

Programmer guide for The Core Parser

(updated on 16/9/2001 by Avi Magid)
(updated on 7/10/2001 by Avi Magid and Shahar Dag)

Warning; since this file is very old, its content is a little bit out dated. Please refer the content of this file only as general guidelines, address the code for the actual data

Data Structure	4
CoreProgram struct	4
TypeSet struct	4
TypeNode struct	4
TypeDefinition struct	4
Expression struct	5
Set struct	5
SetNode struct	5
EnumSet struct	5
EnumNode struct	6
DefineVarSet struct	6
DefineVar struct	6
VariableSet struct	6
Variable struct	6
TypeName struct	6
ModuleSet struct	7
Module struct	7
FormalParameterSet struct	7
FormalParameter struct	7
ModCombin struct	8
PairSet struct	8
PairNode struct	8
Pair struct	8
RenameSet struct	8
RenameNode struct	9
ActualParameterSet struct	9
ActualParameter struct	9
TransitionSet struct	9
TransitionNode struct	9
Assign struct	9
AssignNode struct	10
Header Files	11
Parserglobals.h	11
coreprogram.h	12
typeset.h	13
typenode.h	14
typedefinition.h	14
enumset.h	16
enumnode.h	16
expression.h	17
set.h	20
setnode.h	21
definevarset.h	22
definevar.h	23
variableset.h	24
variable.h	25
typename.h	26
moduleset.h	27
module.h	28
formalparameterset.h	29

formalparameter.h.....	29
modcombin.h.....	30
pairset.h.....	31
pairnode.h.....	31
pair.h.....	32
renameset.h.....	33
renamenode.h.....	33
actualparameterset.h.....	34
actualparameter.h.....	34
transitionset.h.....	35
transitionnode.h.....	36
assign.h.....	37
assignnode.h.....	38

Data Structure

(A Top-Down Description)

CoreProgram struct

(files: coreprogram.h, coreprogram.c)

TypeSet*	Types	Global types of the Core program
DefineVarSet*	Defines	Global definitions of the Core program
DefineVarSet*	Constants	Global constants of the Core program
VariableSet*	GlobalVariables	Global variables of the Core program.
Int	HoldPrevious	0 – HOLD_PREVIOUS in off. 1 – HOLD_PREVIOUS is on.
ModuleSet*	Modules	Modules of the Core program.

TypeSet struct

(files: typeset.h, typeset.c)

TypeNode*	TypeList	Set of types.
Int	Size	Number of types in set.

TypeNode struct

(files: typenode.h, typenode.c)

TypeDefinition*	TypeDef	Definition of the type.
TypeNode*	Next	Next type in set.

TypeDefinition struct

(files: typedefinition.h, typedefinition.c)

Int	Type	TYPE_NAME, TYPE_RANGE, TYPE_ENUM, TYPE_SCALARSET
Char*	TypeName	Name of the type – Identifier of the type.
Char*	Name	Just another identifier to an already defined type.
Expression*	llow, lhigh	[llow, lhigh] define the range.
Expression*	Scalar	[0 .. Scalar-1] define the ScalarSet.
EnumSet*	EnumList	Group of names define the enumeration.

Expression struct

(files: expression.h, expression.c)

Int	Op	NOP, OP_PLUS, OP_MINUS, OP_MULT, OP_DIV, OP_MOD, OP_EQ, OP_NOTEQ, OP_GE, OP_LE, OP_GT, OP_LT, OP_AND, OP_OR, OP_ARRAY, OP_NOT, OP_NEXT, OP_BR, OP_UMINUS
Int	Type	E_INTEGER, E_VARIABLE, E_SET, E_BOOLEAN, E_UNARY, E-BINARY, E-RANGE
Expression*	Left	
Expression*	Right	
Expression*	Father	
Int	LeafInteger	Integer value.
Int	LeafBoolean	Boolean value.
Char*	LeafVariable	Variable name
Set*	LeafSet	Group of variables
Int	llow, lhigh	Range of [llow, lhigh]

Set struct

(files: set.h, set.c)

SetNode*	SetList	Set of expressions
Int	Size	Number of elements in set.

SetNode struct

(files: setnode.h, setnode.c)

Expression*	Exp	Expression presenting the element.
SetNode*	Next	Next SetNode in group.

EnumSet struct

(files: enumset.h, enumset.c)

EnumNode*	EnumList	Set of the elements in the enumeration.
Int	Size	Number of element in the enumeration.

EnumNode struct

(files: enumnode.h, enumnode.c)

Char*	Value	Name of element.
EnumNode	Next	Next element in enumeration.

DefineVarSet struct

(files: definevarset.h, definevarset.c)

DefineVar*	DefineVarList	Set of variable definitions.
Int	Size	Number of variables.

DefineVar struct

(files: definevar.h, definevar.c)

Char*	DefineName	Identifier of the defined.
Expression*	DefineValue	The expression being abbreviated.
DefineVar*	Next	Next define in set.

VariableSet struct

(files: variableset.h, variableset.c)

Variable*	VarList	Group of variables
Int	Size	Number of variables.

