# TyCL

**An interpreter/compiler of a typed language implementation of Tcl/Tk**

**Andres Buss**

**Otlet Technologies**

# Overview

TyCL short description

The architecture

The type system

The parser and TyCL's syntax

The WordCode Generator (very briefly)

The MachineCode Geneator (very briefly)

Status and conclusions

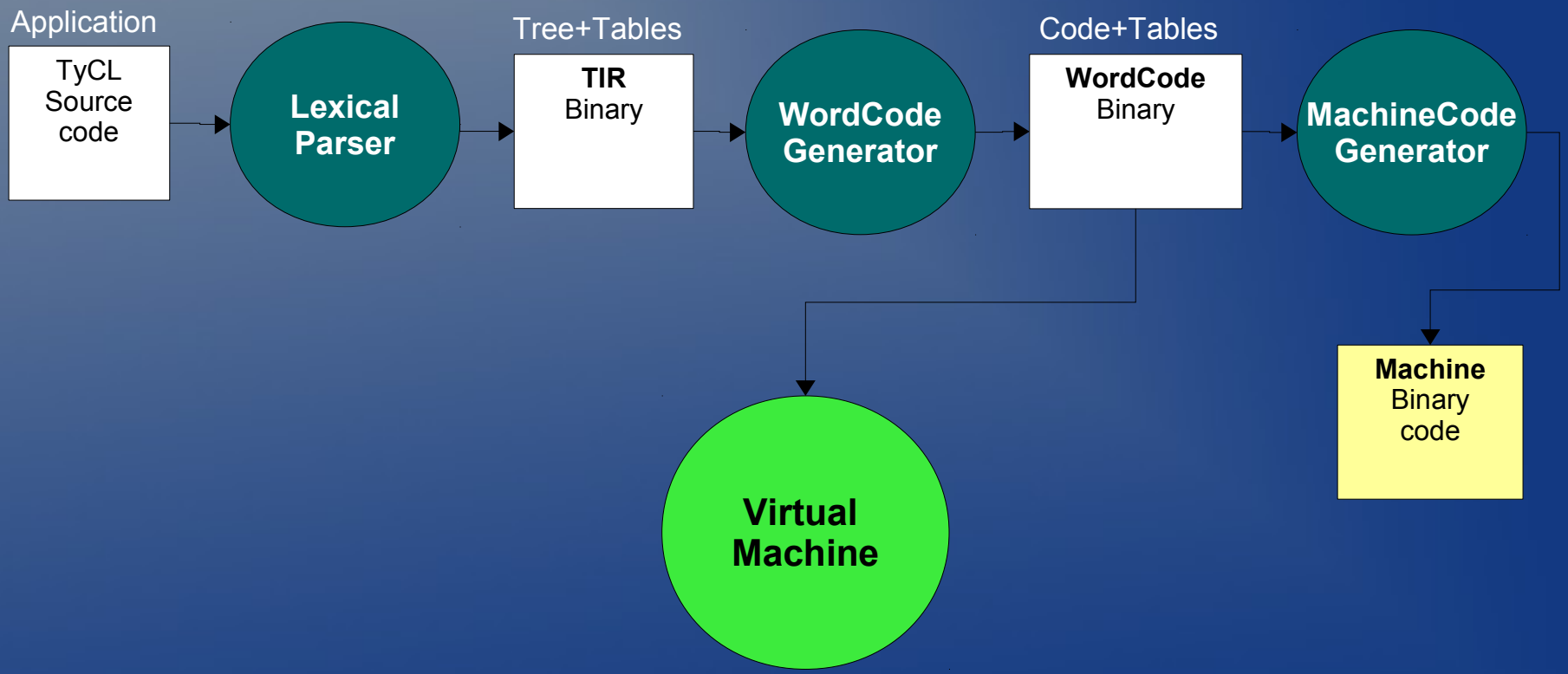
# Description

What is TyCL

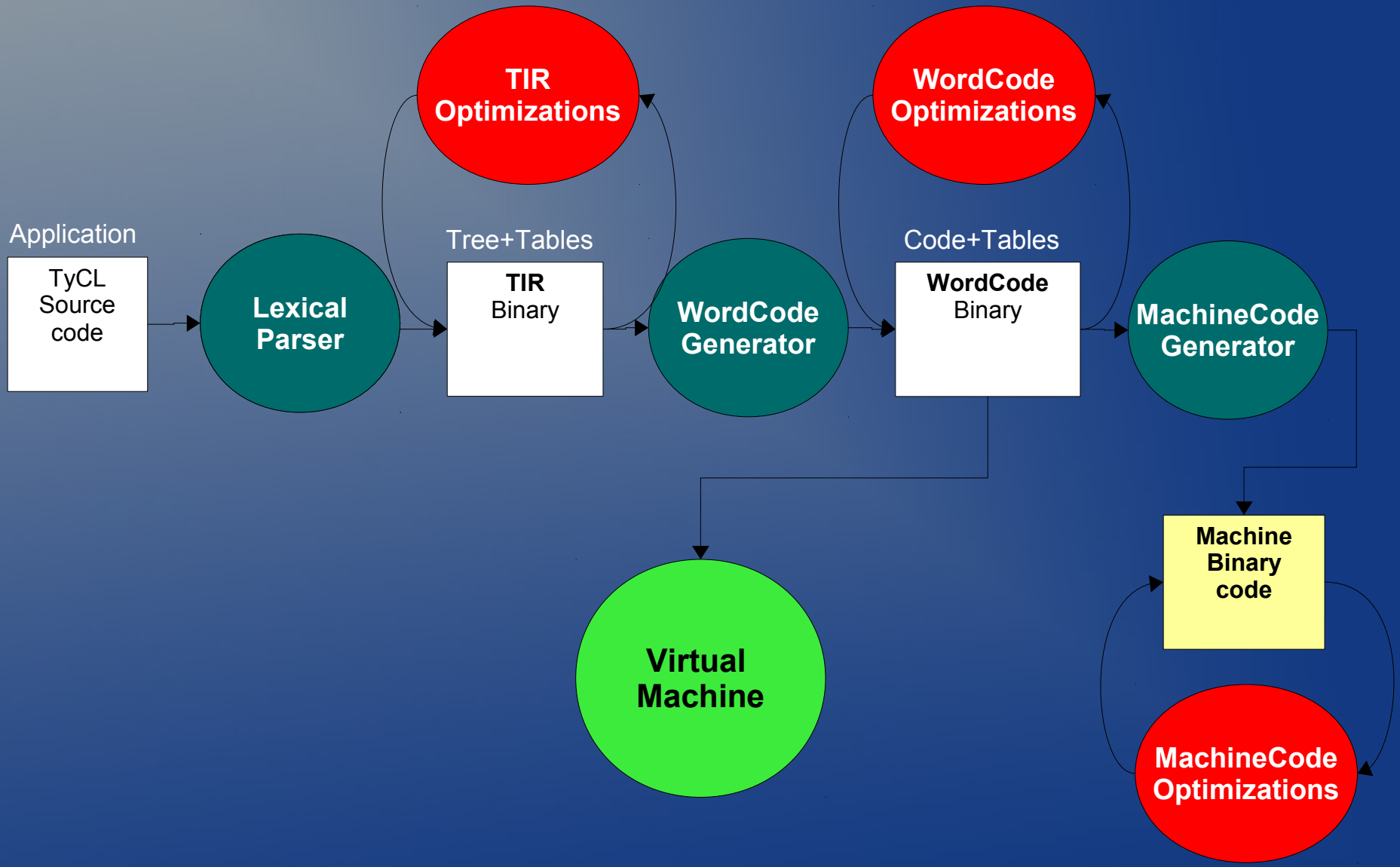
Why TyCL?

