use of ranges, that perform an operation, which produces exactly the same result: an array of squared numbers if the number is even.

```
var list := []
for each n in 0..<1000 where n%2=0 do
  list.add(value: n^2)
end

var list := {n^2 for n in 0..<1000 where n%2=0}
```

Though the second one takes 3 to 4 milliseconds longer, it is much more concise and readable than the first one.

In Gear library 'ranges.gear', which is used in library 'system.gear', the classes Range and RangeIterator are defined:

```
class RangeIterator
  var index := 0
  var range := Null
  var stepBy := 1

  init(range)
    self.range := range
    self.index := range.from
    self.stepBy := range.stepBy
  end

  val hasNext := self.index <= self.range.to

  val next
    var result := self.index
    self.index += self.stepBy
    return result
  end
end

class Range
  var from := Null
  var to := Null
  var stepBy := 1

  init(from, to)
    self.from := from
    self.to := to
  end

  val iterator := RangeIterator(self)
end
```

You use it like this:

```
use system

let list := [0,1,2,3,4,5,6,7,8,9]

for each i in Range(0,list.count-1) do
  print(i)
end
```

or:

```
for each i in 0..<list.count do
  print(i)
end
```

If you add the function step(by: number) to a range, iterating over the range will be in the steps as defined.

```
for each i in (0..20).step(by:2) do
  print(i)
end
```

This only prints the even numbers. The following generates all prime numbers from 3 up to 1000.

```
func isPrime(number)
  let sqrtNum := sqrt(number)
  for var i:=3 where i<= sqrtNum, i+=2 do
    if number % i = 0 then
      return False
    end
  end
  return True
end

var primes := (3..<10000).step(by:2).filter(n=>isPrime(n))
```

The step(by:2) makes sure we don't evaluate even numbers, since they cannot be prime numbers. Alternatively, you can generate primes using the list comprehension method in a one-liner.

```
let primes :=
  {n for n in (3..<1000).step(by:2) where {m for m in (1..sqrt(n)).step(by:2) where n%m=0}.count=1}
```

The first solution is 8 times faster though…

Dictionaries

Like an array, a dictionary can also be defined in two ways: as a type instance or as an expression. In the latter case the standard type Dictionary is applied. Here are a few examples:

```
dictionary Words
  [:]
end
var words := Words()
words['pair'] := 'couple'
words['collection'] := 'group'
var newDict := [:]
var family := ['Harry': 37, 'Susan': 29, 'John': 8, 'Mary': 5]
```

In the latter two cases, the predefined type Dictionary will be used.

As you can see, when you define a dictionary type, it's also possible to define declarations such as functions and values. Variables and other declarations such as enums and classes are not allowed. The dictionary elements can either be empty [:] or filled with expressions [expr:exper, expr:expr]. A dictionary expression will always be of type Dictionary, like we have type Array for array expressions.

Also, for dictionaries, a library file is available which holds extensions to type Dictionary.

```
extension Dictionary
  val count := length(self)
  val first := listFirst(self)
  val last := listLast(self)

  func toString() => self

  func add(.key, .value) => listAdd(self, key, value)
  func insert(at index, .key, .value) => listInsert(self, index, key, value)
  func delete(.key) => listDelete(self, key)
  func contains(.key) => listContains(self, key)
  func index(of key) => listIndexOf(self, key)
  func retrieve(.key) => listRetrieve(self, key)
end
```

The following table shows the standard function and the way it is implemented in the default type Dictionary as an extension.

Default function	Function in Dictionary	Example
length(dict)	.count	words.count
listAdd(dict, key, value)	.add(key:, value:)	words.add(key: 1, value: 'One')
listInsert(dict, index, key, value)	.insert(at:, key:, value:)	words.insert(at: 1, key: 2, value: 'Two')
listDelete(dict, key)	.delete(key:)	words.delete(key: 1)
listContains(dict, key)	.contains(key:)	words.contains(key: 2)
listIndexOf(dict, key)	.index(of:)	words.index(of: 2)
listRetrieve(dict, key)	.retrieve(key:)	words.retrieve(key: 2)
listFirst(dict)	.first	words.first
listLast(dict)	.last	words.last

As an example, see this code:

```
use system
var numbers := [:]

numbers.add(key: 0, value: 'Zero')
numbers.add(key: 1, value: 'One')
numbers.add(key: 2, value: 'Two')
numbers.add(key: 3, value: 'Three')
numbers.add(key: 4, value: 'Four')
numbers.add(key: 5, value: 'Five')
numbers.add(key: 6, value: 'Six')
numbers.add(key: 7, value: 'Seven')
numbers.add(key: 8, value: 'Eight')
numbers.add(key: 9, value: 'Nine')
numbers.add(key: 10, value: 'Ten')

print(numbers)

print(numbers.index(of: 7))
print(numbers.retrieve(key: 8))

print(numbers.retrieve(key: numbers.index(of: 8)))
numbers.insert(at:5, key: 11, value: 'eleven')
print(numbers)
print(numbers.retrieve(key: numbers.index(of: 8)))

print(numbers[5])

numbers[11] := 'not eleven'
print(numbers)
print(numbers[11])

print(numbers.retrieve(key: 15)) error
print(numbers.index(of:18))
```

Enums

In Gear an enumeration is declared as starting with the keyword 'enum'. A few examples of its declaration are shown below.

```
enum Color
  (Red, Blue, Yellow, Green, Orange, Purple)
end
```

This is the simplest version whereby a simple enumeration of elements is given. A variable that uses this enum is declared like this:

```
var myColor := Color.Blue
print(myColor.name)   // prints 'Blue'
print(Color.count)    // prints '6'
```

As you see an enum will have a default field called 'name' that holds the string representation of an enum plus a field that holds the number of items: count.

