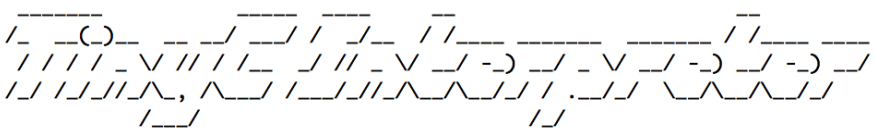
Computer Aided Verification
181.144, UE, 2012S



User Manual

Jürgen Maier e0825749@student.tuwien.ac.at
Philipp Paulweber e0727937@student.tuwien.ac.at
Stefan Mödlhamer e0825895@student.tuwien.ac.at

November 2, 2012

Git Date: `2012-11-01 17:44:56`
Git Tag: `Release, 1.0`
Git Hash: `a31bf5aa4bc08cb865551c6c0d5d2eac4ef7329f`

Contents

1 Overview

This manual describes the behavior and the usage of the *TinyC interpreter* which was developed during the exercise of the course "Computer Aided Veri fication" [1, p. 5]. The questions "What is *TinyC*?" and "What is an *interpreter*?" are answered at the beginning, followed by the description of the installation/compilation process. At the end the usage of the interpreter is described and outlined with some examples.

1.1 TinyC

TinyC is a small subset of the C programming language, defined at [1, p. 4]. The correct version of the TinyC grammar in EBNF is in the appendix section at A.1.

We used the term "correct version" here because unfortunately the defined EBNF grammar contained an error and was corrected. However we found out later that even the corrected version is erroneous. The problem is an expression in the grammar which looked in the first version of the exercise sheet like this:

```
<expr> ::= <test> | <id> "=" <expr> | "!" <expr>
```

The problem is that the parser structure of the interpreter can not clearly identify which lexical element should be used by the negation operator (`!`) – `<test>` or an assignment (`<id> "=" <expr>`). Due to that problems the last transition was changed to:

```
<expr> ::= <test> | <id> "=" <expr> | "!" <test>
```

However every expression in the C programming language can be negated, not only test elements, so we changed that into an expression surrounded by parentheses:

```
      <expr> ::= <test> | <id> "=" <expr> | "!" <paren_expr>
<paren_expr> ::= "(" <expr> ")"
```

1.2 Interpreter

An interpreter is a program that takes a source code file, in our case a TinyC source, and starts the analysis phase like a compiler. This phase consists of a lexer and a grammar parser, developed with the tools `flex`, `yacc` and `ox`.

After transforming the source code into an intermediate representation (IR), residing in the memory, [1] the interpreter converts the IR (AST) into a labeled transition system (LTS), which is afterwards used for calculations.

1.2.1 Execution Modes

There are two possible ways to interpret/calculate and execute a program:

[1] We used a classic abstract syntax tree (AST) transformation.

Numeric Execution (default) is the default operation of the interpreter. For every defined input variable the interpreter requests an integer value from the user. If a variable is used in the source code and is not defined as input variable its starting value is defined as 0. The calculations can either be performed in a *free-run* or *step-through* mode. In free-run the interpreter does the calculation in one step and outputs the value for each defined output variable. If no output variable was defined the message *"no output variables declared"* will be printed instead. In the step-through mode only one transition on the LTS is done at once. That means that the program stops after each transition, printing the numeric values of the actual state and waiting for user actions. Available actions are to compute the next transition by pressing 'n' or abort the calculation with 'a'.

Symbolic Execution is an execution mode, where the user has to provide the interpreter an abstract trace (kripke structure) file[2], which is used to perform a symbolic evaluation on the LTS. Like in the previous described numeric execution mode the user is able to choose between a free-run or step-through calculation. The result of the calculation is either 'satisfiable', if the abstract trace is valid, or 'not satisfiable' if a non valid abstract trace was described. Furthermore the interpreter displays the counterexample symbolic formula if the trace was 'not satisfiable'.

All transition formulas and predicates of the abstract trace are checked with the SMT solver library "MathSAT5" [2].

