bI dymaniac language system

© Dmitry Ponyatov <dponyatov@gmail.com>

January 15, 2016

Contents

Intro

Any program must have scripting ability for configs and user extensions. bI system provides universal script engine for bI language dialect and dynamic data types C^{++} class tree for internal use in generated program. I was impressed by $SmallTalk$ system ideology, bI system follows this way to gui-powered interactive system for translators design, symbolic computations and CAD/CAM/EDA environment.

Goals

- metaprogramming, computer language design and translator development
- symbolic and numeric computations
- clustering and cloud computing
- complex engineering systems design
- statical translation to $C^{++}/Java$ for multiplatform software development ($\boxplus Windows/Linux/Android$)

Applications

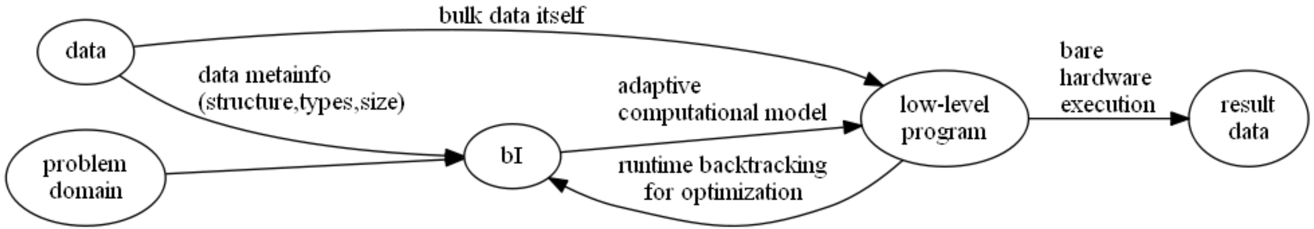
- universal language for configs and parser for computing programs input data presented in text format
- text data and program sources processing

- fast GUI programming for tiny helper programs
- universal template language:
  - files generation based on project templates
  - multiplatform high-level software development
  - config files generation and control in clustering systems

Be Warned

0.0.1 bI not intended for data crunching itself

bI not intended for data crunching itself — it's tool for hand-cranked compiling and program transformations. bI core supports `<num:`1.6.5`>` data type for floating point numbers, but avoid use of bI core for numerical computation. Right way to use bI — construct low-level program which will crunch your data using power of bI metaprogramming.



LLVM framework and JIT libraries looks very interesting for dynamic compilation — this magic can conjure some speedup of bI core[1] itself, and incredible performance of mutable runtime-generated machine code for data crunching.

[1] it's high-level part realized in bI language, and $bI/next$ generation described via core metamodel

0.0.2 There is no memory management at all

Current version of bI core have no any memory management: there is no garbage collector, all created objects will be stay in memory until system crash on memory overflow.

This way was chosen for simplicity. It is sufficient for tiny batch runs and interactive work with "failure and restart from snapshot" hints, but this makes continues or large data crunching impossible.

0.0.3 You must have some skills in compiler design and functional programming

bI system is syntax analyzer and translator framework by design, and user must have some skills in compiler design and functional programming. You must read DragonBook [3], SICP [5] and Harrison/Field [6] before you dig in hedgehog den.

Installation

GitHub: https://github.com/ponyatov/Y

dev branch: https://github.com/ponyatov/Y/tree/dev/

```
git clone -o gh https://github.com/ponyatov/Y/tree/master/ bI_stable
cd bI_stable
```

bI system provided as source-only, and requires some development tools installed:

- host: ⊞$Windows$

  git-scm git client https://git-scm.com/downloads

  MinGW GNU compiler toolchain http://www.mingw.org/download/installer?