You can also define an enum with values, for example if you want to print different values than the name, like this:

```
enum TokenType
  (Plus='+', Min='-', Mul='*', Div='/', Rem='%')
end
var tokenType := TokenType.Mul
print(tokenType.name)   // prints 'Mul'
print(tokenType.value)  // prints '*'
```

Next to this you can create enum sets: use the keyword 'case' followed by the set-name, like this:

```
enum Color
  (None
   case Primary: Red, Yellow, Blue
   case Secondary: Orange, Green, Brown)
end

var myColor := Color.Blue
if myColor in Color.Primary then
  print(myColor.name, ' is a primary color.')  // "Blue is a primary color."
end
```

The 'Elements' field

```
use system
for each color in Color.Elements where color in Color.Primary do
  print(color)
end
```

Color.Elements provides access to the total list of enum elements.

```
print(Color.Elements)  // [None, Red, Yellow, Blue, Orange, Green, Brown]
```

Appendix unit uMath.pas

```
unit uMath;

{ This unit contains standard math functions used in the interpreter.

  Copyright (C) 2018 J. de Haan jdehaan2014@gmail.com

  This source is free software; you can redistribute it and/or modify it under the terms
  of the GNU General Public License as published by the Free Software Foundation; either
  version 2 of the License, or (at your option) any later version.

  This code is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY;
  without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
  PURPOSE.  See the GNU General Public License for more details.

  A copy of the GNU General Public License is available on the World Wide Web at
  <http://www.gnu.org/copyleft/gpl.html>. You can also obtain it by writing to the Free
  Software Foundation, Inc., 51 Franklin Street - Fifth Floor, Boston, MA 02110-1335,
USA.
}

{$mode objfpc}{$H+}
{$modeswitch advancedrecords}

interface

uses
  Classes, SysUtils, uToken, uError, math, Variants,
  uClassIntf, uArrayIntf, uEnumIntf;

type
  TMath = record
    const
      ErrIncompatibleOperands = 'Incompatible operand types for "%s" operation.';
      ErrMustBeBothNumber  = 'Both operands must be a number for "%s" operation.';
      ErrMustBeNumber = 'Operand must be a number for "%s" operation.';
      ErrMustBeBoolean = 'Operand must be a boolean for "%s" operation.';
      ErrMustBeBothBoolean = 'Both operands must be a boolean.';
      ErrDivByZero = 'Division by zero.';
      ErrConcatNotAllowed = 'Both types must be array for concat operation.';
      ErrArrayWrongTypes = 'Both variables must be array of same type.';
      ErrArrayMismatchElem = 'Mismatch in number of array elements in array operation.';

    class function _Add(const Left, Right: Variant; Op: TToken): Variant; static;
    class function _Sub(const Left, Right: Variant; Op: TToken): Variant; static;
    class function _Mul(const Left, Right: Variant; Op: TToken): Variant; static;
    class function _Div(const Left, Right: Variant; Op: TToken): Variant; static;
    class function _Rem(const Left, Right: Variant; Op: TToken): Variant; static;
    class function _Or(const Left, Right: Variant; Op: TToken): Variant; static;
    class function _And(const Left, Right: Variant; Op: TToken): Variant; static;
    class function _XOr(const Left, Right: Variant; Op: TToken): Variant; static;
    class function _Shl(const Left, Right: Variant; Op: TToken): Variant; static;
    class function _Shr(const Left, Right: Variant; Op: TToken): Variant; static;
    class function _Pow(const Left, Right: Variant; Op: TToken): Variant; static;
    class function _Concat(const Left, Right: Variant; Op: TToken): Variant; static;
    class function _Neg(const Value: Variant; Op: TToken): Variant; static;
    class function _Not(const Value: Variant; Op: TToken): Variant; static;

    class function _EQ(const Left, Right: Variant; Op: TToken): Variant; static;
    class function _NEQ(const Left, Right: Variant; Op: TToken): Variant; static;
```

```
    class function _GT(const Left, Right: Variant; Op: TToken): Variant; static;
    class function _GE(const Left, Right: Variant; Op: TToken): Variant; static;
    class function _LT(const Left, Right: Variant; Op: TToken): Variant; static;
    class function _LE(const Left, Right: Variant; Op: TToken): Variant; static;
    class function _In(const Left, Right: Variant; Op: TToken): Variant; static;
    class function _Is(const Left, Right: Variant; Op: TToken): Variant; static;

    // boolean checks
    class function areBothNumber(const Value1, Value2: Variant): Boolean; static;
    class function areBothString(const Value1, Value2: Variant): Boolean; static;
    class function areBothBoolean(const Value1, Value2: Variant): Boolean; static;
    class function oneOfBothBoolean(const Value1, Value2: Variant): Boolean; static;
    class function oneOfBothNull(const Value1, Value2: Variant): Boolean; static;

    //arrays
    class function areBothArray(const Left, Right: Variant): Boolean; static;
    class function sameArrayTypes(const Left, Right: IArrayInstance): Boolean; static;
    class function addTwoArrays(
      const Left, Right: IArrayInstance; Op: TToken):Variant; static;
    class function addToArray(
      const Left: IArrayInstance; Right: Variant; Op: TToken): Variant; static;
    class function subNumberFromArray(
