CLIP - A Crytographic Language with Irritating Parentheses

Author:
Wei Duan wd2214@columbia.edu
Yi-Hsiu Chen
yc2796@columbia.edu

Instructor:
Prof. Stephen A. Edwards

August 16, 2013

Contents

1 Introduction

This manual describes CLIP, which is a programming language designed specifically for cryptographers. The purpose of CLIP is to provide users with an efficient and simple programming language to manipulate cryptographic operations, create innovative cryptographic algorithms and implement existing cryptographic protocols. As inheriting the mathematical nature of cryptography, CLIP is designed as a functional language. Its syntax is inspired by the classical functional language Lisp, but with more special types and operations to implement cryptographic algorithms. Essentially, CLIP allows cryptographers to build programs with more convenience than many other popular programming languages.
There are two common and important operations in cryptography. One is number theory related calculation, in which big numbers are especially vital due to security concerns. Another is bits manipulation which usually occurrs in the symmetry cryptosystem. CLIP is designed to facilitate both kinds of operations. For the big number calculation, we include the GMP library of C language, which has descent performance in big number calculation and random number generation. For bits manipulation, efficiency is not the main concern in our language, instead, we emphasize on the way to effectively express algorithms. In fact, fast encryption/decryption are usually realized from hardware end.

2 Language Tutorial

2.1 Installation and Usage

For first time users of CLIP, you need to install the GMP library. Simply executing "gmp.sh" in the main directory can accomplish GMP installation. The next step is to type "make", it will produce two executable files "clip" and "clipc".
"clip" is used to generate the C++ files while "clipc" can be used to generate executable files from .clip files in directory.
Here is the Hello World program in CLIP, which is extremely simple.

Listing 1: hello_world.clip

```
"Hello World!"
```

The output is

Listing 2: hello_worldi.clip

```
"Hello World!"
```

The Hello World program consists of only one single expression. CLIP will print out the expression in the program automatically.

2.2 Data Declaration and Types

To bind a value to a variable, we can use the keyword **defvar**. When binding a variable, the type of the variable should be indicated explicitly.
There are three basic types in CLIP, integer, bit sequence and string. Their keywords are `int`, `bit#` and `string` respectively.
Here are some examples of variable binding:

```
defvar i:int = 314;
defvar i:int = 0xA3;
defvar i:int = 0b1011010;
defvar b:bit#7 = 'b0100010;
defvar b:bit#8 = 'x5B;
defvar s:string = "I am a string";
```

In CLIP, the integer has an unlimited precision, which means you can assign large numbers such as a thousand-digit integer to a variable if needed. To represent a bit sequence, we add a single quote in front of the digit sequence to indicate it. Notably, the above examples show that integer can also be expressed in Hexadecimal and Binary. The bit sequence can also be expressed in Hexadecimal.

There is another derived data type, vector, in CLIP. Vector is a sequence of elements of same type. **defvar** could also be used to bind vectors. The declaration of a vector is
`type[index-1][index-2]..[index-n]`
Here are some examples of binding vector.

```
defvar i:int[2] = {3 4};
defvar i:int[2][3] = {{1 2 3} {4 5 6}};
defvar b:bit#2[3][2] = {{'11 '10} {'11 '01} {'01 '00'}};
defvar s:string[1] = {"I am a string"};
```

Notably, the format of vector is to use curly brackets to separate elements from different dimensions. For example, `{3 4}` means that it is a vector with two integer elements. While `{{1 2 3} {4 5 6}}` is a vector, which has two vector elements `{1 2 3}` and `{4 5 6}`, and within each vector there are three integer elements.

2.3 Functions

The syntax of function call in CLIP is adopted from Lisp. A pair of parentheses should surround the fucntion call. In every pair of parentheses, the first expression is an identifier bound to a function, or a function call which returns a function, such as lambda function.
Let's try some simple arithmetic function now.

```
(+ 4 (* 2 3) (mod 11 4))
```

The result is the same with $4+(2*3)+(11\%4)$, which is the form we are familiar with. Therefore, the output of this program is

```
13
```

Here is an example of **lambda** function.

```
((lambda <x:int> (* x x)) 5)
```

The **lambda** defines a function which has one integer x as its parameter, and returns the square of x. The whole **lambda** expression returns a square function. The outer most parentheses apply the lambda function to integer 5, so the output should be

```
25
```

There are plenty of built-in functions in CLIP, including arithmetic, bit sequence and vector manipulation. For the complete list and usages of them, please refer to the language reference manual in section 3. However, three important built-in functions would be explained below.

1. **let**

   Sometimes it is convenient to have a local variable, but **defvar** is used to bind only global variables. Hence we introduce **let** here. The last argument of **let** function is the expression to be returned. Remaining arguments are used for binding, so there can be as many arguments as needed in **let** function.

```
(let <x:int (* 5 5)> <y:int (/ 30 5)> <z:int 3> (+ x y z))
```

   The above expression will return the evaluation result of `(+ x y z)` where `x`, `y`, `z` are defined in angle brackets. The output of the program is

```
34
```

2. **map**

   The **map** function is of great use in terms of vectors. It takes two arguments, with the first one being a function and the second one being a vector. **map** applies the first argument to each individual element in the vector. The type of return value of map is always a vector, and it depends on the type of the return value of its first argument.

```
(map int-of-bits {'01 '10 '11})
```

   The above expression will apply the function **int-of-bits** to `'01`, `'10`, and `'11` respectively. The output of the program is

```
{1 2 3}
```

3. **reduce**

   The **reduce** function also takes in two arguments, with the first one being a function and the second one being a vector. Yet, it is different from **map**. **reduce** applies the first argument to the first and second elements in the vector, get a partial result and applies the first argument to the partial result and the third element in the vector. It does so repeatedly till the last element in the vector.

```
(reduce + {10 11 12})
```

   The above expression will apply the function `+` first to `10` and `11` to get a partial result, which is `21`, then apply `+` to the `21` and `12`. The output of the program is

```
33
```

2.4 If and Else

`if` is the only control flow in CLIP. It takes in three arguments. The first argument would be evaluated, and if it is true, the second argument will be evaluated and returned, otherwise, the third argument will be evaluated and returned.

```
(if (eq 2 3) 2 3)
```

The expression `(eq 2 3)` will be first evaluated. Since it is false, the third argument will be evaluated and returned, which is `3`.
The output of the program is

```
3
```

2.5 Function Definition

To bind a function to a variable, we use the keyword **defun**. The returned value type of the function as well as all the input argument types should also be provided.
Here is an example of a simple function definition:

```
defun f:int a:int =(+ a 2);
```

It means that a function named `f` is defined, with only one integer as the input argument. The function `f` takes in an integer and returns a value that is the sum of that integer and `2`.

2.6 Compound Examples

2.6.1 GCD

Let's start from the basic well-known algorithm greatest common divisor in number theory.
The gcd algorithm can be recursively defined. It could be written in CLIP as follows:

```
defun gcd:int a:int b:int=
(if (eq a b)
    a
    (if (greater a b)
        (gcd (- a b) b)
        (gcd a (- b a))));

(gcd 24 18)
(gcd 55 34)
```

The algorithm first test whether `a` equals `b`. If it is true, returns `a`. Otherwise, it evaluates the third expression, which is another `if` statement. The sub `if` statement will recursively cals `gcd` depending on the result of evaluating `(greater a b)`.
The result of the program is

```
6
1
```

2.6.2 SHA-3

SHA-3 is a cryptographic hash function designed by Guido Bertoni, Joan Daemen, Michal Peeters, and Gilles Van Assche. We use part of this algorithm to demonstrate the ability of CLIP to manipulation bit sequences.
In one round block permutation, there are five sub-routines need to be applied. $\theta, \rho, \pi, \chi, \iota$ We will show how to implement the first four functions and explain more details of θ function
Before defining and implementing the function, we should fixiate the size of blocks. In SHA-3, a block size is $5 \times 5 \times 2^l$. Here, we follow the convention and set l as 6, so the block size is $5 \times 5 \times 64$.
The concise formula of those functions are

1. $\theta : b[i][j][k] \oplus = parity(a[0..4][j-1][k]) \oplus parity(a[0..4][j+1][k-1])$
2. $\rho : b[i][j][k] = b[i][j][k-(t+1)(t+2)/2])$ where

$$\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} 3 & 2 \\ 1 & 0 \end{bmatrix}^t \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

3. $\pi : b[j][2i + 3j][k] = b[i][j][k]$

4. $\chi : b[i][j][k] \oplus = \neg b[i][j + 1][k] \wedge b[i][j + 2][k]$

For a input block matrix as below

	$\overbrace{1010\cdots10}^{64\text{ bits}}$	$0000\cdots11$	$0101\cdots10$	$1110\cdots11$	$0110\cdots00$
	$0101\cdots01$	$1001\cdots01$	$0101\cdots00$	$1111\cdots11$	$0111\cdots11$
	$0000\cdots00$	$1111\cdots11$	$0100\cdots01$	$0100\cdots11$	$0011\cdots11$
	$1000\cdots11$	$0111\cdots01$	$1001\cdots11$	$0000\cdots00$	$1101\cdots10$
$\oplus$	$1111\cdots00$	$0010\cdots10$	$0100\cdots11$	$0000\cdots00$	$1010\cdots10$
	$-\ -\ -\ -\ -$	$-\ -\ -\ -\ -$	$-\ -\ -\ -\ -$	$-\ -\ -\ -\ -$	$-\ -\ -\ -\ -$
	$1000\cdots00$	$0001\cdots10$	$1001\cdots11$	$0101\cdots11$	$0011\cdots00$
$>>> 64$	$0011\cdots00$	$1000\cdots00$	$0001\cdots10$	$1001\cdots11$	$0101\cdots11$
$<<< 63$	$0000\cdots00$	$0100\cdots11$	$1010\cdots11$	$1001\cdots10$	$0100\cdots00$

The first row of bit sequence beneath the dash line is the xor result of first five rows. The second row beneath the dash line is the 64 bits right-rotation result of the first row. The last row beneath the dash line is the 63 bits left-shift result of the first row.
Finally, the return value of theta is the xor result of the last two rows to each of the five rows in input. Below is the code of the above algorithm for theta function and other three functions.

```
defun theta:bit#64[5][5] block:bit#64[5][5] =
    (let <col-par:bit#320 (reduce ^ (map merge block))>
        (map (lambda <row:bit#320>
                (group (^ row (>>> col-par 64) (<<< col-par 63)) 64))
            (map merge block)));

defun rho:bit#64[5][5] block:bit#64[5][5] =
    (make-vector <bit#64[5][5]> (>>> block[@1][@2] rho-off-set[@1][@2]));

defun pi:bit#64[5][5] block:bit#64[5][5] =
    (make-vector <bit#64[5][5]> block[@2][(mod (+ (* 2 @1) (* 3 @2)) 5)] );

defun chi:bit#64[5][5] block:bit#64[5][5] =
    (make-vector <bit#64[5][5]>
        (^ block[@1][@2]
        (& block[@1][(mod (+ @2 1) 5)] block[@1][(mod (+ @2 2) 5)])));
```

The first step is to bind the variable `col-par` to a value, and the value corresponds to the first row after the dash line. In the lambda function, it rotate rightwards and leftwards of `col-par`, which corresponds to the last two line in previous diagram.
The reason to use `(map merge block)` is we want to xor every row, but we cannot xor them directly since it is a vector not bit sequence. We have to merge them first.

2.6.3 RSA

RSA is an algorithm for public-key cryptography invented by Ron Rivest, Adi Shamir and Leonard Adleman. Here we simply describe the algorithm without explaining what is public-key cryptosystem or the hardness of breaking it.

1. Settings:

   To build the RSA cryptosystem, first we have to generate two different large prime number p, q. The larger are those prime numbers, the harder is to break the system. Typically, they are 1024 bits long. Then we can generate other parameter as follows.

$$\begin{aligned} n &= p \cdot q \\ N &= \phi(n) = (p-1)(q-1) \\ e &= 65537 \\ d &= e^{-1} \bmod N \end{aligned}$$

2. Encryption:

   Suppose the message m is a number less then n, then the cipher text c is

$$m = c^e \bmod n$$

3. Decryption:

   The decrypt algorithm is

$$c = m^d \bmod n$$

The implementation of RSA in CLIP is as follow

```
~~> calculate base^exp % m  <~~

defun pow-mod:int base:int exp:int m:int =
    (if (eq exp 0)
        1
        (let <half:int (pow-mod base (/ exp 2) m)>
            (if (eq 0 (mod exp 2))
                (mod (* half half) m)
                (mod (* base half half) m))));

~~> define the variables <~~

defvar p:int = (next-prime (int-of-bits (rand 1024)));
defvar q:int = (next-prime (int-of-bits (rand 1024)));
defvar n:int = (* p q);          ~~ public
defvar phi-n:int = (* (- p 1) (- q 1));
defvar e:int = 65537;            ~~ public
defvar d:int = (inverse 65537 phi-n);

"p:"
p
"q:"
q

defun rsa-enc:int message:int n:int =
    (pow-mod message e n);

defun rsa-dec:int cipher:int n:int =
```

```
    (pow-mod cipher d n);

defvar message:int = 42;
defvar cipher:int = (rsa-enc message n);

"cipher text:"
(rsa-enc message n)
"plain text:"
(rsa-dec cipher n)
```

Since the exponential in the power calculation could be a 1024-digit number, we cannot use the built-in pow function and module it directly. That will cause overflow for sure even for most modern computer. So we define the new power function, which use the divide and conquer algorithm. Also module `m` after every multiplication.
For the random number generation, since there is no direct function to generate random integer or prime. We generate random bit sequences first, convert it to an integer, then find the next prime number next to it.
Here is a possible output:

```
p:
107275923850495671805219157657077815844071951797036707087201253081122063110963
265836553370723793863410907146511758902260897195424790699588544829233290078582
665770266537556513900614530312921394658260422980503508424834039639556500889865
846949589582627427120263749928714157551119741594434496876891051338571452521
q:
152186273275783221706304463642636143860325089927553623249730644924286319540314
626420534952149053625141150265844108029912416151919107989849149396232955711990
089065680578016102167107967290969961256231940949502261716238712776387868325O7
937681474361063057818887840021354681548439184487683206871438792047580941303
cipher text:
719723449389518559033392772679818355960382360222703807113409783248008100814973
414228957608133936331400593964030893039190790266035912126912019622374243640730
578279721715413557275007343022708398647608411484634026407935961898987582157851
213400619133045423966055113668536605549773672976905598335341962033059644931890
708607140482265543837830484944075755023710680784302194219320754729171325549717
356793721386927107810445934304604924054629632970516190705955318998293005149223
345359071758154344448950520497705201787632492799440186515749029093716751329087
306436265710327668401779130997867176842405154283417597828815472211199O
plain text:
42
```

Since the prime numbers are generated randomly, it's unlikely to have the repeated cipher text. But the point is, the plain text should always be 42 after decryption.

3 Language Manual

3.1 Lexical Convention

CLIP has five classes of tokens: identifiers, keywords, constants, functions and other separators. Blanks, horizontal and vertical tabs, newlines and comments are ignored except as they separate tokens. Some whitespace is required to separate otherwise adjacent identifiers, keywords and constants.

3.1.1 Comments

As many languages, CLIP has two kinds of comment. A single line comment begin with two consecutive tilde `~~`, and ends till the end of the line. A multi-line comment should start with `~~>` and end with `<~~`. Note that comments cannot be nested.

3.1.2 Identifiers

An identifier in CLIP starts with a letter and is optionally followed by letters, numbers or hyphen. But there cannot be two consecutive hyphens in an identifier.
e.g. `this-is-a-valid-identifier` is a valid identifier, but `this-is--not` is not.

3.1.3 Keywords

1. **Binding**: The following keywords are used for binding the identifier to variables and functions respectively.

   defvar, **defun**

2. **Flow Control**: There is only one keyword used for flow control. Though its syntax is like function call, it works differently to normal function call.

   `if`

3. **Types**: Each basic type, including integer, bit sequence and string, has a keyword the program uses for declarations.

   `int`, `bit#`, `string`

4. **Built-in Functions**: The following keywords are reserved for built-in functions

   - **let**, **set**, **map**, **reduce**, **lambda**
   - **mod**, **pow**, **inverse**, **is−prime**, **next−prime**
   - **zero**, **rand**, **pad**, **flip**, **flip−bit**
   - **less**, **greater**, **leq**, **geq**, **eq**, **neq**, **and**, **or**, **not**
   - **group**, **merge**, **make−vector**, **transpose**
   - **int−of−bits**, **bits−of−int**, **string−of−bits**, **bits−of−string**

3.1.4 Constant

Inside the expression of CLIP, there can be integer, bit and string constant. Also, the compound type – vector, also can be expressed as constant.

1. **Integer**

   CLIP provides three ways to express an integer constant.

   (a) Decimal: The expression of it is simply a sequence of digits. The leading zero will be ignored automatically.

   (b) Binary: The expression is a sequence of `0` and `1` leading with `0b`.

   (c) Hexadecimal: The expression is a sequence of `0-9` and `A-F` leading with `0x`.

2. **Bit sequence**

   A bit sequence is a sequence of `0` and `1` lead by a single quote character.

   e.g. `'01000101`

3. **String**

   A string constant is as in C, surrounded by the double quotes.

4. **Vector**

   The only compound type in CLIP is called vector. To express a vector constant, use the curly bracket to enclose a sequence of objects. Each object is separated by at least one space. The vector can be nested to form a higher dimension vector.

   ```
   {{1 2 3} {5 7 9}}
   ```

5. **Other punctuations**

   There are 4 other punctuations used in CLIP and serve as different purpose

   (a) `;`: The semicolon is used to end the binding. After binding variables or function, there should be a semicolon. For an outer most expression, if there is a semicolon after it, then the expression will only be executed but not printed out its evaluation.

   (b) `[]`: The square bracket is used for expressing a vector type or accessing a vector element. For example, `int[5][8][4]` means the type of 5 by 8 by 4 integer vector, and `a[3]` means fetching the third element from vector `a`

   (c) `()`: As in lisp, the parentheses are used in function call. The first expression appear in it is a functor.

   (d) `<>`: The angle bracket are reserved for special purpose. The only place it appears is in the functions **let**, **lambda** and **make−vector**.

3.2 Type

3.2.1 Basic Data Types

In CLIP, there are three non-function basic data types

- `int`: An integer with unlimited precision.
- `bit#n`: A bit sequence consist of n bits.
- `string`: A series of characters.

To access those type, simply use the identifiers which is binded to the values.
Also note that, specially, the type `bit#1` is also used as boolean value, which is the return type of logic functions such as **and**, **or**. `'0` means false and `'1` means true.

3.2.2 Derived Data Type

The only derived data type is vector. To declare a vector type, we use a sequence of bracket as `type[index-1][index-2]..[index-n]`
This suggests a vector, with `type` being the type of elements consist of vector, and `index-1[index-2]..[index-n]` being a series of positive whole numbers enclosed in square brackets,

indicates the dimensionality of the vector. The element in the vector should be in the same type. The number of square brackets suggests the dimension of vector. For examples, `bit#2[2][3]` is a vector with two dimensions, and the elements in the vector are all 2-bit sequences. It can contain a constant like `{{'01 '00 '10} {'00 '11 '10}}`
The following structure:
`vector-name[index-1][index-2]..[index-n]`
is used to access a particular element in a vector. While `index-1` indicates that the element is at position `index-1` in the first dimension, `index-2` indicates that the element is at position `index-2` in the second dimension, etc. All index start from 0, so suppose the above vector example is bind to `v`, then `v[1][1]` returns `'11` The vector access allows not only the atom element, but also a sub vector. Namely, `v[1]` returns `{'00 '11 '10}`.

3.3 Syntax

3.3.1 Program Structure

A program is consist of a sequence of function binding, variable binding and expressions. They can appear in the program in any sequence.
When execute, the variable is declared orderly, which means in the former declaration of either functions of variables, they cannot use the variables that declared after them. But the recursion of function is valid.
Be cautious about that the same identifier cannot be binded twice.

3.3.2 Expression

An expression can be a constant, variable or a function evaluation. The syntax of function call is lisp-like as follows. `(functor argument-1 argument-2 .. argument-n)` The arguments, even the functor can also be an expression.
The function is evaluated in the applicative order. And arguments are evaluated orderly. Namely, first evaluates `argument-1`, then `argument-2`, etc.

