# A Small Domain Specific Language for Cryptographic Algorithms

Jaimandeep Singh

Brijendra Kumar Joshi
Professor
MCTE

## ABSTRACT
This paper establishes the need for a small Domain Specific Language to support rapid testing and diagnosis of cryptographic algorithm. Such a language will require built-in support for modulo arithmetic, which is used for creating mathematical locks in modern crypto systems.

The paper also provides a framework/prototype for such a language. It can further be used as a building block for broader set of language specific features.

## General Terms
Cryptographic Algorithms, Domain Specific Language, Compiler, Parser, Lexer.

## Keywords
cryptography, flex, bison, lex, yacc, ply, parser, grammar, tokenizer, compiler, lexer, llvm, llvmpy.

## 1. INTRODUCTION
Presently, cryptographic applications are coded in conventional general purpose programming languages like C/C++. These languages do not have built-in features that support testing and diagnosis of cryptographic algorithms. Other languages that support cryptographic applications like MATLAB are very resource heavy, proprietary and require licenses to work.

Also, the existing general purpose languages have turned into a goliath of language specific features. The beginner in programming languages find it very difficult to write even a simple looping construct because of ten different ways of doing the same thing. They are lost in the mire of features rather than working on the application logic/business model. Therefore, there is a need for a small domain specific programming language targeting cryptographic applications.

This paper describes a prototype for a small programming language which has built in support for modular arithmetic. Most of the modern cryptographic algorithms are based on modular arithmetic as it can be used to create mathematical locks that are easy in one direction but hard in another, this is also known as discrete logarithm problem.

The Python Language because of its flexibility has been used for quick prototyping. The llvm compiler infrastructure project is modular and easily understandable and has therefore been chosen over gcc.

## 2. STAGES OF A COMPILER
A compiler is a large and complex piece of code and for simplicity has been divides into various modules/stages. A simple compiler consists of four stages as shown in the Fig 1[1].

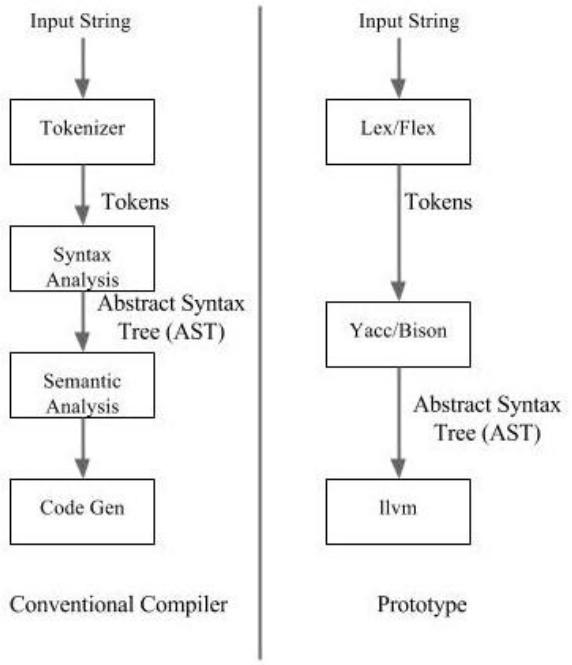


**Fig 1: Stages of a compiler**

The tokenizer takes the input string and breaks it into tokens. An input string of `'5 * 7 + 3'` is split into tokens `5, *, 7, +, 3`. The Syntax Analysis Stage or the parser checks the validity of the expressions based on context-free-grammar rules. Semantic analysis stage detects errors like type mismatch, redefinition, parameters mismatch etc. Code Generation stage converts the Abstract Syntax Tree (AST) into target machine code.

In our prototype, we are using PLY Python package. PLY package consists of *lex* and *yacc* modules. *Lex* performs the function of a *Tokenizer* in a conventional compiler. It tokenizes the input stream based on regular expression rules.

*yacc* performs the functionality of *Syntax Analysis* in a conventional compiler. It checks language syntax validity based on context free grammar rules.

llvm is a Compiler Infrastructure Project, which is used for backend code generation and optimization.