Variable struct

(files: Variable.h, Variable.c)

Char*	VarName	Variable name.
TypeName*	VarType	Variable type.
Expression*	Init	Initial value of the variable.

TypeName struct

(files: typename.h, typename.c)

Int	Type	TYPE_NAME_BOOL, TYPE_NAME_INTEGER, TYPE_NAME_NAME, TYPE_NAME_ARRAY
Char*	Name	Name of the type.
TypeName*	ArrayType	TypeName of the array type.
Expression*	ArrayLength	Length of the array.

ModuleSet struct

(files: moduleset.h, moduleset.c)

Module*	ModuleList	Set of modules.
Int	Size	Number of modules.

Module struct

(files: module.h, module.c)

Char*	Name	Name of module.
Char*	Comment	Comment written in the module.
VariableSet*	Vars	Local variables in the module.
DefineVarSet*	Defines	Definitions in the module.
FormalParamaterSet*	Params	Parameters of the module.
Int	Type	MOD_COMBI, MOD_TRANS
ModCombin*	Combination	Combination of modules by a-synchronization, synchronization, or partial synchronization.
TransitionSet*	Transitions	Group of transitions.
Module*	Next	Next module in group.

FormalParameterSet struct

(files: formalparameterset.h, formalparameterset.c)

FormalParameter*	FormalParameterList	Parameters defined in the module definition.
Int	Size	Number of parameters.

FormalParameter struct

(files: formalparameter.h, formalparameter.c)

Char*	Name	Name of parameter.
TypeName*	Type	Type of the parameter.
FormalParameter*	Next	Next parameter in set.

ModCombin struct

(files: modcombin.h, modcombin.c)

Int	Type	COMBIN_MODIN, COMBIN_SYNC, COMBIN_ASYNC, COMBIN_PART
ModCombin*	Left	
ModCombin*	Right	
ModCombin*	Father	
PairSet*	SyncTransitions	Pairs of transitions defining a partial synchronization.
RenameSet*	Renames	Rename of transitions or pairs.
Char*	ModuleName	Name of module being instantiated.
ActualParameterSet*	Params	Actual parameters of the module instantiation.

PairSet struct

(files: pairset.h, pairset.c)

PairNode*	PairList	Set of pairs.
Int	Size	Number of pairs.

PairNode struct

(files: pairnode.h, pairnode.c)

Pair*	Npair	Pair
PairNode*	Next	Next pair in set.

Pair struct

(files: pair.h, pair.c)

Int	Type	PAIR, TRANSITIONNAME
Pair*	Father	
Pair*	Left	
Pair*	Right	
Char*	TransitionName	Name of transition being paired.

RenameSet struct

(files: renameset.h, renameset.c)

RenameNode*	Renames	Group of renames.
Int	Size	Number of renames.

RenameNode struct

(files: renamenode.h, renamenode.c)

Pair*	PairRenamed	Pair of transition name being renamed.
Char*	Name	The given new name.
RenameNode*	Next	Next rename in group.

ActualParameterSet struct

(files: actualparameterset.h, actualparameterset.c)

ActualParameter*	ActualParameterList	Parameters defined in the module instantiation.
Int	Size	Number of parameters.

ActualParameter struct

(files: actualparameter.h, actualparameter.c)

Expression*	Value	Value given to a parameter during the instantiation.
ActualParameter*	Next	Next parameter in group.

TransitionSet struct

(files: transitionset.h, transitionset.c)

TransitionNode*	TransitionList	Set of transitions.
Int	Size	Number of transitions.

TransitionNode struct

(files: transitionnode.h, transitionnode.c)

Char*	Name	Name of the transition.
Expression*	Enable	Enable condition.
Assign*	Assigns	Assignments of the transition.
Expression*	Relation	Relation condition.
TransitionNode*	Next	Next transition in set of transitions.

Assign struct

(files: assign.h, assign.c)

AssignNode*	AssignList	List of assignments.
Int	Size	Number of assignments.

AssignNode struct

(files: assignnode.h, assignnode.c)

Expression*	VariableName	Name of variable into which the new value is put.
Expression*	Expr	The expression presenting the value put into the variable.
AssignNode*	Next	Next assignment in the assign.

Header Files

Parserglobals.h

Any define, that you wish to declare throughout the parsing process, I recommend to inserted here. All files of the parsing stage include this H-file directly or indirectly.

```
/*  
    MEMORY_ALLOC_ERR is a string, which is present whenever an allocation of  
    memory has failed.
```

```
*/  
#define MEMORY_ALLOC_ERR "Memory Allocation Error."
```

I suggest that in case you change any of the parser files, in particular the YACC file, then turn on the CORE_PARSER_DEBUG to make sure that the result core program object is identical to the Core source code.

```
/*  
    CORE_PARSER_DEBUG is a flag, which enables the user to debug the YACC  
    behavior. If flag holds 1 then for each grammar rule the YACC would present the  
    result object, i.e. the object $$.
```

```
*/  
#define CORE_PARSER_DEBUG 1
```

```
/*  
    ParserError - Prints the proper error message according to the parameter and exits the  
    program.
```

```
*/  
void ParserError(char* errorString);
```

This is the main function of the Core Parser. I suggest keeping it the main function in your translation program. After the yyparse() function call, call your function which begins your translation.

```
/*  
    The Core Parser starts here. For now the main function runs the yyparse() function  
    which does the parsing and fills up the Core program object. The function prints a  
    success message if parsing was successful.
```

```
*/  
int main();
```

coreprogram.h

The Core Program struct defines a complex (but not a complicated) data structure which holds all the information of the Core program. You can say that this information is “Raw” – to begin the translation you must “Cook” the information, i.e. extract the information you need.