3.3.3 Binding Variables

In general, when we define a non-function data type, we use the keyword **defvar** as following
defvar `identifier:type = value;`
The scope of the identifier defined by defvar begins directly after declaration and exists throughout the whole program. Here are some examples of binding global variables to different types of values

```
defvar n:int = 17
defvar b:bit#3 = '001
defvar s:string = "ThisIsAString"
defvar v:bit#5[2][3] = {{'00000 '01110 '11010} {'10101 '11000 '11011}}
```

3.3.4 Binding Functions

The form of function declaration is
defun `functor-name:return-type argument-1:type-1 .. argument-n:type-n = expression;`

The type of the expression must match with the return type.
Here is an example of calculating Fibonacci number

```
defun Fib:int n:int =
(if (< n 1)
    1
    (+ (Fib (- n 1)) (Fib (- n 2)));
```

3.3.5 Output

There is no explicit output function in CLIP. After evaluating the outer most expression, the result will be printed out in its own line automatically. If you don't want the result be printed, then you can add semicolon after the expression.
For example, after running below code

```
(+ 1 1)
(+ 1 2);
(+ 1 3)
```

We will get

```
2
4
```

3.3.6 Control Flow

In clip, the only flow control instruction `if`. The syntax is
`(if argument-1 argument-2 argument-3)`
`argument-1` should be an expression which return true or false (`'1` or `'0`). The second and the third argument should return the same types.
The syntax looks like function call. However, the crucial difference is, after evaluating the first argument, the program will execute `argument-2` or `argument-3` depends on the result. But not executes them both, then decide to output which result. That is an important concept, lets take the look on the recursive factorial example

```
defun Fib:int n:int =
(if (< n 2)
    1
    (* n (Fib (- n 1))));
```

If the if statement always evaluate both `argument-2` and `argument-3`. Then the program will never stop. Because even `Fib(1)` should simply return 1, the program still tries to evaluate the last argument, which makes it stuck in the infinite loop.

3.4 Type Conversion

The built-in functions provide methods to convert the data between bit sequences and integers, as well as bit sequences and strings. They are

- (**bits-of-int** `integer length`)
- (**int-of-bits** `bit-sequence`)
- (**bits-of-string** `a-string length`)

- (**string−of−bits** bit-sequence)

The length parameter indicates how many bits the output should have. If it is too long, then the function will pad zeros. If it is too short, then the function will truncate automatically.

3.5 Built-in Function

We categorize built-in functions into five categories

1. Arithmetic functions: Used to handle the arithmetic of integer with unlimited precision.
2. Logical functions: Logical related functions.
3. Bit sequence functions: Used in manipulation of bit sequences.
4. Vector functions: Functions operated on the vectors.
5. Conversion functions: Convert data between types
6. Miscellaneous clip functions: Includes **let**, **lambda**, **map**, **set** First three are common elements in functional language.

3.5.1 Arithmetic Functions

1. +

   (+ integer-1 integer-2 .. integer-n)

   The function + can take in integers as arguments, and returned the sum of all the arguments. + can have many arguments.

   Example program:

   ```
   (+ 1 2 3 4 5)
   ```

   Output:

   ```
   15
   ```

2. *

   (* integer-1 integer-2 .. integer-n)

   The function * can take in integers as arguments, and returned the sum of all the arguments. * can have many arguments.

   Example program:

   ```
   (* 1 2 3 4 5)
   ```

   Output:

   ```
   120
   ```

3. –

   (– integer-1 integer-2)

   The function – can only take in two integers as arguments. The result of the first argument being subtracted from the second will be returned.

   Example program:

   ```
   (– 10 2)
   ```

   Output:

   ```
   8
   ```

4. /

   (/ integer-1 integer-2)

   The function / can only take in two integers as arguments. The result of the first argument being divided by the second will be returned.

   Example program:

   ```
   (/ 10 2)
   ```

   Output:

   ```
   5
   ```

5. **mod**

   (**mod** integer-1 integer-2)

   The function **mod** can only take in two integers as arguments. The return value is the remainder of division of the first argument by the second one.

   Example program:

   ```
   (mod 10 3)
   ```

   Output:

   ```
   1
   ```

6. **pow**

   (**pow** integer-1 integer-2)

   The function **pow** takes in two integer arguments. The return value is the exponential of the first argument to the second argument.

   Example program:

   ```
   (pow 4 5)
   ```

Output:

```
1024
```

7. **inverse** (**inverse** integer-1 integer-2)

   The function **inverse** takes in two integer arguments. It returns a value x such that integer-1 * x = 1 (**mod** integer-2)

   Example program:

```
(inverse 11 23)
```

   Output:

```
21
```

8. **is−prime**

   (**is−prime** integer)

   The function **is−prime** takes in an integer as input, and return '1 if the input is a prime, or '0 otherwise.

   Example program:

```
(is−prime 23)
(is−prime 27)
```

   Output:

```
'1
'0
```

9. **next−prime**

   (**next−prime** integer)

   The function **next−prime** takes in an integer as input, and return the nearest prime that is greater than it.

   Example program:

```
(next−prime 20)
(next−prime 97)
```

   Output:

```
23
101
```

3.5.2 Bit Sequence Functions

1. **zero**

   (**zero** integer)

   The function **zero** takes in an integer and returns a bit-sequence, whose length is indicated by the input.

   Example program:

   ```
   (zero 3)
   ```

   Output:

   ```
   '000
   ```

2. **rand**

   (**rand** integer)

   The function **rand** takes in an integer and returns a random bit-sequence. The length of the bit-sequence is indicated by the input.

   Example program:

   ```
   (rand 3)
   ```

   Output:

   ```
   '010
   ```

3. **pad**

   (**pad** bit-sequence integer)

   The function **pad** takes in two arguments. The first one is a bit-sequence and the second one is an integer. **pad** extends the length of the first argument to be as indicated by the second argument by adding a number of '0.

   Example program:

   ```
   (pad '1111 6)
   ```

   Output:

   ```
   '111100
   ```

4. **flip**

   (**flip** bit-sequence)

   The function **flip** converts the input bit-sequence into another bit-sequence whose leftmost bit is the same as the rightmost bit in the input, second leftmost bit is the same as the second rightmost bit in the input, etc.

   Example program:

```
(flip '01011)
```

Output:

```
'10100
```

5. **flip−bit**

   (**flip−bit** bit-sequence integer)

   The function **flip−bit** takes in two arguments. The first one is a bit-sequence and the second one is an integer. **flip−bit** converts the nth bit into its reverse, where n is indicated by the second argument.

   Example program:

```
(flip-bit '1001011 2)
```

   Output:

```
'1011011
```

3.5.3 Logical Functions

1. **less**

   (**less** integer-1 integer-2)

   It takes two integers as input argument and compares them. If the first argument is less than the second one, the return value is '1, otherwise '0.

   Example program:

```
(less 4 7)
(less 7 4)
(less 7 7)
```

   Output:

```
'1
'0
'0
```

2. **greater**

   (**greater** integer-1 integer-2)

   It takes two integers as input argument and compares them. If the first argument is greater than the second one, the return value is '1, otherwise '0.

   Example program:

```
(greater 4 7)
(greater 7 4)
(greater 7 7)
```

Output:

```
'0
'1
'0
```

3. **leq**

   (**leq** integer-1 integer-2)

   It takes two integers as input argument and compares them. If the first argument is no greater than the second one, the return value is `'1`, otherwise `'0`.

   Example program:

```
(leq 4 7)
(leq 7 4)
(leq 7 7)
```

   Output:

```
'1
'0
'1
```

4. **geq**

   (**geq** integer-1 integer-2)

   It takes two integers as input argument and compares them. If the first argument is no less than the second one, the return value is `'1`, otherwise `'0`.

   Example program:

```
(geq 4 7)
(geq 7 4)
(geq 7 7)
```

   Output:

```
'0
'1
'1
```

5. **eq**

   `(eq expression-1 expression-2)`

   It takes two expressions with same types as input argument and compares them. If the first argument is equal with the second one, the return value is `'1`, otherwise `'0`.

   Example program:

```
(eq 4 7)
(eq 7 4)
(eq 7 7)
```

   Output:

```
'0
'0
'1
```

6. **neq**

   `(neq integer-1 integer-2)`

   It takes two expressions with same types as input argument and compares them. If the first argument is equal with the second one, the return value is `'0`, otherwise `'1`.

   Example program:

```
(neq 4 7)
(neq 7 4)
(neq 7 7)
```

   Output:

```
'1
'1
'0
```

7. **and**

   `(and bit-1 bit-2 .. bit-n)`

   It do the "and" operation on the arbitrary number of bits.

   Example program:

```
(and '0 '0 '1 '0)
(and '0 '0 '0 '0)
(and '1 '1 '1 '1)
```

   Output:

```
'0
'0
'1
```

8. **or**

   `(or bit-1 bit-2 .. bit-n)`

   It do the "or" operation on the arbitrary number of bits.

   Example program:

```
(or '0 '0 '1 '0)
(or '0 '0 '0 '0)
(or '1 '1 '1 '1)
```

   Output:

```
'1
'0
'1
```

9. **not**

   `(not bit)`

   This function simply flips `bit`.

   Example program:

```
(not '0)
(not '1)
```

   Output:

```
'1
'0
```

3.5.4 Vector Functions

1. **group**

   `(group bit-sequence integer)`

   It slices a `bit-sequence` into several bit sequences and groups them into a vector, where the length of output bit sequence is indicated by integer. The integer should be a divider of the length of `bit-sequence`.

   Example program:

```
(group '100110110111010 5)
```

   Output:

```
{'10011 '01101 '11010}
```

2. **merge**

   (**merge** vector-of-bit-seq)

   It is an inverse function of group, it concatenate all the bit sequences in vector-of-bit-seq and returns it.

   Example program:

```
(merge {'11010 '01101 '11010})
```

   Output:

```
'1101001101110 10
```

3. **make−vector**

   (**make−vector** <vector-type> expression)

   The **make−vector** function creates an arbitrary dimension vector and returns it. The first argument vector-type indicates the return type. The second argument expression generally would be a function indicates the specific elements in the vector. `@i` suggests the i-th index of the vector. The return type of expression should be a basic type of vector-type. In sum up, suppose the vector-type is `int[3][5]`, then `@1` and `@2` can appear in the expression as identifier. For the return vector, the position `[i][j]` will be the value of expression insert `@1` as `i` and `@2` as `j`

   Example program:

```
(make−vector <int[3][3]> (* (+ 1 @1) @2))
```

   Output:

```
{{0 1 2}
{0 2 4}
{0 3 6}}
```

4. **transpose**

   (**transpose** matrix)

   As suggested by the name of the function, it will transpose a matrix, which is a two dimension vector.

   Example program:

```
(transpose {{'00 '01 '10} {'11 '11 '11}})
```

   Output:

```
{{'00 '11}
{'01 '11}
{'10 '11}}
```

3.5.5 Conversion Functions

1. **int−of−bits**

   (**int−of−bits** bit-sequence)

   It converts a bit-sequence into an integer in the big endian sense. The length of bit-sequence can be arbitrary.

   Example program:

```
(int−of−bits '010011)
```

   Output:

```
19
```

2. **bits−of−int**

   (**bits−of−int** integer-1 integer-2)

   It converts the integer integer-2 into the bit sequence, the first integer integer-1 indicates how long the return bit length is.

   Example program:

```
(bits−of−int 12 341)
(bits−of−int 13 341)
```

   Output:

```
'xAA8
'0000101010101
```

3. **string−of−bits**

   (**string−of−bits** bit-sequence)

   It converts bit-sequence into a string by transforming every eight bits into a character in big endian sense. If the bits is not a multiple of eight, then it pads 0 after it.

   The length of bit-sequence can be arbitrary.

   Example program:

```
(string−of−bits '01001001)
```

   Output:

```
I
```

4. **bits−of−string**

   (bit-of-string integer a-string)

   It converts string into a bit sequence in the big endian sense. Every character in the string will be converted into 8 bit sequence.

   Example program:

```
(bits−of−string "HI")
```

Output:

```
'x9212
```

3.5.6 Miscellaneous Clip Functions

- **let**

  (**let** <id-1:type-1 value-1> .. <id-n:type-n value-n> expression)

  let evaluates expression and binds the value-i to the id-i specified in the angle brackets. The return value is the evaluation of the last expression. It can contain the global variables and the identifiers binded in the **let** function. If the identifier is already defined as a global variable or outer **let** function, then it will use the one in nearest layer.

  Example program:

```
defvar m:int=1;
(let <n:int 2> <m:int (+ 3 4)>
    (+ n m))
```

  Output:

```
9
```

- **lambda**

  (**lambda** <id-1:type-1 id-2:type-2 .. id-n:type-n> expression)

  The **lambda** function returns a function which can be used as an argument of another function. Same as in **let** function, we can use variables id-1 to id-n in expression. If the name is conflicted with outer variable, it will mask the outer one.

  Example program:

```
((lambda <x:int> (* x x x)) (+ 1 2))
```

  Output:

```
27
```

- **map**

  (**map** function a-vector)

  The **map** function allows you to apply a function on each element of a vector. If the vector has multiple dimensions, the **map** will treat the second outer most layer vector as an element and apply the function on that vector.

  Example program:

```
(map (lambda <x:int> (* x x x)) {1 2 3 4})
(map merge {{'1111 '1100 '0011} {'1010 '0101 '0000}} )
```

Output:

```
{1 8 27 64}
{'xC3F 'x0A5}
```

- **reduce**

  (**reduce** function a-vector)

  The **reduce** function also takes in a function and a vector as arguments. Here the function must take two argument in the same types and return the same type data. Similar to **map** function, the vector can be more than one dimension. On such cases, it only handles the outer most layer vector. **reduce** takes the first two elements from the second input argument and applies function to it.Then a partial result will be generated. The function will again apples to the partial result and the third element in the second input argument. It does so repeatedly till the end of the vector is reached. The final output of function is the output of **reduce** function.

  Example program:

```
(reduce ^ {'101 '101 '010 '010 '111})
```

  Output:

```
'111
```

3.6 Scope

3.6.1 Global variables

In general, CLIP does not allow declare the variable or function in an expression, so the scope of a variable is simple to define, usually. Except in several special functions, **lambda let make−vector**. In the function declaration, if the local variable has the same name to some global variable, then it will mask the global variable. Therefore the output of following example will be

```
defvar a:int = 3;

defun f:int b:int =
    (+ a b);

(f 4)

defun g:int a:int =
    (+ a a);

(g 4)
```

```
7
8
```

Other than function definition, there are three functions, which can has its own local variables.

1. (**lambda** <var-1:type-1 var-2:type-2 .. var-n:type-n> expression)

   The scope of var-1, var-2 ... var-n are in the expression

2. (**let** <var-1:type-1 exp-1> <var-2:type-2 exp-2> .. <var-n:type-n exp-n> exp)

   The scope of variable var-1, var-2 ... var-n are also in the exp

3. (**make−vector** <type> expression)

   In the expression of `make-vecotr`, the variable `@1`, `@2`, .. , `@d` has the scopes in expression where `d` is the dimension of type.

3.7 Grammar

In the below grammar, the string with capital letters, like `ID`, `INT`, are the token given by the scanner, also the terminal symbol. Some notation also adopts the regular expression representations. We use square bracket to group expressions in regular expression, so `\[ \]` means the square bracket symbol itself.

```
program:
    [defvar | defun | expression SEMICOLON?]*

defvar:
    DEFVAR ID:type = expression SEMICOLON

defun:
    DEFUN (ID:type)+ = expression SEMICOLON

expression:
    constant
    ID
    ID dimensions
    AT_NUM
    (LET [<ID:type> expression]* expression)
    (LAMBDA <[ID:type]*>)
    (MAKE-VECTOR <type> expression)
    (expression+)

type:
    sig_type
    sig_type
    const_dimensions

sig_type:
    INT
    BITS
    STR

const_dimension_list:
    [\[constant\]]*

dimension_list:
    [\[expression\]]*

constant:
```

```
    constant_int
    onstant_bits
    STRING_LIT
    vector

vector:
    {[expression ]+}

constant_int:
    INTEGER
    0b BINARY
    0x HEX

constant_bits:
    ' BIT_BINARY
    'x BIT_HEX
```

4 Project Plan

With the relatively small number of members in our team, we employed a collaborative strategy to accomplish the project. The strategy basically involves the following components:

- The use of a version control infrastructure.

  We created a git repository on Dropbox and branched it on our own directory, because in comparison with the open source environment of github, dropbox can maintain files with higher security. Hence, all source code in this project was produced on the local git repository and being pushed to dropbox and only shared with the other team member.

- Communicate with each other on a regular basis.

  Group meeting was held about two times a week. General problems such as module design, implementation rules and specification details would be discussed. In addition, scheduled meeting with TA was also organized every week, during which progress updates would be reported.

- Keep up a stable working pace

  Time is especially limited in summer terms. Therefore, we pushed ourselves to start thinking about the project from the first week. Then, progress of each week may differ, we tried to focuses all efforts on the project, and managed to accomplish a module per week.

4.1 Planning process

Starting from the very beginning, we decided to design a language that could assist programming in cryptography. We spent the first week to design the features and functionality of this language, and then set up main goals for each phase. Throughout the whole project, we referred to our planned schedule and kept up proceed along with it. Meanwhile, we also retained certain flexibility so that the short-term goal may experience shifting due under some unexpected circumstances.

4.2 Specification process

Initially, in the language reference manual, we specified the propoerty of CLIP being a functional language, and the features include the ability of manipulating big numbers and bits sequences. Targeting at cryptographers, we also planned to create built-in functions which support those most common cryptographic operations. We decided to borrow the syntax of LISP which can provide programmers a rather clear distinction of the scope of each function. Yet for big number calculation, we did not determine its implementation until Professor Edwards recommended us to utilize GMP, which is a very powerful c library for big number operations. Along development, some of the features claimed in the LRM may be adjusted, for example, the format of comments. Moreover, some features may be added, such as to define CLIP as a strongly-typed language and require provision of type for each function(its return value), variable(both local and global) and arguments.

4.3 Development process

We planned to implement CLIP by translating it to C++, then, compiled using g++. Follow the stage of architecture, we immediately turned to the real practice and started working on scanner, then moved on to parser, then semantic check, and finally the translator. The first two stages were finished within the first half of the summer semester, since they are the fundamental basis for all other stages. We then continued to semantic check and accomplished it in the second week, but it was amended frequently later, particularly when we wrote code for translator.

4.4 Testing process

Testing was performed throughout the whole development process. At the beginning, when scanner and parser were completed, test cases consist of positive and negative instances were created and examined. According to the feedback given by testing, modifications and improvements were made to give lexical part more robustness. Then, semantic check module was established. Again, we test its ability of detecting grammar errors by design a series of complex negative test cases. The result of semantic check was printed out in the format of AST parsing tree, which in the meantime tested previous scanner and parser modules as well. When tranlator modul was built up, we started to feed it with authentic CLIP code, and testing results were print out in the format of C++ language, which is our target language. In addition, regression testing strategy was adopted so that old test cases would always run whenever new features were added to ensure the reliability of our compiler as a whole.

4.5 Programming style

- Ocaml programming:

  Being new to Ocaml, we used the following reference guide below to learn the most basic grammar: http://courses.cms.caltech.edu/cs134/cs134b/book.pdf

- Formatting:

  We mostly follow one of the style described in Caml Programming Guideline http://caml.inria.fr/resources/do

  More specifically, we use 4 spaces as indentation and put the `in` of `let .. in` function at the end of the last line.

- Documentation:

  Comments was written at beginning of functions to specify its functionality.

4.6 Project timeline

Date	Task to accomplish
July 15th	Language proposal complete
July 22nd	Language reference manual complete
July 29th	Scanner and parser complete
Aug 5th	Semantic check complete
Aug 10th	Translator complete
Aug 14th	Testing, debugging
Aug 16th	Final report and presentation

Table 1: Scheduled timeline

4.7 Responsibilities

There is no strict division of responsibilities as we are a small team, where each member should participate in multiple parts depending on the progress of the project.

Yi-Hsiu Chen	Scanner, Parser, AST, Semantic check, C++ library, Translator
Wei Duan	Scanner, AST, C++ library, Translator, Test suites

Table 2: Work distribution

4.8 Software development environment

We had the following programming and development environment:

- Programming language for building compiler : Ocaml version 4.00.1 . Ocamlyacc and Ocamllex extensions were used for compiling the scanner and parser.
- Development environment: We used Sublime Text 2 together with its plugins.

