Overview

Monday, February 29, 2016 4:51 PM

Plan:

- Come up with examples
- Build BNF
- Create classes for BNF
- Codify BNF -> HTML

Mono

Monday, February 29, 2016 2:50 PM

https://github.com/mono/monodevelop
https://github.com/mono/mono

Get Mono, the cross-platform, open source .NET runtime implementation used by F#. Preferably use a package from your distribution or Xamarin. If this is not possible, install from source by following these instructions.

Note that if you are installing to a private prefix, follow these instructions and ensure LD_LIBRARY_PATH includes the “lib” directory of that prefix location and PKG_CONFIG_PATH includes the “lib/pkgconfig” directory of that prefix location, e.g.

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/user/mono/lib/
export PKG_CONFIG_PATH=/home/user/mono/lib/pkgconfig/
```

Build and install the F# Compiler (open edition) from source. If using a VM or other memory-constrained system, be aware that errors during compilation may be due to insufficient memory (in particular error 137).

```
sudo apt-get install autoconf libtool pkg-config make git automake
git clone https://github.com/fsharp/fsharp
cd fsharp
./autogen.sh --prefix /usr
make
sudo make install
```

Overview 1

Monday, February 29, 2016 2:49 PM

Data Definitions:

```
var type Name = Value // Single line definitions, the type is optional

var type Name. // When it is to be populated by actions, the end with period

var type Name. // Type is mandatory

var type Name // type is optional
{
        // All long definitions here, the new lines will be ignored
}

// Should I even have Predicates? To start with, NO!
var NameOfPredicate(type1 Name1, type2 Name2, ...) // Types all optional, they could be parameters
{
        Code ...
| // Or
        Code...
}

// Types of mandatory, cannot be parameters
// Need to end with period, will be filled with export actions
var NameOfPredicate(type1 Name1, type2 Name2, ...).
```

Rules:

```
do
{
        // The action to be done
|
        export Name(A, B, C)
}

do NameOfAction(type1 Name1, type2 Name2, ...) // Types all optional, they could be parameters
{
        Code...
| // Or
        Code...
}


// Types are optional everywhere, define being used for functions
define type Fun(type1 Name1, type2 Name2, …)
{
        Code ...
}
```

Types:

```
type NameOfType<P1, P2,…> : interface1<P1, P2,…>, interface2<P1, P2,…>, …
{
        //Classes either end with period or are defined inline
        Name1(type1, type2, type3). // Names are optional
        Name2(type1, type2, type3).  // Type are not optional
          …
}

// Names are optional, types are not. Note the period ending
class NameOfClass<P1, P2,…>(type1 Name1, type2 Name2, …).

// Names are optional, types are not. Note the period ending
class NameOfClass<P1, P2,…>(type1 Name1, type2 Name2, …) :  interface1<P1, P2,…>, ...
{
        // Constants: Types are optional
        // Name cannot be repeated from the list above
        var type Name = expr

        // Name cannot be repeated from the list above
        // Types are optional everywhere
        define type Fun(type1 Name1, type2 Name2, …)
        {
                Code ...
        }
}

interface Name<P1, P2,…>
{
        // Types are not optional anywhere
        // Definitions needs to end its period
        // Functions/Vars should NOT be defined inline
        var type Name.
        define type Fun(type1 Name1, type2 Name2, …).
}
```

Overview 2

Wednesday, March 2, 2016 11:22 PM

Major change: Remove types on all functions. But when you create an object, type should specified, if could not infer on your own.

Data Definitions:

```
var Name Value : type = // Single line definitions, the type is optional

// When it is to be populated by actions, the end with period
 // type is optional, but is determinitic
var Name : type.

var Name : type // type is optional
{
        // All long definitions here, the new lines will be ignored
}

// Should I even have Predicates? To start with, NO!
var NameOfPredicate(type1 Name1, type2 Name2, ...) // Types all optional, they could be parameters
{
        Code ...
| // Or
        Code...
}

// Types of mandatory, cannot be parameters
// Need to end with period, will be filled with export actions
// This is multi-determs
var NameOfPredicate(Name1 : type1, Name2 : type2 , ...).
```

Rules:

```
do
{
        // The action to be done
|
        export Name(A, B, C)
}

do NameOfAction(Name1 : type1, Name2 : type2 , ...) // Types all optional, they could be parameters
{
        Code...
| // Or
        Code...
}
```

```
// Types are optional everywhere, define being used for functions
define fun(Name1 : type1, Name2 : type2 , ...) : type
{
        Code ...
}
```

Types:

```
// Using interface is optional, it uses OCaml style names matching ...
type NameOfType<P1, P2,…> : interface1, interface2, …
{
        //Classes either end with period or are defined inline
        Name1(type1, type2, type3). // Names are optional
        Name2(type1, type2, type3).  // Type are not optional
          …
}

// Names are optional, types are not. Note the period ending
class NameOfClass<P1, P2,…>(type1 Name1, type2 Name2, …).

