The Programming Language Opal

Yagmur SAHIN
Department of Computer Engineering
Baskent University
Ankara, Turkey
22010169@baskent.edu.tr

Brief History

OPAL (Optimized Applicative Language) is a functional programming language first developed at the Technical University of Berlin. Opal is an actively used programming language created in 1994. The language Opal has been designed by Gottfried Egger, Andreas Fett, Carola Gerke, Wolfgang Grieskamp, Michael Jatzeck, Peter Pepper, and Wolfram Schulte. It has been developed by the Compiler Construction and Programming Languages group at the Technical University Berlin headed by Prof. Dr. Peter Pepper from 1985 to 2000.

The overall appearance of Opal programs is strongly algebraic. It is a strict, purely functional programming language that combines ideas from algebraic specification and applicative programming. The core of OPAL is a strongly typed, higher-order strict functional language.

Opal – Hello World

Since Opal is a pure functional language there is no such side-effect as outputting something to stdout the only possibility is to call this from the interpreter and so get the string as a result. A simple hello world text written to standard output is as follows;

```
SIGNATURE HelloWorld
IMPORT  Com[void]       ONLY com
        Void            ONLY void

FUN hello : com[void]   -- top level command
```

```
IMPLEMENTATION HelloWorld

IMPORT  BasicIO ONLY writeLine: denotation -> com[void]

-- FUN hello : com[void] -- already declared in the Signature-Part
DEF hello == writeLine("Hello, World!")
```

Overview

An Opal program consists of a collection of structures. This collection of "structures are connected to each other by import relations. A structure is the Opal counter-part of what is usually referred to as \module", \cluster", \package", \ encapsulation", \class", etc. Opal provides higher-order functions , i.e., functions with functions as arguments and/or results.

Opal distinguish four kinds of lexical symbols:

alphanumeric symbols such as: `hallo x3 1`

graphic symbols such as: `++ % -- ==>`

separators: *blanks* , ([ )] ’

strings such as: `"Hello World!"`

EXPRESSIONS

Expressions are used to define the right-hand sides of function definitions. In general, they are evaluated each time the defined operation is applied. There is one exception, however: constant expressions , i.e., expressions that contain no free variables are evaluated during program initialization. There are essentially six different forms of expressions:

Atomic expressions , i.e., applied occurrences of names or denotations

Examples: Applied occurrences of names are

```
pi        1        +        x
```

Denotations are

```
"Hello World!"      "3.14159"      "2137"      "\t foo \n bar"
```

Tuples of expressions

```
(pi, sin, sin(pi))
```

Function applications

```
FUN @ :  A ** B ** C -> ...
```

Examples: A standard situation of a function application is

```
+(pred(x),succ(y))
```

Lambda abstractions

The general form of lambda abstraction is

\\names . expression

```
\\x.\\y.x+y
```

Case distinctions , i.e., collections of guarded expressions and conditionals

IF $guard_1$ THEN $expression_1$
⋮
IF $guard_n$ THEN $expression_n$
ELSE $expression_{n+1}$ FI

Examples: The lexicographic order <= of texts is based on a case distinction like

```
DEF R<=S ==
      IF R empty?           THEN true
      OTHERWISE
      IF S empty?           THEN false
      OTHERWISE
      IF first(R)<first(S) THEN true
      IF first(R)>first(S) THEN false
                            ELSE rest(R)<=rest(S) FI
```

Extended expressions , i.e., expressions with auxiliary names introduced in LET or WHERE clauses. All expressions in Opal are strongly typed.

expression WHERE *declarations*
LET *declarations* IN *expression*

```
IF ... THEN LET xnew == f(xold) IN h(xnew,xold)
       ELSE LET xnew == g(xold) IN k(xnew,xold) FI
```

SYNTAX

Opal's syntax has been designed to provide great uniformity in use. It is helpful at this point to give a general definition of important syntax.

