X0-Compiler Design Document

Liu, Altair @ ilovehanhan1120@hotmail.com

November 29, 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 ^_^.

1.3 Remarks

Usage:

```
Ubuntu>$ git clone http://github.com/SubjectNoi/X0-Compiler
Ubuntu>$ cd X0-Compiler
Ubuntu>$ make
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.

```
main {
        integer i, j, flag, cnt := 0;
        for (i := 2; i != 101; i++) {
                flag := 0;
                for (j := 2; j != i; j++) {
                        if (i % j == 0) {
                                flag := 1;
                                break;
                        }
                }
                if (flag == 0) {
                        write(i);
                        cnt++;
                }
        }
        write("There're:");
        write(cnt);
        write("Primes.");
}
```

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:

$$\begin{aligned}
\textbf{program} &\rightarrow 'main', \{, \textbf{statement\_list}, \} && (1)\\
\textbf{statement\_list} &\rightarrow \textbf{statement\_list}, \textbf{statement} && (2)\\
&|\textbf{statment} && (3)\\
&|\epsilon && (4)\\
\textbf{statement} &\rightarrow \textbf{expression\_list} && (5)\\
&|\textbf{if\_statement} && (6)\\
&|\textbf{while\_statement} && (7)\\
&|\textbf{read\_statement} && (8)\\
&|\textbf{switch\_statement} && (9)\\
&|\textbf{case\_stat} && (10)\\
&|\textbf{write\_statement} && (11)\\
&|\textbf{compound\_statement} && (12)\\
&|\textbf{for\_statement} && (13)\\
&|\textbf{do\_statement} && (14)\\
&|\textbf{declaration\_list} && (15)\\
&|\textbf{continue\_stat} && (16)\\
&|\textbf{break\_stat} && (17)\\
&|\textbf{yarimasu\_stat} && (18)\\
&|\epsilon && (19)\\
\textbf{declaration\_list} &\rightarrow \textbf{declaration\_list}, \textbf{declaration\_stat} && (20)\\
&|\textbf{declaration\_stat} && (21)\\
&|\epsilon && (22)\\
\textbf{declaration\_stat} &\rightarrow \textbf{typeenum}, \textbf{identlist}, ';' && (23)\\
&|\textbf{typeenum}, \textbf{identarraylist} && (24)\\
&|'const', \textbf{typeenum}, \textbf{identlist}, \textbf{SEMICOLONSTAT} && (25)\\
&|'const', \textbf{typeenum}, \textbf{identarraylist} && (26)\\
\textbf{identlist} &\rightarrow \textbf{identdef} && (27)\\
&|\textbf{identlist}, ',', \textbf{identdef} && (28)\\
&|\epsilon && (29)\\
\textbf{identdef} &\rightarrow IDENT && (30)\\
&|IDENT, ':=', \textbf{factor} && (31)\\
& && (32)
\end{aligned}$$

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

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

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

bool	*break*	*call*	*case*	*char*
const	*continue*	*default*	*do*	*else*
exit	*false*	*for*	*integer*	*if*
main	*real*	*repeat*	*return*	*string*
switch	*true*	*until*	*while*	*write*

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	Detail
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**.

```
enum data_type {
        integer,
        real,
        single_char,
        boolean,
        str,
};

struct data_stack {
        enum data_type  t;
        byte            val[STRING_LEN]; // STRING_LEN is defined as Macro
};
```

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:

```
enum object {
        constant_int,   constant_real,
        constant_bool,  constant_string,
        constant_char,
        variable_int,   variable_real,
        variable_bool,  variable_string,
        variable_char,
        constant_int_array,     variable_int_array,
        constant_real_array,    variable_real_array,
        constant_char_array,    variable_char_array,
        constant_bool_array,    variable_bool_array,
        constant_string_array,  variable_string_array,
        function,
};
```

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