There is only one object of this struct, which is global. A pointer to this object is declared in “parserglobals.c”, and is called: CoreDataStructure.

Understand the structure of this object – Your translation to the target language begins from this object.

```
/*
    Core Program Struct. An object of this struct will contain all the information of the
    Core. There are no omissions from the Core program code.
*/
typedef struct coreprogram
{
    TypeSet*           Types;
    DefineVarSet*     Defines;
    DefineVarSet*     Constants;
    VariableSet*      GlobalVariables;
    int                HoldPrevious;
    ModuleSet*        Modules;
} CoreProgram;

/*    Allocation of a Core Program Struct Object. */
CoreProgram* CreateCoreProgram();

void UpdateCoreTypes           (CoreProgram*, TypeSet*);
void UpdateCoreDefines         (CoreProgram*, DefineVarSet*);
void UpdateCoreConstants      (CoreProgram*, DefineVarSet*);
void UpdateCoreVariables      (CoreProgram*, VariableSet*);
void UpdateCoreHoldPrevious    (CoreProgram*, int);
void UpdateCoreModules        (CoreProgram*, ModuleSet*);
```

All Print functions are debug functions. You can change them in any way you like to suit your needs. There is no other component in the code which is dependent on it, just don't delete these functions – otherwise the code would not compile.

```
void PrintCoreProgram(CoreProgram*);
```

You may change the implementation of the following structures to any implementation you want. Only keep the original interface. You may expand the interface in any way you like.

Just make sure there is no name conflict - All functions in Core Parse are global.

All data structures are based on linked lists.

typeset.h

```
/*      A set of type nodes implemented using a linked list.      */
typedef struct typeset
{
    TypeNode*  TypeList;
    Int        size;
} TypeSet;
```

The data structure that holds the type definitions is defined in typeset.h and typenode.h. If you want to change the structure into any other complex, focus on these files: typeset.h, typeset.c, typenode.h, typenode.c.

The data structure, which holds the definition of the type, is located in typedefinition.h and typedefinition.c. An object of this struct is located in each typenode struct.

For example: Let say you want to create a linked list, which is ordered by a key. The key would dependent on the type definition. What do you do?

- 1. Into typedefinition.h introduce a function ExtractTypeDefinitionKey(TypeDefinition t), which would analyze the type definition structure and create a key value.*
- 2. Into typenode.h introduce a variable TypeNodeKey, which would hold the Type Definition value.*
- 3. Change the InsertTypeNode(TypeSet*, TypeNode*) to insert the new TypeNode in the right place according to the Key.*

```
/*      Create an empty set of types.      */
TypeSet* CreateTypeSet();
```

```
/*      Free memory allocated to the type set.      */
void DeleteTypeSet(TypeSet*);
```

```
/*      Insert a new type into the set of types.      */
void InsertTypeNode(TypeSet*, TypeNode*);
```

Do not search for a function to remove TypeNodes from the set, there isn't any. This function is not needed in the parser stage. I suspect you don't need it either, but if you want to implement it, go right ahead.

```
/*      Print the type set. Used for debugging.      */
void PrintTypeSet(TypeSet*);
```

typenode.h

The type definition could have been inserted into the node, but I wanted a separation cause of the complexity of the TypeDefinition.

```
/* TypeNode contains the information of the type and a pointer to next TypeNode in set. */
typedef struct typenode
{
    TypeDefinition*   TypeDef;
    struct typenode*  Next;
} TypeNode;

/* CreateTypeNode - Create a type node. */
TypeNode* CreateTypeNode(TypeDefinition*, TypeNode*);

/* Free memory allocated for type node. */
void DeleteTypeNode(TypeNode*);

/* Print type node. */
void PrintTypeNode(TypeNode* p);
```

typedefinition.h

When the Core program becomes more complex, there would be additional types. If you want to add a new type to the Core program, insert the appropriate YACC grammar.

```
#define TYPE_NAME          1
#define TYPE_RANGE        2
#define TYPE_ENUM         3
#define TYPE_SCALARSET    4
```

For the new type add : #define TYPE_NEW_TYPE 5.