{} Curly braces are used to create blocks which organize things temporally and spatially. Blocks are used to describe objects and patterns.

: Means "is a", "extends", "implements", or "replaces". The essential characteristic here is that the entity before the colon can be used as or somehow takes the place of the entity after the colon.

< > The greater than and less than symbols are used to indicate parametric polymorphism.

, The comma is used to mean building a list or tuple so that (3, 4) would create a list out of the items 3 and 4. Lists also appear in function calls.

( ) Parentheses are used as grouping in mathematical expressions and around lists and tuples to indicate that they are lists. Function calls are the juxtaposition of a function name and a tuple grouped using parentheses of the parameters to call it will.

. The period is used between names to mean member selection a.b means select b out of a.

LEXEMES

The lexemes are defined on the basis of symbols, sometimes denoted by their character representation. There are seven kinds of lexemes: keywords, identifiers, denotations, separators, comments, pragmas and layout.

Lexeme = *Keyword* ∪ *Ide* ∪ *Denotation* ∪ *Separator* ∪ *Comment* ∪ *Pragma* ∪ *Layout*

Keyword = { ALL AND ANDIF AS COMPLETELY DATA DEF DFD ELSE EX FI FUN IF IMPLEMENTATION IMPORT IN LAW LET NOT ONLY OR ORIF OTHERWISE SIGNATURE SORT THEN TYPE WHERE ** -> . : == _ === <<= ==> <=> \\ }

Ide = (*Alphanum* ∪ *Graphic*) \ (*Keyword* ∪ { -- /* */ /$ $/ })

Denotation = *String*

Separator = *Delimiter*

Comment = -- (*Symbol* \{ nl })* nl ∪ /* (*Symbol* \ { -- /* */} ∪ *Comment*) */

Pragma = /$ (*Lexeme* \ *Pragma*)* $/

Layout = *Blank*+

The keywords are reserved words, they cannot be used as identifiers, except for the keyword . (single dot), which is only recognized after one of the keywords. There are two kinds of comments: a line comment starts with the symbol -- and skips all symbols (and therefore all characters) until the next newline; a nested comment is enclosed within the symbols /* and */ and may again contain comments. Line comments have a higher priority than nested comments, i.e., the nested comment symbols /* and */ inside a line comment are ignored.

ITEMS

The basic syntactical parts of OPAL are items. The following OpAL text consists of four items, which declare and define the data type of sequences and the sequence concatenation.

```
DATA seq  == <>                          -- empty sequence
          ::(ft: data, rt: seq)          -- first element and rest sequence
FUN ++ : seq ** seq -> seq               -- declaration of concatenation function
```

WHAT IS SPECIAL ABOUT OPAL?

We consider it essential when writing easily readable programs to have access to powerful and flexible syntactic concepts that are realized in an orthogonal manner. For this reason OPAL has no built-in syntactic sugar for coping with special situations.

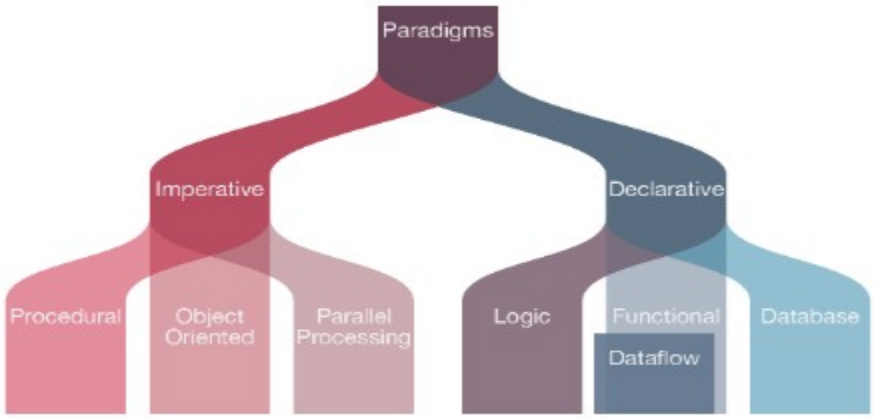
- Identifiers in OPAL consist of sequences of either alphanumeric or graphical characters.
- Function names may be placed arbitrarily before, between or after their arguments in applications.
- Collections of objects are only separated by commas, when the order is important. Hence, parameters in function calls are separated by commas but declarations are not.
- Structures have a distinct export interface, called signature part
- Parameterization provides a high degree of abstraction and reusability of structures

