# X0-Compiler Design Document

Altair, Liu @ ilovehanhan1120@hotmail.com

November 23, 2018

# 1 Introduction

## 1.1 Purpose

The purpose of conducting as technical proposal to describe the global designing of this project, containing basic functionality of the system, run-time designing and error detecting methods. This document is aimed to provide a schema of designing and implement all functionality, which will be the critical document during the process of developing. This document will be read by developers and testers.

## 1.2 Background

This project is to develop a **X0 Language Compiler**, which is a C-like language. This project is mainly for research and study purpose.

| Item | Detail |
|------|--------|
| Project Name | X0-Compiler(mini-C) |
| Developing Platform | Ubuntu 18.04 64-bit |
| Developing Tools | **Flex** and **Bison** |
| Open Source or not | Yes |

All source files can be found at: http://github.com/SubjectNoi/X0-Compiler, star and follow it, please ˆˆ.

## 1.3 Remarks

Usage:

```
1  Ubuntu>$ git clone http://github.com/SubjectNoi/X0-Compiler
2  Ubuntu>$ cd X0-Compiler
3  Ubuntu>$ make
4  Ubuntu>$ ./X0 [Your source file]
```

# 2 Design Summarize

## 2.1 Main purpose of the project

Following are main purposes of this project:

- Run correctly on target OS: Ubuntu 18.04 64-bit

- Compile X0 language

- Report compile error, including syntax and semantic error

## 2.2 Primary demand

The X0 compiler should compile these C-like language, detailed grammar definition will be showed in next section.

```
1   main {
2         integer i, j, flag, cnt := 0;
3         for (i := 2; i != 101; i++) {
4               flag := 0;
5               for (j := 2; j != i; j++) {
6                     if (i % j == 0) {
7                           flag := 1;
8                           break;
9                     }
10              }
11              if (flag == 0) {
12                    write(i);
13                    cnt++;
14              }
15        }
16        write("There're:");
17        write(cnt);
18        write("Primes.");
19  }
```

And correct result should be given. If there exists syntax or semantic error, compiler should report them.

## 2.3 Restrictions of Design

To complete this project, following restrictions should be watched out:

- Project will be only run on Ubuntu 18.04

- Both developing and testing should be finished before 2018-11-26T11:30:00.000Z

## 2.4 Principles and Rules of Design

Following principles should be followed in the process of developing:

- Complete: implement as many features as possible

- Simple: try best to ensure low coupling between modules

- High Efficiency: try best to ensure the highest execution efficiency of virtual machine code.

When developing, following rules should be obey:

- All files should be named under following rules:

| File | Naming rule |
|------|-------------|
| Yacc file | X0-Bison.y |
| Lex file | X0-Lex.l |
| Constructing file | Makefile |
| Testing source | /TestingSrc/Test**XX**_[**Testing Content**] |
| Git ignore file | .gitignore |

- Git is used for version control

- Use **git fetch && git pull**

- Use **git rm -r −cached .**

- Use **git add .**

- Use **git commit -am [Meaningful Comment]**

- Never **git push -f**

# 3 Main Design

## 3.1 Demand

In this sub-section, detailed grammar of X0 Language will be given:

$$\textbf{program} \rightarrow {'main'}, \{, \textbf{statement\_list}, \} \tag{1}$$

$$\textbf{statement\_list} \rightarrow \textbf{statement\_list}, \textbf{statement} \tag{2}$$

$$| \textbf{statment} \tag{3}$$

$$| \epsilon \tag{4}$$

$$\textbf{statement} \rightarrow \textbf{expression\_list} \tag{5}$$

$$| \textbf{if\_statement} \tag{6}$$

$$| \textbf{while\_statement} \tag{7}$$

$$| \textbf{read\_statement} \tag{8}$$

$$| \textbf{switch\_statement} \tag{9}$$

$$| \textbf{case\_stat} \tag{10}$$

$$| \textbf{write\_statement} \tag{11}$$

$$| \textbf{compound\_statement} \tag{12}$$

$$| \textbf{for\_statement} \tag{13}$$

$$| \textbf{do\_statement} \tag{14}$$

$$| \textbf{declaration\_list} \tag{15}$$

$$| \textbf{continue\_stat} \tag{16}$$

$$| \textbf{break\_stat} \tag{17}$$

$$| \textbf{yarimasu\_stat} \tag{18}$$

$$| \epsilon \tag{19}$$

$$\textbf{declaration\_list} \rightarrow \textbf{declaration\_list}, \textbf{declaration\_stat} \tag{20}$$