Important! To add new types and new characteristics to them, change typedefinition.h and typedefinition.c alone. Do not insert variables, which define the type in the typenode – IT IS JUST NOT HELTHY.

```

/*      TypeDefinition holds the information for each possible type in the Core program. */
typedef struct typedefinition
{
    int          Type;
    char*       TypeName;
    char*       Name;
    Expression* llow;
    Expression* lhigh;
    Expression* Scalar;
    EnumSet*    EnumList;

```

Add the appropriate additional data if needed to the struct. Suppose we want a Matrix type of 2*2, add a 2 dimensional array: int Mat[2][2].

```

} TypeDefinition;

```

```

/*  Constructor for Range Type – Name of the new type, lower bound and upper bound. */
TypeDefinition* CreateRangeType(char*, Expression*, Expression*);

```

```

/*
    Constructor for a type, which is identical to another existing type. For example, when
    in language C we type “typedef int Boolean;” we create a new type called Boolean,
    which is just another name for int.

```

```

*/
TypeDefinition* CreateNameType(char*, char*);

```

```

/*
    Constructor for an enumeration type – a name of the type and a set of strings, which
    are the values of this type.

```

```

*/
TypeDefinition* CreateEnumType(char*, EnumSet*);

```

```

/*
    Constructor for a scalar set – a name of the type and a number X which defines a
    range of integer numbers [0..X-1].

```

```

*/
TypeDefinition* CreateScalarSetType(char*, Expression*);

```

For the new type we must create a new constructor.

```

/*      Free memory allocated for the type defintion.          */
void DeleteTypeDefinition(TypeDefinition*);

```

Insert into the DeleteTypeDefinition a free memory for the new type.

```

/*      Prints the type definition. Used for debugging.        */
void PrintTypeDefinition(TypeDefinition*);

```

For debuging purposes, add in the Print function a textual presentation of the new type.

enumset.h

```
/*      A set of enumeration values.  */
typedef struct enumset
{
    EnumNode*  EnumList;
    int        size;
} EnumSet;
```

To change the data structure of the EnumSet you have to update enumset.h, enumset.c, enumnode.h and enumnode.c.

```
/*      Create an empty set of enum values.      */
EnumSet* CreateEnumSet();

/*      Free memory allocated for a set of enum values.      */
void DeleteEnumSet(EnumSet*);

/*      Insert a new enum value into a set of enum values.      */
void InsertEnumNode(EnumSet*, EnumNode*);

/*      Prints the enum set. Used for debugging.      */
void PrintEnumSet(EnumSet*);
```

enumnode.h

```
/*
    EnumNode holds the value, which is always a string.
    EnumNode holds the pointer to the next value.
*/
typedef struct enumnode
{
    char*          Value;
    struct enumnode*  Next ;
} EnumNode;
```

Unlike the TypeNode and TypeDefinition where there is a separation, because of the simplicity of the data stored in the EnumNode I don't make any separation.

I don't project that there would be any change in the complexity of this struct.

```
/*      Create an enum node.      */
EnumNode* CreateEnumNode(char*, EnumNode*);

/*      Free memory allocated for the enum node.      */
void DeleteEnumNode(EnumNode*);

/*      Prints the enum node content.      */
void PrintEnumNode(EnumNode*);
```

expression.h

```
/*      Tags of the Types.      */
#define E_INTEGER      0
#define E_VARIABLE    1
#define E_SET          2
#define E_BOOLEAN     3
#define E_UNARY       4
#define E_BINARY      5
#define E_RANGE       6
```

For creation of another type of constant or another type of expression combination, add a tag to define it.

For example: #define E_MATRIXTWOBYTWO 7.

For example: #define E_THREE_TUPLE 8.

```
/*      Tags of Operations      */
#define NOP            -1
#define OP_PLUS       0
#define OP_MINUS      1
#define OP_MULT       2
#define OP_DIV        3
#define OP_MOD        4
#define OP_EQ         5
#define OP_NOTEQ      6
#define OP_GT         7
#define OP_LT         8
#define OP_GE         9
#define OP_LE        10
#define OP_AND        11
#define OP_OR         12
#define OP_ARRAY      13
#define OP_NOT        14
#define OP_NEXT       15
#define OP_BR         16
#define OP_UMINUS     17
```

You can add new operations.

For example: #define OP_LOG 18

```

/* Defines a recursive definition of an expression. */
typedef struct expression
{
    int Op;
    struct expression* Left;
    struct expression* Right;
    struct expression* Father;
}

```

For now Left expression and Right expression are enough. If a new expression has to be defined by 3 different expression then we must add another “son” expression.

```

int Type;
int LeafInteger;
int LeafBoolean;
char* LeafVariable;
struct set* LeafSet;
int llow,
    lhigh;

```

For adding a new constant expression, for example a float number, then we must add: float LeafFloat;

```

} Expression;

```

```

/* A combination of two expressions with an operation in the middle. */
Expression* CreateBinaryExpr (int op, Expression* left, Expression* right);

```

```

/*
    An expression with an unary operation on it. The expression structure holds the type
    of operation in the root, and the expression on which the operation works at the left
    son.
*/

```

```

Expression* CreateUnariExpr (int op, Expression* left);

