序言

However, I do require QUALITY. I will be very glad to know the methodology which you engage to make sure high quality.

How to require **QUALITY**?

坚持三个原则，能帮助我有效保证代码质量：

1、设计优先——“胸中有丘壑，下笔如有神”；

2、逐级优化——框架级、方案级、代码级；

1）保证框架设计符合规范，在性能与规范性（健壮性、可复用性、模式等）之间寻求平衡点；

2）在某一问题有多种解决方案时，衡量利弊，选择最适合的；

3）针对程序中的 hot spot 进行代码级的细致优化；

3、测试驱动开发——针对每一过程的设计流程，设计测试样例，并使用专业测试软件进行测试；

任务分析

工程的目标是构建一个VCD文件的parser，VCD文件分为两种：Four State & Extended；其各自有自身的文法规则，具体定义位于VerilogStd2001 P_{330} & P_{344}， 该parser程序就是按照相应的文法规则对给定文件进行文法分析。

大致过程：

1、对给定文件进行词法分割，以自然分界符（如空格、tab、回车、换行等）将文件分割为若干子串；

2、识别词法分割结果的类型（如关键字、变量、时间等类型），对某些经过分割的串给予更进一步的词法分析（分割或合并），使其成为在文法中有意义的最小单元；

3、根据文法规则构建文法树，自顶向下地对词法分析结果进行文法分析，在分析的过程中记录需要的数据；

可见，任务主要是设计一个词法分析器（Tokenizer）和一个语法分析器（Parser）。

2009-11-06

词法分析

功能需求：为 parser 不断提供 token word（语法意义上的最小单元）；

初步分析：

Syntax for VCD file 的 token word 有几种类型

1）top_level token word

以 $ 为起始字符的各个 section name keyword，包括 declaration_keyword & simulation_keyword & $end.

2）keyword in sub_section

$scope section 的 scope type

$var section 的 var type 等

3）一般字符串

各类 identifier, value change, decimal number, real number etc.

4）其他

$var section 中 reference 定义如下：

```
reference ::=
      identifier
      | identifier [ bit_select_index ]
      | identifier [ msb_index : lsb_index ]
```

其中的 identifier, '[', msb_index, ':', ']'等都是独立意义上的 token word.

1）2）3）皆可以 space 为界，而 4）则不可，因此，**要根据不同的语法需要提供不同的 tokenizer**

进一步分析

从“需求”——语法分析（parser）出发，设计不同的 tokenizer

1．首先，需要能够识别出不同 section 的 tokenizer，

描述：以 space 为界分割 token word，识别以$作为起始符的各个段的关键字或$end，称之为 next_token()。

输入：VCD file text

输出：关键字以及其他类型 enum token_type

1、SECTION_KEYWORD
2、一般 STRING
3、文件结束 EOF
4、其他情况 NONE

备注：由于该 tokenizer 是以 space 为界进行分析，因此它是识别段内 keyword, identifier, decimal number, real number, value change ect 的基础。

其他：理清 Four state VCD file 的关键字

文档类型：four state VCD file
关键字：
1、top class

// declaration keyword

$comment	$date	$enddefinitions	$scope	$timescale	$upscope
$var	$version				

// simulation keyword

$dumpall	$dumpoff	$dumpon	$dumpvars	

// public keyword

$end

// value keyword – **keyword or not?**

0	1	X	x	Z	z
B	b	R	r		

2、scope section

// scope type

begin	fork	function	module	task

3、time scale section

// time number – **keyword or not?**

1	10	100

// time unit – **keyword or not?**

s	ms	us	ps	fs

4、var section

// var type

event	integer	parameter	real	reg	supply0
supply1	time	tri	triand	trior	trireg
tri0	tri1	wand	wire	wor	