// Names are optional, types are not. Note the period ending
class NameOfClass<P1, P2,…>(type1 Name1, type2 Name2, …)
{
        // Constants: Types are optional
        // Name cannot be repeated from the list above
        var Name = expr

        // Name cannot be repeated from the list above
        // Types are optional everywhere
        define Fun(Name1, Name2, …)
        {
                Code ...
        }
}

interface Name<P1, P2,…>
{
        // Types are not optional anywhere
        // Definitions needs to end its period
        // Functions/Vars should NOT be defined inline
        var type Name.
        define type Fun(type1 Name1, type2 Name2, …).
}
```

```
type args = string * string // Type, Name
type union = string * string * args list // Tag, Description, Args
type typeDefine =
   | Union of string * string * union list// Name, Description, UnionsCases
   | Tuple of string * string * args list // Name, Description, Args
   | Record of string * string * args list // Name, Description, Args

type defination =
   | Singleton of typeDefine
   | Chain of typeDefine list

let FileData =
   let moduleName = "Defination"
   let fileName = moduleName + ".fs"
   let description = "All definations are contained here"
   let data = (moduleName, fileName, description)
   (data, FileHeader data)

let Definations =
   // TODO: For now there are no types
   let program = Record("program", "Structure for the Program", [("module list", "Modules")] )
   let moduleDef = Tuple("defNamespace", "Hold the Namespace", [("string","Name"); ("defDeclation list", "Dec
   let constDec : union = ("defConst", "Constant Declaration", [("Identifier", "Name") ; ("defType option","T
   let defineDef = Union("defDefine", "Declaration", [ constDec ] )
   let position = Tuple("position", "Position to track tokens", [("string","FileName"); ("uint32","LineNo");
   let Identifier = Tuple("Identifier", "Upper Identifier", [("position","Position"); ("string","Name") ] )
   let identifier = Tuple("identifier", "Lower Identifier", [("position","Position"); ("string","Name") ] )
   [Singleton(position); Singleton(identifier); Singleton(Identifier) ; Singleton(defineDef) ; Singleton(prog

(*

// ____________________________________________________________________________________________
//   Identifier: Consists of string chat, one with small letter - other with large
type Identifier = position * string // String should be upper identifier

type identifier = position * string // String should be lower identifier

// ____________________________________________________________________________________________
//   moduleIdentifier: Identifier to track items in modules
type moduleIdentifier =
   | IdentifierThis of Identifier
   | IdentifierThat of Identifier * Identifier

// ____________________________________________________________________________________________
//   expr: Expression
type expr =
   | ExprNumber of position * int64 // Number
   | ExprString of position * string // String
```

```
    | ExprBool of position * bool // String
    | ExprList of position * expr list // List [1, 2, 3, ..]
    | ExprTailList of position * expr list * expr // [1, 2, 3 | X]
    | ExprVar of moduleIdentifier // Variable
    | ExprTuple of position * expr list // {identifier}({expr} ...)
    | ExprTerm of identifier * expr list
    | ExprFunCall of moduleIdentifier * expr list // {identifier}({expr} ...)
    | ExprRecord of position * Map<Identifier, expr>
    | ExprMapCall of moduleIdentifier * expr
    | ExprAdd of expr * expr // {expr1} + {expr2}
    | ExprSub of expr * expr // {expr1} - {expr2}
    | ExprMult of expr * expr // {expr1} * {expr2}
    | ExprDiv of expr * expr // {expr1} / {expr2}
    | ExprMod of expr * expr // {expr1} % {expr2}
    | ExprConcat of expr * expr // String concatenation
    | ExprJoin of expr * expr // {expr1} is head and {expr2} is tail
    | ExprAddList of expr * expr // Adding two list
    | ExprNegate of position * expr // -{expr}
    | ExprIfThen of position * expr * expr // if {cond} then {expr1} else {expr2}
    | ExprIfThenElse of position * expr * expr * expr // if {cond} then {expr1} else {expr2}
    | ExprMemberCheck of expr * expr // {expr1} in {expr2}
    | ExprAnd of expr * expr // {cond1} and {cond2}
    | ExprOr of expr * expr // {cond1} or {cond2}
    | ExprNot of position * expr // not {cond}
    | ExprGtEq of expr * expr // {expr1}>={expr2}
    | ExprGt of expr * expr // {expr1}>{expr2}
    | ExprLtEq of expr * expr // {expr1}<={expr2}
    | ExprLt of expr * expr // {expr1}<{expr2}
    | ExprNotEq of expr * expr // {expr1}!={expr2}
    | ExprEq of expr * expr // {expr1}={expr2}

// ____________________________________________________________________________________________
//   arg: Arguments to Predicate
type arg =
    | ArgExpr of expr
    | ArgIgnoreVar of Identifier
    | ArgIgnore
    | ArgOutput of expr

// ____________________________________________________________________________________________
//   statement: Statement of a body
type statement =
    | StatementIfThenElse of position * expr * block * block // if {cond} then {...} else {...}
    | StatementIfThen of position * expr * block // if {cond} then {statement...}
    | StatementUnify of position * expr * expr // {expr1} = {expr2}
    | StatementYeild of position * expr // yeild {expr}
    | StatementCall of position * moduleIdentifier * arg list // Predicate call
    | StatementContinue of position * expr // continue {expr}
    | StatementMember of position * expr * expr
    | StatementAssert of position * expr // Assert {cond}
    | StatementReturn of position * expr // Return {expr}
    | StatementExport of position * expr * moduleIdentifier // Export {Identifier0} to {Identifier1}
    | StatementExportKey of position * expr * moduleIdentifier * expr // Export {I0} to {Id} with {key}
    | StatementSwitchNoDefault of position * expr * switchCase list // Switch cases, no default
    | StatementSwitchDefault of position * expr * switchCase list * block // Switch cases
```

```
    | StatementGenerate of position * expr * block // {expr} = Statement block with Yeild
    | StatementLoopDef of position * expr * Identifier * expr * block // {expr} = Init ({expr})
    | StatementStop of position // stop statement

//   switchCase: Switch case for switch case
and switchCase = expr * expr * block

//   block: Program definations included in the source file
and block = position * position * (statement list)

// ____________________________________________________________________________________________
//   typedef: Types used to represent expressions
type typeDef =
    | TypeUnion of union list
    | TypeTuple of typeDef list
    | TypeList of typeDef
    | TypeRecord of Map<Identifier, typeDef>
    | TypeReference of string * string
    | Number
    | String
    | Bool
    | Unknown

//   union: Used to represent discrimated union
and union = identifier * typeDef list

