TyCL: an interpreter/compiler of a typed language implementation of Tcl/Tk

Andres Buss, Otlet Technologies Ltda
aabuss@otlettech.com

Abstract

TyCL (Typed Command Language) is an implementation of the Tcl language written in TyCL itself. The language follows Tcl's syntax, but given that TyCL is meant to be a full compiler that generates machine code, a new set of features, expansions and cuts had to be applied to the Tcl's syntax that TyCL understand in order to help the compiler to produce better results. Included in those modifications are the concept of types (hence the name of the compiler), which are optional but when used makes TyCL behave more as an static language than a dynamic one.

Introduction

Even though the performance of Tcl's virtual machine has been improved since its conception (including the Non-Recursive-Engine[1]) it still runs at few orders of magnitude slower than many other languages at this time. After looking at its source code (written in C) one has to wonder how hard would be to try something in such performance endeavors. There is another path that can be taken (may be a lot more painful and time consuming but... hopefully... more gratifying) which is to develop a new implementation of Tcl's features and syntax using another approach, and may be, a different coding-language (other than C), and in the process of doing it (now that one is willing to take the pain) why not try to include some other nice and interesting features? Here is where TyCL come in to the picture.

The main goals of the TyCL compiler/interpreter is to try to improve Tcl's performance, minimize its memory footprint and provide simple mechanisms to interact with other components within a system by creating a whole new implementation of it from scratch. It could be written in C or C++ or even Objective-C (following the people's frenzy for Apple's products at these days) but it might be thought (as Alan Kay did with Smalltalk [2]) that the best language to code this new implementation would be TyCL itself, so we have to deal with one syntax and one set of idiosyncrasies instead of two, besides that any improvements in the language is directly applied to the compiler. In order to do this one must have the first usable implementation of the language already running (chicken & egg problem) to compile itself, in this case, at the beginning, TyCL's implementation is coded in plain Tcl8.5 in order to obtain the first iteration of the needed bootstrapping[3] process and have TyCL compiled by itself.

As a Tcl compiler, TyCL should produce better results as programs/scripts include more detailed information about the data it has to process (information that is not there right know, given the strong dynamic nature of Tcl), which means: is less-difficult to generate efficient machine-code when the compiler knows exactly what data is manipulating... or in other words: TyCL has to include the concept of inline/static data types in its syntax in order to provide this information.

TyCL's architecture

Like any other compiler[4], TyCL has an architecture composed by stages, from the lexical parser to a machine-code generator (see figure 1), the only possible difference is that one of its intermediate-representations could be interpreted by TyCL's Virtual Machine (VM).

