iCalendar Final Report

Mengfei Ren(mr3258)
Chang Hu(ch2950)
Yu Kang(yk2540)
JiaCheng Chen(jc3940)
Yun Feng(yf2294)

1. Introduction

In modern society, people are always busy with a lot of things. In the fast-paced modern life, sometimes it is very difficult for people to remember to do everything they need to. Missing one event makes very bad consequences at many times. From this angle, it’s essential to make a kind of electronic calendar with the help of which people could create their own calendar, store and manage events. In other programming languages such as Java and C, it’s difficult to program that because it could hardly reflect the abstracted operation. As a result, we plan to design a new language, the iCalendar Language, specifically for the implementations about calendars. We wish that with iCalendar language, users could use more efficient and user-friendly programming methods to implement operations on calendars.

This manual describes the iCalendar language. The iCalendar programming language is an effective programming language for recording your daily events. Users could build their event models (structure) and traverse the calendar (container) to manage their events.

2.Language Tutorial

iCalendar is designed to describe calendars and could be used very conveniently with language-supported calendar operators. User of iCalendar can use these features to define their own calendar-like data structures and write their programs.

2.1 An Introduction

iCalendar is a C-like language. Most of the grammar is similar to that of C. Each source file, with the postfix ".ica", should contain a main() function. Each statement in the code should end with a semicolon.

```
void main ()
{
        # iCalendar statements.
}
```

iCalendar includes primitive types, which are integer, float, Boolean, and string. These types are used in the way just like in C and C++. User can define their own types of event in iCalendar. The definition of the tree type should appear before they are used in other parts of the program. Below is an example of user defined event.

```
Event myEve{
        int time;
        string name;
        string place;
}
void main(){
        myEve e1 = [19,"plt presentation","cs building"];
        print(e1.time);
}
```

When finishing writing the code, cd to the src directory in our project, and type the following command line to compile the program:

```
./ica -c sample_program.ica
```

If the test file is not in the same folder with source code,

```
./ica –c folder/sample_program.ica
```

To run the iCalendar program, type the following command line:

```
java sample_program
```

2.2 Another example

The following code is another example of iCalendar language. In this program, the user adds a 'priority' attribute in the event definition, and at last print out the event with the highest priority.

```
Event myEve{
        int priority;
        string name;
        string place;
}
void main(){
        myEve e1 = [1,"plt presentation","cs building"];
        myEve e2 = [2,"eating","Ollise"];

        print(e1.name);
        print();
        print(e2.name);
        print();
        print("which is important?");
        print();

  if(e1.priority < e2.priority){
                print(e1.name);
  }
  else{
                print(e2.name);
  }
}
```

Let's name this program event.ica. You can compile and run it with the following command.

```
./ica –c event.ica
java event
```

And it should print out:

plt presentation

3. Language Manual

3.1 Lexical Conventions

In iCalendar there are several classes of tokens could be supported. Token types are identifiers, keywords, literals, strings and operators. As in C language, whitespace characters are ignored except insofar as they serve to delineate other tokens in the input stream. If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

3.1.1 Notation

Through the document, *nonterminals* are in italics and **terminals** are bold format. Regular expression-like constructs are used to simplify grammar presentation.

- r* means the pattern r may appear zero or more time. r+ means the r may appear one or more times.
- r? means r may appear zero or once.
- r1 | r2 denotes an option between two patterns.
- r1 r2 denotes an option between two patterns, r1 r2 denotes r1 followed by r2.

3.1.2 Comments

In iCalendar, we use # to mark comments which means a line of comments should be started with the character # and ends with another character #. However, in iCalendar only one-line comments could be supported, that means if the length of comments exceeds one line, remember to put # at the beginning of each line.

Accept:

```
#This is a comment#
#This is another#
    #line of comment#
```

Not accept:

```
#This is another
  line of comment#
```

3.1.3 Identifiers

An identifier consists of a letter or an underscore followed by other letters, digits and underscores. Identifiers are case sensitive, so “left” and “Left” are different identifiers. The length of an identifier is not limited.

Accept: _; _a; a11; b; b12; C23fjdla;

Not accept: 123; 12abc;

In iCalendar global and local variables are both supported. A variable defined inside a block is local, and cannot be called outside the block. For example:

```
int Add(int a, int b)
  {
     Int c = 0;
     return a+b+c;
  }
int b = c; #This is illegal
```

3.1.4 Keywords

In iCalendar, the words listed below are reserved as keywords. Users are not allowed to define their own identifiers.

Reserved identifiers in iCalender:

int	float	string	bool
true	false	if	else
while	for	Event	Calendar
return	void	main	
size	print		

3.1.5 Constants

In iCalender, several kinds of constants including integer constants, float constants, string constants and boolean constants are supported.

Integer constants

Integer constants should be decimal and consists of a sequence of numbers without a decimal point. All integers should be unsigned.

Accept: 45, 0

Not accept: -1, +12, 1.5

Float constants

A floating-point constant consists of a sequence of numbers and a decimal point. Before the decimal point is the integer part and the decimal part is after the point. The integer part could be omit. Examples of valid and invalid cases are listed:

Accept: 0.5; .234; 10.55

Not accept: 1 1.; .; 1.1.1; 2..5

String constants

String constants are demarcated by double quote characters, for instance "iCalendar".

Escape characters are supported.

Character Name	Escape Sequence
Newline	\n
Horizontal tab	\t
Backslash	\\

Boolean constants

Boolean constants consist of the keywords true and false. For example: bool b = true

3.2 Types

In iCalendar, seven fundamental types of objects are supported, but no type conversion is allowed. The fundamental type of objects are a following:

- int
- float
- string
- bool
- Event
- User defined Event structure
- Calendar

Integer type

In iCalendar the only supported integer type is int which can store 32-bits worth of data. This data type is signed.

Float type

In iCalendar the only supported double type is double which can store 64-bits worth of data. Tis data type is signed.

String type

In iCalendar we provide string type, which means a string of unlimited length could be supported. However, the length may be limited because of the amount of computer resources.

Bool type

In iCalendar bool type is supported. Bool type could only take a value of either true or false. However, bool type is very useful and helpful.

Event type

Event type is to enable users to define their own event structure.

User Defined Event Structure

User could define their own event model to use for recording events. For example, Use use Event type to define:

```
Event myOwnEvent
      { int time;
      string description;
      int priority;}“.
```

Calendar type

Calendar is like a list to contain events.

3.3 Expressions

In iCalendar, expressions consist of operators and their operands. In this section, definition for each operator is given. To avoid ambiguity, precedence rules of the operators in iCalendar are also defined in this section.

3.3.1 Operators and Punctuations

Operator	Description
+ - * /	plus/minus/multiple/divide
> = != < >= <=	greater than/equal/not equal/less than/greater than/less than
&& || !	logical and/logical or/logical not
.	dot
,	comma(separate expressions)
;	end of expression
() []	separators

Tokens	Operators	Class	Associativity
* /	Multiplicative	Binary	L
+ -	Additive	Binary	L
< <= => >	Relational comparisons	Binary	L
== !=	Equality comparisons	Binary	L
&&	Logical AND	Binary	L
||	Logical OR	Binary	L
!	Logical NOT	Unary	R
=	Assignment	Binary	R
,	Comma	Binary	L
.	Dot	Binary	L
;	end of expression	Unary	L
() []	separators	Unary	L

3.3.2 Primary Expressions

Identifiers, constants, strings. The type of the expressions depends on identifier, constant or string.