//   record: Used to represent a record pair
and record = Identifier * typeDef


// ____________________________________________________________________________________________
//   eval: Evaluated Expression
type eval =
    | EvalNumber of int64 // Number
    | EvalString of string // String
    | EvalBool of bool // String
    | EvalList of value list // List [1, 2, 3, ..]
    | EvalVar of Identifier // Variable
    | EvalTuple of value list // {identifier}({expr} ...)
    | EvalTerm of identifier * value list
    | EvalRecord of Map<Identifier, value>

//   value: Used to represent a type and eval pair
and value = typeDef * eval


// ____________________________________________________________________________________________
//   param: Functional paremet which could be typed or untyped
type param =
    | ParamUntyped of Identifier
    | ParamTyped of Identifier * typeDef

// ____________________________________________________________________________________________
//   defination: Program definations included in the source file
type defination =
```

```
    | DefineCond of Identifier * param list * block // Condition defination
    | DefineFunc of Identifier * param list * block // Function defination
    | DefinePred of Identifier * param list * block // Predicate defination
    | DefineStart of block // Start Predicate defination
    | DefineConst of param * expr // Constant of expr
    | DefineVar of Identifier * typeDef // Variable of type Module Identifier
    | DefineMap of Identifier * typeDef * typeDef // key, value type
    | DefineArray of Identifier * typeDef // List of type Module Identifier
    | DefineType of Identifier * typeDef // Types defination

// ____________________________________________________________________________________________
//   defination: Program definations included in the source file
type moduleDef = string * Identifier * defination list

// ____________________________________________________________________________________________
//   defination: Program definations included in the source file
type program = position * string * moduleDef list


// ____________________________________________________________________________________________
//   defination: Program definations included in the source file
type defination =
    | DefConstant of value
    | DefPredicate of param list * block // Predicate
    | DefFunction of param list * block // Function
    | DefCondition of param list * block // Condition
    | DefSingleVar of typeDef // of Type
    | DefMapVar of typeDef * typeDef // Map with key type and value type
    | DefListVar of typeDef
    | DefType of typeDef

// ____________________________________________________________________________________________
// defination: Program definations included in the source file
type varBag = Map<string, value>

// ____________________________________________________________________________________________
// unifyArge: Argument to Unify
type unifyArg =
| UnifyArgExpr of expr
| UnifyArgValue of value

// ____________________________________________________________________________________________
//   predResult: Result of Pred block ot statement evaluation
type predResult = varBag list option

// ____________________________________________________________________________________________
//   funcResult: Result of Function block ot statement evaluation
type funcResult =
    | FuncResultVars of varBag // Should not happen in Block!
    | FuncResultReturn of value

// ____________________________________________________________________________________________
//   seqResult: Result of Generate block ot statement evaluation
type seqResult = (value list) * ((varBag list) option) // Block will return value list
```

```
// ____________________________________________________________________________________
//   loopResult: Result of Generate block ot statement evaluation
type loopResult =
    | LoopResultContinue of value
    | LoopResultYield of value
    | LoopResultVars of varBag


*)
```

2

Tuesday, March 1, 2016 7:46 AM

```
% TO DO: adding String format to expr

% 1. Complete Defination (with exceptions)
% 2. Design Transformer(Old Program->New Program, ErrorList)
% 3. Executionaer
% 4. Builder
% 5. Main
% 6.

:- module defination.

:- interface.

:- import_module io, int, list, string, bool, char.

:- pred main(io::di, io::uo) is det.

:- type counter == int. % Unsigned Number

:- type number == int.  % Signed Number

:- type position == { string, counter, counter }. % (filename, linepos, counter)

:- type pos(T) ---> position(val::T, position).

:- type block(T) == list(pos(T)).

:- type operator1 ---> opNegate ; opNot.

:- type operator2 ---> opAdd ; opSub ; opMult ; opDiv ; opAppend ; opNeq ; opGt ;
                opGe ; opLt ; opLe ; opAnd ; opOr ; opEq ; opIn.

%:- type identifier ---> this(pos(string)) ; that(pos(string), pos(string)).
:- type identifier == pos(string).

:- type data.
:- type expr.
:- type pattern.
:- type pair.
:- type predType.
:- type ruleType.
:- type statement(_).
:- type typeDef.

:- type data --->
   dataChar(char);
   dataNum(number);
```

```
    dataBool(bool);
    dataFunction(list(string), expr);
    dataOperator1(operator1);
    dataOperator2(operator2);
    dataTuple(list(data));
    dataTerm(string, list(data)).

:- type expr --->
    exprIf(pos(expr), pos(expr), pos(expr)); % exprIf(Condition, Then, Else)
    exprData(data); % Data
    exprVar(identifier); % Variable
    exprCall(identifier, block(expr)); %  Func/ Pred call with all vars bound
    exprOp1(operator1, pos(expr));
    exprOp2(operator2, pos(expr), pos(expr));
    exprTerm(pos(string), block(expr)); % Term expression
    exprLet(list(pair), pos(expr));
    exprSeq(block(statement(pred)), pos(expr));
    exprEval(block(expr), pos(expr));
    exprSwitch(pos(expr), block(switchCaseExpr), pos(expr)). % Expr, Case list, Default Case

:- type pair == { pos(string),  pos(expr) }.

:- type switchCaseExpr == { pattern, pos(expr), pos(expr) }.

:- type pattern --->
    patternExpr(pos(expr));
    patternIgnore(position);
    patternUnify(pos(string));
    patternTerm(pos(string), block(pattern)).

:- type statement(T) --->
    statementAssert(pos(expr));
    statementStop;
    statementContinue;
    statementUnify(pos(pattern), pos(pattern));
    statementIf(pos(expr), block(statement(T)));
    statementIfElse(pos(expr), block(statement(T)), block(statement(T)));
    statementCall(pos(string), block(pattern));
    statementSwitch(pos(pattern), block(switchCase(T)), block(statement(T)));% Expr, Case list, Default Case
    statementMember(pos(pattern), pos(pattern));
    statementCustom(T).