2 Compilation

The installation process is quite simple: `make interpreter` compiles the interpreter right away and you are good to go.
Executing the Makefile with `make` (without options) shows the required dependencies needed to compile the project. Furthermore the README provides additional information.

2.1 Build Dependencies

In general it is possible to compile the project on every Unix/Linux system, however with the integration of the SMT solver library "MathSAT5" [2] the execution of the interpreter got limited to the platforms Linux 32/64bit and Darwin (Mac OS X) 64bit.

To compile the project you need the following tools:

- `make`
- `gcc` and `gcc-c++` (or `clang`)
- `flex` (or `lex`)
- `yacc` (or `bison -y`)

[2]the format is defined in the appendix at A.2 and based on the definition at [1, p. 10]

In addition your system has to provide the command line tool `ox`, which is a preprocessor for the tools `yacc` and `flex`. Furthermore it is necessary to install the "GNU Multiple Precision Arithmetic Library" with C++ support. For the last two dependencies take a look at the `tools` directory of the project folder.

3 Usage

The interpreter needs at least a TinyC source code file to start the execution. Its general syntax is outlined below. In this description the `FILE` tag represents the TinyC source code file. If no other option is selected, the interpreter uses the numeric interpretation, which was defined as the default configuration:

```
interpreter <FILE> [OPTIONS]
```

3.1 Options

3.1.1 `-o <file>` (or `--out <file>`)

The interpreter does not execute the source code at all, it just translates it to a LTS and outputs it into a ASCII encoded file[3]. The argument `<file>` defines the filename of the output file.

(required feature #1)

3.1.2 `-s` (or `--step`)

This option activates the "step through"- or debugger-mode for either the numeric or the symbolic execution. The user can step through the transitions by pressing the button 'n' for next one or aborting the execution with the button 'a'. If the symbolic execution is active (with option `-t`) the user can choose in a loop state if he wants to loop back with the button 'l' or perform a transition to the next state with 'n' again.

(required feature #2,#3)

3.1.3 `-t <file>` (or `--trace <file>`)

With this option the default numeric execution will be disabled and the symbolic execution mode is started instead. The interpreter requires an abstract trace (kripke structure) file[4] provided with the argument `<file>` for the filename.

(required feature #4,#5)

3.1.4 `-b` (or `--bound`)

By using this option, the symbolic execution does not ask the user for loop boundaries on abstract traces but instead uses the values defined in the

abstract trace (kripke structure) file. If no or too few values are defined the standard value 0 is used as boundary for all remaining loops. This option has no effect when used with the numeric execution and is used for automated testing[5].

(additional feature)

3.1.5 `-c` (or `--code`)

This option prints the source code to the console.

(additional feature)

[3] the format is defined at [1, p. 8]

[4] the format is defined in the appendix at A.2 and based on the definition at [1, p. 10]

[5] take a look at our `test` directory in the project folder or execute the test script with the command `make tests`

3.1.6 `-a` (or `--tree`)

When activated this option visualizes the connections between the abstract syntax tree (AST) and the labeled transition system (LTS).

(additional feature)

3.1.7 `-l` (or `--lts`)

Prints the labeled transition system (LTS), like the option `-o` but the output is directly piped into the LTS and the execution starts at once.

(additional feature)

3.1.8 `-g` (or `--grammar`)

The interpreter prints out the TinyC grammar in EBNF and no execution will be performed.

(additional feature)

3.1.9 `-d` (or `--debug`)

More like an internal option to visualize every single step of the interpreter to provide debug information during development.

(additional feature)

3.1.10 `-h` (or `--help`)

Prints out a brief version of this option description to the console.

(additional feature)

3.2 Examples

In this section we present some examples on how to use this interpreter. The following demonstrations use the source code file `test_00.c` located in the `test` directory inside the project folder.