3.3.3 Arithmetic operators

In iCalendar, arithmetic operators are +, -, * and / . + means addition, - means subtraction, * means multiplication and / means division. All of them are binary and left associative. It requires that their operands must be of the same primitive types, and the result will be of the same type.

3.3.4 Comparative operators

In iCalendar, comparative operators are > (greater than), < (less than), >= (greater than or equal to), <= (less than or equal to), ! = (not equal) and == (equal). All of them are binary operators and left-associative. It requires that their operands must be of the same primitive types . The return value is a boolean value indicates the predicate.

3.3.5 Logical operators

Logical operators in iCalendar include && (logical and), || (logical or) and ! (logical not). && and || are binary operators and left-associative. They take two operands of type boolean, and return a boolean value. ! is unary and appears on the left side of the operand. They type of the operand must be of type boolean and the return type is also a boolean value.

3.3.6 Assignment operators

iCalendar's assignment operator is =. It's a binary operator and right-associative. The left operand must be a legal left value, and the right operand must be an expression. When an assignment is taken, the value of the expression on the right is assigned to the left value, and the new value of the left value is returned.

3.3.7 List operations

In iCalendar, users could store their event objects to a calendar.
A calendar is like a list data type. And to visit the element in calendar, use index (e.g. calendar[0]).

3.3.8 Dot operators

User defined event model might have many attributes (like time, event description, location, event priority). These values can be visited by using dot “.”. For example, we have an object: myEven e = [“final exame”,”Dec-10”]. The time can be visited by e.time where “time” is an attribute of myEven type defined by users.

3.4 Statement

Statements in iCalendar contain expressions, return statement, return void statement, conditional statement, loop statement, variable declarations, empty statement, etc.

```
Statement ->
        Block*
        | Expr*
        | Return*
        | ReturnVoid*
        | If statement*
        | For statement*
        | While statement*
        | Vardecl*
        | Empty*
```

3.4.1 Expressions

An expression statement is composed of primary statements with a semicolon at the end of the line. It is used for binary operations.

3.4.2 Return Statements

A compute function returns a value to the caller through return statements.

Return Statement -> **return** Expression **SEMICOLON**

3.4.3 Returnvoid Statements

This function returns a void value to the caller through return statements.

Returnvoid Statement -> **return** void **SEMICOLON**

3.4.4 If Statements

iCalendar supports two kinds of if-else statemens:

If Statement ->
If (Boolean conditions) Statement
|**If** (Boolean condition) Statement **else** Statement

3.4.5 For Statements

In iCalendar, users could use For statement.

For Statement -> **for** (Expr **SEMICOLON** Expr **SEMICOLON** Expr) Statement **SEMICOLON**

3.4.6 While Statements

C programmers are often tolarate with for statement, as they have to type in verbose statement. While, in iCalendar, to avoid verbose, we just use while as the loop statement.
While Statement -> **while** (condition) Statement

3.4.7 Variable Declaration Statements

In iCalendar, if a line contains just a semicolon, that means a null statement and has no meaning.

Variable Declaration -> datatype (**Identifier** (Assign Expr)? **COMMA**)*Identifier (Assign Expr)? **SEMICOLON**

3.4.8 Empty Statements

In iCalendar, if a line contains just a semicolon, that means a null statement and has no meaning.

3.5 Grammar

3.5.1 Program Definition

A program in the iCalendar language consists of a sequence of Event definition structures, variable declarations and functions executing in order.

Program -> Even_Definition_list * Declaration_list * Function_list*

3.5.2 Event Definition

Users could define their own event that consists of a sequence of type name and variable declarations executing in order.

Event -> Typename_string*Declaration_list*

3.5.3 Declarations

All variables must be declared before they can be used. However, variable declarations can be made at any point in a program. Variables become usable after the end of the semi-colon of the statement in which it contained.
In iCalendar, declarations consist of variable declarations and function declarations.

```
Declaration -> Variable Declaration*
               |Function Declaration*
```

iCalendar Language supports functions. Function Declaration is definte below:

Function Declaration -> returnType **Identifier** ((datatype variable)*
|(datatype variable **COMMA**)+ datatype variable)   Block

For example:

```
fun int Add (int a, int b)
{
    return a+b;
}
```

4. Project Plan

4.1 Developing process

Our team met once a week on Wednesday after class in regular weeks. In weeks that had more issues to discuss or tasks than normal, we arranged more meeting in the weekend. We designated a team leader to lead our project; we familiarized ourselves with the deadlines of each part of the project; we set out to learn OCaml; we also exchanged contact information and communicated to each other our weekly availabilities for meetings both in person and online. On the meetings, we first went through things we accomplished the week before, and then set milestones and time plans to the next week.
When it came to real work, we used Dropbox to share code and documents. We used Notepad, vim as source text editor while developing. We worked on personal computers and updated mature work to shared dropbox folder.

4.2 Style Guide

We didn't specify any particular style to use for the project. Since we were with each other while we were wrote the majority of the code, there was no need to have a specified style guide, since we were in constant communication. That is any question on how to write or format a chunk of code we be answered immediately by another member of the group. The only thing we stressed was clarity. When someone wrote something that might be difficult for others to understand, he should create comments.

4.3 Project time line

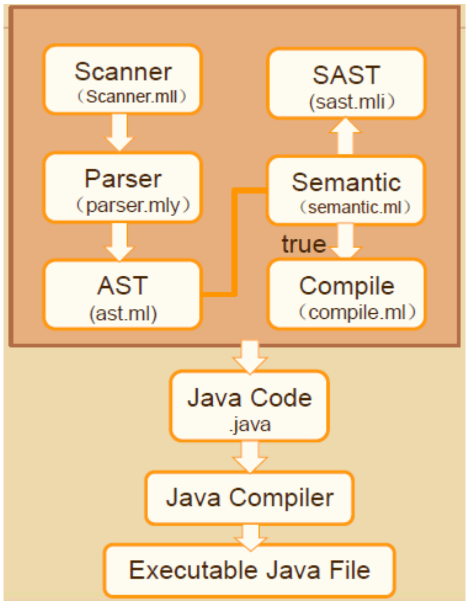
Sept.12 Team formed
Sept.26 Decided on project planning and finished proposal.
Oct. 31 Finished Language Reference Manual and first version of AST
Nov.10 Change language from iChemi to iCalendar
Dec. 1 Finalized Scanner, AST, Parser and printed AST
Dec. 15 Finished a simple version compiler and simple type checking
Dec. 19 Finalized coding of the project and the final report.

5 Architectural Design

This section presents the compiling and executing process of iCalendar programs.