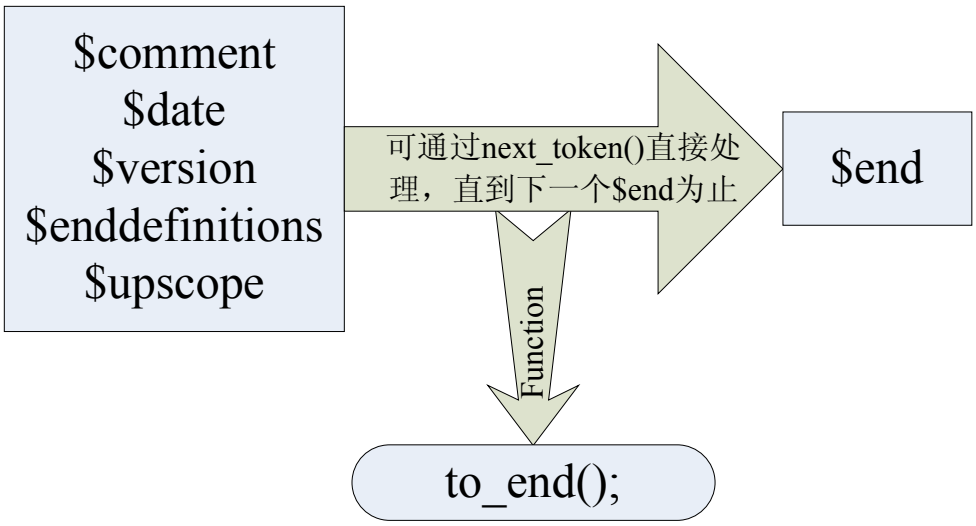
2009-11-07

2、对于 next_token()返回的各种 section 类型，可进一步根据其文法规则整理如下：

declaration_command

✧ $comment, $date, $version, $upscope, $enddefinitions

```
vcd_declaration_comment ::=
        $comment comment_text $end
vcd_declaration_date ::=
        $date date_text $end
vcd_declaration_version ::=
        $version version_text system_task $end
vcd_declaration_enddefinitions ::=
        $enddefinitions $end
vcd_declaration_upscope ::=
        $upscope $end
```



描述：next_token()可以胜任上述 sections 的词法分析，并由此产生一个新的方法 to_end()；且该方法在整个语法分析过程中均适用。

✧ $scope

```
vcd_declaration_scope ::=
        $scope scope_type scope_identifier $end
scope_type ::=
        begin
        | fork
        | function
        | module
        | task
```

描述：将 next_token 返回的一般 STRING 进一步分析识别，返回 scope_type 或者 scope_identifier，称为 next_scopetoken()；

输入：token word

输出：enum scope_type

✧ $timescale

```
vcd_declaration_timescale ::=
        $timescale time_number time_unit $end
time_number ::=
        1 | 10 | 100
time_unit ::=
        s | ms | us | ns | ps | fs
```

描述：将 next_token 返回的一般 STRING 进一步分析识别，返回 time_number 或者 time_unit，称为 next_tscaletoken()；

输入：token word

输出：enum time_scale_type

✧ $var

```
vcd_declaration_vars ::=
        $var var_type size identifier_code reference $end
var_type ::=
        event | integer | parameter | real | reg | supply0 | supply1 | time
        | tri | triand | trior | trireg | tri0 | tri1 | wand | wire | wor
size ::=
        decimal_number
reference ::=
        identifier
        | identifier [bit_select_index ]
        | identifier [msb_index :lsb_index ]
index ::=
        decimal_number
```

描述：在$var section 中，next_token()已不再适用，因为其 token word 不仅仅是以 space 为分界符，因此须重新设计适用于$var 的 tokenizer，可称之为 next_vartoken()；

输入：VCD file text

输出：enum var_token_type

备注：var 的词法分析依赖于 var section 的文法规则，与其他 section 稍有不同，有待进一步设计实现。

simulation_command

✧ $dumpall, $dumpoff, $dumpon, $dumpvars

```
simulation_command ::=
        simulation_keyword { value_change } $end
simulation_keyword ::=
        $dumpall | $dumpoff | $dumpon | $dumpvars
```