:- type switchCase(T) == { pattern, pos(expr), block(statement(T)) }.

:- type predType ---> predStatementOr(list(block(statement(predType)))).

:- type ruleType --->
    ruleStatementAnd(list(block(statement(predType))));
    ruleStatementExport(identifier, block(string));
    ruleStatementDesignError(pos(expr));
    ruleStatementFileGenerate(pos(expr), pos(expr)).

:- type param --->
```

```
   paramTyped(pos(typeDef), pos(string));
   paramUntyped(pos(string)).

:- type coreDef --->
   defFunc(pos(string), block(param), pos(expr));
   defPred(pos(string), block(param), block(statement(predType)));
   defConst(pos(param), pos(data)).

:- type typeDef --->
   typeTuple(block(typeDef));
   typeCustom(pos(string), block(typeDef)).

:- type entity --->
   entityRule(block(statement(ruleType)));
   entityNamedRule(pos(string), block(param), block(statement(ruleType)));
   entityRuleHead(pos(string), block(typeDef)); % Block of Params without names
   entityCore(coreDef).

%:- type moduleDef == { pos(string), block(parse) }.

:- type program == list(moduleDef)


%:- type defKey == { string, int }.

:- implementation.

main(!IO) :- io.write_string("Hello, ", !IO), io.nl(!IO).

% :- type calc_info == map(string, int).


% type Transform = (Identifier list) * (Pred Statement block)

% type Transforms = Transform list

% type ProgramError = Position * pExpr * (Pred Statement block)

% type ProgramFile = Position * pExpr * pExpr * (Pred Statement block)

% type ProgramMap = Map<(string*int), Transforms Def>

% type Program =  ProgramMap * (ProgramError list) * (ProgramFile list)

% type TokenGroup = TokenGroup1 | TokenGroup2 | TokenGroup3

:-type Element
%   | ElementList of TokenGroup * Elements
%   | ElementIdentifier0 of string
%   | ElementIdentifier1 of string
%   | ElementString of string
%   | ElementAnonymous
%   | ElementUnify of string
```

```
%   | ElementNumber of integer
%   | ElementSymbols of string

% and Elements = (Position * Element) list

% type State = Map<string, Data>

% type States = State list option

% type sourceName = string list
% type sourceContent = string list
% type source = sourceName * sourceContent
% type build = source list
```

3

Tuesday, March 1, 2016 7:47 AM

```
(* ************************************************************
    DEFINATIONS.FS: All the definations for the program
  ************************************************************ *)

module Definations

type number =  uint64
type integer = int // int64

(* position(FileName, LineNo, Position): Position to track text *)
type Position = string * number * number

type unionCase<'T> = UnionCaseData of ('T) | UnionCaseTag of (Position * string) | UnionCaseNested of ('T unionCase list)

type block<'T> = (Position * 'T) list

(* identifier(Position, Name): String (with upper case) at position Position *)
type Identifier = (Position * string)

(*| - | not |*)
type UnaryOp = OpNegate | OpNot

(*| + | - | * | / | ++ | <> | > | >= or => | < | <= | =< | and | or | not | = | in |*)
type BinaryOp = OpAdd | OpSub | OpMult | OpDiv | OpAppend | OpNeq | OpGt | OpGe | OpLt | OpLe | OpAnd | OpOr |
OpEq | OpIn

let Symbols1 = ['+'; '-']
let Symbols2 = ["++"; "--"]

type Data =
   | DataChar of char
   | DataNum of int //eger
   | DataBool of bool
   | DataUnion of (Data unionCase list)
   | DataFunction of Identifier * pExpr
   | DataUnaryOp of UnaryOp
   | DataBinaryOp of BinaryOp

and Expr =
   | ExprIf of pExpr * pExpr *  pExpr (* If (Condition) Then and Else *)
   | EvalValue of Data
   | EvalVar of Identifier
   | ExprCall of Identifier * (Expr block) (* Func call / Predicate call with all vars bound *)
   | ExprUnaryOp of UnaryOp * pExpr
   | ExprUnion of (pExpr unionCase list)
   | ExprLet of ((Identifier * pExpr) list) * pExpr
   | ExprSeq of (Pred Statement block) * pExpr
   | ExprEval of pExpr * (Expr block)
   | ExprBinaryOp of BinaryOp * pExpr * pExpr
   | ExprSwitch of pExpr * ((pExpr * pExpr) list) * pExpr
```

```
and pExpr = Position * Expr

and Pattern = // %%% TO DO : Should have pPattern %%%
   | PatternVar of pExpr
   | PatternIgnore
   | PatternUnion of (pPattern unionCase list)
   | PatternUnify of Identifier

and pPattern = Position * Pattern

and SwitchCase<'T> = Pattern * (pExpr option) * ('T Statement block)

and Statement<'T> =
   | StatementAssert of pExpr
   | StatementStop
   | StatementUnify of pPattern * pPattern
   | StatementIf of pExpr * ('T Statement block)
   | StatementIfElse of pExpr * ('T Statement block) * ('T Statement block)
   | StatementCall of Identifier * (pPattern list)
   | StatementSwitch of pPattern * ('T SwitchCase block)
   | StatementMember of pPattern *  pPattern (* Member stuff *)
   | StatementCustom of ('T)

and Pred = PredStatementOr of (Pred Statement block list)


type Rule =
   | RuleStatementAnd of ((Rule Statement block) list)
   | RuleStatementExport of Identifier * (Identifier list)
   | RuleStatementDesignError of pExpr
   | RuleStatementFileGenerate of pExpr * pExpr