5.1 Compiling and Executing

iCalendar compiler takes iCalendar source code files with postfix name .ica, as input. Firstly, the scanner scans the source code and break it into tokens. Then, the parser runs on the sequence of tokens and construct them into an abstract syntax tree(AST) according to the production rules. Taking the AST as the input, the semantic checks the types on it. It raises exceptions if there are type errors on the AST. Otherwise, the semantic returns “ture” and the compiler starts to compile. The compiler translate the AST to the java source file and javac will be used to compile the translated java program. Since we use AST for the compiling part, some type information is not available. Thus, the Calendar type still has some bugs and this is the future work we need to do.
The compiling process is illustrated in the figure below.



5.2 Responsibility

Scanner.mll is coded by Yu Kang.
Parser.mly is coded by Chang Hu and Jiacheng Chen.
ast.ml is coded by Mengfei Ren and Yun Feng.
sast.mli is coded by Mengfei Ren
Semantic.ml is coded by Mengfei Ren and Chang Hu
Compiler.ml is coded by Yu Kang.
Ica.ml is coded by Mengfei Ren.
Makefile is writted by Yu Kang.
Test cases are written by Jiacheng Chen.
Documentations are managed by Jiacheng Chen, Yu Kang, and ChangHu

6. Testing Plan

6.1 Goal

Formal testing began as soon as a rudimentary compiler was constructed. The test suite on this project was designed to be built alongside the iCalendar compiler. In the first stage, we have printed the syntax tree to test our language design. In the second stage, we have done a lot of tests during debugging process. They are located in /test folder.

6.2 Sample Test Cases

```
int a = 0.5;
float b = "string";
bool c = 2;
string d = true;

void main() {
}
```

Result:	failure, report type error

Case1. Type check

```
int add(a,b){
	return a+b ;
}
void main() {
	add (3.5, 5);
}
```

Result:	failure

Case2. Function parameter check

```
Event myEvent {
int time;
}

myEvent e = [20, ""];

void main() {
}
```

Result:	report failure “Event attributes inconsistent”

Case3. Calendar parameter check

```
int gcd(int a, int b){
        while (a != b) {
                if (a > b) a = a - b;
                else b = b - a;
        }
        return a;
}

void main(){
  int r = gcd(27,72);
  print(r);
}
```

Result: successfully generate java file

Case4. GCD function

```
Event myEve{
        int priority;
        string name;
        string place;
}
void main(){
        myEve e1 = [1,"plt presentation","cs building"];
        myEve e2 = [2,"eating","Ollise"];

        print(e1.name);
        print();
        print(e2.name);
        print();
        print("which is important?");
        print();

  if(e1.priority < e2.priority){
                print(e1.name);
  }
  else{
                print(e2.name);
  }
}
```

Result: successfully generate java file

Case5. Event

7. Lessons Learned

Before taking Programming Languages and Translators course, none of our group members has learned this kind of knowledge. Some members have just touched some programming languages such as Java, C and C++, some members who were not in Computer Science major of undergraduate colleges even have just know a little about programming languages. In one word, the notions of compliers, parsers, scanners and Abstract Syntax Tree are far away from us. However, with a semester's study in class and suffers in doing the project, we gradually know what are these things and how to create these things. We really learned a lot from this course. Besides this, there are some others things that teach us a lot.

At the beginning of this semester, iCalendar is not what we want. Firstly we want to create a programming language named iChemi with which users could make programs especially for evaluating chemistry functions. We had a high enthusiasm on iChemi since we heard nobody on Programming Languages and Translators had made such a language. We spend several group meetings and a lot of personal tasks on designing iChemi, however with deeper study of the course and OCaml, we started to realize that iChemi is not such a programming language that Professor wants us to do. After we consulted Yan Zou-the best TA we have never made, we knew that iChemi would not be successful and we must change our plan. Then we just abandoned what we designed for half a semester and came up with iCalendar. During the second half of the semester, study pressure was so large with a lot of homework and projects, so the time spent on iCalendar is limited, and in the end iCalendar is not perfect because some functions that we designed for it has not been finished. From this bitter experience we know that when start with something difficult and unfamiliar, we should figure out what it is and know what we should do. With these things, the plan would be on the right way and there won't be a lot of pointless time and efforts.

Another thing we learned from the course is that OCaml is very different with regular programing languages like Java and C. OCaml is much more difficult, complicated and abstract so that we should have prepared the difficulties at the very beginning. In OCaml it is not only to know how to code and debug though the amount of codes is very large, but also to try to understand the syntax about how a programming language works. It's just like that creating a new language is no doubt much more difficult than just writing a novel with this language.

What we want to suggest to future teams is that finishing scanner, parser and defining AST before finalizing on the LRM is essential. Before investing many efforts, consulting TAs about the feasibility about the plan. Planned meetings through the whole semester would make the teamwork more efficient. The last and most important thing is working as a team.

Note:

As our language is much like C- or Java, we use AST to generate compiled .java file. But in the end we do not know the what type of Event the Calendar contains, and we cannot generate a corresponding java List<Event>.

Therefore code like: Calendar c = [e1,e2], c[0], can be checked by semantic.ml, but we cannot use our compiler to turn it into java file.

Also, we do not have a good method to deal with “c[i]”, that is, in semantic checking, when we meet “c[i]”, we only know “c” is a Calendar, but we do not know which exactly user defined event type c[i] is.

Some other features of events cannot be finished in time (we hope to sort event in a calendar by some attributes like priority), which is also regretful.

8. Appendix

8.1 type.mli

```
(*Author : Mengfei Ren*)
type t =
            | Int
            | Float
            | String
            | Boolean
            | Void
            | Event_type of string
            | Calendar

type literal =
    IntLit of int
  | FloatLit of float
  | BoolLit of bool
  | StringLit of string

type op =
            | Add
            | Sub
            | Mult
            | Div
            | Equal
            | Neq
            | Less_than
            | Leq
            | Greater_than
            | Geq
            | Or
            | And
            | Not
            | Dot
            | Child
```

8.2 scanner.ml

```
(*Author: Yu Kang*)
 open Parser }

(* letter, digit *)

let esp =   "\\\"" | "\\\\" | "\\n" | "\\t" (*Escape character*)
let letter = ['a'-'z' 'A'-'Z']
let digit = ['0' - '9']
let whitespace = ['\t' ' ' '\r' '\n']
let id = letter (letter | digit | '_' )*
let punc = ['~' '`' '!' '@' '#' '$' '%' '^' '&' '*' '(' ')' '-' '+' '=' ',' '.' '?' '/' '<' '>' ':' '''  ';' '{' '}' '[' ']' '|' ' ']
let stringlit = '"' (letter | digit | punc | esp)*  '"'


rule token = parse
```

```
          whitespace { token lexbuf }
          | "#" { comment lexbuf }

    (* Keywords *)
    | "if"          { IF }
    | "else"        { ELSE }
    | "while"       { WHILE }
    | "for"         { FOR }
    | "return"      { RETURN }
          | "null"              { NULL }
    | "int"         { INT_T }
    | "float"       { FLOAT_T }
    | "string"      { STRING_T }
    | "bool"        { BOOL_T }
    | "void"        { VOID }
    | "Event"           { EVENTTYPE }
          | "Calendar"          {CALENDAR}
    | "main"        as main
                { ID(main) }
    | "print"       as print
                { ID(print) }
    | "size"        as size
                { ID(size) }

    (* Constants *)
    | digit+        as integer
                { INT(int_of_string integer) }
          | digit+ '.' digit*
          | '.' digit+  as float { FLOAT(float_of_string float) }

    | "true"
    | "false"       as bool
                { BOOL(bool_of_string bool) }
          | stringlit   as string { STRING(string) }


    | '{'   { LBRACE }
    | '}'   { RBRACE }
    | ';'   { SEMI }
    | ','   { COMMA }
    | '='   { ASSIGN }

    (* Operators *)
    | "||"   { OR }
    | "&&"   { AND }
    | '!'   { NOT }
    | "!="   { NEQ }
    | '>'   { GT }
    | '<'   { LT }
    | "<="   { LEQ }
    | ">="   { GEQ }
    | "=="   { EQ }
    | '+'   { PLUS }
    | '-'   { MINUS }
    | '*'   { TIMES }
    | '/'   { DIVIDE }
    | '['   { LBRACK }
    | ']'   { RBRACK }
    | '.'   { DOT }
    | '('   { LPAREN }
    | ')'   { RPAREN }

          | letter (letter | digit | '_')* as identifier
                                           { ID(identifier) }

    | eof { EOF }
    | _ as err_char { raise (Failure("illegal character " ^ Char.escaped err_char)) }

(* comment *)
and comment = parse
    '\n' { token lexbuf }
          | _ { comment lexbuf }
```