4.9 Project log

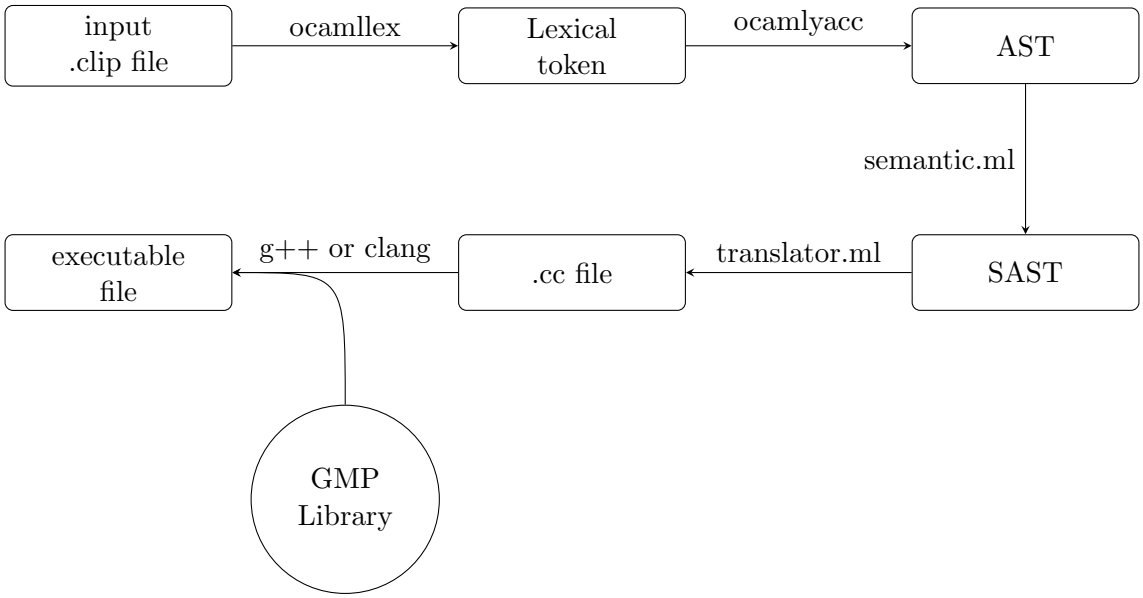
Date	Milestone
July 10th	Language defined
July 15th	Language proposal complete
July 19th	Language features defined
July 22th	Language reference manual complete
July 28th	Scanner and parser complete
July 31th	Test parser complete
Aug 2rd	Testcases designed
Aug 9th	Semantic check complete
Aug 11th	Translator and built-in function library complete
Aug 12th	Debugging and adjusting
Aug 14th	All modules complete
Aug 16th	Final report and Presentation

Table 3: Project log

5 Architectural Design

5.1 The Compiler

The architecture of CLIP compiler mainly consists of four parts: scanner, parser, semantic checker, translator (The last step merely invokes the g++ compiler). All four parts are implemented using OCaml. We also use ocamllex and ocamlyacc to help implementing scanner and parser respectively. The main compiler program is clip.ml, which invoke scanner, parser, semantic checker, translator sequentially. The first step is passing the clip source code to scanner and parser. They will generate the AST, which defines in ast.ml and contains the information about the structure of clip program. In next step, we pass the AST to semantic.ml, which will check the semantic correctness. It mainly check the type consistency. During checking the semantic, it also constructing the SAST, which contains more information than AST. The information help the translator to generate C code easily. The last step of the compiler is to generate compilable C code from SAST. The above tasks are packed into a single executable file "clip" In order to make it more convenient to transform into executable file, we also provide "clipc", which takes .clip file as input and generate executable file as output.



6 Test Plan

6.1 Testing phases

6.1.1 Unit testing(scanner, parser, translator)

In this project, we built up a regression test suite to examine the correctness as well as integrity of CLIP. Each module of CLIP was tested individually by designing both basic and complex test cases and printing out their results respectively. For scanner and parser, we manually input different CLIP programs and printing the AST(.tree) out in the form of a parsing tree. For translator, we tested it by writing various CLIP code and printing them out as C++ code(.cc). Finally, the automation was triggered and to test the final output of each CLIP program by comparing them with reference results(.ref). With the progress of each module being developed and new features being added, the regression test suite was run on a regular basis to ensure that the language was analyzed and operated correctly, and that new adjustments would not damaged previous program behaviors.

6.1.2 Integration testing

In the process starting from scanner to C++ code generation, syntax, lexical conventions and semantics were check by taking advantage of various sets of input: constants, types, variable and functions. The correctly generated code would then be translated to C++ and compile with g++ to verify with their expected output.

- Constants(Literals)

  Integer, bit numbers and strings were tested and verified. Escape sequences could be correctly handled:

  (`\n`, `\t`, `\r`, `"\ "`)

- Types

In CLIP, four types are supported: `int`, `bit#`, `string` and vector. Each was checked with several test cases. For vectors, we also tested their declaration and binding. Testing of those types include negative cases such as assigning an int to a string.

- Variables

  Declaration and bindings of each variable was checked by printing out the result.

- Functions

  We implement nested functions as test cases and refer to the AST tree generated to check if each function could be called in the right order, and if the scope of arguments of each function were corrected handled.

6.1.3 System testing

Several CLIP programs were created and tested. Test cases varies from the most simple Hello World file to the more sophisticated cryptographic algorithms such as RSA. Scope rules and function closure were tested.The test cases could be found in the Test cases section below. In the CLIP to C++ section, we also demonstrate some example programs.

6.2 Testing automation

In order to efficiently run test cases, we created a script called "testall.sh". It takes in all the test cases from a certain directory and compiles them to C++ code.

6.3 Test cases

In the test directory, there is a wrong-cases folder in which negative test cases were stored. All of them are failed in the semantic check stage. Another folder is named right-cases, in which right test cases were stored. For debugging purpose, we gave output files different extension.

File extension	Content
.clip	store source program
.out	store the final result
.ref	store the expected result
.DIFF	store the difference between final result and expected result
.tree	store the AST tree given by parser
.error	store the standar error message

Table 4: Meaning of different file extension

6.4 CLIP to C++

The source language CLIP would be translated into C++ language. Following are a few representative examples of CLIP program along with their target C++ program.

Listing 3: A simple addition example in CLIP

```
(+ 100 200)
```

File	Testing target
test-arith	Test a number of different arithmatic functions
test-binding	Test variable binding and assignment
test-bit	Test built-in functions for bit number manipulation
test-comp	Test built-in functions for integer comparison
test-defun	Test function binding
test-gcd	Test recursion
test-if	Test the execution of built-in function if
test-lambda	Test the execution of built-in function lambda
test-logic	Test built-in function for logic operation
test-map	Test built-in function map
test-misce	Test built-in function zero and int-of-bits
test-mod	Test built-in function mod
test-parser1	Test vector binding
test-parser2	Test built-in function make-vector
test-rsa	Test a cryptographic algorithm RSA
test-shiftrotate	Test built-in function for shift and rotate
test-type	Test different types of variable binding
test-vector	Test vector manipulation

Table 5: Testing suite test cases

Listing 4: The addition example compiled to C++

```
#include <cstdio>
#include <cstdlib>
#include <gmp.h>
#include <library/builtin.cc>
#include <library/dynamic_builtin.cc>
using namespace std;

int main(int argc, char *argv[]){
    //srand( time (NULL) );init();
{
        string id__2 = "100";
        string id__3 = "200";
    string id__1 = add2 (id__2, id__3);
    cout << id__1 << endl;
}
    return 0;
}
```

Listing 5: A recurrsive function gcd in CLIP

```
defun gcd:int a:int b:int=
(if (eq a b)
    a
    (if (greater a b)
        (gcd (- a b) b)
        (gcd a (- b a))));
```

```
(gcd 6 3)
```

Listing 6: The gcd example compiled to C++

```
#include <cstdio>
#include <cstdlib>
#include <gmp.h>
#include <library/builtin.cc>
#include <library/dynamic_builtin.cc>
using namespace std;

string gcd (string a, string b) {
    string id__4;
            string id__8 = a;
            string id__9 = b;
        bitset<1> id__5 = eq__int (id__8, id__9);

    if( id__5 == bitset<1>(1)) {
        string id__6 = a;
        id__4=id__6;
    } else {
        string id__7;
                string id__13 = a;
                string id__14 = b;
            bitset<1> id__10 = greater__ (id__13, id__14);

        if( id__10 == bitset<1>(1)) {
                    string id__20 = a;
                    string id__21 = b;
                string id__19 = subtract (id__20, id__21);
                string id__22 = b;
            string id__11 = gcd (id__19, id__22);
            id__7=id__11;
        } else {
                string id__15 = a;
                    string id__17 = b;
                    string id__18 = a;
                string id__16 = subtract (id__17, id__18);
            string id__12 = gcd (id__15, id__16);
            id__7=id__12;
        }
        id__4=id__7;
    }

    return id__4;
}
int main(int argc, char *argv[]){
    //srand( time (NULL) );init();
{
        string id__2 = "6";
        string id__3 = "3";
    string id__1 = gcd (id__2, id__3);
    cout << id__1 << endl;
```

```
50 }
    return 0;
}
```

6.5 Testing roles

The shell script and the test cases were created by Wei Duan, while Yi-Hsiu Chen helped to implement the cryptographic algorithms.

7 Lessons Learned

- Wei Duan

  In this class, I learned lessons on both technique knowledge and teamwork. From the lectures, I got to understand the achetecture of building up a compiler, the important features for a good programming language as well as how to program in Ocaml. At first, typical PLT concepts such AST seemed a little abstract to me. But when we started to implement our own compiler, I knew the concepts better, and realized that following the knowledged being taught in class truly helped me in practice. In addition, to implement a good programming language, it worth taking more time in design beforehand. A clear big picture about how to implement the compiler would contribute to better development in the end. In our project, the sematic check was initially built in such a way that it did store the information regarding type of variables, arguments and functions. Yet, when we moved into the translator stage, it was hard to intepret CLIP program into C++ programs without those information. Hence, we went back to the semantic check module and made some adjustment. Another crucial thing I learned in this project is that effective communication between team members could be a big plus. We are a small team, with only two people. At the beginning, I was worried that we may produce less than other teams which have three to four people. However, it turned out that sometimes fewer people is not a bad thing. The communication between me and my teammate is very convenient, because we have more time flexibility. What really matters is the meaningful communications among team members.

- Yi-Hsiu Chen

  Doing this project, provide a good opportunity to scrutinize what we have learned in class. Sometimes, I just thought I already understood what Professor was talking about. However, until doing project and really implementing some concepts, then realized that I did not understand throughly, at least not well enough. For instance, about the concept of static scope and dynamic scope, several issues and methods of implementation just popped up when I translated it. Even for the very first step: scanner and parser, I thought I designed the context free grammar of our language perfectly. But it pops out lots of shift/reduce error after ocamlyacc it, which stuck me for an hour. After several correcting, I finally diminished all warnings and knows the mechanism of such algorithm better.

  In the technical part, although OCaml might the useful tool in future tasks, using a functional language forces me to think different about implementing and designing algorithms. Also the pattern matching function is so useful that I just cannot imagine if I have to use another language to implementing a compiler. I also learned two important things or techniques during doing project, version control and shell. Although, at first, using them is painful and makes me frustrated when spending an hour just for fixing a stupid error, every time when git

perfectly merge our modification or testing tens of files in one command. I really appreciate the recommend from Professor.

Another thing I learned from this project is about teammate, even we only have two people, which makes deciding meeting time and communication much easier. There is still lots of method to communicate and collaborate we can improve. For two people, the most challenge part is how to divide the works properly, since we cannot waste any human resource. Honestly, we did not do well at the beginning, we waste lots of time doing the same things or generating a garbage. But afterward, we found out how to split to task, that is building the skeleton of the task at the beginning. Otherwise, the codes created by different people just not mergeable, then you have to abandon one of them.

To sum up, I think the project is absolutely worth doing, even in the summer session. Before doing the project, I though why on earth we have to such complicated thing in such short time. Why can't we just learn more concepts form textbook. Now I realized doing project makes you learn not only lot more practical things, also deeper in some important concepts.

8 Appendix

Listing 7: ast.ml

```
type fun_case =
| Indef
| Fix
| Len_Flex

type c_type =
| Wild_Card of int
| Int
| Bits of int
| String
| Vector of c_type * int list
| Fun of fun_case * c_type * c_type list
| Special of string

type id_with_type = {
    id: string;
    t: c_type;
}

type expr =
| Int_Lit of string
| Bin_Lit of string
| Hex_Lit of string
| Bit_Binary_Lit of string
| Bit_Hex_Lit of string
| String_Lit of string
| Vector_Lit of expr list
| Id of string
| Idd of string * expr list
| Vec_Dimension of int
| Let of let_arg list * expr
| Lambda of id_with_type list * expr
```

```
| Make_Vector of c_type * expr
| Funcall of expr * expr list

and let_arg = id_with_type * expr

type defvar = {
    vname : id_with_type;
    vbody : expr;
}

type defun = {
    fname : id_with_type;
    fargu : id_with_type list;
    fbody : expr;
}

type clip =
| Expr of expr * bool
| Defvar of defvar
| Defun of defun

type program = clip list
```

Listing 8: builtin.ml

```
open Ast
open Sast
open Printf

(* Add builtin functions in varmap which used in semantic analysis *)
let add_builtin_fun varmap =

    let builtinmap =
        VarMap.add "if" (Fun(Fix, Wild_Card(1),
            [Bits(1); Wild_Card(1); Wild_Card(1)])) varmap in

    let builtinmap =
        VarMap.add "+" (Fun(Len_Flex, Int, [Int])) builtinmap in

    let builtinmap =
        VarMap.add "*" (Fun(Len_Flex, Int, [Int])) builtinmap in

    let builtinmap =
        VarMap.add "-" (Fun(Fix, Int, [Int;Int])) builtinmap in

    let builtinmap =
        VarMap.add "/" (Fun(Fix, Int, [Int;Int])) builtinmap in

    let builtinmap =
        VarMap.add "mod" (Fun(Fix, Int, [Int;Int])) builtinmap in

    let builtinmap =
        VarMap.add "pow" (Fun(Fix, Int, [Int;Int])) builtinmap in
```

```
    let builtinmap =
        VarMap.add "inverse" (Fun(Fix, Int, [Int;Int])) builtinmap in

    let builtinmap =
        VarMap.add "and" (Fun(Len_Flex, Bits(1), [Bits(1)])) builtinmap
            in

    let builtinmap =
        VarMap.add "or" (Fun(Len_Flex, Bits(1), [Bits(1)])) builtinmap in

    let builtinmap =
        VarMap.add "not" (Fun(Fix, Bits(1), [Bits(1)])) builtinmap in

    let builtinmap =
        VarMap.add "less" (Fun(Fix, Bits(1), [Int;Int])) builtinmap in

    let builtinmap =
        VarMap.add "greater" (Fun(Fix, Bits(1), [Int;Int])) builtinmap in

    let builtinmap =
        VarMap.add "leq" (Fun(Fix, Bits(1), [Int;Int])) builtinmap in

    let builtinmap =
        VarMap.add "geq" (Fun(Fix, Bits(1), [Int;Int])) builtinmap in

    let builtinmap =
        VarMap.add "eq" (Fun(Fix, Bits(1),
            [Wild_Card(1);Wild_Card(1)])) builtinmap in

    let builtinmap =
        VarMap.add "neq" (Fun(Fix, Bits(1),
            [Wild_Card(1);Wild_Card(1)])) builtinmap in

    let builtinmap =
        VarMap.add "&" (Fun(Len_Flex, Bits(0), [Bits(0)])) builtinmap in

    let builtinmap =
        VarMap.add "|" (Fun(Len_Flex, Bits(0), [Bits(0)])) builtinmap in

    let builtinmap =
        VarMap.add "^" (Fun(Len_Flex, Bits(0), [Bits(0)])) builtinmap in

    let builtinmap =
        VarMap.add "parity" (Fun(Fix, Bits(1), [Bits(0)])) builtinmap in

    let builtinmap =
        VarMap.add "<<" (Fun(Fix, Bits(0), [Bits(0); Int])) builtinmap in

    let builtinmap =
        VarMap.add ">>" (Fun(Fix, Bits(0), [Bits(0);Int])) builtinmap in

    let builtinmap =
        VarMap.add ">>>" (Fun(Fix, Bits(0), [Bits(0);Int])) builtinmap in
```

```
    let builtinmap =
        VarMap.add "<<<" (Fun(Fix, Bits(0), [Bits(0);Int])) builtinmap in

    let builtinmap =
        VarMap.add "flip-bit" (Fun(Fix, Bits(0),
            [Bits(0);Int])) builtinmap in

    let builtinmap =
        VarMap.add "flip" (Fun(Fix, Bits(0), [Bits(0)])) builtinmap in

    let builtinmap =
        VarMap.add "set" (Fun(Fix, Wild_Card(1),
            [Wild_Card(1); Wild_Card(1)])) builtinmap in

    let builtinmap =
        VarMap.add "if" (Fun(Fix, Wild_Card(1),
            [Bits(1); Wild_Card(1); Wild_Card(1)])) builtinmap in

    let builtinmap =
        VarMap.add "group" (Special("group")) builtinmap in

    let builtinmap =
        VarMap.add "merge" (Special("merge")) builtinmap in

    let builtinmap =
        VarMap.add "map" (Special("map")) builtinmap in

    let builtinmap =
        VarMap.add "reduce" (Special("reduce")) builtinmap in

    let builtinmap =
        VarMap.add "transpose" (Special("transpose")) builtinmap in

    let builtinmap =
        VarMap.add "zero" (Special("zero")) builtinmap in

    let builtinmap =
        VarMap.add "rand" (Special("rand")) builtinmap in

    let builtinmap =
        VarMap.add "int-of-bits" (Fun(Fix, Int, [Bits(0)])) builtinmap in

    let builtinmap =
        VarMap.add "string-of-bits"
        (Fun(Fix, String, [Bits(0)])) builtinmap in

    let builtinmap =
        VarMap.add "bits-of-int"
        (Fun(Fix, Bits(0), [Int; Int])) builtinmap in

    let builtinmap =
        VarMap.add "bits-of-string"
        (Fun(Fix, Bits(0), [Int; String])) builtinmap in
```

```
    let builtinmap =
        VarMap.add "pad"
        (Fun(Fix, Bits(0), [Bits(-1); Int])) builtinmap in

    let builtinmap =
        VarMap.add "is-prime"
        (Fun(Fix, Bits(1), [Int])) builtinmap in
    let builtinmap =
        VarMap.add "next-prime"
        (Fun(Fix, Int, [Int])) builtinmap in
    builtinmap
;;


(* generate c code string for the function with dynamic type *)

let gen_if name ts =
    sprintf
"%s if__%s(bitset<1> b, %s x, %s y) {
    if (b == bitset<1>(1))
        return x;
    else
        return y;
}\n" ts name ts ts
;;

let gen_eq name ts =
    sprintf
"bitset<1> eq__%s(%s x, %s y) {
    if (x == y)
        return bitset<1>(1);
    else
        return bitset<1>(0);
}\n" name ts ts
;;

let gen_neq name ts =
    sprintf
"bitset<1> neq__%s(%s x, %s y) {
    if (x == y)
        return bitset<1>(0);
    else
        return bitset<1>(1);
}\n" name ts ts
;;

let rec gen_xor_xors len s =
    if len = 0 then
        s
    else
        let news = sprintf "bs%d ^ %s" len s in
        gen_xor_xors (len-1) news
;;

