COMS W4115 Fall 2012
Course Project
EasKey Final Report

Keqiu Hu (kh2567)

Jinqi Huang (jh3297)

Xiaoyu Huang (xh2185)

Zongheng Wang (zw2223)

Lizhong Zhang (lz2324)

Columbia University

December 19th, 2012

Contents

Chapter 1

An Introduction to EasKey

EasKey aims to facilitate programming of keyboard and mouse actions. It consists of most fundamental operations and functions of regular programming languages, such as branches and loops. Besides, all the potential keyboard and mouse actions are realized by EasKey internal functions and integrated tightly with the compiler, which makes the process of keyboard and mouse programming super easy for users.

1.1 Motivation

Technology makes people's life easier. Computer is supposed to help people complete iterative and boring calculations. However, with the high-speed development of software industry, software installed in a computer becomes more and more complicated and often requires thousands of mouse clicks and keyboard inputs, most of which are repetitive and boring.

Take a few examples, in the course registration period, you may wish to register a class that is already full. So the most common way may be that you constantly press the “Class Search” button to refresh the website in order to see that if someone has just dropped the course so you are lucky enough to fill in the position. Or perhaps you need to update a database system with thousands of operations that are similar to each other, so you have to input all the instructions one by one. Or you may want to click your mouse as fast as possible to win a game which needs immediate reaction.

In a circumstance like any of these, it is definitely a waste of time to do finish the task in normal ways presented above. So, is there something we can do to make it easier and faster and more efficient? If what we do is simply a combination of keyboard inputs and mouse clicks, why don't we let the computer itself to finish the task? These results in our idea to design a language to facilitate people doing iterative keyboard and mouse input. And the solution is our EasKey Languge!

1.2 Description

Our EasKey language is developed using OCaml language, and it is text-based and has a lot of similarity with C language. We support all the basic types and functionality defined in microC, which is the mainly reference language that we turned to, and we have built lots of build in function to facilitate the basic operation of mouse and keyboard. In EasKey, we define the same types of statements as in C but with different grammar, which help to better organize the structure of program. The basic idea and rules in EasKey is to discard the use of parenthesis “{ }” as the notation of block and replace it with key word “end”, besides, make the best convenience for users to manipulate the windows API operations with just sparse code. With EasKey language, user can get the position of cursor and keep it clicking any button thousands of time. Also, just like many other primary language, EasKey supports user self-defined function and the basic loop or recursive function.

1.3 Features

Functional

The mainly advantage that EasKey has over other language is the integration of several operations of Mouse and keyboard in windows environment. It makes programming easy for the user who doesn’t have much programming experience.

Abundant Data Types

EasKey is, to some extent, derived from microC, but far more powerful than microC. We have several data types, including basic Integer, Float, String, Char, Boolean and the extended data type, point, color and key.

Statically-scoped

EasKey is statically-scoped which means that the scope of a defined variable in EasyKey will be extended when a function is called later.

Applicative-ordered

Just like C, EasKey uses applicative-order during the evaluation.

Robust

EasKey has a very detailed-checking analyzer, which helps to minimize the potential bug in the program.

Convenient and Interesting

As we have stated, EasKey is designed to facilitate people to get rid of boring routine computer operation, which we think is very interesting and commercial promising.

Chapter 2

Language Tutorial

EasKey is a really simple language that can easily get started with. User really don't need to have much knowledge about programming and can easily use the inherited function to build the program they want. Now, let's start the tutorial with a simple example.

2.1 Introductory Sample

This is a very simple function that print a string "EasKey" to the terminal:

```
function void out(string s)
      __print(s);
      return;
end function

function int main()
      string s=”EasKey”;
      out(s);
      return 0;
end function
```

2.2Functionalities

2.2.1 Variable declaration

We have 8 kinds of variables, and the declaration should follow the form as bellow:

```
int a=1;
string b=”EasKey”;
char c=”E”;
float d=3.1415;
boolean e=true;
point f={50,50};
key g=$F$;
color h=#123,456,789#;
```

2.2.2 Loop

There are two kinds of loop can be defined in our language:
For loop:

```
for (i=0;i<10;i=i+1)
    forstatement;
end for
```

While loop:

```
while (i !=0)
    whilestatement;
end while
```

2.2.3 recursion

Our language support recursion, which means a function can call itself in its function body

```
function int fib (int n)
    if(n == 1)
        return 1;
    elseif(n == 2)
        return 1;
    else
        return (fib(n-1)+fib(n-2));
    end if
end function
```

2.2.4 condition

There are two types of condition control
if :

```
if (a==b)
    a=a+1;
    else b=b+1;
end if
```

switch:

```
int i;
switch (i)
    case i: a = 1;
    case i: a = 2;
    case i: a = 3;
```

```
          default: a = 4;
     end switch
```

2.2.5 function

We have defined a lot of built in functions which can help user better achieve their goal:

void __leftClickPoint(point p);	-----------perform mouse left click at point p
void __rightClickPoint(point p);	----------perform mouse right click at point p
void __movePoint(point p);	----------move cursor to point p
void __leftDown();	-------perform mouse left click at current point
void __rightDown();	-------perform mouse right click at current point
void __keyStroke (key k);	-------press key k on the keyboard
point __getPoint();	-------get the point value of current cursor location
color __getcolor(point p);	-------get the color value of certain point p
void __print(string s);	-------print out a string s in the terminal
string __stringOfInt(int i);	-------convert a variable of type int to type string
void __delay(int t);	-------delay t time

Besides, user can build their own function with the form like this:

```
function foo(int i, int j)

    if (i==j)
        __print(“equal”);
    else __print(“nonequal”);
    end if

end function
```

Chapter 3

Language Reference Manual

3.1 Lexical conventions

Four kinds of tokens are included in EasKey: identifiers, keywords, constants and expression operators. Blanks, tabs, newlines and comments can only serve to separate tokens, otherwise they will be ignored. Tokens are distinguished by adding at least one of the characters above to separate them.

3.1.1 Comments

Any character between a /* and the first */ after that are considered as a comment. Nested /* and */ pairs are illegible.

3.1.2 Identifiers

An identifier is a sequence of letters and digits; the first character must be alphabetic. The underscore ‘‘_’’ counts as alphabetic. Upper and lower case letters are considered different.

3.1.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

void
float
string
point
key
elseif
while
continue
switch
default
return
true
int
char
boolean
color
if
else
for
break
case
function
end
false

3.1.4 Constants

There are several kinds of constants as follows.

3.1.4.1 Integer constants

An integer constant is a sequence of digits.

3.1.4.2 Character constants

A character constant is one or two characters enclosed in single quotes ‘‘′ ’’. Within a character constant a single quote must be preceded by a backslash “\’’. Certain non-graphic characters, and ‘‘\’’ itself, may be escaped according to the following table:

back space	\b
newline	\n
carriage return	\r
horizontal tab	\t
null byte	\0
ddd	\ddd
\	\\

The escape ‘‘\ddd’’ consists of the backslash followed by exact 3 digits which are taken to specify the value of the desired character. The number ddd must fall in the region 000 to 255.

3.1.4.3 Floating constants

A floating constant consists of an integer part, a decimal point, a fraction part, an e, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing.

3.1.4.4 Boolean constants

A boolean constant only has two possible values: true or false.

3.1.4.5 Point constants

A point constant is two integer constants separated by a comma “,” and enclosed in parentheses “{”, “}”, indicating the coordinates of a pixel in the computer screen.

3.1.4.6 Key constants

A key constant is one or more characters enclosed in “$”, indicating the most of the keys of an ordinary keyboard. The key constants are as follows:

\$A\$	\$B\$	\$C\$	\$D\$						
\$E\$	\$F\$	\$G\$	\$H\$	\$I\$	\$J\$	\$K\$	\$L\$	\$M\$	\$N\$
\$O\$	\$P\$	\$Q\$	\$R\$	\$S\$	\$T\$	\$U\$	\$V\$	\$W\$	\$X\$
\$Y\$	\$Z\$	\$0\$	\$1\$	\$2\$	\$3\$	\$4\$	\$5\$	\$6\$	\$7\$
\$8\$	\$9\$	\$`\$	\$~\$	\$!\$	\$@\$	\$#\$	\$\$\$	\$%\$	\$^\$

$&$ $*$ $($ $)$ $-$ $_$ $+$ $=$ $[$ ${$
$]$ $}$ $\$ $|$ $;$ $:$ $'$ $"$ $,$ $<$
$.$ $>$ $/$ $?$

$Enter$ $Space$ $Backspace$ Esc Tab $CapsLk$
$Shift$ $Ctrl$ Alt $Windows$ Up $Down$
$Left$ $Right$ $Delete$ $Home$ End $PgUp$
$PgDn$ $PrtSc$ $ScrLk$ $Insert$ $F1$ $F2$
$F3$ $F4$ $F5$ $F6$ $F7$ $F8$
$F9$ $F10$ $F11$ $F12$ $Null$

3.1.4.7 Color constants

A color constant consists of three integer constants enclosed in “#”, indicating the color of pixel in the computer screen. The three integers must fall in the region 000 to 255, representing the three parameters in a RGB color model.

3.1.4.8 String constants

A string is a sequence of characters surrounded by double quotes “"”. String itself is a type. In a string, the character “"” must be preceded by a ‘‘\’’; in addition, the same escapes as described for character constants may be used.

3.2 Conversions

EasKey supports the following type conversions based on built-in functions:
string __stringOfInt(int i);
string __stringOfFloat(float f);
string __stringOfChar(char c);
string __stringOfBoolean(boolean b);
string __stringOfPoint(point p);
string __stringOfKey(key k);
string __stringOfColor(color c);

3.3 Expressions

3.3.1 Primary expressions

Primary expressions include the following.

3.3.1.1 Identifier

An identifier is a primary expression, provided it has been suitably declared as discussed below. Its type is specified by its declaration.

3.3.1.2 Constant

Any constant is a primary expression.

3.3.1.3 (expression)

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression.

3.3.1.4 primary-expression (expression-listopt)

A function call is a primary expression followed by parentheses containing a possibly empty, comma separated list of expressions which constitute the actual arguments to the function. The result of the function call is of return type of that function.

3.3.2 Unary operators

Expressions with unary operators group right-to-left.

3.3.2.1 –expression

The result is the negative of the expression, and has the same type. The type of the expression must be int or float.

3.3.2.2 !expression

The result of the logical negation operator ! is true if the value of the expression is false, false if the value of the expression is true. Note that this operator is applicable to int or char type.

3.3.2.3 expression @x

The result is the x-coordinate of the expression, and has the integer type. The type of the expression must be point.

3.3.2.4 expression @y

The result is the y-coordinate of the expression, and has the integer type. The type of the expression must be point.

3.3.2.5 expression @r

The result is the red parameter of the expression, and has the integer type. The type of the expression must be color.

3.3.2.6 expression @g

The result is the green parameter of the expression, and has the integer type. The type of the expression must be color.

3.3.2.7 expression @b

The result is the blue parameter of the expression, and has the integer type. The type of the expression must be color.

3.3.2.8 expression @k

The result is the string of the expression, and has the string type. The type of the expression must be key.

3.3.3 Additive operators

3.3.3.1 expression + expression

The result is the sum of the expressions. Both operands must have the same type, which should be int, char, float. If the expressions are of type point or color, then each field is added up respectively. If the expressions are of type string, then the string are concatenated.

3.3.3.2 expression – expression

The result is the difference of the operands. Same type consideration is applied here, except that type string cannot be used here.

3.3.3 Multiplicative operators

3.3.1 *expression * expression*

The binary * operator indicates multiplication. Both operands must be int or float, and the result is of the same type.

3.3.3.4 expression / expression

The binary / operator indicates division. The same type considerations as for multiplication apply.

3.3.3.5 expression % expression

The binary % operator obtains the remainder from the division of the first expression by the second. Both operands must be int, and the result is int.

3.3.4 Relational operators

The relational operators group left or right. Relational operators cannot be used sequentially. For example, ‘‘a<b<c’’ is not permitted by EasKey.

expression < expression;
expression > expression;
expression <= expression;

expression >= expression

The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) return boolean type. Relational operators can apply to int, char, float, string types.

3.3.5 Equality operators

EasKey do not have pointer, the equality operators only operates the contents.

3.3.5.1 *expression* == ex*pression; expression* != *expression*

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. Equality operators can be used to compare all the types defined in EasKey.

3.3.5.2 expression && expression

The && operator returns true if both its operands are true, false otherwise. The precedence of && is higher than ||, but lower than relational operators and equality operators.

3.3.5.3 expression || expression

The || operator returns true if either of its operands is true, false otherwise.

3.3.6 Assignment operators

There is one assignment operator, which is ‘‘=’’. The value of the right group is assigned to left. The assignment operator requires an identifier as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place.

3.3.6.1 identifier = expression

The value of the expression replaces that of the identifier. The operands must have the same type. The value of the expression is simply stored into the object referred to by the identifier.

3.4 Declarations

Declarations are used within function definitions to specify the interpretation of each identifier. Declarations have the form

declaration:
    decl-specifiers declarators**;**

The declaratory is the identifier being declared. Unlike C or Java or other languages, EasKey does not have storage class specifier; it only consists of type-specifier and it only consists of at most one type-specifier.

3.4.1 The type-specifiers are:

type-specifier:
    int
    char
    float
    string
    boolean
    key
    point
    color

Examples:

int a	(Uninitialized Integer identifier will be set to 0 as default value)
char b	(Uninitialized char identifier will be set to '\000' as default value)
string c	(Uninitialized string identifier will be set to "" as default value)
float d	(Uninitialized float identifier will be set to 0.0 as default value)
boolean e	(Uninitialized boolean identifier will be set to false as default value)
color f	(Uninitialized color identifier will be set to white, which represents a RGB form #0, 0, 0#, as default value)
key g	(Uninitialized key identifier will be set to $null$, as default value)
point h	(Uninitialized point identifier will be set to tuple {0, 0}, as default value)
int i=10	(This will initialize the value of h to 0)
key j=A	(This will initialize the value of j to A)

3.4.2 Declarators

The specifier in the declaration indicates the type of the objects to which the declarator refer.

Declarators have the syntax:

declarator:

identifier
declarator ( )

3.4.3 Meaning of declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type. Each declarator contains exactly one identifier; it is this identifier that is declared.

If an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

For example,

```
int a;
```

means that the identifier a, which is to be declared, has the type of int.

If a declarator has the form

D( )

then the contained identifier has the type "function returning …", where "…" is the type which the identifier would have had if the declarator had been simply D.

3.5 Constant expressions

In one case it requires expressions which evaluate to a constant: after case. In this case, the expression can involve only constants of any type except float, possibly connected by the binary operators

+ – * / %

or by the unary operators:

! - @x @y @r @g @b @k

3.6 Built-in functions

The built-in functions are classified into three categories: (1) Mouse operations; (2) Key operations; (3) Other functions.

3.6.1 Mouse operations

3.6.1.1 void __leftClickPoint(point p);

Click the left mouse button on a designated point.

3.6.1.2 void __rightClickPoint(point p);

Click the right mouse button on a designated point.

3.6.1.3 void __movePoint(point p);

Move the mouse from current point to a designated point.

3.6.1.4 void __leftDown();

Press down the left mouse button.

3.6.1.5 void __rightDown();

Press down the right mouse button.

3.6.2 Keyboard operations

3.6.2.1 void __keyStroke (key k);

Stroke a designated key on the keyboard.

3.6.3 Other functions

3.6.3.1 point __getPoint();

3.6.3.2 color __getcolor(point p);

Get the color of a designated point.

3.6.3.3 void __print(string s);

Print out designated content.

3.6.3.4 void __delay(int t);

Perform time delay of operation

3.7 Statements

Except as indicated, statements will be executed in sequence within certain Blocks, and the execution of Statements in different Block will also follow the sequence of block appearance.

3.7.1. Expression Statement

Most statements are expression statements, which have the form

```
expression ;
```

Usually expression statements are assignments or function calls.

Expression statement is a construct consisting of variables, operators, certain real value and even functions.

Examples:

```
int a=32
int a=b+32+c;
point p={2,4}+g;
```

3.7.2. Conditional Statement

There are 4 forms of conditional statement:

```
if (expression) then (statement) end if
if (expression) then (statement) else (statement) end if
if (expression) then (statement) elseif(expression) then (statement) end if
if (expression) then (statement) elseif(expression) then (statement) else (statement) end if
```

Each keyword if must end with up with a keyword end if. The elseif and the else after the keyword if are optional. With these forms, the body of a conditional statement are covered in a pair of if and end if, with no exception. Besides, each keyword end if will be paired with the last encountered if.