Application Areas

Opal concepts from Algebraic Specification and Functional Programming, which shall favor the development of production-quality software that is written in a purely functional style. The algebraic programming language OPAL has been designed as a testbed for experiments with the specification and development of functional programs. OPAL is also used for research on the highly optimizing compilation of applicative languages. This has resulted in a compiler which produces very efficient code.

The Category Of The Language

Opal is a functional programming language in the categorical list of programming languages.



Compiler

An Opal program can be compiled and linked together with a small runtime library to produce a self-contained, executable program. The OPAL compiler itself is entirely written in OPAL.

Example Of Opal Program

This is an example OPAL program, which calculates the GCD recursively. GCD is the greatest common divisor of two or more integers, which are not zero, is the largest positive integer that divides each of integers.

Signature file (declaration)

```
SIGNATURE GCD
FUN GCD: nat ** nat -> nat
```

Implementation file (definition)

```
IMPLEMENTATION GCD
IMPORT Nat COMPLETELY
DEF GCD(a,b) == IF a % b = 0 THEN b
                    ELSE IF a-b < b THEN GCD(b,a-b)
                        ELSE GCD(a-b,b)
                    FI
                FI
```

Other program that echoes the user's input until an empty line is entered. The structure MyFirstProgram is the top-level structure with echo as the top-level command;

```
SIGNATURE MyFirstProgram
IMPORT  Void          ONLY void
        Com[void]     ONLY com
FUN echo: com[void]                              -- top-level command
```

```
IMPLEMENTATION MyFirstProgram
IMPORT  Void          ONLY void nil
        Nat           ONLY nat 0
        Char          ONLY char newline
        String        ONLY string empty?
        Com           ONLY com ans exit okay fail
        ComCompose    ONLY ;
        Stream        ONLY input stdIn readLine
                           output stdOut write
DEF echo == readLine(stdIn) ; processline
FUN processline: ans[string]->com[void]
DEF processline(okay(s)) ==
    IF s empty? THEN exit(0)
    ELSE write(stdOut,s) ; (write(stdOut,newline) ;
                    (readLine(stdIn) ; processline)) FI
DEF processline(fail(_)) ==
    write(stdOut,"Cannot read user input")
```

Readability

- Too complicated
- Orthogonality is very poor due to some missing data types likes sets
- Very few special words

Writability

- Limited outputting facilities
- Strongly typed
- So abstract and difficult to use

Reliablity

- No multiple inheritance
- No interfaces
- No support
- Typing rules are very strict
- No compiler optimization

Cost

- Does not have a modern identical
- Never became a widespread language
- Has a free installation guide
- Training people to use Opal can be costly, especially if the people are used to programming with an imperative language, as it is difficult to learn a functional language

Portability

- Compatible for Linux, MacOS X and Unix environments

Pros And Cons

What Is Opal?