```

```

/* An expression which holds a value of integer type. */
Expression* CreateIntegerExpr (int Ivalue);

```

```

/* An expression which holds a value of boolean type. */
Expression* CreateBooleanExpr (int Bvalue);

```

```

/* An expression which holds a name of a variable. */
Expression* CreateVarExpr (char* varname);

```

```

/*
    An expression which holds a set of expressions. The core language permits only sets
    with values. The data structure can hold any expression.
*/

```

```

Expression* CreateSetExpr (struct set* exprset);

```

```

/* An expression which holds a range of integers (as in pascal 1..10). */
Expression* CreateIRangeExpr (int lowR, int highR);

```

```
/*  
    Frees memory allocated by any type of expression. The function knows exactly what  
    kind of expression it is by the 'Type' instance variable. The function is recursive.  
*/  
void DeleteExpr (Expression*);  
  
/*  
    Prints the expression into the stderr. The function knows exactly what kind of  
    expression it is by the 'Type' instance variable. The function is recursive.  
*/  
void PrintExpr (Expression*);  
  
/*  
    Returns true (1) if the expression contains the 'Next' operator, that is if the expression  
    has a variable 'x' then the function returns true, else returns false.  
    The function is used to determine whether an assignment is legal.  
*/  
int IsExprContainNext (Expression*);
```

It is recommended to create all kinds of expression manipulation functions and expression checkers. For the sake of order, put the new functions you do here.

These functions could be used during translating to the target language.

set.h

```
/* Defines a set of expressions. */
typedef struct set
{
    struct setnode*    setList;
    int                size;
} Set;
```

This data structure is very expressive. You can define set of subsets – something which is not present (at least yet) in the Core language.

```
/* Creates an empty set. In the beginning size equals zero. */
Set* CreateSet ();
```

```
/* Free the memory allocated by the set. Frees the memory of all the expressions in the set. */
void DeleteSet (Set* s);
```

```
/* Inserts an expression into the set and increases the size by 1. */
void SetInsert (Set* s, struct setnode* nexp);
```

```
/* Prints all expressions in the set. Used mostly for debugging. */
void PrintSet (Set* s);
```

```
/*
    This function returns true (1) if at least one of the expressions contains the operator
    'Next'.
*/
int IsSetContainNext (Set* s);
```

setnode.h

```
/*
    SetNode holds an expression.
    SetNode holds a pointer to next node in set.
*/
typedef struct setnode
{
    struct expression*   exp;
    struct setnode*     next;
} SetNode;

/*    Receives an expression and a pointer to another set node.    */
SetNode* CreateSetNode (struct expression* e, SetNode* n);

/*    Changes the operator to the next set node in the set node.    */
void EditNext (SetNode* s, SetNode* newNext);

/*
    Frees memory of the set node.
    WARNING: The function does not free the memory of the next set node.
*/
void DeleteSetNode(SetNode* s);

/*    Prints the content of the set node. Used for debugging.    */
void PrintSetNode (SetNode* s);

/*    Returns true (1) if the expression in the set node contains a 'Next' operator.    */
int IsSetNodeContainNext(SetNode* s);
```

definevarset.h

```
/*
    The struct holds a set of objects which are a define: an identifier which is a
    representation of an expression.
*/
typedef struct definevarset
{
    DefineVar*   DefineVarList;
    Int          size;
} DefineVarSet;
```

This defines a set of defines. This DefineVarSet is used to define Global defines and defines per each module.

```
/*    Defines a set of definitions of constants.    */
DefineVarSet* CreateDefineVarSet();

/*    Free memory allocated for the set of defines.    */
void DeleteDefineVarSet(DefineVarSet*);

/*    Inserts a new defined variable into the set.    */
void InsertDefineVar(DefineVarSet*, DefineVar*);

/*    Prints the set of defined variables.    */
void PrintDefineVarSet(DefineVarSet* );
```

definevar.h

```
/*  
    The struct presents a define: a name, which is a representation of an expression value.  
    The struct has a pointer to next define.
```

```
*/  
typedef struct definevar  
{  
    char*          DefineName;  
    Expression*    DefineValue;  
    struct definevar* next;  
} DefineVar;
```

You can add a function to check that the define is not recursive – This is something which is very hard to detect in the YACC stage.

*For example: DEFINE Hello := (Hello * 2) ;*

*So, when you type Hello, it is as thou you typed an endless expression in the form (((... * 2) * 2) * 2) * 2).*

```
/*    Creates a new Definevar struct.        */  
DefineVar* CreateDefineVar(char*, Expression*, DefineVar*);
```

```
/*    Frees memory allocated for the struct.    */  
void DeleteDefineVar(DefineVar*);
```

```
/*    Prints the struct - used for debugging.    */  
void PrintDefineVar(DefineVar* p);
```

variableset.h

```
/*    A set of variables.    */
typedef struct variableset
{
    Variable*    VarList;
    int          size;
} VariableSet;
```