If the outcome of first expression is Boolean true then the first statement will be executed, else the following statement with Boolean true expression outcome will be executed.

Example:

```
1. if (a==b) then a=0 end if;
2. if (a==b) then a=0 else a=1 end if;
3. if (a==b) then a=0 elseif (a<b) then a=1 end if;
4. if (a==b) then a=0 elseif (a<b) then a=0 else a=1 end if;
5.   if (a==b) then
  if (a>c) then a=0
            end if
    end if;
```

3.7.3 While Statement

The while statement has the form:

```
while ( expression )
statement
end while
```

The substatement will be executed repeatedly so long as the value of the expression remains true.

Example:

```
 while (a>b)
          c=a;
          a=b;
          b=c;
        end while
```

The statement means a single statement or a sequence of statements, which are placed in the block between while and end while, will be executed consecutively if the condition in expression is true.

3.7.4. For Statement

The for statement has the form

```
for ( expression1; expression2; expression3)
statement
end for
```

It is equivalent to

```
expression1;
while ( expression2)
statement
expression3;
end while
```

The first expression specifies initialization for the loop. The second expression specifies a conditional judgment, which is tested before each iteration. The loop exits when the second expression is false; the third expression specifies an incrementation or decrementation which is performed after each iteration.

Example:

```
for(int i=0;i<100;i = i+1)
        a=a+i;
 end for
```

3.7.5. Switch Statement

The switch statement causes control to be transferred to one of several statements depending on the value of an expression. It has the form:

```
switch ( expression )
case constant_statement: statement
...
(default: statement)opt
end switch
```

The expression outcome can be any type other than float. The statement is typically a list of statements and each statement within the statement may follow a case prefixes as follows:

```
case constant-expression:
```

where the constant expression can be any type other than float. No two of the case constants in a switch may have the same value.

There may also be at most one statement prefix of the form

```
default :
```

When the switch statement is executed, its expression is evaluated and compared with each case constant in an undefined order. If one of the case constants is equal to the value of the expression, control is passed to the statement following the matched case prefix. If no case constant matches the expression, and if there is a default prefix, control passes to the prefixed statement. In the absence of a default prefix none of the statements in the switch is executed.

3.8 Scope

In EasKey, local variables can shadow the presence of global variables of the same name. In each block statement, such as a while loop, etc. a new local scope will be built. In that scope, all the variables outside this scope are visible except a local variable is declared with the same name. In the scope of a function declaration, the names of formals and locals mustn't be the same, since they belong to the same scope.

3.9 Examples

3.12.1. Basic function implementation

In this function, a dividend calculation function is implemented. Two doubles are passed into the function as parameters. Then the function calculates the dividend of the two doubles and returns the dividend which is a double value. If the second parameter is zero, the function will return an error message and return -1.0.

```
%calculate i divided by j
function float dividend(float i, float j)
if j==0 then __print(“error”);return -1.0;
else
return i/j;
end if;
end function
```

3.9.2. Basic array operation

In this example, an integer array is passed to the function to calculate the sum of the elements inside the array. This function takes two arguments, the first one is the array and the second one is the size of the array. At the end of the function, the sum of the array is returned.

```
function sum(int s[], int num)
int i=0;
int sum=0;
for(;i<num;i=i+1)
sum=sum+s[i];
end for
end function
```

3.9.3. Drag window to a different position

In this function, two point type parameters are passed to the function to implement the action that move a dialog window from point A to point B. This function left presses the mouse and keeps the mouse down to drag the window to another position.

```
function void dragwin(point origin, point desti)
__movePoint(origin);
__leftDown();
__movePoint(desti);
__leftUp();
end function
```

3.9.4. Enter a string into a textbox

In this function, a point textb and a string msg are passed to the function to implement the function that enters the string msg into the textbox pointed by the textb point. The function left click the region inside the textbox to focus on the textbox and then enter the string msg into the box by traversing each character in the string.

```
function void enter(point textb, string msg)
%point textb is the point within the area of the textbox
```

```
__leftClickPoint(point);
while msg.i!=eof do     //msg.i is not implemented)
__keyStroke(msg.i);     //__keyStroke is not implemented for string yet
__delay(100);
end while
end function
```

3.9.5. Color dependent mouse click

In this function, a color type variable and a point pointing to the button are passed to the function to implement the action that if the color at or is the same as the red variable then click the button, else do not press the button.

```
function void press(color red, point or, point but)
if(__getcolor(or)==red)
then
__leftClickPoint(but);
end if
end function
```

Chapter 4

Project Plan

4.1 Project Management process

Our team is formed with 5 members, and each member has their own specialty. So, we just try to make the best use of our intelligence through uneven task distribution. After we confirmed the project title, each member start to gather some useful reference and try to get used to OCaml language as soon as possible. Also, we have several conversations with TA Jiabin Hu, who has given us a lot of advice.

We held our group meeting once per week, discussing the knowledge that we learned in the class and trying to convert this knowledge into our language. During Oct and Dec, we have several long meetings and finally confirmed the LRM, which specified what we should do in our Language.

Whenever we came up with a new problem either about OCaml or our own language, we will reserve a room in butler the next day, and tried to figure it out as soon as possible.

We started to work on the test when we finally merged the source code and successfully produced our first bytecode file. Every team member tries to come up with his or her own test to find the bug in our compiler.

Finally, we really don’t have time to further discuss in the final week, and we just work on the demo and presentation after 12/15/2012.

4.2 Programming Style guide

- ✓ For every code file, the author name must be notated, as well as the modified time and the version, where may potentially has bug.
- ✓ The OCaml code should try to follow the style of microC and the good project done by former team, and the C code should be neat and clean, making it easy to read.
- ✓ Almost no IDE is used, we just try to ensure that we work on the same path and on the same environment, and that’s why the plain text editor was chosen.
- ✓ Commit the code on Dropbox and Github whenever it is modified and without bug, but have to make sure that the version and data are notated.
- ✓ Each team member has their own file directory in Dropbox and a shared repository in Github, and a public directory is used to store the bug-free code.
- ✓ Two main directory were built in the public directory, one is EasKey_Compiler, and the other one is EasKey_Intepreter, each stored the files as their name imply.

4.3 Project Timeline

Started Data	Event	Finished Data
09/10/2012	**Team Formation, some rough ideas about project**	**09/10/2012**
09/12/2012	**OCaml language learning, project title discussion**	**09/24/2012**
09/23/2012	**Project title confirmed and talked with TA**	**10/02/2012**
09/26/2012	**Learning the structure of compiler, and design our own compiler, task distributed**	**10/27/2012**
10/03/2012	**Holding several meetings, finally confirmed the grammar and submitted LRM**	**10/29/2012**
11/05/2012	**Interpreter is complete**	**11/16/2012**
11/05/2012	**Scanner is complete**	**11/26/2012**
11/05/2012	**Ast and parser are complete**	**12/01/2012**
12/06/2012	**Sast is complete**	**12/11/2012**
12/06/2012	**Analyzer and compile are complete**	**12/16/2012**
12/16/2012	**Test cases applied and demo program is complete**	**12/18/2012**
12/16/2012	**Final report is complete, and prepare for presentation**	**12/18/2012**
12/19/2012	**Presentation**	**12/19/2012**

4.4 Roles and Responsibilities

4.4.1 Keqiu Hu

Leader of this team, and he is responsible for distributing the work and design structure of each part of the project. Besides, he participated in the back-end of the project including compiler and interpreter.

4.4.2 Jinqi Huang

Mainly worked on the front-end of the project, including scanner and ast. Besides, participated in the writing of test cases.

4.4.3 Xiaoyu Huang

Worked on the record of our team Log, participated in the implementation of analyzer and sast. Also, she participated in the writing the test cases.

4.4.4 Zongheng Wang

He participated in the design of front-end, including scanner, ast, parser, sast and analyzer and bytecode.

4.4.5 Lizhong Zhang

Most of his work is focused on the debug and designing byte code, as well as the implementation of the compile. Every time when our program collapse, he will check the bytecode line by line, and tell us where goes wrong.

4.5 Software Development Environment

4.5.1 Operating System

Actually, we had to use both Windows and Linux operating system, because we need to use the winAPI, which can only be accessed via windows platform. However, most of our codes are written in Linux environment except the interpreter part, which is written in windows with visual studio.

4.5.2 Language Used

We finish the front-end, including scanner, parser, ast, sast, analyzer and compile, with OCaml language, and the back-end, including interpreter and mainly API function, is built with C language.

4.5.3 Softeware Used

4.5.3.1 Dropbox and Github

We used Dropbox as working directory and Github to backup and log codes.

4.5.3.2 Vim

Most of our group members use vim in either Linux or Windows

4.5.3.3 Visual Studio

We finished the interpreter part with visual studio in windows.

4.6 Project Log

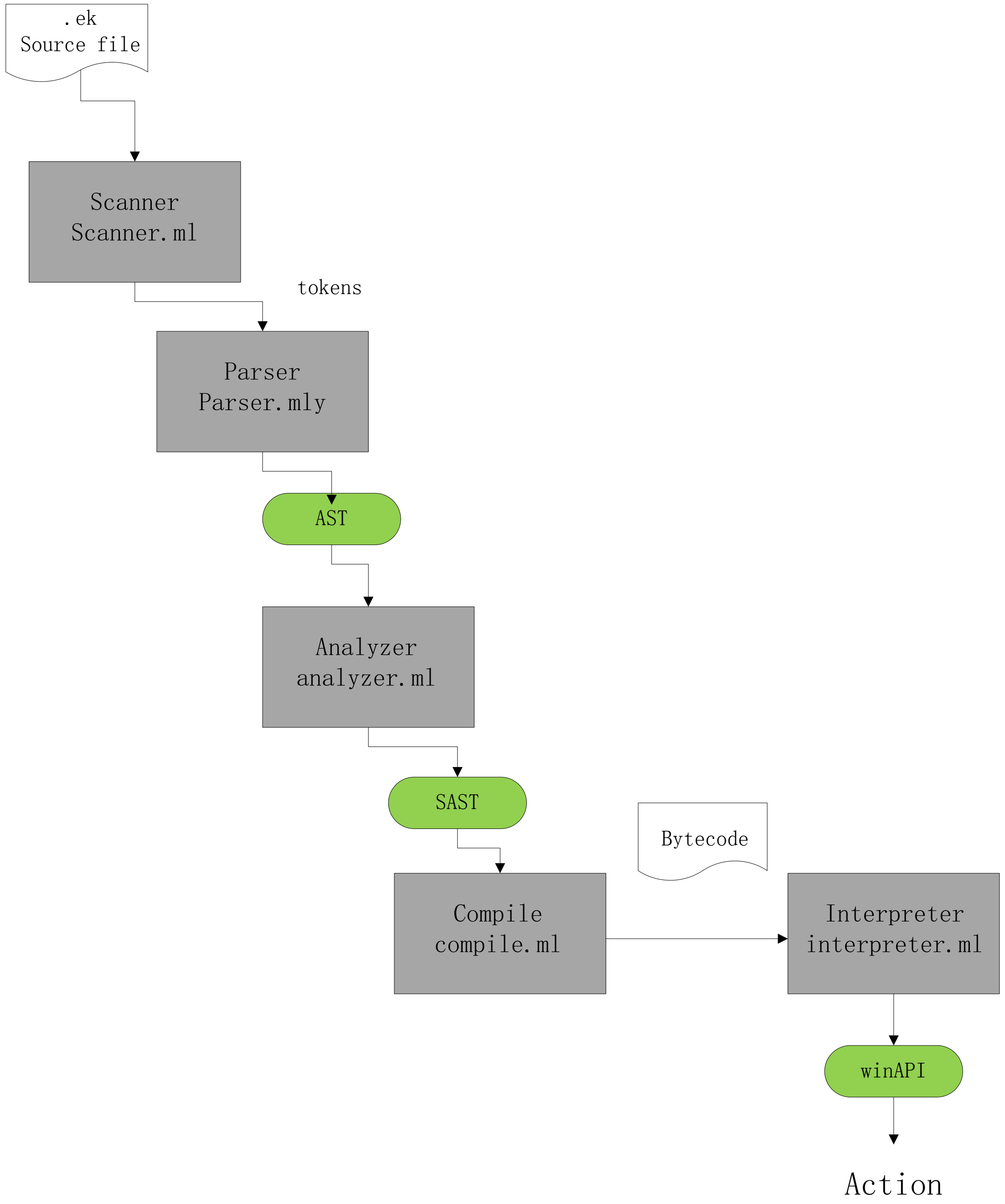
Data	Events
09/10/2012	**Team formed, 5 members get to know with each other**
09/12/2012	**Discussed about the project title and exchange opinion**
09/20/2012	**Project title confirmed, begin to evaluate the difficulty of title we choose**
09/23/2012	**Begin writing the Project Proposal, and ask TA Jiabin Hu for advice**
09/26/2012	**Submit Project Proposal, installed OCaml environment for each member**

10/02/2012	**Feedback from TA and discussed about the feasibility of our project**
10/03/2012	**1st meeting after project submitted, discussed about OCaml**
10/15/2012	**2nd meeting, discussed about the parser and scanner**
10/20/2012	**3rd meeting, discussed the basic structure of our compiler**
10/24/2012	**4th meeting, started to work on the LRM, confirmed the grammar and data types**
10/29/2012	**5th meeting, discussed and confirmed the LRM, then submitted**
11/05/2012	**6th meeting, started to work on project, and set up the coding style**
11/16/2012	**7th meeting, a team member has finished the interpreter and C API parts**
11/26/2012	**8th meeting, discussed about AST, the scanner is finished**
12/01/2012	**9th meeting, discussed about compile, Sast and analyzer, all the other parts are finished**
12/06/2012	**10th meeting, still haven't begin the work of compile, will work on it after the final**
12/11/2012	**11th meeting, group coding and finished Sast**
12/16/2012	**12th meeting, finished analyzer but still some issue with compile**
12/17/2012	**13th meeting, finished compile and started to debug and test cases**
12/18/2012	**14th meeting, finished test cases and final report, prepared for the representation**
12/19/2012	**Representation**

Chapter 5

Architecture Design

5.1 Block Diagram



5.2 Interface between Components

As we can see from the diagram above, the source file of EasKey language which has suffix (. ek), will be first processed by the scanner and produce a sequence of tokens. Parser will recognize these tokens and fill them into the AST tree, ensure the correctness of syntax. Then, analyzer will translate the AST to the SAST, and check any potential inconsistence that may result in the failure of compiler. If the semantic is correct, the compile will translate the source file into bytecode file with suffix (.e), which uses almost the same definition as microC. Finally, the interpreter will run the bytecode and call winAPI to perform Mouse or Keyboard operations.

5.3 Roles and Responsibility

Components	Implementer(s)
Scanner.mll	Jinqi Huang, Zongheng Wang
Parser.mly	Zongheng Wang, Lizhong Zhang
Ast.mli	Zongheng Wang
Analyzer.ml	Zongheng Wang, Xiaoyu Huang
Sast.mli	Xiaoyu Huang
Bytecode.ml	Jinqi Huang
Compile.ml	Keqiu Hu, Lizhong Zhang
Interpreter	Keqiu Hu
Testcases	Xiaoyu Huang, Jinqi Huang

We can roughly divide our compiler into 6 components, and distribute the tasks evenly to each team member. Scanner, bytecode and some test cases are written by Jinqi Huang. Parser, AST, Analyzer, SAST are finished by Zongheng Wang and Xiaoyu Huang. Compile is implemented by Keqiu Hu and Lizhong Zhang. Interpreter is implemented by Keqiu Hu.

Chapter 6

Test Plan

6.1 Representative Source Code

The input of EasyKey is the source code with suffix (.ek). The output of EasyKey compiler is bytecode file, which will be further translated by the interpreter written in C, and then a serial of mouse and keyboard operation will be executed.

As we have discussed about, we are not able to present the output of program here. Besides, because these operations have to call the winAPI function, which can only be invoked in windows environment, we may need to transfer the output bytecode file from Linux to windows.

Hence, we will show the bytecode we generate here. Since we use the same bytecode format as microC, we can verify whether our output is right or wrong through the comparison of bytecode between our output and microC's output.