(* def: Definations for the Program *)
type Def<'T> =
   | DefFunc of Identifier * (Identifier list) * pExpr
   | DefPred of Identifier * (Identifier list) * (Pred Statement block)
   | DefConst of Identifier * (Position * Data) (* DefConst(Name, (Position, Constant Value) *)
   | DefCustom of ('T)
  (* %%%% TO DO: Add Type here %%%%% *)


type Parse =
   | ParseRule of (Rule Statement block)
   | ParseNamedRule of Identifier * (Identifier list) * (Rule Statement block)
   | ParseRuleHead of Identifier * number (* DefineHead(Name, Arity): Predicate define with Name and Arity *)

type Transform = (Identifier list) * (Pred Statement block)

type Transforms = Transform list

type ProgramError = Position * pExpr * (Pred Statement block)

type ProgramFile = Position * pExpr * pExpr * (Pred Statement block)
```

```
type ProgramMap = Map<(string*int), Transforms Def>


type Program =  ProgramMap * (ProgramError list) * (ProgramFile list)


type TokenGroup = TokenGroup1 | TokenGroup2 | TokenGroup3


type Element =
   | ElementList of TokenGroup * Elements
   | ElementIdentifier0 of string
   | ElementIdentifier1 of string
   | ElementString of string
   | ElementAnonymous
   | ElementUnify of string
   | ElementNumber of integer
   | ElementSymbols of string

and Elements = (Position * Element) list

type State = Map<string, Data>

type States = State list option

type sourceName = string list
type sourceContent = string list
type source = sourceName * sourceContent
type build = source list
```

4

Tuesday, March 1, 2016 7:47 AM

```
% DECLARATIONS FOR PARSER

GLOBAL DOMAINS
        identifier = identifier(string, token)
        identifierlist = identifier*
        number = number(integer, token)
        variable = variable(string, token) ; str(string, token)
        str = str(string, token)
        %list = list(exprlist)
        data = str(string, token) ; list(datalist) ; number(integer, token)
        datalist = data*
        member = member(identifier, expr)
        minorlist = member*
        majorlist = minorlist*
        condition =
                member(expr, expr) ;
                cond_if1(condition, condition) ;
                cond_if2(condition, condition, condition) ;
                cond_and(condition, condition) ;
                cond_or(condition, condition) ;
                lt(expr, expr) ;
                le(expr, expr) ;
                gt(expr, expr) ;
                ge(expr, expr) ;
                eq(expr, expr) ;
                ne(expr, expr) ;
                call(identifier, exprlist) ;
                cond_not(condition)

        expr =
                expr_if(condition, expr, expr) ;
                add(expr, expr) ;
                sub(expr, expr) ;
                multiply(expr, expr) ;
                modulus(expr, expr) ;
                divide(expr, expr) ;
                concat(expr, expr) ;
                join(expr, expr) ;
                add_list(expr, expr) ;
                generate(expr, expr) ;
                map1(majorlist, expr) ;
                map2(majorlist, expr, condition) ;
                enum(expr) ;
                call(identifier, exprlist) ;
                mcall(identifier, identifier, exprlist) ;
                number(integer, token) ;
                variable(string, token) ;
                str(string, token) ;
```

```
        list(exprlist)

exprlist = expr*
port = expr(expr) ; port(identifier)
portlist = port*
mstatement =
        begin ;
        alt ;
        end ;
        function(identifier, identifierlist) ;
        condition(identifier, identifierlist) ;
        export(identifier, expr) ;
        import(identifier, identifier) ;
        define(identifier) ;
        component(identifier, identifierlist, identifierlist) ;
        compose(identifier, identifierlist, identifierlist) ;
        slot(identifier, identifierlist, identifierlist) ;
        multislot(identifier, identifierlist, identifierlist) ;
        statement_or(mstatement, mstatement) ;
        statement_and(mstatement, mstatement) ;
        component1(identifier, identifier) ;
        component2(identifier, identifier, identifier) ;
        multibind(identifier, identifier, identifier, portlist, portlist) ;
        bind(identifier, identifier, identifier, portlist, portlist) ;
        member(identifier, expr) ;
        statement_if(condition);
        statement_else ;
        statement_if_then(condition, mstatement) ;
        statement_if_then_else(condition, mstatement, mstatement) ;
        set(identifier, expr) ;
        mset1(identifierlist, identifier, exprlist) ;
        mset2(identifierlist, identifier, identifier, exprlist) ;
        statement_while1(condition) ;
        statement_while2(condition, mstatement) ;
        statement_do ;
        statement_for1(majorlist) ;
        statement_for2(majorlist, mstatement) ;
        cond(condition) ;
        module(identifier) ;
        build(identifier) ;
        bfile(variable, identifier, exprlist, exprlist) ;
        filecopy(variable, variable, exprlist) ;
        bfolder(variable, exprlist) ;
        binclude(identifier, exprlist)

pstatement =
        begin ;
        alt ;
        end  ;
        pinclude(str) ;
        target(identifier) ;
        project(identifier) ;
        set(identifier, data) ;
```

```
        report(identifier, identifier, identifier)


statement = m(string, integer, mstatement) ; p(string, integer, pstatement)
statementslist = statement*

% ***** Project Structure ******
mstructures = string*
target = target(identifier, statementslist, statementslist)
targetlist = target*
% PStructure : Name, FileName, Module Structures, taget lists
pstructure = project(identifier, string,  text, targetlist)
```

5

Tuesday, March 1, 2016 7:48 AM

```
(*
program = pos{def}*.

   def =
      defFunc(pos{string} Name, pos{string}* Params, pos{statement}* Body);
      defPred(pos{string} Name, pos{string}* Params, pos{statement}* Body).

   param =
      paramConst(pos{expr});
      paramUnify(pos{expr});
      paramIgnore.