The Variable Set is used for the global variables and for the local variables which are defined in each module.

```
/*    Creates an empty set of variables.    */
VariableSet* CreateVariableSet();
```

```
/*    Frees allocated memory of a variable set.    */
void DeleteVariableSet(VariableSet*);
```

```
/*    Inserts a variable into a variable set.    */
void InsertVariable(VariableSet*, Variable*);
```

```
/*    Prints a variable set.    */
void PrintVariableSet(VariableSet*);
```

variable.h

```
/*
    Definition of a variable.
    Identifier, type of variable (only type name – not its definition), and an initial value.
*/
typedef struct variable
{
    char*          VarName;          /* Identifier */
    TypeName*     VarType;          /* Type name */
    Expression*   Init;             /* Initial Value. */
    struct variable* Next;
} Variable;
```

Expression Init can be NULL. In that case, no initial value to the variable was given. This expression cannot contain the variable itself – this would be bootstrapping (trying to pull yourself on a horse with your shoe laces).

```
/*    CreateVariable - Create a variable.    */
Variable* CreateVariable(char*, TypeName*, Expression*, Variable*);
```

```
/*    Free memory allocated for the variable.    */
void DeleteVariable(Variable*);
```

```
/*    Prints variable details - used for debugging.    */
void PrintVariable(Variable*);
```

typename.h

```
#define TYPE_NAME_BOOL      1
#define TYPE_NAME_INTEGER  2
#define TYPE_NAME_NAME     3
#define TYPE_NAME_ARRAY    4

/*    Struct which stores a type name.    */
typedef struct typename
{
    int          Type;
    char*       Name;
    struct typename*  ArrayTypeName;
    Expression* ArrayLength;
} TypeName;
```

In analyzing the code we must check that the type name presented corresponds to a defined or existing type in the Core program.

```
/*
    Constructors
    1. CreateTypeNameBool - type BOOLEAN.
    2. CreateTypeNameInteger - type INTEGER.
    3. CreateTypeNameName - type **** which was already defined in the Core program.
    4. CreateTypeNameArray - An name specifying an array with 1 to n dimensions.
*/

/*    Create type name BOOLEAN.    */
TypeName* CreateTypeNameBool();

/*    Creates type name INTEGER.    */
TypeName* CreateTypeNameInteger();

/*    Creates type name which is user defined.    */
TypeName* CreateTypeNameName(char*);

/*    Creates type name which is an array.    */
TypeName* CreateTypeNameArray(TypeName*, Expression*);

/*    Free memory allocated for the TypeName struct.    */
void DeleteTypeName(TypeName*);

/*    Prints type name - used for debugging.    */
void PrintTypeName(TypeName*);
```

moduleset.h

```
/*    Defines a set of modules – the essence of the Core program.    */
typedef struct moduleset
{
    Module*    ModuleList;
    int        size;
} ModuleSet;

/*    Create an empty set of modules.    */
ModuleSet* CreateModuleSet();

/*    Free memory allocated by the module set.    */
void DeleteModuleSet(ModuleSet*);

/*    Inserts a module into the module set.    */
void InsertModule(ModuleSet*, Module*);

/*    Prints the module set.    */
void PrintModuleSet(ModuleSet*);
```

module.h

```
#define MOD_COMBI 1
#define MOD_TRANS 2

/*      Struct presenting a module – a set of transitions or a combination of other modules. */
typedef struct module
{
    char*          Name;
    char           Comment;
    VariableSet*  Vars;
    DefineVarSet* Defines;
    FormalParameterSet* Params;
    Expression*   Cojoin;
    int           Type;
    ModCombin*    Combination;
    TransitionSet* Transitions;
    struct module* Next;
} Module;

/*      Creates a module which is combination of other modules.      */
Module* CreateModuleCombi(ModCombin*);

/*      Creates a module which is a set of transitions.      */
Module* CreateModuleTrans(TransitionSet*);

/*      Update Name in Module struct.      */
void UpdateModuleName(Module*, char*);

/*      Update Comment in Module struct.      */
void UpdateModuleComment(Module*, char*);

/*      Update Local Variables in Module struct.      */
void UpdateModuleVars(Module*, VariableSet*);

/*      Update defines in Module Struct.      */
void UpdateModuleDefines(Module*, DefineVarSet*);

/*      Update formal parameters in Module struct.      */
void UpdateModuleParams(Module*, FormalParameterSet*);

/*      Update Cojoin Flag in Module Struct.      */
void UpdateModuleCojoin(Module*, Expression*);

/*      Free allocated memory of Module struct.      */
void DeleteModule(Module*);

/*      Print Module struct.      */
void PrintModule(Module*);
```

formalparameterset.h

```
/*      Set of formal parameter list. This is the list of parameters of a module definition. */
typedef struct formalparameterset
{
    FormalParameter*   FormalParameterList;
    Int                 size;
} FormalParameterSet;

/*      Creates an empty set of formal parameters.          */
FormalParameterSet* CreateFormalParameterSet();

/*      Free memory allocated by the formal parameter set.  */
void DeleteFormalParameterSet(FormalParameterSet*);

/*      Inserts a formal parameter into the set.           */
void InsertFormalParameter(FormalParameterSet*, FormalParameter*);

/*      Prints set of formal parameters.                   */
void PrintFormalParameterSet(FormalParameterSet*);
```