6.2 Test Suits

Part of the test cases we chose are derived from microC test suite, and the rest are designed to test our special build in function. These tests is a mixture of both simple and complex program, which consists of arithmetic, logic, control structures, different data types, and our built-in functions.

6.2.1 Arithmetic

Test case	Operation	Output	Result

int a=(1+2);
__print(__stringOfInt(a));
	Integer addition	3	Correct

int c=1-2;
__print(__stringOfInt(c));
	Integer subtraction	-1	Correct

int b=1*2;
__print(__stringOfInt(b));
	Integer multiplication	2	Correct

int d=4/2;
__print(__stringOfInt(c));
	Integer division	2	Correct

float f=5%2;
__print(__stringOfFloat(f));
	Integer Modulus	2.5	Correct

float g=1+3*3-7;
__print(__stringOfFloat(g));
	Order of operations	3	Correct

6.2.2 Logic

Test case	Operation	Output	Result

if(true||false)
__print("true");
end if
	Test ||	true	Correct

if(true&&false)
__print("true");
else
__print("false");
end if
	Test &&	false	Correct

if(true||(false||true))
__print("true");
end if
	Test ||, && combination	true	Correct

if((true&&false)||true)
__print("true");
else
__print("false");
end if
	Test ||, && combination	true	Correct

6.2.3 if

<table>
<tr><th>Test case</th><th>Motivation</th><th>Output</th><th>Result</th></tr>
<tr><td><pre>
int i=20;
if (i==20)
        __print("if");
elseif (i<20)
        __print("elseif");
else
        __print("else");
end if
</pre></td><td>Test if and if/elseif/else</td><td>if</td><td>Correct</td></tr>
</table>

6.2.3 for

Test case	Motivation	Output	Result

int i;
for (i=0;i<10;i=i+1)
__print(__stringOfInt (i));
end for
	Test for	0
		1
		2
		3
		4
		5
		6
		7
		8
		9
			Correct

6.2.4 Derived Types

Test case	Operation	Output	Result

point p={200,400};
int a=p@x;
__print(__stringOfInt(a));
	Test type point @x operator	200	Correct

color c=#1,2,4#;
int a=c@r;
__print(__stringOfInt(a));
	Test type color @r operator	1	Correct

key k1=R;
string a=k1@k;
__print(a);
	Test type key @k operator	R	Correct

6.2.5 EasyKey Built in function

Test case	Operation	Output	Result
	This is used to test built in function __movePoint(point p), __leftClickPoint(point p)	Mouse will move to the given point and click the point	**Correct**

```
color c=#0,54,99#;
point p1={600,320};
color c1;
while(true)
        c1=__getcolor(p1);
        if(c1!=c)
       __movePoint(p1);
        __leftClickPoint(p1);
        c1=c;
        end if
end while
```

6.3 Testing Method

We use black box testing; write box testing and unit testing.

Chapter 7

Lessons Learned

7.1 Lessons learned from each member

7.1.1 Keqiu Hu

OCaml is such a language that you can never discover its elegance that until you deeply get into it. Around 1500 lines C++ are needed to implement our interpreter (because we have to call Windows API) while only 50 lines are needed for a similar interpreter in OCaml to work. For a team to finish a project peacefully and gracefully, I think at least one lady is needed to be a part of a team so that the team environment will be much softer and make in-team discussion much more rational.

. For lessons, a team must start the project earlier and divide work more wisely; so that every team member can work at the same time in parallel, instead of dividing work into a sequence of jobs that one can only work after another member finish his job.

This project does impress me with the beauty of OCaml and gain a much much deeper understanding of Programming Languages and Translators.

7.1.2 Jinqi Huang

Compiler course is always torture, but you can surely learn a lot of stuff. Scanner, parser, ast, analyzer and compile will teach you a lot not only about OCaml, but also about how a language is built and what we should think about when we program.

I can't say how much I love programming, but I can tell I have devoted lot of time to this course. Compile and Analyzer, which take most of our time, did teach me a lot. Apart from the program itself, we have learned a lot about the testing and debugging. We should keep in mind that bug is always there, but each time when you find a bug and nail it down, you will find that you have gone even further, and your program will become much more powerful.

Also, advice from me is the same as most of former students. Start early! Or you will have to stay up late every night in the final week.

Trust me, OCaml is not worse than java, just make sure you know how to write a decent, neat and powerful OCaml program.

7.1.3 Xiaoyu Huang

The most important thing I learned is to start early. You never know what you will meet next time. Even if there are only a few hours before the deadline, there may still have a lot of problems to fix. Therefore, start early, you may fix the problems early.
And designing period is a very hard and difficult time. However, a good design will help to improve the whole project quality.

Sometimes, it will save time. Since, a bad design may lead to reconfiguration the whole project. And I learned a lot about the how the scanner, parser, analyzer, compiler and interpreter work.

Last, but not the least, I learned a lot from my teammates. They are warm-hearted and passion. Zonghen Wang is a careful consideration person. Keqiu Hu is a good designer. Lizhong Zhang always fixes the problems. And Jinqi Huang helps to keep all our project done successfully.

7.1.4 Zongheng Wang

In this project, it is better to have a focus when implementing your language, rather than try to include every function, every type and every good characteristic that you have seen in other languages. The reason behind it is obvious. First of all, the time limitation is always a great constraint. Although everyone knows it is very necessary to start this project early, but the truth is that when you have time, you know very little about this project and this course; when you knows what to do, you will find there is not much time left. The second reason is that the more you try to implement a lot of general functions, perhaps the less the innovation of your programming language is. In addition, a big project will lead to a large number of debugging processes for sure. At the end, you probably mess up something and leave the professor with a bad impression, while in fact, your group might have done a lot of hard work.

7.1.5 Lizhong Zhang

The most important thing is to start doing something. Ideas keep coming out when having a group discussion. However, it is not possible, or not possible in a limited amount of time, to implement every feature generated from discussion. Some ideas which take infinite time to discuss may not be practical at all. So try to keep everything in a narrow scope at first and start to implement a small demo program. During the process, you'll find that greatest ideas come from practice.

Learn to compromise with teammates. It is even more important than the project to learn to work with others. Everyone has his own working and learning style. Most of the time there is no best ideas; it is just trade-off. So when you find yourselves in a argument, insist on yourself may sometimes not be very efficient.

The last but not the least, start early. It is not only because the whole project is huge, but also it takes a lot of time to debug when complete the first version of the project. You may be surprised to find there are so many bugs in you code. But don't worry, it will get better.

7.2 Advice for future teams

- ✓ Start early! Or you will stay up late every night in the final week.
- ✓ Try to come up with a good proposal, and don’t bother talk to professor and TA about the feasibility of your project.
- ✓ LRM should be written carefully, it will make your work easier.
- ✓ Need to pay more attention in version control system, it makes coding much better and easier.
- ✓ Holding meeting every week and telling your teammate if you run into trouble
- ✓ Compile and analyzer should be designed carefully and start early, it will take most of your time.
- ✓ Don’t worship microC as the god, it only implements some basic functions.

Reference

[1] Alfred V. Aho, Monica Lam, Ravi Sethi, and Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools. (2010). Addison-Wesley, 2006. Second Edition.
[2] Jason Hickey. Introduction to Objective Caml, Cambridge University Press.
[3] MicroC
[4] Jiabin Hu, Akash Sharma, Shuai Sun, Yan Zou, TML Final Report, 2011fall, Columbia University

Appendix

1. Makefile.

```
/Makefile:easkey: scanner.cmo parser.cmo analyzer.cmo compile.cmo bytecode.cmo easkey.ml
./Makefile:	ocamlc -o easkey scanner.cmo parser.cmo analyzer.cmo bytecode.cmo compile.cmo easkey.ml
./Makefile:#	cp easkey ..
./Makefile:
./Makefile:scanner_test: scanner.cmo parser.cmo scanner_test.ml
./Makefile:	ocamlc -o scanner_test scanner.cmo parser.cmo scanner_test.ml
./Makefile:
./Makefile:analyzer_test: scanner.cmo parser.cmo analyzer.cmo analyzer_test.ml
./Makefile:	ocamlc -o analyzer_test scanner.cmo parser.cmo analyzer.cmo analyzer_test.ml
./Makefile:
./Makefile:scanner.ml: scanner.mll
./Makefile:	ocamllex scanner.mll
./Makefile:
./Makefile:parser.ml parser.mli: parser.mly ast.cmi
./Makefile:	ocamlyacc parser.mly
./Makefile:
./Makefile:ast.cmi: type.cmi
./Makefile:sast.cmi: type.cmi
./Makefile:bytecode.cmo: type.cmi
./Makefile:
./Makefile:scanner.cmo: parser.cmi
./Makefile:parser.cmo: parser.cmi ast.cmi
./Makefile:compile.cmo: sast.cmi bytecode.cmo
./Makefile:analyzer.cmo: ast.cmi sast.cmi
./Makefile:#generator.cmo: sast.cmi bytecode.cmi
./Makefile:
./Makefile:%.cmo : %.ml
./Makefile:	ocamlc -c $<
./Makefile:
./Makefile:%.cmi : %.mli
./Makefile:	ocamlc -c $<
```

```
./Makefile:
./Makefile:.PHONY: clean
./Makefile:clean:
./Makefile:	rm -f *.cmi *.cmo scanner.ml parser.mli parser.ml easkey scanner_test analyzer_test
./Makefile:#	-rm ../easkey
./Makefile:
./Makefile:# Generated by ocamldep
./Makefile:#compiler.cmo: scanner.cmo parser.cmi
./Makefile:#compiler.cmx: scanner.cmx parser.cmx
./Makefile:#parser.cmo: parser.cmi
./Makefile:#parser.cmx: parser.cmi
./Makefile:
./Makefile:#scanner.cmx: parser.cmx
./Makefile:#scanner_test.cmo: scanner.cmo parser.cmi
./Makefile:#scanner_test.cmx: scanner.cmx parser.cmx
./Makefile:#parser.cmi:
```