```

```
let rec gen_xor_args len s bt =
    if len = 0 then
        s
    else
        let news = sprintf "%s bs%d, %s" bt len s in
        gen_xor_args (len-1) news bt
;;

let gen_xor bit_len arg_len =
    let bt = sprintf "bitset<%d>" bit_len in
    let args =
        gen_xor_args (arg_len-1) (sprintf "%s bs%d" bt arg_len) bt in
    let xors =
        gen_xor_xors (arg_len-1) (sprintf "bs%d" arg_len) in
    sprintf "%s xor__%d__%d (%s) {%s    return %s;%s}\n"
    bt bit_len arg_len args "\n" xors "\n"

let rec gen_or_ors len s =
    if len = 0 then
        s
    else
        let news = sprintf "bs%d | %s" len s in
        gen_or_ors (len-1) news;;

let gen_or bit_len arg_len =
    let bt = sprintf "bitset<%d>" bit_len in
    let args =
        gen_xor_args (arg_len-1) (sprintf "%s bs%d" bt arg_len) bt in
    let ors = gen_or_ors (arg_len-1) (sprintf "bs%d" arg_len) in
    sprintf "%s or__%d__%d (%s) {%s    return %s;%s}\n"
    bt bit_len arg_len args "\n" ors "\n"

let rec gen_and_ands len s =
    if len = 0 then
        s
    else
        let news = sprintf "bs%d & %s" len s in
        gen_or_ors (len-1) news;;

let gen_and bit_len arg_len =
    let bt = sprintf "bitset<%d>" bit_len in
    let args =
        gen_xor_args (arg_len-1) (sprintf "%s bs%d" bt arg_len) bt in
    let ands = gen_and_ands (arg_len-1) (sprintf "bs%d" arg_len) in
    sprintf "%s and__%d__%d (%s) {%s    return %s;%s}\n"
    bt bit_len arg_len args "\n" ands "\n"

let gen_merge out_b_len name in_b_len =
    sprintf
"bitset<%d> %s(vector< bitset<%d> > vb) {
    string s = \"\";
    for (int i = 0; i < vb.size(); i++)
        s = s + vb[i].to_string();
    return bitset<%d> (s);
```

```
}\n" out_b_len name in_b_len out_b_len
;;

let gen_group out_b_len name in_b_len =
    sprintf
"vector< bitset<%d> > %s(bitset<%d> b, string ns) {
    int v_len = atoi(ns.c_str());
    v_len = %d / v_len;
    string s = b.to_string();
    vector < bitset<%d> > result;
    result.resize(v_len);
    for (int i = 0; i < v_len; i++) {
        result[i] = bitset<%d>(s.substr(%d*i, %d));
    }
    return result;
}\n"
    out_b_len name in_b_len
    in_b_len
    out_b_len
    out_b_len out_b_len out_b_len
;;

let gen_map out_t_s name in_t_s =
    sprintf
"vector< %s > %s(function<%s (%s)> f, vector< %s > b) {
    vector< %s > result;
    result.resize(b.size());
    for (int i = 0; i < b.size(); i++) {
        result[i] = f(b[i]);
    }
    return result;
}\n"
    out_t_s name out_t_s in_t_s in_t_s
    out_t_s
;;

let gen_reduce out_t_s name in_t_s =
    sprintf
"%s %s(function<%s (%s, %s)> f, vector< %s > bsv) {
    %s result = bsv[0];
    for (int i = 1; i < bsv.size(); i++)
        result = f(result, bsv[i]);
    return result;
}\n"
    out_t_s name out_t_s in_t_s in_t_s in_t_s
    out_t_s
;;

let gen_transpose ts name =
    sprintf
"vector<vector<%s> > %s(vector<vector<%s> > m) {
    int nrow = m.size();
    int ncol = m[0].size();
    vector<vector<%s> > newm;
```

```
    newm.resize(ncol);
    for (int i = 0; i < ncol; i++)
        newm[i].resize(nrow);
    for (int i = 0; i < nrow; i++)
        for (int j = 0; j < ncol; j++)
            newm[j][i] = m[i][j];
    return newm;
}\n" ts name ts ts
;;

let gen_rotate_r bit_len =
    sprintf
"bitset<%d> rotate_r__%d(bitset<%d> bs, string ns) {
    int n = atoi(ns.c_str());
    int _n = n %% %d;
    string s = bs.to_string();
    s = s + s;
    bitset<%d> dbs = bitset<%d>(s);
    dbs <<= %d - _n;
    dbs >>= %d;
    s = dbs.to_string();
    s = s.substr(%d, %d);
    bitset<%d> result = bitset<%d>(s);
    return result;
}\n"
    bit_len bit_len bit_len
    bit_len
    (bit_len*2) (bit_len*2)
    bit_len
    bit_len
    bit_len bit_len
    bit_len bit_len
;;

let gen_rotate_l bit_len =
    sprintf
"bitset<%d> rotate_l__%d(bitset<%d> bs, string ns) {
    int n = atoi(ns.c_str());
    int _n = n %% %d;
    string s = bs.to_string();
    s = s + s;
    bitset<%d> dbs = bitset<%d>(s);
    dbs <<= _n;
    dbs >>= %d;
    s = dbs.to_string();
    s = s.substr(%d, %d);
    bitset<%d> result = bitset<%d>(s);
    return result;
}\n"
    bit_len bit_len bit_len
    bit_len
    (bit_len*2) (bit_len*2)
    bit_len
    bit_len bit_len
```

```
    bit_len bit_len
;;

let gen_shift_r bit_len =
    sprintf
"bitset<%d> shift_r__%d(bitset<%d> bs, string ns) {
    int n = atoi(ns.c_str());
    return bs >> n;
}\n" bit_len bit_len bit_len
;;

let gen_shift_l bit_len =
    sprintf
"bitset<%d> shift_l__%d(bitset<%d> bs, string ns) {
    int n = atoi(ns.c_str());
    return bs << n;
}\n" bit_len bit_len bit_len
;;

let gen_flip_bit bit_len =
    sprintf
"bitset<%d> flip_bit__%d(bitset<%d> bs, string ns) {
    int n = atoi(ns.c_str());
    bs[%d-n-1] = !bs[%d-n-1];
    return bs;
}\n" bit_len bit_len bit_len bit_len bit_len
;;

let gen_flip bit_len =
        sprintf
"bitset<%d> flip__%d(bitset<%d> bs) {
    return bs.flip();
}\n" bit_len bit_len bit_len

let gen_zero n name =
    sprintf
"bitset<%d> %s (string s) {
    return bitset<%d>(0);
}\n" n name n
;;

let gen_rand n name =
    sprintf
"bitset<%d> %s (string ns) {
    int seed = clock();
    mpz_t a;
    mpz_init(a);
    gmp_randstate_t state;
    gmp_randinit_mt(state);
    string s = \"1\";
    for (int i = 1; i < %d; i++) {
        seed = clock();
        gmp_randseed_ui (state, seed);
        mpz_urandomb(a, state, 1);
```

```
        if (string(mpz_get_str(NULL, 10, a)) == \"0\")
            s = s + \"0\";
        else
            s = s + \"1\";
    }
    return bitset<%d>(s);
}\n" n name n n
;;

let gen_int_of_bits n =
    sprintf
"string int_of_bits__%d (bitset<%d> b) {
    string s = b.to_string();
    char *cp = new char [s.length()];
    strcpy(cp, s.c_str());
    mpz_t n;
    mpz_init(n);
    mpz_set_str(n, cp, 2);
    mpz_get_str(cp, 10, n);
    return string(cp);
}\n" n n
;;

let gen_bits_of_int n =
    sprintf
"bitset<%d> bits_of_int__%d (string a, string ns) {
    int n = atoi(ns.c_str());
    return bitset<%d>(n);
}\n" n n n
;;

let gen_bits_of_string n =
    sprintf
"bitset<%d> bits_of_string__%d(string a, string s) {
    string result = \"\";
    string pad = \"\";
    if (s.length()*8 < %d) {
        string tmp((%d - s.length()*8), '0');
        pad = tmp;
    }
    for (int i = 0; i < s.length(); i++) {
        short c = s.at(i);
        result += (bitset<8>(c)).to_string();
    }
    result += pad;
    return bitset<%d>(result);
}\n" n n n n n
;;

let gen_string_of_bits n =
    sprintf
"string string_of_bits__%d(bitset<%d> bs) {
    string result = \"\";
    string bss = bs.to_string();
```

```
    string pad;
    int pad_len = bss.length() %% 8;
    if (pad_len > 0) {
        pad_len = 8 - pad_len;
        string tmp(pad_len, '0');
        pad = tmp;
    }
    bss += pad;
    for (int i = 0; i < bss.length(); i = i+8) {
        string subbss = bss.substr(i, 8);
        bitset<8> subbs(subbss);
        unsigned int un = subbs.to_ulong();
        int n = un;
        string tmp(1, n);
        result += tmp;
    }
    return result;
}\n" n n
;;


let gen_parity n =
    sprintf
"bitset<1> parity_%d (bitset<%d> bs) {
    return bitset<1>((bs.count()) %% 2);;
}\n" n n
;;

let gen_pad m n =
    sprintf
"bitset<%d> pad__%d__%d(bitset<%d> bs, string ns) {
    string bss = bs.to_string();
    string pad = \"\";
    int pad_len;
    if (%d > %d) {
        pad_len = %d - %d;
        string tmp(pad_len, '0');
        pad = tmp;
    }
    bss += pad;
    return bitset<%d>(bss);
}" m m n n m n m n m

;;
```

Listing 9: parser.mly

```
%{
  open Ast
%}

%token DEFUN DEFVAR
%token INT STR
%token <int> BITS
```

```
%token FUN

%token LET
%token LAMBDA
%token MAKE_VECTOR

%token <string> ID

%token <string> BINARY
%token <string> HEX
%token <string> BIT_BINARY
%token <string> BIT_HEX
%token <string> INTEGER
%token <string> STRING

%token COLON SEMI LPAREN RPAREN LBRACK RBRACK LBRACE RBRACE LANGLE RANGLE
%token ASSIGN
%token <int> VECDIMENSION
%token EOF

%start program
%type <Ast.program> program

%%

program:
| /* nothing */    { [] }
| clip_list        { List.rev $1 }

clip_list:
| clip             { [$1] }
| clip_list clip   { $2 :: $1 }

clip:
| defvar           { $1 }
| defun            { $1 }
| expr             { Expr($1, true) }
| expr SEMI        { Expr($1, false) }

defvar:
| DEFVAR id_with_type ASSIGN expr SEMI    { Defvar ({ vname = $2;
                                                      vbody = $4 } )}

defun:
| DEFUN id_with_type arguments_opt ASSIGN expr SEMI { Defun ({ fname = $2
   ;
                                                              fargu = $3;
                                                              fbody = $5 }
                                                                  )}

expr:
| constant                                { $1 }
| ID                                      { Id($1) }
| ID dimension_list                       { Idd($1, List.rev $2) }
```

```
| VECDIMENSION                                { Vec_Dimension($1) }
| LPAREN LET let_args expr RPAREN             { Let(List.rev $3, $4) }
| LPAREN LAMBDA lambda_arg expr RPAREN       { Lambda($3, $4) }
| LPAREN MAKE_VECTOR LANGLE c_type RANGLE expr RPAREN { Make_Vector($4,
   $6) }
| LPAREN expr expr_opt RPAREN                { Funcall($2, $3) }

lambda_arg:
| LANGLE arguments_opt RANGLE     { $2 }

let_args:
| let_arg                         { [$1] }
| let_args let_arg                { $2 :: $1 }

let_arg:
LANGLE id_with_type expr RANGLE   { ($2, $3) }

expr_opt:
| /* nothing */                   { [] }
| expr_list                       { List.rev $1 }

expr_list:
| expr                            { [$1] }
| expr_list expr                  { $2 :: $1 }

arguments_opt:
| /* nothing */                   { [] }
| argument_list                   { List.rev $1 }

argument_list:
| id_with_type                    { [$1] }
| argument_list id_with_type      { $2 :: $1 }

id_with_type:
| ID COLON c_type                 { { id = $1; t = $3 } }

/* Returns Type(basic_type, dimensions)
   e.g. parse int[8][5][4] as Vector(INT, [8, 5, 4])
              int          as INT
              bits#7[6][5] as Vector(Bits(7), [6, 5])
*/

c_type:
| sig_type                              { $1 }
| sig_type const_dimension_list         { Vector($1, List.rev $2) }
| FUN                                   { Fun(Indef, Wild_Card(0), []) }

sig_type:
| INT                                   { Int }
| BITS                                  { Bits($1) }
| STR                                   { String }

const_dimension_list:
| const_dimension                       { [$1] }
```

```
| const_dimension_list const_dimension  { $2::$1 }

const_dimension:
| LBRACK INTEGER RBRACK              { (int_of_string $2) }

dimension_list:
| dimension                        { [$1] }
| dimension_list dimension         { $2::$1 }

dimension:
| LBRACK expr RBRACK               { $2 }

constant:
| constant_int    { $1 }
| constant_bits   { $1 }
| STRING          { String_Lit($1) }
| vector          { Vector_Lit($1) }

vector:
| LBRACE expr_list RBRACE  { List.rev $2 }

constant_int:
| INTEGER         { Int_Lit($1) }
| BINARY          { Bin_Lit($1) }
| HEX             { Hex_Lit($1) }

constant_bits:
| BIT_BINARY      { Bit_Binary_Lit($1) }
| BIT_HEX         { Bit_Hex_Lit($1) }  /* not yet handled */
```

Listing 10: printast.ml

```
open Ast
open Printf
open Exception

let out = stdout;;

let rec string_indent layer =
if layer > 0 then
    "" ^ string_indent (layer-1)
else
    "";;

let rec string_type = function
| Wild_Card(i) -> "Wild_Card"
| Int -> "int"
| Bits(i) -> "bits#" ^ (string_of_int i)
| String -> "string"
| Vector(t, int_list)
    -> List.fold_left (fun s i -> s ^ "[" ^ (string_of_int i)
        ^ "]") (string_type t) int_list
| Fun(x, y, z) -> "fun"
| Special(s) -> "fun-"^s
```

```
;;

let string_idt idt =
sprintf "%s(%s)" idt.id (string_type idt.t);;

let rec string_vector_help = function
| [] -> ""
| hd::tl -> ", " ^ string_expr 0 hd ^ string_vector_help tl

and string_vector vec =
    "{" ^ string_expr 0 (List.hd vec) ^ string_vector_help (List.tl vec)
       ^ "}"

and string_expr layer exp =
    string_indent layer ^
    match exp with
    | Int_Lit(s) -> s
    | Bin_Lit(s) -> s
    | Hex_Lit(s) -> s
    | Bit_Binary_Lit(s) -> "'" ^ s
    | Bit_Hex_Lit(s) -> "'x" ^ s
    | String_Lit(s) -> "\"" ^ s ^ "\""
    | Vector_Lit(v) -> string_vector v
    | Id(s) -> s
    | Idd(s, v) -> s ^ string_vector v
    | Vec_Dimension(i) -> sprintf "[@]%d" i
    | Lambda(idt_list, exp) -> "[Lambda]"
    | Let(let_arg_list, exp) -> "[Let]"
    (*
    fprintf out "[Let]";
    List.fold_left (fun () () -> ()) ()
    (List.map (fun (idt, exp) ->
    fprintf out "%s = " (string_idt idt); print_expr 0 exp) let_arg_list)
       ;
    fprintf out "\n";
    print_expr (layer+1) exp
    *)
    | Make_Vector(c_type, exp) -> "[Make_Vector]"
    | Funcall(exp, exps) ->
        (List.fold_left
            (fun s1 s2 -> s1 ^ s2)
            ("(" ^ string_expr 0 exp)
            (List.map (fun e -> sprintf " " ^ string_expr (1+layer) e)
               exps))
        ^ ")"
;;

let string_clip c =
match c with
| Expr(exp, _) -> string_expr 0 exp ^ "\n"
| Defvar(defv) -> "Defvar\n"
| Defun(def) -> "Defun\n";;

let rec string_clips p =
```

```
match p with
| [] -> ""
| x::y -> string_clip x ^ string_clips y;;

let print_ast p =
    fprintf out "============= program start =============\n";
    fprintf out "%s" (string_clips p);
    fprintf out "============== program end ==============\n";;
```

Listing 11: printsast.ml

```
open Ast
open Sast
open Semantic
open Printast
open Printf
open Exception

let xout = open_out "sast.txt";;

let rec str_indent layer =
if layer = 1 then
    "  - " ^ str_indent (layer-1)
else if layer > 1 then
    "  " ^ str_indent (layer-1)
else
    "";;

let rec xstr_type = function
| Wild_Card(i) -> sprintf "W<%d>" i
| Int -> "mpz"
| Bits(i) -> sprintf "bs<%d>" i
| String -> "str"
| Vector(t, l) ->
    let dim_part = List.fold_left (fun s i -> sprintf "%s[%i]" s i) "" l
       in
    sprintf "%s%s" (xstr_type t) dim_part
| Fun(fc, t, t_l) ->
    let fcs = begin match fc with
    | Fix -> "fix"
    | Len_Flex -> "flex"
    | Indef -> "indefun" end in
    let in_t_s = begin match t_l with
    | [] -> "none"
    | hd::tl -> List.fold_left
        (fun s t -> s ^ "*" ^ (xstr_type t)) (xstr_type hd) tl end in
    sprintf "%s(%s -> %s)" fcs in_t_s (xstr_type t)
| Special(s) -> s

let rec str_xexpr indent xexpr =
str_indent indent ^
match xexpr with
| Xint_Lit(s) -> sprintf "[mpz] %s" s
| Xbin_Lit(s) -> sprintf "[mpz-b] %s" s
```

```
| Xhex_Lit(s) -> sprintf "[mpz-x] %s" s
| Xbit_Binary_Lit(s, i) -> sprintf "[bs<%d>] %s" i s
| Xbit_Hex_Lit(s, i) -> sprintf "[bs-x<%d>] %s" i s
| Xstring_Lit(s) -> sprintf "[str]%s" s
| Xvector_Lit(xexpr_l) ->
    let s_type = xstr_type (Semantic.get_type (Xvector_Lit(xexpr_l))) in
    let sexpr_l = List.map (fun xe -> str_xexpr 0 xe) xexpr_l in
    let content = List.fold_left (fun s1 s2 -> s1 ^ ", " ^ s2)
        (List.hd sexpr_l) (List.tl sexpr_l) in
    sprintf "%s{%s}" s_type content
| Xid(s, t) -> sprintf "[ID-%s] %s" (xstr_type t) s
| Xidd(s, xexpr_l, t) -> sprintf "[ID-%s] %s%s" (xstr_type t) s
    (List.fold_left (fun s xe -> sprintf "%s[%s]" s (str_xexpr 0 xe)) ""
       xexpr_l)
| Xvec_Dimension(i) -> sprintf "[index] %d" i
| Xfuncall(fn, xe_l, t, _) ->
    List.fold_left (fun s1 s2 -> s1 ^ s2)
    (sprintf "[fun-%s] %s" (xstr_type t) fn)
    (List.map (fun xe -> sprintf "\n%s" (str_xexpr (indent+1) xe)) xe_l)
| Xmake_Vector(t, xexpr) ->
    sprintf "[%s] make-vector\n%s" (xstr_type t) (str_xexpr (indent + 1)
       xexpr)
| Xlet(l_l, xe) ->
    sprintf "[%s] let <%s>\n%s"
        (xstr_type (Semantic.get_type xe))
        (List.fold_left
            (fun s idt -> sprintf "%s, [%s] %s" s (xstr_type idt.t) idt.
               id)
            (let idt = fst (List.hd l_l) in
            sprintf "[%s] %s" (xstr_type idt.t) idt.id)
            (fst (List.split (List.tl l_l))))
        (str_xexpr (indent + 1) xe)
| Xlambda(idt_l, xe, i) ->
    sprintf "[lambda] <%s -> %s>\n%s"
        "abc"
        (xstr_type (Semantic.get_type xe))
        (str_xexpr (indent + 1) xe)

let str_xdefvar xdefv =
    sprintf "BIND [%s] %s = %s" (xstr_type xdefv.xvname.t) xdefv.xvname.
       id (str_xexpr 0 xdefv.xvbody)

let str_xdefun xdefun =
    let in_t_s = begin match (List.map (fun idt -> idt.t) xdefun.xfargu)
       with
    | [] -> "none"
    | hd::tl -> List.fold_left
        (fun s t -> s ^ "*" ^ (xstr_type t)) (xstr_type hd) tl end in
    sprintf "BIND [%s -> %s] %s =\n%s"
        in_t_s
        (xstr_type xdefun.xfname.t)
        xdefun.xfname.id
        (str_xexpr 1 xdefun.xfbody)
```

```
let print_xclip = function
| Xexpr(xexpr, _) -> fprintf xout "%s\n" (str_xexpr 0 xexpr)
| Xdefvar(xdefvar) -> fprintf xout "%s\n" (str_xdefvar xdefvar)
| Xdefun(xdefun) -> fprintf xout "%s\n" (str_xdefun xdefun);;

let print_sast p =
    fprintf xout"================================== sast ";
    fprintf xout"==================================\n";
    List.iter (fun xc -> print_xclip xc) p;;
```

Listing 12: sast.ml

```
open Ast

module VarMap = Map.Make(struct
    type t = string
    let compare x y = Pervasives.compare x y
end)

type xexpr =
| Xint_Lit of string
| Xbin_Lit of string
| Xhex_Lit of string
| Xbit_Binary_Lit of string * int
| Xbit_Hex_Lit of string * int
| Xstring_Lit of string
| Xvector_Lit of xexpr list
| Xid of string * c_type
| Xidd of string * xexpr list * c_type
| Xvec_Dimension of int
| Xlet of xlet_arg list * xexpr
| Xlambda of id_with_type list * xexpr * int
| Xmake_Vector of c_type * xexpr
| Xfuncall of string * xexpr list * c_type * xexpr

and xlet_arg = id_with_type * xexpr

type xdefvar = {
    xvname : id_with_type;
    xvbody : xexpr;
}

type xdefun = {
    xfname : id_with_type;
    xfargu : id_with_type list;
    xfbody : xexpr;
}

type xclip =
| Xexpr of xexpr * bool
| Xdefvar of xdefvar
| Xdefun of xdefun

type xprogram = xclip list
```