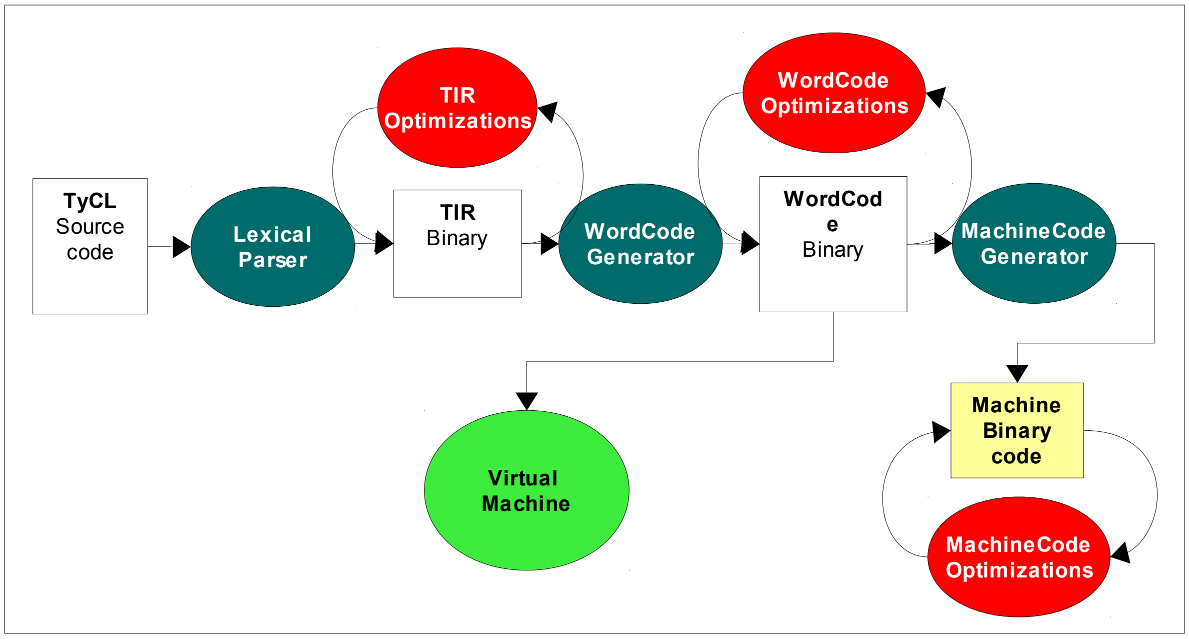

Figure 1. TyCL's block diagram

The program's source code begins its transformation at the lexical parser, which converts the whole text in a kind of Abstract-Syntax-Tree[5] called TIR (TyCL-Intermediate-Representation), by later be synthesized in a WordCode format (WC)[1] by the WC-Generator. At this point, the partial result is composed by a binary block of Word-Codes and a set of data tables (a sort of literals, symbols and references tables) that either could be passed to the TyCL's VM or injected into the machine-code-generator to be transformed in an executable or a system-binary-object (NativeCode).

At the moment, TyCL doesn't implement any form of code optimizations, it runs as a very naive compiler, but they could be integrated into the processing-chain at the TIR, WC and NativeCode sub-products stages as a series sub-processes.

1 The 'WordCode' name was preferred over 'ByteCode' because the granularity of the format is 4-bytes.

The Lexical Parser

As described before, this process converts the source-code from the text representation to a TIR representation using the modified/expanded Tcl's syntax understood by TyCL. Such syntax is very close to Tcl's, except for:

- **The type system**

  The parser tries to imply every value's type by analyzing its value/structure, for example: the text *23.65* is identified as a real number and then its type is *float*, but if the same number is written as: *"23.65"* its type is taken as *string* (which means plain text) because of being quoted. The basic available types are:

  - integer:
    any text composed only by digits from 0 to 9.
    any text in hexadecimal notation (0x##)
  - boolean:
    any of: *t*|*T*|*true*, *f*|*F*|*false*.
  - float:
    any number that includes a single decimal separator.
  - char:
    any single character surrounded by quotes or braces.
  - string:
    any text surrounded by quotes or braces. This is the chosen default type if no other could be detected.

  Besides the automatic detection of data types, the program can inform the parser that a certain value should be understood in a particular way by directly declaring its type, a process commonly known as casting, in this particular case: casting from text to the desired type. The way of declaring a casting is:

  TYPE:VALUE

  as an example, the following fragment shows how the types are detected/declared:

```
set a 7                ;# auto-detected integer = 7
set b integer:"456"    ;# integer = 456
set c true             ;# auto-detected boolean = true
set d "false"          ;# auto-detected string = 'false'
set e boolean:45       ;# boolean = true
set f 3.1415           ;# auto-detected float = 3.1415
set g float:3.4        ;# float = 3.4
set h float:{56.8576}  ;# float = 56.8576
set i char:w           ;# char = 'w'
set j char:67          ;# char = 'C'
set k hello            ;# default-detected string = 'hello'
set l "hello world"    ;# auto-detected string = 'hello world'
set m string:6.28      ;# string = '6.28'
```

Each of the created variables in the previous example has then a value with a particular type, but also each one of them can have any other typed-value later on. In order to force a variable to always have values of an specified type, the new command: "*define*" was added to the syntax to complement the traditional assignment command: "*set*", that does exactly the same assignment but also bounds a type with the variable being assigned. So instead of writing:

```
set a float:5.6
```

one would write:

```
define b float:5.6
```

the difference between *a* and *b* is that *b* can only hold values of type *float* and its type can not be changed until it is destroyed and created again.

