**MCFromC Project Overview**

# Avishay Ben-Shabtai Zachi Mann

Contents

[Cover 1](#_Toc269016990)

[Glossary 4](#_Toc269016991)

[General Architecture Overview 4](#_Toc269016992)

[Syntax Parser 4](#_Toc269016993)

[CFG Creation 4](#_Toc269016994)

[CFG Simplifier 4](#_Toc269016995)

[MC Creation 5](#_Toc269016996)

[User Manual 6](#_Toc269016997)

[Installation 6](#_Toc269016998)

[Inspecting the Install Directory 6](#_Toc269016999)

[Usage 6](#_Toc269017000)

[Generating CFG from C code 7](#_Toc269017001)

[Inspecting Results 7](#_Toc269017002)

[Algorithms 9](#_Toc269017003)

[Converting AST to CFG 9](#_Toc269017004)

[Algorithm Details 9](#_Toc269017005)

[Remarks 9](#_Toc269017006)

[MC generation 10](#_Toc269017007)

[Expressions Evaluation 10](#_Toc269017008)

[Abstract Evaluation of Expressions to MCs 13](#_Toc269017009)

[*Data Structures* 13](#_Toc269017010)

[*Definitions* 13](#_Toc269017011)

[Future Work 16](#_Toc269017012)

[Syntax Parsing 16](#_Toc269017013)

[MC Generation and Expression Evaluation 16](#_Toc269017014)

[References 17](#_Toc269017015)

[Appendix A - Supported grammar (in BNF format) 18](#_Toc269017016)

[Which syntactic structures are legal? 18](#_Toc269017017)

[Remarks on BNF Grammar 18](#_Toc269017018)

[The Grammar in BNF 19](#_Toc269017019)

[Appendix B - C. S. Lee MCS Format 21](#_Toc269017020)

# Glossary

* AST – Abstract Syntax Tree – a tree data structure that contains the given source code as a syntax tree.
* CFG – Control Flow Graph
* MC – Monotonicity Constraints
* ACFG – Annotated Control Flow Graph is CFG that contains MCs on its edges.

# General Architecture Overview

The project architecture is based on stages where each stage takes an input, processes it, and outputs it as input to the next stage. **Figure 1** below shows a high level overview of the data flow between the system stages.

Figure 1 - System Stages

Output

CFG

CFG

AST

C Source

Syntax Parser

CFG Creation

CFG Simplifier

MC Creation

## Syntax Parser

This stage is responsible for converting a text file containing a subset of the  
**C programming language**[[1]](#footnote-1) into an AST that which will be processed into a CFG in the next stage. This stage is implemented using the following 3rd party tools:

* **Flex** – This tool uses **lex** file and produces source file that holds function that takes input stream and returns tokens according to the **lex** file.
* **Bison** – This tool is an implementation for **YACC**. The tool receives a **YACC** input file (language grammar in a format similar to BNF, where each rule contains C code snippet) and generates a C source file which is the code of a parser for the given language.

Since the above tools produce a source files in the C language (and not C++), while the system main algorithms are coded in C++, this stage is implemented as a separate DLL with an interface which takes a file name and returns a pointer to the root node of the AST.

## CFG Creation

This stage is responsible for generating a CFG based on the AST input. The output is a CFG and all the program state variable names. The graph data structure implementation is based on a 3rd party library – the **boost graph library** (**BGL**). The library is a C++ library, implemented in templates and contains only header files. This stage is implemented in a separate DLL named CFGGen, which is written in C++. The algorithm for CFG generation based on the AST of the input program will be covered in a later section.

## CFG Simplifier

The CFG simplifier is a small stage, implemented in the same DLL as the previous stage (CFG Creation) and is responsible to removal of redundant flow points from the CFG. One example for such redundancy is appearance of consecutive expression flow points (basic block). Those flow points can be reduced into a single flow point since there are no branches within the block. The current stage aggregates those flow points into a single flow point that which represents and holds the information of all other expression flow points.