Listing 13: semantic.ml

```
open Ast
open Sast
open Builtin
open Printf
open Printast
open Exception

(* The map records bits length which decides during compilation. *)
module BitlenMap = Map.Make(struct
    type t = int
    let compare x y = Pervasives.compare x y
end);;

(* The assign the index to special function to avoid names conflict. *)
let f_counter = ref 0;;

module Semantic = struct

(* Given a function in type of xexpr, return its name as string. *)
let find_fun_name = function
| Xid(s, _) -> s
| Xlambda(_, _, i) -> sprintf "lambda_%d" i
| _ -> raise(Dev_Error("semantic.find_fun_name: it's not a function."))
;;

(* Return a list filled with a, whose length is l. *)
let rec build_list a l =
    if l = 1 then [a]
    else a::(build_list a (l-1));;

(* Given the basic type of vector, return a expr with that type. *)
let rec vector_to_expr t i_l =
    if List.length i_l = 1 then
        Vector_Lit(build_list (type_to_expr t) (List.hd i_l))
    else
        Vector_Lit(build_list (vector_to_expr t (List.tl i_l)) (List.hd
            i_l))

and string_len_n n =
    if n = 1 then
        "1"
    else
        "1" ^ string_len_n (n-1)

(* Given a type, return a expr with that type. *)
and type_to_expr = function
| Int -> Int_Lit("0")
| Bits(i) ->
    Bit_Binary_Lit(string_len_n i)
| String -> String_Lit("a")
| Vector(t, i_l) -> vector_to_expr t i_l
| Fun(_, _, _) -> raise(Dev_Error("type_to_expr.Special or Wild_Card"))
| Special(s) -> raise(Dev_Error("type_to_expr.Special or Wild_Card"))
```

```
| Wild_Card(i) -> raise(Dev_Error("type_to_expr.Special or Wild_Card"));;

(* Given the basic type of vector, return a xexpr with that type. *)
let rec vector_to_xexpr t i_l =
    if List.length i_l = 1 then
        Xvector_Lit(build_list (type_to_xexpr t) (List.hd i_l))
    else
        Xvector_Lit(build_list (vector_to_xexpr t (List.tl i_l)) (List.hd
            i_l))

(* Given a type, return a xexpr with that type. *)
and type_to_xexpr = function
| Int -> Xint_Lit("0")
| Bits(i) ->
    Xbit_Binary_Lit(string_of_int(int_of_float(10.0**(float_of_int (i-1))
        )), i)
| String -> Xstring_Lit("a")
| Vector(t, i_l) -> vector_to_xexpr t i_l
| Fun(_, _, _) -> raise(Dev_Error("type_to_xexpr.Special or Wild_Card"))
| Special(s) -> raise(Dev_Error("type_to_xexpr.Special or Wild_Card"))
| Wild_Card(i) -> raise(Dev_Error("type_to_xexpr.Special or Wild_Card"))
   ;;

(* Return the type of xexpr. *)
let rec get_type = function
| Xint_Lit(_)
| Xbin_Lit(_)
| Xhex_Lit(_) -> Int
| Xbit_Binary_Lit(_, i) -> Bits(i)
| Xbit_Hex_Lit(_, i) -> Bits(i)
| Xstring_Lit(_) -> String
| Xvector_Lit(hd::tl) ->
    let t = get_type hd in
    let len = (List.length tl) + 1 in
    begin match t with
    | Int -> Vector(Int, [len])
    | Bits(i) -> Vector(Bits(i), [len])
    | String -> Vector(String, [len])
    | Vector(t, l) -> Vector(t, len::l)
    | _ -> raise(Dev_Error("vector_lit must have concrete type")) end
| Xid(_, t) -> t
| Xidd(_, _, t) -> t
| Xvec_Dimension(_) -> Int
| Xlet(_, xe) -> get_type xe
| Xlambda(idt_l, xe, _) ->
    let t_list = List.map (fun idt -> idt.t) idt_l in
    Fun(Fix, get_type xe, t_list)
| Xmake_Vector(t, _) -> t
| Xfuncall(_, _, t, _) -> t
| Xvector_Lit([]) -> raise(Dev_Error("vector_lit cannot be zero size"))
;;

(* Confirm the uncertain type according to b_map. *)
let rec ass_type b_map = function
```

```
| Wild_Card(i) -> BitlenMap.find i b_map
| Bits(i) ->
    if i <= 0 then BitlenMap.find i b_map
    else Bits(i)
| Vector(t, l) -> Vector(ass_type b_map t, l)
| Fun(fc, t, t_l) ->
    let t' = ass_type b_map t in
    let t_l' = List.map (fun t -> ass_type b_map t) t_l in
    Fun(fc, t', t_l')
| x -> x
;;

(* Determine whether two lists of index are the same
    negative number can match with any number. *)
let rec vector_list_eq l1 l2 =
    match l1, l2 with
    | [], [] -> true
    | h1::t1, h2::t2 ->
        if h1 = 0 || h2 = 0 then
            true
        else if h1 = h2 || h1 < 0 || h2 < 0 then
            vector_list_eq t1 t2
        else
            false
    | _, _ -> false;;

(* Determine whether t1 t2 can be the same types
   b_map is used for handling the Bits(0), Bits(-1), Wild_Card(i)... type
      . *)
let rec compatible_type t1 t2 b_map =
    (*fprintf stderr "t1=%s, t2=%s\n" (string_type t1) (string_type t2);
       *)
    match (t1, t2) with
    | Wild_Card(i), Wild_Card(j) ->
        raise(Dev_Error("compare wild_card type to wild_card ?"))
    | t, Wild_Card(i) | Wild_Card(i), t ->
        if BitlenMap.mem i b_map then
            ((BitlenMap.find i b_map) = t), b_map
        else
            true, BitlenMap.add i t b_map
    | (Bits(n), Bits(m)) ->
        if (((n > 0 && m > 0) || (n <= 0 && m <= 0)) && n <> m) then
            false, b_map
        else if n == m then
            true, b_map
        else if n <= 0 then
            if BitlenMap.mem n b_map then
                ((BitlenMap.find n b_map) = Bits(m)), b_map
            else
                true, BitlenMap.add n (Bits(m)) b_map
        else
            if BitlenMap.mem m b_map then
                ((BitlenMap.find m b_map) = Bits(n)), b_map
            else
```

```
                true, BitlenMap.add m (Bits(n)) b_map
    | Fun(Indef, _, _), Fun(_, _, _) -> true, b_map
    | Fun(_, _, _), Fun(_, _, _) -> true, b_map
    | Vector(t1', l1), Vector(t2', l2) ->
        ((fst (compatible_type t1' t2' b_map)) && (vector_list_eq l1 l2))
           , b_map
    | x, y ->
        x = y, b_map

(* try to evaluate some expressions at compile time *)
let easy_eval = function
| Xint_Lit(s) -> int_of_string s
| _ -> 0

(* Return the last n elements in list l *)
let rec cut_list l n =
    if n = 0 then
        ([], l)
    else
        let (l1, l2) = cut_list (List.tl l) (n-1) in
        ((List.hd l)::l1, l2)

(* Add dimension identifiers @1 ... @n to varmap *)
let rec add_vec_dim_ids n varmap =
    if n <= 0 then
        varmap
    else
        let new_varmap = VarMap.add ("@" ^ (string_of_int n)) Int varmap
           in
        add_vec_dim_ids (n-1) new_varmap;;

(* Check the semantic of an expr, return the xexpr and variable map.
   If there is an error, throw the exceptions. *)
let rec check_expr varmap = function
| Int_Lit(s) -> Xint_Lit(s), varmap
| Bin_Lit(s) -> Xbin_Lit(s), varmap
| Hex_Lit(s) -> Xhex_Lit(s), varmap
| Bit_Binary_Lit(s) -> Xbit_Binary_Lit(s, String.length s), varmap
| Bit_Hex_Lit(s) -> Xbit_Hex_Lit(s, 4 * (String.length s)), varmap
| String_Lit(s) -> Xstring_Lit(s), varmap
| Vector_Lit(expr_l) ->
    (* Check whether all types in expr_l are same *)
    let xexpr_l = List.map (fun e -> (fst (check_expr varmap e))) expr_l
       in
    let expr_t_l = List.map (fun xe -> get_type xe) xexpr_l in
    ignore (List.fold_left
        (fun t1 t2 ->
            if t1 = t2 then
                t1
            else
                raise(Invalid_Vector((string_expr 0 (Vector_Lit(expr_l)))
                   )))
        (List.hd expr_t_l) (List.tl expr_t_l));
    Xvector_Lit(xexpr_l), varmap
```

```
| Id(s) ->
    if VarMap.mem s varmap then
        Xid(s, VarMap.find s varmap), varmap
    else
        raise(Indefined_Id(s))
| Idd(s, expr_l) ->
    let xexpr_l = List.map (fun e -> (fst (check_expr varmap e))) expr_l
       in
    let expr_t_l = List.map (fun xe -> get_type xe) xexpr_l in
    List.iter (fun t -> if t = Int then () else
        raise(Invalid_Ind(string_expr 0 (Idd(s, expr_l))))) expr_t_l;
    let s_type =
        if VarMap.mem s varmap then
            VarMap.find s varmap
        else
            raise(Indefined_Id(s)) in
    begin
        match s_type with
        | Vector(ctype, int_l) ->
            (* If the query dimension is higher then declaration, it
               fails. *)
            if List.length expr_l > List.length int_l then
                raise(Vector_Dim(s, List.length int_l))
            else
                let (l1, l2) = cut_list int_l (List.length expr_l) in
                (* Check if the index is out of bound. *)
                let valid = List.fold_left2
                    (fun b max i -> b && max > easy_eval i && easy_eval i
                        >= 0)
                    true l1 xexpr_l in
                if valid && List.length l2 > 0 then
                    Xidd(s, xexpr_l, Vector(ctype, l2)), varmap
                else if valid then
                    Xidd(s, xexpr_l, ctype), varmap
                else
                    raise(Invalid_Ind(string_expr 0 (Idd(s, expr_l)))),
                       varmap
        | _ -> raise(Not_Vector(s))
    end
| Vec_Dimension(i) ->
    if VarMap.mem ("@" ^ (string_of_int i)) varmap && (i >= 0) then
        Xvec_Dimension(i), varmap
    else
        raise(Make_Vec_Bound(i))
| Funcall(expr, expr_l) ->
    let xfun, varmap' = check_expr varmap expr in
    let xexpr_l, varmap = List.fold_left (fun (xl, m) e ->
        let (xe, map) = (check_expr m e) in
        xe::xl, map)
        ([], varmap') expr_l in
    let xexpr_l = List.rev xexpr_l in
    let xexpr_t_l = List.map (fun xe -> get_type xe) xexpr_l in
    let xexpr, varmap = check_expr varmap expr in
    let xexpr_t = get_type xexpr in
```

```
    begin match xexpr_t with
    | Fun(Indef, _, _) ->
        raise(Dev_Error("Semantic.Funcall unexpected case - indef"))
    | Fun(Fix, out_t, int_t_l) ->
        if List.length xexpr_t_l <> List.length int_t_l then
            raise(Wrong_Argu_Len(string_expr 0 expr))
        else
            let bit_map = List.fold_left2
                (fun b_map t1 t2 ->
                    let b, new_b_map = compatible_type t1 t2 b_map in
                    if b then
                        new_b_map
                    else
                        raise(Wrong_Argu_Type(string_expr 0 (expr))))
                BitlenMap.empty xexpr_t_l int_t_l in
            let fun_name = find_fun_name xexpr in
            let exact_out_t = begin match fun_name with
            | "bits-of-int" -> Bits((easy_eval (List.hd xexpr_l)))
            | "bits-of-string" -> Bits((easy_eval (List.hd xexpr_l)))
            | "pad" -> Bits((easy_eval (List.nth xexpr_l 1)))
            | _ -> ass_type bit_map out_t end in
            Xfuncall(fun_name, xexpr_l, exact_out_t, xexpr), varmap
    | Fun(Len_Flex, out_t, [in_t]) ->
        if List.length xexpr_t_l <= 0 then
            raise(Wrong_Argu_Len(string_expr 0 expr))
        else
            let bit_map = List.fold_left
                (fun b_map t ->
                    let b, new_b_map = compatible_type in_t t b_map in
                    if b then
                        new_b_map
                    else
                        raise(Wrong_Argu_Type(string_expr 0 (expr))))
                BitlenMap.empty xexpr_t_l in
            let exact_out_t = ass_type bit_map out_t in
            Xfuncall(find_fun_name xexpr, xexpr_l, exact_out_t, xexpr),
               varmap
    | Special("group") ->
        if List.length xexpr_l <> 2 then
            raise(Wrong_Argu_Len(string_expr 0 expr))
        else
            begin match (List.hd xexpr_t_l)
                , (List.nth xexpr_t_l 1) with
            | Bits(i), Int ->
                let l = easy_eval (List.nth xexpr_l 1) in
                    if l > 0 then
                        let out_t = Vector(Bits(l), [(i+l-1) / l]) in
                        let fun_name = sprintf "group__%d" !f_counter in
                        f_counter := 1 + !f_counter;
                        Xfuncall(fun_name, xexpr_l, out_t, xexpr), varmap
                    else
                        raise(Eval_Fail(string_expr 0 (List.nth expr_l 1)
                           ))
            | _ ->
```

```
                raise(Wrong_Argu_Type(
                    string_expr 0 (Funcall(expr, expr_l)))) end
    | Special("merge") ->
        if List.length xexpr_l <> 1 then
            raise(Wrong_Argu_Len(string_expr 0 expr))
        else
            begin match (List.hd xexpr_t_l) with
            | Vector(Bits(i), [vec_l]) ->
                let out_t = Bits(i * vec_l) in
                let fun_name = sprintf "merge__%d" !f_counter in
                f_counter := 1 + !f_counter;
                Xfuncall(fun_name, xexpr_l, out_t, xexpr), varmap
            | _ -> raise(Wrong_Argu_Type(string_expr 0
                    (Funcall(expr, expr_l)))) end
    | Special("transpose") ->
        if List.length xexpr_l <> 1 then
            raise(Wrong_Argu_Len(string_expr 0 expr))
        else
            begin match (List.hd xexpr_t_l) with
            | Vector(t, [d1;d2]) ->
                let out_t = Vector(t, [d2; d1]) in
                let fun_name = sprintf "transpose__%d" !f_counter in
                f_counter := 1 + !f_counter;
                Xfuncall(fun_name, xexpr_l, out_t, xexpr), varmap
            | _ -> raise(Wrong_Argu_Type(string_expr 0
                    (Funcall(expr, expr_l)))) end
    | Special("map") ->
        if List.length xexpr_l <> 2 then
            raise(Wrong_Argu_Len(string_expr 0 expr))
        else
            let xexpr_t_1 = List.hd xexpr_t_l in
            let xexpr_t_2 = List.nth xexpr_t_l 1 in
            begin match xexpr_t_1, xexpr_t_2 with
            | Fun(Fix, out_t, [in_t]), Vector(t, l) ->
                let in_fun_name = begin match List.hd xexpr_l with
                | Xid(name, _) -> name
                | Xlambda(_, _, i) -> "lambda__"
                | _ -> raise(Dev_Error("Semantic.Special.map")) end in
                let tmp_out, b_map =
                    if List.length l = 1 then
                        let valid, m =
                            compatible_type in_t t BitlenMap.empty in
                        if valid then Vector(out_t, l), m
                        else raise(Wrong_Argu_Type(
                            (string_expr 0 (Funcall(expr, expr_l)))))
                    else
                        let valid, m = compatible_type
                            in_t (Vector(t, List.tl l)) BitlenMap.empty
                               in
                        if valid then Vector(out_t, [List.hd l]), m
                        else raise(Wrong_Argu_Type(
                            (string_expr 0 (Funcall(expr, expr_l))))) in
                let real_out = begin match tmp_out with
                | Vector(Vector(t, inl), outl) -> Vector(t, inl@outl)
```

```
                | t -> t end in
                let fun_name = sprintf "map__%d" !f_counter in
                f_counter := 1 + !f_counter;
                if in_fun_name = "lambda__" then
                    Xfuncall(fun_name, xexpr_l, real_out, xexpr), varmap
                else Xfuncall(fun_name,
                        [Xid(in_fun_name, Fun(Fix, out_t,
                            [ass_type b_map in_t])); (List.nth xexpr_l 1)
                               ]
                        , real_out, xexpr), varmap
            | Special(s), Vector(t, l) ->
                let in_t = if List.length l = 1
                    then t else Vector(t, List.tl l) in
                let tmp_expr = type_to_expr in_t in
                begin match check_expr varmap (Funcall(Id(s), [tmp_expr])
                   ) with
                | Xfuncall(fun_name, [in_xe_l], out_t, xexpr), _ ->
                    let in_t = get_type in_xe_l in
                    let tmp_out, b_map =
                        if List.length l = 1 then
                            let valid, m =
                                compatible_type in_t t BitlenMap.empty in
                            if valid then Vector(out_t, l), m
                            else raise(Wrong_Argu_Type(
                                (string_expr 0 (Funcall(expr, expr_l)))))
                        else
                            let valid, m = compatible_type
                                in_t (Vector(t, List.tl l)) BitlenMap.
                                   empty in
                            if valid then Vector(out_t, [List.hd l]), m
                            else raise(Wrong_Argu_Type((string_expr 0
                                (Funcall(expr, expr_l))))) in
                    let real_out = begin match tmp_out with
                    | Vector(Vector(t, inl), outl) -> Vector(t, inl@outl)
                    | t -> t end in
                    let fun_name' = sprintf "map__%d" !f_counter in
                    f_counter := 1 + !f_counter;
                    Xfuncall(fun_name',
                        [Xid(fun_name,
                            Fun(Fix, out_t, [in_t])); (List.nth xexpr_l
                               1)]
                        , real_out, xexpr), varmap
                | _ -> raise(Dev_Error("")) end
            | _ -> raise(Wrong_Argu_Type("1" ^
                (string_expr 0 (Funcall(expr, expr_l))))) end
    | Special("reduce") ->
        if List.length xexpr_l <> 2 then
            raise(Wrong_Argu_Len(string_expr 0 expr))
        else
            let xexpr_t_1 = List.hd xexpr_t_l in
            let xexpr_t_2 = List.nth xexpr_t_l 1 in
            begin match xexpr_t_1, xexpr_t_2 with
            | Fun(Len_Flex, out_t, [in_t]), Vector(t, l) ->
                let b_map =
```

```
                    if List.length l = 1 then
                        let valid, m = compatible_type in_t t BitlenMap.
                            empty in
                        if valid then m
                        else raise(Wrong_Argu_Type(
                            (string_expr 0 (Funcall(expr, expr_l)))))
                    else
                        let valid, m = compatible_type
                            in_t (Vector(t, List.tl l)) BitlenMap.empty
                                in
                        if valid then m
                        else raise(Wrong_Argu_Type(
                            (string_expr 0 (Funcall(expr, expr_l))))) in
                let out_t = ass_type b_map out_t in
                let in_t = ass_type b_map in_t in
                let fun_name = sprintf "reduce__%d" !f_counter in
                f_counter := 1 + !f_counter;
                let fst_xe' =
                    begin match List.hd xexpr_l with
                    | Xid(s, Fun(Len_Flex, o, [i])) ->
                        Xid(s, Fun(Fix, out_t, [in_t;in_t]))
                    | _ ->
                        raise(Dev_Error("Semantic: special(reduce).
                            len_flex"))
                    end in
                Xfuncall(fun_name, fst_xe'::(List.tl xexpr_l), out_t,
                    xexpr),
                varmap
            | Fun(Fix, out_t, [in_t; in_t']), Vector(t, l) ->
                let _ =
                    if List.length l = 1 && in_t' = in_t then
                        let valid, m = compatible_type in_t t BitlenMap.
                            empty in
                        if valid then m
                        else raise(Wrong_Argu_Type(
                            (string_expr 0 (Funcall(expr, expr_l)))))
                    else
                        let valid, m = compatible_type
                            in_t (Vector(t, List.tl l)) BitlenMap.empty
                                in
                        if valid then m
                        else raise(Wrong_Argu_Type(
                            (string_expr 0 (Funcall(expr, expr_l))))) in
                let fun_name = sprintf "reduce__%d" !f_counter in
                f_counter := 1 + !f_counter;
                Xfuncall(fun_name, xexpr_l, out_t, xexpr), varmap
            | _ -> raise(Wrong_Argu_Type(
                (string_expr 0 (Funcall(expr, expr_l))))) end
    | Special("zero") ->
        if List.length xexpr_l <> 1 then
            raise(Wrong_Argu_Len(string_expr 0 expr))
        else
            begin match (List.hd xexpr_t_l) with
            | Int ->
```

```
                let l = easy_eval(List.hd xexpr_l) in
                let out_t = Bits(l) in
                let fun_name = sprintf "zero__%d" !f_counter in
                f_counter := 1 + !f_counter;
                Xfuncall(fun_name, xexpr_l, out_t, xexpr), varmap
            | _ -> raise(Wrong_Argu_Type(
                string_expr 0 (Funcall(expr, expr_l)))) end
    | Special("rand") ->
        if List.length xexpr_l <> 1 then
            raise(Wrong_Argu_Len(string_expr 0 expr))
        else
            begin match (List.hd xexpr_t_l) with
            | Int ->
                let l = easy_eval(List.hd xexpr_l) in
                let out_t = Bits(l) in
                let fun_name = sprintf "rand__%d" !f_counter in
                f_counter := 1 + !f_counter;
                Xfuncall(fun_name, xexpr_l, out_t, xexpr), varmap
            | _ -> raise(Wrong_Argu_Type(
                string_expr 0 (Funcall(expr, expr_l)))) end
    | x -> raise(Not_Function(string_expr 0 expr))
    end
| Lambda(idt_list, exp) ->
    let new_varmap = List.fold_left
        (fun m idt -> VarMap.add idt.id idt.t m) varmap idt_list in
    let xexpr, _ = check_expr new_varmap exp in
    let n = !f_counter in
    f_counter := 1 + !f_counter;
    Xlambda(idt_list, xexpr, n), varmap

| Let(let_arg_l, expr) ->
    let xlet_arg_l, varmap' = List.fold_left
        (fun (l', m) (idt, e) ->
            let xe, m = check_expr m e in
            if (get_type xe) = idt.t then
                (idt, xe)::l', VarMap.add idt.id idt.t m
            else
                raise(Bind_Wrong_Type(idt.id)))
        ([], varmap)
        let_arg_l in
    let xlet_arg_l = List.rev xlet_arg_l in
    let xexpr, t = check_expr varmap' expr in
    Xlet(xlet_arg_l, xexpr), varmap
| Make_Vector(c_type, exp) ->
    begin
    match c_type with
    | Vector(_, l) ->
        let new_varmap = add_vec_dim_ids (List.length l) varmap in
        let xexpr, m = check_expr new_varmap exp in
        Xmake_Vector(c_type, xexpr), varmap
    | _ -> raise(Wrong_Argu_Type("make-vector")) end
;;

(* Check the semantic of defvar, return xdefvar and variable map. *)
```

```
let check_defvar varmap defv =
    if VarMap.mem defv.vname.id varmap then
        raise(Bind_Twice(defv.vname.id))
    else
        let xexpr, newmap = check_expr varmap defv.vbody in
        if fst (compatible_type defv.vname.t
            (get_type xexpr) BitlenMap.empty) then
            {xvname = defv.vname; xvbody = xexpr}
                , (VarMap.add defv.vname.id defv.vname.t newmap)
        else
            raise(Bind_Wrong_Type(defv.vname.id));;

(* Check the semantic of defun, return xdefun and variable map. *)
let check_defun varmap defun =
    if VarMap.mem defun.fname.id varmap then
        raise(Bind_Twice(defun.fname.id))
    else
        let varmap' = VarMap.add defun.fname.id
            (Fun(Fix, defun.fname.t,
                (List.map (fun idt -> idt.t) defun.fargu) )) varmap in
        let infunmap = List.fold_left
            (fun m idt -> VarMap.add idt.id idt.t m)
            varmap'
            defun.fargu in
        let xexpr, infunmap = check_expr infunmap defun.fbody in
        if fst (compatible_type defun.fname.t
            (get_type xexpr) BitlenMap.empty) then
            {xfname = defun.fname; xfargu = defun.fargu; xfbody = xexpr},
            VarMap.add
                defun.fname.id
                (Fun(Fix, defun.fname.t,
                    (List.map (fun idt -> idt.t) defun.fargu)))
                varmap
        else
            raise(Defun_Wrong_Type(defun.fname.id))

let check_clip varmap = function
| Expr(expr, b) ->
    let (xe, m) = check_expr varmap expr in
    Xexpr(xe, b), m
| Defvar(defvar) ->
    let (xdefv, m) = check_defvar varmap defvar in
    Xdefvar(xdefv), m
| Defun(defun) ->
    let (xdefun, m) = check_defun varmap defun in
    Xdefun(xdefun), m
;;

let check_ast program =
    List.rev (snd (List.fold_left
        (fun (m ,p_list) c ->
            let (xc, map) = (check_clip m c) in
            (map, xc::p_list))
        ((add_builtin_fun VarMap.empty), [])
```

```
        program));;

end
```