    - **g++** C^{++} compiler
    - **flex** lexer generator
    - **bison** parser generator

  [$(g)Vim$] text editor ftp://ftp.vim.org/pub/vim/pc/gvim74.exe

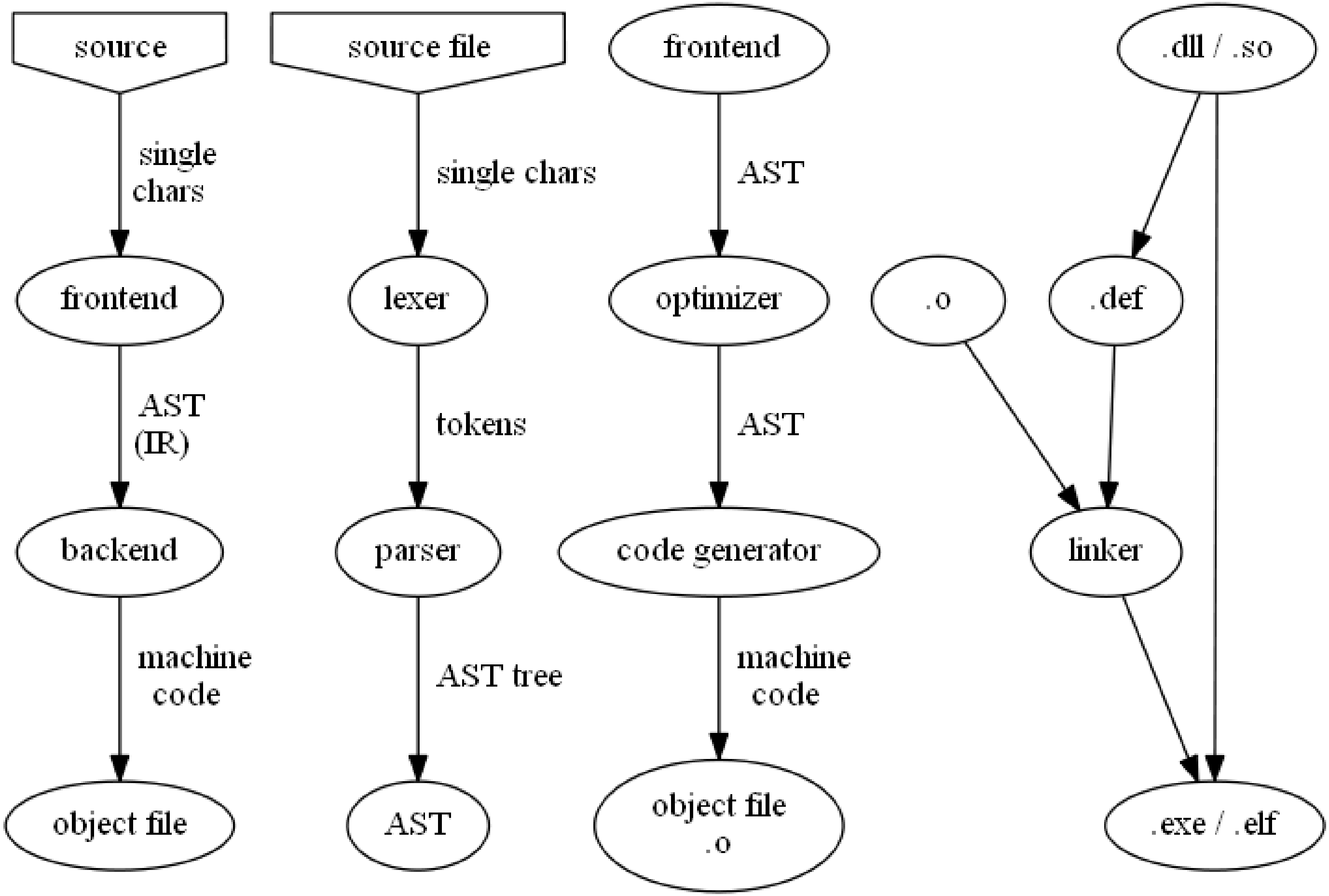
  ```
  mingw32-make EXE=.exe RES=res.res
  ```

- host: $Linux$, powered with LLVM dynamic compilation

  ```
  apt install git make g++ flex bison llvm-3.5
  ```

  ```
  make EXE= RES= LLVER=3.5
  ```