- **References**

Additionally to these basic types, there is on more basic but almost hidden type of value called: "*reference*" that, as its name suggests, allows a variable to reference another one in a transparent way. It is thought to be almost hidden because values of this type are not directly manipulated as the other basic types (like integers or floats, which can be changed in expressions), instead, they are operated through another new command called: "*let*" , whose only function is to assign and acquire references to and from variables. Its syntax is:

```
let VAR           ;# returns a reference to the variable: VAR
let VAR_1 VAR_2   ;# sets VAR_1 as a reference to VAR_2
```

Only the command "*let*" is allowed to set, get and change reference-values, this means that for the rest of the program, when such a variable is operated, what is actually manipulated is the value of the final variable referenced by the original one, for example:

```
set a 4     ;# variable a = 4
let b a     ;# variable b refers to a
let c b     ;# variable c refers to b
set c 9     ;# setting c = 9 actually sets: a = 9 because:
             # c → b → a
```

As this example shows, references are transparently followed internally. From a strictly Tcl's point of view they don't exist.

- **Structural data types**

  Just as having basic data types to identify values more precisely can help the compiler in its job, structural data types could make its life better yet, allowing the compiler to generate more compact and hopefully efficient code by knowing how the data/values should be organized in memory.

  There are many ways of organizing data but the most generic ones could be:

Name	Length	Internal types	Description
arrays	fixed	multi-typed	A group with a fixed number of values
typed arrays	fixed	single-typed	An array of values of the same type
lists	variable	multi-typed	A group of values that can be expanded/shrunken
typed lists	variable	single-typed	A list of values of the same type
structs	static	static-typed	A group of explicit and ordered set of typed-values
unions	static	mixed-typed	A group of values that shares the same memory space

Table 1. Generic structural data types

  Currently, Tcl only implements the *array* (as a hash table) and *list* data types in a very general way, the problem with this implementation is that it isn't very compiler-friendly in the sense that it is very difficult to infer the real structure of the data being put together. For this reason TyCL includes types to strictly declare the structural organization of data, trying not to loose Tcl's general forms also. Included so far are:

  - Arrays
    Implemented as an ordered collection of values with a fixed quantity of items (completely opposed to Tcl's arrays, which are variable in length and internally unordered)

    In order to use arrays, a definition of its characteristics (length and value’s type) must exist before it can be manipulated, for example:

```
set a array:4                ;# array definition of length = 4
                              # composed by values of any type

set b array:{3 integer}      ;# typed-array definition of length = 3
                              # composed by integer values

set c a:{44 “hello” 0 5.8}   ;# array of length = 4

set d b:{1 2 3}              ;# typed-array of integers of length = 3
                              # with values = {1 2 3}
```

- Lists
  In this case, TyCL tries to follow Tcl's way of handling lists of values, the only difference is that the parser doesn't imply/convert lists automatically, they have to be explicitly declared as such, like:

```
set w list:{a b c {1 2 3} d e}      ;# without the explicit cast
                                     # to a list, 'w' would have
                                     # been handled as string
```

- Groups
  Groups are really typed-valued-lists, they can increase or decrease their size dynamically but only can hold values of the same declared type. As typed-arrays, a definition of a *group* must exist before its values can be accessed of modified.

  Following is an example of a group called: '*a*', defined by items of type *float* and another group defined by the group '*a*' composed by the values: *2.1 , 0 , 5.4*

```
      set a group:float       ;# group definition composed by
                               # values of type: float

      set b a:{2.1 0 5.4}     ;# a group of floats
```

- Structs
  Corresponds to a collection of values with fixed types and order, and that uses an specific amount of memory (just as the C-language define them). Again, an *struct* definition must be present before it can be accessed later on.

  An *struct* definition is composed by a list of item declarations of the form: *TYPE:ITEM_NAME* , where to each item is given a type. So, in order to create an *struct* definition for, say a rectangle, one could write something like:

```
set rectangle struct:{
      float:h           ;# height component, type = float
      float:w           ;# width component, type = float
      integer:color     ;# color, defined as an integer
}

set r1 rectangle:{12.3 9.65 255}    ;# r1 = rectangle struct
```

- Unions
  Unions, also present in the C-language, are basically structs whose values share the same physical space, thus having a footprint defined by the value whose type

uses the most amount of memory.

Their syntax is exactly as *struct*, for example:

```
set data union:{
      integer:i
      float:f
      boolean:b
}
```

The variable: *data* can hold values of type *integer*, *float* or *boolean* when its items: *i*, *f* and *b* are accessed.