Listing 14: semantic.mli

```
open Ast
open Sast
(*
module VarMap : Map.S with type key = string
*)
module Semantic :
    sig
        val type_to_expr: c_type -> expr
        val type_to_xexpr: c_type -> xexpr
        val get_type: xexpr -> c_type

        val check_expr: c_type VarMap.t -> expr -> xexpr * c_type VarMap.
           t
        val check_clip: c_type VarMap.t -> clip -> xclip * c_type VarMap.
           t

        val check_ast: program -> xprogram
    end
```

Listing 15: translate.ml

```
open Ast
open Sast
open Printast
open Printsast
open Semantic
open Exception
open Printf
open Builtin
open Str

(* To remember whether a function in C is already created *)
module FunSet = Set.Make(struct
    type t = string
    let compare x y = Pervasives.compare x y
end)

let dyna_out_init = open_out "./library/dynamic_builtin.cc";;
let dyna_out = open_out_gen
    [Open_wronly; Open_append; Open_creat; Open_text]
    0o666 "./library/dynamic_builtin.cc";;
let c_header_list = ["<cstdio>"; "<cstdlib>";
                        "<gmp.h>"; "<library/builtin.cc>";
                        "<library/dynamic_builtin.cc>"];;

(* in order to assign unique id in C *)
let id_counter = ref 0;;
```

```
let fun_db = ref FunSet.empty;;

module Translate = struct

exception Funciton_call_without_function_name

(* generate the syntax of c *)
let rec c_header_generator = function
| [] -> ""
| head::tail -> "#include " ^ head ^ "\n" ^ (c_header_generator tail);;

let c_main_left =
    "int main(int argc, char *argv[]){\n";;
let c_main_right = "    return 0;\n}\n";;

let rec string_of_type = function
| Int -> "string"
| Bits(i) -> sprintf "bitset<%d>" i
| String -> "string"
| Vector(t, i_l) ->
    let in_str_type =
        if List.length i_l = 1 then
            string_of_type t
        else
            string_of_type (Vector(t, List.tl i_l)) in
    sprintf "vector< %s >" in_str_type
| Fun(Fix, out_t, in_t_l) ->
    let in_ts_l = List.map string_of_type in_t_l in
    sprintf "function<%s (%s)>" (string_of_type out_t)
    (List.fold_left (fun s1 s2 -> s1 ^ ", " ^ s2)
        (List.hd in_ts_l)
        (List.tl in_ts_l))
| Fun(_, _, _) -> raise(Dev_Error("the type should be confirmed (fun)"))
| Wild_Card(i) -> raise(Dev_Error("the type should be confirmed (wc)"))
| Special(s) -> raise(Dev_Error("the type should be confirmed (sp)"))

(* create indent in C *)
let rec str_ind layer =
if layer >= 1 then
    "    " ^ str_ind (layer-1)
else
    "";;

(* create [1;2;3;...;len-1] *)
let rec range len =
    if len = 0 then []
    else (len-1) :: range (len - 1)

let rec cout_vector var t i_l =
    if List.length i_l = 1 then
        let len = List.hd i_l in
        match t with
        | Bits(j) ->
            sprintf "%scout <<\"{\";\n%scout << \"}\";\n" (str_ind 1)
```

```
            (List.fold_left
                (fun s i -> sprintf
                    "%s printf(\" \");\nprintbit<%d>(%s[%d]);\n"
                    s j var i)
                (sprintf "printbit<%d>(%s[%d]);\n" j var 0)
                (List.tl (List.rev (range len))))
        | _ ->
            sprintf "%scout <<\"{\";\n%s << \"}\";\n" (str_ind 1)
            (List.fold_left (fun s i -> sprintf "%s <<\" \"<< %s[%d]" s
               var i)
                (sprintf "cout << %s[0]" var)
                (List.tl (List.rev (range len))))
    else
        let len = List.hd i_l in
        (List.fold_left
            (fun s i -> s ^
                "    cout << \"\\n\";\n" ^
                (cout_vector (sprintf "%s[%d]" var i) t (List.tl i_l)))
            (sprintf "%scout << \"{\";\n%s"
                (str_ind 1) (cout_vector
                    (sprintf "%s[%d]" var 0) t (List.tl i_l) ))
            (List.tl (List.rev (range len)))) ^
        "    cout <<\"}\";\n"
;;

(* input Vector(Int, [3;8;5]) return Vector(Int, [8;5] *)
let decrease_dim = function
| Vector(t, i_l) ->
    if List.length i_l > 1 then
        Vector(t, List.tl i_l)
    else
        t
| _ -> raise(Dev_Error("non-vector type cannot decrease dimension"))
;;

(* input (hello, 3), output hello, hello, hello *)
let rec string_repeat s n =
    if n <= 1 then
        s
    else
        s ^ ", " ^ (string_repeat s (n-1))
;;

(* generate the name of vector argument *)
let rec gen_make_vec_args n =
    if n <= 1 then
        sprintf "string at__%d" n
    else
        sprintf "%s, string at__%d" (gen_make_vec_args (n-1)) n
;;

(* generate the code of initialize vector *)
let rec init_vector indent ids dim_l =
    if List.length dim_l = 1 then
```

```
        sprintf "%s%s.resize(%d);\n" (str_ind indent) ids (List.hd dim_l)
    else
        let l = List.length dim_l in
        (sprintf "%s%s.resize(%d);\n" (str_ind indent) ids (List.hd dim_l
            )) ^
        (sprintf "%sfor (int i%d = 0; i%d < %d; i%d++) {\n"
            (str_ind indent) l l (List.hd dim_l) l) ^
        (init_vector (indent+1) (sprintf "%s[i%d]" ids l) (List.tl dim_l)
            )^
        (sprintf "%s}" (str_ind indent))
;;

(* generate the for loop to implement make-vector *)
let rec for_make_vector indent dim_l =
    let up_limit = List.hd dim_l in
    let l = List.length dim_l in
    if l = 1 then
        sprintf "%sfor (int i%d = 0; i%d < %d; i%d++)\n"
            (str_ind indent) l l up_limit l
    else
        sprintf "%sfor (int i%d = 0; i%d < %d; i%d++)\n%s"
            (str_ind indent) l l up_limit l
        (for_make_vector (indent+1) (List.tl dim_l))
;;

(* generate the dimension index of make-vector *)
let rec dim_ind_make_vec n =
    if n = 0 then
        ""
    else
        (sprintf "[i%d]" n) ^ (dim_ind_make_vec (n-1))
;;

(* generate the tmp index in for loop for make-vector *)
let rec ind_arg_make_vec n =
    if n = 1 then
        sprintf "to_string(i%d)" n
    else
        (sprintf "to_string(i%d), " n) ^ (ind_arg_make_vec (n-1))
;;

(* return the name of the type corresponding in C *)
let rec name_of_type = function
| Int -> "int"
| Bits(i) -> sprintf "bit%d" i
| String -> "string"
| Vector(t, i_l) ->
    let in_str_type =
        if List.length i_l = 1 then
            name_of_type t
        else
            name_of_type (Vector(t, List.tl i_l)) in
    sprintf "v_%s" in_str_type
| Fun(_, _, _) -> raise(Dev_Error("we don't compare function"))
```

```
| Wild_Card(i) -> raise(Dev_Error("we don't compare wild_card"))
| Special(s) -> raise(Dev_Error("we don't compare special"))
;;

(* transform the name in clip to name in c, prevent '-'' *)
let c_name s =
    String.map (fun c -> if c = '-' then '_' else c) s
;;

(* transform the function name in clip to name in c
    prevent '-' and create the name for dynamic functions *)
let c_fun_name s t =
    if s = "if" then
        begin match t with
        | Fun(Fix, t1, [Bits(1); t2; t3]) ->
            sprintf "if__%s" (name_of_type t1)
        | _ -> raise(Dev_Error("eq's arguments not confirmed.")) end
    else if s = "eq" then
        begin match t with
        | Fun(Fix, Bits(1), [t1; t2]) ->
            sprintf "eq__%s" (name_of_type t1)
        | _ -> raise(Dev_Error("eq's arguments not confirmed.")) end
    else if s = "neq" then
        begin match t with
        | Fun(Fix, Bits(1), [t1; t2]) ->
            sprintf "neq__%s" (name_of_type t1)
        | _ -> raise(Dev_Error("neq's arguments not confirmed.")) end
    else if s = "^" then
        begin match t with
        | Fun(Fix, Bits(i), in_l) ->
            sprintf "xor__%d__%d" i (List.length in_l)
        | _ -> raise(Dev_Error("^'s arguments not confirmed.")) end
    else if s = "|" || s = "or" then
        begin match t with
        | Fun(Fix, Bits(i), in_l) ->
            sprintf "or__%d__%d" i (List.length in_l)
        | _ -> raise(Dev_Error("|'s arguments not confirmed.")) end
    else if s = "&" || s = "and" then
        begin match t with
        | Fun(Fix, Bits(i), in_l) ->
            sprintf "and__%d__%d" i (List.length in_l)
        | _ -> raise(Dev_Error("&'s arguments not confirmed.")) end
    else if s = "parity" then
        begin match t with
        | Fun(Fix, _, [Bits(i)]) -> sprintf "parity_%d" i
        | _ ->
            raise(Dev_Error("nit-of-bits' arguments not confirmed.")) end
    else if s = ">>>" then
        begin match t with
        | Fun(Fix, Bits(i), _) -> sprintf "rotate_r__%d" i
        | _ -> raise(Dev_Error(">>>'s arguments not confirmed.")) end
    else if s = "<<<" then
        begin match t with
        | Fun(Fix, Bits(i), _) -> sprintf "rotate_l__%d" i
```

```
        | _ -> raise(Dev_Error("<<<'s arguments not confirmed.")) end
    else if s = ">>" then
        begin match t with
        | Fun(Fix, Bits(i), _) -> sprintf "shift_r__%d" i
        | _ -> raise(Dev_Error(">>'s arguments not confirmed.")) end
    else if s = "<<" then
        begin match t with
        | Fun(Fix, Bits(i), _) -> sprintf "shift_l__%d" i
        | _ -> raise(Dev_Error("<<'s arguments not confirmed.")) end
    else if s = "flip-bit" then
        begin match t with
        | Fun(Fix, Bits(i), _) -> sprintf "%s__%d" (c_name s) i
        | _ -> raise(Dev_Error("<<'s arguments not confirmed.")) end
    else if s = "flip" then
        begin match t with
        | Fun(Fix, Bits(i), _) -> sprintf "%s__%d" (c_name s) i
        | _ -> raise(Dev_Error("<<'s arguments not confirmed.")) end
    else if s = "int-of-bits" then
        begin match t with
        | Fun(Fix, _, [Bits(i)]) -> sprintf "%s__%d" (c_name s) i
        | _ ->
            raise(Dev_Error("nit-of-bits' arguments not confirmed.")) end
    else if s = "string-of-bits" then
        begin match t with
        | Fun(Fix, _, [Bits(i)]) -> sprintf "%s__%d" (c_name s) i
        | _ ->  raise(Dev_Error
            ("string-of-bits' arguments not confirmed.")) end
    else if s = "bits-of-int" then
        begin match t with
        | Fun(Fix, Bits(i), _) -> sprintf "%s__%d" (c_name s) i
        | _ -> raise(Dev_Error
            ("bit-of-int's arguments not confirmed.")) end
    else if s = "bits-of-string" then
        begin match t with
        | Fun(Fix, Bits(i), _) -> sprintf "%s__%d" (c_name s) i
        | _ -> raise(Dev_Error
            ("bit-of-string's arguments not confirmed.")) end
    else if s = "pad" then
        begin match t with
        | Fun(Fix, Bits(i), [Bits(j); _]) ->
            sprintf "%s__%d__%d" (c_name s) i j
        | _ -> raise(Dev_Error
                ("pad's arguments not confirmed.")) end
    else if s = "+" then
        begin match t with
        | Fun(Fix, Int, [Int; Int]) -> "add2"
        | _ -> "add" end
    else if s = "*" then
        begin match t with
        | Fun(Fix, Int, [Int; Int]) -> "mul2"
        | _ -> "mul" end
    else if s = "-" then "subtract"
    else if s = "/" then "divide"
    else if s = "not" then "not__"
```

```
    else if s = "less" then "less__"
    else if s = "greater" then "greater__"
    else if s = "leq" then "leq__"
    else if s = "geq" then "geq__"
    else if s = "pow" then "power"
    else
        c_name s
;;

(* translate an expression into c code *)
let rec translate_expr indent ids = function
| Xint_Lit(s) -> sprintf
    "%sstring %s = \"%s\";\n"
    (str_ind indent) ids s
| Xbin_Lit(s) -> sprintf
    "%sstring %s = dec_of_bin(\"%s\");\n"
    (str_ind indent) ids s
| Xhex_Lit(s) -> sprintf
    "%sstring %s = dec_of_hex(\"%s\");\n"
    (str_ind indent) ids s
| Xbit_Binary_Lit(s, i) -> sprintf
    "%sbitset<%d> %s = bitset<%d>(string(\"%s\"));\n"
    (str_ind indent) i ids i s
| Xbit_Hex_Lit(s, i) -> sprintf
    "%sbitset<%d> %s = bitset<%d>(bin_of_hex(string(\"%s\")));\n"
    (str_ind indent) i ids i s
| Xstring_Lit(s) -> sprintf
    "%sstring %s = \"%s\";\n"
    (str_ind indent) ids s
| Xvector_Lit(xexpr_l) ->
    let len = List.length xexpr_l in
    let ele_dec_and_ele_l = (List.mapi (fun i x ->
        id_counter := 1 + !id_counter;
        let ids = (sprintf "id__%d" !id_counter) in
        (translate_expr (1+indent) ids x), ids, i) xexpr_l) in
    let ele_dec = List.fold_left (fun s1 (ele_dec, _, _) -> sprintf "%s%s
       " s1 ele_dec) "" ele_dec_and_ele_l in
    let eles_assign = List.fold_left
        (fun s (_, ele, i) -> sprintf "%s%s%s[%d] = %s;\n" s (str_ind (
           indent+1)) ids i ele)
        ""
        ele_dec_and_ele_l in
    sprintf "%s%svector< %s > %s;\n%s%s.resize(%d);\n%s"
        ele_dec
        (str_ind indent) (string_of_type (Semantic.get_type (List.hd
           xexpr_l))) ids
        (str_ind (indent+1)) ids len eles_assign
| Xid(s, t) -> begin match t with
    | Fun(Fix, out_t, in_t_l) ->
        let in_t_l_s = List.fold_left
            (fun s t -> sprintf "%s, %s" s (string_of_type t))
            (string_of_type (List.hd in_t_l))
            (List.tl in_t_l) in
        sprintf "%sfunction <%s (%s)> %s = &%s;\n"
```

```
        (str_ind indent) (string_of_type out_t) in_t_l_s ids (c_fun_name
           s t)
    | _ -> sprintf "%s%s %s = %s;\n"
        (str_ind indent) (string_of_type t) ids (c_name s) end
| Xidd(s, xexpr_l, t) ->
    let arg_dec_and_arg_l = (List.map (fun x ->
        id_counter := 1 + !id_counter;
        let ids = (sprintf "id__%d" !id_counter) in
        (translate_expr (1+indent) ids x), ids) xexpr_l) in
    let args_dec = List.fold_left (fun s1 (arg_dec, _) -> sprintf "%s%s"
       s1 arg_dec) "" arg_dec_and_arg_l in
    let args_l = snd (List.split arg_dec_and_arg_l) in
    let vector_part_s = List.fold_left
        (fun s ids -> s ^ "[string_of_int(" ^ ids ^ ")]")
        ("[string_of_int(" ^ (List.hd args_l) ^ ")]")
        (List.tl args_l) in
    sprintf "%s%s%s %s = %s%s;\n"
    args_dec (str_ind indent)
    (string_of_type t) ids
    (c_name s) vector_part_s
| Xvec_Dimension(i) -> sprintf "%sstring %s = at__%d;\n" (str_ind indent)
    ids i
| Xlet(l_list, xexpr) ->
    let bind_s = List.fold_left
        (fun s (idt, xexp) ->
            id_counter := 1 + !id_counter;
            let var = (sprintf "id__%d" !id_counter) in
            let bind_s = translate_expr (indent+1) var xexp in
            let type_s = (string_of_type idt.t) in
            s ^ (sprintf("%s%s%s %s = %s;\n") bind_s (str_ind indent)
               type_s (c_name idt.id) var))
        ""
        l_list in
    id_counter := 1 + !id_counter;
    let var = (sprintf "id__%d" !id_counter) in
    let last_eval_s = translate_expr (indent+1) var xexpr in
    sprintf("%s%s%s%s %s = %s;\n")
        bind_s last_eval_s (str_ind indent) (string_of_type (Semantic.
           get_type xexpr)) ids var
| Xlambda(idt_l, xexpr, i) ->
    let return_t_s = string_of_type (Semantic.get_type xexpr) in
    let gen_idt_arg = (fun idt -> sprintf "%s %s" (string_of_type idt.t)
       idt.id) in
    let args_t = List.fold_left
        (fun s idt -> sprintf "%s, %s" s (string_of_type idt.t))
        (string_of_type (List.hd idt_l).t)
        (List.tl idt_l) in
    let args = List.fold_left
        (fun s idt -> sprintf "%s, %s" s (gen_idt_arg idt))
        (gen_idt_arg (List.hd idt_l))
        (List.tl idt_l) in
    id_counter := 1 + !id_counter;
    let ids' = (sprintf "id__%d" !id_counter) in
    sprintf "%sfunction<%s (%s)> %s = [&](%s) {\n%s%sreturn %s;\n%s};\n"
```

```
    (str_ind indent) return_t_s args_t ids args
    (translate_expr (indent+1) ids' xexpr) (str_ind (indent+1)) ids'
    (str_ind indent)
| Xmake_Vector(t, xe) ->
    let bt, dim_l = begin match t with
        | Vector(bt, dim_l) -> bt, dim_l
        | _ -> raise(Dev_Error("translate.Xmake_Vector")) end in
    let n_dim = List.length dim_l in
    let out_ts = (string_of_type (Vector(bt, dim_l))) in
    let ts = (string_of_type bt) in
    id_counter := 1 + !id_counter;
    let ids' = (sprintf "id__%d" !id_counter) in
    let exs = (translate_expr (indent+1) ids' xe) in
    let args = gen_make_vec_args n_dim in
    (sprintf "\n%sfunction< %s (%s)> __f =\n"
        (str_ind indent) ts (string_repeat "string" n_dim)) ^
    (sprintf "%s[&](%s) {\n%s%sreturn %s; };\n\n"
        (str_ind indent) args exs (str_ind (indent+1)) ids') ^
    (sprintf "%s%s %s;\n%s\n%s")
        (str_ind indent) out_ts ids (init_vector indent ids dim_l)
        (for_make_vector indent dim_l) ^
    (sprintf "%s%s%s = __f(%s);\n"
        (str_ind (indent+n_dim)) ids
        (dim_ind_make_vec n_dim)
        (ind_arg_make_vec n_dim))
| Xfuncall(fun_name, xexpr_l, t, xexpr) ->
    if fun_name = "if" then (
        id_counter := 1 + !id_counter;
        let ids_bool = (sprintf "id__%d" !id_counter) in
        id_counter := 1 + !id_counter;
        let ids_true = (sprintf "id__%d" !id_counter) in
        id_counter := 1 + !id_counter;
        let ids_false = (sprintf "id__%d" !id_counter) in
        let bool_s = (translate_expr (1+indent) ids_bool (List.hd xexpr_l
           )) in
        (sprintf "%s%s %s;\n%s\n"
            (str_ind indent) (string_of_type t) ids bool_s)^
        (sprintf"%sif( %s == bitset<1>(1)) {\n%s%s%s=%s;\n%s} else {\n%s%
           s%s=%s;\n%s}\n"
            (str_ind indent) ids_bool
            (translate_expr (1+indent) ids_true (List.nth xexpr_l 1) )
            (str_ind (indent+1)) ids ids_true (str_ind indent)
            (translate_expr (1+indent) ids_false (List.nth xexpr_l 2) )
            (str_ind (indent+1)) ids ids_false) (str_ind indent)
    ) else if fun_name = "set" then (
        id_counter := 1 + !id_counter;
        let ids_value = (sprintf "id__%d" !id_counter) in
        let cal_s = (translate_expr (1+indent) ids_value (List.nth
           xexpr_l 1)) in
        let s, setee = begin match List.hd xexpr_l with
            | Xid(s, _) -> (c_name s), (sprintf "%s%s = %s;\n"
                (str_ind indent) (c_name s) ids_value)
            | Xidd(s, xexpr_l, t) ->
                let arg_dec_and_arg_l = (List.map (fun x ->
```

```
                    id_counter := 1 + !id_counter;
                    let ids = (sprintf "id__%d" !id_counter) in
                    (translate_expr (1+indent) ids x), ids) xexpr_l) in
                let args_dec = List.fold_left (fun s1 (arg_dec, _) ->
                   sprintf "%s%s" s1 arg_dec) "" arg_dec_and_arg_l in
                let args_l = snd (List.split arg_dec_and_arg_l) in
                let vector_part_s = List.fold_left
                    (fun s ids -> s ^ "[string_of_int(" ^ ids ^ ")]")
                    ("[string_of_int(" ^ (List.hd args_l) ^ ")]")