2. analyzer.ml

```
./analyzer.ml:(* 11:37 Dec 16, 2012 hxy, wzh *)
./analyzer.ml:
./analyzer.ml:open Type
./analyzer.ml:open Sast
./analyzer.ml:
./analyzer.ml:type symbol_table = {
./analyzer.ml:   parent : symbol_table option;
./analyzer.ml:   variables : (t * string) list;(* variable type * variable name *)
./analyzer.ml:   in_loop : bool;               (* true if in a loop *)
./analyzer.ml:   in_switch : bool                      (* true if in a switch *)
./analyzer.ml:}
./analyzer.ml:
./analyzer.ml:type function_table = {                     (* reture type * function name * formals *)
./analyzer.ml:   functions : (t * string * t list) list;
./analyzer.ml:}
./analyzer.ml:
./analyzer.ml:let string_of_type = function
./analyzer.ml:    Void_T -> "void"
./analyzer.ml:| Int_T -> "int"
./analyzer.ml:| Char_T -> "char"
./analyzer.ml:| Float_T -> "float"
./analyzer.ml:| String_T -> "string"
./analyzer.ml:| Boolean_T -> "boolean"
./analyzer.ml:| Point_T -> "point"
```

```
        ./analyzer.ml:| Color_T -> "color"
        ./analyzer.ml:| Key_T -> "key"
        ./analyzer.ml:
        ./analyzer.ml:let string_of_unop = function
        ./analyzer.ml:    Neg -> "\'-\'"
        ./analyzer.ml:| Not -> "\'!\'"
        ./analyzer.ml:| Atx -> "\'@x\'"
        ./analyzer.ml:| Aty -> "\'@y\'"
        ./analyzer.ml:| Atr -> "\'@r\'"
        ./analyzer.ml:| Atg -> "\'@g\'"
        ./analyzer.ml:| Atb -> "\'@b\'"
        ./analyzer.ml:| Atk -> "\'@k\'"
        ./analyzer.ml:
        ./analyzer.ml:let string_of_binop = function
        ./analyzer.ml:    Add -> "\'+\'"
        ./analyzer.ml:| Sub -> "\'-\'"
        ./analyzer.ml:| Mul -> "\'*\'"
        ./analyzer.ml:| Div -> "\'/\'"
        ./analyzer.ml:| Mod -> "\'%\'"
        ./analyzer.ml:| Gt -> "\'>\'"
        ./analyzer.ml:| Lt -> "\'<\'"
        ./analyzer.ml:| Geq -> "\'>=\'"
        ./analyzer.ml:| Leq -> "\'<=\'"
        ./analyzer.ml:| Eq -> "\'==\'"
        ./analyzer.ml:| Neq -> "\'!=\'"
        ./analyzer.ml:| And -> "\'&&\'"
        ./analyzer.ml:| Or -> "\'||\'"
        ./analyzer.ml:
        ./analyzer.ml:let oprand_check_unop t unop =  (* unop : Type.unop -> unit = <fun>, check operand
type of unary operator *)
        ./analyzer.ml:   let correct =
        ./analyzer.ml:           match unop with
        ./analyzer.ml:               Neg -> (match t with Int_T | Float_T -> true | _ -> false)
        ./analyzer.ml:           | Not -> (t = Boolean_T)
        ./analyzer.ml:           | Atx | Aty -> (t = Point_T)
        ./analyzer.ml:           | Atr | Atg | Atb -> (t = Color_T)
        ./analyzer.ml:           | Atk -> (t = Key_T) in
        ./analyzer.ml:   if not correct then
        ./analyzer.ml:           raise (Failure("The oprand of " ^ string_of_unop unop ^ " can not be \'" ^
string_of_type t ^ "\'"))
        ./analyzer.ml:
```

```
./analyzer.ml:let oprand_check_binop t leftside binop =    (* binop : Type.binop -> unit = <fun>, check operand type of binary operator *)
./analyzer.ml:    let correct =
./analyzer.ml:        match binop with             (* '+' for string is to concatenate two strings *)
./analyzer.ml:            Add -> (match t with Int_T | Char_T | Float_T | Point_T | Color_T | String_T -> true | _ -> false)
./analyzer.ml:        | Sub -> (match t with Int_T | Char_T | Float_T | Point_T | Color_T -> true | _ -> false)
./analyzer.ml:        | Mul | Div -> (match t with Int_T | Float_T -> true | _ -> false)
./analyzer.ml:        | Mod -> (t = Int_T)
./analyzer.ml:        | Gt | Lt | Geq | Leq -> (match t with Int_T | Char_T | Float_T | String_T -> true | _ -> false)
./analyzer.ml:        | Eq | Neq -> (match t with
./analyzer.ml:                Int_T | Char_T | Float_T | String_T | Boolean_T | Point_T | Color_T | Key_T -> true | _ -> false)
./analyzer.ml:        | And | Or -> (t = Boolean_T) in
./analyzer.ml:    if not correct then
./analyzer.ml:        if leftside = true then
./analyzer.ml:                raise (Failure("The left oprand of " ^ string_of_binop binop ^ " can not be \"" ^ string_of_type t ^ "\""))
./analyzer.ml:        else raise (Failure("The right oprand of " ^ string_of_binop binop ^ " can not be \"" ^ string_of_type t ^ "\""))
./analyzer.ml:
./analyzer.ml:let rec find_variable scope name =
./analyzer.ml:    try
./analyzer.ml:        List.find (fun (_, s) -> s = name) scope.variables
./analyzer.ml:    with Not_found ->
./analyzer.ml:        match scope.parent with
./analyzer.ml:            Some(parent) -> find_variable parent name
./analyzer.ml:        | _ -> raise Not_found
./analyzer.ml:
./analyzer.ml:let rec expr scope funcs = function       (* AST.expr -> SAST.expr *)
./analyzer.ml:    Ast.Id(id) ->
./analyzer.ml:    (try
./analyzer.ml:        let (var_type, _) = find_variable scope id in
./analyzer.ml:                (Sast.Id(id), var_type, false)
./analyzer.ml:    with Not_found ->
./analyzer.ml:        raise (Failure ("\'" ^ id ^ "\' was not declared in this scope")))
./analyzer.ml:| Ast.Literal(lit) ->
./analyzer.ml:    let lit_type =
./analyzer.ml:        match lit with
```

```
./analyzer.ml:         Integer(i) -> Int_T
./analyzer.ml:       | Char(c) -> Char_T
./analyzer.ml:       | Float(f) -> Float_T
./analyzer.ml:       | String(str) -> String_T
./analyzer.ml:       | Boolean(b) -> Boolean_T
./analyzer.ml:       | Point(x,y) -> Point_T
./analyzer.ml:       | Color(r,g,b) -> Color_T
./analyzer.ml:       | Key(key) -> Key_T
./analyzer.ml:   in (Sast.Literal(lit), lit_type, true)
./analyzer.ml:| Ast.Assign(vname, e2) ->
./analyzer.ml:   let new_e1 = expr scope funcs (Ast.Id(vname)) and new_e2 = expr scope funcs e2 in
./analyzer.ml:       let (_, t1, _) = new_e1 and (_, t2, _) = new_e2 in
./analyzer.ml:           if t1 = t2 then
./analyzer.ml:               (Sast.Assign(vname, new_e2), t1, false)
./analyzer.ml:           else
./analyzer.ml:               raise(Failure ("type mismatch of \'" ^ string_of_type t1
./analyzer.ml:                   ^ "\' and \'" ^ string_of_type t2 ^ "\' in the assignment of \'" ^
vname ^ "\'"))
./analyzer.ml:| Ast.Binop(e1, binop, e2) ->
./analyzer.ml:   let new_e1 = expr scope funcs e1 and new_e2 = expr scope funcs e2 in
./analyzer.ml:       let (_, t1, c1) = new_e1 and (_, t2, c2) = new_e2 in
./analyzer.ml:           let _ = oprand_check_binop t1 true binop and _ = oprand_check_binop t2
false binop in (* check left & right oprands *)
./analyzer.ml:               if t1 = t2 then
./analyzer.ml:                   (Sast.Binop(new_e1, binop, new_e2), (fun t ->
./analyzer.ml:                       match binop with
./analyzer.ml:                         Gt | Lt | Geq | Leq | Eq | Neq -> Boolean_T
./analyzer.ml:                       | _ -> t) t1, c1 && c2) (* constant expr if both oprands are
constant expr *)
./analyzer.ml:               else
./analyzer.ml:                   raise (Failure("type mismatch of \'" ^ string_of_type t1
./analyzer.ml:                   ^ "\' and \'" ^ string_of_type t2 ^ "\' for the binary operator " ^
string_of_binop binop))
./analyzer.ml:| Ast.Unop(unop, e) ->
./analyzer.ml:   let new_e = expr scope funcs e in
./analyzer.ml:       let (_, t, c) = new_e in
./analyzer.ml:           let _ = oprand_check_unop t unop in
./analyzer.ml:               (Sast.Unop(unop, new_e), (fun t ->
./analyzer.ml:                       match unop with
./analyzer.ml:                         Neg | Not -> t
./analyzer.ml:                       | Atx | Aty | Atr | Atg | Atb -> Int_T
```

```
./analyzer.ml:                                  | Atk -> String_T) t, c)
./analyzer.ml:| Ast.Call(fname, actuals) ->
./analyzer.ml:   let (return_type, _, formals_types) = try
./analyzer.ml:          List.find (fun (_, s, _) -> s = fname) funcs.functions
./analyzer.ml:   with Not_found ->
./analyzer.ml:          raise(Failure("\'" ^ fname ^ "\' was not declared in this scope")) in
./analyzer.ml:   let new_actuals = List.map (expr scope funcs) actuals in
./analyzer.ml:          let actuals_types = List.map (fun (_, t, _) -> t) new_actuals in
./analyzer.ml:                 if formals_types = actuals_types then
./analyzer.ml:                        (Sast.Call(fname, new_actuals), return_type, false)
./analyzer.ml:                 else
./analyzer.ml:                        raise(Failure("type mismatch between formals and actuals of function \'" ^ fname ^ "\'"))
./analyzer.ml:| Ast.NoExpr -> (Sast.NoExpr, Void_T, false)
./analyzer.ml:
./analyzer.ml:let init_transform scope funcs = function     (* Ast.expr -> Sast.expr for initialization of variable decleration *)
./analyzer.ml:    Some(e) -> Some(expr scope funcs e)
./analyzer.ml:| None -> None
./analyzer.ml:
./analyzer.ml:let rec count_var_break scope num =
./analyzer.ml:   if scope.in_loop || scope.in_switch then
./analyzer.ml:          num
./analyzer.ml:   else
./analyzer.ml:          match scope.parent with
./analyzer.ml:              Some(parent)-> count_var_break parent (num + List.length scope.variables)
./analyzer.ml:          | None -> raise(Failure("break statement not within loop or switch"))
./analyzer.ml:
./analyzer.ml:let rec count_var_continue scope num =
./analyzer.ml:   if scope.in_loop then
./analyzer.ml:          num
./analyzer.ml:   else
./analyzer.ml:          match scope.parent with
./analyzer.ml:              Some(parent)-> count_var_continue parent (num + List.length scope.variables)
./analyzer.ml:          | None -> raise(Failure("continue statement not within a loop"))
./analyzer.ml:
./analyzer.ml:let var_decl_transform decl =      (* Ast.var_decl -> Sast.var_decl, needs init_transform to transform init *)
./analyzer.ml:   let t = fst decl in
./analyzer.ml:          List.rev (List.fold_left (fun l -> function
./analyzer.ml:              (name, Ast.NoExpr) -> (t, name, None)::l
./analyzer.ml:            | (name, e) -> (t, name, Some(e))::l) [] (snd decl))
./analyzer.ml:
./analyzer.ml:let add_var (scope, funcs, new_var_list) (t, name, e) =
./analyzer.ml:   try
./analyzer.ml:            let _ =
./analyzer.ml:                    List.find (fun (_, n) -> n = name) scope.variables
./analyzer.ml:            in raise(Failure("redeclaration of \'" ^ string_of_type t ^ " " ^ name ^ "\'"))
./analyzer.ml:   with Not_found ->
./analyzer.ml:            let new_scope = { scope with
./analyzer.ml:                    variables = (t, name)::scope.variables
./analyzer.ml:            } in
./analyzer.ml:            let new_init = init_transform scope funcs e in
./analyzer.ml:            let init_t =
./analyzer.ml:            match new_init with
./analyzer.ml:              Some(a, b, c) -> b
./analyzer.ml:            | None -> t in (* No need to check types, thus t = init_t is trivial *)
./analyzer.ml:            let new_var_decl =
./analyzer.ml:                    if t = init_t then
./analyzer.ml:                            (t, name, new_init)
./analyzer.ml:                    else
./analyzer.ml:                            raise(Failure ("type mismatch of \'" ^ string_of_type t
./analyzer.ml:                                    ^ "\' and \'" ^ string_of_type init_t ^ "\' in the assignment of \'" ^
name ^ "\'"))
./analyzer.ml:                    in (new_scope, funcs, new_var_decl::new_var_list)
./analyzer.ml:
./analyzer.ml:let get_inherit_list = function
./analyzer.ml:    [] -> []
./analyzer.ml:| l -> List.map (fun (t, s, _) -> (t, s)) l
./analyzer.ml:
./analyzer.ml:let rec stmt scope funcs inherit_list = function     (* AST.stmt -> SAST.stmt *)
./analyzer.ml:    Ast.Expr(e) -> Sast.Expr(expr scope funcs e)
./analyzer.ml:| Ast.Block(block) ->
./analyzer.ml:   let block_scope = { parent = Some(scope) ; variables = get_inherit_list inherit_list ;
in_loop = false; in_switch = false } in
./analyzer.ml:            let check_stmt (scope', st1) s =
./analyzer.ml:                    match s with
./analyzer.ml:                      Ast.VarDecl(var_decl) ->     (* variable declarations can update the scope
*)
./analyzer.ml:                            let var_list = var_decl_transform var_decl in
```

```
    ./analyzer.ml:                      let (new_scope, _, new_var_list) = List.fold_left add_var (scope',
funcs, []) var_list
    ./analyzer.ml:                          in (new_scope, (List.map (fun x -> Sast.VarDecl(x))
new_var_list) @ st1)
    ./analyzer.ml:              | _ -> (scope', (stmt scope' funcs [] s)::st1) in
    ./analyzer.ml:              let (new_block_scope, st1) = List.fold_left check_stmt (block_scope, [])
block
    ./analyzer.ml:                  in Sast.Block(List.rev st1, List.rev new_block_scope.variables)
    ./analyzer.ml:| Ast.Return(e) -> Sast.Return(expr scope funcs e)
    ./analyzer.ml:| Ast.If(exprlist, stmtlist) ->
    ./analyzer.ml:  let exprList1 = (List.rev (List.fold_left (fun l e ->
    ./analyzer.ml:      let expr1 = expr scope funcs e in
    ./analyzer.ml:          let (_, t, _) = expr1 in
    ./analyzer.ml:              if t = Boolean_T then
    ./analyzer.ml:                  expr1::l
    ./analyzer.ml:             else raise(Failure("predicates of if statments must be type \'boolean\'"))))
[] exprlist)) in
    ./analyzer.ml:      let stmtlist1 = List.rev (List.fold_left (fun l s->
    ./analyzer.ml:          let st1 = stmt scope funcs [] s in
    ./analyzer.ml:              let stmt1 =
    ./analyzer.ml:                  match st1 with
    ./analyzer.ml:                    Sast.Block(_, _) -> st1
    ./analyzer.ml:                  | _ -> raise(Failure("unexpected error"))
    ./analyzer.ml:              in stmt1::l) [] stmtlist)
    ./analyzer.ml:                  in Sast.If(exprList1, stmtlist1)
    ./analyzer.ml:| Ast.For(expr1, expr2, expr3, stmt1) ->
    ./analyzer.ml:  let e1 = expr scope funcs expr1 in
    ./analyzer.ml:      let e2 = expr scope funcs expr2 in
    ./analyzer.ml:          let (_, t2, _) = e2 in
    ./analyzer.ml:              if t2 = Boolean_T then
    ./analyzer.ml:                  let for_scope = { scope with in_loop = true }    in
    ./analyzer.ml:                  let block_for =
    ./analyzer.ml:                      match stmt1 with
    ./analyzer.ml:                        Ast.Block(_) -> stmt1
    ./analyzer.ml:                      | _ -> raise(Failure("unexpected error")) in
    ./analyzer.ml:                  let e3 = expr scope funcs expr3 in
    ./analyzer.ml:                      let new_stmt = stmt for_scope funcs [] block_for in
    ./analyzer.ml:                          Sast.For (e1, e2, e3, new_stmt)
    ./analyzer.ml:              else
    ./analyzer.ml:                  raise(Failure("the second expression in for statements must be
type \'boolean\'"))
```

```
./analyzer.ml:| Ast.ForWithDecl(var_decl, expr1, expr2, stmt1) ->
./analyzer.ml:   let var_list = var_decl_transform var_decl in
./analyzer.ml:        let (new_scope, _, new_var_list) = List.fold_left add_var (scope, funcs, []) var_list in
./analyzer.ml:            let e1 = expr new_scope funcs expr1 in
./analyzer.ml:                let (_, t1, _) = e1 in
./analyzer.ml:                    if t1 = Boolean_T then
./analyzer.ml:                        let for_scope = { scope with in_loop = true } in (* original scope as the parent *)
./analyzer.ml:                        let block_for =
./analyzer.ml:                            match stmt1 with
./analyzer.ml:                              Ast. Block(_) -> stmt1
./analyzer.ml:                            | _ -> raise(Failure("unexpected error")) in
./analyzer.ml:                        let e2 = expr new_scope funcs expr2 in
./analyzer.ml:                        let new_stmt = stmt for_scope funcs new_var_list block_for in
./analyzer.ml:                            Sast.ForWithDecl(List.rev new_var_list, e1, e2, new_stmt)
./analyzer.ml:                    else
./analyzer.ml:                        raise(Failure("the second expression in for statements must be type \'boolean\'"))
./analyzer.ml:| Ast.While(expr1, stmt1) ->
./analyzer.ml:   let new_expr = expr scope funcs expr1 in
./analyzer.ml:        let (_, t1, _) = new_expr in
./analyzer.ml:            if t1 = Boolean_T then
./analyzer.ml:                let loop_scope = { scope with in_loop = true }
./analyzer.ml:                    in let block_while =
./analyzer.ml:                    match stmt1 with
./analyzer.ml:                      Ast.Block(_) -> stmt1
./analyzer.ml:                    | _ -> raise(Failure("unexpected error"))
./analyzer.ml:                in let new_stmt = stmt loop_scope funcs [] block_while
./analyzer.ml:                    in Sast.While (new_expr, new_stmt)
./analyzer.ml:            else
./analyzer.ml:                raise (Failure("predicates of while statments must be type \'boolean\'"))
./analyzer.ml:| Ast.Switch(expr1, exprlist, stmtlist) ->
./analyzer.ml:   let e' = expr scope funcs expr1 in
./analyzer.ml:        let (_, t', _ ) = e' in
./analyzer.ml:            if t' != Float_T then
./analyzer.ml:                let exprList1 = (List.rev (List.fold_left (fun l e ->
./analyzer.ml:                let e1 = expr scope funcs e in
./analyzer.ml:                let (_, t1, c1) = e1 in
./analyzer.ml:                          if c1 = false then
./analyzer.ml:                              raise(Failure("case label must be a constant-expression"))
./analyzer.ml:                          else if t1 != t' then
./analyzer.ml:                              raise(Failure("case label must have the same type as switch quantity"))
./analyzer.ml:                          else
./analyzer.ml:                              e1::l) [] exprlist)) in
./analyzer.ml:                          let new_scope = { scope with in_switch = true } in
./analyzer.ml:                          let stmtlist1 = List.map (fun case_stmt ->
./analyzer.ml:                              let case_block =
./analyzer.ml:                                  match case_stmt with
./analyzer.ml:                                    Ast.Block(_) -> case_stmt
./analyzer.ml:                                  | _ -> raise(Failure("unexpected error"))
./analyzer.ml:                              in stmt new_scope funcs [] case_block
./analyzer.ml:                          ) stmtlist
./analyzer.ml:                              in Sast.Switch(e', exprList1, stmtlist1)
./analyzer.ml:                      else raise(Failure("switch quantity can not be a float"))
./analyzer.ml:| Ast.Continue -> Sast.Continue(count_var_continue scope 0)
./analyzer.ml:| Ast.Break -> Sast.Break(count_var_break scope 0)
./analyzer.ml:| Ast.VarDecl(_) -> raise(Failure("unexpected error"))        (* Ast.VarDecl is handled in Ast.Block *)
./analyzer.ml:
./analyzer.ml:let rec check_return fdecl = function          (* check whether return statement exists *)
./analyzer.ml:    Sast.Expr(e) -> false
./analyzer.ml:| Sast.Block(s_list, _) -> List.fold_left (fun a b -> a || b) false (List.map (check_return fdecl) s_list)
./analyzer.ml:| Sast.Return(e) -> let (_, t, _) = e in
./analyzer.ml:    if t = fdecl.Ast.rtype then true else raise(Failure ("return expression has the type \"" ^ string_of_type t ^
./analyzer.ml:                                  "\' but a return expression was expected of type \"" ^ string_of_type fdecl.Ast.rtype ^
./analyzer.ml:                                  "\' in function \"" ^ string_of_type fdecl.Ast.rtype ^ " " ^
./analyzer.ml:                                  fdecl.Ast.fname ^ "(" ^ ((fun l -> String.concat ", "
./analyzer.ml:                                 (List.map string_of_type (List.map fst l))) fdecl.Ast.formals) ^ ")\'"))
./analyzer.ml:| Sast.If(e_list, s_list) -> List.fold_left (fun a b -> a || b) false (List.map (check_return fdecl) s_list)
./analyzer.ml:| Sast.For(e1, e2, e3, s) -> check_return fdecl s
./analyzer.ml:| Sast.ForWithDecl(vdecl_list, e1, e2, s) -> check_return fdecl s
./analyzer.ml:| Sast.While(e, s) -> check_return fdecl s
```

```
./analyzer.ml:| Sast.Switch(e, e_list, s_list) -> List.fold_left (fun a b -> a || b) false (List.map (check_return fdecl) s_list)
./analyzer.ml:| Sast.Continue(i) -> false
./analyzer.ml:| Sast.Break(i) -> false
./analyzer.ml:| Sast.VarDecl(vdecl) -> false
./analyzer.ml:
./analyzer.ml:let semantically_check program =          (* AST.program -> SAST.program *)
./analyzer.ml:   let init_funcs = { functions = [
./analyzer.ml:          (Void_T, "__leftClickPoint", [Point_T]);
./analyzer.ml:          (Void_T, "__rightClickPoint", [Point_T]);
./analyzer.ml:          (Void_T, "__doubleClick", [Point_T]);
./analyzer.ml:          (Void_T, "__movePoint", [Point_T]);
./analyzer.ml:          (Void_T, "__leftDown", []);
./analyzer.ml:          (Void_T, "__rightDown", []);
./analyzer.ml:          (Void_T, "__leftUp", []);
./analyzer.ml:          (Void_T, "__rightUp", []);
./analyzer.ml:          (Void_T, "__keyDown", [Key_T]);
./analyzer.ml:          (Void_T, "__keyUp", [Key_T]);
./analyzer.ml:          (Void_T, "__keyStroke", [Key_T]);
./analyzer.ml:          (Void_T, "__comboStroke", [Key_T; Key_T; Key_T; Key_T; Key_T; Key_T; Key_T]);
./analyzer.ml:          (Point_T, "__getPoint", []);
./analyzer.ml:          (Color_T, "__getcolor", [Point_T]);
./analyzer.ml:          (Void_T, "__print", [String_T]);
./analyzer.ml:          (String_T, "__stringOfInt", [Int_T]);
./analyzer.ml:          (String_T, "__stringOfFloat", [Float_T]);
./analyzer.ml:          (String_T, "__stringOfChar", [Char_T]);
./analyzer.ml:          (String_T, "__stringOfBoolean", [Boolean_T]);
./analyzer.ml:          (String_T, "__stringOfPoint", [Point_T]);
./analyzer.ml:          (String_T, "__stringOfKey", [Key_T]);
./analyzer.ml:          (String_T, "__stringOfColor", [Color_T]);
./analyzer.ml:          (Int_T, "__getASCII", [Char_T]);
./analyzer.ml:          (Char_T, "__getChar", [Int_T]);
./analyzer.ml:          (Int_T, "__getStringLength", [String_T]);
./analyzer.ml:          (Char_T, "__getCharOfString", [String_T; Int_T]);
./analyzer.ml:          (String_T, "__setCharOfString", [String_T; Int_T; Char_T]);
./analyzer.ml:          (String_T, "__subOfString", [Int_T; Int_T]);
./analyzer.ml:          (Void_T, "__delay", [Int_T]);
./analyzer.ml:          ] } in
./analyzer.ml:   let init_scope = {
./analyzer.ml:          parent = None;
./analyzer.ml:          variables = [];
./analyzer.ml:          in_loop = false;
./analyzer.ml:          in_switch = false } in
./analyzer.ml:      let (new_var_decl, new_func_decl) =
./analyzer.ml:          let rec check var_decl func_decl scope funcs = function
./analyzer.ml:            [] -> (var_decl, List.rev func_decl)
./analyzer.ml:          | Ast.Var_decl(vdecl)::tl ->
./analyzer.ml:              let new_vdecl = var_decl_transform vdecl in
./analyzer.ml:                  let sast_vdecl = List.map (fun (t, name, init) ->
./analyzer.ml:                  try
./analyzer.ml:                      let _ = List.find (fun (_, s) -> s = name) scope.variables
./analyzer.ml:                          in raise(Failure("redeclaration of \"" ^ string_of_type t ^ " "
^ name ^ "\""))
./analyzer.ml:                  with Not_found ->
./analyzer.ml:                      let new_init = init_transform scope funcs init in
./analyzer.ml:                      let init_t =
./analyzer.ml:                      match new_init with
./analyzer.ml:                        Some(a, b, c) -> b
./analyzer.ml:                      | None -> t in (* No need to check types, thus t = init_t is trivial
*)
./analyzer.ml:                          if t = init_t then
./analyzer.ml:                              (t, name, new_init)
./analyzer.ml:                          else
./analyzer.ml:                              raise(Failure ("type mismatch of \"" ^ string_of_type t
./analyzer.ml:                                  ^ "\' and \"" ^ string_of_type init_t ^ "\' in the
assignment of \"" ^ name ^ "\""))
./analyzer.ml:                      ) new_vdecl in
./analyzer.ml:                  let updated_scope = { scope with
./analyzer.ml:                      variables = scope.variables @ List.map (fun (t, name, _) -> (t,
name)) new_vdecl
./analyzer.ml:                  }
./analyzer.ml:                  in check (var_decl @ sast_vdecl) func_decl updated_scope funcs tl
./analyzer.ml:          | Ast.Func_decl(fdecl)::tl ->
./analyzer.ml:              try
./analyzer.ml:                  let _ = List.find (fun (_, s, _) -> s = fdecl.Ast.fname) funcs.functions
./analyzer.ml:                      in raise(Failure ("redeclaration of function \"" ^ string_of_type
fdecl.Ast.rtype ^ " " ^
./analyzer.ml:                          fdecl.Ast.fname ^ "(" ^ ((fun l -> String.concat ", "
(List.map string_of_type
./analyzer.ml:                          (List.map fst l))) fdecl.Ast.formals) ^ ")\""))
./analyzer.ml:              with Not_found ->
```

```
    ./analyzer.ml:          let rec formals_name_test = function
    ./analyzer.ml:            [] -> (false, "")
    ./analyzer.ml:          | hd::tl -> let name = snd hd in
    ./analyzer.ml:              if List.exists (fun (_, s) -> s = name) tl then
    ./analyzer.ml:                  (true, name)
    ./analyzer.ml:              else formals_name_test tl in
    ./analyzer.ml:          let (same_name, name) = formals_name_test fdecl.Ast.formals in
    ./analyzer.ml:              if same_name then
    ./analyzer.ml:                  raise(Failure("\'" ^ name ^ "\' has a previous declaration"))
    ./analyzer.ml:              else
    ./analyzer.ml:          let formals_types = List.map fst fdecl.Ast.formals in
    ./analyzer.ml:              let updated_funcs = {
    ./analyzer.ml:                  functions = (fdecl.Ast.rtype, fdecl.Ast.fname,
formals_types)::funcs.functions
    ./analyzer.ml:              } in
    ./analyzer.ml:              let func_scope = { scope with
    ./analyzer.ml:                  parent = Some(scope);
    ./analyzer.ml:                  variables = fdecl.Ast.formals
    ./analyzer.ml:              } in
    ./analyzer.ml:              let sast_block = stmt func_scope updated_funcs []
(Ast.Block(fdecl.Ast.body)) in
    ./analyzer.ml:              let valid_return = check_return fdecl sast_block in
    ./analyzer.ml:                  if(not valid_return) then
    ./analyzer.ml:                      raise(Failure ("no return statement in function \'" ^
string_of_type fdecl.Ast.rtype ^
    ./analyzer.ml:                      " " ^ fdecl.Ast.fname ^ "(" ^ ((fun l -> String.concat ",
" (List.map string_of_type                              (List.map fst l))) fdecl.Ast.formals) ^
")\'"))
    ./analyzer.ml:                  else
    ./analyzer.ml:              let (sast_body, sast_locals) = match sast_block with
    ./analyzer.ml:                Sast.Block(stmts_list, locals_list) ->
    ./analyzer.ml:                  List.iter (fun (_, name) ->
    ./analyzer.ml:                      try
    ./analyzer.ml:                          let _ = List.find (fun (_, s) -> s = name)
locals_list
    ./analyzer.ml:                      in raise(Failure("declaration of \'" ^ name ^ "\' shadows
a parameter"))
    ./analyzer.ml:                      with Not_found -> ()) fdecl.Ast.formals; (stmts_list,
locals_list)
    ./analyzer.ml:              | _ -> raise(Failure("unexpected error")) in
    ./analyzer.ml:              let sast_fdecl = {
```

```
./analyzer.ml:                          rtype = fdecl.Ast.rtype;
./analyzer.ml:                          fname = fdecl.Ast.fname;
./analyzer.ml:                          formals = fdecl.Ast.formals;
./analyzer.ml:                          locals = sast_locals;
./analyzer.ml:                          body = sast_body
./analyzer.ml:                      } in check var_decl (sast_fdecl::func_decl) scope updated_funcs tl
./analyzer.ml:          in check [] [] init_scope init_funcs (List.rev program)
./analyzer.ml:   in (new_var_decl, new_func_decl)
```