Main goals

# Architecture



# Architecture



# The Type system

Needed for better description of data to the compiler

Optional

Should bring better results in performance and memory usage

# The lexical-parser

Reads the source code and:

- Detects syntax errors (by file, line and pos.)
- Generates the TIR

TIR = TyCL-Intermediate-Representation

Can process “~~mostly~~” all Tcl's syntax

# TyCL's Syntax

## What TyCL understand that Tcl don't

The type system

Every value has a type (nothing new here)

Basic data-types:

Integer, float, boolean, char, string

reference

Types can be explicitly declared by casting values

# TyCL's Syntax - Casting

*TYPE:VALUE*

```
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:”false”   ;# boolean = false
set f boolean:45           ;# boolean = true
set g 3.1415               ;# auto-detected float = 3.1415
set h float:3.4            ;# float = 3.4
set i float:{56.8576}   ;# float = 56.8576
set j char:w               ;# char = 'w'
set k char:67              ;# char = 'C'
set l hello             ;# default-detected string = 'hello'
set m “hello world”  ;# auto-detected string = 'hello world'
set n string:6.28    ;# string = '6.28'
```

# TyCL's Syntax – Typed Variables

Regular variables:

```
set VARIABLE_NAME VALUE
set VARIABLE_NAME TYPE:VALUE
```

Typed variables:

```
define VARIABLE_NAME VALUE
define VARIABLE_NAME TYPE:VALUE
```

# TyCL's References

allows a variable to reference another one in a transparent way

```
let VARIABLE_NAME
let VARIABLE_NAME VARIABLE_NAME
```

```
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
```

# TyCL's Structural types

For collections of values

Values of different type

Values of the same type

Collections with static size

Collections with fixed size

Collections with variable/dynamic size

# TyCL's Arrays

Implemented as an ordered collection of values with a fixed quantity of items

```
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}
```

# TyCL's Lists

As Tcl's lists. The only difference is that the parser doesn't imply/convert lists automatically, they have to be explicitly declared

```
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
```

# TyCL's Groups

are really typed-valued-lists, they can increase or decrease their size dynamically but only can hold values of the same declared type

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

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

# TyCL's Structs

a collection of values with fixed types and order, and that uses an specific amount of memory (just as the C-language define them)

*TYPE:ITEM_NAME*

```
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
```

# TyCL's Unions

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

```
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

# TyCL's Functions

Are like Tcl normal functions/procedures

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

But, there could be anonymous functions also:

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

# TyCL's Functions with typed parameters

```
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.

# TyCL's Functions with typed parameters and default values

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

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).

# TyCL's Functions arguments

only the basic types: integer, float,  boolean, char and reference are passed as values, all the rest of types are passed by reference

The copy command:

```
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
```

# TyCL's Functions arguments (cont.)

only the basic types: integer, float,  boolean, char and reference are passed as values, all the rest of types are passed by reference

The clone command:

```
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
```

# TyCL's 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***.

objects are a collection of named-variables implemented as hash-maps within a hash-table.

# TyCL's Object-Oriented System (cont)

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/modified, such member is inserted into the object's hash-table without following its prototype

# TyCL's Object-Oriented System (cont)

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
```

# TyCL's Object-Oriented System (cont)

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

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

# TyCL's Object-Oriented System (cont)

An example:

```
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]
   }
}
```

# TyCL's Object-Oriented System (cont)

Another way to populate objects:

```
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
}
```

# TyCL's Object-Oriented System (cont)

Creating a new object from another:

```
set mysquare object:{
   prototype $square
   ~area {{magnification 1.0}} {
      set a [next]
      return [expr $a * $magnification]
   }
}
```

# TyCL's Scopes & data access

TyCL supports the global and local scopes but not namespaces yet, however the syntax to reference such variables is a bit different

**Local variables:**

They are just as Tcl's ones

Accessed inside execution blocks only

# TyCL's Scopes data & access (cont)

**Global variables:**

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)

by prefixing the name with a dot (.)

# TyCL's Scopes & data access (cont)

**Global variables (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
```

# TyCL's Scopes & data access (cont)

**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

# TyCL's Scopes & data access (cont)

**Object's members (example):**

```
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'
```

# TyCL's Scopes & data access (cont)

**Struct's and Union's fields:**

Just like objects, the fields of any *struct* or *union* are referenced by following the same previous rules

```
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
```

# TyCL's 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*

```
# for indexes
VARIABLE_NAME(INDEX)


# for ranges
VARIABLE_NAME(INITIAL_INDEX .. FINAL_INDEX)
```

# TyCL's Indexes and Ranges (cont.)

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.