   statement =
      statementIf(pos{expr} Condition, pos{statement}* Body);
      statementIfElse(pos{expr} Condition, pos{statement}* BodyThen, pos{statement}* BodyElse);
      statementUnify(pos{expr} ExprLeft, pos{expr} ExprRight);
      statementCall(pos{string} PredName, pos{param};
      %statementAssert(pos{expr});
      %statementContinue(pos{expr});
      statementReturn(pos{expr});
      %statementYield(pos{expr});
      statementStop.

   expr =
      exprIf(pos{expr} Condition, pos{expr}* ExprThen, pos{expr}* ExprElse);
      exprStr(pos{string});
      exprNum(pos{integer});
      exprNum(pos{bool});
      exprNegate(pos{expr});
      exprList(pos{expr}* );
      exprVar(pos{string});
      exprAdd(pos{expr},pos{expr});
      exprSub(pos{expr},pos{expr});
      exprMult(pos{expr},pos{expr});
      exprDiv(pos{expr},pos{expr});
      exprConcat(pos{expr},pos{expr});
      exprAppend(pos{expr},pos{expr});
      exprJoin(pos{expr},pos{expr});
      exprCall(pos{string},pos{expr}* ).
      exprEq(pos{expr},pos{expr});
      exprNeq(pos{expr},pos{expr});
      exprGt(pos{expr},pos{expr});
      exprLt(pos{expr},pos{expr});
      exprGe(pos{expr},pos{expr});
      exprLe(pos{expr},pos{expr});
      exprAnd(pos{expr},pos{expr});
      exprOr(pos{expr},pos{expr});
      exprNot(pos{expr},pos{expr});
      exprIn(pos{expr}).

// *******************************************************
//   DEFINATIONS.FS: All the definations for the program
```

```
// ********************************************************
//
module Defination

// ____________________________________________________________________________________________
//   position: Position to track tokens
type position = string * uint32 * uint32 // FileName, LineNo, Position

// ____________________________________________________________________________________________
//   Identifier: Consists of string chat, one with small letter - other with large
type Identifier = position * string // String should be upper identifier

type identifier = position * string // String should be lower identifier

// ____________________________________________________________________________________________
//   moduleIdentifier: Identifier to track items in modules
type moduleIdentifier =
   | IdentifierThis of Identifier
   | IdentifierThat of Identifier * Identifier

// ____________________________________________________________________________________________
//   expr: Expression
type expr =
   | ExprNumber of position * int64 // Number
   | ExprString of position * string // String
   | ExprBool of position * bool // String
   | ExprList of position * expr list // List [1, 2, 3, ..]
   | ExprTailList of position * expr list * expr // [1, 2, 3 | X]
   | ExprVar of moduleIdentifier // Variable
   | ExprTuple of position * expr list // {identifier}({expr} ...)
   | ExprTerm of identifier * expr list
   | ExprFunCall of moduleIdentifier * expr list // {identifier}({expr} ...)
   | ExprRecord of position * Map<Identifier, expr>
   | ExprMapCall of moduleIdentifier * expr
   | ExprAdd of expr * expr // {expr1} + {expr2}
   | ExprSub of expr * expr // {expr1} - {expr2}
   | ExprMult of expr * expr // {expr1} * {expr2}
   | ExprDiv of expr * expr // {expr1} / {expr2}
   | ExprMod of expr * expr // {expr1} % {expr2}
   | ExprConcat of expr * expr // String concatenation
   | ExprJoin of expr * expr // {expr1} is head and {expr2} is tail
   | ExprAddList of expr * expr // Adding two list
   | ExprNegate of position * expr // -{expr}
   | ExprIfThen of position * expr * expr // if {cond} then {expr1} else {expr2}
   | ExprIfThenElse of position * expr * expr * expr // if {cond} then {expr1} else {expr2}
   | ExprMemberCheck of expr * expr // {expr1} in {expr2}
   | ExprAnd of expr * expr // {cond1} and {cond2}
   | ExprOr of expr * expr // {cond1} or {cond2}
   | ExprNot of position * expr // not {cond}
   | ExprGtEq of expr * expr // {expr1}>={expr2}
   | ExprGt of expr * expr // {expr1}>{expr2}
   | ExprLtEq of expr * expr // {expr1}<={expr2}
   | ExprLt of expr * expr // {expr1}<{expr2}
   | ExprNotEq of expr * expr // {expr1}!={expr2}
   | ExprEq of expr * expr // {expr1}={expr2}

// ____________________________________________________________________________________________
//   arg: Arguments to Predicate
```

```
type arg =
    | ArgExpr of expr
    | ArgIgnoreVar of Identifier
    | ArgIgnore
    | ArgOutput of expr

// ____________________________________________________________________________________
//   statement: Statement of a body
type statement =
    | StatementIfThenElse of position * expr * block * block // if {cond} then {...} else {...}
    | StatementIfThen of position * expr * block // if {cond} then {statement...}
    | StatementUnify of position * expr * expr // {expr1} = {expr2}
    | StatementYeild of position * expr // yeild {expr}
    | StatementCall of position * moduleIdentifier * arg list // Predicate call
    | StatementContinue of position * expr // continue {expr}
    | StatementMember of position * expr * expr
    | StatementAssert of position * expr // Assert {cond}
    | StatementReturn of position * expr // Return {expr}
    | StatementExport of position * expr * moduleIdentifier // Export {Identifier0} to {Identifier1}
    | StatementExportKey of position * expr * moduleIdentifier * expr // Export {I0} to {Id} with {key}
    | StatementSwitchNoDefault of position * expr * switchCase list // Switch cases, no default
    | StatementSwitchDefault of position * expr * switchCase list * block // Switch cases
    | StatementGenerate of position * expr * block // {expr} = Statement block with Yeild
    | StatementLoopDef of position * expr * Identifier * expr * block // {expr} = Init ({expr})
    | StatementStop of position // stop statement

