# OpenSC – A Smart Contract Language Reference Manual

*Jun Sha* `js5506`
*Linghan Kong* `lk2811`
*Ruibin Ma* `rm3708`
*Rahul Sehrawat* `rs3688`
*Chong Hu* `ch3467`

`https://github.com/JackSnowWolf/OpenSC/tree/master`
May, 2020

**Introduction**

OpenSC is a functional programming language which has similar functionality compared to `Scilla` and `Pact`. It is statically typed and will support several features. It is a high-level language that will be primarily used to implement smart contracts, which are programs that provide protocol for handling account behavior in Ethereum.

Compared to other languages, we model contracts as some simple transition systems, with the transitions being pure functions of the contract state. These functions are expressed from one state to another state in a list of storage mutations.

Inspired by the MiniC language, part of the DeepSEA compiler, we aim to develop a language which allows interactive formal verification of smart contracts with security guarantees. From a specific input program, the compiler generates executable bytecode, as well as a model of the program that can be loaded into the `Coq` proof assistant. Our eventual goal is that smart contracts like storage, auction and token can be written by OpenSC, and that these contracts can be compiled via the translator into binary codes that can be executed on EVM.

## Contents

# 1 Using the Compiler

## 1.1 Installation

1. Environment Dependencies

   - Install `ocaml`, which is what our translator is written in
   - Install `opam`, the ocaml package manager
   - `opam install cryptokit` which is used in the `Minic` (IR code) generation phase of the compiler front-end for cryptographic hashing

→ Appendix. B

2. Folder structure

   - `opensc.ml` is the top-level program of the compiler
   - A `Makefile` is included to automate the compilation of the compiler

```
opensc
├── doc
├── src
│   ├── backend
│   ├── opensc.ml -> B.6
│   ├── ast.ml
│   ├── sast.ml
│   ├── parser.mly -> B.1
│   ├── scanner.mly -> B.2
│   ├── semant.ml -> B.3
│   ├── translateMinic.ml -> B.4
│   ├── test.ml
│   ├── test2.ml
│   ├── test3.ml
│   ├── test4.ml
│   └── Makefile -> B.5
├── testcase -> C
│   ├── testcase_01
│   ├── testcase_02
│   ├── """
│   └── testcase_43
└── README.md
```

## 1.2 Synopsis

```
# at root directory
cd src
Make
./opensc.native [source.sc] [mode]
```

## 1.3 Modes

`ast`

- generate raw AST and print its structure

`sast`

- generate SAST (semantically checked AST) and print its structure

`minic`

- generate minic AST (the IR code) and print its structure

`bytecode`

- generate EVM bytecode and print it

## 1.4 Running tests and examples

The directory `testcase` contains a bunch of small programs which test particular features of `OpenSC`.

The directory `example_contract` contains a few example contracts which are also explained in the next section of this manual.

→ Sec. 1.2

If you just want to try compiling some programs, all you need is to run the commands specified in the Synopsis Sec. 1.2 part above. If you want to run the test cases or the examples yourself, just let the compiler take a test case or an example as the input source.

# 2 Grammar Details

We now begin a more formal description of the language.

A source file consists of a number of comments, a signature, a constructor and a number of method definitions.

## 2.1 Lexical Conventions

### 2.1.1 Comments

```
1 "/-"  "-/" (* Comments *)
```

### 2.1.2 Whitespaces

```
1 [' ' '\t' '\r' '\n'] (* Whitespaces *)
```

### 2.1.3 Identifiers

Identifiers should start with a letter and then be followed by letters and/or digits and/or underscores.

```
1 (* Identifiers *)
2 | letter (digits | letter | '_')* as lem { ID(lem) }
```

### 2.1.4 Separators

```
(* Separators *)
| '('                       { LPAREN }
| ')'                       { RPAREN }
| '{'                       { LBRACE }
| '}'                       { RBRACE }
| '['                       { LBRACK }
| ']'                       { RBRACK }
| ':'                       { COLON } (* Type declaration *)
(* end of type of assignments *)
| '.'                       { POINT } (* Point for extract information *)
| ';'                       { SEMI }
| ','                       { COMMA }
```

### 2.1.5 Operators

```
(* Operators *)
| "=="                      { EQ }
| "!="                      { NEQ }
| ">"                       { LGT }
| ">="                      { LGTEQ }
| "<="                      { RGTEQ }
| "<"                       { RGT}
| "+"                       { ADD }
| "-"                       { SUB }
| "*"                       { MUL }
| "/"                       { DIVIDE }
| "and"                      { AND }
| "or"                      { OR }
| "=>"                      { MAPASSIGN }
| '='                       { ASSIGN }
```

### 2.1.6 Reserved Keywords

The following are reserved keywords so they are not allowed to be used as variable names. and or UInt True False Bool Address map voidlit void signature storage event of method constructor Env guard effects logs returns int

## 2.2 Data Types

Below is our type definitions as well as keywords of allowed data types in an OpenSC program:

```
(* data type *)
type typ =
```

```
3         | Bool (* "Bool" *)
4         | Int (* "int" *)
5         | Uint of string (* "UInt" *)
6         | Address of string (* "Address" *)
7         | Void of string (* "voidlit" *)
8         | Mapstruct of typ list * typ (* "map" *)
```

## 2.3 Expressions

Below is our type definition of expression, which is basically generated with a type constructor and a literal:

```
1  (* expression *)
2  type expr =
3         | NumLit of int  (* number literal, such as 11 *)
4         | BoolLit of bool (* True, False *)
5         | StrLit of string (* "hello world" *)
6         | Id of string (* letter (digits | letter | '_')* *)
7         | EnvLit of string * string (* Env.sender *)
8         | Mapexpr of expr * expr list (* balances[Env.sender] *)
9         | Binop of expr * op * expr (* x + 11 *)
10        | Logexpr of expr * expr list (* logs Transfer (Env.sender, a, v) *)
11        | Storageassign of expr * expr  (* beneficiary  |-> a *)
12        | Comparsion of expr * op * expr (* Env.value > lead *)
13        | Voidlit of string (* voidlit *)
```

## 2.4 Signature

An OpenSC program consists of a signature with its methods implementations. A signature basically consists of global variables declarations, map assignment declarations, event declarations, and a constructor declaration and methods declarations with the argument types and return type of each method.

```
1  /- interface -/
2
3  signature TOKEN{
4
5    storage supply : UInt;
6
7    map balances : (Address) => UInt;
8    map allowances : (Address, Address) => UInt;
9
10   event Transfer = Transfer of (Address, Address
         , UInt);
11   event Approval = Approval of (Address, Address
         , UInt);
12
13   constructor c : (UInt) -> void;
14   method totalSupply : (void) -> UInt;
15   method balanceOf : (Address) -> UInt;
16   method transfer : (Address, UInt) -> Bool;
```

```
17    method transferFrom : (Address, Address, UInt)
          -> Bool;
18    method approve : (Address, UInt) -> Bool;
19    method allowance : (Address, Address) -> UInt;
20  }
```

## 2.5 Constructor

Although constructor is not supported to translate into Minic AST and EVM bytecode, users could still implement the constructor with name, storage section and returns. Below is an example to implement a simple constructor

```
1  constructor c (s : UInt){
2    storage
3      supply                |-> s;
4      balances[Env.sender]  |-> s;
5    returns void;
6  }
```