# TyCL's Indexes and Ranges (cont.)

Using indexes to insert, append replace data

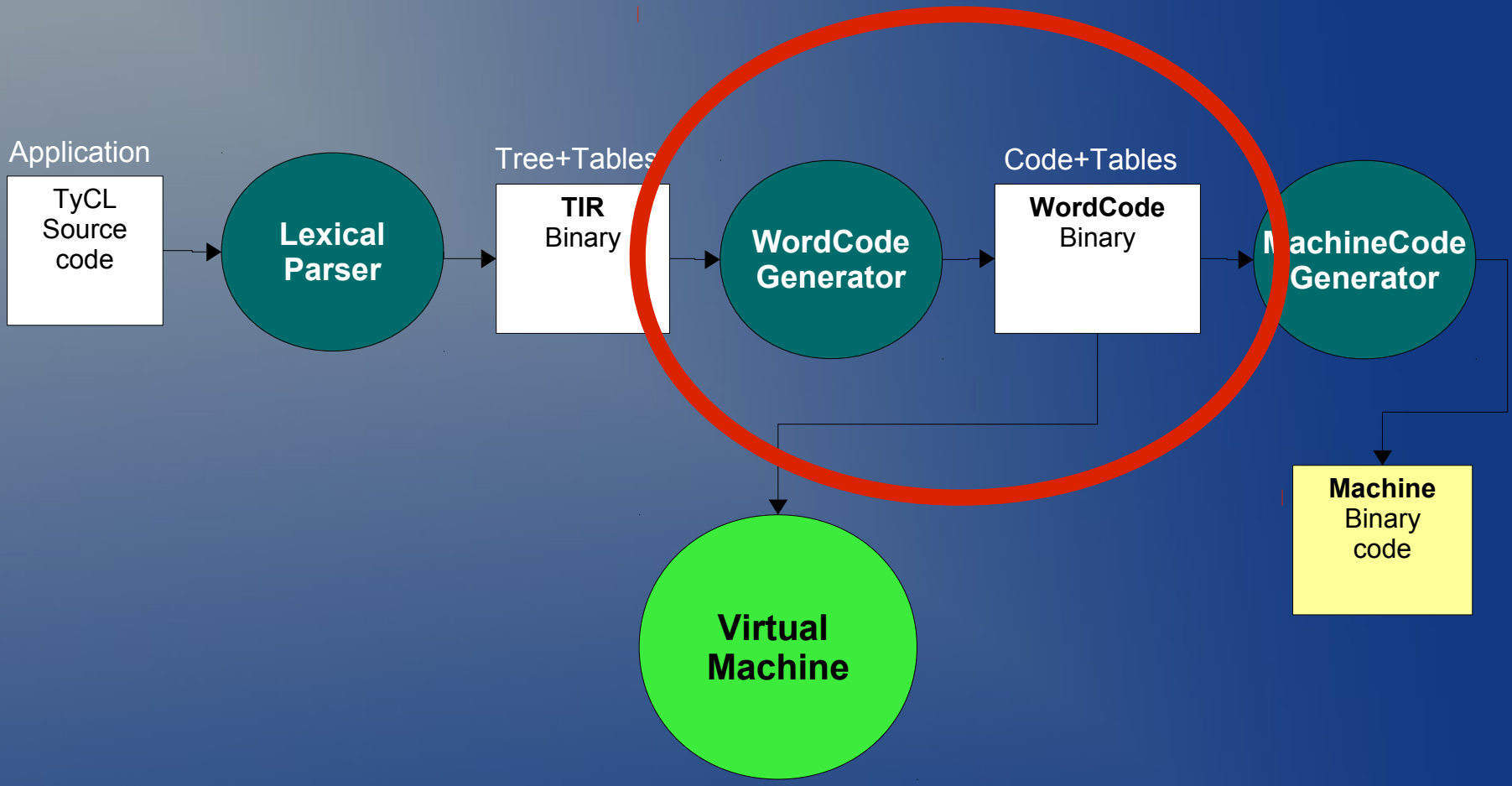
```
# inserts the value before index
set VARIABLE_NAME(*INDEX) VALUE

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

if the index is -1, it references the last 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



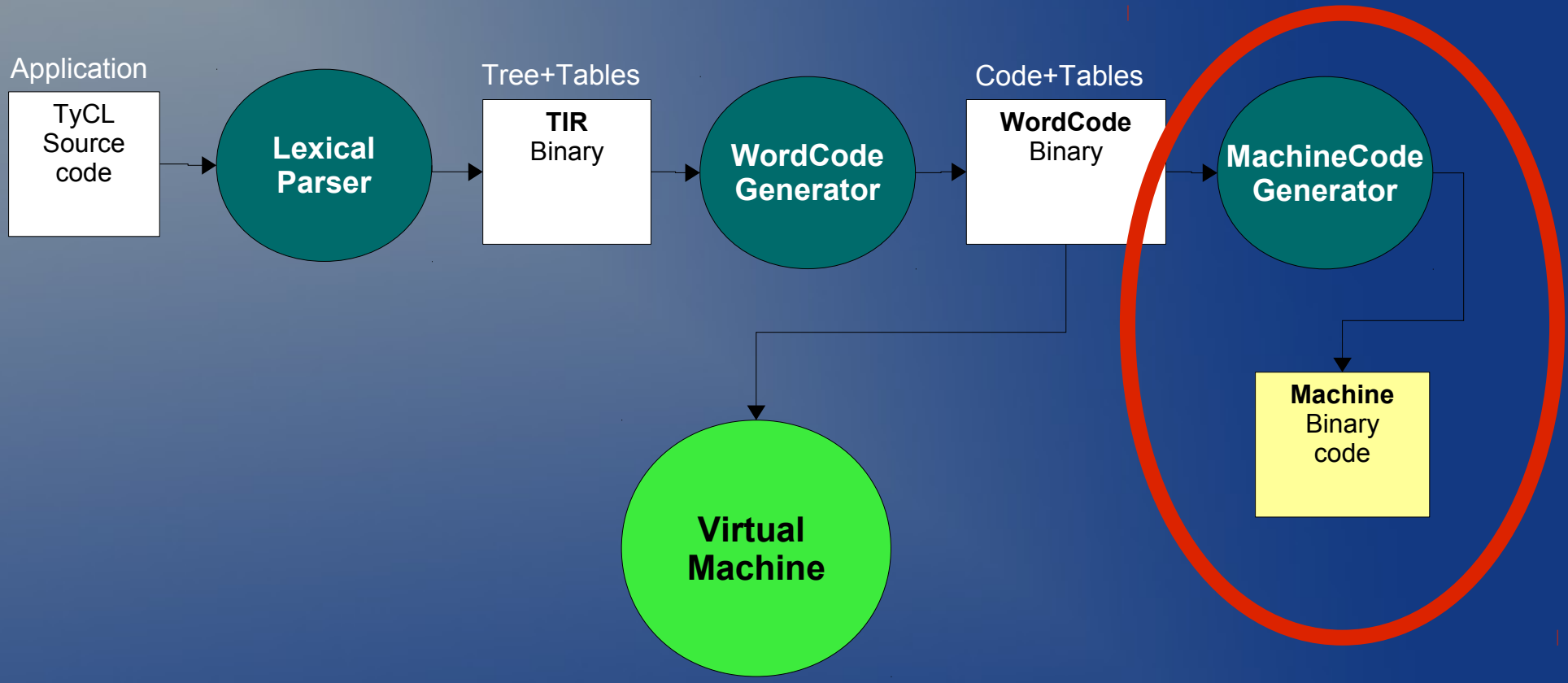
# The WordCode Generator

**WordCode format:**

**32-bit binary chunks**

| **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 | | | | |

# The MachineCode Generator



# Status and conclusions

The project is just starting

A lot of work still have to be done

It only “~~works~~” on Linux X86

It could be an interesting platform to play with

# That's it...

Any help is welcome!!!