$$| \textbf{declaration\_stat} \tag{21}$$

$$| \epsilon \tag{22}$$

$$\textbf{declaration\_stat} \rightarrow \textbf{typeenum}, \textbf{identlist}, {';'} \tag{23}$$

$$| \textbf{typeenum}, \textbf{identarraylist} \tag{24}$$

$$| {'const'}, \textbf{typeenum}, \textbf{identlist}, \textbf{SEMICOLONSTAT} \tag{25}$$

$$| {'const'}, \textbf{typeenum}, \textbf{identarraylist} \tag{26}$$

$$\textbf{identlist} \rightarrow \textbf{identdef} \tag{27}$$

$$| \textbf{identlist}, {','}, \textbf{identdef} \tag{28}$$

$$| \epsilon \tag{29}$$

$$\textbf{identdef} \rightarrow IDENT \tag{30}$$

$$| IDENT, {':='}, \textbf{factor} \tag{31}$$

$$\tag{32}$$

$$\textbf{typeenum} \rightarrow 'integer' \tag{33}$$
$$|'string' \tag{34}$$
$$|'bool' \tag{35}$$
$$|'real' \tag{36}$$
$$|'char' \tag{37}$$
$$\textbf{identarraylist} \rightarrow \textbf{identarraydef} \tag{38}$$
$$|\textbf{identarraylist},',',\textbf{identarraydef} \tag{39}$$
$$\textbf{identarraydef} \rightarrow IDENT,'[',\textbf{dimensionlist},']' \tag{40}$$
$$\textbf{dimensionlist} \rightarrow \textbf{dimension} \tag{41}$$
$$|\textbf{dimensionlist},',',\textbf{dimension} \tag{42}$$
$$\textbf{dimension} \rightarrow INTEGER \tag{43}$$
$$\textbf{switch\_statement} \rightarrow 'switch','(',\textbf{expression},')','\{',\textbf{case\_list},\textbf{default\_statement},'\}' \tag{44}$$
$$\textbf{case\_list} \rightarrow \textbf{case\_list},\textbf{case\_stat} \tag{45}$$
$$|\textbf{case\_stat} \tag{46}$$
$$|\epsilon \tag{47}$$
$$\textbf{case\_stat} \rightarrow 'case',\textbf{expression},':',\textbf{compound\_statement} \tag{48}$$
$$|\epsilon \tag{49}$$
$$\textbf{default\_statement} \rightarrow 'default',':',\textbf{compound\_statement} \tag{50}$$
$$\textbf{continue\_stat} \rightarrow 'continue',';' \tag{51}$$
$$\textbf{break\_stat} \rightarrow 'break',';' \tag{52}$$
$$\textbf{if\_statement} \rightarrow 'if','(',\textbf{expression},')',\textbf{compound\_statement},\textbf{else\_list} \tag{53}$$
$$\textbf{else\_list} \rightarrow 'else',\textbf{compound\_statement} \tag{54}$$
$$|\epsilon \tag{55}$$
$$\textbf{while\_statement} \rightarrow 'while','(',\textbf{expression},')',\textbf{compound\_statement} \tag{56}$$
$$\textbf{write\_statement} \rightarrow 'write','(',\textbf{expression},')' \tag{57}$$
$$\textbf{read\_statement} \rightarrow 'read','(',\textbf{var},')' \tag{58}$$
$$\textbf{compound\_statement} \rightarrow '\{',\textbf{statement\_list},'\}' \tag{59}$$
$$\textbf{for\_statement} \rightarrow 'for','(',\textbf{expression},';',\textbf{expression},';',\textbf{expression},')', \tag{60}$$
$$\textbf{compound\_statement} \tag{61}$$
$$\textbf{do\_statement} \rightarrow 'do',\textbf{compound\_statement},'while','(',\textbf{expression},')',';' \tag{62}$$
$$\textbf{var} \rightarrow IDENT \tag{63}$$
$$|IDENT,'[',\textbf{expression\_list},']' \tag{64}$$
$$\textbf{expression\_list} \rightarrow \textbf{expression} \tag{65}$$
$$|\textbf{expression\_list},',',\textbf{expression} \tag{66}$$
$$\textbf{expression} \rightarrow \textbf{var},':=',\textbf{expression} \tag{67}$$
$$|\textbf{simple\_expr} \tag{68}$$
$$\textbf{simple\_expr} \rightarrow \textbf{additive\_expr} \tag{69}$$
$$|\textbf{additive\_expr},\textbf{OPR},\textbf{additive\_expr} \tag{70}$$
$$|\textbf{additive\_expr},\textbf{SINGLEOPR} \tag{71}$$
$$|\textbf{SINGLEOPR},\textbf{additive\_expr} \tag{72}$$
$$\textbf{SINGLEOPR} \rightarrow '++'|'--'|'!' \tag{73}$$
$$\textbf{OPR} \rightarrow '=='|'!='|'<'|'<='|'>'|'>='|'\&\&'|'||'|'\wedge\wedge'|'<<'|'>>' \tag{74}$$
$$\tag{75}$$