## MC Creation

This MC Creation layer is responsible for trying to evaluate changes in the values of the program state variables in each abstract transition of the program (represented as an edge in the CFG). The input for this stage is a CFG (simplified from the previous stage) and the names of all state variables (found by traversing the AST, during the CFG generation). Abstract Interpretation is used in order to track changes in the values of the program state variables. This stage will be covered in detail in **MC generation** section.

The following chart describes the normal application flow between the different code projects.

Figure 2 - Code Project Flow

# User Manual

## Installation

The installation of the project is pretty simple all you need to do is to double click on **MCFromCSetup.msi** and proceed with the installation.   
**Note:**

* You have to be administrator in order to perform installation.
* During installation you might be asked to download and install .NET Framework 3.5

### Inspecting the Install Directory

Under the installation directory you will find the following items:

* Main project's binary outputs that includes the following: SyntaxParserDLL.dll,CFGGen.dll, ParserDotNetBridge.dll, CFGViewer.exe
* A Zip file, containing the project's source along with documentation.

## Usage

When launching the **CFGViewer.exe** application the main application window will be opened:

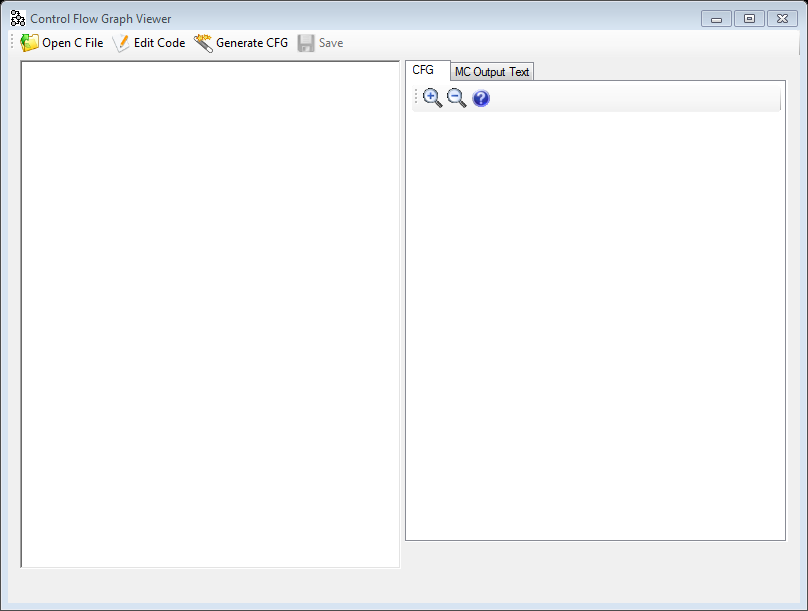


Figure 3 - Main Application Window

### Generating CFG from C code

In order to generate a CFG from a "C" source you will need to provide the "C" source by either loading the code from a source file:

1. Click on the "**Open C File**" button on the toolbar.
2. Select the "C" source file.

Or by manually editing the code:

1. Click on the "**Edit Code**" button on the toolbar.
2. Edit the code in the left pane.

Then just click on the "**Generate CFG**" button on the toolbar.

### Inspecting Results

After the CFG is generated you can view the results in the right pane:

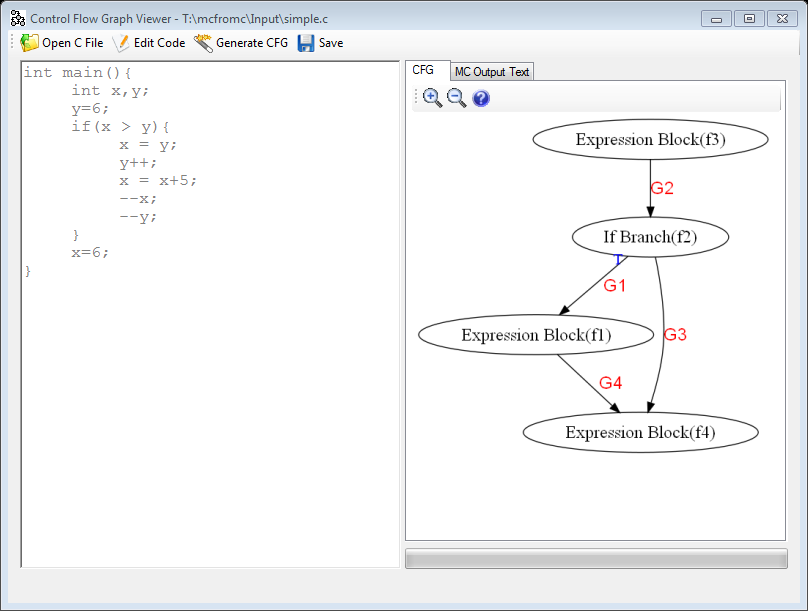


Figure 4 - Result Window

As you can see in Figure 4 above, the input "C" source code is on the left pane and the result CFG is on the right pane. When hovering with the mouse cursor over a flow point (vertex in the CFG), the code represented by the flow point is displayed in a tool-tip and in the status-bar below the CFG. In order to inspect a transition you can double click on the MC graph red label, located on the edge representing the requested transition. This action will pop a window where you can inspect how specific transition affects the program state variables:

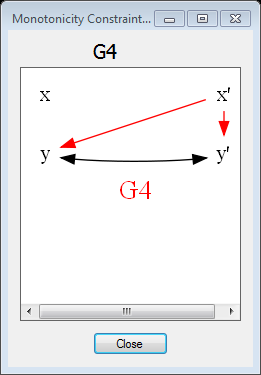


Figure 5 - MC Inspection window

In case you would like you can inspect the result in "C. S. Lee MCS Format" (you can find the format in Appendix B - C. S. Lee MCS Format) by clicking the "**MC Output Text**" tab that is located below the right pane toolbar. You can also save this text to file by clicking on the "**Save**" button that is located at the main application window's toolbar.

# Algorithms

## Converting AST to CFG

This page describes general outlines of the algorithm used for converting the Abstract Syntax Tree(AST), as received by the Syntax Parser, into a Control Flow Graph(CFG).

### Algorithm Details

*Input*: **AST** - an Abstract Syntax Tree.

*Output*: **CFG** - a Control Flow Graph, containing Flow Points and directed edges between them.

1. For each handled node in the AST (mainly expressions and control flow constructs), create a Flow Point in the CFG.
2. Merge consecutive commands Flow Points into a compound block - a block of commands which are consecutive in the program code. This block does not appear in the CFG.
3. Merge consecutive expression Flow Points into a single Flow Point in the CFG(representing an expression block)
4. Recursively traverse the AST:
   1. Connect each node Flow Point to its sub-nodes Flow Points.
   2. If current node is a loop node, connect all end-nodes Flow Points to its Flow Point.
   3. If current node's Flow Point is part of a compound block, connect its end points (might be the node's Flow Point itself) to the next Flow Point inside the compound block.
   4. Return current end points to the calling level.
5. If there are consecutive expression blocks, merge them into a single block.

### Remarks

Each branch command (if / if else / while / for / do while) in the AST is handled differently by the AST to CFG conversion algorithm, according to the specific command semantics. For instance, a For Loop of the following form:

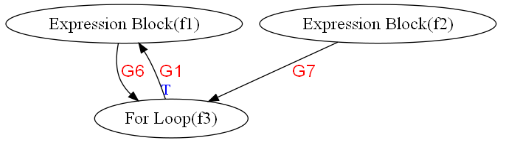
for (i = 0; i < 100; ++i)

{

x = x + i;

}

Is interpreted into the following graph:



Where flow point **f2** represents the for loop initialization (i = 0;), and flow point **f1** represents the commands:   
 1. x = x + i;  
 2. ++i;

## MC generation

### Expressions Evaluation

#### Definitions

* Atom Expression – an expression that contains one of the following:
  + Variable name
  + Integer constant
  + Undefined value.
* Binary Expression – expressions consisting of 2 sub-expressions and a binary operator, for example x+y or 3\*x.