描述：value_change 部分分析设计详见下文，其他 next_token()即

可胜任，不需要额外的 tokenizer。

✧ simulation_time: #decimal_number

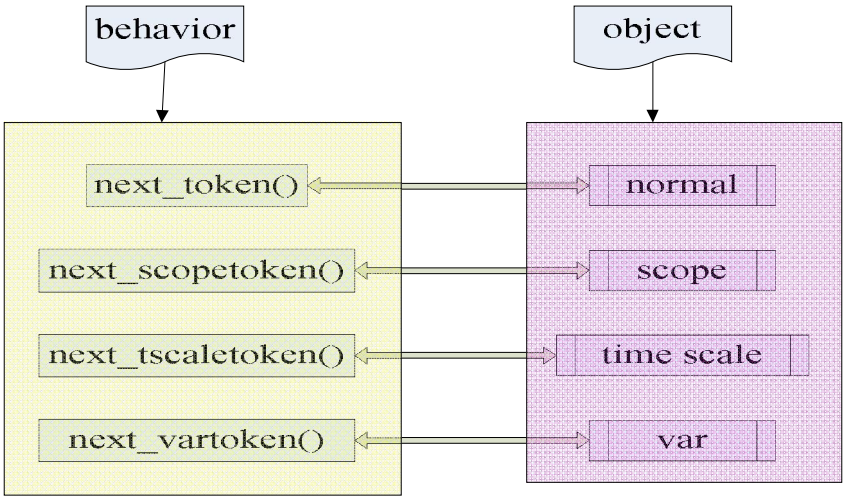
```
simulation_time ::=
        # decimal_number
```

描述：可对 next_token()返回的一般 STRING 中以'#'开头并跟一个 decimal number 的返回此文法类型，可在 parser 中直接处理，无需单独的 tokenizer。

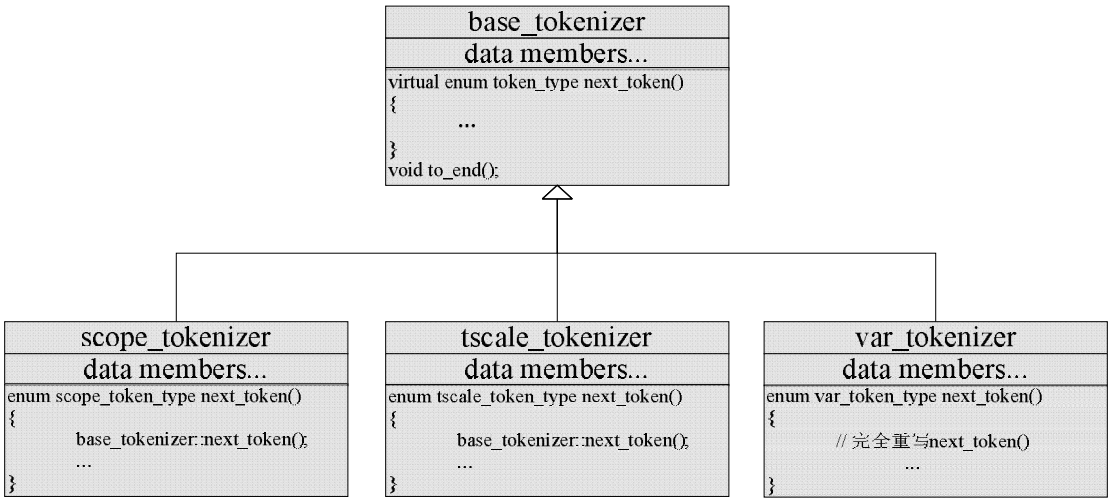
✧ value_change

描述：next_token()返回的一般 STRING 类型亦包括 value change，其具体遵从 value change 的文法规则，可在 parser 阶段设计实现，因此也无需单独的 tokenizer。