4

$$\text{additive\_expr} \rightarrow \text{term} \tag{76}$$
$$| \text{additive\_expr}, \text{PLUSMINUS}, \text{term} \tag{77}$$
$$\text{PLUSMINUS} \rightarrow' +' |' -' \tag{78}$$
$$\text{term} \rightarrow \text{factor} \tag{79}$$
$$| \text{term}, \text{TIMESDIVIDE}, \text{factor} \tag{80}$$
$$\text{TIMESDEVIDE} \rightarrow' *' |'/'|'\%' \tag{81}$$
$$\text{factor} \rightarrow'(', \text{expression},')' \tag{82}$$
$$| \text{var} \tag{83}$$
$$| INTEGER \tag{84}$$
$$| REAL \tag{85}$$
$$| STRING \tag{86}$$
$$| BOOL \tag{87}$$
$$| CHAR \tag{88}$$
$$| YAJU \tag{89}$$
$$\text{yarimasu\_stat} \rightarrow' yarimasune',';' \tag{90}$$
$$\tag{91}$$

This language should follow this grammar, detailed development of every modules will be mentioned below.

## 3.2 Environment

This project is developed on Ubuntu 18.04 64-bit, using **make** and corresponding **Makefile** to construct. External tools needed are: **Bison**, **Flex**, **VsCode**, **git**.

## 3.3 Modules

This part contain main modules that is to be implemented in this compiler. Including not only basic functionality, but also some bonus functionality. Items with * are bonus modules.

| Module Name | Brief Description |
|---|---|
| Variable store and load | Basic functionality |
| *Constant store and load | Support constant identifiers |
| *Multi-type supporting | Support integer, float, string, char and boolean |
| *Implicit type converting | Convert integer to float if necessary |
| read and write | Basic input and output, supporting multiple types |
| Arithmetic operation | Basic arithmetic operation including +, -, *, /, % |
| Logic operation | Basic logic operation including ==, !=, etc. |
| Instant number in instruction | Essential modules for multiple types supporting |
| Expression | Complex, mixed type expression |
| *Unary operator | Support ++, - -, ! |
| Basic condition statement | If-else statement |
| Basic loop statement | Do-while, while statement |
| *Advanced condition statement | Switch-case-default statement |
| *Advanced loop statement | For statement |
| *N-dimension array | Support theoretically unlimited dimension array |
| *Break/Continue | Support break/continue in for, do-while, while, switch, etc. |
| Error processing | Reporting Syntax and Semantic errors. |
| Magic identifiers | 114514, 1919810, yarimasune, etc. |

### 3.4 Hardware

| Item | Model |
|---|---|
| CPU | Intel Xeon E5-2699v3@2.30GHz(18C36T) |
| Main Board | ASUS ROG Rampage V Extreme |
| RAM | Corsair DDR4 2133@15-15-36-50 64GB |
| GPU | Nvidia Geforce RTX 2080Ti 11GB $\times$ 2 |
| Hard Disk | Intel 750 NVMe SSD 1.2TB $\times$ 2 |
| OS | Ubuntu 18.04 LTS 64-bit |

# 4 Details of modules developing

## 4.1 Data Stack

To support multiple types, the structure of data stack is different with origin version, **All data are stored in Binary format, and specified by pointer**.

```
1  enum data_type {
2          integer ,
3          real ,
4          single_char ,
5          boolean ,
6          str ,
7  };
8
9  struct data_stack {
10         enum data_type  t;
11         byte            val[STRING_LEN]; // STRING_LEN is defined as Macro
12 };
```

When specifying the data, using pointer in .y source: ***var = \*([type]\*)val***.

## 4.2 Symbol Table

To support multiple types and array, the structure of symbol table is different with origin version, following is detailed structure:

```
1  enum object {
2          constant_int ,   constant_real ,
3          constant_bool ,  constant_string ,
4          constant_char ,
5          variable_int ,   variable_real ,
6          variable_bool ,  variable_string ,
7          variable_char ,
8          constant_int_array ,     variable_int_array ,
9          constant_real_array ,    variable_real_array ,
10         constant_char_array ,    variable_char_array ,
11         constant_bool_array ,    variable_bool_array ,
12         constant_string_array ,  variable_string_array ,
13         function ,
14 };
```