## 3. LEXER
There are two options for a lexer: either we can handcraft a lexer or use existing toolsets like *Lex/Flex*. Handcrafting a lexer could be buggy and result in unforeseen errors in later stages of our compilation process. Therefore, in our prototype we will be using standard utility call *Lex* which stands for *A*

*Lexical Analyzer Generator* originally written by *Mike Lesk* and *Eric Schmidt*[2].

*Lex* is a program generator for lexical processing of input streams. Simply stated, it is used to split the input string into tokens or lexemes.

Python has *PLY (Python Lex-Yacc)* package for lexing and parsing[3]. Our prototype uses `lex.py` module from this package for tokenizing or lexing the input stream.

The output of a typical tokenizer is given below in Fig 2. It produces tokens along with their associated values. The object returned by the PLY lexer has attributes (type, value, lineno, lexpos)[4].

```
Enter expression: 3 * -4 + 5 / 7
LexToken(NUMBER,3,1,0)
LexToken(TIMES,'*',1,2)
LexToken(MINUS,'-',1,4)
LexToken(NUMBER,4,1,5)
LexToken(PLUS,'+',1,7)
LexToken(NUMBER,5,1,9)
LexToken(DIVIDE,'/',1,11)
LexToken(NUMBER,7,1,13)
```

**Fig 2: Output of a Typical Lexer**

## 4. PARSER

The input string is first tokenized with the help of lexer, these tokens are then given as input to the parser which then produces the Abstract Syntax Tree (AST) based on the context-free grammar rules.

There are two major techniques of parsing known as *top-down parsing* and *bottom-up parsing*. These techniques differ in the way in which they build up the parse tree. The top-down parsers build the parse tree starting from the root node and work down to the leaves whereas bottom-up parsers start form the leaves and go all the way up to root node.

LR (Left to right, Rightmost Derivation) is a *bottom-up parser*. Here *"L"* is for left to right scanning of input and *"R"* is for constructing rightmost derivation in reverse [5].

`yacc.py` module of PLY package uses LALR(1) parsing and is based on unix utility of *yacc*[3]. LALR parser or Look-Ahead LR parser is a variant of LR parser.

### 4.1 Abstract Syntax Tree

The prototype uses simple operator grammar for parsing standard arithmetic operators. This is given as docstings in the Listing 1 below. Operator precedence and associativity is used to remove any kind of ambiguity in grammar.

`class Expr` is the base class. `BinOp` and `Number` are the derived classes of the base class `Expr.` Whenever reduction rule is applied a new object of the respective class gets instantiated and becomes part of the parse tree.

Once the parse tree is built and we are at the root node, `Codegen` method of respective classes gets called recursively.

**Listing 1. Parser for Printing AST for Simple Arithmetic Operations**

```
# Precedence rules for the arithmetic
# operators
precedence = (
 ('left','PLUS','MINUS'),
 ('left','TIMES','DIVIDE'),
 )
class Expr: pass

class BinOp(Expr):
 def __init__(self,left,operator,right):
   self.type = "binop"
   self.left = left
   self.right = right
   self.operator = operator

 def CodeGen(self):
   left = self.left.CodeGen()
   right = self.right.CodeGen()

   if self.operator == '+':
     return (left, right, 'addtmp')
   elif self.operator == '-':
     return (left, right, 'subtmp')
   elif self.operator == '*':
     return (left, right, 'multmp')

class Number(Expr):
 def __init__(self,value):
   self.type = "number"
   self.value = value

 def CodeGen(self):
   return (self.type, self.value)

def p_statement_expr(p):
 """statement : expression"""
 print (p[1].CodeGen())

def p_expression_binop(p):
 """expression : expression PLUS expression
               | expression MINUS expression
               | expression TIMES expression
               | expression DIVIDE expression
 """
 p[0] = BinOp(p[1],p[2],p[3])

def p_expression_number(p):
 """expression : NUMBER"""
 p[0] = Number(p[1])
```

The above code would spit out a tree data structure, as in Fig 3, taking into consideration the operator precedence and context-free grammar rules defined inside the code.

```
> 2 + 3
(('number', 2), ('number', 3), 'addtmp')
> 2 + 3 * 5
(('number', 2), (('number', 3), ('number', 5), 'multmp'), 'addtmp')
> 4 * 5 + 1
((('number', 4), ('number', 5), 'multmp'), ('number', 1), 'addtmp')
```

**Fig 3: Output of Parser as Abstract Syntax Tree**

### 4.2 Extension of Grammar for Modular Arithmetic

The typical arithmetic grammar was extended to support modulo operations. The extended grammar is given as docstrings in Lisitng 2. Here, `Modulo` class extends the base `class Expr` given in Listing 1. A new instance of `Modulo` class is created whenever the reduction rule is applied.