formalparameter.h

```
/*      Defines a formal parameter. The formal parameter has a name and a type name. */
typedef struct formalparameter
{
    char*               Name;
    TypeName*          Type;
    struct formalparameter* next;
} FormalParameter;

/*      Creates a formal parameter.                        */
FormalParameter* CreateFormalParameter(char*, TypeName*, FormalParameter*);

/*      Frees memory allocated by the formal parameter struct. */
void DeleteFormalParameter(FormalParameter*);

/*      Print formal parameter.                            */
void PrintFormalParameter(FormalParameter*);
```

modcombin.h

```
#define COMBIN_MODIN 1
#define COMBIN_SYNC 2
#define COMBIN_ASYNC 3
#define COMBIN_PART 4
```

You can here add a new type of combination between modules.

```
/*
    A modCombin can be of 4 forms:
    1. A module instantiation.
    2. A synchronous connection.
    3. An asynchronous connection.
    4. A partial synchronous connection.
*/
typedef struct modcombin {
    int                Type;
    /*    In case of synchronous or asynchronous connection.    */
    struct modcombin*  Left;
    struct modcombin*  Right;
    struct modcombin*  Father;
    PairSet*           SyncTransitions;
    RenameSet*         Renames;
    char*              ModuleName;
    ActualParameterSet* Params;
} ModCombin;

/*    Create a module instantiation.    */
ModCombin* CreateCombinModin(char*, ActualParameterSet*, RenameSet*);

/*    Create a module synchronic combination. (II)    */
ModCombin* CreateCombinSync (ModCombin*, ModCombin*, PairSet*,
                             RenameSet*);

/*    Create a module a-synchronic combination. (III)    */
ModCombin* CreateCombinASync(ModCombin*, ModCombin*, PairSet*,
                              RenameSet*);

/*    Create a module partial-synchronization combination. (I()...()I)    */
ModCombin* CreateCombinPart (ModCombin*, ModCombin*, PairSet*, RenameSet*);

/*    Free memory allocated for the module combination.    */
void DeleteModCombin(ModCombin*);

/*    Prints module combination.    */
void PrintModCombin(ModCombin* p);
```

pairset.h

```
/*      Struct defines a set of pairs.          */
typedef struct pairset
{
    PairNode*   PairList;
    int         size;
} PairSet;

/*      Creates an empty set of pairs.          */
PairSet* CreatePairSet();

/*      Frees memory allocated for the set of pairs.          */
void DeletePairSet(PairSet*);

/*      Inserts a new pair into the set of pairs.          */
void InsertPair(PairSet*, PairNode*);

/*      Print a pair set.          */
void PrintPairSet(PairSet*);
```

pairnode.h

```
/*      Struct holds a pair and a pointer to next pair.          */
typedef struct pairnode
{
    Pair*       Npair;
    struct pairnode* next ;
} PairNode;

/*      Builds a pair node. Usually the point 'next' is set to NULL.          */
PairNode* CreatePairNode(Pair* p , PairNode* n);

/*      Free memory allocated for the pair node.          */
void DeletePairNode(PairNode*);

/*      Prints the pair node. Used for debugging.          */
void PrintPairNode(PairNode*);
```

pair.h

```
#define PAIR 1
#define TRANSITIONNAME 2

/* A pair is a pair of pairs of a transition name. */
typedef struct pair
{
    int Type;
    struct pair* Father;
    struct pair* Left;
    struct pair* Right;
    char* TransitionName;
} Pair;

/*
    Constructors :
    1. CreatePair -
    2. CreateTransitionName -
*/
/*
    Creates a pair which can be constructed from 2 pairs, or a pair and a transition, or 2
    transitions.
*/
Pair* CreatePair (Pair*, Pair*);

/*
    Creates a "pair" which stores a name of a transition which is going to be a part of a
    pair.
*/
Pair* CreateTransitionName (char*);

/* Frees memory allocated for the pair. */
void DeletePair (Pair*);

/* Prints pair. Used mostly for debugging. */
void PrintPair (Pair*);
```

renameset.h

```
/*      A rename set of pairs of transition names.          */
typedef struct renameset
{
    RenameNode*    Renames;
    Int            size;
} RenameSet;

/*      Creates an empty set of renames.          */
RenameSet* CreateRenameSet();

/*      Frees memory allocated by the set of renames.    */
void DeleteRenameList(RenameSet*);

/*      Inserts a new rename into the set of renames.    */
void InsertRename(RenameSet*, RenameNode*);

/*      Prints a renameset - used for debugging.        */
void PrintRenameSet(RenameSet*);
```