至此，可以看到，共有 4 种 tokenizer 对应于 4 类不同的 sections



初步设计——依据面向对象的设计原则，做出如下设计：



分析：

很显然，这是一种基于 polymorphism（多态）的设计，其根本目的就在于一个继承体系中的所有实例对象的行为达到运行时的 polymorphism，也即行为客制化。但该设计在本应用中却存在如下两个问题：

1、 如果在 parser 中定义了一个 base_tokenizer 的指针或引用 p，其实际指涉四个对象中（一个 base，三个 derieved，这四个对象需要在 parser 中实实在在存在）的任何一个；在使用过程中通过判断当前分析的段来切换 p 的值；此处的切换是需要程序员手动控制的，这也就曲解了 polymorphism 的本质（运行时“动态”调用实际对象的相关行为！）

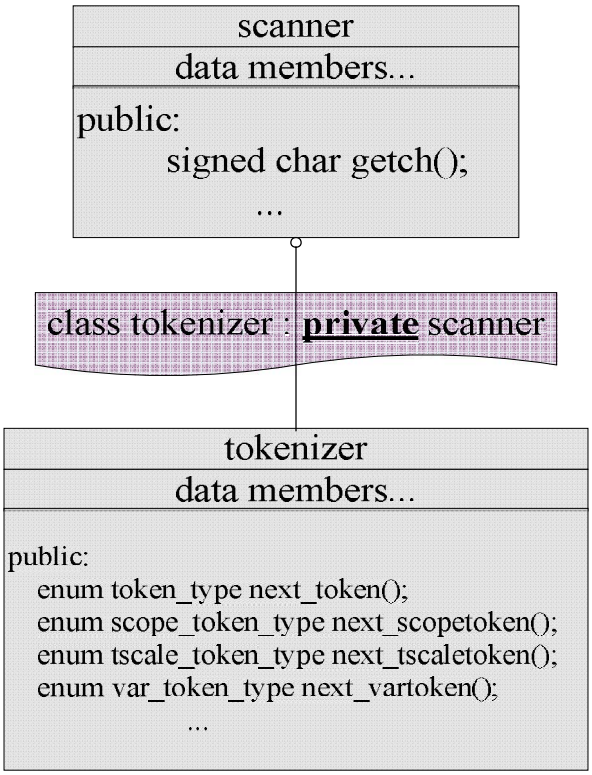
2、 四个对象需要共享一份父类对象，因为父类对象中可能维护着文件对象、当前分析位置等公有信息，而且这些信息的改动需要保持一致性；企图用 virtual base class 解决之，然而 virtual base class 的应用场合及其实现效果并不适用。

解决方案：

1、 polymorphism 无非是实现运行时的行为客制化，而在该应用中，客制化的每一个行为又是程序员预先知晓的，因此可将其运行时的客制化提前至编译期，考虑使用 overload functions（重载函数族）来实现，但仔细分析，发现各个行为的参数列表没什么差别，因此干脆直接使用最原始的 related functions（族函数）就可以了，他们函数名称相近，完成的功能也相似，只是实现细节以及应用场合不同。因此上述继承体系亦不存在，取而代之的是一个 tokenizer class 及其相关的一族成员方法来实现各个 tokenzier。

2、 如果针对问题 1 的解决方案得到施行，方案 2 的问题似乎已经不存在（不存在多个子类对象，也无需保持对父类部分数据访问的一致性了）。但方案 2 体现的却是一个“内聚”的问题，也就是说现有的 tokenizer（包括方案 1 中）做的事情太多，尤其是与 IO 相关的部分（包括文件、当前位置等），不符合“高内聚”的要求；因此我们考虑将其提取出来设计成为一个单独的 class，称之为 scanner，其与 tokenizer 的接口就是由 getch() 向 tokenizer 顺序提供单个字符，这样的设计亦符合“低耦合”的要求。