Compiler structure



Chapter 1

Core system

1.1 Files

ypp.ypp	flex	parser 1.1.2
lpp.lpp	bison	lexer 1.1.1
hpp.hpp	C^{++}	headers 1.1.3
cpp.cpp	C^{++}	core 1.1.4
Makefile	make	build script 1.1.5
rc.rc	windres	win32 resource description
bat.bat	$(g)Vim$	win32 start helper
filetype.vim	$(g)Vim$	`.bI|.blog` file type processing
syntax.vim	$(g)Vim$	syntax coloring
doc/	LaTeX	
doc/Makefile		1.1.5
doc/bI.pdf		manual

1.1.1 Lexer

Lexer uses **flex** generator, produces **lex.yy.c**.

All defines moved to **hpp.hpp**, lexer header includes buffer for string parsing.

lpp.lpp

```
%{
#include "hpp.hpp"
string StringLexBuffer;                 /* string parsing buffer */
void incLude(Sym*inc) {                 /* .inc processing */
    if (!(yyin = fopen(inc->val.c_str(),"r"))) yyerror("");     // open
    yypush_buffer_state(yy_create_buffer(yyin,YY_BUF_SIZE));    // push to lexer
}
%}
```

Options disables `yywrap()` function usage and enables line number autocount for error reporting.

lpp.lpp

```
%option noyywrap
%option yylineno
```

Rules section described part by part in scalar types 1.6 and operators **??** manual sections.

lpp.lpp

```
%%
...%%
...
```

Unused chars will be dropped by this rules at end of lexer:

lpp.lpp

```
^\.inc[ \t]+[^\n]+   { yylval.o = new Directive(yytext);     /* .inc lude */
^\.[a-z]+[^\n]*      TOC(Directive,DIR)                       /* .directive */
[ \t\r\n]+           {}                               /* drop spaces */
.                    {}                               /* drop undetected chars */
```

Lexer C^{++} API includes this objects: `TOC()` macro used in lexer rules, creates

hpp.hpp

```
// == lexer interface ==
extern int yylex();                         // parse next token
extern int yylineno;                        // current source line
extern char* yytext;                        // found token text
#define TOC(C,X) { yylval.o = new C(yytext); return X; }// token macro used in lexer
```

1.1.2 Parser

Core parser uses **bison** for **ypp.tab.cpp**, **ypp.tab.hpp**

Parser header looks like lexer header, all defines done in **hpp.hpp**.

ypp.ypp

```
%{
#include "hpp.hpp"
%}
```

hpp.hpp

```
// == parser interface ==
extern int yyparse();                       // run parser
extern void yyerror(std::string);           // error callback
```

1.1.3 Headers

Header file contents wrapped by include-once preprocessor hint:

hpp.hpp

```
#ifndef _H_bI
#define _H_bI
#endif // _H_bI
```

Some metainfo constants defined, including `-DMODULE=$(CURDIR)` defined in **Makefile**:

hpp.hpp

```
#define AUTHOR "(c) Dmitry Ponyatov <dponyatov@gmail.com>, all rights reserved"
#define LICENSE "http://www.gnu.org/copyleft/lesser.html"
#define GITHUB "https://github.com/ponyatov/Y/tree/dev"
#define AUTOGEN "/***** DO NOT EDIT: this file was autogened by bI *****/"
#define LOGO "logo64x64"
#define LISPLOGO "warning64x64"
```

Standard C^{++} includes used in core:

hpp.hpp

```
                                            // == std.includes ==
#include <iostream>
```

```
#include <sstream>
#include <cstdlib>
#include <vector>
#include <map>
using namespace std;
```

mingw32.hpp: win32/MinGW

Some OS/platform specifics headers selected into separate files,

mingw32.hpp

```
#ifndef _H_MINGW32
#define _H_MINGW32

#include <direct.h>
namespace win32 {
#include <windows.h>
}

#endif // _H_MINGW32
```

