bI dymaniac language system

© Dmitry Ponyatov <dponyatov@gmail.com>

January 6, 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

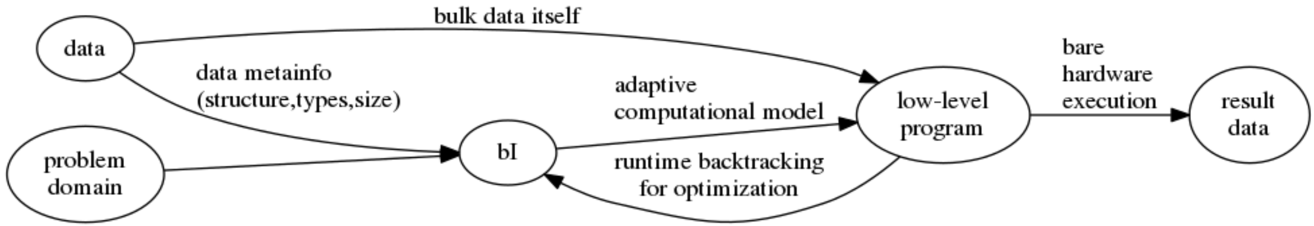
- universal language for configs and parser for computing programs input data presented in text format
- text data and program sources processing

- 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:**??**> 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: $\boxplus Windows$

  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`

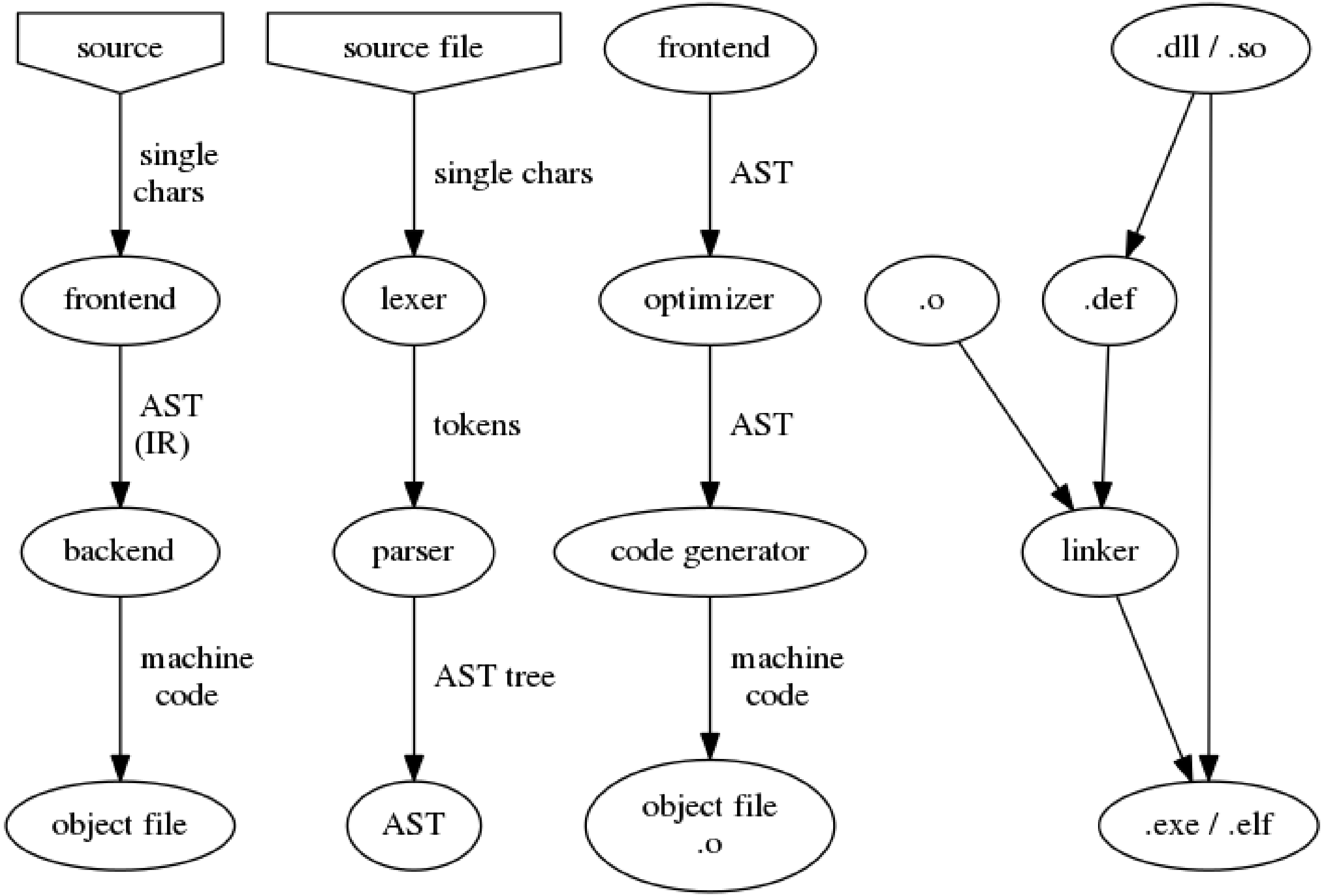
```
mingw32-make EXE=.exe RES=res.res
```