8.3 ast.ml

```
(*Author: Mengfei Ren , Yun Feng*)
open Type

type expr = (* Expressions *)
   Literal of literal (* 42 *)
  | Id of string (* foo *)
  | Binop of expr * op * expr (* a + b *)
  | Assign of expr * expr (* a = b *)
  | Call of string * expr list
  | Noexpr (* While() *)
  | Uniop of op * expr   (*for unary operators *)
  | ObjValue of (expr list)

type init = WithInit of string * expr
                             | WithoutInit of string

type init_list = init list

type var_decl = t * init_list

type event_def = {
          typename: string;
          members : var_decl list;
}

type stmt = (* Statements  nothing *)
    Block of stmt list
          | Expr of expr   (*foo = bar + 3; *)
          | Return of expr (* return 42 also includes return function_name *)
          | ReturnVoid
          | If of expr * stmt * stmt (* if (foo == 42) {} else {} *)
          | For of expr * expr * expr * stmt
          | While of expr * stmt (* while (i<10) { i = i + 1 } *)
          | Vardecl of var_decl
          | Empty

type func_decl = {
          return_type: t;
   fname : string;
          params : (t * string) list;
   body : stmt list;
}
```

```
type program = {
            eventdef : event_def list;
            globalvar : var_decl list;
            funcdef : func_decl list;
}


(* print the AST*)
let string_of_op  = function
            Add ->"+"
            | Sub ->"-"
            | Mult ->"*"
            | Div ->"/"
            | Equal ->"=="
            | Neq ->"!="
            | Less_than->"<"
            | Leq ->"<="
            | Greater_than ->">"
            | Geq ->">="
            | Or ->"||"
            | And ->"&&"
            | Not ->"!"
            | Dot ->"."
            | Child ->""

let string_of_literal = function
            IntLit(i)-> string_of_int i
  | FloatLit(f) -> string_of_float f
  | BoolLit(b)->string_of_bool b
  | StringLit(s) -> s

let string_of_dt = function
            Int ->"int"
            | Float ->"float"
            | String ->"String"
            | Boolean ->"boolean"
            | Void ->"void"
            | Event_type(myEvent) -> myEvent
            | Calendar-> "Calendar"

let rec string_of_expr = function
    Literal(l) -> string_of_literal l
  | Id(s) -> s
  | Assign(e1,e2)->string_of_expr e1 ^" = "^string_of_expr e2
  | Call(f, el) -> f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr (* While() *)->""
  | Uniop(o,e)->string_of_op o ^" "^string_of_expr e
  | ObjValue(l)->"[" ^ (String.concat ", " (List.map string_of_expr l)) ^" ]"
  | Binop(e1,o,e2)-> match o with
                                                  Child -> string_of_expr e1 ^ "[" ^ string_of_expr e2 ^ "]"
                                                 | _     -> string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^
string_of_expr e2


let string_of_init = function
            WithInit(v,e)-> v ^ "=" ^ string_of_expr e
            |WithoutInit(v)-> v

let string_of_initlist (init) =
            (String.concat ", " (List.map string_of_init init))^";\n"

let string_of_initlist_f (init) =
            (String.concat "f, " (List.map string_of_init init ))^"f;\n"

let string_of_vdecl (t, l)=
            match t with
            Float -> string_of_dt t^" "^ string_of_initlist_f l^"\n"
            |_    -> string_of_dt t^" "^ string_of_initlist l^"\n"

let string_of_event (myevent) =
            "Event "^ (myevent.typename) ^ "{\n"^
            String.concat "\n" (List.map string_of_vdecl myevent.members)^
            "}\n"

let string_of_arg(t, name) =
            (string_of_dt t) ^ " " ^ name

let rec string_of_stmt = function
            Block(stmts) -> "{\n" ^ String.concat "\n" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(e) -> string_of_expr e ^";"
  | Return(e) ->"return "^ (string_of_expr e) ^";"
  | ReturnVoid ->"return ;"
  | If(e, s, Block([]))->"if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1,e2,e3,s)->"for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^ string_of_expr e3  ^
")\n " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
  | Vardecl(v)-> string_of_vdecl v
  | Empty->""

let string_of_func (f) =
            (string_of_dt f.return_type)^" "^ f.fname ^ " (" ^ String.concat ", " (List.map string_of_arg
f.params)^")"^
            "{\n" ^ (String.concat "" (List.map string_of_stmt f.body) )^ "}\n"

let string_of_events = function
            [] -> ""
            | elist -> String.concat "\n" (List.map string_of_event elist)

let string_of_vars = function
   [] -> ""
            | v -> String.concat "\n" (List.map string_of_vdecl v)

let string_of_funs = function
            [] -> ""
            | flist -> String.concat "\n" (List.map string_of_func flist)

let string_of_program (p:program) =
            string_of_events p.eventdef ^ string_of_vars p.globalvar ^  string_of_funs p.funcdef
```