      const Left: IArrayInstance; Right: Variant; Op: TToken): Variant; static;
    class function subTwoArrays(const Left, Right: IArrayInstance; Op: TToken
      ): Variant; static;
    class function mulNumberToArray(
      const Left: IArrayInstance; Right: Variant; Op: TToken): Variant; static;
    class function mulTwoArrays(const Left, Right: IArrayInstance; Op: TToken
      ): Variant; static;
    class function divArrayByNumber(
      const Left: IArrayInstance; Right: Variant; Op: TToken): Variant; static;
    class function divTwoArrays(const Left, Right: IArrayInstance; Op: TToken
      ): Variant; static;
    class function negArray(const Value: IArrayInstance; Op: TToken): Variant; static;
    class function eqTwoArrays(const Left, Right: IArrayInstance; Op: TToken
      ): Variant; static;
    class function _Dot(const Left, Right: Variant; Op: TToken): Variant; static;

    //enums
    class function areBothEnum(const Left, Right: Variant): Boolean; static;
    class function sameEnumTypes(const Left, Right: Variant): Boolean; static;
  end;

implementation
uses uArray, uLanguage;

{ TMath }

class function TMath._Add(const Left, Right: Variant; Op: TToken): Variant;
begin
  if VarIsStr(Left) then
    Exit(Left + VarToStr(Right));
  if VarIsBool(Left) then
    Raise ERuntimeError.Create(Op, Format(ErrIncompatibleOperands, ['+']));
  if areBothNumber(Left, Right) then
    Exit(Left + Right);
  //if areBothString(Left, Right) then
    //Exit(Left + Right);
  if areBothArray(Left, Right) then
    Exit(addTwoArrays(Left, Right, Op));
  if VarSupports(Left, IArrayInstance) then
    Exit(addToArray(Left, Right, Op));
  Raise ERuntimeError.Create(Op, Format(ErrIncompatibleOperands, ['+']));
end;
```

```
class function TMath._Sub(const Left, Right: Variant; Op: TToken): Variant;
begin
  if oneOfBothBoolean(Left, Right) then
    Raise ERuntimeError.Create(Op, Format(ErrIncompatibleOperands, ['-']));
  if areBothNumber(Left, Right) then
    Exit(Left - Right);
  if areBothArray(Left, Right) then
    Exit(subTwoArrays(Left, Right, Op));
  if VarSupports(Left, IArrayInstance) then
    Exit(subNumberFromArray(Left, Right, Op));
  Raise ERuntimeError.Create(Op, Format(ErrMustBeBothNumber, ['-']));
end;

class function TMath._Mul(const Left, Right: Variant; Op: TToken): Variant;
begin
  if oneOfBothBoolean(Left, Right) then
    Raise ERuntimeError.Create(Op, Format(ErrIncompatibleOperands, ['*']));
  if areBothNumber(Left, Right) then
    Exit(Left * Right);
  if areBothArray(Left, Right) then
    Exit(mulTwoArrays(Left, Right, Op));
  if VarSupports(Left, IArrayInstance) then
    Exit(mulNumberToArray(Left, Right, Op));
  Raise ERuntimeError.Create(Op, Format(ErrMustBeBothNumber, ['*']));
end;

class function TMath._Div(const Left, Right: Variant; Op: TToken): Variant;
begin
  if oneOfBothBoolean(Left, Right) then
    Raise ERuntimeError.Create(Op, Format(ErrIncompatibleOperands, ['/']));
  if areBothNumber(Left, Right) then begin
    if Right <> 0 then
      Exit(Left / Right)
    else
      Raise ERuntimeError.Create(Op, ErrDivByZero);
  end;
  if areBothArray(Left, Right) then
    Exit(divTwoArrays(Left, Right, Op));
  if VarSupports(Left, IArrayInstance) then
    Exit(divArrayByNumber(Left, Right, Op));
  Raise ERuntimeError.Create(Op, Format(ErrMustBeBothNumber, ['/']));
end;

class function TMath._Rem(const Left, Right: Variant; Op: TToken): Variant;
begin
  if oneOfBothBoolean(Left, Right) then
    Raise ERuntimeError.Create(Op, Format(ErrIncompatibleOperands, ['%']));
  try
    Result := Left mod Right;
    Exit(Result);
  except
    Raise ERuntimeError.Create(Op, Format(ErrMustBeBothNumber, ['%']));
  end;
end;

class function TMath._Or(const Left, Right: Variant; Op: TToken): Variant;
begin
  if areBothBoolean(Left, Right) then begin
    if Left then
      Result := Left
    else
      Result := Right;
  end
```

```
  else if oneOfBothBoolean(Left, Right) then
    Raise ERuntimeError.Create(Op, ErrMustBeBothBoolean)
  else if areBothNumber(Left, Right) then
    Result := Left or Right
  else
    Raise ERuntimeError.Create(Op, ErrMustBeBothBoolean);
end;

class function TMath._And(const Left, Right: Variant; Op: TToken): Variant;
begin
  if areBothBoolean(Left, Right) then begin
    if not Left then
      Result := Left
    else
      Result := Right;
  end
  else if oneOfBothBoolean(Left, Right) then
    Raise ERuntimeError.Create(Op, ErrMustBeBothBoolean)
  else if areBothNumber(Left, Right) then
    Result := Left and Right
  else
    Raise ERuntimeError.Create(Op, ErrMustBeBothBoolean);
end;

class function TMath._XOr(const Left, Right: Variant; Op: TToken): Variant;
begin
  if areBothBoolean(Left, Right) then
    Result := Left xor Right
  else if oneOfBothBoolean(Left, Right) then
    Raise ERuntimeError.Create(Op, ErrMustBeBothBoolean)
  else if areBothNumber(Left, Right) then
    Result := Left xor Right
  else
    Raise ERuntimeError.Create(Op, ErrMustBeBothBoolean);
end;

class function TMath._Shl(const Left, Right: Variant; Op: TToken): Variant;
begin
  if oneOfBothBoolean(Left, Right) then
    Raise ERuntimeError.Create(Op, Format(ErrIncompatibleOperands, ['<<']));
  try
    Result := Left shl Right;
  except
    Raise ERuntimeError.Create(Op, Format(ErrMustBeBothNumber, ['<<']));
  end;
end;