## 2.6 Method Definitions

### 2.6.1 Params

In our parser, we parse the parameters as a var list. The var structure is an Id * type. Below is the example of how parameters get parsed:

```
1
2  source program:
3  method transfer (a : Address, v : UInt){
4
5  AST:
6  method transfer(Var(a: address(ADDRESS)), Var(v:
       uint(uint)))
```

### 2.6.2 Guardbody

A guard body is a list of expressions and basically involved with comparison expression. Below is an example of source guard body and parsed guard body:

```
1  source program:
2  method transfer (a : Address, v : UInt){
3
4    guard{
5      Env.value == 0;
6      balances[Env.sender] >= v;
7      /- overflow checking -/
8      balances[a] > balances[a] - v;
9      balances[Env.sender] > balances[Env.sender]
           + v;
10   }
11
12 /- AST:-/
13 guard
```

```
14    Comparsion: EnvLit(Envvalue) == NumLit(0)
15    Comparsion: Mapexpr(balances elements:EnvLit(
          Envsender)) >= v
16    Comparsion: Mapexpr(balances elements:a) >
          Binop(Mapexpr(balances elements:a) - v)
17    Comparsion: Mapexpr(balances elements:EnvLit(
          Envsender)) > Binop(Mapexpr(balances
          elements:EnvLit(Envsender)) + v)
```

### 2.6.3 Storagebody

A storage body is also a list of expression and it basically is a list of storage assign. Below is an example of source storage body and AST of store the body:

```
1  source program:
2   storage{
3      balances[Env.sender] |-> balances[Env.sender
           ] - v;
4      balances[a]          |-> (balances[a] + v);
5    }
6  /- AST:-/
7  StorageAssign: Mapexpr(balances elements:EnvLit(
       Envsender)) PASSIGN: |->Binop(Mapexpr(
       balances elements:EnvLit(Envsender)) - v))
8  StorageAssign: Mapexpr(balances elements:a)
       PASSIGN: |->Binop(Mapexpr(balances elements:
       a) + v))
```

### 2.6.4 Event body

Ethereum events are not implemented yet. However, we are able to parse the event and translate into AST.

```
1  source program:
2    effects{
3      logs Transfer (Env.sender, a, v);
4    }
5
6  /- AST:-/
7  effects
8    Logexpr( Transfer EnvLit(Envsender) a v)
```

### 2.6.5 Return value

For returns, it just a simple return expr in our program. Below is an example:

```
1  source program:
2  returns True;
3  /- AST:-/
4  returns BoolLit(true)
```

## 3 OpenSC by examples

In this part, three examples smart contracts are given. They are basic types of simple smart contracts widely used in blockchain. `Simplestorage` (See Appendix. A.1 ) is a simple storage contract program used to describe the process of storing data; `Auction` (See Appendix. A.2 ) is an open auction contract program for people sending their bids where the auction is ended with the highest bid sent to the beneficiary; `Token` (See Appendix. A.3 )is a token implementation program to transfer tokens, as well as allow tokens to be approved

→ Appendix. A.1
→ Appendix. A.2
→ Appendix. A.3

## 4 Acknowledgements

Thanks to Professor Ronghui Gu, the instructor of our course, who brought us to the PLT world and let us realize the charm of functional programming and formal verification, both of which are what our project is based on.

Thanks to River Dillon Keefer and Amanda Liu, TAs of our course, who introduced the DeepSEA project to us and provided very inspiring and helpful ideas on the OpenSC language syntax among other project details.

Thanks to Vilhelm Sjöberg, our project advisor, researcher at Yale and the primary creator of the DeepSEA project, who provided us with great information on everything about the DeepSEA project, and answered our many questions, which has been super helpful.

## References

[1] Language `Scilla`: `https://scilla.readthedocs.io/en/latest/`

[2] Language `pact`: `https://github.com/kadena-io/pact`

[3] Language `DeepSEA`: `https://certik.io/blog/technology/an-introduction-to-deepsea`

# Appendix

## A Examples

### A.1 simpleStorage.sc

```
/- A simple storage program -/

signature SimpleStorage {
    storage storedData : int;

  constructor c : (void) -> void;
  method set : (int) -> void;
}

constructor c (){
  storage
  returns void;
}

method set(x: int) {
  guard{
    x > 0 ;
  }
  storage{
      storedData |-> x;
  }
  effects{}
  returns voidlit;
}
```

### A.2 auction.sc

```
/- simple open auction -/
/- https://solidity.readthedocs.io/en/v0.4.24/solidity-by-example.html#
    simple-open-auction -/


signature AUCTION{
  /- parameters -/
  storage beneficiary : Address;
  storage end : UInt;

  /- current state of the auction -/
  storage leader : Address;  /- leading bidder; -/
  storage lead : UInt;       /- highest bid; -/

  /- allowed withdrawals of previous bids -/
  map withdrawals : (Address) => UInt;

  /- events -/
  event HighestBidIncreased = HighestBidIncreased of (Address, UInt);
  event AuctionEnded        = AuctionEnded  of (Address, UInt);

  /- methods -/
  constructor c : (UInt, Address) -> void;
  method bid : (void) -> void;
  method withdraw : (void) -> void;
  method terminate : (void) -> void;
```

```
26
27  }
28
29
30  constructor c (t : UInt, a : Address){
31    storage
32      end          |-> (Env.now) + t;
33      beneficiary  |-> a;
34    returns void;
35  }
36
37
38  method bid (){
39    guard{
40      Env.now    <= end;
41      Env.value  >  lead;
42      withdrawals[leader] >= withdrawals[leader] - lead;
43    }
44    storage{
45      withdrawals[leader]  |-> withdrawals[leader] + lead;
46      leader               |-> Env.sender;
47      lead                 |-> Env.value;
48    }
49    effects{
50      logs HighestBidIncreased (Env.sender, Env.value);
51    }
52    returns voidlit;
53  }
54
55  method withdraw (){
56    guard{
57      withdrawals[Env.sender]  != 0;
58    }
59    storage{
60      withdrawals[Env.sender]  |-> 0;
61    }
62    effects{
63      /- sends Env.sender withdrawals[Env.sender]; -/
64    }
65    returns voidlit;
66  }
67
68  method terminate (){
69    guard{
70      Env.now  >= end;
71    }
72    storage{}
73    effects{
74      logs AuctionEnded (leader, lead);
75    }
76    returns voidlit;
77  }
```