8.4 parser.ml

(Author: Chang Hu , Jiacheng Chen)

```
%{ open Type %}
%{ open Ast %}

%token <string> ID
```

```
%token IF ELSE WHILE FOR RETURN
%token INT_T FLOAT_T STRING_T BOOL_T VOID EVENTTYPE CALENDAR
%token <int>INT
%token <float>FLOAT
%token <bool>BOOL
%token <string>STRING
%token NULL
%token LBRACE RBRACE SEMI COMMA
%token ASSIGN

%token OR AND NOT
%token NEQ GT LT LEQ GEQ EQ
%token PLUS MINUS TIMES DIVIDE MOD
%token LBRACK RBRACK DOT
%token LPAREN RPAREN
%token EOF

%nonassoc NOELSE
%nonassoc ELSE

%right ASSIGN

%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS   /* for binop they are left, but for unop should they be right? */
%left TIMES DIVIDE MOD
%right NOT
%nonassoc LBRACK
%left DOT

%start program
%type <Ast.program> program      /* this type should be AST.program. just put int because AST is not
finished */
%%


program:
program_  { {
                    eventdef = List.rev $1.eventdef;
                    globalvar = List.rev $1.globalvar;
                    funcdef = List.rev $1.funcdef;
  } }

program_:
  /* nothing */     { {eventdef = []; globalvar = []; funcdef = []} }
 | program_ type_def         {{ $1 with eventdef = ($2 :: $1.eventdef)}}
 | program_ decl   { {$1 with globalvar = ($2 :: $1.globalvar) } }
 | program_ func_def         { {$1 with funcdef = ($2 :: $1.funcdef) } }


type_def:
   EVENTTYPE ID LBRACE decl_list RBRACE
                                          { { typename = $2;
                                             members = List.rev $4;
                                            }}

decl_list:
   decl                            { [$1] }
   | decl_list decl                    { $2::$1 }

decl:
   type_specifier init_list SEMI          { ($1, List.rev $2) }

type_specifier:
   INT_T                           { Int }
   | FLOAT_T                          { Float }
   | STRING_T                         { String }
   | BOOL_T                          { Boolean }
   | ID                        { Event_type($1) }
          | CALENDAR                                                                    { Calendar }
          | VOID                          { Void }


init_list:
   init                         { [$1] }
   | init_list COMMA init                { $3::$1 }

init:
   ID                          { WithoutInit($1) }
   | ID ASSIGN expr                      { WithInit($1 ,$3) }

func_def:
   type_specifier ID LPAREN para_list RPAREN stmt_block          { {  return_type = $1;
                                                       fname = $2;
                                                       params = List.rev $4;

                                                                    body = $6

                                                         } }

para_list:
   /* nothing */                                        {[]}
   | para_decl                        { [$1] }
   | para_list COMMA para_decl            { $3::$1 }

para_decl:
   type_specifier ID                    { ($1, $2) }

stmt_block:
   LBRACE stmt_list RBRACE               { List.rev $2 }

stmt_list:
   /* nothing */                     { [] }
   | stmt_list stmt                    { $2::$1 }

stmt:
   expr SEMI                                               { Expr($1) }
   | decl                                { Vardecl($1) }
   | stmt_block                              { Block($1) }
   | IF LPAREN expr RPAREN stmt %prec NOELSE               { If($3, $5, Block([])) }
   | IF LPAREN expr RPAREN stmt ELSE stmt                { If($3, $5, $7) }
   | WHILE LPAREN expr RPAREN stmt                       { While($3, $5) }
   | FOR LPAREN expr SEMI expr SEMI expr RPAREN stmt        { For($3, $5, $7, $9) }
   | RETURN expr SEMI                               { Return($2) }
   | RETURN SEMI                                  { ReturnVoid }
   | SEMI                                  { Empty }

expr:
        | literal                   { Literal($1) }

   | expr PLUS expr                 { Binop($1, Add, $3) }
   | expr MINUS expr                  { Binop($1, Sub, $3) }
   | expr TIMES expr                 { Binop($1, Mult, $3) }
   | expr DIVIDE expr                 { Binop($1, Div, $3) }

   | expr GT expr                          { Binop($1, Greater_than, $3) }
   | expr LT expr                 { Binop($1, Less_than, $3) }
   | expr GEQ expr                  { Binop($1, Geq, $3) }
   | expr LEQ expr                  { Binop($1, Leq, $3) }
   | expr NEQ expr                  { Binop($1, Neq, $3) }
   | expr EQ expr                 { Binop($1, Equal, $3) }

   | expr AND expr                  { Binop($1, And, $3) }
   | expr OR expr                  { Binop($1, Or, $3) }
   | NOT expr                    { Uniop(Not, $2) }

   | lvalue                   { $1 }

   | lvalue ASSIGN expr              { Assign($1, $3) }

   | LPAREN expr RPAREN              { $2 }
   | ID LPAREN arg_list RPAREN          { Call($1, List.rev $3) }
        | LBRACK obj_list RBRACK        {ObjValue(List.rev $2)}

obj_list:
        /*nothing*/                              { [] }
        | expr                                          { [$1] }
        |obj_list COMMA expr              { $3::$1}

literal:
   INT                        { IntLit($1) }
   | FLOAT                        { FloatLit($1) }
   | STRING                        { StringLit($1) }
   | BOOL                        { BoolLit($1) }


lvalue:
   ID                        { Id($1) }
   | ID DOT ID                   { Binop(Id($1), Dot, Id($3)) }
   | expr LBRACK expr RBRACK          { Binop($1, Child, $3) }

arg_list:
   /* nothing */                 { [] }
   | expr                     { [$1] }
   | arg_list COMMA expr              { $3::$1 }
```

8.5 sast.ml

```
(*Author: Mengfei Ren*)
open Type


type expr_c =
    Literal of literal (* 42 *)
  | Id of string (* foo *)
  | Binop of expr * op * expr (* a + b *)
  | Assign of expr * expr (* a = b *)
  | Call of string * (expr list) (* foo(1, 25) *)
  | Noexpr (* While() *)
  | Uniop of op * expr
  | ObjValue of (expr list)

and expr = expr_c * t

type event_def = {
            typename: string;
            members : (t * string * (expr option)) list;
}

type stmt =
     Block of (stmt list) (* statement list and var list *)
   | Expr of expr
   | Return of expr
   | If of expr * stmt * stmt (* if (foo == 42) {} else {} *)
   | For of expr * expr * expr * stmt (* for loop *)
   | While of expr * stmt
   | Vardecl of (t * string * (expr option)) list
   | Empty

(*type func_decl = {
            return_type : t;
    fname : string;
    params : (t * string) list;
    body : stmt list;
}

type program = {
                        eventdef: event_def list;
                        globalvar: (t * string * (expr option)) list;
                        funcdef: func_decl list;
}*)
```