3. ast.mli

```
./ast.mli:(* 10:45 Dec 16, 2012 wzh *)
./ast.mli:
./ast.mli:open Type
./ast.mli:
./ast.mli:type expr =                               (* Expressions *)
./ast.mli:    Id of string                          (* foo *)
./ast.mli:| Literal of constant                         (* 12 $PgDn$ '@' *)
./ast.mli:| Assign of string * expr                     (* foo = 823 *)
./ast.mli:| Binop of expr * binop * expr                    (* x=a+b *)
./ast.mli:| Unop of unop * expr                         (* !a -t p@x *)
./ast.mli:| Call of string * expr list                  (* foo(a,23) *)
./ast.mli:| NoExpr                                  (* for(;;) *)
./ast.mli:(*
./ast.mli:type init =                           (* Variable initialization *)
./ast.mli:    WithInit of string * expr                 (* int a = 3; *)
./ast.mli:| WithoutInit of string                   (* float b; *)
./ast.mli:*)
./ast.mli:type var_decl =                           (* Variable declaration *)
./ast.mli:    t * (string * expr) list                  (* int a, b = 3, c; *)
./ast.mli:
./ast.mli:type stmt =                           (* Statements *)
./ast.mli:    Expr of expr                              (* foo = bar + 3; *)
./ast.mli:| Block of stmt list                          (* while(i<10) ... end while *)
./ast.mli:| Return of expr                          (* return 42; *)
./ast.mli:| If of expr list * stmt list                 (* if(foo==42) ... elseif(foo<42) ... else ... end if *)
./ast.mli:| For of expr * expr * expr * stmt            (* for(i=0;i<10;i=i+1) ... end for *)
./ast.mli:| ForWithDecl of var_decl * expr * expr * stmt   (* for(int i=0;i<10;i=i+1) ... end for *)
./ast.mli:| While of expr * stmt                        (* while(i<10) ... end while *)
./ast.mli:| Switch of expr * expr list * stmt list      (* switch(x+y) case 1: a=8; b=23; case 2: c=8; d=23; end switch *)
```

```
./ast.mli:| Continue                              (* continue; *)
./ast.mli:| Break                                 (* break; *)
./ast.mli:| VarDecl of var_decl                       (* int a, b = 3, c; *)
./ast.mli:
./ast.mli:type func_decl = {                      (* Function declaration *)
./ast.mli:   rtype : t;                       (* return type of the function *)
./ast.mli:   fname : string;                  (* name of the function *)
./ast.mli:   formals : (t * string) list;           (* types and names of formal arguments *)
./ast.mli:   body : stmt list;                (* body of the function *)
./ast.mli:}
./ast.mli:
./ast.mli:type decls =
./ast.mli:    Var_decl of var_decl
./ast.mli:| Func_decl of func_decl
./ast.mli:
./ast.mli:type program = decls list                   (* global vars, funcs *)
```

4. bytecode.ml

```
./bytecode.ml:(* Nov 29, @Jinqi*)
./bytecode.ml:
./bytecode.ml:open Type
./bytecode.ml:
./bytecode.ml:type bstmt =
./bytecode.ml:    Glb of int                  (* Indicate the number of gloabal variables *)
./bytecode.ml:| Pop of int              (* Pop out several elements from the stack *)
./bytecode.ml:| Bin of Type.binop           (* Perform arithmetic on the top two elements of stack *)
./bytecode.ml:| Uni of Type.unop            (* Perform unary operation on the top of stack *)
./bytecode.ml:| Psi of int              (* Push an integer *)
./bytecode.ml:| Psp of int * int        (* Push a pointer *)
./bytecode.ml:| Psco of int * int * int(* push a color *)
./bytecode.ml:| Psk of string           (* push a key *)
./bytecode.ml:| Psf of float            (* Push a floating number *)
./bytecode.ml:| Psb of bool                 (* Push a boolean number *)
./bytecode.ml:| Psc of char                 (* Push a char *)
./bytecode.ml:| Pss of string           (* Push a string *)
./bytecode.ml:| Lod of int              (* Fetch global variable *)
./bytecode.ml:| Str of int              (* Store global variable *)
./bytecode.ml:| Lfp of int              (* Load frame pointer relative *)
./bytecode.ml:| Sfp of int              (* Store frame pointer relative *)
./bytecode.ml:| Jsr of int              (* Call function by absolute address *)
./bytecode.ml:| Ent of int              (* Push FP, FP -> SP, SP += i *)
./bytecode.ml:| Rts of int              (* Restore FP, SP, consume formals, push result *)
```

```
./bytecode.ml:| Beq of int          (* Branch relative if top-of-stack is zero *)
./bytecode.ml:| Bne of int          (* Branch relative if top-of-stack is non-zero *)
./bytecode.ml:| Bra of int          (* Branch relative *)
./bytecode.ml:| Hlt                 (* Terminate *)
./bytecode.ml:
./bytecode.ml:type prog = {
./bytecode.ml:      num_globals : int;         (* Number of global variables *)
./bytecode.ml:      text : bstmt array;     (* Code for all the functions *)
./bytecode.ml:}
./bytecode.ml:
./bytecode.ml:let string_of_stmt = function
./bytecode.ml:    Glb(i) -> "glb " ^ string_of_int i
./bytecode.ml:| Psi(i) -> "psi " ^ string_of_int i
./bytecode.ml:| Psp(i,j) -> "psp " ^ string_of_int i ^ " " ^ string_of_int j
./bytecode.ml:| Psco(i,j,k) -> "psco " ^ string_of_int i ^ " " ^ string_of_int j ^ " " ^ string_of_int k
./bytecode.ml:| Psf(i) -> "psf " ^ string_of_float i
./bytecode.ml:| Psb(i) -> if i then "psb " ^ "1" else "psb " ^ "0"
./bytecode.ml:| Psc(i) -> "psc " ^ Char.escaped i
./bytecode.ml:| Pss(i) -> "pss " ^ i ^ "\\0"
./bytecode.ml:| Psk(k) -> "psk " ^ k
./bytecode.ml:| Pop(i)-> "pop " ^ string_of_int i
./bytecode.ml:| Bin(Type.Add) -> "add"
./bytecode.ml:| Bin(Type.Sub) -> "sub"
./bytecode.ml:| Bin(Type.Mul) -> "mul"
./bytecode.ml:| Bin(Type.Div) -> "div"
./bytecode.ml:| Bin(Type.Mod) -> "mod"
./bytecode.ml:| Bin(Type.And) -> "and"
./bytecode.ml:| Bin(Type.Or)    -> "or"
./bytecode.ml:| Bin(Type.Eq) -> "eql"
./bytecode.ml:| Bin(Type.Neq) -> "neq"
./bytecode.ml:| Bin(Type.Lt) -> "lt"
./bytecode.ml:| Bin(Type.Leq) -> "leq"
./bytecode.ml:| Bin(Type.Gt) -> "gt"
./bytecode.ml:| Bin(Type.Geq) -> "geq"
./bytecode.ml:| Uni(Type.Neg) -> "neg"
./bytecode.ml:| Uni(Type.Not) -> "not"
./bytecode.ml:| Uni(Type.Atx) -> "atx"
./bytecode.ml:| Uni(Type.Aty) -> "aty"
./bytecode.ml:| Uni(Type.Atr) -> "atr"
./bytecode.ml:| Uni(Type.Atg) -> "atg"
./bytecode.ml:| Uni(Type.Atb) -> "atb"
```

```
./bytecode.ml:| Uni(Type.Atk) -> "atk"
./bytecode.ml:| Lod(i) -> "lod " ^ string_of_int i
./bytecode.ml:| Str(i) -> "str " ^ string_of_int i
./bytecode.ml:| Lfp(i) -> "lfp " ^ string_of_int i
./bytecode.ml:| Sfp(i) -> "sfp " ^ string_of_int i
./bytecode.ml:| Jsr(i) -> "jsr " ^ string_of_int i
./bytecode.ml:| Ent(i) -> "ent " ^ string_of_int i
./bytecode.ml:| Rts(i) -> "rts " ^ string_of_int i
./bytecode.ml:| Bne(i) -> "bne " ^ string_of_int i
./bytecode.ml:| Beq(i) -> "beq " ^ string_of_int i
./bytecode.ml:| Bra(i) -> "bra " ^ string_of_int i
./bytecode.ml:| Hlt     -> "hlt"
./bytecode.ml:
./bytecode.ml:let string_of_prog p =
./bytecode.ml:  "" ^ let funca = Array.mapi (fun i s -> "" ^ string_of_stmt s) p.text
./bytecode.ml:        in String.concat "\n" (Array.to_list funca)
```