## A.3 token.sc

```
1  /-
2    token implementation satisfying the ERC20 standard:
3     https://eips.ethereum.org/EIPS/eip-20
4
```

```
5     interface
6  -/
7
8  signature TOKEN{
9
10   storage supply : UInt;
11
12   map balances : (Address) => UInt;
13   map allowances : (Address, Address) => UInt;
14
15   event Transfer = Transfer of (Address, Address, UInt);
16   event Approval = Approval of (Address, Address, UInt);
17
18   constructor c : (UInt) -> void;
19   method totalSupply : (void) -> UInt;
20   method balanceOf : (Address) -> UInt;
21   method transfer : (Address, UInt) -> Bool;
22   method transferFrom : (Address, Address, UInt) -> Bool;
23   method approve : (Address, UInt) -> Bool;
24   method allowance : (Address, Address) -> UInt;
25 }
26
27
28 /- implementation -/
29
30 constructor c (s : UInt){
31   storage
32     supply                |-> s;
33     balances[Env.sender]  |-> s;
34   returns void;
35 }
36
37 method totalSupply (){
38   guard{
39     Env.value == 0;
40   }
41   storage{}
42   effects{}
43   returns supply;
44 }
45
46 method balanceOf (a : Address){
47   guard{
48     Env.value == 0;
49   }
50   storage{}
51   effects{}
52   returns balances[a];
53   }
54
55 method allowance (owner : Address, spender : Address){
56   guard{
57     Env.value == 0;
58   }
59   storage{}
60   effects{}
61   returns allowances[spender, owner];
62 }
63
64 method transfer (a : Address, v : UInt){
```

```
65
66    guard{
67      Env.value == 0;
68      balances[Env.sender] >= v;
69      /- overflow checking -/
70      balances[a] > balances[a] - v;
71      balances[Env.sender] > balances[Env.sender] + v;
72    }
73    storage{
74      balances[Env.sender] |-> balances[Env.sender] - v;
75      balances[a]          |-> (balances[a] + v);
76    }
77    effects{
78
79      logs Transfer (Env.sender, a, v);
80    }
81    returns True;
82  }
83
84  method approve (spender : Address, v : UInt){
85
86    guard{
87      Env.value == 0;
88    }
89    storage{
90      allowances[spender, Env.sender] |-> v;
91    }
92    effects{
93      logs Approval (Env.sender, spender, v);
94    }
95    returns True;
96  }
97
98  method transferFrom (from : Address, to : Address, v : UInt){
99
100   guard{
101     Env.value == 0;
102     balances[from] >= v;
103     allowances[Env.sender, from] >= v;
104
105     /- overflow checking -/
106
107     allowances[Env.sender, from] - v < allowances[Env.sender, from];
108     balances[from] - v < balances[from];
109     balances[to] + v > balances[to];
110   }
111   storage{
112     allowances[Env.sender, from]  |-> allowances[Env.sender, from] - v;
113     balances[from]                  |-> balances[from] - v;
114     balances[to]                    |-> balances[to] + v;
115   }
116   effects{}
117   returns True;
118 }
```

# B Source Code

## B.1 Parser

```
%{ open Ast
%}

%token SIGNATURE UINTTYPE STROAGE EVENT OF METHOD CONSTRUCTOR GUARD EFFECTS LOGS RETURNS MAP UINTType STORAGE
%token ASSIGN ARROW MAPASSIGN ASSIGN COLON SEMI PASSIGN COMMA POINT
%token LBRACE RBRACE LPAREN RPAREN LBRACK RBRACK
%token EQ NEQ LGT ADD SUB MUL DIVIDE AND OR BOOL LGTEQ RGTEQ RGT
%token INT
%token <int> NUMLITERAL
%token <string> ID ADDRESSTYPE END STRLIT UINTTYPE
%token <string> UNIT ENVIRONMENT VOID
%token <bool> BooLit
%token EOF

%start program
%type <Ast.program> program



/* %left OR
%left AND

%left LT  */
%right PASSIGN
%left EQ NEQ LGT LGTEQ RGTEQ RGT
%left OR AND

%left ADD SUB
%left MUL DIVIDE

%%

program:
         defs EOF {$1}

defs:
   /* nothing */
        | interfacedecl implementationdecl { $1, $2 }


interfacedecl:
        SIGNATURE id_ok LBRACE interfaceBody_list RBRACE
        {
                {
                        signaturename = $2;
                        interfacebody =  $4
```

```
                }
        }

interfaceBody_list:
                { [] }
        |interfaceBody interfaceBody_list { $1::$2 }


interfaceBody:
        | STORAGE ID COLON type_ok SEMI {TypeAssigndecl (Id($2), $4)}
        | MAP ID COLON LPAREN type_list RPAREN MAPASSIGN type_ok SEMI{MapAssigndecl (Id($2), Mapstruct($5, $8))}
        | EVENT ID ASSIGN ID OF LPAREN type_list RPAREN SEMI {Eventdecl (Id($2), $7)}
        | CONSTRUCTOR ID COLON LPAREN type_list RPAREN ARROW type_ok SEMI{Constructordecl (Id($2), $5, $8)}
        | METHOD ID COLON LPAREN type_list RPAREN ARROW type_ok SEMI{Methodecls (Id($2), $5, $8)}


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

argument:
        | ID {Id($1)}
        | ENVIRONMENT POINT ID {EnvLit($1, $3)}

/* (owner : Address, spender : Address) */
param_list:
  /*nothing*/ { [] }
        | param { [$1] }
  | param COMMA param_list  { $1 :: $3 }


param:
  ID COLON type_ok { Var(Id($1), $3) }

id_ok:
        | ID {Id($1)}

expr_list:
  /*nothing*/ { [] }
        | expr { [$1] }
  | expr SEMI expr_list  { $1 :: $3 }

expr:
        | NUMLITERAL { NumLit($1) }
  | BooLit { BoolLit($1) }
        | ID LBRACK arg_list RBRACK {Mapexpr(Id($1), $3)}
        | ID {Id($1)}
        | VOID {Voidlit($1) }
        | ENVIRONMENT POINT ID {EnvLit($1, $3)}
```

```
97          | expr ADD expr   {Binop ($1, Add, $3) }
98          | expr SUB expr   {Binop ($1, Sub, $3) }
99          | expr MUL expr   {Binop ($1, Times, $3) }
100         | expr DIVIDE expr   {Binop ($1, Divide, $3) }
101         | expr OR expr   {Binop ($1, Or, $3) }
102         | expr AND expr   {Binop ($1, And, $3) }
103         | expr LGT expr  {Comparsion ($1, LGT, $3)}
104         | expr EQ expr  {Comparsion ($1, Equal, $3)}
105         | expr NEQ expr  {Comparsion ($1, Neq, $3)}
106         | expr RGT expr  {Comparsion ($1, RGT, $3)}
107         | expr LGTEQ expr  {Comparsion ($1, LGTEQ, $3)}
108         | expr RGTEQ expr  {Comparsion ($1, RGTEQ, $3)}
109         | expr PASSIGN expr { Storageassign($1, $3) }
110         | LPAREN expr RPAREN {$2}
111
112
113
114 type_list:
115   /*nothing*/ { [] }
116         |        type_ok {[$1]}
117         | type_ok COMMA type_list { $1 :: $3 }
118
119 type_ok:
120     INT   { Int  }
121    | UINTTYPE { Uint($1) }
122    | BOOL  { Bool }
123    | ADDRESSTYPE {Address($1)}
124    | UNIT { Void($1) }
125
126
127
128 implementationdecl:
129         constructordecl methoddecls
130         {
131                 {
132                         consturctor = $1;
133                         methods =  $2
134                 }
135         }
136
137 constructordecl:
138         CONSTRUCTOR id_ok LPAREN param_list RPAREN LBRACE STORAGE expr_list RETURNS type_ok SEMI RBRACE
139         {
140                 {
141                         name = $2;
142                         params = $4;
143                         consturctor_body = $8;
144                         return_type = $10;
145                 }
146         }
```

```
147
148 methoddecls:
149                 { [] }
150         |        methoddecl methoddecls {$1 :: $2 }
151
152 methoddecl:
153         METHOD id_ok LPAREN param_list RPAREN LBRACE
154         GUARD LBRACE expr_list RBRACE
155         STORAGE LBRACE expr_list RBRACE
156         EFFECTS LBRACE effects_bodylist RBRACE
157         RETURNS expr SEMI RBRACE
158         {
159                 {
160                         methodname = $2;
161                         params = $4;
162                         guard_body = $9;
163                         storage_body = $13;
164                         effects_body = $17;
165                         returns = $20;
166                 }
167         }
168
169
170 effects_bodylist:
171                 { [] }
172         |effects_body effects_bodylist { $1::$2 }
173
174 effects_body:
175         | LOGS id_ok LPAREN arg_list RPAREN SEMI { Logexpr($2, $4) }
```