**Listing 2. Grammar for Modular Arithmetic**

```
def p_expression_modulo(p):
 """expression : LPAREN expression RPAREN
MODULO LPAREN expression RPAREN"""
 p[0] = Modulo(p[2], p[6])
```

# 5. CODE GENERATION

llvm compiler infrastructure was chosen over GCC for prototyping because of its modularity and ease of usage. Also, GCC is built as a monolithic static compiler, which makes it extremely difficult to use as an API and integrate into other tools. Further, its historic design and current policy makes it difficult to decouple the front-end from the rest of the compiler[6].

## 5.1 Adding a Function

The block diagram for adding a function to llvm is given in Fig 4.

The first step in adding a function to llvm is to create a new module. Modules are top level containers which contain functions, global variables, and symbol table entries[7]. After creating a new function we provide it with arguments. Functions do not have free-standing and are therefore added to the modules.

Instructions are now added to functions in form of Basic Blocks.

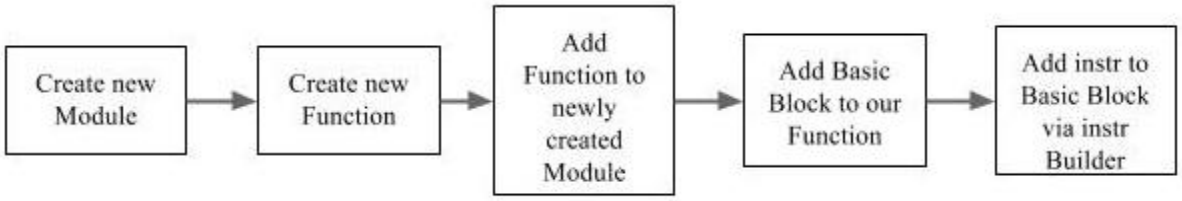


**Fig 4: Block Diagram for Adding a Function to llvm**

## 5.2 Compiling and Running Code

The steps involved in compiling and running the prototype is given in terms of a block diagram in Fig 5.

Firstly, we create an object of Execution engine and then add the module to the execution engine. Thereafter, we provide arguments to the functions inside the module.

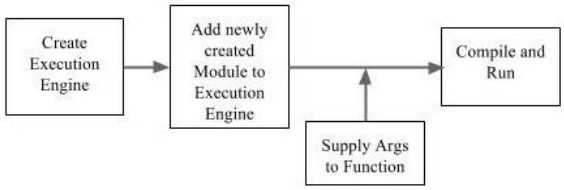


**Fig 5: Block diagram for Running Code**

## 5.3 Code Generation for Modular Arithmetic

Modular arithmetic is also known as clock arithmetic where the numbers wrap around on reaching a certain value also called as modulus.

Let us take two examples, firstly when both the operands are positive such as 13 mod 5 = 3. Now, let us consider case of a negative operand such as -5 mod 3 = 1.

The class `Modulo` given in Listing 3 takes into account both the cases when the operands are positive and when they are negative.