#### Which Expressions Are handled ?

The following expression types are handled:

1. Consecutive command blocks containing commands of the following types:
   1. Assignment commands.
   2. Simple pre/post increment/decrement commands.
2. Branch conditions(while/do while/for/if):
   1. Order conditions (e.g x < y, y >= z)
   2. Logical conjunction of order conditions (x < y && y >= z && z >= x).

#### Assignment Commands Expressions

Each expression is broken down to a binary tree containing Atom Expressions in its leaf nodes and Binary Operator Expressions in its non-leaf nodes. For example, the following expression:

3\*x+5\*y-7

Is broken into the following expression tree:

BinExpr(ADD)  
  |  
  |--BinExpr(MUL)  
  |    |  
  |    |--AtomExpr(3)  
  |    |--AtomExpr(x)  
  |  
  |--BinExpr(SUB)  
       |  
       |--BinExpr(MUL)  
       |    |  
       |    |--AtomExpr(5)  
       |    |--AtomExpr(y)  
       |  
       |--AtomExpr(7)

#### Assignment Expression

When expression is evaluated, each variable is substituted with its value:

1. If all variable values are known (and numeric), then the expression is evaluated to the computation result.
2. If some variable's value is Undefined, then an Undefined value will substitute the variable within the expression. This undefined value is unique and is distinguishable from other undefined values.

Consider the following code snippet:

(1) x = y;  
(2) x = x + 5;

Let us assume that the variable 'y' is undefined (has unknown value). The following describes how the lines above are evaluated:

1. Since 'y' is undefined, it has an undefined value of U1. In the assignment command, x will assume the undefined value U1.
2. In this assignment command, x will assume the expression U1+5 :

BinExpr(ADD)  
     |  
     |--AtomExpr(U1)  
     |--AtomExpr(5)

When substituting a variable with a composite expression it is possible in some cases to reduce the expression into a simpler form which might be easier to check for size change. This simplification is done using an Expression Simplifier object, which tries to match an expression into a pattern out of a predefined set of expression patterns. If such a pattern is found, then the expression is simplified according to the rules defined in Expression Simplifier object for the found pattern. For example, the expression **(U4+17)-17** is simplified into **U4** by matching an expression pattern '($+c1)-c1' and returning the simplified expression '$'. '$' in the pattern above represents a wildcard – any type of Atom expression, and 'c1' represents a constant.

Limitations

The following expressions are syntactically allowed but may cause wrong MC results to be generated:

* x++++ / --x++ / ----x / etc.

Abstract Evaluation of Expressions to MCs

#### Data Structures

* **Val** – mapping from variable names to expression values.
* **Invariant** – a tuple in the following form <x1,o,x2> where .

#### Definitions

* **Val(f)** – A **Val** mapping which is attached to flow point **f**.
* **current** - A "temporary" **Val** mapping
* **current(x)** - The value of **x** in **current**.
* **ex[5/x]** – The expression ex where all occurrences of the variable **x** in the expression **ex** is substituted by the value **5**.
* **MC(e)** - The Monotonicity Constraints graph associated with an edge **e** of the CFG.
* **UNIQUE\_UNDEFINED\_VALUE** – A function which returns a unique undefined value each time that is called (.
* **MC Equality**  - Let be 2 MCs we say that if the following holds:
  + Both handle the same transition
  + Edge Set of Edge Set of

#### Helper Algorithms

##### EVALUATE

**Input:** ex – expression to be evaluated

Val – mapping between variable name to expression

**Output:**  The evaluation result (expression)

* 1. For each variable **x** in **ex**
     1. If **val** does not contain value for **x** then **x**
     2. **ex ex[val(x)/x]**
  2. **ex2 simplify and evaluate ex**
  3. **Return ex2**

##### MERGE

This algorithm merges between two **Val** mappings.