renamenode.h

```
/*      A rename contains a name which is given to a pair or a transition name.          */
typedef struct renamenode
{
    Pair*          PairRenamed;
    char*          Name;
    struct renamenode* Next;
} RenameNode;

/*      Creates a Rename node.          */
RenameNode* CreateRenameNode(Pair*, char*, RenameNode*);

/*      Frees memory allocated to the Rename Node.    */
void DeleteRenameNode (RenameNode*);

/*      Prints the rename node. Used mostly for debugging.    */
void PrintRenameNode (RenameNode*);
```

actualparameterset.h

```
/*      A set of actual parameters – parameters of a module instantiation.      */
typedef struct actualparameterset
{
    ActualParameter*   ActualParameterList;
    Int                size;
} ActualParameterSet;

/*      Create an actual parameter set.      */
ActualParameterSet* CreateActualParameterSet();

/*      Free memory allocated by a set of actual parameters.      */
void DeleteActualParameterSet(ActualParameterSet*);

/*      Inserts an actual parameter into a set.      */
void InsertActualParameter(ActualParameterSet*, ActualParameter*);

/*      Prnt actual parameter set.      */
void PrintActualParameterSet(ActualParameterSet*);
```

actualparameter.h

```
/*      Actual parameter is only an expression value.      */
typedef struct actualparameter
{
    Expression*        Value;
    struct actualparameter* next;
} ActualParameter;

/*      Create an actual parameter.      */
ActualParameter* CreateActualParameter(Expression*, ActualParameter*);

/*      Free memory allocated by the actual parameter.      */
void DeleteActualParameter(ActualParameter*);

/*      Print actual parameter.      */
void PrintActualParameter(ActualParameter*);
```

transitionset.h

```
/*    A set of transition set.          */
typedef struct transitionset
{
    TransitionNode*    TransitionList;
    Int                size;
} TransitionSet;

/*    Creates an empty set of transitions. In the beginning size equals zero.    */
TransitionSet* CreateTransitionSet();

/*    Frees all the memory allocated for all the transitions.          */
void DeleteTransitionSet(TransitionSet*);

/*    Inserts a new transition node into the set of transitions. Increments the size by one. */
void InsertTransition (TransitionSet*, TransitionNode*);

/*    Prints all the transitions in the set.          */
void PrintTransitionSet(TransitionSet*);
```

transitionnode.h

```
/*
    Struct defines a transition.
    1. A name (identifier of the transition).
    2. Enable condition.
    3. Group of assignments (multiple assignments)
    4. A relation condition.
*/
typedef struct transitionnode
{
    char*          name;
    Expression*    Enable;
    Assign*        Assigns;
    Expression*    Relation;
    struct transitionnode* next;
} TransitionNode;

/*
    Receives a name, enabling condition, an assignment set, a relation expression and a
    pointer to the next transition node which is usually set to NULL.
*/
TransitionNode* CreateTransitionNode(char*, Expression*, Assign*, Expression*,
                                     TransitionNode*);

/*
    Frees memory allocated for the transition node. Frees memory of all the expressions,
    all the assignments. WARNING: The function does not free the next transition nodes.
*/
void DeleteTransitionNode(TransitionNode*);

/*
    Prints all the fields of the transition node. Usually used for debugging.
*/
void PrintTransitionNode(TransitionNode*);
```

assign.h

```
/*    A multiple assignment – a set of assignment.    */
typedef struct assign
{
    struct assignnode*   AssignList;
    int                  size;
} Assign;

/*
    Constructors :
    The assign contains a list of assignment nodes.
    1. CreateAssign –
    The function creates an empty set of assignments. In the beginning size equals zero.
*/
Assign* CreateAssign();

/*    The function frees all the assignment nodes in the list.    */
void DeleteAssign(Assign*);

/*    Inserts an assignment node into the assign. The function increases the size by 1.    */
void InsertAssign(Assign*, struct assignnode*);

/*    Prints all the assignment nodes in the assign. Used for debugging.    */
void PrintAssign(Assign*);
```

assignnode.h

```
/*      Struct defines a single assignment.          */
typedef struct assignnode
{
    Expression*      VariableName;
    Expression*      expr;
    struct assignnode*  next;
} AssignNode;

/*
    Constructors :
    An assign node contains a variable name with an operator 'Next',
    an expression which is assigned into the variable, and
    a pointer to another assignment node.
    1. CreateAssignNode -
    Receives 2 expressions and a pointer to the next assignment node, usually NULL.
*/
AssignNode* CreateAssignNode(Expression*, Expression*, AssignNode*);

/*
    Frees memory allocated for the assignment node. The function frees the 2 expressions.
    WARNING: The function does not free the rest of the nodes.
*/
void DeleteAssignNode(AssignNode*);

/*
    This function checks whether the assignment is legal. The function checks whether the
    right value of the assignment does not contain the operator next and the left value is a
    variable with a 'Next' operator.
    IF THERE IS A NEED TO CHECK SOMETHING ELSE ABOUT THE
    ASSIGNMENT, IT SHOULD BE INSERTED HERE !
*/
int IsLegalAssign (AssignNode*);

/* Prints the assignment node into the stderr. This function is used mostly for debugging. */
void PrintAssignNode (AssignNode*);
```