class function TMath._Shr(const Left, Right: Variant; Op: TToken): Variant;
begin
  if oneOfBothBoolean(Left, Right) then
    Raise ERuntimeError.Create(Op, Format(ErrIncompatibleOperands, ['>>']));
  try
    Result := Left shr Right;
  except
    Raise ERuntimeError.Create(Op, Format(ErrMustBeBothNumber, ['>>']));
  end;
end;

class function TMath._Pow(const Left, Right: Variant; Op: TToken): Variant;
begin
  if oneOfBothBoolean(Left, Right) then
    Raise ERuntimeError.Create(Op, Format(ErrIncompatibleOperands, ['^']));
  try
    Result := Power(Left, Right);
```

```
  except
    Raise ERuntimeError.Create(Op, Format(ErrMustBeBothNumber, ['^']));
  end;
end;

function makeArray(TypeName: string; Token: TToken): IArrayInstance;
var
  ArrayType: IArrayable;
begin
  ArrayType := IArrayable(
    Language.Interpreter.Memory.Load(TypeName, Token));
  Result := IArrayInstance(TArrayInstance.Create(ArrayType as TArrayClass));
end;

class function TMath._Concat(const Left, Right: Variant; Op: TToken): Variant;
var
  newArray, leftArray, rightArray: IArrayInstance;
begin
  if not areBothArray(Left, Right) then
    Raise ERuntimeError.Create(Op, ErrConcatNotAllowed);

  leftArray := IArrayInstance(Left);
  rightArray := IArrayInstance(Right);
  if not sameArrayTypes(leftArray, rightArray) then
    Raise ERuntimeError.Create(Op, ErrArrayWrongTypes);

  newArray := makeArray(leftArray.TypeName, Op);
  newArray.Elements.Assign(leftArray.Elements);
  newArray.Elements.Concat(rightArray.Elements);
  Result := newArray;
end;

class function TMath._Neg(const Value: Variant; Op: TToken): Variant;
begin
  if VarIsBool(Value) then
    Raise ERuntimeError.Create(Op, Format(ErrMustBeNumber, ['-']));
  if VarType(Value) = varDouble then
    Exit( -Value);
  if VarSupports(Value, IArrayInstance) then
    Exit(negArray(Value, Op));
  Raise ERuntimeError.Create(Op, Format(ErrMustBeNumber, ['-']));
end;

class function TMath._Not(const Value: Variant; Op: TToken): Variant;
begin
  if (VarType(Value) = varBoolean) or VarIsNumeric(Value) then
    Result := not Value
  else
    Raise ERuntimeError.Create(Op, Format(ErrMustBeBoolean, ['!']));
end;

class function TMath._EQ(const Left, Right: Variant; Op: TToken): Variant;
begin
  if oneOfBothNull(Left, Right) then
    Exit(Left = Right);
  if areBothBoolean(Left, Right) then
    Exit(Left = Right);
  if areBothNumber(Left, Right) then
    Exit(Left = Right);
  if areBothString(Left, Right) then
    Exit(Left = Right);
  if areBothArray(Left, Right) then
    Exit(eqTwoArrays(Left, Right, Op));
  if areBothEnum(Left, Right) and sameEnumTypes(Left, Right) then
```

```
    Exit(IEnumInstance(Left).ElemName = IEnumInstance(Right).ElemName);
  Raise ERuntimeError.Create(Op, Format(ErrIncompatibleOperands, ['equal']));
end;

class function TMath._NEQ(const Left, Right: Variant; Op: TToken): Variant;
begin
  Result := not _EQ(Left, Right, Op);
end;

class function TMath._GT(const Left, Right: Variant; Op: TToken): Variant;
begin
  if oneOfBothNull(Left, Right) then
    Exit(False);
  if areBothNumber(Left, Right) then
    Exit(Left > Right);
  if areBothString(Left, Right) then
    Exit(Left > Right);
  Raise ERuntimeError.Create(Op, Format(ErrIncompatibleOperands, ['>']));
end;

class function TMath._GE(const Left, Right: Variant; Op: TToken): Variant;
begin
  if oneOfBothNull(Left, Right) then
    Exit(False);
  if areBothNumber(Left, Right) then
    Exit(Left >= Right);
  if areBothString(Left, Right) then
    Exit(Left >= Right);
  Raise ERuntimeError.Create(Op, Format(ErrIncompatibleOperands, ['>=']));
end;

class function TMath._LT(const Left, Right: Variant; Op: TToken): Variant;
begin
  if oneOfBothNull(Left, Right) then
    Exit(False);
  if areBothNumber(Left, Right) then
    Exit(Left < Right);
  if areBothString(Left, Right) then
    Exit(Left < Right);
  Raise ERuntimeError.Create(Op, Format(ErrIncompatibleOperands, ['<']));
end;

class function TMath._LE(const Left, Right: Variant; Op: TToken): Variant;
begin
  if oneOfBothNull(Left, Right) then
    Exit(False);
  if areBothNumber(Left, Right) then
    Exit(Left <= Right);
  if areBothString(Left, Right) then
    Exit(Left <= Right);
  Raise ERuntimeError.Create(Op, Format(ErrIncompatibleOperands, ['<=']));
end;

class function TMath._In(const Left, Right: Variant; Op: TToken): Variant;
begin
  if oneOfBothNull(Left, Right) then
    Exit(False);
  if VarSupports(Right, IArrayInstance) then
    Exit(IArrayInstance(Right).Elements.Contains(Left));
  if VarSupports(Left, IEnumInstance) then
    Exit(IEnumInstance(Left).ElemSetName = Right);
  Raise ERuntimeError.Create(Op, Format(ErrMustBeBoolean, ['in']));
end;
```

```
class function TMath._Is(const Left, Right: Variant; Op: TToken): Variant;
begin
  if oneOfBothNull(Left, Right) then
    Exit(False);
  if VarSupports(Left, IGearInstance) and VarSupports(Right, IClassable) then
    Exit(IGearInstance(Left).ClassName = IClassable(Right).Name);
  Raise ERuntimeError.Create(Op, Format(ErrIncompatibleOperands, ['is']));
end;

class function TMath.areBothNumber(const Value1, Value2: Variant): Boolean;
begin
  Result := VarIsNumeric(Value1) and VarIsNumeric(Value2);
end;

class function TMath.areBothString(const Value1, Value2: Variant): Boolean;
begin
  Result := VarIsStr(Value1) and VarIsStr(Value2);
end;

class function TMath.areBothBoolean(const Value1, Value2: Variant): Boolean;