1.1.4 C^{++} core

C^{++} code described part by part over this manual in every symbolic type section.

cpp.cpp

```
#include "hpp.hpp"
```

Error callback function: it will be called from parser on error. YYERR macro used for doubling error message: to stdout redirected to .blog, and stderr goes to **make** output log[1].

cpp.cpp

```
#define YYERR "\n\n"<<yylineno<<":"<<msg<<"["<<yytext<<"]\n\n"
void yyerror(string msg) { cout<<YYERR; cerr<<YYERR; exit(-1); }
```

main() function: call global environment setup and parser:

cpp.cpp

```
int main() { env_init(); return yyparse(); }          // == main() ==
```

mingw32.cpp: win32/MinGW

OS/platform specifics C^{++} code selected into separate files,

mingw32.cpp

```
#include "hpp.hpp"

Window::Window(Sym*o):Sym("window",o->val) {}

void Window::show() { par["show"]=nil; }
```

1.1.5 Build script

Project builds with command [mingw32-]make [vars]. Vars can be:

[1] and IDE report

variable	win32	unix	
`EXE`	`.exe`		executable file extension, empty if Linux/UNIX
`RES`	`res.res`		resource file name (win32 only)
`TAIL`	`-n17`	`-n7`	number of `.blog` lines will be printed on `make exec` build
`LLVER`		`3.5`	LLVM version if used

`MODULE` variable sets name for current module. It was set to bI, but can use current dir name as module name.

Makefile

```
MODULE = $(notdir $(CURDIR))
MODULE = bI
```

`exec` target build bI system core and runs high-level system build from **bI.bI** master source:

Makefile

```
.PHONY: exec
exec: ./$(MODULE)$(EXE) $(MODULE).bI
	./$(MODULE)$(EXE) < $(MODULE).bI > $(MODULE).blog && tail $(TAIL) $(MODULE).blog
```

`make clean` removes all temporary and produced files, makes all project clean:

Makefile

```
.PHONY: clean
clean:
	rm -rf ./$(MODULE)$(EXE) *.*log ypp.tab.?pp lex.yy.c $(RES)
```

`C\H` contains files will be compiled by `CXX` C^{++} compiler into interpreter executable:

Makefile

```
C = cpp.cpp $(OS).cpp ypp.tab.cpp lex.yy.c
H = hpp.hpp $(OS).hpp ypp.tab.hpp
```

C^{++} compiler run:

Makefile

```
OS = $(shell $(CXX) -dumpmachine)
CXXFLAGS += -I. -std=gnu++11 -DMODULE=\"$(MODULE)\"
./$(MODULE)$(EXE):␣$(C)␣$(H)␣$(RES)␣Makefile
␣␣␣␣$(CXX)␣$(CXXFLAGS)␣-o␣$@␣$(C)␣$(RES)
```

bison parser generator run:

Makefile

```
ypp.tab.cpp: ypp.ypp
	bison $<
```

flex lexer generator run:

Makefile

```
lex.yy.c: lpp.lpp
	flex $<
```

win32 resource compiler run:

Makefile

```
res.res: rc.rc
	windres $< -O coff -o $@
```

1.2 Sym: Abstract Symbolic Type

bI language based on operations on Abstract [Sym]bolic Type: it's close to classical Abstract Syntax Tree elements, and uses same acronim. For dymanic languages Sym much complicated comparing to *Lisp* ceils/lists, and scalar primitive types[2], but it was selected considering primary *bI* area: computer language processing, where annotated AST trees is basic data type.