3.2.1 LTS translation to ASCII file

```
$ ./interpreter test/test_00.c -o test/test_00.lts

$ cat test/test_00.lts
9
--
0 1 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 1
0 0 0 1 0 0 0 0 0
0 0 0 0 1 1 0 0 0
0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 1 0
0 0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
--
0 1 -> i = j
1 2 -> assume(!(z == 0))
1 8 -> assume(z == 0)
2 3 -> i = i + z
3 4 -> assume(0 < z)
3 5 -> assume(!(0 < z))
4 5 -> z = z - 1
5 6 -> assume(z < 0)
5 7 -> assume(!(z < 0))
6 7 -> z = z + 1
7 1 -> assume(true)
```

3.2.2 Numeric execution

```
$ ./interpreter test/test_00.c -c -l

Source code:
        input j, z;
        output i;
        i = j;
        while(!(z == 0)) {
            i = i + z;
            if(0 < z)
                z = z - 1;
            if(z < 0)
                z = z + 1;
        }

Labeled transition system:
        0 1 -> i = j
        1 2 -> assume(!(z == 0))
        1 8 -> assume(z == 0)
        2 3 -> i = i + z
        3 4 -> assume(0 < z)
        3 5 -> assume(!(0 < z))
        4 5 -> z = z - 1
        5 6 -> assume(z < 0)
        5 7 -> assume(!(z < 0))
        6 7 -> z = z + 1
        7 1 -> assume(true)

Input value(s):
        j = 5
        z = 2

Output value(s):
        i = 8
```

3.2.3 Numeric execution with step-through mode

```
$ ./interpreter test/test_00.c -s

Input value(s):
        j = 5
        z = 2

Numeric execution trace:
(press 'n' for next transition, 'a' to abort)

        ⟨ 0, (j:=5, z:=2, i:=0) ⟩
                    |                   0 -> 1: i = j  √
        ⟨ 1, (j:=5, z:=2, i:=5) ⟩
                    |                   1 -> 2: assume(!(z == 0))  √
        ⟨ 2, (j:=5, z:=2, i:=5) ⟩
                    |                   2 -> 3: i = i + z  √
        ⟨ 3, (j:=5, z:=2, i:=7) ⟩
                    |                   3 -> 4: assume(0 < z)  √
        ⟨ 4, (j:=5, z:=2, i:=7) ⟩
                    |                   4 -> 5: z = z - 1  √
        ⟨ 5, (j:=5, z:=1, i:=7) ⟩
                    |                   5 -> 6: assume(z < 0)  x
                    |                   5 -> 7: assume(!(z < 0))  √
        ⟨ 7, (j:=5, z:=1, i:=7) ⟩
                    |                   7 -> 1: assume(true)  √
        ⟨ 1, (j:=5, z:=1, i:=7) ⟩
                    |                   1 -> 2: assume(!(z == 0))  √
        ⟨ 2, (j:=5, z:=1, i:=7) ⟩
                    |                   2 -> 3: i = i + z  √
        ⟨ 3, (j:=5, z:=1, i:=8) ⟩
                    |                   3 -> 4: assume(0 < z)  √
        ⟨ 4, (j:=5, z:=1, i:=8) ⟩
                    |                   4 -> 5: z = z - 1  √
        ⟨ 5, (j:=5, z:=0, i:=8) ⟩
                    |                   5 -> 6: assume(z < 0)  x
                    |                   5 -> 7: assume(!(z < 0))  √
        ⟨ 7, (j:=5, z:=0, i:=8) ⟩
                    |                   7 -> 1: assume(true)  √
        ⟨ 1, (j:=5, z:=0, i:=8) ⟩
                    |                   1 -> 2: assume(!(z == 0))  x
                    |                   1 -> 8: assume(z == 0)  √
        ⟨ 8, (j:=5, z:=0, i:=8) ⟩
                                        8 -> END

Output value(s):
        i = 8
```

3.2.4 Symbolic execution (SAT)

```
$ ./interpreter test/test_00.c -t test/test_yes_00_02.tks

Abstract trace:
        kripke  LTS     tinyC
        states  states  predicate(s)
        0       0       {i > j}
        3       1       {i == j}
        5       2       {i == j}
        14      3       {i > j}
        15      4       {i > j}
        16      5       {i > j}
        18      7       {i > j}
        19      1       {i > j}
        21      2       {i > j}
        14      3       {i > j}

Upper bound for kripke state loop(s):
        [...->14->15->16->18->19->21->14->...]: 1

Symbolic execution trace:
        0->3->5->14->15->16->18->19->21->14->
        15->16->18->19->21->14->END

Output value:
        SATISFIABLE
```