Opal is a object-oriented programming language. It attempts to achieve the following goals:

- Replace C, C++, Java, C#, Eiffel, Simula, and Small Talk
- Improve on current object oriented languages
- Integrate Functional Ideas
- Be both high level and low level
- Use the most advanced compiler techniques to produce efficient programs

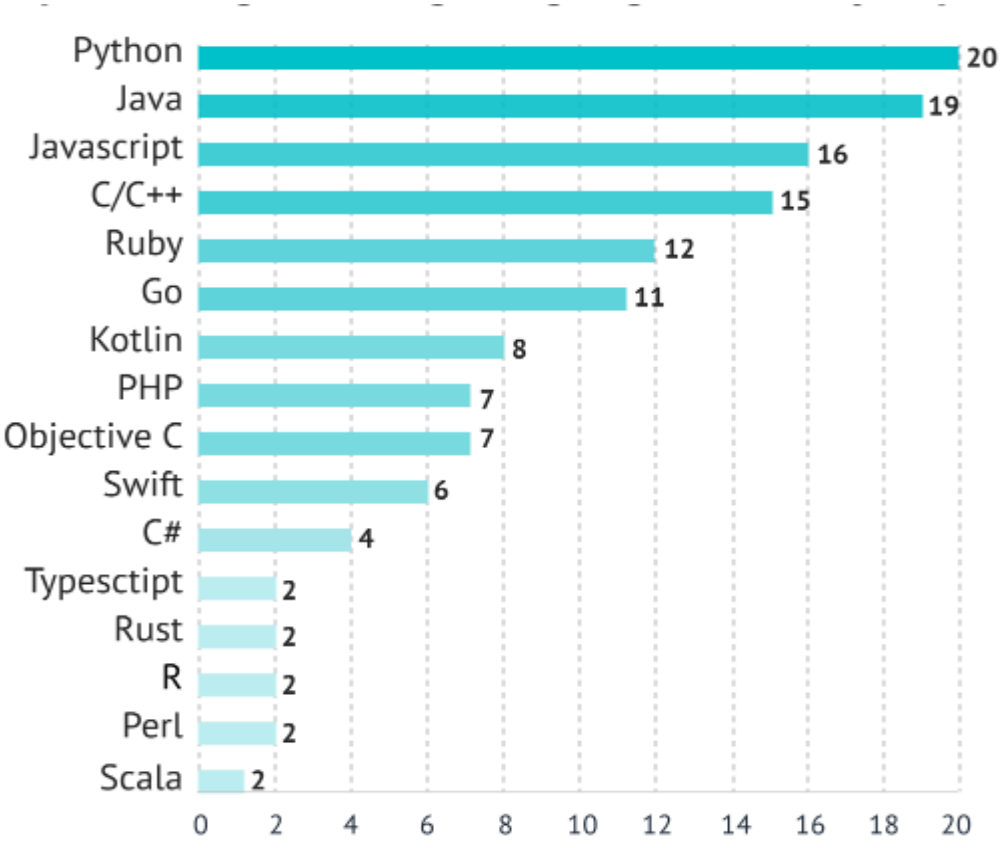
To meet this goal Opal unifies the entities of modules, functions/methods, classes and properties into its concepts of object and pattern. It also supports multiple programming paradigms (procedural, object-oriented, functional) allowing it to truly take the place of all the languages it is meant to replace. Functions as first-class objects and powerful immutability constructs give it a functional nature. Advanced compilers allow programmers to work even at the highest levels of abstraction and be assured that their program will be as efficient as possible (more efficient than many OO languages, often as efficient as none OO languages).

What Is Opal Not?

