bI dymanic language system

© Dmitry Ponyatov <dponyatov@gmail.com>

January 5, 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

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

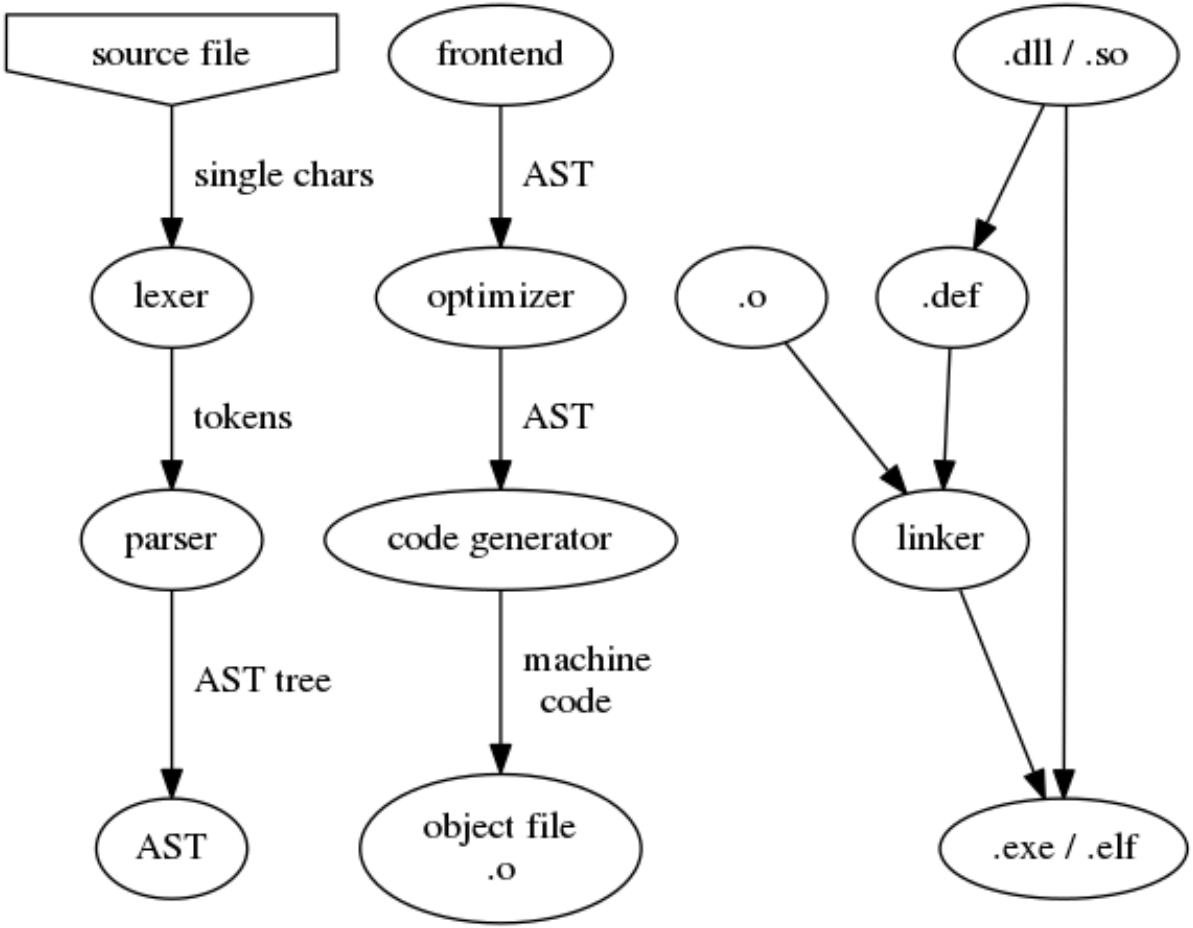
```
mingw32-make EXE=.exe RES=res.res
```

- 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/bl.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;
%}
```

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 1.4 manual sections.

lpp.lpp

```
%%
...
```

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

lpp.lpp

```
[ \t\r\n]+      {}      /* spaces */
.               {}      /* undetected chars */
%%
```

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

hpp.hpp

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

hpp.hpp

```
1                                                    // == parser interface ==
2 extern int yyparse();                              // run parser
3 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

```
1 %{
2 #include "hpp.hpp"
3 %}
```

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 <vector>
#include <map>
#include <direct.h>      // win32
#include <sys/stat.h>    // linux
```

1.1.4 C^{++} core

1.1.5 Build scripts

1.2 AST symbolic data type

bI core 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`	dump tree in text form (with depth padding)
	`fn:tagval()->string`	dump <T:V> header only
	`fn:pad(int)->string`	return padding string: n tabs

hpp.hpp

```hpp
struct AST {                                          // == AST symbolic type ==
// ------------------------------------------------------------------------
    std::string tag;                                  // class/data type tag
    std::string val;                                  // value
// ------------------------------------------------------------------------
    AST(std::string,std::string);                     // <T:V> constructor
    AST(AST*);                                        // copy constructor
// ------------------------------------------------------------------------
};
```

1.3 Comments

1.3.1 Line comment

lpp.lpp

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

1.4 Scalars