至此，我们已经完成了对 tokenizer 的所有分析，并为之确定了一个单一继承体系作为实现，如下图所示：



如图所示，tokenizer 与 scanner 确实是 **private** inheritance，理由：

1、 public inheritance 塑模的是 is-a 的关系，也即 derieved 是一种 base，在此并不符合。

2、 private inheritance 体现的是 is-implemented-in-terms-of（根据某物实现出）的关系，而 tokenizer 正是根据一个个的 getch 而实现了 next_token()，符合此关系。

3、 private inheritance 纯粹只是一种实现技术，base 与 derieved 之间并不存在任何观念上的继承关系。

4、 private inheritance 意味着只有实现部分被继承，接口部分应略去，该应用正是仅仅继承了 getch()实现。

注：上述依据摘自《effective C++ 3rd Edition》第 6 部分 – Inheritance and Object-Oriented Design.

2009-11-08

详细设计

scanner

功能：为 tokenizer 提供一个个单独的 character

接口：signed char get_char();

初步设计：

scanner
ifstream vcd_file;

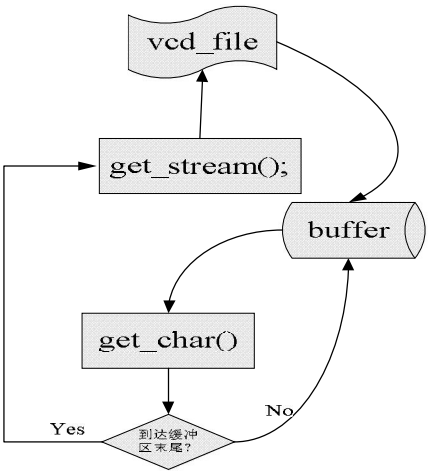
public:
signed char get_char()
{
signed char ch;
vcd_file.read(&ch, 1);
return ch;
}

问题：

1、 输入操作>>（或者是 get()）会在输入过程中跳过空白（空格、制表符、换行符、换页符和回车符，由<cctype>里的 isspace()定义），因此此处的输入只能使用 read()，但频繁地做单个字符的 IO 操作会大大影响性能。

解决方案：

1、 将文件一次读入一个缓冲区 buf，再从缓冲区逐个取 character；缺点是如果文件较大，一次读入耗费空间较多，因此可采用定长缓冲区策略，分批读文件至缓冲区中，其基本过程可由下图表示：



此时涉及到系统的一种基本类型的选择：string **or** c array?作为C++阵营的一员，本应完全摈弃 c array；字符串有 string，数组有各种容器，因此貌似并无 c array 的用武之地；

但对效率敏感的地方，仍需要 c array；比如在该应用中，对字符和字符串的操作贯穿整个应用程序，是应用程序的一个重要的方案决策点；而且，**应用程序对字符的操作以顺序读写和字符串比较为主，c array 完全可以胜任**，string 有关 safe check、allocator 等的代价并不低廉，而 string 所提供的众多接口却很少用得到；

因此，**选择 c array 而非 string**。

为保持接口一致性，文件也使用 C 中的 FILE 指针。

至此，可设计 scanner class 雏形如下：

scanner
private: FILE* vcd_file; char buf[BUF_SIZE]; char* pt_cur; char* pt_tail;

private:
int get_stream();
public:
signed char get_char();

对 scanner 的不断改进过程也证明了**模块化设计**的重要性，只要接口不变，scanner 与 file 的交互方式可以随意改进，而不影响其客户 tokenizer。

接下来，我们需要**考查 scanner 作为一个 class（new type）的行为**，尤其是对象的创建和销毁（影响到 constructor & destructor），对象的初始化与赋值（copy constructor & assignment operator）等，对每一个 new type 的设计都应如“语言设计者当初设计语言内置类型时”一样的谨慎来研讨 class 的设计。

1）创建和销毁

scanner 作为系统的 IO 接口，负责与 file 交互，而且每个 scanner 必须负责一个 FILE 对象，因此需要一个 FILE 作为参数来创建 scanner，并在 constructor 中打开文件，destructor 中关闭文件。

```
scanner( const char* ){ /*…*/ };
```