- **Functions/Procedures**

Besides the previous data and structural types, functions have their own type also and are treated like any other typed-value referenced directly by a name within a variable or indirectly when they are created right on the spot where they are needed (anonymous functions). Regular functions (named functions) are created/declared as in Tcl using the command: *proc* like:

```
proc add {a b} {
      return [expr $a + $b]
}
```

and anonymous functions are created by the same command but without the name, for example:

```
set add [proc {a b} {
      return [expr $a + $b]
}]
```

both declarations actually do the same thing except that for the second one, the variable '*add*' could be changed later in the program while the first one not, the variable is bound to that function and only its body can change.

The function's parameters (including its return value) could be value-typed also by following the same syntax that *structs* and *unions* have for their members, mixed with the current Tcl's way of declaring parameters. Using the previous function *add*, one could write a complete typed function as:

```
proc add {integer:* integer:a integer:b} {
      return [expr $a + $b]
}
```

The special parameter '*' actually refers to the return type of the function. If by any chance a *return* call tries to pass a value with a different expected type, an exception should be thrown. For parameters that have a default value, Tcl's syntax is used, in this case, for our add function, the complete example would be like:

```
proc add {integer:* {integer:a 0} {integer:b 0}} {
      return [expr $a + $b]
}
```

where the value '*0*' is the default value for parameters *a* and *b*.

Finally, in order to allow principally anonymous functions to call themselves (recursion) the special function name: '*self*' was added to the syntax, which refers to the function being executed (named or anonymous).

- **Function arguments**

Normally, Tcl passes arguments as values (as opposed to by reference) into functions. TyCL continues with this practice but only for the basic types: integer, float, boolean, char and reference, all the rest of types are passed by reference, which means that they can be modified by the function's body when it is executed. The only way to avoid this behavior is to create a copy of the data that is going to be passed by using one of the following commands:

  - copy: creates a new value by copying the first layer of data from the source. As an example:

```
set a list: {1 2 list:{x y z} 3}
set b [copy $a]   ;# b = {1 2 {x y z} 3}
                   # {x y z} refers to the same list for a and b
```

  - clone: creates a new value by copying all data from the source. Again, as an example:

```
set a list: {1 2 list:{x y z} 3}
set b [clone $a]  ;# b = {1 2 {x y z} 3}
                   # {x y z} is a different list from a's
```

- **The object-oriented system**

TyCL includes a native object-oriented system based on prototypes rather than classes. The base type for this kind of data is the type: *object*, which builds the objects as a collection of named-variables implemented as hash-maps within a hash-table. When a

member is needed, first is searched inside the object's hash-table, if the member is not found then is searched inside its prototype (if the object has a prototype) and so long until there is no prototype to follow. When a new member is added, such member is inserted into the object's hash-table without following its prototype.
Objects are composed by members of two kinds: variables and methods. Variables are declared as:

VARIABLE_NAME VALUE ;# for regular variables
TYPE:VARIABLE_NAME VALUE ;# for typed-variables

and methods are declared as:

~METHOD_NAME PARAMETERS BODY

the character: '~' at the beginning of the method's name indicates that the member is going to be a method instead of a variable, also sets the member as function-typed, which means that only its body can be changed later on but not its general description (parameters and return type). There is another way of declaring methods with the difference that methods described in this way can be changed for anything after its declaration, they even can become variables, such syntax is:

METHOD_NAME [proc PARAMETERS BODY] ;# by using an anonymous function

The syntax for creating/declaring an object is like the following example, where an object: *square* is created with 4 members (3 variables and 1 method/function):

```
set square object:{
      color "red"             ;# a regular variable
      float:width 12          ;# this is a typed-variable
      float:height 9          ;# another typed-variable
      ~area {} {              ;# this is a method
            return [expr $my.width * $my.height]
      }
}
```

After the object is created, new members can be added into the object, variables are added by using the commands: *set* or *define,* and methods are created using the command: *proc*, for example:

```
set square.outline false      ;# a new variable: outline was added

proc square.maxside {} {      ;# new method: maxside
      if {$my.width > $my.height} {
            return $my.width
      }
      return $my.height
}
```

In case that the program would like to modify the square object, it could do so by applying the modifications directly into the square object or by creating a new one and having *square* be its prototype, in this last case, the code could be like:

```
set mysquare object:{
      prototype $square

      ~area {{magnification 1.0}} {
            set a [next]
            return [expr $a * $magnification]
      }
}
```

Now we have a new object: *mysquare* whose prototype is the previous: *square* and its method: *area* has been extended to handle a *magnification* value to be multiplied to the calculated total area, which is actually computed by the *area* method owned by the *square* object (called by using the command: *next*)

- **Scope and accessing variables, fields, function and methods**

  Currently, Tcl has few variable's scopes, there is global (variables defined at the root level in the stack and accessed via the *global* command inside functions), local (accessed directly inside functions and namespaces) and namespace's scopes (accessed directly or by providing its namespace path). TyCL supports the global and local scopes but not namespaces yet, however the syntax to reference such variables is a bit different.