`class:Sym`		abstract symbolic type
	`string:tag`	type, class tag
	`string:val`	value
`constructors:`	`Sym(string,string)`	`<T:V>` constructor
	`Sym(string)`	token constructor
	`Sym(Sym)`	copy constructor
`nest[]ed:`	`List<Sym>:nest[]`	nested elements
	`fn:push(Sym)`	add nested
`par{}ameters:`	`Dict<string,Sym>:par[]`	parameters dict (string-keyed list)
	`fn:setpar(Sym)`	add/set parameter
`dump:`	`fn:dump(int)->string`	recursive `dump(+1)` tree in text form (with depth padding)
	`fn:tagval()->string`	dump `<T:V>` header only
	`fn:pad(int)->string`	return padding string: n tabs
`operators:`	`fn:eval()->Sym`	compute/evaluate object
	`op:@(Sym)->Sym`	`A @ B` apply
	`op:=(Sym)->Sym`	`A = B` equal

hpp.hpp

[2] numbers, strings

Using virtual base class Sym{} allows to use RTTI and process inherited class instances using pointers to base class, first of all it allows to use storage collections vector<Sym*> and map<string,Sym*> for any objects[3].

1.3 Writers

Writer — function writes argument to bI log (.blog):

hpp.hpp

```
extern void W(Sym*);                                    // == writers ==
extern void W(string);
```

cpp.cpp

```
void W(Sym*o)          { cout<<o->dump(); }             // == writers ==
void W(string s)       { cout<<s; }
```

1.4 Global environment

hpp.hpp

```
extern map<string,Sym*> env;                            // == global environment ==
extern void env_init();                                 // init env[] on startup
```

[3] instances of inherited classes

cpp.cpp

```
int main() { env_init(); return yyparse(); }          // == main() ==
    Sym*E = env[val]; if (E) return E;                // lookup in glob.env[]
Sym* Sym::eq(Sym*o)     { env[val]=o; return o; }
map<string,Sym*> env;                                 // == environment ==
void env_init() {                                     // init on startup
    env["nil"]=nil;
    env["MODULE"]   = new Str(MODULE);                // module name (CFLAGS -DMODULE)
    env["AUTHOR"]   = new Str(AUTHOR);                // author (c)
    env["LICENSE"]  = new Str(LICENSE);               // license
    env["GITHUB"]   = new Str(GITHUB);                // github home
    env["AUTOGEN"]  = new Str(AUTOGEN);               // autogenerated code signature
    env["LOGO"]     = new Str(LOGO);                  // bl logo (w/o file extension)
    env["LISPLOGO"] = new Str(LISPLOGO);              // Lisp Warning logo
    env["window"] = new Fn("window",window);
```

1.5 Comments

1.5.1 Line comment

bl.bl

```
```

lpp.lpp

```
#[^\|][^\n]*\n      {}                              /* line comment */
```

1.5.2 Block comment

Current version have undetected problems with block comments: on multiline block comments lexer hangs until file end, ignoring all source and causing strange syntax errors.

bl.bl

```
```

lpp.lpp

```
                                          /* lexer state: #| block comment |# */
%x lexcomment
#\|                  {BEGIN(lexcomment);}                  /* block comment*/
<lexcomment>\|#      {BEGIN(INITIAL);}
<lexcomment>\n       {}
<lexcomment>.        {}
```