Type **function** is not used in current version, may be function will be added in following release. Expect it!

```
1  struct symbol_table {
2          char            name[ID_NAME_LEN];
3          enum object     kind;
4          int             addr;
5          byte            val[STRING_LEN];
6          int             init_or_nor;
7          int             array_size;
8          int             array_const_or_not;
9          int             array_dim[MAX_ARR_DIM];
10 };
```

## 4.3   ISA

This section mainly describe all technical details of the instruction set, including meaning, usage, etc. All instruction are in following format:

$$[\textbf{operation}], [\textbf{opran1}], [\textbf{opran2}]$$

[**operation**] include **lit**, **opr**, **lod**, **sto**, **cal**, **ini**, **jmp**, **jpc**, **off**. [**opran1**] specify the type of [**opran2**] in some operation, corresponding to following table:

| Value of [**opran1**] | Type of [**opran2**] | Identifier in language |
|:---:|:---:|:---:|
| 2 | Intel Integer | **integer** |
| 3 | Real number | **real** |
| 4 | C-like string | **string** |
| 5 | Bool val | **boolean** |
| 6 | Single char | **char** |

### 4.3.1   lit

This instruction is used to load instant number to the top of data stack, leading to increment of stack top. This instruct support all type in this language. Usage:

- **lit, 2, 1919**

- **lit, 5, false**

- **lit, 4, "LvYingZheNiuBi"**

- **lit, 3, 114.514**

This instruction will be used in instant number expression, variable declaration, etc.

### 4.3.2   opr

This instruction is the most used instruction in this compiler, accounting for all of arithmetic and logic operation and some other operation, listed in the table:

| Value of [opran2] | Operation |
|---|---|
| 0 | return from function, will be developed in following release |
| 1 | Negative the stack top, support only integer and real |
| 2 | Binary operator + |
| 3 | Binary operator - |
| 4 | Binary operator * |
| 5 | Binary operator / |
| 6 | Binary operator % |
| 7 | Binary operator == |
| 8 | Binary operator != |
| 9 | Binary operator < |
| 10 | Binary operator <= |
| 11 | Binary operator > |
| 12 | Binary operator >= |
| 13 | Binary operator && |
| 14 | Binary operator \|\| |
| 15 | Binary operator ∧∧ |
| 16 | Single operator ! |
| 17 | **Reserved for ++** |
| 18 | **Reserved for - -** |
| 19 | Output the top of the stack, type specified by [opran1] |
| 20 | Input a value, put it on the top of the stack |
| 21 | **Reserved for <<** |
| 22 | **Reserved for >>** |
| 23 | Pop an element from the stack |
| 24 | Binary operator ==, but will not pop value from the stack Used for switch-case |

### 4.3.3  lod

Load a variable or constant from symbol table to the top of the stack.

### 4.3.4  sto

Store the top of the stack to an identifier.

### 4.3.5  cal

Calling function, currently unused.

### 4.3.6  ini

Initialize a space to store data. Top of the stack will increase by the opran 2 of this instruction.

### 4.3.7  jmp

Jump to the instruction of opran 2.

### 4.3.8  jpc

Jump to the instruction of opran 2 if the top of the stack is false.

### 4.3.9  off

This instruction is especially used for array accessing. This instruction calculate
**array.dimension[0:dim-1] * stack.value[0:opran 1 - 1] + stack.value[opran 1]** to be the offset of array accessing. Because all expression value will be calculated run-time, so this result can only be back patched to the instruction run-time, the next step of this instruction is back patch to **opran2** of the next **lod/sto** instruction.

## 4.4 Variable store and load

- In parsing phase: Variable type can be determined when parsing, and should be return along with the unterminal with their property for further code generating job.

- In declaration phase: When the variable is declared, it will be written into the symbol table with an address, if using $var := [VALUE]$ to initialize the variable, the $init\_or\_not$ in the symbol table will be set to **true**. Using **read()** or := to assign value will also set this flag. When calling a variable with **false** flag, compiler will raise an error.

- In using phase: It should be noted that when accessing a variable, the stack top should increase, or fatal data error will be caused. Whether the variable is used duplicated or not will be checked in both declaration and using phase. In this phase, **lod, [type], [address]** will be used in calling, and **sto, [type], [address]** will be used in storing.

## 4.5 Constant store and load

## 4.6 Multi-type supporting

## 4.7 read and write

## 4.8 Arithmetic operation

## 4.9 Logic operation

## 4.10 Expression

## 4.11 Unary operator

## 4.12 Basic condition statement

## 4.13 Basic loop statement

## 4.14 Advanced condition statement

## 4.15 Advanced loop statement

## 4.16 N-dimension array

## 4.17 Break/Continue

## 4.18 Error Processing