                    (List.tl args_l) in
                ((c_name s)^vector_part_s, (sprintf "%s%s%s = %s;\n"
                    args_dec (str_ind indent) ((c_name s)^vector_part_s)
                       ids_value))
            | _ -> raise(Dev_Error("can only set Xid")) end in
        (sprintf "%s" cal_s) ^
        (sprintf "%s" setee) ^
        (sprintf "%s%s %s = %s;\n"
            (str_ind indent) (string_of_type t) ids s)
    ) else
        let arg_dec_and_arg_l = (List.map (fun x ->
            id_counter := 1 + !id_counter;
            let ids = (sprintf "id__%d" !id_counter) in
            (translate_expr (1+indent) ids x), ids) xexpr_l) in
        let args_dec = List.fold_left (fun s1 (arg_dec, _) -> sprintf "%s
           %s" s1 arg_dec) "" arg_dec_and_arg_l in
        let args = List.fold_left
            (fun s1 (_, arg) -> sprintf "%s, %s" s1 arg)
            (sprintf "%s" (snd (List.hd arg_dec_and_arg_l)))
            (List.tl arg_dec_and_arg_l) in

        args_dec ^
        if string_match (regexp "lambda_[0-9]+") fun_name 0 then begin
            id_counter := 1 + !id_counter;
            let ids' = (sprintf "id__%d" !id_counter) in
            sprintf "%s%s%s %s = %s (%s);\n"
                (translate_expr (1+indent) ids' xexpr)
                (str_ind indent) (string_of_type t) ids
                (c_fun_name ids' (Fun(Fix, t, List.map Semantic.get_type
                   xexpr_l))) args
            end
        else if (fun_name = "+" || fun_name = "*") && List.length xexpr_l
            <> 2 then
            let add_args = List.fold_left
                (fun s (_, arg) -> sprintf "%s, %s.c_str()" s arg) ""
                arg_dec_and_arg_l in
            sprintf "%s%s %s = %s (%d%s);\n"
                (str_ind indent) (string_of_type t) ids
                (c_fun_name fun_name (Fun(Fix, t, List.map Semantic.
                   get_type xexpr_l)))
                (List.length xexpr_l)
                add_args
        else
            sprintf "%s%s %s = %s (%s);\n"
                (str_ind indent) (string_of_type t) ids
```

```
                (c_fun_name fun_name (Fun(Fix, t, List.map Semantic.
                    get_type xexpr_l))) args
;;

(* translate an defvar into c code *)
let translate_defvar xdefvar =
    id_counter := 1 + !id_counter;
    let glovar = (sprintf "id__%d" !id_counter) in
    let decl_value = translate_expr 2 glovar xdefvar.xvbody in
    (* decl_var stores the type and variable name as a string *)
    let type_s = (string_of_type xdefvar.xvname.t) in

    sprintf "%s%s init_%s () {\n%s%sreturn %s;\n%s}\n%s %s = init_%s();\n
        "
        (str_ind 1) type_s glovar decl_value (str_ind 2) glovar
        (str_ind 1) type_s (c_name xdefvar.xvname.id) glovar

(* translate an defun into c code *)
let translate_xdefun xdefun =
    let return_type = string_of_type xdefun.xfname.t in
    let gen_idt_arg = (fun idt -> sprintf "%s %s" (string_of_type idt.t)
       idt.id) in
    let args = List.fold_left
        (fun s idt -> sprintf "%s, %s" s (gen_idt_arg idt))
        (gen_idt_arg (List.hd xdefun.xfargu))
        (List.tl xdefun.xfargu) in
    id_counter := 1 + !id_counter;
    let return_id = (sprintf "id__%d" !id_counter) in
    let body = translate_expr 1 return_id xdefun.xfbody in
    sprintf("%s %s (%s) {\n%s\n    return %s;\n}")
    return_type (c_name xdefun.xfname.id) args body return_id

(* translate a "clip" into c code *)
let translate_clip = function
| Xexpr(xexpr, b) ->
    id_counter := 1 + !id_counter;
    let ids = (sprintf "id__%d" !id_counter) in
    let calculation = translate_expr 1 ids xexpr in
    let c_xexpr_str = if b then
        begin match Semantic.get_type xexpr with
        | Int -> sprintf "%scout << %s << endl;\n" (str_ind 1) ids
        | Bits(i) -> sprintf "%sprintbit<%d>(%s);\ncout << endl;\n" (
           str_ind 1) i ids
        | String -> sprintf "%scout << %s << endl;\n" (str_ind 1) ids
        | Vector(t, i_l) -> sprintf "%s%scout << endl;\n" (cout_vector
           ids t i_l) (str_ind 1)
        | _ -> sprintf "    cout << \"%s is not printable yet.\" << endl;
           " ids end
    else
        "" in
    sprintf "{\n%s%s}" calculation c_xexpr_str
| Xdefvar(xdefvar) -> translate_defvar xdefvar
| Xdefun(xdefun) -> translate_xdefun xdefun
;;
```

```
let rec translate_program = function
| [] -> ""
| hd_clip::tl_clips -> translate_clip hd_clip ^ "\n" ^ translate_program
    tl_clips
;;

(* generate the dynamic built-in function in C *)
let rec gen_builtin_in_xexpr = function
| Xint_Lit(_)
| Xbin_Lit(_)
| Xhex_Lit(_)
| Xbit_Binary_Lit(_, _)
| Xbit_Hex_Lit(_, _)
| Xstring_Lit(_) -> ()
| Xvector_Lit(xe_l) -> List.iter gen_builtin_in_xexpr xe_l
| Xid(s, t) ->
    begin match t with
    | Fun(Fix, out_t, in_t_l) ->
        let fake_xexpr_l = List.map (Semantic.type_to_xexpr) in_t_l in
        gen_builtin_in_xexpr (Xfuncall(s, fake_xexpr_l ,out_t, Xint_Lit("
            123")))
    | _ -> () end
| Xidd(_, xexpr_l, _) -> List.iter gen_builtin_in_xexpr xexpr_l
| Xvec_Dimension(_) -> ()
| Xlet(let_arg_l, xe) ->
    List.iter (fun ( _, xexp) -> gen_builtin_in_xexpr xexp) let_arg_l;
    gen_builtin_in_xexpr xe
| Xlambda (idt_l, xe, i) ->
    gen_builtin_in_xexpr xe
| Xmake_Vector(t, xe) -> gen_builtin_in_xexpr xe
| Xfuncall(fun_name, xexpr_l, t, xexpr) ->
    List.iter gen_builtin_in_xexpr xexpr_l;
    let in_t = (List.map Semantic.get_type xexpr_l) in
    if string_match (regexp "lambda_[0-9]+") fun_name 0 then
        gen_builtin_in_xexpr xexpr

    else if string_match (regexp "if") fun_name 0 then
        let fun_name = (c_fun_name fun_name (Fun(Fix, t, List.map
            Semantic.get_type xexpr_l))) in
        let already = FunSet.mem fun_name !fun_db in
        if already then
            ()
        else
            (fun_db := FunSet.add fun_name !fun_db;
            fprintf dyna_out "%s\n" (gen_if (name_of_type t) (
                string_of_type t)))

    else if string_match (regexp "eq") fun_name 0 then
        let fun_name = (c_fun_name fun_name (Fun(Fix, t, List.map
            Semantic.get_type xexpr_l))) in
        let already = FunSet.mem fun_name !fun_db in
        if already then
            ()
```

```
        else
            (fun_db := FunSet.add fun_name !fun_db;
            begin match in_t with
            | [t1; t2] -> fprintf dyna_out "%s\n" (gen_eq (name_of_type
                t1) (string_of_type t1))
            | _ -> raise(Dev_Error("output of function eq should be bits"
                )) end)

    else if string_match (regexp "neq") fun_name 0 then
        let fun_name = (c_fun_name fun_name (Fun(Fix, t, List.map
            Semantic.get_type xexpr_l))) in
        let already = FunSet.mem fun_name !fun_db in
        if already then
            ()
        else
            (fun_db := FunSet.add fun_name !fun_db;
            begin match in_t with
            | [t1; t2] -> fprintf dyna_out "%s\n" (gen_neq (name_of_type
                t1) (string_of_type t1))
            | _ -> raise(Dev_Error("output of function neq should be bits
                ")) end)

    else if string_match (regexp "\\^") fun_name 0 then
        let fun_name = (c_fun_name fun_name (Fun(Fix, t, List.map
            Semantic.get_type xexpr_l))) in
        let already = FunSet.mem fun_name !fun_db in
        if already then
            ()
        else
            (fun_db := FunSet.add fun_name !fun_db;
            begin match t with
            | Bits(i) -> fprintf dyna_out "%s\n" (gen_xor i (List.length
                xexpr_l))
            | _ -> raise(Dev_Error("output of function ^ should be bits")
                ) end)

    else if (string_match (regexp "|") fun_name 0) || fun_name = "or"
        then
        let fun_name = (c_fun_name fun_name (Fun(Fix, t, List.map
            Semantic.get_type xexpr_l))) in
        let already = FunSet.mem fun_name !fun_db in
        if already then
            ()
        else
            (fun_db := FunSet.add fun_name !fun_db;
            begin match t with
            | Bits(i) -> fprintf dyna_out "%s\n" (gen_or i (List.length
                xexpr_l))
            | _ -> raise(Dev_Error("output of function | should be bits")
                ) end)

    else if (string_match (regexp "&") fun_name 0) || fun_name = "and"
        then
        let fun_name = (c_fun_name fun_name (Fun(Fix, t, List.map
```

```
            Semantic.get_type xexpr_l))) in
        let already = FunSet.mem fun_name !fun_db in
        if already then
            ()
        else
            (fun_db := FunSet.add fun_name !fun_db;
            begin match t with
            | Bits(i) -> fprintf dyna_out "%s\n" (gen_and i (List.length
                xexpr_l))
            | _ -> raise(Dev_Error("output of function & should be bits")
                ) end)

    else if string_match (regexp "parity") fun_name 0 then
        let fun_name = (c_fun_name fun_name (Fun(Fix, t, List.map
            Semantic.get_type xexpr_l))) in
        let already = FunSet.mem fun_name !fun_db in
        if already then
            ()
        else
            (fun_db := FunSet.add fun_name !fun_db;
            begin match in_t with
            | [Bits(i)] -> fprintf dyna_out "%s\n" (gen_parity i)
            | _ -> raise(Dev_Error("input of function int-of-bits should
                be bits")) end)

    else if string_match (regexp "map__[0-9]+") fun_name 0 then
        let f_in_type = Semantic.get_type (List.nth xexpr_l 1) in
        let f_in_type_s = string_of_type (decrease_dim f_in_type) in
        let f_out_type_s = string_of_type (decrease_dim t) in
        fprintf dyna_out "%s\n" (gen_map f_out_type_s fun_name
            f_in_type_s)

    else if string_match (regexp "merge__[0-9]+") fun_name 0 then
        begin match Semantic.get_type (List.hd xexpr_l) with
        | Vector(Bits(i), [len]) ->
            fprintf dyna_out "%s\n" (gen_merge (len*i) fun_name i)
        | _ -> raise(Dev_Error("wrong input of function merge not
            detected")) end

    else if string_match (regexp "group__[0-9]+") fun_name 0 then
        let in_t = Semantic.get_type (List.hd xexpr_l) in
        begin match in_t, t with
        | Bits(in_len),  Vector(Bits(out_len), _) ->
            fprintf dyna_out "%s\n" (gen_group out_len fun_name in_len)
        | _ -> raise(Dev_Error("wrong input of function group not
            detected")) end

    else if string_match (regexp "transpose__[0-9]+") fun_name 0 then
        begin match t with
        | Vector(t', [_; _]) ->
            fprintf dyna_out "%s\n" (gen_transpose (string_of_type t')
                fun_name)
        | _ -> raise(Dev_Error("wrong input of function transpose not
            detected")) end
```

```
    else if string_match (regexp "reduce__[0-9]+") fun_name 0 then
        let f_in_type = Semantic.get_type (List.nth xexpr_l 1) in
        let f_in_type_s = string_of_type (decrease_dim f_in_type) in
        let f_out_type_s = string_of_type t in
        fprintf dyna_out "%s\n" (gen_reduce f_out_type_s fun_name
            f_in_type_s)

    else if string_match (regexp ">>>") fun_name 0 then
        let fun_name = (c_fun_name fun_name (Fun(Fix, t, List.map
            Semantic.get_type xexpr_l))) in
        let already = FunSet.mem fun_name !fun_db in
        if already then
            ()
        else
            (fun_db := FunSet.add fun_name !fun_db;
            begin match t with
            | Bits(i) -> fprintf dyna_out "%s\n" (gen_rotate_r i)
            | _ -> raise(Dev_Error("output of function >>> should be bits
                ")) end)

    else if string_match (regexp "<<<") fun_name 0 then
        let fun_name = (c_fun_name fun_name (Fun(Fix, t, List.map
            Semantic.get_type xexpr_l))) in
        let already = FunSet.mem fun_name !fun_db in
        if already then
            ()
        else
            (fun_db := FunSet.add fun_name !fun_db;
            begin match t with
            | Bits(i) -> fprintf dyna_out "%s\n" (gen_rotate_l i)
            | _ -> raise(Dev_Error("output of function <<< should be bits
                ")) end)

    else if string_match (regexp ">>") fun_name 0 then
        let fun_name = (c_fun_name fun_name (Fun(Fix, t, List.map
            Semantic.get_type xexpr_l))) in
        let already = FunSet.mem fun_name !fun_db in
        if already then
            ()
        else
            (fun_db := FunSet.add fun_name !fun_db;
            begin match t with
            | Bits(i) -> fprintf dyna_out "%s\n" (gen_shift_r i)
            | _ -> raise(Dev_Error("output of function >> should be bits"
                )) end)

    else if string_match (regexp "<<") fun_name 0 then
        let fun_name = (c_fun_name fun_name (Fun(Fix, t, List.map
            Semantic.get_type xexpr_l))) in
        let already = FunSet.mem fun_name !fun_db in
        if already then
            ()
        else
```

```
            (fun_db := FunSet.add fun_name !fun_db;
            begin match t with
            | Bits(i) -> fprintf dyna_out "%s\n" (gen_shift_l i)
            | _ -> raise(Dev_Error("output of function << should be bits"
               )) end)

    else if string_match (regexp "flip-bit") fun_name 0 then
        let fun_name = (c_fun_name fun_name (Fun(Fix, t, List.map
           Semantic.get_type xexpr_l))) in
        let already = FunSet.mem fun_name !fun_db in
        if already then
            ()
        else
            (fun_db := FunSet.add fun_name !fun_db;
            begin match t with
            | Bits(i) -> fprintf dyna_out "%s\n" (gen_flip_bit i)
            | _ -> raise(Dev_Error("output of function flip-bit should be
                bits")) end)

    else if string_match (regexp "flip") fun_name 0 then
        let fun_name = (c_fun_name fun_name (Fun(Fix, t, List.map
           Semantic.get_type xexpr_l))) in
        let already = FunSet.mem fun_name !fun_db in
        if already then
            ()
        else
            (fun_db := FunSet.add fun_name !fun_db;
            begin match t with
            | Bits(i) -> fprintf dyna_out "%s\n" (gen_flip i)
            | _ -> raise(Dev_Error("output of function flip should be
               bits")) end)

    else if string_match (regexp "zero__[0-9]+") fun_name 0 then
        begin match t with
        | Bits(i) -> fprintf dyna_out "%s\n" (gen_zero i fun_name)
        | _ -> raise(Dev_Error("output of function zeros should be bits")
           ) end

    else if string_match (regexp "rand__[0-9]+") fun_name 0 then
        begin match t with
        | Bits(i) -> fprintf dyna_out "%s\n" (gen_rand i fun_name)
        | _ -> raise(Dev_Error("output of function rand should be bits"))
            end

    else if string_match (regexp "int-of-bits") fun_name 0 then
        let fun_name = (c_fun_name fun_name (Fun(Fix, t, List.map
           Semantic.get_type xexpr_l))) in
        let already = FunSet.mem fun_name !fun_db in
        if already then
            ()
        else
            (fun_db := FunSet.add fun_name !fun_db;
            begin match in_t with
            | [Bits(i)] -> fprintf dyna_out "%s\n" (gen_int_of_bits i)
```

```
            | _ -> raise(Dev_Error("input of function int-of-bits should
               be bits")) end)

    else if string_match (regexp "string-of-bits") fun_name 0 then
        let fun_name = (c_fun_name fun_name (Fun(Fix, t, List.map
           Semantic.get_type xexpr_l))) in
        let already = FunSet.mem fun_name !fun_db in
        if already then
            ()
        else
            (fun_db := FunSet.add fun_name !fun_db;
            begin match in_t with
            | [Bits(i)] -> fprintf dyna_out "%s\n" (gen_string_of_bits i)
            | _ -> raise(Dev_Error("input of function string-of-bits
               should be bits")) end)

    else if string_match (regexp "bits-of-int") fun_name 0 then
        let fun_name = (c_fun_name fun_name (Fun(Fix, t, List.map
           Semantic.get_type xexpr_l))) in
        let already = FunSet.mem fun_name !fun_db in
        if already then
            ()
        else
            (fun_db := FunSet.add fun_name !fun_db;
            begin match t with
            | Bits(i) -> fprintf dyna_out "%s\n" (gen_bits_of_int i)
            | _ -> raise(Dev_Error("output of function bits-of-int should
                be bits")) end)

    else if string_match (regexp "bits-of-string") fun_name 0 then
        let fun_name = (c_fun_name fun_name (Fun(Fix, t, List.map
           Semantic.get_type xexpr_l))) in
        let already = FunSet.mem fun_name !fun_db in
        if already then
            ()
        else
            (fun_db := FunSet.add fun_name !fun_db;
            begin match t with
            | Bits(i) -> fprintf dyna_out "%s\n" (gen_bits_of_string i)
            | _ -> raise(Dev_Error("output of function bits-of-string
               should be bits")) end)

    else if string_match (regexp "pad") fun_name 0 then
        let fun_name = (c_fun_name fun_name (Fun(Fix, t, List.map
           Semantic.get_type xexpr_l))) in
        let already = FunSet.mem fun_name !fun_db in
        if already then
            ()
        else
            (fun_db := FunSet.add fun_name !fun_db;
            begin match t, List.hd in_t with
            | Bits(i), Bits(j) -> fprintf dyna_out "%s\n" (gen_pad i j)
            | _ -> raise(Dev_Error("output of function pad should be bits
               ")) end)
```

```
    else ()
;;

let gen_builtin_in_clip = function
| Xexpr(xexpr, _) -> gen_builtin_in_xexpr xexpr
| Xdefvar(xdefvar) -> gen_builtin_in_xexpr xdefvar.xvbody
| Xdefun(xdefun) -> gen_builtin_in_xexpr xdefun.xfbody
;;

let gen_builtin_in_program p =
    List.iter gen_builtin_in_clip p
;;

let translate_to_c program =
    fprintf dyna_out "#include <cstdlib>\n#include <ctime>\n#include <
       vector>\n#include <bitset>\n\nusing namespace std;\n\nvoid init(){
       srand(time(NULL));}\n\n";
    (gen_builtin_in_program program);
    (c_header_generator c_header_list) ^
    "using namespace std;\n\n" ^
    let def_l, exp_l = List.partition (fun c ->
        match c with
        | Xexpr(xexpr, _) -> false
        | _ -> true) program in
    (translate_program def_l) ^
    c_main_left ^
    (translate_program exp_l) ^
    c_main_right
;;


end
```