5. compile.ml

```
./compile.ml:open Bytecode
./compile.ml:open Type
./compile.ml:open Sast
./compile.ml:module StringMap =Map.Make(String)
./compile.ml:(*test region*)
./compile.ml:(*@Keqiu Hu & Lizhong Zhang*)
./compile.ml:(*Referece: Microc--Stephen Edwards & TML--Yan Zou *)
./compile.ml:(*Symbol table as MICROC*)
./compile.ml:type env={
./compile.ml:     function_index:int StringMap.t;
./compile.ml:     global_index:int StringMap.t;
./compile.ml:    (* local_index:int StringMap.t;*)
./compile.ml:}
./compile.ml:
./compile.ml:(*function to merge var_decl into a Block t*string list*)
./compile.ml:(*let merge (t,str,_)=function*)
./compile.ml:
./compile.ml:
./compile.ml:(*add names in list to the map (referto zou)*)
./compile.ml:let add_to_map list map start_index =
./compile.ml:     snd (List.fold_left (fun (i, map) name ->
./compile.ml:     (*print_endline (name ^ ": " ^ (string_of_int i)); (* debug *) *) (i + 1,
StringMap.add name i map)) (start_index, map) list)
./compile.ml:
```

```
./compile.ml:(*function to perform preglobal expression handling*)
./compile.ml:let add_to_map list map start_index =
./compile.ml:snd (List.fold_left (fun (i, map) name ->
./compile.ml:(*print_endline (name ^ ": " ^ (string_of_int i)); (* debug *) *) (i + 1, StringMap.add name i map)) (start_index, map) list)
./compile.ml:(**)
./compile.ml:let exprId =function
./compile.ml:	(Id(a),b,c)->a
./compile.ml:	|_->""
./compile.ml:let takestr (a,b,c)=a
./compile.ml:(*not needed anymore...
./compile.ml:let expr2 env =function
./compile.ml:	Assign(s,e)->[Str (StringMap.find s env.global_index)]*)
./compile.ml:let rec enum stride n=function
./compile.ml:	[]->[]
./compile.ml:	|hd::tl->(n,hd)::enum stride (n+stride) tl
./compile.ml:
./compile.ml:(*add pairs into a map as MICORC*)
./compile.ml:let string_map_pairs map pairs=
./compile.ml:	List.fold_left (fun m (i,n) ->StringMap.add n i m) map pairs
./compile.ml:
./compile.ml:(*main translate function, modified from MICROC*)
./compile.ml:(*let p=function (a,b)->b
./compile.ml:
./compile.ml:let res=function
./compile.ml:	WithInit(a,b)->a
./compile.ml:	|WithoutInit(c)->c
./compile.ml:
./compile.ml:let initos=function
./compile.ml:	(t,v)->List.map res v*)
./compile.ml:(*take the t*string list out from a BLOCK*)
./compile.ml:let takelistout=function
./compile.ml:			Block(a,e)->e
./compile.ml:			|d->[Int_T,"noinit"]
./compile.ml:(*turn variable list into t*string list*)
./compile.ml:let var_list t=
./compile.ml:	List.map (fun(a,b,c)->(a,b)) t
./compile.ml:
./compile.ml:let init_value t=function
./compile.ml:	    | Some(e) -> e
./compile.ml:  | None ->
```

```
./compile.ml:                   let lit = match t with
./compile.ml:
./compile.ml:                                           Void_T->Integer(0)
./compile.ml:                                           | Point_T->Point(0,0)
./compile.ml:                                           | Int_T -> Integer(0)
./compile.ml:                                           |Key_T->String("Null")
./compile.ml:                                           |Color_T->Color(0,0,0)
./compile.ml:                           | Float_T -> Float(0.0)
./compile.ml:                           | Char_T -> Char('\000')
./compile.ml:                           | String_T -> String("\0")
./compile.ml:                           | Boolean_T -> Boolean(false)
./compile.ml:
./compile.ml:                   in Literal(lit), t,true
./compile.ml:
./compile.ml:(*translate the program passed in*)
./compile.ml:
./compile.ml:let translate (globals,functions)=
./compile.ml:     (* let glbexpr=(List.map (fun (a,b,c)->c) globals) in*)
./compile.ml:       let precute=List.map (fun (a,b,c)->b) globals in
./compile.ml:    (* Allocate "addresses" for each global variable *)
./compile.ml:       let global_indexes = string_map_pairs StringMap.empty (enum 1 0 precute) in
./compile.ml:
./compile.ml:    (* Assign indexes to function names; built-in "print" is special *)
./compile.ml:(*    let built_in_functions = StringMap.add "print" (-1) StringMap.empty in*)
./compile.ml:       (*buildin function manipulation*)
./compile.ml:       let built_in_functions =
./compile.ml:              let builtin =
./compile.ml:
["__leftClickPoint";"__rightClickPoint";"__doubleClick";"__movePoint";"__leftDown";"__rightDown";"_
_leftUp";"__rightUp";"__keyDown";"__keyUp";"__keyStroke";"__comboStroke";"__getPoint";"__getcolo
r";"__print";"__stringOfInt";"__stringOfFloat";"__stringOfChar";"__stringOfBoolean";"__stringOfPoint";"
__stringOfKey";"__stringOfColor";"__getASCII";"__getChar";"__getStringLength";"__getCharOfString";
"__setCharOfString";"__subOfString";"__delay";"__compareColor"] in string_map_pairs
StringMap.empty (enum (-1) (-1) builtin) in
./compile.ml:
./compile.ml:    let function_indexes = string_map_pairs built_in_functions
./compile.ml:           (enum 1 0 (List.map (fun f -> f.fname) functions)) in
./compile.ml:
./compile.ml:    (* Translate a function in AST form into a list of bytecode statements *)
./compile.ml:    let rec block local_index next_index num_params sl=
./compile.ml:              let rec expr (exprdetail,t,int)=match exprdetail with
```

```
./compile.ml:               Literal i->begin match i with
./compile.ml:                   Integer(j)->[Psi j]
./compile.ml:                   |Char(c)->[Psc c]
./compile.ml:                   |String(s)->[Pss s]
./compile.ml:                   |Float(f)->[Psf f]
./compile.ml:                   |Boolean(b)->[Psb b]
./compile.ml:                   |Point(x,y)->[Psp(x,y)]
./compile.ml:                   |Key(k)->[Pss k]
./compile.ml:                   |Color(r,g,l)->[Psco (r,g,l)](*later chage*)
./compile.ml:                   end
./compile.ml:               |Id(s)->
./compile.ml:                   (try [Lfp (StringMap.find s local_index)]
./compile.ml:                   with Not_found -> try [Lod (StringMap.find s global_indexes)]
./compile.ml:                   with Not_found -> raise (Failure ("undeclared variable"^s)))
./compile.ml:
./compile.ml:               |Binop(e1,op,e2) -> expr e1 @ expr e2 @[Bin op]
./compile.ml:
./compile.ml:         |Unop (op, e) -> (expr e) @ [Uni op]
./compile.ml:
./compile.ml:               |Assign (s,e)->expr e @
./compile.ml:                   (try [Sfp (StringMap.find s local_index)]
./compile.ml:                   with Not_found->try [Str (StringMap.find s
./compile.ml:                   global_indexes)]
./compile.ml:                   with Not_found->raise (Failure("Undeclared variable"^(s))))
./compile.ml:               |Call(fname,actuals)->
./compile.ml:                       if(fname="__leftClickPoint") then
./compile.ml:                           if(List.length actuals>0) then
./compile.ml:                               (List.concat (List.map (fun e->
./compile.ml:                                   (expr e) @ [Jsr (-1);Pop 1]) actuals))@
./compile.ml:                               [Psi 0]
./compile.ml:                           else []
./compile.ml:                       else if(fname="__rightClickPoint") then
./compile.ml:                          List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
(-2);Pop 1]@[Psi 0]
./compile.ml:
./compile.ml:                       else if(fname="__doubleClick") then
./compile.ml:                            List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                            (-3);Pop 1]@[Psi 0]
./compile.ml:                       else if(fname="__movePoint") then
./compile.ml:                           List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                           (-4)]@[Psi 0]
```

```
./compile.ml:                    else if(fname="__leftDown") then
./compile.ml:                        List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                        (-5)]@[Psi 0]
./compile.ml:                    else if(fname="__rightDown") then
./compile.ml:                        List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                        (-6)]@[Psi 0]
./compile.ml:                    else if(fname="__leftUp") then
./compile.ml:                        List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                        (-7);Pop 1;Pop 1]@[Psi 0]
./compile.ml:                    else if(fname="__rightUp") then
./compile.ml:                        List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                        (-8);Pop 1;Pop 1]@[Psi 0]
./compile.ml:                    else if(fname="__keyDown") then
./compile.ml:                       List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                       (-9);Pop 1;Pop 1]@[Psi 0]
./compile.ml:                    else if(fname="__keyUp") then
./compile.ml:                         List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                         (-10);Pop 1;Pop 1]@[Psi 0]
./compile.ml:                    else if(fname="__keyStroke") then
./compile.ml:                        List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                        (-11);Pop 1]@[Psi 0]
./compile.ml:                    else if(fname="__comboStroke") then
./compile.ml:                        List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                        (-12);Pop 1;Pop 1]@[Psi 0]
./compile.ml:                    else if(fname="__getPoint") then
./compile.ml:                        List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                        (-13)]
./compile.ml:
./compile.ml:                    else if(fname="__getcolor") then
./compile.ml:                        List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                        (-14)]
./compile.ml:                    else if(fname="__print") then
./compile.ml:                        List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                        (-15);Pop 1]@[Psi 0]
./compile.ml:                    else if(fname="__stringOfInt") then
./compile.ml:                        List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                        (-16);]
./compile.ml:                    else if(fname="__stringOfFloat") then
./compile.ml:                        List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                        (-17);Pop 1;Pop 1]@[Psi 0]
./compile.ml:                    else if(fname="__stringOfChar") then
```

```
./compile.ml:                            List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                            (-18);Pop 1;Pop 1]@[Psi 0]
./compile.ml:                        else if(fname="__stringOfBoolean") then
./compile.ml:                            List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                            (-19);Pop 1;Pop 1]@[Psi 0]
./compile.ml:                        else if(fname="__stringOfPoint") then
./compile.ml:                            List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                            (-20);Pop 1;Pop 1]@[Psi 0]
./compile.ml:                        else if(fname="__stringOfKey") then
./compile.ml:                            List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                            (-21);Pop 1;Pop 1]@[Psi 0]
./compile.ml:                        else if(fname="__stringOfColor") then
./compile.ml:                             List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                             (-22);Pop 1;Pop 1]@[Psi 0]
./compile.ml:                        else if(fname="__getASCII") then
./compile.ml:                             List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                             (-23);Pop 1;Pop 1]@[Psi 0]
./compile.ml:                        else if(fname="__getChar") then
./compile.ml:                             List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                             (-24);Pop 1;Pop 1]@[Psi 0]
./compile.ml:                        else if(fname="__getStringLength") then
./compile.ml:                           List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                           (-25);Pop 1;Pop 1]@[Psi 0]
./compile.ml:                        else if(fname="__getCharOfString") then
./compile.ml:                            List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                            (-26);Pop 1;Pop 1]@[Psi 0]
./compile.ml:                        else if(fname="__setCharOfString") then
./compile.ml:                             List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                             (-27);Pop 1;Pop 1]@[Psi 0]
./compile.ml:                        else if(fname="__subOfString") then
./compile.ml:                            List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                            (-28);Pop 1;Pop 1]@[Psi 0]
./compile.ml:                        else if(fname="__delay") then
./compile.ml:                             List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                             (-29);Pop 1]@[Psi 0]
./compile.ml:                        else if(fname="__compareColor") then
./compile.ml:                             List.concat (List.map (fun e->(expr e)) actuals) @[Jsr
./compile.ml:                             (-30);Pop 1;Pop 1]@[Psi 0]
./compile.ml:                        else
./compile.ml:                                (try
./compile.ml:                    (List.concat (List.map expr (List.rev actuals)))@[Jsr
./compile.ml:                   (StringMap.find fname function_indexes)]
./compile.ml:                   with Not_found->raise (Failure("undefined function "^fname)))
./compile.ml:               |NoExpr->[]
./compile.ml:                in
./compile.ml:                (*modified in order to create new variables inside a block(trial)*)
./compile.ml:                let rec stmt = function
./compile.ml:           Block (sl,vl)       ->
./compile.ml:                        let vname_list=List.map (fun (_,n)->n) vl in
./compile.ml:                        let new_local_index=
./compile.ml:                            add_to_map vname_list local_index next_index
./compile.ml:                        in
./compile.ml:                        let num_locals=List.length vl in
./compile.ml:                        (block new_local_index (next_index+num_locals) num_params
./compile.ml:                        sl) @(if num_locals>0 then
./compile.ml:                            [Pop num_locals]
./compile.ml:                        else[])
./compile.ml:
./compile.ml:
./compile.ml:                  | Expr e           -> expr e @ [Pop 1]
./compile.ml:                  | Return e        -> expr e @ [Rts num_params]
./compile.ml:         (*no idea how to change this if*)
./compile.ml:                  | If (p, t) -> begin match (p,t) with
./compile.ml:                    |([],[])->[Bra 1]
./compile.ml:                    |([],t)->let t'=(List.hd t) in stmt t'
./compile.ml:                    |(p,t)->let p'=List.hd p and t'= stmt (List.hd t) and f'= stmt (If(List.tl
p,List.tl t)) in expr p' @ [Beq(2 + List.length t')] @t' @ [Bra(1 + List.length f')] @ f'
./compile.ml:                    end
./compile.ml:                  | VarDecl(t,name,init)->expr(init_value t init)(*begin match e1 with
(a,b,Some(c))->expr c
./compile.ml:                  |(a,b,None)-> [Bra 1] end*)
./compile.ml:                  | For (e1, e2, e3, b) ->
./compile.ml:                          stmt (Block([Expr(e1); While(e2,
./compile.ml:                          Block([b;Expr(e3)],takelistout b))],takelistout b))
./compile.ml:
./compile.ml:                  (*Just need to add the initialization in v1 into the variable
./compile.ml:                   * stack of b and others are the same*)
./compile.ml:(*                  (takelistout b)@(var_list v1) *)
./compile.ml:                  |   ForWithDecl (v1, e2, e3, b) ->
./compile.ml:                          let li=(takelistout
./compile.ml:                          b)@(var_list v1) in
./compile.ml:                          stmt (Block([While(e2,Block([b;Expr(e3)],li))],li))
```

```
./compile.ml:
./compile.ml:                   |   Switch (e, el, sl) -> begin match (e, el, sl) with
./compile.ml:                       (_, [], []) -> [Bra 1]
./compile.ml:                   | (_, [], t) -> let t'=(List.hd t) in stmt t'
./compile.ml:                   | (e, el, sl) -> let p = List.hd el and t' = stmt (List.hd sl) and f' = stmt (Switch(e, List.tl el, List.tl sl)) in expr e @ expr p @ [Bin(Type.Eq)] @ [Bne(2)] @ [Bra(2 + List.length t')] @ t' @[Bra(1+List.length f')] @ f'
./compile.ml:                         end
./compile.ml:
./compile.ml:
./compile.ml:                   |   Break (i) -> [Pop (i+1)]
./compile.ml:
./compile.ml:                   |   Continue (i) -> [Pop (i+1)]
./compile.ml:
./compile.ml:
./compile.ml:                   | While (e, b) ->
./compile.ml:               let b' = stmt b and e' = expr e in
./compile.ml:                 [Bra (1+ List.length b')] @ b' @ e' @
./compile.ml:                 [Bne (-(List.length b' + List.length e'))]
./compile.ml:
./compile.ml:           in
./compile.ml:           List.concat (List.map stmt sl)
./compile.ml:       in
./compile.ml:       let data=
./compile.ml:           let assign_globals=
./compile.ml:               List.map (fun (t,name,init)->
./compile.ml:                   Expr(Assign(name,init_value t init),t,true)) globals
./compile.ml:           in
./compile.ml:           [Glb (List.length assign_globals)]@
./compile.ml:           block StringMap.empty 0 0 assign_globals @
./compile.ml:           try
./compile.ml:               [Jsr (StringMap.find "main" function_indexes);Hlt]
./compile.ml:           with Not_found->
./compile.ml:               raise (Failure "main function not found")
./compile.ml:       in
./compile.ml:       let text=
./compile.ml:           let trans_func f=
./compile.ml:               let param_names=List.map snd f.formals in
./compile.ml:               let num_params=List.length param_names in
./compile.ml:               let param_index=
./compile.ml:                   add_to_map param_names StringMap.empty (-1 -num_params)
```

```
./compile.ml:           in
./compile.ml:           let local_names=List.map snd f.locals in
./compile.ml:           let num_locals=List.length local_names in
./compile.ml:           let local_index=
./compile.ml:               add_to_map local_names param_index 1
./compile.ml:           in
./compile.ml:           let bodycodes=
./compile.ml:               block local_index (num_locals +1) num_params f.body
./compile.ml:           in
./compile.ml:           List.iter (function b->()) bodycodes;(*not quite understand*)
./compile.ml:           [Ent num_locals]@bodycodes@
./compile.ml:           (*
./compile.ml:                               Void_T->Integer(0)
./compile.ml:                               | Point_T->Point(0,0)
./compile.ml:                               | Int_T -> Integer(0)
./compile.ml:                               |Key_T->String("")
./compile.ml:                               |Color_T->Color(0,0,0)
./compile.ml:             | Float_T -> Float(0.0)
./compile.ml:             | Char_T -> Char('\000')
./compile.ml:             | String_T -> String("")
./compile.ml:             | Boolean_T -> Boolean(false)*)
./compile.ml:           (match f.rtype with
./compile.ml:
./compile.ml:           |Int_T->[Psi 0]
./compile.ml:           |Key_T->[Psk ""]
./compile.ml:           |Color_T->[Psco (0,0,0)]
./compile.ml:           |Point_T->[Psp (0,0)]
./compile.ml:           |Float_T -> [Psf 0.0]
./compile.ml:           | Char_T -> [Psc '\000']
./compile.ml:         | String_T -> [Pss ""]
./compile.ml:             | Boolean_T -> [Psb false]
./compile.ml:         | Void_T -> [Psi 0]) @
./compile.ml:       [Rts num_params]
./compile.ml:       in
./compile.ml:       let func_bodies=
./compile.ml:           (List.map trans_func functions)
./compile.ml:       in
./compile.ml:       let (_,func_offsets)=
./compile.ml:           List.fold_left (fun (i,offsets) body->
./compile.ml:               ((List.length body) +i,i::offsets))
./compile.ml:           ((List.length data),[])func_bodies
```

```
./compile.ml:               in
./compile.ml:               let array_offsets=Array.of_list (List.rev func_offsets) in
./compile.ml:               List.map (function
./compile.ml:                       |Jsr i when i>=0->Jsr array_offsets.(i)
./compile.ml:                       |_ as s->s)
./compile.ml:               (List.concat (data::func_bodies))
./compile.ml:       in
./compile.ml:       Bytecode.string_of_prog {num_globals=10;text=Array.of_list text}
```