## B.2 Scanner

```
1  {open Parser}
2
3  let digits = ['0'-'9']
4  let letter = ['a'-'z' 'A'-'Z']
5
6  rule token = parse
7     [' ' '\t' '\r' '\n']        { token lexbuf }
8    |"/-"                                                    {multicomment lexbuf} (* multiple com
9    | '('                    { LPAREN }
10         | ')'                     { RPAREN }
11         | '{'                     { LBRACE }
12         | '}'                     { RBRACE }
13         | '['                     { LBRACK }
14         | ']'                     { RBRACK }
15         (*  General op *)
16         | "=="                                          { EQ }
17         | "!="                                          { NEQ }
```

```
18          | ">"                                                                { LGT }
19          | ">="                                                           { LGTEQ }
20          | "<="                                                           { RGTEQ }
21          | "<"                                                                  { RGT}
22          | "+"                                                               { ADD }
23          | "-"                                                               { SUB }
24          | "*"                                                               { MUL }
25          | "/"                                                               { DIVIDE }
26          | "and"                                                               { AND }
27          | "or"                                                               { OR }
28          (* end of general ops *)
29          (*  Types *)
30          | "UInt"                                                  { UINTTYPE("uint") }
31          | "True"                  { BooLit(true)  }
32          | "False"                 { BooLit(false) }
33          | "Bool"                  { BOOL }
34          | "Address"                                          { ADDRESSTYPE("ADDRESS") }
35          | "map"                                                        { MAP } (* as hash table *)
36          | "voidlit"                                          {VOID("voidlit")} (* void is a literal type ... *)
37          | "void"                                                        { UNIT("void") } (* instead of () use void
38          (* end of types *)
39          (* type of assignement*)
40          | "->"                                                           { ARROW }
41          | "|->"                                                          { PASSIGN }
42          | "=>"                                                           { MAPASSIGN }
43          | '='                                                                  { ASSIGN }
44          | ':'                     { COLON } (* Type declaration *)
45          (* end of type of assignments *)
46          | '.'                                                                { POINT } (* Point for extract informa
47          | ';'                                                               { SEMI }
48          | ','                                                               { COMMA }
49          (*  ========================================================  *)
50          | "signature"                                    { SIGNATURE }
51          (* | "end"                                          { END("END") }        separation op *)
52          | "storage"                               { STORAGE }
53          | "event"                                                  { EVENT }
54          | "of"                                                        { OF }
55          | "method"                                          { METHOD }
56          | "constructor"                           { CONSTRUCTOR }
57          | "Env"                                                        { ENVIRONMENT("Env") }
58          | "guard"                                                  { GUARD }
59          | "effects"                                          { EFFECTS }
60          | "logs"                                           { LOGS }
61          | "returns"                                          { RETURNS }
62          (* NEED more type *)
63          | '"' (([^'"']*) as s) '"'  { STRLIT(s) }
64          | "int"                                          { INT }
65          | digits+ as lem  { NUMLITERAL(int_of_string lem) }
66          | letter (digits | letter | '_')* as lem { ID(lem) }
67          | eof { EOF }
```

```
68
69          and multicomment = parse
70            "-/"  { token lexbuf }
71          | _     { multicomment lexbuf }
```