begin
  Result := VarIsBool(Value1) and VarIsBool(Value2);
end;

class function TMath.oneOfBothBoolean(const Value1, Value2: Variant): Boolean;
begin
  Result := VarIsBool(Value1) or VarIsBool(Value2);
end;

class function TMath.oneOfBothNull(const Value1, Value2: Variant): Boolean;
begin
  Result := VarIsNull(Value1) or VarIsNull(Value2);
end;

class function TMath.areBothArray(const Left, Right: Variant): Boolean;
begin
  Result := VarSupports(Left, IArrayInstance) and
            VarSupports(Right, IArrayInstance);
end;

class function TMath.sameArrayTypes(const Left, Right: IArrayInstance): Boolean;
begin
  Result := Left.TypeName = Right.TypeName;
end;

class function TMath.addTwoArrays(const Left, Right: IArrayInstance; Op: TToken):Variant;
var
  i: Integer;
  newArray: IArrayInstance;
begin
  if not sameArrayTypes(Left, Right) then
    Raise ERuntimeError.Create(Op, ErrArrayWrongTypes);
  if Left.Count <> Right.Count then
    Raise ERuntimeError.Create(Op, ErrArrayMismatchElem);

  newArray := makeArray(Left.TypeName, Op);
  for i := 0 to Left.Count-1 do
    newArray.Elements.Add(_Add(Left[i], Right[i], Op));

  Result := newArray;
end;

class function TMath.addToArray(
  const Left: IArrayInstance; Right: Variant; Op: TToken): Variant;
```

```
var
  i: Integer;
  newArray: IArrayInstance;
begin
  newArray := makeArray(Left.TypeName, Op);
  for i := 0 to Left.Count-1 do
    newArray.Elements.Add(_Add(Left[i], Right, Op));

  Result := newArray;
end;

class function TMath.subNumberFromArray(
  const Left: IArrayInstance; Right: Variant; Op: TToken): Variant;
var
  i: Integer;
  newArray: IArrayInstance;
begin
  newArray := makeArray(Left.TypeName, Op);
  for i := 0 to Left.Count-1 do
    newArray.Elements.Add(_Sub(Left[i], Right, Op));

  Result := newArray;
end;

class function TMath.subTwoArrays(
  const Left, Right: IArrayInstance; Op: TToken): Variant;
var
  i: Integer;
  newArray: IArrayInstance;
begin
  if not sameArrayTypes(Left, Right) then
    Raise ERuntimeError.Create(Op, ErrArrayWrongTypes);
  if Left.Count <> Right.Count then
    Raise ERuntimeError.Create(Op, ErrArrayMismatchElem);

  newArray := makeArray(Left.TypeName, Op);
  for i := 0 to Left.Count-1 do
    newArray.Elements.Add(_Sub(Left[i], Right[i], Op));

  Result := newArray;
end;

class function TMath.mulNumberToArray(
  const Left: IArrayInstance; Right: Variant; Op: TToken): Variant;
var
  i: Integer;
  newArray: IArrayInstance;
begin
  newArray := makeArray(Left.TypeName, Op);
  for i := 0 to Left.Count-1 do
    newArray.Elements.Add(_Mul(Left[i], Right, Op));

  Result := newArray;
end;

class function TMath.mulTwoArrays(
  const Left, Right: IArrayInstance; Op: TToken): Variant;
var
  i: Integer;
  newArray: IArrayInstance;
begin
  if not sameArrayTypes(Left, Right) then
    Raise ERuntimeError.Create(Op, ErrArrayWrongTypes);
  if Left.Count <> Right.Count then
```

```
    Raise ERuntimeError.Create(Op, ErrArrayMismatchElem);

  newArray := makeArray(Left.TypeName, Op);
  for i := 0 to Left.Count-1 do
    newArray.Elements.Add(_Mul(Left[i], Right[i], Op));

  Result := newArray;
end;

class function TMath.divArrayByNumber(
  const Left: IArrayInstance; Right: Variant; Op: TToken): Variant;
var
  i: Integer;
  newArray: IArrayInstance;
begin
  newArray := makeArray(Left.TypeName, Op);
  for i := 0 to Left.Count-1 do
    newArray.Elements.Add(_Div(Left[i], Right, Op));

  Result := newArray;
end;

class function TMath.divTwoArrays(
  const Left, Right: IArrayInstance; Op: TToken): Variant;
var
  i: Integer;
  newArray: IArrayInstance;
begin
  if not sameArrayTypes(Left, Right) then
    Raise ERuntimeError.Create(Op, ErrArrayWrongTypes);
  if Left.Count <> Right.Count then
    Raise ERuntimeError.Create(Op, ErrArrayMismatchElem);

  newArray := makeArray(Left.TypeName, Op);
  for i := 0 to Left.Count-1 do
    newArray.Elements.Add(_Div(Left[i], Right[i], Op));

  Result := newArray;
end;

class function TMath.negArray(const Value: IArrayInstance; Op: TToken): Variant;
var
  i: Integer;
  newArray: IArrayInstance;
begin
  newArray := makeArray(Value.TypeName, Op);
  for i := 0 to Value.Count-1 do
    newArray.Elements.Add(_Neg(Value[i], Op));

  Result := newArray;
end;

class function TMath.eqTwoArrays(const Left, Right: IArrayInstance; Op: TToken
  ): Variant;
var
  i: Integer;
begin
  if not sameArrayTypes(Left, Right) then Exit(False);
  if Left.Count <> Right.Count then Exit(False);

  Result := True;
  for i := 0 to Left.Count-1 do
    if _NEQ(Left[i], Right[i], Op) then
      Exit(False);
```

```
end;

class function TMath._Dot(const Left, Right: Variant; Op: TToken): Variant;
var
  tempArray: IArrayInstance;
  i: Integer;
begin
  if areBothArray(Left, Right) then begin
    tempArray := _Mul(Left, Right, Op);
    Result := 0;
    for i := 0 to tempArray.Count-1 do
      Result += tempArray[i];
  end
  else
    Raise ERuntimeError.Create(Op, Format(ErrArrayWrongTypes, ['::']));
end;

class function TMath.areBothEnum(const Left, Right: Variant): Boolean;
begin
  Result := VarSupports(Left, IEnumInstance) and VarSupports(Right, IEnumInstance);
end;

class function TMath.sameEnumTypes(const Left, Right: Variant): Boolean;
begin
  Result := IEnumInstance(Left).EnumName = IEnumInstance(Right).EnumName;
end;

end.
```