**Input**: **Source** - A **Val** mapping.  
 **Target** - A **Val** mapping.

**Output** – **Target** - A **Val** mapping that is merged.

* 1. If **Target** is empty then
     1. **Target** **Source**
     2. Return **Target**
  2. For each variable **x** which has a value in **Source** Do
     1. If **Target** has value for x and **Target(x)** **Source(x)**
        1. **Target(x)** .
  3. Return **Target**

##### COMPUTE\_MC\_GRAPH

**Input**: **e** - The edge to compute its attached MC  
  **inValues**, **outValues** – **Val** mappings.

**Output**: MC containing the computed transition variants edges.

1. For each variable **xOut** in **outValue**
   1. For each variable **yIn** in **inValue**
      1. **diff** <- **EVALUATE**((**outValue**(**xOut**) – **inValue**(**yIn**)),**outValue**)
      2. If **diff** is not a constant value then continue to next iteration.
      3. If **diff** > 0 add a strict edge **xOut** **yIn** to **MC(e)**
      4. If **diff** < 0 add a strict edge **yIn** **xOut** to **MC(e)**
      5. If **diff** = 0 add non-strict edges **xOutyIn** and **yIn** **xOut** to **MC(e)**

##### COMPUTE-LOGICAL-CLOSURE

**Input**: **f**, **g** - Source and target flow points  
  **mc** is the MC that is attached to the edge **fg**

**Output**: The logical closure of the MC

1. Add arcs which represent **f**'s invariants.
2. Add arcs which represent **g**'s invariants.
3. For each edge **e** of **mc**, Do:
4. If **e** is strict than give weight of -1 to **e** otherwise give it weight 0.
5. If **mc** contains a negative cycle return error.
6. For each vertex ordered pair **u,** **v** in **mc**, Do:
   1. Compute shortest path between **u** and **v** if there is no path continue to next pair.
   2. If the path is negative then add strict arc **uv** to **mc**
   3. else add non strict arc **uv** to **mc**

##### Main Algorithm for Conversion of CFG to ACFG

The algorithm general approach is running a Least Fixed Point evaluation of all expressions associated with CFG flow points. The Least Fixed Point function operates on the MCs attached to the CFG edges, so the algorithm halts when there are no more changes to any MC. The algorithm must halt because:

1. The number of edges of any MC is non-increasing on each LFP iteration besides the first one.
2. Any new edge **e'** replacing an existing edge **e** in some iteration is a super-approximation of **e**, thus can only be replaced once (strict to non-strict).

###### The Algorithm

1. For each flow point **f** in the CFG:
   1. val(**f**) {} (initialize to empty map)
2. flow point f in the CFG
3. While there exists an edge **e** s.t. its MC was changed, Do:
   1. Perform DFS:
      1. When handling the DFS root flow point **f**, Do:
         1. If this is the first time the DFS was performed then initialize all variables of **val(f)** to be undefined values.
      2. When examining each edge **e**, Do:
         1. **f**  source(**e**)
         2. **g**  target(**e**)
         3. **current**  val(**f**)
         4. **newMC** empty MC
         5. If **f** is an expression block Then:
            1. For each expression **ex**, which changes variable **y**, Do:

**current(y)**  **EVALUATE(ex, current)**

* + - 1. Else (this is a branch flow point):
         1. **ex**  conditional expression of **f**
         2. if **e**  is the logical complement edge (**ex** is false)

**ex** logical complement of **ex**

* + - * 1. **invariants** decomposition of **ex** into order expressions which must be true in order for **ex** to be true.
        2. for each invariant **inv** in **invariants**, Do:

Add edge representing **inv** to **newMC**

* + - 1. **COMPUTE\_MC\_GRAPH**(**e**, **val(f)**, **current, newMC**)
      2. if **newMC**  **MC(e)**
         1. **MC(e)** **newMC**
      3. **MERGE**(**val**(**g**), **current**)

1. For each edge in CFG, Do:
   1. **COMPUTE\_LOGICAL\_CLOSURE**(source(**e**),target(**e**)**,MC(e)**)

# Future Work

The projects architecture allows expanding the conclusion deduction process in several areas:

## Syntax Parsing

* Data Types - Handling variety of data types which are integer by nature (char, enum, short, long, pointers, etc.).
* Static Arrays – Can be handled as a set of variables from the same data type.
* goto command – Can be handled as other branches are. Support for label finding is required.
* Switch – Can be handled as multiple if else clauses.

## MC Generation and Expression Evaluation

* Signed Undefined Values – Divide undefined values into 3 types – Positive(>=0), Negative(<0), Top, and improve the expression evaluation engine to support deduction of size change based on the undefined value sign.
* Logical "or" Branch Condition (||) – Flow points containing logical "or" conditions can be split into several flow points for each of the "or" sub-conditions, and connected accordingly:
  + .

# References

* Project Site: <http://code.google.com/p/mcfromc/>
* Boost Library: <http://www.boost.org/>
* Ansi C syntax in Yacc BNF-like grammar:   
  <http://www.lysator.liu.se/c/ANSI-C-grammar-y.html>
* Lex input for Ansi C parser generation:   
  <http://www.lysator.liu.se/c/ANSI-C-grammar-l.html>
* Amir M. Ben-Amram. Size-Change Termination, Monotonicity Constraints and Ranking Functions. October 2009

# Appendix A - Supported grammar (in BNF format)

This section describes the subset of the C language grammar which is handled by the bison parser and the CFG generation algorithm. The grammar is described in Backus-Naur Form.

## Which syntactic structures are legal?

Basically any structure legal in the C language except, mainly, the following elements:

1. Pointers.
2. Arrays.
3. **goto**, **break**, **continue** and **return** commands.
4. **switch** statement.
5. Logical not operator (**!**).
6. Bitwise operators ( ^, ~, >>, <<, &, | ) and bitwise assignment operators  
    ( <<=, >>=, ~=, &=, ^=, |= ).
7. Enumerators (**enum**).
8. Initialization lists ( int arr[] = {0,1,2,3,4}; )
9. Multiple assignment ( x = y = z = 5; )

In addition, the following applies:

1. Only one function definition is allowed per translation unit (enforced by the BNF structure).
2. All variables are considered as integers (only **int** variable declaration is allowed) and assignment of floating point values is disallowed.
3. Prefix/Postfix decrement/increment expressions (--/++) are not supported within assignment expressions or conditional expressions. This is not enforced by the BNF structure, but later on checked for validity.

## Remarks on BNF Grammar

The grammar used has many redundancies, as it was created only by removing unsupported syntactic constructs while not removing constructs which may be an alias of other structures. For example, since bitwise and is not supported, **and\_expression** construct is just an alias to **equality\_expression**, while before it was defined:

**and\_expression** ::= equality\_expression | and\_expression '&' equality\_expression

## The Grammar in BNF

**translation\_unit**

::= external\_declaration

**external\_declaration**

::= function\_definition | declaration

**function\_definition**

::= declaration\_specifiers declarator declaration\_list compound\_statement

| declaration\_specifiers declarator compound\_statement | declarator declaration\_list compound\_statement | declarator compound\_statement

**primary\_expression**

::= IDENTIFIER

| CONSTANT | '(' expression ')'

**postfix\_expression**

::= primary\_expression

| postfix\_expression '++' | postfix\_expression '--'

**unary\_expression**

::= postfix\_expression | '++' unary\_expression | '--' unary\_expression | unary\_operator unary\_expression

**unary\_operator**

::= '+' | '-'