## B.3 Semantic Check

```
1   open Ast
2   open Sast
3   open List
4
5   module StringMap = Map.Make(String)
6
7
8   (*
9   let strore_ids ta = function *)
10
11
12  (* need to implement *)
13  let check (signature, implementation) =
14
15    (* Add variable id in interface to symbol table *)
16    let add_var map var =
17      let dup_err v = "duplicate variable " ^ (string_of_expr v) ^ " in interface"
18      and make_err er = raise (Failure er)
19      in match var with (* No duplicate variables or redefinitions of built-ins *)
20        Var(x, t) when StringMap.mem (string_of_expr x) map -> make_err (dup_err x)
21      | Var(x, t) ->  StringMap.add (string_of_expr x) var map
22      | TypeAssigndecl(x, t) when StringMap.mem (string_of_expr x) map -> make_err (dup_err x)
23      | TypeAssigndecl(x, t) ->  StringMap.add (string_of_expr x) var map
24      | MapAssigndecl(x, t) when StringMap.mem (string_of_expr x) map -> make_err (dup_err x)
25      | MapAssigndecl(x, t) ->  StringMap.add (string_of_expr x) var map
26      | Eventdecl(x, t) when StringMap.mem (string_of_expr x) map -> make_err (dup_err x)
27      | Eventdecl(x, t) ->  StringMap.add (string_of_expr x) var map
28      | _ -> map
29    in
30
31    (* Collect all variable names into one symbol table *)
32    let var_decls = List.fold_left add_var StringMap.empty signature.interfacebody in
33
34    (* Add method name in interface to symbol table *)
35    let add_func map func =
36      let dup_err v = "duplicate method " ^ (string_of_expr v) ^ " in interface"
37      and make_err er = raise (Failure er)
38      in match func with (* No duplicate variables or redefinitions of built-ins *)
39        Constructordecl(l, t1, t2) when StringMap.mem (string_of_expr l) map -> make_err (dup_err l)
40      | Constructordecl(l, t1, t2) ->  StringMap.add (string_of_expr l) func map
41      | Methodecls (l, t1, t2) when StringMap.mem (string_of_expr l) map -> make_err (dup_err l)
42      | Methodecls (l, t1, t2) ->  StringMap.add (string_of_expr l) func map
```

```
43      | _ -> map
44    in
45
46    (* Collect all function names into one symbol table *)
47    let func_decls = List.fold_left add_func StringMap.empty signature.interfacebody in
48
49    (* Return a function from our symbol table *)
50    let find_func s =
51      try StringMap.find s func_decls
52      with Not_found -> raise (Failure ("unrecognized method " ^ s))
53    in
54
55    let count_constructor num func =
56      match func with
57        Constructordecl(l, t1, t2) -> num +1
58        | _ -> num
59    in
60
61    (* check constructor only announce once in interface *)
62    let _  =
63      let constructor_num = List.fold_left count_constructor 0 signature.interfacebody in
64      match constructor_num with
65      0 -> raise (Failure "No constructor in interface")
66      | 1 -> constructor_num
67      | _ -> raise (Failure "Multiple constructors in interface")
68    in
69
70    (* Check all methods are implemented only once *)
71
72    let add_implement map impl =
73      let dup_err v = "duplicate method " ^ (string_of_expr v) ^ " in implementation"
74      and make_err er = raise (Failure er)
75      in match impl with
76        impl when StringMap.mem (string_of_expr impl.methodname) map -> make_err (dup_err impl.methodname)
77        | impl ->  StringMap.add (string_of_expr impl.methodname) impl map
78    in
79
80    let _ = List.fold_left add_implement StringMap.empty  implementation.methods in
81
82    let rec check_expr = function
83      | NumLit l -> (Int, SNumLit l)
84      | BoolLit l -> (Bool, SBoolLit l)
85      | StrLit l -> (Void("void"), SStrLit l)
86      (* check Id retrun with the correct type, keep Int for now *)
87      | Id x -> (Int, SId(Sglobal, x))
88      | EnvLit(x, y) -> (Void("Env"), SEnvLit(x,y))
89      | Mapexpr(e1, e2) -> (Int, SMapexpr(check_expr e1, List.map check_expr e2))
90      | Binop(e1, op, e2) -> (Int, SBinop(check_expr e1, op, check_expr e2))
91      | Logexpr(e1, e2) -> (Void("void"), SLogexpr(check_expr e1, List.map check_expr e2))
92      | Storageassign (e1, e2) -> (Int, SStorageassign(check_expr e1, check_expr e2))
```

```
93      | Comparsion (e1, op, e2) -> (Int, SComparsion(check_expr e1, op, check_expr e2))
94      | Voidlit(s) -> (Void("void"), SVoidlit(s) )
95    in
96
97    let check_func func =
98
99      let check_args_type var1 t2 =
100       let check_type x1 t1 t2 =  let tag = (t1 = t2)
101       and unmatch_err = "function argument " ^ string_of_expr x1 ^ " has type "
102       ^ string_of_typ t1 ^ " ,which is unmatch with declaration type " ^ string_of_typ t2 in
103         match tag with
104         true -> t1
105         | false -> raise (Failure unmatch_err)
106       in
107       match var1, t2 with
108       Var(x1, t1), t2 -> check_type x1 t1 t2
109       | _, _ -> raise (Failure "Not a legal variables in arguments")
110
111     in
112
113     let sfunc = function
114        | Id l -> (Void("void"), SStrLit l)
115        | e -> raise (Failure ("Not a function name " ^ string_of_expr e))
116     in
117
118     let func_decl = find_func (string_of_expr func.methodname) in
119
120     let params_types, return_type = match func_decl with
121       Methodecls(expr, typli, typ) -> (typli, typ)
122       | _ -> raise (Failure "Not legal method")
123     in
124
125     (* If the only arg is void and no arguments in method, then skip check args *)
126     let skip_check_args =
127       if List.length params_types = 1 then
128         let first_arg = List.hd params_types in
129         match first_arg with
130         Void("void") -> if List.length func.params = 0 then true else false
131         | _ -> false
132       else false
133
134
135     in
136
137     let _ = if skip_check_args then true else
138
139       (* Check argument types length matches with declaration  *)
140       let _ = let typ_len_func = List.length func.params
141         in let typ_len_decl = List.length params_types in
142         match typ_len_func, typ_len_decl with
```

```
143         typ_len_func, typ_len_decl when (typ_len_func > typ_len_decl)
144         -> raise (Failure ("Redundant arguments in method " ^ string_of_expr func.methodname))
145         | typ_len_func, typ_len_decl when (typ_len_func < typ_len_decl)
146         -> raise (Failure ("Missing arguments in method " ^ string_of_expr func.methodname))
147         | _, _ -> typ_len_func
148       in
149
150       (* Check whether variable argument type matches with declaration *)
151       let _ = (List.map2 check_args_type func.params params_types) in false
152
153     in
154     let add_var_args map var =
155       let dup_err v = "duplicate variable " ^ (string_of_expr v) ^ " in method arguments"
156       and make_err er = raise (Failure er)
157       in match var with (* No duplicate variables or redefinitions of built-ins *)
158         Var(x, t) when StringMap.mem (string_of_expr x) map -> make_err (dup_err x)
159       | Var(x, t) ->  StringMap.add (string_of_expr x) var map
160       | _ -> raise (Failure "Only variable allows in method arguments")
161     in
162
163     let var_sym = List.fold_left add_var_args var_decls func.params in
164
165     (* Return a variable from our symbol table *)
166     let find_var s = let s_type =
167       try StringMap.find s var_sym
168       with Not_found -> raise (Failure ("unrecognized variable " ^ s))
169       in
170       match s_type with
171       Var(x, t) -> t
172       | TypeAssigndecl(x, t) -> t
173       | MapAssigndecl(x, t) -> t
174       | Eventdecl(x, t) -> Void("void")
175       | _ -> raise (Failure ("unrecognized variable " ^ string_of_decl s_type ))
176     in
177
178     let rec check_expr = function
179       | NumLit l -> (Int, SNumLit l)
180       | BoolLit l -> (Bool, SBoolLit l)
181       | StrLit l -> (Void("void"), SStrLit l)
182       (* check Id retrun with the correct type, keep Int for now *)
183       | Id x ->
184         let t = find_var x in