Appendix unit uStandard.pas

```
unit uStandard;


{
  This code is based on the online book 'Crafting Interpreters',
  written by Bob Nystrom. http://craftinginterpreters.com
  The code is originally written in Java.
}

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, uInterpreter, uCallable, uToken, uFunc, Variants,
  uError, math;

type

  TPi = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

  TSqrt = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

  TSqr = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

  TTrunc = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

  TRound = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

  TAbs = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

  TSin = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

  TCos = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

  TExp = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

  TLn = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;
```

```
TFrac = class(TInterfacedObject, ICallable)
  function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
end;

TArctan = class(TInterfacedObject, ICallable)
  function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
end;

TOrd = class(TInterfacedObject, ICallable)
  function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
end;

TChr = class(TInterfacedObject, ICallable)
  function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
end;

TMilliSeconds = class(TInterfacedObject, ICallable)
  function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
end;

TDate = class(TInterfacedObject, ICallable)
  function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
end;

TTime = class(TInterfacedObject, ICallable)
  function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
end;

TNow = class(TInterfacedObject, ICallable)
  function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
end;

TToday = class(TInterfacedObject, ICallable)
  function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
end;

TRandom = class(TInterfacedObject, ICallable)
  function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
end;

TRandomLimit = class(TInterfacedObject, ICallable)
  function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
end;

TLength = class(TInterfacedObject, ICallable)
  function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
end;

TListAdd = class(TInterfacedObject, ICallable)
  function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
end;

TListInsert = class(TInterfacedObject, ICallable)
  function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
end;

TListDelete = class(TInterfacedObject, ICallable)
  function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
end;

TListContains = class(TInterfacedObject, ICallable)
  function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
end;
```

```
  TListIndexOf = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

  TListRetrieve = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

  TListFirst = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

  TListLast = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

  TListKeys = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

  TListValues = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

  TAssigned = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

  TReadLn = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

  TFloor = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

  TCeil = class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

  TToNum =class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

  TToStr =class(TInterfacedObject, ICallable)
    function Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
  end;

implementation
uses uArrayIntf, uDictIntf;

{ TPi }

function TPi.Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList
  ): Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 0);
  Result := pi;
end;
```

```
{ TSqrt }

function TSqrt.Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList
  ): Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 1);
  Result := Sqrt(ArgList[0].Value);
end;

{ TSqr }

function TSqr.Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList
  ): Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 1);
  Result := Sqr(ArgList[0].Value);
end;

{ TTrunc }

function TTrunc.Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList
  ): Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 1);
  Result := Trunc(ArgList[0].Value);
end;

{ TRound }

function TRound.Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList
  ): Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 1);
  Result := Round(ArgList[0].Value);
end;

{ TAbs }

function TAbs.Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList
  ): Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 1);
  Result := Abs(ArgList[0].Value);
end;


{ TSin }

function TSin.Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList
  ): Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 1);
  Result := Sin(ArgList[0].Value);
end;

{ TCos }

function TCos.Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList
  ): Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 1);
  Result := Cos(ArgList[0].Value);
end;
```

```
{ TExp }

function TExp.Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList
  ): Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 1);
  Result := Exp(ArgList[0].Value);
end;

{ TLn }

function TLn.Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList
  ): Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 1);
  Result := Ln(ArgList[0].Value);
end;

{ TFrac }

function TFrac.Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList
  ): Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 1);
  Result := Frac(ArgList[0].Value);
end;

{ TArctan }

function TArctan.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 1);
  Result := ArcTan(ArgList[0].Value);
end;

{ TOrd }

function TOrd.Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList
  ): Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 1);
  Result := Ord(Char(ArgList[0].Value));
end;

{ TChr }

function TChr.Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList
  ): Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 1);
  Result := Chr(ArgList[0].Value);
end;

{ TMilliSeconds - milliseconds past midnight}
function TMilliSeconds.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  TS : TTimeStamp;
begin
  TFunc.CheckArity(Token, ArgList.Count, 0);
  TS := DateTimeToTimeStamp(Now);
  Result := TS.Time;
end;
```

```
{ TDate }

function TDate.Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList
  ): Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 0);
  Result := DateToStr(Date);
end;

{ TTime }

function TTime.Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList
  ): Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 0);
  Result := TimeToStr(Time);
end;

{ TNow }

function TNow.Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList
  ): Variant;
var
  ST: TSystemTime;
  S: String;
begin
  TFunc.CheckArity(Token, ArgList.Count, 0);
  DateTimeToSystemTime(Now, ST);
  with ST do
    WriteStr(S, Hour,':',Minute,':',Second,'.',MilliSecond);
  Result := S;
end;

{ TToday }

function TToday.Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList
  ): Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 0);
  Result := FormatSettings.LongDayNames[DayOfWeek(Date)];
end;

{ TRandom }

function TRandom.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 0);
  Result := Random;
end;

{ TRandomLimit }

function TRandomLimit.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 1);
  Result := Int(Random(ArgList[0].Value));
end;
```

```
{ TLength }