//   switchCase: Switch case for switch case
and switchCase = expr * expr * block

//   block: Program definations included in the source file
and block = position * position * (statement list)

// ____________________________________________________________________________________
//   typedef: Types used to represent expressions
type typeDef =
    | TypeUnion of union list
    | TypeTuple of typeDef list
    | TypeList of typeDef
    | TypeRecord of Map<Identifier, typeDef>
    | TypeReference of string * string
    | Number
    | String
    | Bool
    | Unknown

//   union: Used to represent discrimated union
and union = identifier * typeDef list

//   record: Used to represent a record pair
and record = Identifier * typeDef


// ____________________________________________________________________________________
//   eval: Evaluated Expression
type eval =
    | EvalNumber of int64 // Number
    | EvalString of string // String
    | EvalBool of bool // String
```

```
    | EvalList of value list // List [1, 2, 3, ..]
    | EvalVar of Identifier // Variable
    | EvalTuple of value list // {identifier}({expr} ...)
    | EvalTerm of identifier * value list
    | EvalRecord of Map<Identifier, value>

//   value: Used to represent a type and eval pair
and value = typeDef * eval


// ____________________________________________________________________________________________
//   param: Functional paremet which could be typed or untyped
type param =
    | ParamUntyped of Identifier
    | ParamTyped of Identifier * typeDef

// ____________________________________________________________________________________________
//   defination: Program definations included in the source file
type defination =
    | DefineCond of Identifier * param list * block // Condition defination
    | DefineFunc of Identifier * param list * block // Function defination
    | DefinePred of Identifier * param list * block // Predicate defination
    | DefineStart of block // Start Predicate defination
    | DefineConst of param * expr // Constant of expr
    | DefineVar of Identifier * typeDef // Variable of type Module Identifier
    | DefineMap of Identifier * typeDef * typeDef // key, value type
    | DefineArray of Identifier * typeDef // List of type Module Identifier
    | DefineType of Identifier * typeDef // Types definations

// ____________________________________________________________________________________________
//   defination: Program definations included in the source file
type moduleDef = string * Identifier * defination list

// ____________________________________________________________________________________________
//   defination: Program definations included in the source file
type program = position * string * moduleDef list


// ____________________________________________________________________________________________
//   defination: Program definations included in the source file
type definationType =
    | DefConstant of value
    | DefPredicate of param list * block // Predicate
    | DefFunction of param list * block // Function
    | DefCondition of param list * block // Condition
    | DefSingleVar of typeDef // of Type
    | DefMapVar of typeDef * typeDef // Map with key type and value type
    | DefListVar of typeDef
    | DefType of typeDef

// ____________________________________________________________________________________________
// defination: Program definations included in the source file
type varBag = Map<string, value>

// ____________________________________________________________________________________________
// unifyArge: Argument to Unify
type unifyArg =
| UnifyArgExpr of expr
```

```
| UnifyArgValue of value

// _____________________________________________________________________________________
//   predResult: Result of Pred block ot statement evaluation
type predResult = varBag list option

// _____________________________________________________________________________________
//   funcResult: Result of Function block ot statement evaluation
type funcResult =
    | FuncResultVars of varBag // Should not happen in Block!
    | FuncResultReturn of value

// _____________________________________________________________________________________
//   seqResult: Result of Generate block ot statement evaluation
type seqResult = (value list) * ((varBag list) option) // Block will return value list

// _____________________________________________________________________________________
//   loopResult: Result of Generate block ot statement evaluation
type loopResult =
    | LoopResultContinue of value
    | LoopResultYield of value
    | LoopResultVars of varBag

*)
```

6

Tuesday, March 1, 2016 7:48 AM

```
// *********************************************************
//   DEFINATIONS.FS: All the definations for the program
// *********************************************************
//
module Definations

// ____________________________________________________________________________________________
//   position: Position to track tokens
type position = string * uint32 * uint32 // FileName, LineNo, Position

// ____________________________________________________________________________________________
//   Identifier: Consists of string chat, one with small letter - other with large
type Identifier = position * string // String should be upper identifier

type identifier = position * string // String should be lower identifier

// ____________________________________________________________________________________________
//   moduleIdentifier: Identifier to track items in modules
type moduleIdentifier =
   | IdentifierThis of Identifier
   | IdentifierThat of Identifier * Identifier

// ____________________________________________________________________________________________
//   expr: Expression
type expr =
   | ExprNumber of position * int64 // Number
   | ExprString of position * string // String
   | ExprBool of position * bool // String
   | ExprList of position * expr list // List [1, 2, 3, ..]
   | ExprTailList of position * expr list * expr // [1, 2, 3 | X]
   | ExprVar of moduleIdentifier // Variable
   | ExprTuple of position * expr list // {identifier}({expr} ...)
   | ExprTerm of identifier * expr list
   | ExprFunCall of moduleIdentifier * expr list // {identifier}({expr} ...)
   | ExprRecord of position * Map<Identifier, expr>
   | ExprMapCall of moduleIdentifier * expr
   | ExprAdd of expr * expr // {expr1} + {expr2}
   | ExprSub of expr * expr // {expr1} - {expr2}
   | ExprMult of expr * expr // {expr1} * {expr2}
   | ExprDiv of expr * expr // {expr1} / {expr2}
   | ExprMod of expr * expr // {expr1} % {expr2}
   | ExprConcat of expr * expr // String concatenation
   | ExprJoin of expr * expr // {expr1} is head and {expr2} is tail
   | ExprAddList of expr * expr // Adding two list
   | ExprNegate of position * expr // -{expr}
   | ExprIfThen of position * expr * expr // if {cond} then {expr1} else {expr2}
   | ExprIfThenElse of position * expr * expr * expr // if {cond} then {expr1} else {expr2}
   | ExprMemberCheck of expr * expr // {expr1} in {expr2}
   | ExprAnd of expr * expr // {cond1} and {cond2}
   | ExprOr of expr * expr // {cond1} or {cond2}
   | ExprNot of position * expr // not {cond}
```