至于 default constructor 是否必要，暂不下结论。

2）初始化与赋值

将一个对象赋值给另一个对象或者用一个对象初始化另外一个对象，在 scanner 的含义是将对该文件分析的当前状态（包括当前文件指针的位置）做一份拷贝；在系统中，scanner 是唯一负责 IO 交互的接口，而且**对于一份文件而言，我们希望只有一个 scanner 对其负责（多个 scanner 负责一个 file 会带来同步等令人头痛的问题）**，因此做此类赋值或拷贝没有意义，因此，我们应**禁止 scanner 的拷贝行为**。方式很多，我们采取 **base class 策略（见下图）**。

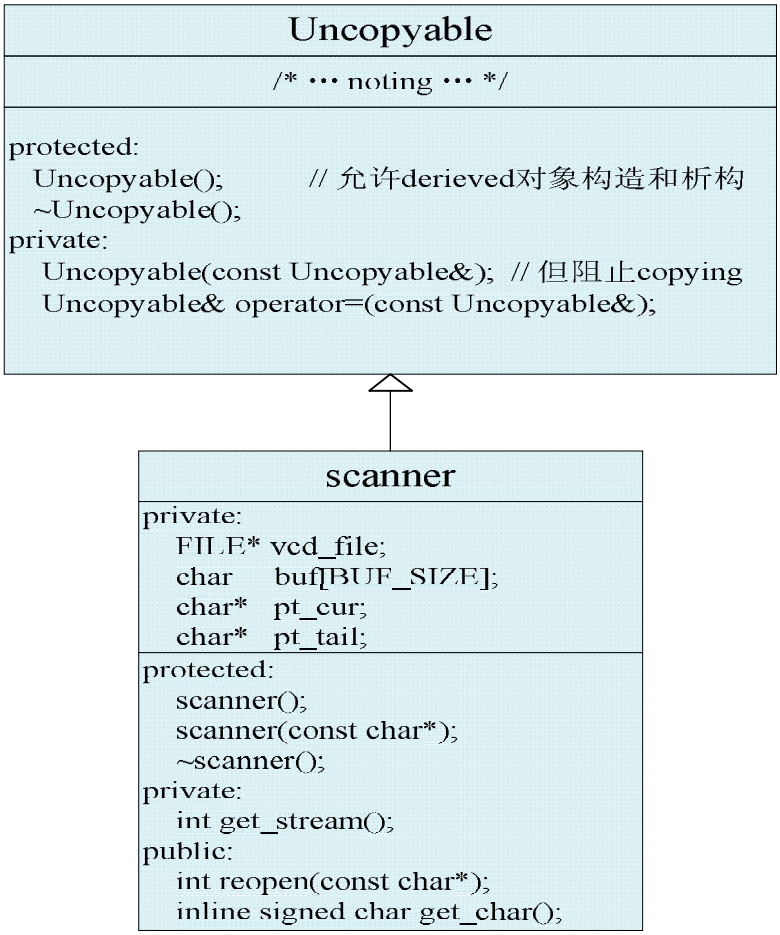
3）对象复制的引申

由 2）的分析可知，每个 file 只有一个 scanner 负责，但 scanner 是否只能为一个 file 服务呢？否！当我们有多个 VCD 文件待分析时，完全没有必要为每个文件创建一个 scanner 对象，相反，可以让 scanner 完成一个文件的 IO 之后，使之为另外的文件服务，这时，我们貌似需要一个重新打开文件的操作 reopen(const char*);

4）回到对象创建

既然 scanner 可以重新打开一个文件，那么默认创建一个不打开任何文件的 scanner 也就顺利成章，因此 **default constructor 仍然需要**！