3.2.5 Symbolic execution (SAT) with step-through mode

```
$ ./interpreter test/test_00.c -t test/test_yes_00_02.tks -s

Abstract trace:
        kripke  LTS     tinyC
        states  states  predicate(s)
        0       0       {i > j}
        3       1       {i == j}
        5       2       {i == j}
        14      3       {i > j}
        15      4       {i > j}
        16      5       {i > j}
        18      7       {i > j}
        19      1       {i > j}
        21      2       {i > j}
        14      3       {i > j}


Symbolic execution trace:
(press 'n' for next transition, 'l' to perform a loop, 'a' to abort)

        ⟨ 0 | 0, {i0 > j0} ⟩ => SAT
                |
(n)             |       (0 | 0/{i > j}) -> (3 | 1/{i == j})  :  i = j
                |
        ⟨ 3 | 1, {i0 > j0  ∧  i1 = j0} ⟩ => SAT
                |
(n)             |       (3 | 1/{i == j}) -> (5 | 2/{i == j})  :  assume(!(z == 0))
                |
        ⟨ 5 | 2, {i0 > j0  ∧  i1 = j0  ∧  ¬(z0 = 0)} ⟩ => SAT
                |
(n)             |       (5 | 2/{i == j}) -> (14 | 3/{i > j})  :  i = i + z
                |
        ⟨ 14 | 3, {i0 > j0  ∧  i1 = j0  ∧  ¬(z0 = 0)  ∧  i2 = i1 + z0  ∧
        i2 > j0} ⟩ => SAT
                |
(n)             |       (14 | 3/{i > j}) -> (15 | 4/{i > j})  :  assume(0 < z)
                |
        ⟨ 15 | 4, {i0 > j0  ∧  i1 = j0  ∧  ¬(z0 = 0)  ∧  i2 = i1 + z0  ∧
        i2 > j0  ∧  0 < z0} ⟩ => SAT
                |
(n)             |       (15 | 4/{i > j}) -> (16 | 5/{i > j})  :  z = z - 1
                |
        ⟨ 16 | 5, {i0 > j0  ∧  i1 = j0  ∧  ¬(z0 = 0)  ∧  i2 = i1 + z0  ∧
        i2 > j0  ∧  0 < z0  ∧  z1 = z0 - 1} ⟩ => SAT
                |
(n)             |       (16 | 5/{i > j}) -> (18 | 7/{i > j})  :  assume(!(z < 0))
                |
        ⟨ 18 | 7, {i0 > j0  ∧  i1 = j0  ∧  ¬(z0 = 0)  ∧  i2 = i1 + z0  ∧
        i2 > j0  ∧  0 < z0  ∧  z1 = z0 - 1  ∧  ¬(z1 < 0)} ⟩ => SAT
                |
(n)             |       (18 | 7/{i > j}) -> (19 | 1/{i > j})  :  assume(true)
                |
        ⟨ 19 | 1, {i0 > j0  ∧  i1 = j0  ∧  ¬(z0 = 0)  ∧  i2 = i1 + z0  ∧
        i2 > j0  ∧  0 < z0  ∧  z1 = z0 - 1  ∧  ¬(z1 < 0)} ⟩ => SAT
                |
(n)             |       (19 | 1/{i > j}) -> (21 | 2/{i > j})  :  assume(!(z == 0))
                |
        ⟨ 21 | 2, {i0 > j0  ∧  i1 = j0  ∧  ¬(z0 = 0)  ∧  i2 = i1 + z0  ∧
        i2 > j0  ∧  0 < z0  ∧  z1 = z0 - 1  ∧  ¬(z1 < 0)  ∧
        ¬(z1 = 0)} ⟩ => SAT
                |
(n)             |       (21 | 2/{i > j}) -> (14 | 3/{i > j})  :  i = i + z
                |
        ⟨ 14 | 3, {i0 > j0  ∧  i1 = j0  ∧  ¬(z0 = 0)  ∧  i2 = i1 + z0  ∧