function TLength.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Instance: Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 1);
  Instance := ArgList[0].Value;
  if VarSupports(Instance, IArrayInstance) then
    Result := IArrayInstance(Instance).Count
  else if VarSupports(Instance, IDictInstance) then
    Result := IDictInstance(Instance).Count
  else if VarIsStr(Instance) then
    Result := Length(Instance)
  else
    Raise ERuntimeError.Create(Token,
      'Length function not possible for this type.');
end;

{ TListAdd }


function TListAdd.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Instance: Variant;
begin
  Instance := ArgList[0].Value;
  if VarSupports(Instance, IArrayInstance) then begin
    TFunc.CheckArity(Token, ArgList.Count, 2);
    IArrayInstance(Instance).Add(ArgList[1].Value);
    Result := IArrayInstance(Instance);
  end
  else if VarSupports(Instance, IDictInstance) then begin
    TFunc.CheckArity(Token, ArgList.Count, 3);
    IDictInstance(Instance).Add(ArgList[1].Value, ArgList[2].Value);
    Result := IDictInstance(Instance);
  end
  else
    Raise ERuntimeError.Create(Token,
      'listAdd function not possible for this type.');
end;

{ TListInsert }

function TListInsert.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Instance: Variant;
begin
  Instance := ArgList[0].Value;
  if VarSupports(Instance, IArrayInstance) then begin
    TFunc.CheckArity(Token, ArgList.Count, 3);
    IArrayInstance(Instance).Insert(ArgList[1].Value, ArgList[2].Value, Token);
    Result := IArrayInstance(Instance);
  end
  else if VarSupports(Instance, IDictInstance) then begin
    TFunc.CheckArity(Token, ArgList.Count, 4);
    IDictInstance(Instance).Insert(
      ArgList[1].Value, ArgList[2].Value, ArgList[3].Value, Token);
    Result := IDictInstance(Instance);
  end
  else
```

```
    Raise ERuntimeError.Create(Token,
      'listInsert function not possible for this type.');
end;

{ TListDelete }

function TListDelete.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Instance: Variant;
begin
  Instance := ArgList[0].Value;
  if VarSupports(Instance, IArrayInstance) then begin
    TFunc.CheckArity(Token, ArgList.Count, 2);
    IArrayInstance(Instance).Delete(ArgList[1].Value, Token);
    Result := IArrayInstance(Instance);
  end
  else if VarSupports(Instance, IDictInstance) then begin
    TFunc.CheckArity(Token, ArgList.Count, 2);
    IDictInstance(Instance).Delete(ArgList[1].Value, Token);
    Result := IDictInstance(Instance);
  end
  else
    Raise ERuntimeError.Create(Token,
      'listDelete function not possible for this type.');
end;

{ TListContains }

function TListContains.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Instance: Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 2);
  Instance := ArgList[0].Value;
  if VarSupports(Instance, IArrayInstance) then
    Result := IArrayInstance(Instance).Contains(ArgList[1].Value)
  else if VarSupports(Instance, IDictInstance) then
    Result := IDictInstance(Instance).Contains(ArgList[1].Value)
  else
    Raise ERuntimeError.Create(Token,
      'listContains function not possible for this type.');
end;

{ TListIndexOf }

function TListIndexOf.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Instance: Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 2);
  Instance := ArgList[0].Value;
  if VarSupports(Instance, IArrayInstance) then
    Result := IArrayInstance(Instance).IndexOf(ArgList[1].Value)
  else if VarSupports(Instance, IDictInstance) then
    Result := IDictInstance(Instance).IndexOf(ArgList[1].Value)
  else
    Raise ERuntimeError.Create(Token,
      'listIndexOf function not possible for this type.');
end;
```

```
{ TListRetrieve }

function TListRetrieve.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Instance: Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 2);
  Instance := ArgList[0].Value;
  if VarSupports(Instance, IArrayInstance) then
    Result := IArrayInstance(Instance).Retrieve(ArgList[1].Value, Token)
  else if VarSupports(Instance, IDictInstance) then
    Result := IDictInstance(Instance).Retrieve(ArgList[1].Value, Token)
  else
    Raise ERuntimeError.Create(Token,
      'listRetrieve function not possible for this type.');
end;

{ TListFirst }

function TListFirst.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Instance: Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 1);
  Instance := ArgList[0].Value;
  if VarSupports(Instance, IArrayInstance) then begin
    if IArrayInstance(Instance).Count > 0 then
      Result := IArrayInstance(Instance).Elements.First
    else Result := Unassigned;
  end
  else if VarSupports(Instance, IDictInstance) then begin
    if IDictInstance(Instance).Count > 0 then
      Result := IDictInstance(Instance).Elements.Keys[0]
    else Result := Unassigned;
  end
  else if VarIsStr(Instance) then
    Result := String(Instance)[1]
  else
    Raise ERuntimeError.Create(Token,
      'listFirst function not possible for this type.');
end;

{ TListLast }

function TListLast.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Instance: Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 1);
  Instance := ArgList[0].Value;
  if VarSupports(Instance, IArrayInstance) then begin
    if IArrayInstance(Instance).Count > 0 then
      Result := IArrayInstance(Instance).Elements.Last
    else Result := Unassigned;
  end
  else if VarSupports(Instance, IDictInstance) then begin
    if IDictInstance(Instance).Count > 0 then
      Result := IDictInstance(Instance).Elements.Keys[IDictInstance(Instance).Count-1]
    else Result := Unassigned;
  end
  else if VarIsStr(Instance) then
```

```
    Result := String(Instance)[Length(Instance)]
  else
    Raise ERuntimeError.Create(Token,
      'listLast function not possible for this type.');
end;

{ TListKeys }

function TListKeys.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Instance: Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 1);
  Instance := ArgList[0].Value;
  if VarSupports(Instance, IDictInstance) then begin
    Result := IDictInstance(Instance).Keys;
  end
  else
    Raise ERuntimeError.Create(Token,
      'listKeys function not possible for this type.');
end;

{ TListValues }

function TListValues.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Instance: Variant;
begin
  TFunc.CheckArity(Token, ArgList.Count, 1);
  Instance := ArgList[0].Value;
  if VarSupports(Instance, IDictInstance) then begin
    Result := IDictInstance(Instance).Values;
  end
  else
    Raise ERuntimeError.Create(Token,
      'listValues function not possible for this type.');
end;

{ TAssigned }

function TAssigned.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
begin
  Result := ArgList[0].Value <> Unassigned;
end;

{ TReadLn }

function TReadLn.Call
  (Token: TToken; Interpreter: TInterpreter; ArgList: TArgList): Variant;
var
  Input: String;
begin
  try
    Readln(input);
    Result := input;
  except
    Result := Null;
  end;
end;
```

```
{ TFloor }

function TFloor.Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList
  ): Variant;
begin
  Result := Floor(ArgList[0].Value);
end;

{ TCeil }

function TCeil.Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList
  ): Variant;
begin
  Result := Ceil(ArgList[0].Value);
end;

{ TToNum }

function TToNum.Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList
  ): Variant;
var
  Success: Boolean = False;
  Value: Double;
begin
  try
    Success := TryStrToFloat(ArgList[0].Value, Value);
    if Success then
      Result := Value
    else
      Result := Null;
  except
    Result := Null;
  end;
end;


{ TToStr }

function TToStr.Call(Token: TToken; Interpreter: TInterpreter; ArgList: TArgList
  ): Variant;
begin
  try
    Result := VarToStr(ArgList[0].Value);
  except
    Result := Null;
  end;
end;

initialization
  Randomize;
end.
```