185         if StringMap.mem x var_decls then
186           (find_var x, SId(Sglobal, x))
187         else (find_var x, SId(Slocal, x))
188       | EnvLit(x, y) -> (Void("Env"), SEnvLit(x,y))
189       | Mapexpr(e1, e2) as e ->
190         let id_err = string_of_expr e1 ^ " is not a id in " ^ string_of_expr e in
191         let (t1, e1') = match e1 with
192           Id(id) -> check_expr e1
```

```
193           | _ -> raise (Failure id_err)
194         in
195         let type_err = "Id " ^ string_of_expr e1 ^ " " ^
196                        " is " ^ string_of_typ t1 ^
197                        " type, not a map struct in " ^ string_of_expr e in
198         let e2' = List.map check_expr e2 in
199         let check_map_key_type key_type sexpr2 =
200           match sexpr2 with
201             (type2, sx2) ->
202             let key_err = "Expresion "  ^ (string_of_sexpr sexpr2)
203                       ^ " has type " ^ (string_of_typ type2) ^ ", but type "
204                       ^ (string_of_typ key_type) ^ " is required in "
205                       ^ string_of_expr e
206             in
207
208             if type2 = Void("Env") then sexpr2
209             else if key_type = type2 then
210             match key_type with
211              Int | Uint("uint") | Address("ADDRESS") -> sexpr2
212             | _ -> raise (Failure ("Type " ^ string_of_typ key_type ^
213             " is not allowed as key type in map " ^ string_of_expr e ))
214             else
215               raise (Failure key_err)
216         in
217         let value_type = match t1 with
218           Mapstruct(key_typli, value_type) ->
219           (* Check map query types length matches with map declaration  *)
220           let key_type_ls_len = List.length key_typli
221           and query_type_ls_len = List.length e2' in
222           let _ =
223           match key_type_ls_len, query_type_ls_len with
224           key_type_ls_len, query_type_ls_len when
225           (key_type_ls_len > query_type_ls_len) ->
226           raise (Failure ("Missing query value in map " ^ string_of_expr e))
227           | key_type_ls_len, query_type_ls_len when
228           (key_type_ls_len < query_type_ls_len) ->
229           raise (Failure ("Redundant query value in map " ^ string_of_expr e))
230           | _ -> key_type_ls_len
231           in
232           let _ = List.map2 check_map_key_type key_typli e2' in
233           value_type
234           | _ -> raise (Failure type_err)
235         in
236         (value_type, SMapexpr((t1, e1'), e2'))
237       (* Binop : Add | Sub | Times | Divide | And | Or *)
238       | Binop(e1, op, e2) as e ->
239         let (t1, e1') = check_expr e1
240         and (t2, e2') = check_expr e2 in
241         let err = "Illegal binary operator " ^
242                   string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
```

```
243                 string_of_typ t2 ^ " in " ^ string_of_expr e
244           in
245           (* All binary operators require operands of the same type*)
246
247           if t1 = Void("Env") then
248             (t2, SBinop((t1, e1'), op, (t2, e2')))
249           else if t2 = Void("Env") then
250             (t1, SBinop((t1, e1'), op, (t2, e2')))
251           else if t1 = t2 then
252             (* Determine expression type based on operator and operand types *)
253             let t = match op with
254               Add | Sub | Times | Divide when t1 = Uint("uint") -> Uint("uint")
255               | Add | Sub | Times | Divide when t1 = Int -> Int
256               | And | Or when t1 = Bool -> Bool
257               | _ -> raise (Failure err)
258             in
259             (t, SBinop((t1, e1'), op, (t2, e2')))
260           else if (t1 = Uint("uint") && t2 = Int) || (t1 = Int && t2 = Uint("uint")) then
261             let t = match op with
262             Add | Sub | Times | Divide -> Int
263             | _ -> raise (Failure err)
264             in
265             (t, SBinop((t1, e1'), op, (t2, e2')))
266           else raise (Failure err)
267
268       | Logexpr(e1, e2) -> (Void("Log"), SLogexpr(check_expr e1, List.map check_expr e2))
269       | Storageassign (e1, e2) as e ->
270         let (t1, e1') = check_expr e1
271         and (t2, e2') = check_expr e2 in
272         let err = "Illegal storage assign: " ^
273                   string_of_typ t1 ^ " <- " ^
274                   string_of_typ t2 ^ " in " ^ string_of_expr e
275         in
276         (* All binary operators require operands of the same type*)
277         if t2 = Void("Env") && t1 != Void("Env") then
278           (t1, SStorageassign((t1, e1'), (t2, e2')))
279         else if (t1 = Uint("uint") && t2 = Int) || (t1 = Int && t2 = Uint("uint")) then
280           (Void("void"), SStorageassign((t1, e1'), (t2, e2')))
281         else if t1 = t2 then
282           (Void("void"), SStorageassign((t1, e1'), (t2, e2')))
283         else raise (Failure err)
284
285       (* Comparsion : Equal | Neq | LGT | RGT | LGTEQ | RGTEQ *)
286       | Comparsion (e1, op, e2) as e ->
287         let (t1, e1') = check_expr e1
288         and (t2, e2') = check_expr e2 in
289         let err = "Illegal binary operator " ^
290                   string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
291                   string_of_typ t2 ^ " in " ^ string_of_expr e
292         in
```

```
293         (* All binary operators require operands of the same type*)
294         if t1 = Void("Env") then
295           (t2, SComparsion((t1, e1'), op, (t2, e2')))
296         else if t2 = Void("Env") then
297           (t1, SComparsion((t1, e1'), op, (t2, e2')))
298         else if t1 = t2 then
299           (* Determine expression type based on operator and operand types *)
300           let t = match op with
301             | Equal | Neq | LGT | RGT | LGTEQ | RGTEQ when t1 = Uint("uint") || t1 = Int -> Bool
302             | _ -> raise (Failure err)
303           in
304           (t, SComparsion((t1, e1'), op, (t2, e2')))
305         else if (t1 = Uint("uint") && t2 = Int) || (t1 = Int && t2 = Uint("uint")) then
306           let t = match op with
307           | Equal | Neq | LGT | RGT | LGTEQ | RGTEQ -> Bool
308           | _ -> raise (Failure err)
309           in
310           (t, SComparsion((t1, e1'), op, (t2, e2')))
311         else raise (Failure err)
312       | Voidlit(s) -> (Void("void"), SVoidlit(s) )
313     in
314
315     let sreturns =
316       let (t, sx) = check_expr func.returns in
317       match t with
318       Void("Env") -> (return_type, sx)
319       | _ -> if t = return_type then (t, sx)
320       else
321         let return_type_unmatch_err = "Return type of method " ^ (string_of_expr func.methodname)
322         ^ " is: " ^ (string_of_typ return_type) ^ ",\n with is unmatch with "
323         ^ " expression: " ^ (string_of_sexpr (t, sx)) in
324         if t = return_type then (t, sx)
325         else raise (Failure return_type_unmatch_err)
326
327     in
328     {
329       smethodname = sfunc func.methodname;
330       sparams = func.params;
331       sguard_body = List.map check_expr func.guard_body;
332       sstorage_body = List.map check_expr func.storage_body;
333       seffects_body = List.map check_expr func.effects_body;
334       sreturns = sreturns;
335     }
336   in
337
338   let sinterface_def =
339       {
340         ssignaturename = check_expr signature.signaturename;
341         sinterfacebody = signature.interfacebody;
342       }
```

```
  in

  let simplementation_def =
    {
      sconsturctor = {
        sname = check_expr implementation.consturctor.name;
        sparams = implementation.consturctor.params;
        sconsturctor_body = List.map check_expr implementation.consturctor.consturctor_body;
        sreturn_type = implementation.consturctor.return_type;
      };

      smethods = List.map check_func implementation.methods;
    }
  in

  let sprogram = (sinterface_def, simplementation_def)
  in
  sprogram
```