**Listing 3. Class for Modulo Arithmetic**

```
class Modulo(Expr):
 def __init__(self, left, right):
  self.type = "modulo"
  self.left = left
  self.right = right

 def CodeGen(self):
  global g_llvm_builder
  left = self.left.CodeGen()
  right = self.right.CodeGen()

  mod_result=g_llvm_builder.srem(left, right,
'modtmp')

  # Convert condition to a bool by comparing
'greater than equal to' 0
  condition_bool=g_llvm_builder.icmp(ICMP_SGE
, mod_result, Constant.int(Type.int(), 0),
'modcond')

  function=g_llvm_builder.basic_block.functio
n

  # Create blocks for the +ve and -ve result
cases.
  positive_block=function.append_basic_block(
'positive')
  negative_block=function.append_basic_block(
'negative')
  merge_block=function.append_basic_block('mo
dcond')

  g_llvm_builder.cbranch(condition_bool,
positive_block, negative_block)

  # Block for positive mod value.
  g_llvm_builder.position_at_end(positive_blo
ck)
  positive_value = mod_result
  g_llvm_builder.branch(merge_block)

  positive_block = g_llvm_builder.basic_block

  # Block for negative mod value.
  g_llvm_builder.position_at_end(negative_blo
ck)
```

```
  negative_value=g_llvm_builder.add(mod_resul
t, right, 'addbase')
  g_llvm_builder.branch(merge_block)

  negative_block = g_llvm_builder.basic_block

  # Emit merge block.
  g_llvm_builder.position_at_end(merge_block)
  phi=g_llvm_builder.phi(Type.int(), 'iftmp')
  phi.add_incoming(positive_value,
positive_block)
  phi.add_incoming(negative_value,
negative_block)

  return phi
```

The output of the code generated by the class `Modulo` is given in Fig 6.

```
 > (2 - 5) modulo (1 + 1)

; ModuleID = 'my calculator'

define i32 @calculator() {
entry:
  br i1 false, label %positive, label %negative

positive:                                         ; preds = %entry
  br label %modcond

negative:                                         ; preds = %entry
  br label %modcond

modcond:                                          ; preds = %negative, %positive
  %iftmp = phi i32 [ -1, %positive ], [ 1, %negative ]
  ret i32 %iftmp
}

returned 1
```

**Fig 6: Output Code Generation of Modular Arithmetic**

## 6. CONCLUSION

There is a kind of race in providing more and more advanced features in any leading general purpose programming language. An entry level programmer is overawed by the feature list and the so called the *best practices* of doing even a simple thing.

Also, the current generation general purpose programming languages like C/C++ do not have built-in support for modulo arithmetic. The languages like MATLAB that do support modulo arithmetic are resource heavy and proprietary.

Therefore, there is a niche area for a small domain specific language that has built-in support for modulo arithmetic and which provides useful support in testing and diagnosis of cryptographic algorithms. This paper addresses this very area and builds a prototype for such a language.

## 7. REFERENCES

[1] V Raghavan; 2010; *"Principles of Compiler Design"*; 1st edition, Tata McGraw Hill Education Private Limited, New Delhi, p 7, 70-80.

[2] John R. Levine, Tony Mason and Doug Brown; 1992; *"lex & yacc"*; 2nd Edition, O'Reilly, p 1–2.

[3] https://pypi.python.org/pypi/ply/3.1 (accessed on June 24, 2014).

[4] http://www.dabeaz.com/ply/ply.html#ply_nn9 (accessed on June 24, 2014).

[5] Alfred V. Aho, Ravi Sethi and Jeffery D. Ullman; 2002; *"Compilers: Principles, Techniques and Tools"*; 9th Indian edition, Pearson Education (Singapore) Pte. Ltd., Indian Branch, Delhi, p 215.

[6] http://clang.llvm.org/comparison.html (accessed on June 24, 2014).

[7] http://llvm.org/docs/ (accessed on July 12, 2014).

[8] http://www.llvmpy.org/ (accessed on June 25, 2014).

[9] Andrew W. Appel; 2001; *"Modern Compiler Implementation in C"*; Special Edition for Sale in South asia; Cambridge University Press.

[10] Jean Paul Tremblay and Paul G.Sorenson; 1985; *"The Theory and Practice of Compiler Writing"*; McGraw-Hill Book Company, USA.

[11] S. C. Johnson; 1975; *"YACC: Yet Another Compiler Compiler"*; Computing Science Technical Report no 32, Bell Laboratories, Murray Hills, New Jersey.

[12] http://flex.sourceforge.net/ (accessed on July 12, 2014).

[13] Brain W. Kernighan and Rob Pike, 1995, *"The UNIX Programming Environment"*; 9th Indian edition, Prentice-Hall of India Private Limited, New Delhi, p 233-260.