- 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		win32 $(g)Vim$ helper
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"
std::string StringLexBuffer;     /* string parsing buffer /string lexer state/ */

void incFile(AST* inc) {                              /* .include processing */
    yyin = fopen(inc->val.c_str(),"r");
    if (!yyin) yyerror(inc->tagval());
    yypush_buffer_state(yy_create_buffer(yyin,YY_BUF_SIZE));
}
%}
```

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 1.5 manual sections.

lpp.lpp

```
%%
...
```

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

lpp.lpp

```
[ \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; }
```

hpp.hpp

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

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"
%}
```

1.1.3 Headers

Header file contents wrapped by include-once preprocessor hint:

hpp.hpp

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

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

hpp.hpp

```
                                                    // == std.includes ==
#include <iostream>
#include <sstream>
#include <cstdlib>
#include <cstdio>
#include <cassert>
#include <vector>
#include <map>
#include <direct.h>     // win32
#include <sys/stat.h>   // linux
```

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(std::string msg) { std::cout<<YYERR; std::cerr<<YYERR; exit(-1); }
```

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

cpp.cpp

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

1.1.5 Build script

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

`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

[1] and IDE report

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

Makefile

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

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

Makefile

```
.PHONY: exec
exec: doc ./$(MODULE)$(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 ypp.tab.cpp lex.yy.c
H = hpp.hpp ypp.tab.hpp
```

C^{++} compiler run:

Makefile

```
CXXFLAGS += -I. -std=gnu++11
```

```
./$(MODULE)$(EXE): $(C) $(H) $(RES)
	$(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 AST symbolic data type

bI language based on operations on AST symbolic type: [A]bstract [S]yntax [T]ree elements.

class:AST

	string:tag	type, class tag
	string:val	value
	AST(string,string)	<T:V> constructor
	AST(AST)	copy constructor
	List<AST>:nest[]	nested elements
	fn:push(AST)	add nested
	Dict<string,AST>:par[]	parameters dict (string-keyed list)
	fn:setpar(AST)	add/set parameter
	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
	fn:eval()->AST	compute/evaluate object
operators:	op:@(AST)->AST	A @ B apply
	op:=(AST)->AST	A = B equal

hpp.hpp

```
struct AST {                                    // == AST symbolic type ==
// ----------------------------------------------------------------------
    std::string tag;                            // class/type tag
    std::string val;                            // value
// ----------------------------------------------------------------------
    AST(std::string,std::string);               // <T:V> constructor
    AST(AST*);                                  // copy constructor
// ----------------------------------------------------------------------
    std::vector<AST*> nest;                     // nest[]ed elements
    void push(AST*);                            // push nested as stack
// ----------------------------------------------------------------------
```

```
    std::map<std::string,AST*> par;                 // par{}ameters
    void setpar(AST*);                              // add/set parameter
// ----------------------------------------------------------------
    std::string dump(int depth=0);                  // recursive dump(+1)
    virtual std::string tagval();                   // <tag:val> header
    std::string pad(int);                           // padding string
// ----------------------------------------------------------------
    virtual AST* eval();
};
```

Using virtual base class `AST{}` allows to use RTTI and process inherited class instances using pointers to base class, first of all it allows to use storage collections `vector<AST*>` and `map<std::string,AST*>` for any objects[2].

1.2.1 Writers

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

hpp.hpp

```
extern void W(AST*);                                // == writers ==
extern void W(std::string);
```

cpp.cpp

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

[2] instances of inherited classes

1.3 Global environment

hpp.hpp

```
extern std::map<std::string,AST*> env;                    // == global environment ==
extern void env_init();
```

cpp.cpp

```
int main() { env_init(); return yyparse(); }
std::map<std::string,AST*> env;
void env_init() {
    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);                       // bI logo (w/o file extension)
    env["LISPLOGO"] = new Str(LISPLOGO);                   // Lisp Warning logo
    AST*E = env[val]; if (E) return E;
```

1.4 Comments

1.4.1 Line comment

lpp.lpp

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

1.5 Scalars

Bibliography

Dragon Book

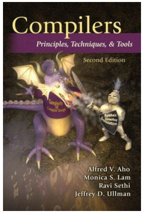
[1] **Dragon Book @ Stanford.edu**
Some lection sets on computer language compilers in free e-books

[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

Functional programming

[6] Peter G. Harrison, Anthony J. Field
Functional Programming

9 1/2 books

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

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

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