## B.4 Minic translator

```
open Ast
open Sast
let sprintf = Printf.sprintf

open Language


let rec positive_of_int n =
  let open BinNums in
  if n = 1 then
    Coq_xH
  else if (n land 1) = 1 then
    Coq_xI (positive_of_int (n asr 1))
  else
    Coq_xO (positive_of_int (n asr 1))

let coq_Z_of_int n =
  let open BinNums in
  if n = 0 then Z0
  else if n > 0 then Zpos (positive_of_int n)
  else Zneg (positive_of_int (-n))

let rec int_of_positive p =
let open BinNums in
  match p with
  | Coq_xI rest -> 2*(int_of_positive rest) + 1
  | Coq_xO rest -> 2*(int_of_positive rest)
  | Coq_xH -> 1
```

```
29
30  let int_of_z =
31  let open BinNums in
32    function
33    | Z0 -> 0
34    | Zpos rest -> int_of_positive rest
35    | Zneg rest -> -(int_of_positive rest)
36
37  let rec coqlist_of_list =
38    let open Datatypes in
39    function
40    | [] -> Coq_nil
41    | x::xs -> (Coq_cons (x, coqlist_of_list xs))
42
43  let rec filter_map f ls =
44    let open Datatypes in
45      match ls with
46      | [] -> []
47      | x::xs -> match f x with
48            | Some y -> y :: filter_map f xs
49            | None -> filter_map f xs
50
51  let ident_table : (string, int) Hashtbl.t = Hashtbl.create 1000
52  let ident_counter : int ref = ref 550
53
54  (** ident_generator : positive **)
55  let ident_generator = fun prefix midfix postfix ->
56    let id = (prefix ^ midfix ^ "_"^ postfix) in
57    try positive_of_int (Hashtbl.find ident_table id)
58    with Not_found -> begin
59        let n = !ident_counter in
60        ident_counter := !ident_counter + 1;
61        Hashtbl.add ident_table id n;
62        positive_of_int n
63      end
64
65  let struct_name_to_ident2 = ident_generator "" "struct"
66  let struct_field_name_to_ident2 = ident_generator "" "field"
67  let backend_ident_of_globvar  = ident_generator "var_" "var2"
68  let backend_ident_of_funcname = ident_generator "ident_" "function"
69  let backend_ident_of_tempvar =  ident_generator "temp_" "var"
70
71  let rec gen_ctype =
72    let open Ctypes in
73    function
74    | Bool -> Tint (I256, Unsigned)
75    | Int -> Tint (I256, Signed)
76    | Uint x  -> Tint (I256, Unsigned)
77    | Void x -> Tvoid
78    | Address x -> Tint (I256, Unsigned)
```

```
79    | Mapstruct (key_ty, val_ty) -> Thashmap (gen_ctype (List.hd key_ty), gen_ctype val_ty)
80
81  let gen_unop =
82    let open Cop in
83    function
84    | Neq -> Oneg
85    | _ -> raise (Failure "Not a unop!")
86
87  let gen_binop =
88    let open Cop in
89    function
90    | Add -> Oadd
91    | Sub -> Osub
92    | Times -> Omul
93    | Divide -> Odiv
94    | And -> Oand
95    | Or -> Oor
96    | Equal -> Oeq
97    | Neq -> One
98    | RGT -> Olt
99    | RGTEQ -> Ole
100   | LGT -> Ogt
101   | LGTEQ -> Oge
102   | PASSIGN -> raise (Failure "PASSIGN should be solved as Storageassign in expr")
103
104
105 let rec gen_rexpr e =
106   let open Integers in
107   let open Language in
108   match e with
109   | (t, SId(Sglobal,l)) -> Evar (backend_ident_of_globvar l, gen_ctype t)
110   | (t, SId(Slocal,l)) -> Etempvar (backend_ident_of_tempvar l, gen_ctype t)
111   | se -> raise (Failure ("Not implemented: " ^ string_of_sexpr se))
112
113 let rec gen_lexpr e =
114   let open Ctypes in
115   let open Integers in
116   let open Language in
117   let open MachineModel in
118   match e with
119   | (t, SNumLit l) -> Econst_int256 (Int256.repr (coq_Z_of_int l), gen_ctype Int)
120   | (t, SBoolLit l) -> (match l with
121                         |true -> Econst_int256 (Int256.one, Tint (I256, Unsigned))
122                         |false -> Econst_int256 (Int256.zero, Tint (I256, Unsigned)) )
123   | (t, SId(Sglobal,l)) -> Evar (backend_ident_of_globvar l, gen_ctype t)
124   | (t, SId(Slocal,l)) -> Etempvar (backend_ident_of_tempvar l, gen_ctype t)
125   | (t1, SBinop ((t2, se1), op, (t3, se2))) -> Ebinop (gen_binop op, gen_lexpr (t2, se1), gen_lexpr (t3, se2), gen_ct
126   | (t, SComparsion ((t1, se1), op, (t2, se2))) -> Ebinop (gen_binop op, gen_lexpr (t1, se1), gen_lexpr (t2, se2), ge
127   | (t, SMapexpr((t1, se1), selist)) ->
128     (* TODO: convert selist's type to Tstruct *)
```

```
129       let se2 = List.hd selist in
130       Ehashderef(gen_lexpr (t1, se1), gen_lexpr se2, gen_ctype t)
131     | (t, SEnvLit(s1, s2)) ->
132       (
133         match s2 with
134         | "sender" -> Ecall0 (Bcaller, Tvoid)
135         | "value" -> Ecall0 (Bcallvalue, Tvoid)
136         | "origin" -> Ecall0 (Borigin, Tvoid)
137         | "sig" -> Ecall0 (Baddress, Tvoid)
138         | "data" -> Ecall0 (Baddress, Tvoid)
139         | _ -> let _ = print_endline ("Waring: Env key may not support") in
140         Ecall0 (Baddress, Tvoid)
141       )
142     | se -> raise (Failure ("Not implemented: " ^ string_of_sexpr se))
143
144   (** gen_assign_stmt : statement **)
145   let gen_assign_stmt e1 e2 =
146     let open Language in
147     Sassign(gen_lexpr e1, gen_lexpr e2)
148
149   let gen_set_stmt id e1 =
150     let open Language in
151     Sset (positive_of_int id, gen_rexpr e1)
152
153   let gen_guard_stmt e =
154     let open Language in
155     Sifthenelse(gen_lexpr e, Sskip, Srevert)
156
157   (* sparams: decls list *)
158   (** gen_params :
159       (ident, coq_type) prod list; **)
160   let gen_params sparams =
161     let open Datatypes in
162     let open Globalenvs.Genv in
163     let cvt = function
164       | Var(Id str, typ) -> Some (Coq_pair(backend_ident_of_tempvar str, gen_ctype typ))
165       | _ -> None
166     in
167     coqlist_of_list (filter_map cvt sparams)
168
169   (* storagebody: sexpr list *)
170   (** gen_storage_cmd : statement **)
171   let gen_storage_cmd storebody =
172     let open Datatypes in
173     let rec list2seq = function
174       | [] -> Sskip
175       | hd::[] -> hd
176       | hd::tl -> Ssequence(hd, list2seq tl)
177     in
178     let sexpr2Sassign = function
```

```
    | (typ, SStorageassign(lsexpr, rsexpr)) -> Some(gen_assign_stmt lsexpr rsexpr) (* (gen_assign_stmt (Int, (SId "st
    | _ -> None
  in
  list2seq (filter_map sexpr2Sassign storebody)

let gen_guard_cmd guardbody =
  let open Datatypes in
  let rec list2seq = function
    | [] -> Sskip
    | hd::[] -> hd
    | hd::tl -> Ssequence(hd, list2seq tl)
  in
  let sexpr2stmt = function
    | se -> Some(gen_guard_stmt se)
  in
  list2seq (filter_map sexpr2stmt guardbody)

let gen_return_cmd (return_type, sx) =
  let open Datatypes in
  let return_expr =
    match return_type with
    Void(_) -> None
    | _ -> Some(gen_lexpr (return_type, sx))
  in
  Sreturn(return_expr)

(** gen_methoddef : coq_function **)
let gen_methoddef m =
  let open Datatypes in
  let method_classify (ty, _) = match ty with
    | Void s -> false
    | _ -> true
  in
  (* let dest = builtinBase_local_ident_start in   *) (* let builtinBase_local_ident_start = 10 *)
  (* let is_pure, has_return = method_classify mt in *)
  let has_return = method_classify m.sreturns in
  (* let body = gen_set_stmt  builtinBase_local_ident_start (List.hd m.sstorage_body) in *)
  (* let body =  *)
    (* gen_storage_cmd m.sstorage_body *)
    (* Ssequence(gen_guard_cmd m.sguard_body, gen_storage_cmd m.sstorage_body) *)
  let ret_type (ty, sx) = gen_ctype ty in
  {
    fn_return = ret_type m.sreturns;
    fn_params = gen_params m.sparams; (* (ident, coq_type) prod list; *)
    fn_temps  = Coq_nil; (* coqlist_of_list (gen_tempenv ((dest,mt.aMethodReturnType.aTypeCtype) :: gen_cmd_locals m.
    fn_body =  (if has_return then
      Ssequence(Ssequence(gen_guard_cmd m.sguard_body, gen_storage_cmd m.sstorage_body), gen_return_cmd m.sreturns)
                else
      Ssequence(gen_guard_cmd m.sguard_body, gen_storage_cmd m.sstorage_body))
  }
```

```
229
230 let gen_object_methods gen_methodname gen_method o =
231   let open Datatypes in
232   coqlist_of_list
233     (List.map
234       (fun m -> Coq_pair (gen_methodname m, gen_method m))
235       o.smethods)
236
237 (** gen_object_fields :
238     vars: (ident, coq_type) prod list **)
239 let gen_object_fields declist =
240   let open Datatypes in
241   let open Globalenvs.Genv in
242   let decl2gvars = function
243     | TypeAssigndecl(Id s, t) -> Some (Coq_pair(backend_ident_of_globvar s, gen_ctype t))
244     | MapAssigndecl(Id s, t) -> Some (Coq_pair(backend_ident_of_globvar s, gen_ctype t))
245     | _ -> None
246   in
247   coqlist_of_list (filter_map decl2gvars declist)
248
249 (** gen_object : genv **)
250 (* (i, o) = (sinterface, simplementation) *)
251 let gen_object (i, o) =
252   let open Datatypes in
253   let open Globalenvs.Genv in
254   let open Cryptokit in
255   let keccak_intval (_, SStrLit str) =
256     let hashval = hash_string (Hash.keccak 256) str in
257       (0x01000000) * Char.code (String.get hashval 0)
258     + (0x00010000) * Char.code (String.get hashval 1)
259     + (0x00000100) * Char.code (String.get hashval 2)
260     +                Char.code (String.get hashval 3)
261   in
262   (* let make_funcname m = backend_ident_of_funcname o.sconsturctor_def.sname m.smethodname in *)
263   (* let make_methname m = coq_Z_of_int 1101101111 in *)
264   (* let make_methname m = coq_Z_of_int (function_selector_intval_of_method m) in *) (* function_selector_intval_of_m
265   let make_methname m = coq_Z_of_int (keccak_intval m.smethodname) in
266     new_genv (* new_genv: vars -> funcs -> methods -> constructor  *)
267       (gen_object_fields i.sinterfacebody) (* vars: (ident, coq_type) prod list *)
268       Coq_nil (* funcs: (id, coq_fun) prod list. Only the lower layers have funcs *)
269       (gen_object_methods make_methname gen_methoddef o) (* methods: (Int.int, coq_fun) prod list *)
270       None
271
272 let minicgen sprogram = gen_object sprogram
```