此时，我们可将我们的 scanner 完善如下：



注：**scanner** 是我们系统中遇到的第一个 **class**，而 **C++**程序设计的主要任务就是对 **class** 的设计，因此描述的详尽一些，体现了设计的一些基本思路和遵循的原则；当然，接下来的每一个 **class** 以及他们之间的关系，我都会按照一定的原则进行细致的设计，但描述可能不会那么详尽，特在此处说明一下。

tokenizer

首先，是相关类型设计

```
typedef enum token_type { … } Tokens;
static char *tokens[] = { … } ;

typedef enum scope_token_type { … } ScopeTokens;
static char *scopetokens[] = { … } ;

typedef enum tscale_token_type { … } TScaleTokens;
static char *tscaletokens[] = { … } ;

typedef enum var_token_type { … } VarTokens;
static char *vartokens[] = { … } ;
```

其次，data members

1) 存储当前 token word 的字符数组 char token_word[];

注：该数组初始大小由 MAX_TOKEN_LEN 确定，因为 value

change 的 var 名称可以使任意长度，因此，该数组需要提供扩容的功能；

2) token word 的有效长度 len（以'\0'为结束符，不计算在内）；

3) 上述 MAX_TOKEN_LEN 可能会改变，因此不能定义为宏或全局静态量，因此定义为静态成员变量 max_token_len；

再次，member functions

1) 取 token word 的 inline 函数 get_word();

2) 为 token_word 扩容，inline 函数 put_char();

3) 掠过空白区的函数 skim_space();

4) 一般的 next_token();

5) scope section 的 next_scope_token();

6) time scale section 的 next_tscale_token();

7) var section 的 next_var_token();

因此，tokenizer 的雏形可设计如下：

tokenizer : private scanner	
private:	
char	token_word[];
int	token_len;
int	max_token_len;
private:	
inline void	put_char();
void	skim_space();
public:	
Tokens	next_token();
ScopeTokens	next_scope_token();
TScaleTokens	next_tscale_token();
VarTokens	next_var_token();

2009-11-10

实现过程改动

1、Var_token()过程中，需要一个字符保存当前掠过字符 prevchar，作为 tokenizer 的新的 data member；

2、做 identifier[msb:lsb]分析时还需要一个提前观察下一个字符（却不改变当前指针位置）的方法，称为 next_peek()；**这是对之前设计的 class scanner 的改动**；其实，“设计先行”的方式对于工程质量有很大帮助，但先行一步的设计总是没那么完善，在实现过程中需要不断的改进，但只要大致思路不变，就算的上一个不错的设计，对工程进展是具备指导意义的。

3、tokenizer 中，next_var_token 不适于处理 value change name（因为它的文法规则为任意类型任意长度的非空白 ASCII 代码），因此设计专为取 value change 变量名的 next_str_token()，该函数异常简单，只需以 space 作为分界符即可；

代码优化一览

CScanner

1、将 get_char() inline 处理，该函数调用频度高，函数复杂度低，很适合做 inline 处理；

CTokenizer

1、对 keyword 表按照调用频度的先验知识进行排序，使全局查找的总比较次数将为最低；

2、对各种 keyword 表的查找，利用 map_token_base()处理，提高代码复用率；

3、使用单独的函数接口对每一个 keyword 表进行查找操作，易于针对每个表的查找优化，只修改相应函数而调用该接口的地方不用改变；

4、将部分函数 inline 处理，如 put_char()，to_end()，get_word()等；

5、将“掠过空白区”这样的功能定义为宏 skim_over_space，既提高了代码复用率，也没有使用 inline 函数需要判断返回值的冗余代价；

6、将对 white space 的判断从调用 isspace()提炼为”ch <= ‘ ’”这样的条件语句，得益于输入是 text 文件，这种比较在整个应用程序中频率非常高，对其进行优化很有价值；

7、对数字字符（0~9）的判断写作”ch <= ‘9’ && ch >= ‘0’ ”而非“ch >= ‘0’ && ch <= ‘9’ ”；因为大部分非 number 字符 ASCII 值都比 9 大，先判断 ch <= ‘9’可以提前使逻辑表达式为假，后面的表达式不必再判断，从宏观上可以改善性能；