6. easkey.ml

```
./easkey.ml:(*Top Level @Keqiu
./easkey.ml: *Later change the Parser.program Scanner...part into SAST part
./easkey.ml: * *)
./easkey.ml:(*open Bytecode*)
./easkey.ml:(*open Bytecode*)
./easkey.ml:
./easkey.ml:if (Array.length Sys.argv<2) then
./easkey.ml:       print_endline "Usage: easkey <file_name>"
./easkey.ml:
./easkey.ml:else
./easkey.ml:       let in_filename=Sys.argv.(1) in
./easkey.ml:       let fin =open_in in_filename in
./easkey.ml:       let lexbuf=Lexing.from_channel fin in
./easkey.ml:
./easkey.ml:
./easkey.ml:       let program=Analyzer.semantically_check (Parser.program Scanner.token lexbuf) in
let listing=Compile.translate program in
./easkey.ml:       print_endline listing
```

7. instructions.h

```
./instructions.h:#pragma once

./instructions.h:#include<memory>

./instructions.h:#include <string>

./instructions.h:#include "tyname.h"

./instructions.h:/*

./instructions.h:enum Type

./instructions.h:{
```

```
./instructions.h: glb,psi,psf,pss,psb,psc,pop,uop,bin,lod,str,lfr,

./instructions.h: alc,fld,sfd,pst,scd,next,add,sub,mul,divs,mod,eq,neq,lt,leq,gt,geq,val,chd,

./instructions.h: neg,num,at,cln,fat,null

./instructions.h:};

./instructions.h:

./instructions.h:enum subT

./instructions.h:{

./instructions.h:

./instructions.h:};

./instructions.h:*/

./instructions.h:using namespace std;

./instructions.h:class instructions

./instructions.h:{

./instructions.h:public:

./instructions.h: instructions(string type,shared_ptr<void>operand);

./instructions.h: ~instructions(void);

./instructions.h: //Type getType();

./instructions.h: //subT getSubType();

./instructions.h: shared_ptr<void> getOperand();

./instructions.h: string getType();
```

```
./instructions.h:

./instructions.h:private:

./instructions.h:string type;

./instructions.h://Type type;

./instructions.h://subT subType;

./instructions.h:shared_ptr<void> operand;

./instructions.h:};

./instructions.h:
```

8. parser.mly.

```
/parser.mly:/* 10:45 Dec 16, 2012 wzh */

./parser.mly:

./parser.mly:%{ open Type %}

./parser.mly:%{ open Ast %}

./parser.mly:

./parser.mly:%token LPAREN RPAREN LBRACKET RBRACKET COLON SEMI COMMA PLUS MINUS TIMES DIVIDE MOD ASSIGN EQ NEQ LT

./parser.mly:%token LEQ GT GEQ NOT AND OR ATX ATY ATR ATG ATB ATK IF ELSEIF ELSE SWITCH CASE DEFAULT FOR WHILE

./parser.mly:%token CONTINUE BREAK FUNCTION RETURN END VOID INT FLOAT CHAR STRING BOOLEAN POINT COLOR KEY EOF

./parser.mly:

./parser.mly:%token <int> INT_LITERAL

./parser.mly:%token <char> CHAR_LITERAL
```

```
./parser.mly:%token <float> FLOAT_LITERAL

./parser.mly:%token <string> STRING_LITERAL

./parser.mly:%token <bool> BOOLEAN_LITERAL

./parser.mly:%token <int*int> POINT_LITERAL

./parser.mly:%token <int*int*int> COLOR_LITERAL

./parser.mly:%token <string> KEY_LITERAL

./parser.mly:%token <string> ID

./parser.mly:

./parser.mly:%right ASSIGN

./parser.mly:%left OR

./parser.mly:%left AND

./parser.mly:%left EQ NEQ

./parser.mly:%left LT GT LEQ GEQ

./parser.mly:%left PLUS MINUS

./parser.mly:%left TIMES DIVIDE MOD

./parser.mly:%nonassoc UMINUS NOT ATX ATY ATR ATG ATB ATK

./parser.mly:

./parser.mly:%start program

./parser.mly:%type <Ast.program> program

./parser.mly:
```

```
./parser.mly:%%

./parser.mly:

./parser.mly:program:

./parser.mly:    /* nothing */ { [] }

./parser.mly:| program vdecl { Var_decl($2) :: $1 }

./parser.mly:| program fdecl { Func_decl($2) :: $1 } /* in reverse order */

./parser.mly:

./parser.mly:fdecl:

./parser.mly:     FUNCTION return_type_specifier ID LPAREN formals_opt RPAREN stmt_list END
FUNCTION

./parser.mly:            { {    rtype= $2;

./parser.mly:                   fname       = $3;

./parser.mly:                   formals     = $5;

./parser.mly:                   body= List.rev $7 } }

./parser.mly:

./parser.mly:formals_opt:

./parser.mly:    /* nothing */                    { [] }

./parser.mly:| formal_list                  { List.rev $1 }

./parser.mly:

./parser.mly:formal_list:

./parser.mly:    type_specifier ID                              { [($1, $2)] }
```

```
./parser.mly:| formal_list COMMA type_specifier ID          { ($3, $4)::$1 } /* in reverse order */

./parser.mly:

./parser.mly:stmt_list:

./parser.mly:    /* nothing */                  { [] }

./parser.mly:| stmt_list stmt            { $2 :: $1 }     /* in reverse order */

./parser.mly:

./parser.mly:stmt:

./parser.mly:    expr SEMI                                               { Expr($1) }

./parser.mly:| RETURN expr SEMI                                                 { Return($2) }

./parser.mly:| RETURN SEMI                                                      { Return(NoExpr) }

./parser.mly:| IF LPAREN expr RPAREN stmt_list END IF                              { If([$3],
[Block(List.rev $5)]) }

./parser.mly:| IF LPAREN expr RPAREN stmt_list ELSE stmt_list END IF                  { If([$3],
[Block(List.rev $5); Block(List.rev $7)]) }

./parser.mly:| IF LPAREN expr RPAREN stmt_list elseif_stmt END IF                     { If($3::fst
$6, Block(List.rev $5)::snd $6) }

./parser.mly:| FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN stmt_list END FOR
{ For($3, $5, $7, Block(List.rev $9)) }

./parser.mly:| FOR LPAREN vdecl expr_opt SEMI expr_opt RPAREN stmt_list END FOR
{ ForWithDecl($3, $4, $6, Block(List.rev $8)) }

./parser.mly:| WHILE LPAREN expr RPAREN stmt_list END WHILE                         { While($3,
Block(List.rev $5)) }

./parser.mly:| SWITCH LPAREN expr RPAREN stmt_list case_stmt END SWITCH
{ Switch($3, fst $6, snd $6) }
```

```
./parser.mly:| SWITCH LPAREN expr RPAREN stmt_list END SWITCH
{ Switch($3, [], []) } /* stmts before cases will not be executed */

./parser.mly:| CONTINUE SEMI                                  { Continue }

./parser.mly:| BREAK SEMI                                     { Break }

./parser.mly:| vdecl                                      { VarDecl((fst $1, snd $1)) }

./parser.mly:

./parser.mly:init_list:                            /* in reverse order */

./parser.mly:    init                    { [ $1 ] }

./parser.mly:| init_list COMMA init         { $3 :: $1 }

./parser.mly:

./parser.mly:init:

./parser.mly:    ID ASSIGN expr             { ($1, $3) }

./parser.mly:| ID                       { ($1, NoExpr) }

./parser.mly:

./parser.mly:vdecl:

./parser.mly:    type_specifier init_list SEMI     { ($1, List.rev $2) }

./parser.mly:

./parser.mly:expr:

./parser.mly:    ID                     { Id($1) }

./parser.mly:| constant                 { Literal($1) }

./parser.mly:| ID ASSIGN expr           { Assign($1, $3) }
```

```
./parser.mly:| expr PLUS expr              { Binop($1, Add, $3) }
./parser.mly:| expr MINUS expr                  { Binop($1, Sub, $3) }
./parser.mly:| expr TIMES expr             { Binop($1, Mul, $3) }
./parser.mly:| expr DIVIDE expr                 { Binop($1, Div, $3) }
./parser.mly:| expr MOD expr               { Binop($1, Mod, $3) }
./parser.mly:| expr GT expr                     { Binop($1, Gt, $3) }
./parser.mly:| expr LT expr                     { Binop($1, Lt, $3) }
./parser.mly:| expr GEQ expr               { Binop($1, Geq, $3) }
./parser.mly:| expr LEQ expr               { Binop($1, Leq, $3) }
./parser.mly:| expr EQ expr                     { Binop($1, Eq, $3) }
./parser.mly:| expr NEQ expr               { Binop($1, Neq, $3) }
./parser.mly:| expr AND expr               { Binop($1, And, $3) }
./parser.mly:| expr OR expr                     { Binop($1, Or, $3) }
./parser.mly:| MINUS expr %prec UMINUS              { Unop(Neg, $2) }
./parser.mly:| NOT expr                    { Unop(Not, $2) }
./parser.mly:| expr ATX                    { Unop(Atx, $1) }
./parser.mly:| expr ATY                    { Unop(Aty, $1) }
./parser.mly:| expr ATR                    { Unop(Atr, $1) }
./parser.mly:| expr ATG                    { Unop(Atg, $1) }
./parser.mly:| expr ATB                    { Unop(Atb, $1) }
./parser.mly:| expr ATK                    { Unop(Atk, $1) }
```

```
./parser.mly:| ID LPAREN actuals_opt RPAREN     { Call($1, $3) }

./parser.mly:| LPAREN expr RPAREN                 { $2 }

./parser.mly:

./parser.mly:constant:

./parser.mly:    INT_LITERAL                       { Integer($1) }

./parser.mly:| CHAR_LITERAL                      { Char($1) }

./parser.mly:| STRING_LITERAL                   { String($1) }

./parser.mly:| FLOAT_LITERAL                     { Float($1) }

./parser.mly:| BOOLEAN_LITERAL                       { Boolean($1) }

./parser.mly:| POINT_LITERAL                     { Point(fst $1, snd $1) }

./parser.mly:| KEY_LITERAL                         { Key($1) }

./parser.mly:| COLOR_LITERAL                    { Color((let fst (a,_,_) = a in fst $1), (let snd (_,b,_) = b
in snd $1), (let thd (_,_,c) = c in thd $1)) }

./parser.mly:

./parser.mly:elseif_stmt:

./parser.mly:    ELSEIF LPAREN expr RPAREN stmt_list                        { ([$3], [Block(List.rev
$5)]) }

./parser.mly:| ELSEIF LPAREN expr RPAREN stmt_list elseif_stmt               { ($3::fst $6,
Block(List.rev $5)::snd $6) }

./parser.mly:| ELSEIF LPAREN expr RPAREN stmt_list ELSE stmt_list { ([$3], [Block(List.rev $5);
Block(List.rev $7)]) }

./parser.mly:
```

```
./parser.mly:case_stmt:

./parser.mly:    CASE expr COLON stmt_list                        { ([$2], [Block(List.rev $4)]) }

./parser.mly:| CASE expr COLON stmt_list case_stmt                     { ($2::fst $5, Block(List.rev
$4)::snd $5) }

./parser.mly:| DEFAULT COLON stmt_list                           { ([], [Block(List.rev $3)]) }

./parser.mly:

./parser.mly:expr_opt:

./parser.mly:    /* nothing */                    { NoExpr }

./parser.mly:| expr                           { $1 }

./parser.mly:

./parser.mly:actuals_opt:

./parser.mly:    /* nothing */                    { [] }

./parser.mly:| actuals_list                   { List.rev $1 }

./parser.mly:

./parser.mly:actuals_list:

./parser.mly:    expr                             { [$1] }

./parser.mly:| actuals_list COMMA expr          { $3 :: $1 }       /* in reverse order */

./parser.mly:

./parser.mly:type_specifier:

./parser.mly:    INT                              { Int_T }

./parser.mly:| FLOAT                          { Float_T }
```

```
./parser.mly:| CHAR                    { Char_T }
./parser.mly:| STRING                  { String_T }
./parser.mly:| BOOLEAN                 { Boolean_T }
./parser.mly:| POINT                   { Point_T }
./parser.mly:| COLOR                   { Color_T }
./parser.mly:| KEY                     { Key_T }
./parser.mly:
./parser.mly:return_type_specifier:
./parser.mly:    VOID                           { Void_T }
./parser.mly:| type_specifier          { $1 }
./point.cpp:#include "point.h"
```

9. point.cpp

```
./point.cpp:
./point.cpp:/*@Author: Keqiu Hu
./point.cpp:the point class
./point.cpp:*/
./point.cpp:point::point(int a,int b)
./point.cpp:{
./point.cpp:	x=a;
./point.cpp:	y=b;
./point.cpp:}
```

```
./point.cpp:point::point()

./point.cpp:{

./point.cpp:	x=0;

./point.cpp:	y=0;

./point.cpp:}

./point.cpp:

./point.cpp:

./point.cpp:point::~point(void)

./point.cpp:{

./point.cpp:}
```

10. point.h

```
./point.h:#pragma once

./point.h:/*@Author: Keqiu Hu

./point.h:the point class declaration

./point.h:*/

./point.h:class point

./point.h:{

./point.h:public:

./point.h:	int x;

./point.h:	int y;

./point.h:	point(int,int);

./point.h:	point();
./point.h:  ~point(void);

./point.h:};

./point.h:
```