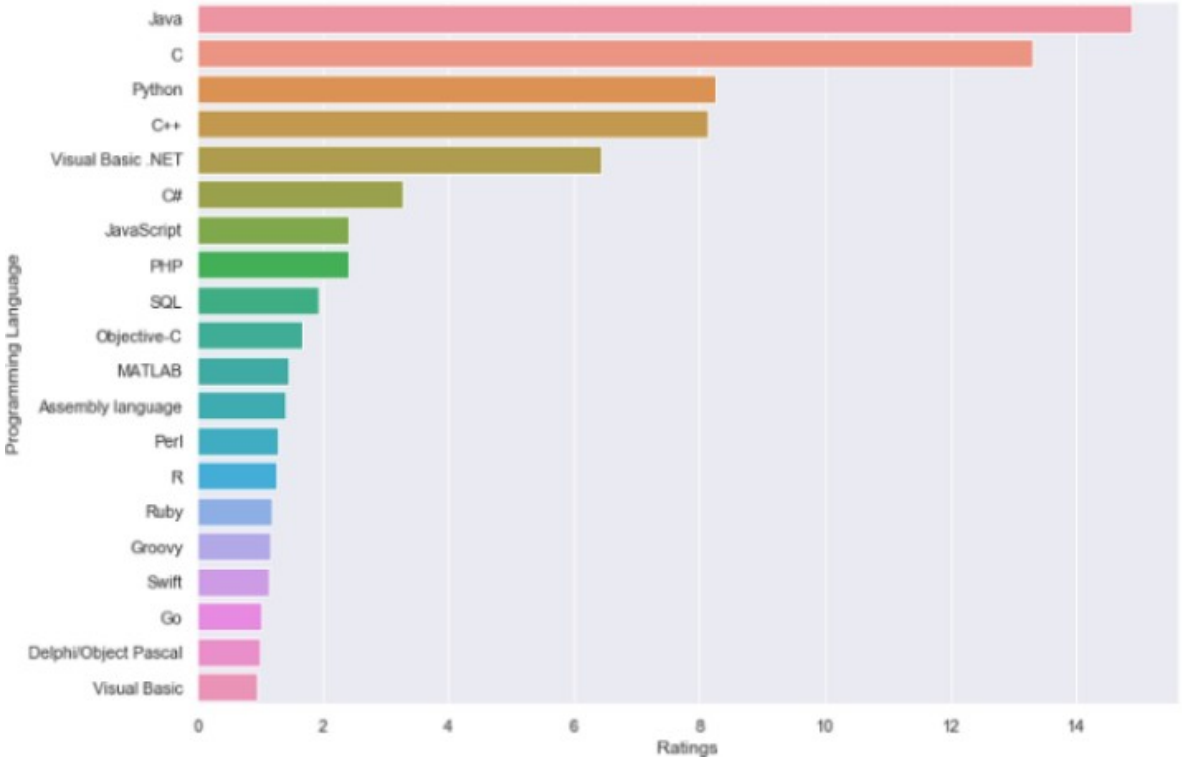
Opal is not the end all be all of programming languages. It is not perfect. Neither is it an academic or toy language. Opal is designed for solving real-world, large to small scale programming challenges.

Popularity?

Opal is neither a preferred nor a popular programming language for the developers and companies. It does not have a rating. The maintenance of Opal has been discontinued in 2015.



of Top 25 Companies Using It



References

[1] F. L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gantz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Semelson, M. Wirsing, and H. Wössner. The Munich Project Cip, volume 1. LNCF Springer, Berlin, 1985.

[2] I. Classen. Semantik der revidierten Version der algebraischen Spezifikationssprache ACT ONE. Technical Report 88/24, TU Berlin, 1988.

[3] H. Ehrig and B. Mahr. Fundamentals of Algebraic Specifications I, Equations and Initial Semantics. EATCS Monographs on Theoretical Computer Science 6, Springer, Berlin, 1985.

[4] M.S. Feather. A survey and classification of some program transformation approaches and techniques. In L.G.L.T. Meertens, editor, Program Specification and Transformation. North-Holland, 1987.

[5] K. Futatsugi, J. A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In Proc. POPL, 1985.

[6] Simon L. Peyton Jones. Implementing Functional Languages. Prentice Hall, 1992.

[7] R. Milner, M. Tofte, and R. Harper. The Definition of Standard ML. MIT Press, 1990.

[8] H. Partsch. Specification and Transformation of Programs - a Formal Approach to Software Development. Springer{Verlag, Berlin, 1990.

[9] P. Pepper. The Programming Language OPAL. Technical Report 91/10, TU Berlin, June 1991.

[10] N. Perry. Hope+. Internal repo