        i2 > j0  ∧  0 < z0  ∧  z1 = z0 - 1  ∧  ¬(z1 < 0)  ∧
        ¬(z1 = 0)  ∧  i3 = i2 + z1} ⟩ => SAT
                |
(n)             |       (14 | 3/{i > j}) -> END
(l)             |       (14 | 3/{i > j}) -> (15 | 4/{i > j})  :  assume(0 < z)
                |
        ⟨ 15 | 4, {i0 > j0  ∧  i1 = j0  ∧  ¬(z0 = 0)  ∧  i2 = i1 + z0  ∧
        i2 > j0  ∧  0 < z0  ∧  z1 = z0 - 1  ∧  ¬(z1 < 0)  ∧
```

```
        ¬(z1 = 0) ∧ i3 = i2 + z1 ∧ 0 < z1 ∧ i3 > j0} ⟩ => SAT
                  |
(n)               |       (15 | 4/{i > j}) -> (16 | 5/{i > j})  :  z = z - 1
                  |
        ⟨ 16 | 5, {i0 > j0 ∧ i1 = j0 ∧ ¬(z0 = 0) ∧ i2 = i1 + z0 ∧
        i2 > j0 ∧ 0 < z0 ∧ z1 = z0 - 1 ∧ ¬(z1 < 0) ∧
        ¬(z1 = 0) ∧ i3 = i2 + z1 ∧ 0 < z1 ∧ i3 > j0 ∧
        z2 = z1 - 1} ⟩ => SAT
                  |
(n)               |       (16 | 5/{i > j}) -> (18 | 7/{i > j})  :  assume(!(z < 0))
                  |
        ⟨ 18 | 7, {i0 > j0 ∧ i1 = j0 ∧ ¬(z0 = 0) ∧ i2 = i1 + z0 ∧
        i2 > j0 ∧ 0 < z0 ∧ z1 = z0 - 1 ∧ ¬(z1 < 0) ∧
        ¬(z1 = 0) ∧ i3 = i2 + z1 ∧ 0 < z1 ∧ i3 > j0 ∧
        z2 = z1 - 1 ∧ ¬(z2 < 0)} ⟩ => SAT
                  |
(n)               |       (18 | 7/{i > j}) -> (19 | 1/{i > j})  :  assume(true)
                  |
        ⟨ 19 | 1, {i0 > j0 ∧ i1 = j0 ∧ ¬(z0 = 0) ∧ i2 = i1 + z0 ∧
        i2 > j0 ∧ 0 < z0 ∧ z1 = z0 - 1 ∧ ¬(z1 < 0) ∧
        ¬(z1 = 0) ∧ i3 = i2 + z1 ∧ 0 < z1 ∧ i3 > j0 ∧
        z2 = z1 - 1 ∧ ¬(z2 < 0)} ⟩ => SAT
                  |
(n)               |       (19 | 1/{i > j}) -> (21 | 2/{i > j})  :  assume(!(z == 0))
                  |
        ⟨ 21 | 2, {i0 > j0 ∧ i1 = j0 ∧ ¬(z0 = 0) ∧ i2 = i1 + z0 ∧
        i2 > j0 ∧ 0 < z0 ∧ z1 = z0 - 1 ∧ ¬(z1 < 0) ∧
        ¬(z1 = 0) ∧ i3 = i2 + z1 ∧ 0 < z1 ∧ i3 > j0 ∧
        z2 = z1 - 1 ∧ ¬(z2 < 0) ∧ ¬(z2 = 0)} ⟩ => SAT
                  |
(n)               |       (21 | 2/{i > j}) -> (14 | 3/{i > j})  :  i = i + z
                  |
        ⟨ 14 | 3, {i0 > j0 ∧ i1 = j0 ∧ ¬(z0 = 0) ∧ i2 = i1 + z0 ∧
        i2 > j0 ∧ 0 < z0 ∧ z1 = z0 - 1 ∧ ¬(z1 < 0) ∧
        ¬(z1 = 0) ∧ i3 = i2 + z1 ∧ 0 < z1 ∧ i3 > j0 ∧
        z2 = z1 - 1 ∧ ¬(z2 < 0) ∧ ¬(z2 = 0) ∧ i4 = i3 + z2} ⟩ => SAT
                  |
(n)               |       (14 | 3/{i > j}) -> END
(l)               |       (14 | 3/{i > j}) -> (15 | 4/{i > j})  :  assume(0 < z)
                  |
        ⟨ 14 | 3, {i0 > j0 ∧ i1 = j0 ∧ ¬(z0 = 0) ∧ i2 = i1 + z0 ∧
        i2 > j0 ∧ 0 < z0 ∧ z1 = z0 - 1 ∧ ¬(z1 < 0) ∧
        ¬(z1 = 0) ∧ i3 = i2 + z1 ∧ 0 < z1 ∧ i3 > j0 ∧
        z2 = z1 - 1 ∧ ¬(z2 < 0) ∧ ¬(z2 = 0) ∧ i4 = i3 + z2 ∧
        i4 > j0} ⟩ => SAT

Output value:
        SATISFIABLE
```