Listing 16: translate.mli

```
open Ast
open Sast

module Translate :
  sig
    val translate_expr: int -> string -> xexpr -> string
    val translate_defvar: xdefvar -> string
    val translate_xdefun: xdefun -> string
    val translate_program: xprogram -> string
    val translate_to_c: xprogram -> string
  end
```

Listing 17: clip.ml

```
open Ast
open Sast
open Translate
```

```
open Printast
open Printsast
open Semantic
open Exception
open Printf

let err_pre = "~~Error~~";;
let bug_out = open_out "debug.cc";;

let _ =
    try
    let input =
        if Array.length Sys.argv > 1 then
            open_in Sys.argv.(1)
        else stdin in

    let lexbuf = Lexing.from_channel input in
    let ast = Parser.program Scanner.token lexbuf in
    (* Printast.print_ast ast; *)
    let sast = Semantic.check_ast ast in
    (* Printsast.print_sast sast *)
    let result = Translate.translate_to_c sast in
    fprintf stdout "%s" result
    with
    | Illegal_Char(c) ->
        fprintf stderr "%s Illegal character %c.\n" err_pre c
    | Illegal_Id(s) ->
        fprintf stderr "%s Identifier cannot contain %s.\n" err_pre s

    | Parsing.Parse_error ->
        fprintf stderr "%s Parse error.\n" err_pre
    | Invalid_Vector(s) ->
        fprintf stderr "%s Invalid vector %s.\n" err_pre s
    | Indefined_Id(s) ->
        fprintf stderr "%s %s is not defined.\n" err_pre s
    | Vector_Dim(s, i) ->
        fprintf stderr "%s %s has only %d dimension.\n" err_pre s i
    | Invalid_Ind(s) ->
        fprintf stderr "%s Invalid index in %s\n" err_pre s
    | Not_Vector(s) ->
        fprintf stderr "%s %s is not a vector.\n" err_pre s
    | Make_Vec_Bound(i) ->
        fprintf stderr "%s @%d is out of bound.\n" err_pre i
    | Wrong_Argu_Len(s) ->
        fprintf stderr "%s Wrong number of arguments in function call %s
            .\n"
        err_pre s
    | Wrong_Argu_Type(s) ->
        fprintf stderr "%s Wrong argument types in function call %s.\n"
           err_pre s
    | Bind_Wrong_Type(s) ->
        fprintf stderr "%s Cannot bind %s (wrong type).\n" err_pre s
    | Eval_Fail(s) ->
        fprintf stderr "%s Cannot evaluate %s.\n" err_pre s
```

```
    | Not_Function(s) ->
        fprintf stderr "%s %s is not a function.\n" err_pre s
    | Bind_Twice(s) ->
        fprintf stderr "%s Cannot define %s twice.\n" err_pre s
    | Defun_Wrong_Type(s) ->
        fprintf stderr "%s Return types in definition of %s are
           inconsistent.\n"
        err_pre s

    | Failure(s) ->
        fprintf stderr "Not yet implemented - %s\n" s
    | Dev_Error(s) ->
        fprintf stderr "Dev error ... %s\n" s
    (*| _ -> fprintf stderr "unexpected error ...\n"*)
;;
```

Listing 18: scanner.mll

```
{
    open Parser
    open Char
    open Exception
}

let space = [' ' '\t' '\r' '\n']

rule token = parse
| space                      { token lexbuf }
| "~~>"                      { comment lexbuf }
| "~~"                       { line_comment lexbuf }

| "defun"                    { DEFUN }
| "defvar"                   { DEFVAR }

| "int"                      { INT }
| "bit#" (['0'-'9']+ as lit) { BITS(int_of_string lit) }
| "string"                   { STR }
| "fun"                      { FUN }

| "let"                      { LET }
| "lambda"                   { LAMBDA }
| "make-vector"              { MAKE_VECTOR }

(* Prevent consecutive -- in identifier *)
| "--" as c                  { raise (Illegal_Id(c)) }
| ['A'-'Z' 'a'-'z'] ['A'-'Z' 'a'-'z' '0'-'9' '-']* as lit   { ID(lit) }

| "0b"(['0' '1']+ as lit)                 { BINARY(lit)}
| "0x"(['0'-'9' 'A'-'F' 'a'-'f']+ as lit) { HEX(lit)}
| "'"(['0' '1']+ as lit)                  { BIT_BINARY(lit)}
| "'x"(['0'-'9' 'A'-'F' 'a'-'f']+ as lit) { BIT_HEX(lit)}
| ['0'-'9']+ as lit             { INTEGER(lit) }
| '"' ([^ '"']* as content) '"'           { STRING(content) }
```

```
(* Special identifier used by built-in functions *)
| ">>>" as lit                              { ID(lit) }
| "<<<" as lit                              { ID(lit) }
| ">>"  as lit                              { ID(lit) }
| "<<"  as lit                              { ID(lit) }
| "&"   as lit                              { ID(escaped lit) }
| "|"   as lit                              { ID(escaped lit) }
| "^"   as lit                              { ID(escaped lit) }
| "+"   as lit                              { ID(escaped lit) }
| "-"   as lit                              { ID(escaped lit) }
| "*"   as lit                              { ID(escaped lit) }
| "/"   as lit                              { ID(escaped lit) }

| '='                                      { ASSIGN }
| '@'(['0'-'9']+ as lit)                   { VECDIMENSION(int_of_string lit)
  }

| ':'                      { COLON }
| ';'                      { SEMI }
| '('                      { LPAREN }
| ')'                      { RPAREN }
| '{'                      { LBRACE }
| '}'                      { RBRACE }
| '<'                      { LANGLE }
| '>'                      { RANGLE }
| '['                      { LBRACK }
| ']'                      { RBRACK }

| eof                      { EOF }
| _ as c                   { raise (Illegal_Char(c)) }
and comment = parse
| "<~~"                    { token lexbuf }
| _                        { comment lexbuf }
and line_comment = parse
| ['\n' '\r'] { token lexbuf }
| eof         { token lexbuf }
| _           { line_comment lexbuf }
```

Listing 19: builtin.cc

```
#include <iostream>
using namespace std;
#include <cstdlib>
#include <gmp.h>
#include <cstdarg>
#include <vector>
#include <bitset>
#include <string>

template <int N>
void printbit(const bitset<N> &b) {
    if (N % 4 == 0) {
        cout << "'x";
        for (int i = 0; i < b.size(); i = i+4) {
```

```
            int n = b[i]*8 + b[i+1]*4 + b[i+2]*2 + b[i+3];
            printf("%X", n);
        }
    }
    else
        cout << "'" <<  b;
}

template <int arr_l, int bit_l>
bitset<arr_l*bit_l> merge(bitset<bit_l>* bs) {
    string s = "";
    for (int i = 0; i < arr_l; i++) {
        s = s + bs[i].to_string();
    }
    return bitset <arr_l*bit_l>(s);
}

int string_of_int (string s) {
    return atoi(s.c_str());
}

string bin_of_hex (string s) {
    string result = "";
    for (int i = 0; i < s.length(); ++i) {
        string digit = s.substr(i, 1);
        char *cp;
        int n = strtol(digit.c_str(), &cp, 16);
        digit = bitset<4>(n).to_string();
        result += digit;
    }
    return result;
}

string dec_of_bin (string s) {
    char *cp = new char [s.length()];
    strcpy(cp, s.c_str());
    mpz_t n;
    mpz_init(n);
    mpz_set_str(n, cp, 2);
    mpz_get_str(cp, 10, n);
    return string(cp);
}

string dec_of_hex (string s) {
    char *cp = new char [2*s.length()];
    strcpy(cp, s.c_str());
    mpz_t n;
    mpz_init(n);
    mpz_set_str(n, cp, 16);
    mpz_get_str(cp, 10, n);
    return string(cp);
}

string add2(string a, string b) {
```

```
    mpz_t sum, ia, ib;
    mpz_init(sum);
    mpz_init_set_str(ia, a.c_str(), 10);
    mpz_init_set_str(ib, b.c_str(), 10);
    mpz_add(sum, ia, ib);
    return string(mpz_get_str(NULL, 10, sum));
}

string add(int n, ... ) {
    va_list arguments;
    string result = "";
    mpz_t sum;
    mpz_init_set_si(sum, 0);
    va_start (arguments, n);
    for (int i = 0; i < n; i++) {
        char *c = va_arg (arguments, char*);
        mpz_t tmp;
        mpz_init_set_str(tmp, c, 10);
        mpz_add(sum, sum, tmp);
    }
    va_end (arguments);
    return string(mpz_get_str(NULL, 10, sum));
}

string mul2(string a, string b) {
    mpz_t sum, ia, ib;
    mpz_init(sum);
    mpz_init_set_str(ia, a.c_str(), 10);
    mpz_init_set_str(ib, b.c_str(), 10);
    mpz_mul(sum, ia, ib);
    return string(mpz_get_str(NULL, 10, sum));
}

string mul(int n, ... ) {
    va_list arguments;
    string result = "";
    mpz_t sum;
    mpz_init_set_si(sum, 1);
    va_start (arguments, n);
    for (int i = 0; i < n; i++) {
        char *c = va_arg (arguments, char*);
        mpz_t tmp;
        mpz_init_set_str(tmp, c, 10);
        mpz_mul(sum, sum, tmp);
    }
    va_end (arguments);
    return string(mpz_get_str(NULL, 10, sum));
}

string divide(string a, string b) {
    string s;char* c;
    char* a0 = new char [a.length()+1];
    char* b0 = new char [b.length()+1];
    strcpy(a0,a.c_str());
```

```
    strcpy(b0,b.c_str());
    mpz_t a1;
    mpz_t a2;
    mpz_t a3;
    mpz_init(a1);
    mpz_init(a2);
    mpz_init(a3);
    mpz_set_str(a1,a0,10);
    mpz_set_str(a2,b0,10);
    mpz_tdiv_q(a3,a1,a2);
    c = mpz_get_str(NULL,10,a3);
    s = string(c);
    delete[] a0;
    delete[] b0;
    delete[] c;
    return s;
}

string subtract(string a, string b) {
    string s;char* c;
    char* a0 = new char [a.length()+1];
    char* b0 = new char [b.length()+1];
    strcpy(a0,a.c_str());
    strcpy(b0,b.c_str());
    mpz_t a1;
    mpz_t a2;
    mpz_t a3;
    mpz_init(a1);
    mpz_init(a2);
    mpz_init(a3);
    mpz_set_str(a1,a0,10);
    mpz_set_str(a2,b0,10);
    mpz_sub(a3,a1,a2);
    c = mpz_get_str(NULL,10,a3);
    s = string(c);
    delete[] a0;
    delete[] b0;
    delete[] c;
    return s;

}

string mod(string a, string b) {
    string s;char* c;
    char* a0 = new char [a.length()+1];
    char* b0 = new char [b.length()+1];
    strcpy(a0,a.c_str());
    strcpy(b0,b.c_str());
    mpz_t a1;
    mpz_t a2;
    mpz_t a3;
    mpz_t a4;
    mpz_init(a1);
    mpz_init(a2);
```

```
    mpz_init(a3);
    mpz_init(a4);
    mpz_set_str(a1,a0,10);

    mpz_set_str(a2,b0,10);

    mpz_set_str(a3,"1",10);
    mpz_powm(a4,a1,a3,a2);

    c = mpz_get_str(NULL,10,a4);
    s = string(c);
    delete[] a0;
    delete[] b0;
    delete[] c;
return s;
}

string power(string a, string b) {
   string s;char* c;
   char* a0 = new char [a.length()+1];
   char* b0 = new char [b.length()+1];
   strcpy(a0,a.c_str());
   strcpy(b0,b.c_str());
   mpz_t a1;
   mpz_t a2;
   mpz_t a3;
   unsigned long a4;
   mpz_init(a1);
   mpz_init(a2);
   mpz_init(a3);
   mpz_set_str(a1,a0,10);
   mpz_set_str(a2,b0,10);
   a4 = mpz_get_ui(a2);
   mpz_pow_ui(a3,a1,a4);
   c = mpz_get_str(NULL,10,a3);
   s = string(c);
   delete[] a0;
   delete[] b0;
   delete[] c;
   return s;

}

string inverse(string a, string b) {
    string s;char* c;
    char* a0 = new char [a.length()+1];
    char* b0 = new char [b.length()+1];
    strcpy(a0,a.c_str());
    strcpy(b0,b.c_str());
    mpz_t a1;
    mpz_t a2;
    mpz_t a3;
    mpz_init(a1);
    mpz_init(a2);
```

```
    mpz_init(a3);
    mpz_set_str(a1,a0,10);
    mpz_set_str(a2,b0,10);
    mpz_invert(a3,a1,a2);
    c = mpz_get_str(NULL,10,a3);
    s = string(c);
    delete[] a0;
    delete[] b0;
    delete[] c;
    return s;
}

bitset<1> not__(bitset<1> b) {
    if (b[0] == 1)
        return bitset<1>(0);
    else
        return bitset<1>(1);
}

bitset<1> less__(string x, string y) {
    mpz_t a, b;
    mpz_init_set_str(a, x.c_str(), 10);
    mpz_init_set_str(b, y.c_str(), 10);
    int c = mpz_cmp(a, b);
    if (c < 0)
        return bitset<1>(1);
    else
        return bitset<1>(0);
}

bitset<1> greater__(string x, string y) {
    mpz_t a, b;
    mpz_init_set_str(a, x.c_str(), 10);
    mpz_init_set_str(b, y.c_str(), 10);
    int c = mpz_cmp(a, b);
    if (c > 0)
        return bitset<1>(1);
    else
        return bitset<1>(0);
}

bitset<1> leq__(string x, string y) {
    mpz_t a, b;
    mpz_init_set_str(a, x.c_str(), 10);
    mpz_init_set_str(b, y.c_str(), 10);
    int c = mpz_cmp(a, b);
    if (c <= 0)
        return bitset<1>(1);
    else
        return bitset<1>(0);
}

bitset<1> geq__(string x, string y) {
    mpz_t a, b;
```

```
    mpz_init_set_str(a, x.c_str(), 10);
    mpz_init_set_str(b, y.c_str(), 10);
    int c = mpz_cmp(a, b);
    if (c >= 0)
        return bitset<1>(1);
    else
        return bitset<1>(0);
}

bitset<1> is_prime(string s) {
    mpz_t a;
    mpz_init_set_str(a, s.c_str(), 10);
    int isp = mpz_probab_prime_p(a, 25);
    if (isp == 1 || isp == 2)
        return bitset<1>(1);
    else
        return bitset<1>(0);
}

string next_prime(string s) {
    mpz_t a;
    mpz_t b;
    mpz_init(b);
    mpz_init_set_str(a, s.c_str(), 10);
    int isp = mpz_probab_prime_p(a, 25);
    mpz_nextprime(b, a);
    return string(mpz_get_str(NULL, 10, b));
}
```

Listing 20: clipccore

```
CLIP="./clip"
CC="clang++"
keep=0

SOURCE=$1
#FILENAME=$(basename "$SOURCE")
FILENAME=$SOURCE
EXTENSION="${FILENAME##*.}"
FILENAME="${FILENAME%.*}"
TARGET=$FILENAME
CFILE="$FILENAME.cc"

#echo "source: $SOURCE, target: $TARGET, cfile: $CFILE"

if [ "$EXTENSION" != "clip" ]; then
    echo "The filename extension should be \"clip\""
    exit 1
fi

shift $((OPTIND))
while getopts ":co:" opt; do
    case $opt in
        c) # Keep .c files
```

```
            keep=1
            #echo "ccc, Parameter: $OPTARG"
            ;;
        o) # Specify the output filename
            #echo "ooo, Parameter: $OPTARG"
            TARGET=$OPTARG
            ;;
        \?)
            echo "Invalid option: -$OPTARG"
            exit 1
            ;;
        :)
            echo "Option -$OPTARG requires an argument." >&2
            exit 1
            ;;
    esac
done

$CLIP < $SOURCE > $CFILE

$CC -std=c++11 -stdlib=libc++ -lgmp -I$HOME -O3 $CFILE -o $TARGET

if [ $keep == "0" ]; then
    rm -f $CFILE;
fi
#Run "$CLIP"  "<" $SOURCE "> $CFILE"
```

Listing 21: gmp.sh

```
#!/bin/sh

#Below is the directory where gmp library is saved
HOME=/Users/sylvia_duan/Documents/Columbia/PLT/gmp-5.1.2

cd $HOME

    ./configure &&
    make &&
    make check &&
    make &&
    make install
```

Listing 22: testall.sh

```
#!/bin/sh

CLIPC="./clipc"
CLIP="./clip"

HOME=/Users/sylvia_duan/Documents/Columbia/PLT/project/clip

#Set time limit for all operations
ulimit -t 30
```

```
10
globallog=testall.log
rm -f $globallog
error=0
globalerror=0
15
keep=0



20  SignalError() {
        if [ $error -eq 0 ] ; then
            echo "FAILED"
            error=1
        fi
25          echo "  $1"
    }

    # Compare <outfile> <reffile> <difffile>
    # Compares the outfile with reffile.  Differences, if any, written to
       difffile
30  Compare() {
        diff -b "$1" "$2" > "$3" 2>&1 ||
        {
          SignalError "$1 differs" > $ERRORFILE
        }
35  }

    Check() {
        error=0

40      filename=$(basename "$1")
        FILENAME=$1
        FILENAME="${FILENAME%.clip}"
        filename="${filename%.clip}"
        TESTFILE="$FILENAME.clip"
45      ERRORFILE="$FILENAME.error"
        REFFILE="$FILENAME.ref"
        OUTTREE="$FILENAME.tree"
        OUTFILE="$FILENAME.out"
        CCFILE="$FILENAME.cc"
50      TARGET="$FILENAME"
        DIFFFILE="$FILENAME.DIFF"

        printf "Testing $filename ... "
        $CLIPC $TESTFILE -c 2> $ERRORFILE
55      if [[ -x "$TARGET" ]] ; then
            ./$TARGET > $OUTFILE 2> $ERRORFILE
            Compare $OUTFILE $REFFILE $DIFFFILE
            if [ $error -eq 0 ] ; then
                rm -f $DIFFFILE
60              rm -f $CCFILE
                rm -f $FILENAME
                printf "\r\t\t\t\t Accept!!"
```

```
        else
            printf "\r\t\t\t\t Wrong Answer"
            globalerror=$error
        fi
    else
        printf "\r\t\t\t\t Not executable or found, refer .error file"
    fi
    echo ""
}

if [ $# -ge 1 ] ; then
    files=$@
else
    files="test/*/*.clip"
fi

rm -f test/*/*.error
rm -f test/*/*.out

for file in $files
do
    case $file in
        *test-*)
        Check $file
          ;;
    esac
done

exit $globalerror
```

Listing 23: Makefile

```
OBJS = ast.cmo sast.cmo exception.cmo parser.cmo scanner.cmo \
printast.cmo builtin.cmo semantic.cmo printsast.cmo translate.cmo clip.
    cmo

all: clip clipc

clip: $(OBJS)
    ocamlc -o clip str.cma $(OBJS)

scanner.ml: scanner.mll
    ocamllex scanner.mll

parser.ml parser.mli: parser.mly
    ocamlyacc parser.mly

%.cmo: %.ml
    ocamlc -c $<

%.cmi: %.mli
    ocamlc -c $<

```

```
HOME=`pwd`
clipc: clip
	echo "#!/bin/bash" > $@
	echo "HOME=$(HOME)" >> $@
	cat clipccore >> $@
	chmod +x clipc

.PHONY: clean
clean:
	rm -f parser.ml parser.mli scanner.ml *.cmo *.cmi

# Generated by ocamldep *.ml *.mli
ast.cmo :
ast.cmx :
builtin.cmo : sast.cmo ast.cmo
builtin.cmx : sast.cmx ast.cmx
clip.cmo : sast.cmo translate.cmi semantic.cmi printsast.cmo printast.cmo \
    parser.cmi exception.cmo ast.cmo
clip.cmx : sast.cmx translate.cmx semantic.cmx printsast.cmx printast.cmx \
    parser.cmx exception.cmx ast.cmx
exception.cmo :
exception.cmx :
parser.cmo : ast.cmo parser.cmi
parser.cmx : ast.cmx parser.cmi
printast.cmo : exception.cmo ast.cmo
printast.cmx : exception.cmx ast.cmx
printsast.cmo : sast.cmo semantic.cmi printast.cmo exception.cmo ast.cmo
printsast.cmx : sast.cmx semantic.cmx printast.cmx exception.cmx ast.cmx
semantic.cmo : sast.cmo printast.cmo exception.cmo builtin.cmo ast.cmo \
    semantic.cmi
semantic.cmx : sast.cmx printast.cmx exception.cmx builtin.cmx ast.cmx \
    semantic.cmi
translate.cmo : sast.cmo printast.cmo exception.cmo ast.cmo translate.cmi
translate.cmx : sast.cmx printast.cmx exception.cmx ast.cmx translate.cmi
sast.cmo : ast.cmo
sast.cmx : ast.cmx
parser.cmi : ast.cmo
semantic.cmi : sast.cmo ast.cmo
translate.cmi : sast.cmo ast.cmo
```