8.6 semantic.ml

```
(*Author: Mengfei Ren, Chang Hu*)
open Type
open Ast
open Sast

module StringMap = Map.Make(String)
(*============ symbol table ============*)
type event_table = {
        type_name : string;
        member_list : (t * string) list
}

type symbol_table = {
        parent : symbol_table option;
        mutable vars : (t * string) list;
        mutable funcs : (t * string * (t list)) list;
        mutable events : event_table list;
        is_loop : bool (* true means this is a loop scope, false otherwise *)
}

(*============== Checking Functions ===============*)
(*string of, used for print out error*)
let string_of_type = function
        | Int -> "integer"
        | Float -> "float"
        | String -> "string"
        | Boolean -> "boolean"
        | Event_type tname -> ("Event(" ^ tname ^ ")")
        | Void -> "void"
        | Calendar -> "Calendar"

let string_of_op = function
          Add -> "+"
        | Sub -> "-"
        | Mult -> "*"
        | Div -> "/"
        | Equal -> "=="
        | Neq -> "!="
        | Less_than -> "<"
        | Leq -> "<="
        | Greater_than -> ">"
        | Geq -> ">="
        | Or -> "||"
        | And -> "&&"
        | Not -> "!"
        | Dot -> "."
        | Child -> "[]"

(* operand type error *)
let e_op_type op t =
        raise (Failure ("The operand of operator " ^
                (string_of_op op) ^ " can not be " ^ (string_of_type t)))

let is_same_type t1 t2 =
        match (t1, t2) with
                | (Calendar, Event_type tt2)-> true(*TODO, which eventtype does calendar contains? 
how to do it???*)
                | (t1, t2) -> t1 = t2

(* split declaration list into single var declaration. e.g.  int a,b=0;=> int a; int b=0;*)
let partition decl =
                let t = fst decl in
                        if t = Void then raise (Failure ("variables cannot be declared with type 
void"))
                        else
                                List.rev (List.fold_left (fun l -> function
                                        Ast.WithInit (name, e) -> (t, name, Some(e))::l
                                        | Ast.WithoutInit name -> (t, name, None)::l) [] (snd decl))

(*find function by name, functions are all in global environment symbol table*)
let rec find_function fname env =
        let funcs = env.funcs in
                try
                        let _ = List.find (fun (_, n, _) -> n = fname) funcs in true
                with Not_found ->
                        match env.parent with (*find event in parent symbol table*)
                                Some(parent) ->find_function fname parent
                                | _ -> false

let rec get_function fname env =
        let funcs = env.funcs in
                try
                        List.find (fun (_, n, _) -> n = fname) funcs

                with Not_found ->
                        match env.parent with (*find event in parent symbol table*)
                                Some(parent) -> get_function fname parent
                                | _ -> raise(Failure("The funtion "^fname ^" is not declared"))


(*find variable by name*)
let rec find_variable vname env =
        try
                let _ = List.find (fun (_, n) -> n = vname) env.vars in true
        with Not_found ->
                match env.parent with (*find event in parent symbol table*)
                                Some(parent) ->find_variable vname parent
                                | _ -> false

let rec get_variable vname env =
        try
                List.find (fun (_, n) -> n = vname) env.vars
        with Not_found ->
                match env.parent with (*find event in parent symbol table*)
                                Some(parent) ->get_variable vname parent
                                | _ -> raise(Failure("The variable "^ vname^" is not declared"))


(*find event type by name*)
let rec find_event ename env =
        try
                let _ = List.find (fun event -> event.type_name = ename) env.events in true
        with Not_found ->
                match env.parent with (*find event in parent symbol table*)
```

```
                                Some(parent) ->find_event ename parent
                                | _ -> false

let rec get_event ename env =
        try
                List.find (fun event -> event.type_name = ename) env.events
        with Not_found ->
                match env.parent with (*find event in parent symbol table*)
                                Some(parent) ->get_event ename parent
                                | _ -> raise (Failure ("Event model " ^ ename ^ " is not found"))

(*================ checking ================*)
let rec check_expr env t = function
        | Ast.Assign (e1, e2) ->
                let et1 = check_expr env t e1 and et2 = check_expr env t e2 in
                        let (_, t1) = et1 and (_, t2) = et2 in
                                if (is_same_type t1 t2) then
                            Sast.Assign(et1, et2), t1
                                else
                        raise (Failure ("type mismatch in assignment"))
        | Ast.Uniop (un_op, e) ->
                let et = check_expr env t e in
                        let t1 = snd et in
                                let tt = match un_op with
                                        | Not -> if t1 = Boolean then Boolean else
                                                raise (Failure ("Only boolean is allowed for
boolean operators"))
                                       | _ -> raise (Failure ("The operator " ^(string_of_op un_op)
^ " is not unary"))
                                in
                                        Sast.Uniop(un_op, et), tt
        | Ast.ObjValue(elist) -> (match t with
                Event_type ename -> let e = get_event ename env in
                        let len1 = List.length e.member_list and len2 = List.length elist in
                                if (len1 = len2) then Sast.Noexpr, t (*TODO: each attribute
checking*)
                                else raise(Failure("Event attributes inconsistent"))
                | Calendar ->  let e = check_expr env t (List.nth elist 0) in (*TODO: all events should
be same type*)
                        let tt = snd e in
                        (match tt with
                                Event_type ename -> Sast.Noexpr, tt
                                |_ -> raise(Failure("Calendar cannot contain "^ string_of_type tt))
                        )
                |_ -> raise(Failure("List is not valided for type "^ string_of_type t))        )
        | Ast.Call (func_name, params) ->
                let fc = get_function func_name env in
                        let(ftype,_,required_param_types) = fc in
                                let typed_params = List.map (check_expr env t) params in
                                        let param_types = List.map (fun et -> snd et)
typed_params in
                                                let _ = match func_name with
                                                        | "print" -> (List.iter (function
                                                                | Event_type _ ->
                                                                        raise (Failure ("print
function couldn't print event_type"))
                                                                | Calendar ->
                                                                        raise (Failure ("print
function couldn't print calendar"))
                                                                | _ -> ()) param_types)
```

```
                                                    | "printCalendar" -> List.iter (function
                                                            | Calendar -> ()
                                                            | _ ->
                                                                    raise            (Failure
("printCalendar function can only print Calendar type"))) param_types
                                                    |    _    ->    if    param_types    <>
required_param_types then
                                                            raise      (Failure      ("function
parameter types don't match"))
                                            in
                                            Sast.Call(func_name,typed_params),ftype
        | Ast.Noexpr -> Sast.Noexpr, Type.Void
        | Ast.Binop (e1, bin_op, e2) ->
                let et1 = check_expr env t e1 in
                        if bin_op = Dot then
                                match (snd et1) with
                                | Event_type ename ->
                                        let event = get_event ename env in (*type=event_table *)
                                        let  event_scope  =  {  parent  =  None;  vars  =
event.member_list;funcs=[];events=[];is_loop=false}
                                        in
                                        let et2 = check_expr event_scope (snd et1) e2 in
                                        Sast.Binop(et1, bin_op, et2), (snd et2)
                                | _ -> raise (Failure ("left operand of . should be an event type"))
                        else if bin_op = Child then
                                (match (snd et1) with
                                | Calendar -> let et2 =check_expr env (snd et1) e2 in
                                        (match (snd et2) with
                                        Int    ->    Sast.Binop(et1,    bin_op,    et2),    snd
et1(*TODO: calendar bug*)
                                        | _ -> raise(Failure ("expression inside [] should
be int")))
                                | _ -> raise (Failure ("left operand of [] should be a calendar")))
                        else
                                let et2 = check_expr env t e2 in
                                        let (_, t1) = et1 and (_, t2) = et2 in
                                        if not (is_same_type t1 t2) then
                                                raise  (Failure  ("Type  mismatch  for
operator " ^(string_of_op bin_op) ^ ": left: " ^ (string_of_type t1) ^ ", right: " ^ (string_of_type t2)))
                                        else  (match  bin_op  with  (*  check  operand  type
for different operators *)
                                                | Add -> (match t1 with
                                                        | Int | Float | String ->
t1
                                                        |    _    ->    e_op_type
bin_op t1)
                                                | Sub | Mult | Div -> (match t1 with
                                                        | Int | Float -> t1
                                                        |    _    ->    e_op_type
bin_op t1)
                                                | Equal | Neq -> (match t1 with
                                                        |  Int  |  Float  |  String  |
Boolean -> Boolean
                                                        |    _    ->    e_op_type
bin_op t1)
                                                | Less_than | Leq | Greater_than | Geq ->
(match t1 with
                                                        | Int | Float -> Boolean
                                                        |    _    ->    e_op_type
bin_op t1)
```

```
                                                  | And | Or -> (if t1 = Boolean then
Boolean else
                                                  raise (Failure ("Only boolean is allowed
for boolean operators")))
                                                  | _ -> e_op_type bin_op t1);
                                             Sast.Binop(et1, bin_op, et2), t1

     | Ast.Literal lit ->
          let lit_type = match lit with
               | IntLit i -> Int
               | FloatLit f -> Float
               | BoolLit b -> Boolean
               | StringLit s -> String
          in
          Sast.Literal(lit), lit_type
     | Ast.Id id ->
          try
               let (var_type, _) = get_variable id env in
                    Sast.Id(id), var_type
          with Not_found ->
               raise (Failure ("undeclared identifier " ^ id))
     |_-> raise(Failure("Expr not catch"))

and check_init env t = function
     Some e -> let e = (check_expr env t e) in
          let _,t1 = e in
          if t = Calendar then
               Some(e)(*TODO: not check calendar*)
          else if t<>t1 then
               raise(Failure ("Left is " ^(string_of_type t)^ ", while right is
"^(string_of_type t1) ))
          else Some(e)
     | None -> None

and check_vdecl vars env =
     let _ = match (fst vars) with
          Event_type ename -> if(not (find_event ename env)) then raise(Failure("Event
"^ename^" is not exist"))
          |_->()
     in
     let partv = partition vars in
     let vmem = List.map (fun (t, name, init) ->
          try
               let _ =
                    List.find (fun (_, n) -> n = name) env.vars
               in raise (Failure ("Variable " ^ name ^ " is already exist in the current
block"))
          with Not_found ->
               (t, name, check_init env t init)) partv
     in (*update symbol table*)
     env.vars <- (env.vars @ List.map (fun (t, name, _) -> (t, name)) vmem);
     Sast.Vardecl(vmem)

let rec check_stmt env = function
     Ast.Block(b) ->
          let block_scope = { env with
               parent = Some(env); vars = []}
          in
          Sast.Block(check_stmtlist b block_scope)
     | Ast.Expr e -> Sast.Expr(check_expr env Type.Void e)
```

```
        | Ast.Return e -> Sast.Return(check_expr env Type.Void e)
        | Ast.ReturnVoid -> Sast.Return(Sast.Noexpr,Type.Void)
        | Ast.If(e, s1, s2) ->
                let e = check_expr env Type.Void e in
                if ((snd e) = Boolean) then
                        let st1 = check_stmt env s1 in
                                (* create a fake block for if statement to guarantee a new scope *)
                                let new_st1 = match st1 with
                                        | Sast.Block(_) -> st1
                                        | _ -> check_stmt env (Ast.Block [s1])
                                in
                                let st2 = check_stmt env s2 in
                                (* create a fake block for else clause to guarantee a new scope *)
                                let new_st2 = match st2 with
                                        | Sast.Block(_) -> st2
                                        | _ -> check_stmt env (Ast.Block [s2])
                                in
                                Sast.If(e, new_st1, new_st2)
                        else
                                raise (Failure ("the expression in if statement is not boolean"))

        | Ast.For(e1, e2, e3, s) ->
                let e2 = check_expr env Type.Void e2 in
                if ((snd e2) = Boolean) then
                        let loop_scope = { env with is_loop = true } in

                        let block_s = match s with
                                        | Ast.Block _ -> s
                                        | _ -> Ast.Block [s]
                                in
                                let st = check_stmt loop_scope block_s in
                                        Sast.For (check_expr env Type.Void e1, e2, check_expr
env Type.Void e3, st)
                else
                        raise (Failure ("the second expression in for statement is not boolean"))

        | Ast.While(e, s) ->
                let e = check_expr env Type.Void e in
                if ((snd e) = Boolean) then
                        let loop_scope = { env with is_loop = true } in

                                let block_s = match s with
                                        | Ast.Block _ -> s
                                        | _ -> Ast.Block [s]
                                in
                                let st = check_stmt loop_scope block_s in
                                Sast.While (e, st)
                        else
                                raise (Failure ("the expression in while statement is not boolean"))

        | Ast.Vardecl(v) ->(check_vdecl v env)
        | Ast.Empty -> Sast.Empty
and check_stmtlist slist env =
        match slist with
        [] -> []
        | head :: tail -> check_stmt env head :: check_stmtlist tail env



let check_event event env =
        let ename = event.Ast.typename in
                try
                        let _ = List.find (fun e -> e.type_name = ename) env.events
                                in raise (Failure ("Event model " ^ ename ^ " is already exist"))
                with Not_found ->
                        let emem =
                                List.concat (List.map partition event.Ast.members)
                        in
                                let et = {
                                        type_name = ename;
                                        member_list = List.map (fun (t, name, _) -> (t, name))
emem
                                } in
                                 env.events <- et::env.events;
                                (*Sast.Event.members*)
                                        let new_tmem =
                                                List.map (fun (t,name,init)->
                                                let _ = match t with (* recursive tree type in
members *)
                                                        Event_type ename->
raise(Failure("Cannot declare nested event type"))
                                                        |Calendar -> raise(Failure("Cannot
declare Calendar type"))
                                                        | _ -> ()
                                                in ( t, name, check_init env t init)) emem
                                        (*return Sast*)
                                        in {Sast.typename = ename; Sast.members= new_tmem}



let check_func func env =
        let ft = func.return_type
                and fn = func.fname
                and fp = func.params
        in
        if (find_function fn env) then
                raise (Failure ("function" ^ fn ^ "redeclared"))
        else (* function is effective *)
                (List.iter (fun (t, name) ->
                        if (t = Type.Void) then
                                raise (Failure ("parameter" ^ name ^ "cannot be void"))
                        else
                                let num =
                                        List.fold_left (fun i (_, n) ->
                                                if (n = name) then i + 1 else i) 0 fp
                                in if (num > 1) then
                                        raise (Failure ("duplicate parameter name" ^ name))
                )fp);
                let required_param_types = (*get type info of params*)
                        List.map (fun p -> fst p) fp
                in
                let new_scope = { env with funcs = (*add function to globe_scope*)
                        (ft, fn, required_param_types)::env.funcs
                } in
                let param_scope = { new_scope with (*add params to function scope*)
                        parent = Some(new_scope); vars = fp}
                in
                let new_body =
                        check_stmtlist func.body param_scope
                in
                let required_param_types = (* only type info of the params *)
                                                List.map (fun p -> fst p) fp
                in env.funcs <- (ft,fn,required_param_types)::env.funcs


let rec check_eventlist elist env =
        match elist with
        [] -> []
        | head :: tail -> check_event head env :: check_eventlist tail env

let rec check_vdecllist vlist env =
        match vlist with
        [] -> []
        | head :: tail -> let _ = check_vdecl head env in check_vdecllist tail env



let rec check_funclist flist env =
        match flist with
        [] -> []
        | head :: tail -> check_func head env :: check_funclist tail env

let check_main env =
        if (not (find_function "main" env)) then raise(Failure("function main() is not found"))

let check (p:program) =
        let global_env = {
                parent = None;
                vars = [];
                funcs = [(Void, "print", [String]);(Void,"printCalendar",[Calendar])]; (* built-in
functions *)
                events = [];
                is_loop = false;
        } in
        (*use global_env*)
        let _ = check_eventlist p.eventdef global_env in
                let _ = check_vdecllist p.globalvar global_env in
                        let _ = check_funclist p.funcdef global_env in
                                let _ = check_main global_env
                                        in true

                                        (*{Sast.eventdef = events; Sast.globalvar = variables;
Sast.funcdef = functions; print_endline "\nSemantic analysis completed successfully.\n";}
                                else *)
```