  - Local variables
    In this case TyCL follows exactly what Tcl does, it supports local variables inside any functional block (functions and methods) and they are accessed by the variable's names like Tcl does.

  - Global variables
    These variables, as previously mentioned, are variables created at the root of the calling stack, TyCL has its own way to reference this variables from any part of the program without using the command: *global* (as Tcl does) but by using a different way to write their names at the moment where their data is needed, more exactly: by prefixing the name with a dot (.), thus, any access to a variable that starts with a dot means that TyCL has to search for that particular variable in the global scope, for example:

```
set a 4           ;# global variable

proc foo {x} {
      set .a $x   ;# the global 'a' is accessed
}
```

```
foo 99           ;# actually: 'a' is set to the value: 99
```

- Object's members

  As objects have their own set of members (variables and methods), they can be accessed by joining the object's name and the variable's name with a dot (.) and by following the previous rules if the object is a global object or a local one. If the access to a member is present inside one of its own methods, a new prefix: 'my' must be used to indicate that such member is present inside itself.

  So, as an example, the following code creates an object '*a'* with a method *count* that shows how many times this method have been called:

  ```
  set a object:{
        x 0
        ~count {} {
              incr my.x 1                   ;# access to a.x
              puts "call number: $my.x"     ;# access to a.x
        }
  }

  a.count     ;# It should print: 'call number: 1'
  a.count     ;# It should print: 'call number: 2'
  ```

- Struct's and union's fields

  Just like objects, the fields of any *struct* or *union* are referenced by following the same previous rules, thus for an *struct,* a short example could be like:

  ```
  set point struct:{
        float:x
        float:y
  }

  set point.x 88          ;# access to point's field: x
  set point.y 314         ;# access to point's field: w
  ```

- **Indexes and Ranges**

A new syntax was created to handle indexes an ranges for data-types that support them, which account for all except *objects* and *structs.* Their syntax is:

VARIABLE_NAME(INDEX) ;# for indexes
VARIABLE_NAME(INITIAL_INDEX .. FINAL_INDEX) ;# for ranges

their functionality is conditioned by the data-type of the operated variable as:

- Integers: gets or sets particular bits.
- Floats: gets or sets particular bits.
- Boolean: gets or sets bits, but the complete values is stored always as 0 or 1.
- Chars: gets or sets particular bits.
- Strings: gets or sets particular characters.
- Arrays: gets or sets particular values.
- Lists: gets or sets particular values.
- Groups: gets or sets particular values.

Additionally to its general use, indexes can also be used to insert, append or remove items within a variable's content (although not all data-types support this behavior) by using the following syntax:

*set VARIABLE_NAME(*INDEX) VALUE* ;# inserts the value before index
set VARIABLE_NAME(INDEX) VALUE* ;# inserts the value after the index

if the index is -1, it references the las item in the content, thus an append would be written as:

set VARIABLE_NAME(-1) VALUE* ;# this is an append

if the value is the *null* value, the operation is actually a remove operation.

The WordCode-Generator

As previously mentioned, the WC-Generator (WCG) takes the created TIR tree by the lexical-parser and converts it in a series of binary codes composed by chunks 32-bits wide each which contains TyCL's virtual machine opcodes, indexes to the literals table and inline values to name a few. The WordCodes are able to handle 0, 1 or 2 8-16-32-64-bit operands depending on the opcode's needs, thus the minimum size of a WordCode is 4-bytes (0,1 or 2 8-bit operands) and its maximum size is 20-bytes (2 64-bit operands). The format for each WordCode varies between opcodes but a generalization can be made as follows:

FULL OPCODE (32-bit)				**OPERAND** (32-bit)	**OPERAND** (32-bit)	**OPERAND** (32-bit)	**OPERAND** (32-bit)
OPCODE (12-bit)	OSIZE (4-bit)	L-OP (8-bit)	R-OP (8-bit)	optional	optional	optional	optional