**multiplicative\_expression**

::= unary\_expression | multiplicative\_expression '\*' unary\_expression | multiplicative\_expression '/' unary\_expression

| multiplicative\_expression '%' unary\_expression

**additive\_expression**

::= multiplicative\_expression | additive\_expression '+' multiplicative\_expression | additive\_expression '-' multiplicative\_expression

**shift\_expression**

::= additive\_expression

**relational\_expression**

::= shift\_expression | relational\_expression '<' shift\_expression | relational\_expression '>' shift\_expression | relational\_expression '<=' shift\_expression | relational\_expression '>=' shift\_expression

**equality\_expression**

::= relational\_expression | equality\_expression '==' relational\_expression | equality\_expression '!=' relational\_expression

**and\_expression**

::= equality\_expression

**exclusive\_or\_expression**

::= and\_expression

**inclusive\_or\_expression**

::= exclusive\_or\_expression

**logical\_and\_expression**

::= inclusive\_or\_expression | logical\_and\_expression '&&' inclusive\_or\_expression

**logical\_or\_expression**

::= logical\_and\_expression | logical\_or\_expression '||' logical\_and\_expression

**conditional\_expression**

::= logical\_or\_expression

**assignment\_expression**

::= conditional\_expression | unary\_expression assignment\_operator conditional\_expression

**assignment\_operator**

::= '=' | '**=' | '/=' | '+=' | '-='**

**| '%='**

**expression**

::= assignment\_expression | expression ',' assignment\_expression

**declaration**

::= declaration\_specifiers ';' | declaration\_specifiers init\_declarator\_list ';'

**declaration\_specifiers**

::= type\_specifier

**init\_declarator\_list**

::= init\_declarator | init\_declarator\_list ',' init\_declarator

**init\_declarator**

::= declarator | declarator '=' initializer

**type\_specifier**

::= 'int'

**declarator**

::= direct\_declarator

**direct\_declarator**

::= IDENTIFIER | '(' declarator ')' | direct\_declarator '(' parameter\_type\_list ')' | direct\_declarator '(' identifier\_list ')' | direct\_declarator '(' ')'

**parameter\_type\_list**

::= parameter\_list

**parameter\_list**

::= parameter\_declaration | parameter\_list ',' parameter\_declaration

**parameter\_declaration**

::= declaration\_specifiers declarator | declaration\_specifiers

**identifier\_list**

::= IDENTIFIER | identifier\_list ',' IDENTIFIER

**initializer**

::= conditional\_expression

**statement**

::= compound\_statement | expression\_statement | selection\_statement | iteration\_statement

**compound\_statement**

::= '{' '}' | '{' statement\_list '}' | '{' declaration\_list '}' | '{' declaration\_list statement\_list '}'

**declaration\_list**

::= declaration | declaration\_list declaration

**statement\_list**

::= statement | statement\_list statement

**expression\_statement**

::= ';' | expression ';'

**selection\_statement**

::= 'if' '(' expression ')' statement | 'if' '(' expression ')' statement 'else' statement

**iteration\_statement**

::= 'while' '(' expression ')' statement | 'do' statement 'while' '(' expression ')' ';' | 'for' '(' expression\_statement expression\_statement expression ')' statement

# Appendix B - C. S. Lee MCS Format

**Syntax**

*MCS* ::= *MC* { *MC* } every rule on separate line

*MC* ::= *head* **':-'** *body* **';'** *tail*

*head, tail* ::= *FPid* **'('** *Vid* { , *Vid* } **')'**

*body* ::= **'['** *inequality* { , *inequality* } **']'**

*inequality* ::= *Vid* [ **'>'** | **'>='** | **'='** | **'<'** | **'<='** ] *Vid*

**Lexical Tokens:**

*FPid* flow-point identifier

*Vid* variable identifier

**Rules:**

Every *Vid* that appears in a rule's body has to appear either among the variables of the head or those of the tail.

1. See **Supported grammar (in BNF format)** section for the exact language definition [↑](#footnote-ref-1)