8.7 compile.ml

```
(*Author: Yu Kang*)
open Type
open Ast

let string_of_call (f, el) =
        match f with
                "print" -> "System.out.println("^(String.concat "," (List.map string_of_expr el))^")"
                |_   -> f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"

let rec string_of_expr = function
  Literal(l) -> Ast.string_of_literal l
  | Id(s) -> s
  | Assign(e1,e2)->string_of_expr e1 ^" = "^string_of_expr e2
  | Call(f, el) -> string_of_call (f, el) (*(match f with
                                   print -> print_endline "print"
                                   | _ -> print_endline "other"); *)
  | Noexpr (* While() *)-> ""
  | Uniop(o,e)->Ast.string_of_op o ^" "^string_of_expr e
  | ObjValue(l)->"[" ^ (String.concat ", " (List.map string_of_expr l)) ^" ]"
  | Binop(e1,o,e2)-> (match o with
                                                 Child -> string_of_expr e1 ^ "[" ^ string_of_expr e2 ^ "]"
                                                |Dot     ->  string_of_expr  e1  ^  Ast.string_of_op  o  ^
string_of_expr e2
                                                | _    -> string_of_expr e1 ^ " " ^ Ast.string_of_op o ^ " " ^
string_of_expr e2)


let string_of_init = function
        WithInit(v,e)-> v ^ "=" ^ string_of_expr e
        |WithoutInit(v)-> v

let string_of_initlist (init) =
        (String.concat ", " (List.map string_of_init init))^";\n"

let string_of_event_init myEvent = function
        WithInit(v,e)->  v  ^  "=  new  "^  myEvent  ^  Str.global_replace(Str.regexp  "\\]")  ")"
(Str.global_replace(Str.regexp "\\[") "(" (string_of_expr e))
        |WithoutInit(v)-> v

let string_of_event_initlist myEvent (init) =
        (String.concat ", " (List.map (string_of_event_init myEvent) init))^";\n"

let string_of_vdecl (t, l) =
        match t with
         Calendar -> string_of_initlist l
        |Event_type(myEvent) -> myEvent ^" "^ string_of_event_initlist myEvent l^"\n"
        | _ -> Ast.string_of_dt t^" "^ string_of_initlist l^"\n"



let string_of_mem (t, l) =
        "this."^(Str.global_replace (Str.regexp ";\n") "" (string_of_initlist l)) ^" = "^ (string_of_initlist
l)

let string_of_event (myevent) =
        "class    "^myevent.typename^"{\n"^    String.concat    "\n"    (List.map    string_of_vdecl
myevent.members)^
        "public " ^ myevent.typename ^"("^ Str.global_replace (Str.regexp ";\n\n") "" (String.concat
"," (List.map string_of_vdecl myevent.members))^ "){\n"^
        String.concat "" (List.map string_of_mem myevent.members)^
        "}\n}\n"

let string_of_arg(t, name) =
        (Ast.string_of_dt t) ^ " " ^ name

let rec string_of_stmt = function
        Block(stmts) -> "{\n" ^ String.concat "\n" (List.map string_of_stmt stmts) ^ "}\n"
   | Expr(e) -> string_of_expr e ^";"
   | Return(e) ->"return "^ (string_of_expr e) ^";"
   | ReturnVoid ->"return ;"
   | If(e, s, Block([]))->"if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1,e2,e3,s)->"for (" ^ string_of_expr e1^ " ; " ^ string_of_expr e2 ^ " ; " ^ string_of_expr e3  ^
")\n " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
  | Vardecl(v)-> string_of_vdecl v
  | Empty->""

let string_of_func (f) =
        match f.fname with
        "main" -> "public static "^(Ast.string_of_dt f.return_type)^" main(String[] args)"^
        "{\n" ^ (String.concat "" (List.map string_of_stmt f.body) )^ "}\n"
        |_     -> "public static "^(Ast.string_of_dt f.return_type)^" "^ f.fname ^ " (" ^ String.concat ", "
(List.map string_of_arg f.params)^")"^
        "{\n" ^ (String.concat "" (List.map string_of_stmt f.body) )^ "}\n"

let string_of_events = function
        [] -> ""
        | elist -> String.concat "\n" (List.map string_of_event elist)

let string_of_vars_g = function
   [] -> ""
        | v -> "static " ^ String.concat "\n static" (List.map string_of_vdecl v)


let string_of_funs = function
        [] -> ""
        | flist -> String.concat "\n" (List.map string_of_func flist)

let string_of_program (p:program) prog_name =
        let out_chan = open_out (prog_name ^ ".java") in
                let translated_prog =
                        "import java.util.ArrayList;\n\n" ^
                        string_of_events p.eventdef ^"public class " ^
                        prog_name ^"{\n"^ string_of_vars_g p.globalvar ^
                        string_of_funs p.funcdef^"\n}" in

                        let proc_status = ignore(Printf.fprintf out_chan "%s" translated_prog);
                                close_out out_chan;
                                print_string translated_prog;
                                Sys.command (Printf.sprintf "javac %s.java" prog_name) in
                                        match proc_status with
                                          0 -> "\n\nCompilation successful\n"
                                        | _ -> "\nCompilation unsuccessful!\n"
```