```
struct symbol_table {
        char            name[ID_NAME_LEN];
        enum object     kind;
        int             addr;
        byte            val[STRING_LEN];
        int             init_or_nor;
        int             array_size;
        int             array_const_or_not;
        int             array_dim[MAX_ARR_DIM];
};
```

4.3 ISA

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

[operation], **[opran1]**, **[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	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	Unary operator !
17	Bit wise &
18	Bit wise |
19	Output the top of the stack, type specified by [**opran1**]
20	Input a value, put it on the top of the stack
21	Binary operator >>
22	Binary operator <<
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 un-terminal 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

- In parsing phase, compiler will check if constant is initialized, if not, compiler will raise an error: ***Constant require initialization***. Also, the type can be determined in this phase and should be return along with the un-terminal for further code generating job.
- In declaration phase: It should be noted that constant will not occupy the stack when un-used, it will only occupy if is loaded by the instruction **lod, [type], [address]**. And declaration of a constant will not generate **sto, [type], [address]** since the value of constant can be determined in paring phase.
- In using phase: Be careful, anything tend to change the value of constant in the symbol table will raise error including **read()**, :=, etc.

4.6 Multi-type supporting

- To support multiple type, the value of all variables and constants are store in **byte** format, which means all data will be converted into **byte array** to store in the symbol table, stack, and when using these variables and constants, parser will convert them from **byte array** into their origin type through **opran 1**, which is generated ***by the information provided by the un-terminal***, so it's very important to maintain the type information in parsing stage.

- Since the compiler also support instant number, so the instruction should support multiple type instant number too, which means the structure of the instruction is different, to be specific, the **opran 2** is also **byte array** type to support instant number. The instruction load an instant number is **lit, [type], [value]**, the assignment of **opran 2** of this instruction is different with others, listed as following:

```
byte opran[];
if (inst.fct == lit) {
        memcpy((void*)(code[pc].opr), (const void*)opran, STRING_LEN);
}
else {
        memcpy((void*)(code[pc].opr), (const void*)&opran, STRING_LEN);
}
```

When using the opran of an instruction, it will be the same as following format besides string opran:

```
int     opran_int                = *(int*)       &code[pc].val;
float   opran_float              = *(float*)     &code[pc].val;
char    opran_char               = *(char*)      &code[pc].val;
bool    opran_int                = *(int*)       &code[pc].val == 1;
string  opran_string             =                code[pc].val;
```

- This compiler support implicit type convert from **integer** to **real**, **integer** to **char**, etc. Mixed type arithmetic operation is also supported by adding type of every element in the data stack.

4.7 read and write

In this compiler, all **expression** can be written to the console, but only non-constant **var** can be read from the console. When calling **write()**, **opr, [type], 19** will be generated, and when calling **read()**, **opr, [type], 20** will be generated. **[type]** can be specified by the property of **expression** and **var**.

4.8 Arithmetic operation

There are 3 categories of arithmetic operator:

- Binary operator and will pop value from the stack.
- Binary operator but will not pop value from the stack (only for **switch-case** statement).
- Unary operator.

All the operator will simply generate an instruction: **opr, [type], [opran]**.

4.9 Logic operation

There are 3 categories of logic operator:

- Binary operator and will pop value from the stack.
- Binary operator but will not pop value from the stack (only for **switch-case** statement).
- Unary operator.

All the operator will simply generate an instruction: **opr, [type], [opran]**. The relationship of value of **opran 2** with operation is in table in section 4.3.2.

4.10 Expression

Expression contains 2 parts: assignment statement and arithmetic and logic statement.

- Assignment statement: when assigning value to a variable, a **sto, [type], [value]** will be generated. And the value of assignment statement is set to be zero. So, the compiler doesn't support assignment like $a := b := 10$, which will be fixed in future release.
- Arithmetic and Logic statement: since there exist priority issue in arithmetic and logic operation, so, priority is set fixed in this compiler with 6 level.

Operator	Priority	Associativity
()	0	N/A
$*, /, \%$	1	Right-associative
$+, -$	2	Right-associative
$<<, >>, \|, \&$	3	Right-associative
$== / \neq, < / \leq, > / \geq, \&\&/\|\|/\wedge\wedge$	4	Right-associative
$++ / --. \ !$	5	N/A

factor is the minimal part of an expression, it can be expression surrounded by a pair of (), a variable or a constant, or just instant value, in these situations, different IR will be generated:

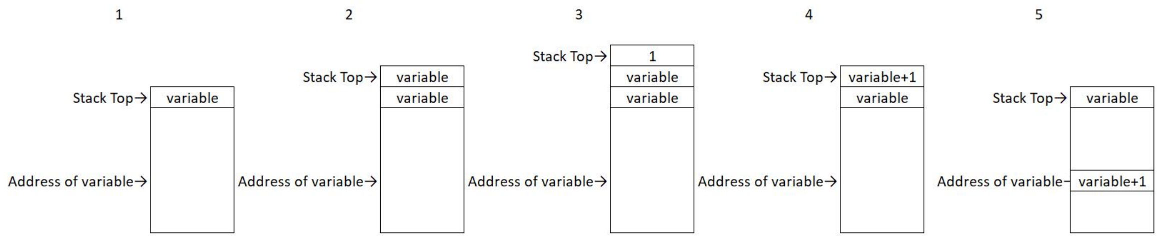
Situation	IR
(expressiopn)	Parse **expression** recursively
variable	**lod, [type], [address_data_stack]**
constant	**lit, [type], [value_in_symbol_table]**
Instant value	**lit, [type], [value_parsed_by_lex]**

After having generated the code of **factor**, the code of all operation can be generated simply in **term** and **additive_expr** as **opr, [type], [opran]**.

4.11 Unary operator

There are 2 solution to solve unary operator, one is to design specific instruction for $++/--$, however, I used following method for $variable++/--$, $++/--variable$ is similar:

- Since when calling $++/--$, variable is called first, so the top of the stack has already contained a copy of variable.
- Load one more copy to the top of the stack by using **lod, [type], [address_data_stack]**.
- Load instant number 1 to the top of the stack by using **lod, 2, 1**.
- Do the operation by using **opr, 2,** $+/-$
- Store new value to the address of the variable by using **sto, [type], [address_data_stack]**.



4.12 Basic condition statement

Basic condition statement stands for **if-else** statement. Since there may be recursive if-else, while, etc. statements in an **if-else**, so the back-patch address should be recorded by the un-terminal in the grammar rather than some global variable. Detailed level condition will be discussed in section 4.17. For **if-else** itself, first, it will generate a **jpc, 0, [destination_address]** since there must be one condition, if **else** exists, **else** will generate an extra **jmp, 0, [destination_address]** to jump through the statement should not be executed. The destination address will be back patched after finishing parsing the whole **if-else** statement. To be specific, the end of **if-else** will be back patched into the destination address.

4.13 Basic loop statement

Basic loop statement stands for **do-while, while** statements. Also, complex level condition will be discussed in section 4.17.

- For **while** statement, first, after the expression, parser will generate a **jpc, 0, [destination_address]**, this destination address will be back patched with the end of whole **while** statement. Then, after parsing the main body of the **while**, parser will generate a **jmp, 0, [begin_address _of_expression_in_while]** to start a new round of condition comparing.
- For **do-while**, it is very similar with **while** statement, just put the **jpc, 0, [destination_address]** before the former **jmp** in **while** statement. And back patch it with the end of whole **do-while** statement.

4.14 Advanced condition statement

Advanced condition statement stands for **switch-case-default** statement. Since there will be many comparing in this statement, it will be complex to use existing binary condition operators since these operators will pop result from the stack top which means the value should be calculated again. So, I implement new binary condition operators but will not pop the value from the stack top for **switch-case-default** statement. Using these operators, parser will generate a **opr, [type], ==(not pop stack top)** to compare the case and a **jpc, 0, [next_case_address]**, to back patch the address of next case, back patch list as the property of un-terminal is implemented with following structure:

- Structure of back patch list as the property of un-terminal. When using it, noted that a memory space should be dispatched to it by ***malloc()***.