1.6 Scalar types

1.6.1 str: string

1.6.2 int: integer

1.6.3 hex: machine hex

1.6.4 bin: machine binary

1.6.5 num: floating point number

1.7 Composites

1.7.1 List

hpp.hpp

cpp.cpp

1.7.2 Pair

hpp.hpp

cpp.cpp

1.7.3 Vector

hpp.hpp

cpp.cpp

1.7.4 Tuple

hpp.hpp

cpp.cpp

1.8 Functionals

1.8.1 Operator

All operators described in 1.9

1.8.2 Function

1.8.3 Lambda

1.9 Operators

1.9.1 = assignment

1.9.2 @ apply

1.9.3 . index

1.9.4 + add

1.9.5 - sub

1.9.6 * mul

1.9.7 / div

1.9.8 ˆ pow

Chapter 2

GUI subsystem

2.1 Display

2.2 Window

2.3 Group: widget grouping

2.3.1 Tiler window manager

2.3.2 Tabber

2.3.3 Grid

2.3.4 FreeForm

2.4 Form controllers

Chapter 3

Data storage

3.1 Volume management

3.1.1 plug/unplug/status

3.2 RDBMS interface

3.2.1 Generic interface

3.2.2 SQLite

3.2.3 MySQL

3.2.4 Postgres

3.2.5 Cursor

Chapter 4

Network

Chapter 5

Math engine

5.1 Math types

5.1.1 Complex number

5.1.2 Matrix

5.2 Symbolic algebra

5.3 Numeric methods

5.4 Signal processing

Chapter 6

CAD/CAM

6.1 CAD base

6.1.1 Primitives

6.1.2 Data interchange

STEP

IGES

STL

DXF

6.1.3 Parametric solver

6.1.4 Assembly

Chapter 7

Dynamic syntax analisys

7.1 Lexer

7.2 Parser

Chapter 8

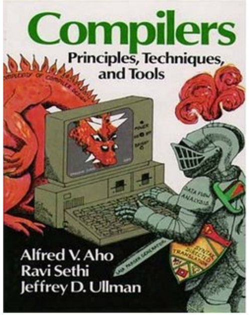
LLVM integration

Bibliography

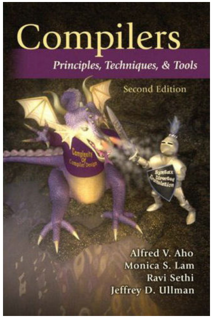
Dragon Book

[1] **Dragon Book @ Stanford.edu**
Some lection sets on computer language compilers in free e-books

[2] Green Dragon Book'77
Alfred V. Aho, Jeffrey D. Ulman
Principles of Compiler Design
Addison-Wesley, ISBN 0-201-00022-9, 1977



[3] classical Red Dragon Book
Alfred V. Aho, Ravi Sethi, Jeffrey D. Ulman
Compilers: Principles, Techniques, and Tools (2nd edition)

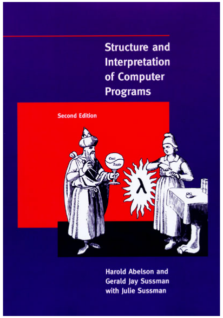


[4] Purple Dragon Book
Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ulman
Compilers: Principles, Techniques, and Tools (2nd edition)
Addison-Wesley, 2006

- directed translation
- new data flow analyses
- parallel machines

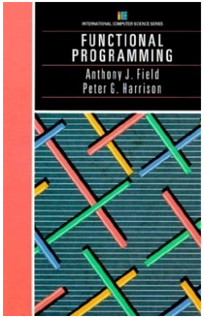
- JIT compiling
- garbage collection
- new case studies

SICP



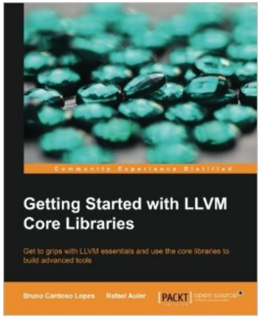
[5] SICP
Harold Abelson, Gerald Jay Sussman, Julie Sussman
Structure and Interpretation of Computer Programs second edition
© 1996 by The Massachusetts Institute of Technology

Functional programming

[6] 

Peter G. Harrison, Anthony J. Field
Functional Programming

LLVM

[7] 
Community Experience Distilled
Getting Started with LLVM Core Libraries
Get to grips with LLVM essentials and use the core libraries to build advanced tools
Bruno Cardoso Lopes
Rafael Auler
PACKT open source

Bruno Cardoso Lopes, Rafael Auler
Getting Started with LLVM Core Libraries

9 1/2 books

[8] The Top $9\frac{1}{2}$ In a Hackers Bookshelf
by Jess Johnson in Books & Tools

[9] Fredrick P. Brooks
The Mythical Man Month: Essays on Software Engineering
Anniversary Edition

[10] Brian W. Kernighan, Dennis M. Ritchie
The ANSI C Programming Language, Second Edition
Prentice Hall, AT&T, 1988