Appendix unit uCollections.pas

This unit is based on standard free pascal unit fgl. Besides introducing new types TArray and TDictionary, it contains some additional functionality compared to fgl.

```
unit uCollections;

{ This unit defines classes TArray, TDictionary and TStack based on the
  Free Pascal FGL library.

  Copyright (C) 2018 J. de Haan jdehaan2014@gmail.com

  This source is free software; you can redistribute it and/or modify it under the terms
  of the GNU General Public License as published by the Free Software Foundation; either
  version 2 of the License, or (at your option) any later version.

  This code is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY;
  without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
  PURPOSE.  See the GNU General Public License for more details.

  A copy of the GNU General Public License is available on the World Wide Web at
  <http://www.gnu.org/copyleft/gpl.html>. You can also obtain it by writing to the Free
  Software Foundation, Inc., 51 Franklin Street - Fifth Floor, Boston, MA 02110-1335,
USA.
}

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, fgl, Variants;

type

  generic TArray<T> = class(specialize TFPGList<T>)
    function Contains(const AValue: T) : Boolean;
    function Contains(const AValue: T; var Index: LongInt) : Boolean;
    function Copy: TArray;
    procedure Concat(withArray: TArray);
  end;

  generic TArrayObj<T: class> = class(specialize TFPGObjectList<T>)
    function Contains(const AValue: T) : Boolean;
    function Contains(const AValue: T; var Index: LongInt) : Boolean;
    procedure Concat(withArray: TArrayObj);
  end;

  generic TDictionary<TKey, TData> = class(specialize TFPGMap<TKey, TData>)
    procedure SetValue(const AKey: TKey; AData: TData);
    function TryGetValue(const AKey: TKey; var AValue: TData): Boolean;
    function At(const Index: LongInt): TData;
    function Contains(const AKey: TKey): Boolean;
    function Contains(const AValue: TKey; var Index: LongInt) : Boolean;
    function Copy: TDictionary;
    procedure Concat(withDict: TDictionary);
  end;
```

```
  generic TDictionaryObj<TKey; TData: class> =
      class(specialize TFPGMapObject<TKey, TData>)
    procedure SetValue(const AKey: TKey; AData: TData);
    function TryGetValue(const AKey: TKey; var AValue: TData): Boolean;
    function At(const Index: LongInt): TData;
    function Contains(const AKey: TKey): Boolean;
    function Contains(const AValue: TKey; var Index: LongInt) : Boolean;
  end;

  generic TStack<T: TObject> = class
    private
      type TItems = specialize TArrayObj<T>;
    private
      FItems: TItems;
      function getCount: Integer;
      function getItem(i: Integer): T;
    public
      property Count: Integer read getCount;
      property Items[i: Integer]: T read getItem; default;
      constructor Create;
      destructor Destroy; override;
      procedure Push(Item: T);
      procedure Pop;
      function Top: T;
      function isEmpty: Boolean;
  end;

implementation

{ TArray }

function TArray.Contains(const AValue: T): Boolean;
begin
  Result := IndexOf(AValue) >= 0;
end;

function TArray.Contains(const AValue: T; var Index: LongInt): Boolean;
begin
  Index := IndexOf(AValue);
  Result := Index >= 0;
end;

function TArray.Copy: TArray;
begin
  Result := TArray.Create;
  Result.Assign(Self);
end;

procedure TArray.Concat(withArray: TArray);
var
  i: Integer;
begin
  for i := 0 to withArray.Count-1 do
    Self.Add(withArray[i]);
end;

{ TArrayObj }

function TArrayObj.Contains(const AValue: T): Boolean;
begin
  Result := IndexOf(AValue) >= 0;
end;
```

```
function TArrayObj.Contains(const AValue: T; var Index: LongInt): Boolean;
begin
  Index := IndexOf(AValue);
  Result := Index >= 0;
end;

procedure TArrayObj.Concat(withArray: TArrayObj);
var
  i: Integer;
begin
  for i := 0 to withArray.Count-1 do
    Self.Add(withArray[i]);
end;

{ TDictionary }

procedure TDictionary.SetValue(const AKey: TKey; AData: TData);
begin
  if Contains(AKey) then
    KeyData[AKey] := AData
  else
    Add(AKey, AData);
end;

function TDictionary.TryGetValue(const AKey: TKey; var AValue: TData): Boolean;
var
  Index: LongInt = -1;
begin
  Result := False;
  AValue := Default(TData);
  if Contains(AKey, Index) then begin
    AValue := At(Index);
    Result := True;
  end;
end;

function TDictionary.At(const Index: LongInt): TData;
begin
  Result := Data[Index];
end;

function TDictionary.Contains(const AKey: TKey): Boolean;
begin
  Result := IndexOf(AKey) >= 0;
end;

function TDictionary.Contains(const AValue: TKey; var Index: LongInt): Boolean;
begin
  Index := IndexOf(AValue);
  Result := Index >= 0;
end;

function TDictionary.Copy: TDictionary;
var
  i: Integer;
begin
  Result := TDictionary.Create;
  for i := 0 to Self.Count-1 do
    Result.Add(Self.Keys[i], Self.Data[i]);
end;
```

```
procedure TDictionary.Concat(withDict: TDictionary);
var
  i: Integer;
begin
  for i := 0 to withDict.Count-1 do
    Self.Add(withDict.Keys[i], withDict.Data[i]);
end;

{ TDictionaryObj }

procedure TDictionaryObj.SetValue(const AKey: TKey; AData: TData);
begin
  if Contains(AKey) then
    KeyData[AKey] := AData
  else
    Add(AKey, AData);
end;

function TDictionaryObj.TryGetValue(const AKey: TKey; var AValue: TData
  ): Boolean;
var
  Index: LongInt = -1;
begin
  Result := False;
  AValue := Nil;
  if Contains(AKey, Index) then begin
    AValue := At(Index);
    Result := True;
  end;
end;

function TDictionaryObj.At(const Index: LongInt): TData;
begin
  Result := Data[Index];
end;

function TDictionaryObj.Contains(const AKey: TKey): Boolean;
begin
  Result := IndexOf(AKey) >= 0;
end;

function TDictionaryObj.Contains(const AValue: TKey; var Index: LongInt
  ): Boolean;
begin
  Index := IndexOf(AValue);
  Result := Index >= 0;
end;

{ TStack }

function TStack.getCount: Integer;
begin
  Result := FItems.Count;
end;

function TStack.getItem(i: Integer): T;
begin
  if (i >= 0) and (i < FItems.Count) then
    Result := FItems[i]
  else
    Result := Nil;
end;
```

```
constructor TStack.Create;
begin
  FItems := TItems.Create();
end;

destructor TStack.Destroy;
begin
  FItems.Free;
  inherited Destroy;
end;

procedure TStack.Push(Item: T);
begin
  FItems.Add(Item);
end;

procedure TStack.Pop;
begin
  if FItems.Last <> Nil then
    FItems.Remove(FItems.Last);
end;

function TStack.Top: T;
begin
  Result := FItems.Last;
end;

function TStack.isEmpty: Boolean;
begin
  Result := FItems.Count = 0;
end;

end.
```