```
struct bp_list {
        int     case_start;
        int     case_end;
};
%union {
        char    *ident;
        int     number;
        char    *text;
        char    single_char;
        int     flag;
        double  realnumber;
        struct  bp_list *bp;
}
%type <bp>      case_stat default_statement
```

- Since there will be **break** in it, the end of the whole **switch-case-default** should be back patched to the address of every **break** statements.

4.15 Advanced loop statement

Advanced loop statement stands for **for (**$E1$**;** $E2$**;** $E3$**)** S statement. To implement this, following position should be maintained: **start of** $E2$**, start of** S**, start of** $E3$**, end of** S by additional un-terminal in the grammar. There are following steps:

- After parsing the condition expression $E2$, a **jpc, 0,** [**destination_address**] will be generate if $E2 \neq \epsilon$. And a **jmp, 0,** [**start_of_**S] will be generated to find the entrance of S.
- After parsing S, the iteration variable should be modified, so, a **jmp, 0,** [**start_of_**$E3$] will be generated to find the entrance of $E3$.
- After modifying the iteration variable, condition should be check again, so, a **jmp, 0,** [**start_of_**$E2$] will be generated to check again.

4.16 N-dimension array

It should be noted that, no matter the dimension of the array, when accessing the array, the value of the expression can only be determined run-time, so, back patching is also used for array accessing. The new instruction **off** is used in this process. In the symbol table, *dimension* will be maintained for the array. When accessing the array, parser will generate a **off,** [**number_of_dimension**]**,** [**base_address**] first, base address is also recorded in the symbol table. At run time, after calculated all expression in array accessing list, **off** will calculate:

$$(\sum array.dimension[0:n-1] \times stack[top - number\_of\_dimension : top-1]) + array.dimension[n] \tag{92}$$

This will be the offset of accessing the array, last, add this offset to the base address, and back patch this result to next **lod/sto** since next **lod/sto** must be array accessing instruction. This back patch will be done ***RUN TIME***.

4.17 Break/Continue

Since there may be multiple level of **for, while, if, switch**, etc. So, level should be recorded when implement **break/continue**. Every single **break/continue** will only generate a **jmp, 0,** [**destination_address**], and this destination address will be back patched in the statement call **break/continue**.

- Generate **jmp, 0, 0** at every **break/continue**.
- Record the **level** and address of **break/continue** to be back patched.
- After parsing **for, while, if, switch**, back patch the end of them to the address of **break/continue**.

Since the level is implemented, when processing $i+1$ level, it will follow the order which means when a new **break/-continue** is encountered, former **break/continue** in former **for, while, if, switch** has already been processed, so there will not be problem of data warp. After processing the $i+1$ level, i level will be processed, and the address of $i+1$ and i is stored in different location. The whole process is like a stack.

4.18 Error Processing

- This compiler can't raise multiple syntax error at one time, may be it will be implemented in late release.
- Multiple semantic error will be raised, including:

Semantic Error
Duplicated variable defined.
Constant without initialized.
Undefined variable.
Trying to change constant.
Incompatible type.
Different type between boolean operator.
Operators incompatible.
⋮

5 Maintaining the software

5.1 Maintaining Personnel

This software will mainly be maintained by the developer **Liu, Altair**, he can be reached by ilovehanhan1120@hotmail.com, altair.liu@sap.com.

5.2 Maintenance Object

- Developer will maintain the modules that the bugs are already known.
- User of this software will raise report about the bugs.
- Optimizing and reconstructing of the software will also be conducted if developer is in a good mood, expect it.

5.3 Maintenance Type

- Bug Fixing.
- Code Reconstructing.

5.4 Extension and Future imagination

Following modules may be developed if **Liu, Altair** is in a good mood.

- UI with debugger and run-time stack.
- function with parameter and return value.
- class, struct
- $\Leftarrow$ To Be Continue.