11. program.h

```
./program.h:#pragma once

./program.h:#include "instructions.h"

./program.h:#include <vector>

./program.h:#include <stack>

./program.h:#include <memory>

./program.h:

./program.h:using namespace std;

./program.h:class program

./program.h:{

./program.h:public:

./program.h:     int inpoint;

./program.h:     vector<tyname> progs;

./program.h:     //stack< tyname > progs;

./program.h:     vector<instructions> instru;

./program.h:     void dump();

./program.h:     vector< tyname > globals;

./program.h:     program(vector <instructions> ins);
./program.h:     ~program(void);

./program.h:     void execute();

./program.h:};

./program.h:
```

12. rgb.cpp

```
./rgb.cpp:#include "rgb.h"

./rgb.cpp:

./rgb.cpp:

./rgb.cpp:rgb::rgb(int a,int d,int c)

./rgb.cpp:{

./rgb.cpp: r=a;

./rgb.cpp: g=d;

./rgb.cpp: b=c;

./rgb.cpp:

./rgb.cpp:}

./rgb.cpp:rgb::rgb(){

./rgb.cpp: r=0;

./rgb.cpp: g=0;

./rgb.cpp: b=0;

./rgb.cpp:}

./rgb.cpp:
```

```
./rgb.cpp:

./rgb.cpp:rgb::~rgb(void)

./rgb.cpp:{

./rgb.cpp:}

./rgb.h:#pragma once

./rgb.h:class rgb
```

13. rgb.h

```
./rgb.h:{

./rgb.h:public:

./rgb.h:	int r;

./rgb.h:	int g;

./rgb.h:	int b;

./rgb.h:

./rgb.h:	rgb(int,int,int);

./rgb.h:	rgb();

./rgb.h:	~rgb(void);

./rgb.h:};

./rgb.h:
```

14. sast.mli

```
./sast.mli:(* 11:37 Dec 16, 2012 hxy, wzh *)

./sast.mli:

./sast.mli:open Type
```

```
./sast.mli:

./sast.mli:type expr_detail =                   (* Expressions *)

./sast.mli:    Id of string                     (* foo *)

./sast.mli:| Literal of constant                (* 12 $PgDn$ '@' *)

./sast.mli:| Assign of string * expr                (* foo = 823 *)

./sast.mli:| Binop of expr * binop * expr               (* x=a+b *)

./sast.mli:| Unop of unop * expr                        (* !a -t p@x *)

./sast.mli:| Call of string * expr list         (* foo(a,23)) *)

./sast.mli:| NoExpr                             (* for(;;) *)

./sast.mli:and expr = expr_detail * t * bool            (* Expressions with type and integer that
indicates constant expression or not *)

./sast.mli:

./sast.mli:type var_decl = t * string * expr option     (* Variable declaration *)

./sast.mli:

./sast.mli:type stmt =                          (* Statements *)

./sast.mli:    Expr of expr                     (* foo = bar + 3; *)

./sast.mli:| Block of stmt list * (t * string) list     (* statement list and var list *)

./sast.mli:| Return of expr                         (* return 42 *)

./sast.mli:| If of expr list * stmt list            (* if(foo==42) ... elseif(foo<42) ... else ... end if
*)

./sast.mli:| For of expr * expr * expr * stmt           (* for(i=0;i<10;i=i+1) ... end for *)
```

```
./sast.mli:| ForWithDecl of var_decl list * expr * expr * stmt    (* for(int i=0;i<10;i=i+1) ... end for *)
./sast.mli:| While of expr * stmt                      (* while (i <10) ... end while *)
./sast.mli:| Switch of expr * expr list * stmt list    (* switch(x+y) case 1: a=8; b=23; case 2: c=8; d=23; end switch *)
./sast.mli:| Continue of int                           (* continue statement with num of locals that need to be popped *)
./sast.mli:| Break of int                          (* break statement num of locals that need to be popped *)
./sast.mli:| VarDecl of var_decl                       (* int a, b = 3, c; *)
./sast.mli:
./sast.mli:type func_decl = {                          (* Function declaration *)
./sast.mli: rtype : t;                             (* return type of the function *)
./sast.mli: fname : string ;                   (* name of the function *)
./sast.mli: formals : (t * string) list;           (* types and names of formal arguments *)
./sast.mli: locals : (t * string) list ;           (* local variables *)
./sast.mli: body : stmt list ;                     (* body of the function *)
./sast.mli:}
./sast.mli:
./sast.mli:type program =
./sast.mli: var_decl list * func_decl list         (* global vars, funcs *)
```

15. scanner.mll

```
./scanner.mll:(* 03:40 Dec 16, 2012 wzh & jinqi*)
./scanner.mll:
./scanner.mll:{ open Parser (* Get the token types *)
./scanner.mll:
```

```
./scanner.mll:let get_point str =        (* string -> int * int = <fun>, e.g. "{12,345}" -> (12,345) *)
./scanner.mll:let index = String.index str ',' in
./scanner.mll:(int_of_string (String.sub str 1 (index-1)), int_of_string (String.sub str (index+1) (String.length str-index-2)))
./scanner.mll:
./scanner.mll:let get_color str =       (* string -> int * int * int = <fun>, e.g. "#12,23,34#" -> (12,23,34) *)
./scanner.mll:let index1 = String.index_from str 0 ',' in
./scanner.mll:let index2 = String.index_from str (index1+1) ',' in
./scanner.mll:(int_of_string (String.sub str 1 (index1-1)), int_of_string (String.sub str (index1+1) (index2-index1-1)), int_of_string (String.sub str (index2+1) (String.length str-index2-2)))
./scanner.mll:
./scanner.mll:let check_color_boundary (r, g, b) =
./scanner.mll:let list = [r; g; b] in
./scanner.mll:let checked_list = List.map (fun c -> if c >= 0 && c <= 255 then c else
./scanner.mll:raise(Failure("color #" ^ (string_of_int r) ^ ", " ^ (string_of_int g) ^ ", " ^ (string_of_int b) ^ "# out of boundary"))) list
./scanner.mll:          in (fun [r; g; b] -> (r, g, b)) checked_list
./scanner.mll:
./scanner.mll:let get_key str =   (* string -> string = <fun>, e.g. "$Home$" -> "Home" *)
./scanner.mll:String.sub str 1 (String.length str-2)
./scanner.mll:
./scanner.mll:let get_ascii str =        (* string -> string = <fun>, e.g. "'\121'" -> 121 *)
./scanner.mll:String.sub str 2 3
./scanner.mll:
./scanner.mll:let get_char n str =       (* string -> char = <fun>, e.g. "\'\\n\'" -> '\n' *)
./scanner.mll:let c = String.get str n
./scanner.mll:in match c with
./scanner.mll:    '\\' -> '\\'
./scanner.mll:| 'b'    -> '\b'
./scanner.mll:| 't'    -> '\t'
./scanner.mll:| 'n'    -> '\n'
./scanner.mll:| 'r'    -> '\r'
./scanner.mll:| '\'' -> '\''
./scanner.mll:| '\"' -> '\"'
./scanner.mll:| _       -> raise(Failure("Illegal escape character : " ^ "\\" ^ (String.make 1 c)))
./scanner.mll:
./scanner.mll:let rec get_str str tar_str =    (* val get_str : string -> string -> string = <fun>, e.g. "Hello\\n" -> "Hello\n" *)
./scanner.mll:match str with
./scanner.mll:    "" -> tar_str
```

```
./scanner.mll:| _    -> if(String.get str 0) = '\\' then
./scanner.mll:           let c = String.get str 1 in
./scanner.mll:           if (c >= '0' && c <= '9') then
./scanner.mll:                   get_str (String.sub str 4 ((String.length str)-4)) (tar_str ^ (String.make 1 (let lxm = String.sub str 1 3 in
./scanner.mll:                           try Char.chr (int_of_string lxm) with
./scanner.mll:                                   Invalid_argument "Char.chr" -> raise (Failure("Character \\" ^ lxm ^ " out of boundary")))))
./scanner.mll:           else
./scanner.mll:                   get_str (String.sub str 2 ((String.length str)-2)) (tar_str ^ (String.make 1 (get_char 1 (String.sub str 0 2))))
./scanner.mll:   else
./scanner.mll:           get_str (String.sub str 1 ((String.length str)-1)) (tar_str ^ (String.make 1 (String.get str 0)))
./scanner.mll:
./scanner.mll:}
./scanner.mll:
./scanner.mll:let letter = ['a'-'z' 'A'-'Z']
./scanner.mll:let digit = ['0'-'9']
./scanner.mll:let character = ['\b' '\t' '\n' '\r' ' ' '!' '#' '$' '%' '&' '(' ')' '*' '+' ',' '-' '.' '/' ':' ';' '<' '=' '>' '?' '@' '[' ']'
./scanner.mll:            '^' '_' '`' '{' '|' '}' '~'] | letter | digit            (*No backslash and quotation marks here *)
./scanner.mll:
./scanner.mll:let exponent = 'e' ['+' '-']? ['0'-'9']+
./scanner.mll:let keys = ("a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k"|"l"|"m"|"n"|"o"|"p"|"q"|"r"|"s"|"t"|"u"|"v"|"w"|"x"|"y"|"z"
./scanner.mll: |"A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"|"M"|"N"|"O"|"P"|"Q"|"R"|"S"|"T"|"U"|"V"|"W"|"X"|"Y"|"Z"
./scanner.mll: |"0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"|"`"|"~"|"!"|"@"|"#"|"$"|"%"|"^"|"&"|"*"|"("|")"|"-"|"_"|"+"|"="
./scanner.mll: |"["|"{"|"]"|"}"|"\\"|"|"|";"|":"|"\'"|"\""|","|"<"|"."|">"|"/"|"?"|"Enter"|"Space"|"Backspace"|"Esc"|"Tab"|"CapsLk"
./scanner.mll: |"Shift"|"Ctrl"|"Alt"|"Windows"|"Up"|"Down"|"Left"|"Right"|"Delete"|"Home"|"End"|"PgUp"|"PgDn"|"PrtSc"|"ScrLk"
./scanner.mll:          |"Insert"|"F1"|"F2"|"F3"|"F4"|"F5"|"F6"|"F7"|"F8"|"F9"|"F10"|"F11"|"F12"|"Null")
./scanner.mll:let escape = ("\\\\"|"\\b"|"\\t"|"\\n"|"\\r"|"\\\'"|"\\\"")
./scanner.mll:
./scanner.mll:rule token = parse
```

```
./scanner.mll:[' ' '\t' '\r' '\n'] { token lexbuf }      (* Whitespace *)
./scanner.mll:| "/*"                { comment lexbuf } (* Comments *)
./scanner.mll:| '('         { LPAREN }          (* Punctuation *)
./scanner.mll:| ')'         { RPAREN }
./scanner.mll:| '['         { LBRACKET }
./scanner.mll:| ']'         { RBRACKET }
./scanner.mll:| ':'         { COLON }
./scanner.mll:| ';'         { SEMI }
./scanner.mll:| ','         { COMMA }
./scanner.mll:| '+'         { PLUS }
./scanner.mll:| '-'         { MINUS }
./scanner.mll:| '*'         { TIMES }
./scanner.mll:| '/'         { DIVIDE }
./scanner.mll:| "%"         { MOD }
./scanner.mll:| '='         { ASSIGN }
./scanner.mll:| "=="        { EQ }
./scanner.mll:| "!="        { NEQ }
./scanner.mll:| '<'         { LT }
./scanner.mll:| "<="        { LEQ }
./scanner.mll:| ">"         { GT }
./scanner.mll:| ">="        { GEQ }
./scanner.mll:| '!'         { NOT }
./scanner.mll:| "&&"        { AND }
./scanner.mll:| "||"        { OR }
./scanner.mll:| "@x"        { ATX }
./scanner.mll:| "@y"        { ATY }
./scanner.mll:| "@r"        { ATR }
./scanner.mll:| "@g"        { ATG }
./scanner.mll:| "@b"        { ATB }
./scanner.mll:| "@k"        { ATK }
./scanner.mll:| "if"        { IF }              (* Keywords *)
./scanner.mll:| "elseif"    { ELSEIF }
./scanner.mll:| "else"      { ELSE }
./scanner.mll:| "switch"    { SWITCH }
./scanner.mll:| "case"      { CASE }
./scanner.mll:| "default"   { DEFAULT }
./scanner.mll:| "for"       { FOR }
./scanner.mll:| "while"     { WHILE }
./scanner.mll:| "continue" { CONTINUE }
./scanner.mll:| "break"     { BREAK }
./scanner.mll:| "function" { FUNCTION }
```

```
    ./scanner.mll:| "return"      { RETURN }
    ./scanner.mll:| "end"         { END }
    ./scanner.mll:| "void"        { VOID }
    ./scanner.mll:| "int"         { INT }
    ./scanner.mll:| "float"       { FLOAT }
    ./scanner.mll:| "char"        { CHAR }
    ./scanner.mll:| "string"      { STRING }
    ./scanner.mll:| "boolean"     { BOOLEAN }
    ./scanner.mll:| "point"       { POINT }
    ./scanner.mll:| "color"       { COLOR }
    ./scanner.mll:| "key"         { KEY }
    ./scanner.mll:| eof           { EOF }          (* End-of-file *)
    ./scanner.mll:| digit+                                           as lxm
{ INT_LITERAL(int_of_string lxm) }          (* integers *)
    ./scanner.mll:| '\'' character '\''                              as lxm
{ CHAR_LITERAL(String.get lxm 1) }          (* char *)
    ./scanner.mll:| '\'' escape '\''                                 as lxm
{ CHAR_LITERAL(get_char 2 lxm) }
    ./scanner.mll:| '\'' '\\' digit digit digit '\''                 as lxm { CHAR_LITERAL(try
Char.chr (int_of_string (get_ascii lxm))
    ./scanner.mll:                                              with Invalid_argument "Char.chr" ->
raise
    ./scanner.mll:                                              (Failure("Character " ^ lxm ^ " out of
boundary"))) }
    ./scanner.mll:| digit+ ('.' digit* exponent? | exponent) | digit* '.' digit+ exponent?      as lxm
{ FLOAT_LITERAL(float_of_string lxm) }     (* float *)
    ./scanner.mll:| '\"' (character | escape | '\\' digit digit digit )* '\"'          as lxm
{ STRING_LITERAL(get_str (get_key lxm) "") }    (* string *)
    ./scanner.mll:| "true" | "false"                                 as lxm
{ BOOLEAN_LITERAL(bool_of_string lxm) }       (* boolean *)
    ./scanner.mll:| '{' digit+ ',' digit+ '}'                        as lxm
{ POINT_LITERAL(get_point lxm) }            (* point *)
    ./scanner.mll:| '#' digit+ ',' digit+ ',' digit+ '#'                   as lxm
{ COLOR_LITERAL(check_color_boundary(get_color lxm)) }              (* color *)
    ./scanner.mll:| '$' keys '$'                                     as lxm
{ KEY_LITERAL(get_key lxm) }                (* key *)
    ./scanner.mll:| ['a'-'z' 'A'-'Z' '_']['a'-'z' 'A'-'Z' '0'-'9' '_']*          as lxm { ID(lxm) }
    ./scanner.mll:| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }
    ./scanner.mll:
    ./scanner.mll:and comment = parse
    ./scanner.mll:    "*/" { token lexbuf }                    (* End-of-comment *)
```

```
./scanner.mll:| _        { comment lexbuf }          (* Eat everything else *)
```

16. tyname.cpp.

```
/tyname.cpp:#include "tyname.h"

./tyname.cpp:

./tyname.cpp:

./tyname.cpp:tyname::tyname(string a,shared_ptr<void> b)

./tyname.cpp:{

./tyname.cpp:    name=a;

./tyname.cpp:    ptr=b;

./tyname.cpp:}

./tyname.cpp:

./tyname.cpp:

./tyname.cpp:tyname::~tyname(void)

./tyname.cpp:{

./tyname.cpp:}
```

17. tyname.h

```
./tyname.h:#pragma once

./tyname.h:#include<string>

./tyname.h:#include<memory>

./tyname.h:using namespace std;

./tyname.h://to implement type name specification

./tyname.h:class tyname

./tyname.h:{
```

```
./tyname.h:public:
./tyname.h:	tyname(string,shared_ptr<void>);
./tyname.h:	~tyname(void);
./tyname.h:	string name;
./tyname.h:	shared_ptr<void> ptr;
./tyname.h:};
./tyname.h:
```