Table 2. General description of a WordCode

Basically, a WordCode (WC) is composed of a full-opcode followed by 0, 1, 2, 3 or 4 32-bit operand values, depending on the actual opcode located at the highest 12-bits of the full-opcode and its 4-bit OSIZE field, which is composed by two 2-bit values that indicates each operand's current size like this:

L-OSIZE or R-OSIZE (2-bit)	**Operand's size (L-OSIZE referes to the left operand and R-OSIZE to the right operand)**
0 0	8-bits
0 1	16-bits
1 0	32-bits
1 1	64-bits

Table 3. Operand's size descriptor values

As it would be expected, there are some special WCs intended for jumps (including conditional jumps) that may use the OSIZE in conjunction with L-OP and R-OP to create a single 20-bit value.

The Virtual Machine (VM)

Like any other virtual machine, it walks through the WCs, decoding and executing each instruction and manipulating the variables and values, which at the moment are kept in memory with memory manager and a garbage-collector based on reference-counting[6], using the data structures described in figure 2.

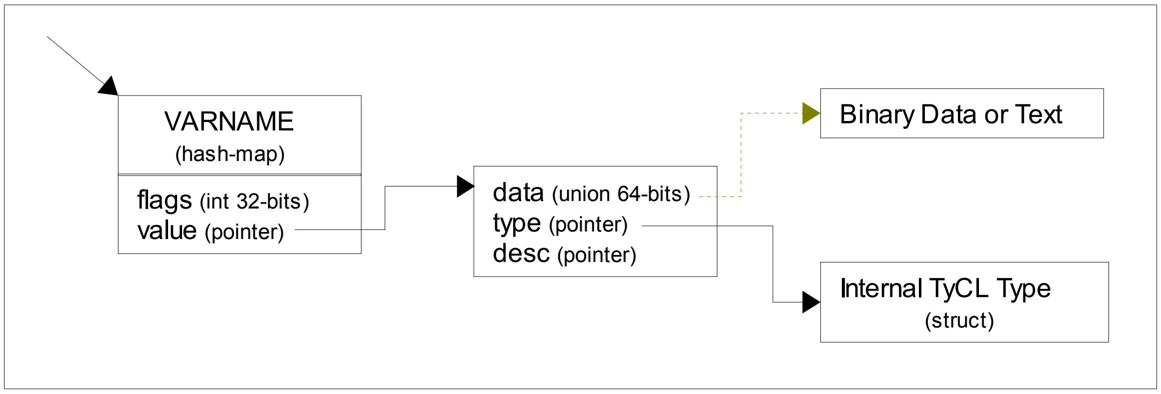

Figure 2. General diagram of the data structures for variables and values.

The Machine-Code-Generator

This stage, at the moment, just translates the WordCodes into machine-codes using a series of previously defined tables, almost like writing in native-code what the VM needs to do in order to process a particular WC. This solution is only temporal and was developed in this way in order to minimize the developing time of the initial iterations of the compiler.

Conclusion

Even though, writing a compiler for a very dynamic language like Tcl is a very difficult and time consuming task, its benefits surely pay for all the pain that one have to suffer in the process. The inclusion of types as first class-citizens into the syntax and the language in general, at the end, not only should help the compiler produce better results but helps the programmer also by providing a way to reduce bugs due to data-type collisions or invalid-values.

By having TyCL written in itself, it minimizes the dependencies on other components (like external compilers or linkers), avoids the need to handle two different languages (one for the coding of the compiler itself and the language that the compiler process) and allows to reuse any improvement made in the compiler on itself and on the applications it builds.

References

[1] Miguel Sofer, "NRE: the non-recursive engine in Tcl8.6", 15th Annual Tcl/Tk Conference, October 16, 2008
[2] Alan Kay, "The Computer Revolution hasn't happend yet", "12th Annual ACM SIGPLAN Conference on Object-Oriented Proramming Systems, Languages and Applications", October 5, 1997
[3] http://en.wikipedia.org/wiki/Bootstrapping_%28compilers%29
[4] Aho A., Lam M., Sethi R., Ullman J., "Compilers Principles, Techniques, & Tools", Pearson, Addison Wesley, 2nd Edition, 1987
[5] Steven Muchnick, "Advanced Compiler Design and Implementation", Morgan Kaufmann Publishers, 1997
[6] http://en.wikipedia.org/wiki/Reference_counting