```
    | ExprGtEq of expr * expr // {expr1} >= {expr2}
    | ExprGt of expr * expr // {expr1} > {expr2}
    | ExprLtEq of expr * expr // {expr1} <= {expr2}
    | ExprLt of expr * expr // {expr1} < {expr2}
    | ExprNotEq of expr * expr // {expr1} != {expr2}
    | ExprEq of expr * expr // {expr1} = {expr2}

// ____________________________________________________________________________________________
//   arg: Arguments to Predicate
type arg =
    | ArgExpr of expr
    | ArgIgnoreVar of Identifier
    | ArgIgnore
    | ArgOutput of expr

// ____________________________________________________________________________________________
//   statement: Statement of a body
type statement =
    | StatementIfThenElse of position * expr * block * block // if {cond} then {...} else {...}
    | StatementIfThen of position * expr * block // if {cond} then {statement...}
    | StatementUnify of position * expr * expr // {expr1} = {expr2}
    | StatementYeild of position * expr // yeild {expr}
    | StatementCall of position * moduleIdentifier * arg list // Predicate call
    | StatementContinue of position * expr // continue {expr}
    | StatementMember of position * expr * expr
    | StatementAssert of position * expr // Assert {cond}
    | StatementReturn of position * expr // Return {expr}
    | StatementExport of position * expr * moduleIdentifier // Export {Identifier0} to {Identifier1}
    | StatementExportKey of position * expr * moduleIdentifier * expr // Export {I0} to {Id} with {key}
    | StatementSwitchNoDefault of position * expr * switchCase list // Switch cases, no default
    | StatementSwitchDefault of position * expr * switchCase list * block // Switch cases
    | StatementGenerate of position * expr * block // {expr} = Statement block with Yeild
    | StatementLoopDef of position * expr * Identifier * expr * block // {expr} = Init ({expr})
    | StatementStop of position // stop statement

//   switchCase: Switch case for switch case
and switchCase = expr * expr * block

//   block: Program definations included in the source file
and block = position * position * (statement list)

// ____________________________________________________________________________________________
//   typedef: Types used to represent expressions
type typeDef =
    | TypeUnion of union list
    | TypeTuple of typeDef list
    | TypeList of typeDef
    | TypeRecord of Map<Identifier, typeDef>
    | TypeReference of string * string
    | Number
    | String
    | Bool
    | Unknown

//   union: Used to represent discrimated union
and union = identifier * typeDef list
```

```
//   record: Used to represent a record pair
and record = Identifier * typeDef


// ____________________________________________________________________________________
//   eval: Evaluated Expression
type eval =
    | EvalNumber of int64 // Number
    | EvalString of string // String
    | EvalBool of bool // String
    | EvalList of value list // List [1, 2, 3, ..]
    | EvalVar of Identifier // Variable
    | EvalTuple of value list // {identifier}({expr} ...)
    | EvalTerm of identifier * value list
    | EvalRecord of Map<Identifier, value>

//   value: Used to represent a type and eval pair
and value = typeDef * eval


// ____________________________________________________________________________________
//   param: Functional paremet which could be typed or untyped
type param =
    | ParamUntyped of Identifier
    | ParamTyped of Identifier * typeDef

// ____________________________________________________________________________________
//   defination: Program definations included in the source file
type defination =
    | DefineCond of Identifier * param list * block // Condition defination
    | DefineFunc of Identifier * param list * block // Function defination
    | DefinePred of Identifier * param list * block // Predicate defination
    | DefineStart of block // Start Predicate defination
    | DefineConst of param * expr // Constant of expr
    | DefineVar of Identifier * typeDef // Variable of type Module Identifier
    | DefineMap of Identifier * typeDef * typeDef // key, value type
    | DefineArray of Identifier * typeDef // List of type Module Identifier
    | DefineType of Identifier * typeDef // Types definations

// ____________________________________________________________________________________
//   defination: Program definations included in the source file
type moduleDef = string * Identifier * defination list

// ____________________________________________________________________________________
//   defination: Program definations included in the source file
type program = position * string * moduleDef list


// ____________________________________________________________________________________
//   defination: Program definations included in the source file
type definationType =
    | DefConstant of value
    | DefPredicate of param list * block // Predicate
    | DefFunction of param list * block // Function
    | DefCondition of param list * block // Condition
    | DefSingleVar of typeDef // of Type
    | DefMapVar of typeDef * typeDef // Map with key type and value type
```

```
    | DefListVar of typeDef
    | DefType of typeDef

// ____________________________________________________________________________________________
// defination: Program definations included in the source file
type varBag = Map<string, value>

// ____________________________________________________________________________________________
// unifyArge: Argument to Unify
type unifyArg =
| UnifyArgExpr of expr
| UnifyArgValue of value

// ____________________________________________________________________________________________
//   predResult: Result of Pred block ot statement evaluation
type predResult = varBag list option

// ____________________________________________________________________________________________
//   funcResult: Result of Function block ot statement evaluation
type funcResult =
    | FuncResultVars of varBag // Should not happen in Block!
    | FuncResultReturn of value

// ____________________________________________________________________________________________
//   seqResult: Result of Generate block ot statement evaluation
type seqResult = (value list) * ((varBag list) option) // Block will return value list

// ____________________________________________________________________________________________
//   loopResult: Result of Generate block ot statement evaluation
type loopResult =
    | LoopResultContinue of value
    | LoopResultYield of value
    | LoopResultVars of varBag
```