8.8 makefile

```
(*Author: Yu Kang*)
OBJS = ast.cmo parser.cmo scanner.cmo compile.cmo semantic.cmo ica.cmo

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

.PHONY : test
test : lsystem testall.sh
        ./testall.sh
```

```
ast.cmo : type.mli ast.ml
	ocamlc -c type.mli
	ocamlc -c ast.ml

scanner.cmo : scanner.ml
	ocamlc -c scanner.ml

scanner.ml : scanner.mll
	ocamllex scanner.mll

parser.cmo : parser.ml
	ocamlc -c parser.ml

parser.ml : parser.mli
	ocamlc -c parser.mli

parser.mli : parser.mly ast.cmo
	ocamlyacc parser.mly

compile.cmo: compile.ml
	ocamlc -c compile.ml

ica.cmo: ica.ml ast.cmo compile.cmo semantic.cmo
	ocamlc -c ica.ml

semantic.cmo: type.cmi semantic.ml ast.cmo sast.cmi
	ocamlc -c semantic.ml

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

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

.PHONY : clean
clean :
	rm -f ica parser.ml parser.mli scanner.ml \
	*.cmo *.cmi  *.java *.class
```

References

[1] B. W. Kernighan and D. Ritchie. The C Programming Language, Second Edition. Prentice-Hall, 1988.