## B.5 OpenSC Makefile

```
1 .PHONY: opensc
2 opensc:
```

```
	ocamlbuild -pkg cryptokit -I backend opensc.native
```

## B.6 OpenSC translator

```
open Sast
open TranslateMinic
open Ast

type mode = AST | SAST | MINIC | BYTECODE


let usage () =
  prerr_endline ( "usage: ./opensc.native program.sc (ast | sast | minic | bytecode) \n");
  exit 1

let main argv =
  let open LanguageExt in
  let open Datatypes in
  let open Glue in
  let open ASM in
  let open DatatypesExt in
  (if (Array.length argv <> 3) then usage());
  let filename = argv.(1) in
  let mode_flag = match Array.get argv 2 with
    | "ast" -> AST
    | "sast" -> SAST
    | "minic" -> MINIC
    | "bytecode" -> BYTECODE
    | _ -> usage() in
  let ch = open_in filename in
  let lexbuf = Lexing.from_channel ch in
  let program = Parser.program Scanner.token lexbuf in
  let sprogram = Semant.check program in
(*   print_endline (string_of_sprogram sprogram) in *)
  let minicAST = TranslateMinic.minicgen sprogram in
    match mode_flag with
    | AST ->   print_endline (string_of_program program)
    | SAST -> print_endline (string_of_sprogram sprogram)
    | MINIC -> print_endline (show_genv minicAST)
    | BYTECODE ->
     match full_compile_genv minicAST with
     | None -> print_endline "Compilation failed"; exit 1
     | Some (Coq_pair (program, entrypoint)) ->
        let asm =
          transform
            (List.rev (caml_list program))
            entrypoint in
            print_endline (assemble asm)

```

```
46
47
48 let _ = main Sys.argv
49
50 (* ocamlbuild -pkg cryptokit -I backend opensc.native *)
```

## C Test Cases

```
1  01_check_var_exist_succ.sc
2  02_check_var_exist_fail.sc
3  03_check_var_duplicate_announce_fail.sc
4  04_check_func_exist_succ.sc
5  05_check_func_exist_fail.sc
6  06_check_func_duplicate_announce_fail.sc
7  07_check_func_duplicate_implement_fail.sc
8  08_check_func_constructor_announce_once_fail.sc
9  09_check_func_constructor_announce_once_fail2.sc
10 10_check_var_in_method_succ.sc
11 11_check_var_duplicate_in_method_fail.sc
12 12_check_var_method_miss_args_fail.sc
13 13_check_var_method_redundant_args_fail.sc
14 14_check_var_args_unmatch_decl_type_fail.sc
15 15_check_var_duplicate_with_global_var_in_args_fail.sc
16 16_check_var_in_method_type_succ.sc
17 17_check_var_in_method_return_type_succ.sc
18 18_check_var_in_method_return_type_fail.sc
19 19_check_binop_add_sub_succ.sc
20 20_check_binop_add_sub_fail.sc
21 21_check_binop_times_divide_succ.sc
22 22_check_binop_times_divide_fail.sc
23 23_check_binop_eq_neq_succ.sc
24 24_check_binop_eq_neq_fail.sc
25 25_check_binop_lgt_rgt_eq_succ.sc
26 26_check_binop_lgt_rgt_eq_fail.sc
27 27_check_binop_and_or_succ.sc
28 28_check_binop_and_or_fail.sc
29 29_check_binop_passign_succ.sc
30 30_check_binop_passign_fail.sc
31 31_check_binop_combine01_succ.sc
32 32_check_binop_combine02_succ.sc
33 33_check_binop_combine03_succ.sc
34 34_check_binop_combine04_succ.sc
35 35_check_map_query_succ.sc
36 36_check_map_id_err_fail.sc
37 37_check_map_not_map_type_fail.sc
38 38_check_map_query_value_miss_fail.sc
39 39_check_map_query_redundant_fail.sc
40 40_check_map_query_wrong_type_fail.sc
```

```
41_check_map_query_key_type_not_allowed_fail.sc
42_check_map_query_assign_succ.sc
43_check_map_query_assign_unmatch_fail.sc
```