3.2.6 Symbolic execution (UNSAT)

```
$ ./interpreter test/test_00.c -t test/test_no_00_01.tks

Abstract trace:
        kripke  LTS     tinyC
        states  states  predicate(s)
        1       0       {i == j}
        3       1       {i == j}
        5       2       {i == j}
        6       3       {i < j}
        8       5       {i < j}
        10      7       {i < j}
        11      1       {i < j}
        12      8       {i < j}


Symbolic execution trace:
        1->3->5->6->8->10->11->12->END

Output value:
        NOT SATISFIABLE

Counter example:
        i0 = j0 ∧ i1 = j0 ∧ ¬(z0 = 0) ∧ i2 = i1 + z0 ∧
        i2 < j0 ∧ ¬(0 < z0) ∧ ¬(z0 < 0)
```

3.2.7 Symbolic execution (UNSAT) with step-through mode

```
$ ./interpreter test/test_00.c -t test/test_no_00_01.tks -s

Abstract trace:
        kripke  LTS     tinyC
        states  states  predicate(s)
        1       0       {i == j}
        3       1       {i == j}
        5       2       {i == j}
        6       3       {i < j}
        8       5       {i < j}
        10      7       {i < j}
        11      1       {i < j}
        12      8       {i < j}


Symbolic execution trace:
(press 'n' for next transition, 'l' to perform a loop, 'a' to abort)

        ⟨ 1 | 0, {i0 = j0} ⟩ => SAT
                |
(n)             |       (1 | 0/{i == j}) -> (3 | 1/{i == j})  :  i = j
                |
        ⟨ 3 | 1, {i0 = j0 ∧ i1 = j0} ⟩ => SAT
                |
(n)             |       (3 | 1/{i == j}) -> (5 | 2/{i == j})  :  assume(!(z == 0))
                |
        ⟨ 5 | 2, {i0 = j0 ∧ i1 = j0 ∧ ¬(z0 = 0)} ⟩ => SAT
                |
(n)             |       (5 | 2/{i == j}) -> (6 | 3/{i < j})  :  i = i + z
                |
        ⟨ 6 | 3, {i0 = j0 ∧ i1 = j0 ∧ ¬(z0 = 0) ∧ i2 = i1 + z0 ∧
        i2 < j0} ⟩ => SAT
                |
(n)             |       (6 | 3/{i < j}) -> (8 | 5/{i < j})  :  assume(!(0 < z))
                |
        ⟨ 8 | 5, {i0 = j0 ∧ i1 = j0 ∧ ¬(z0 = 0) ∧ i2 = i1 + z0 ∧
        i2 < j0 ∧ ¬(0 < z0)} ⟩ => SAT
                |
(n)             |       (8 | 5/{i < j}) -> (10 | 7/{i < j})  :  assume(!(z < 0))
                |
        ⟨ 10 | 7, {i0 = j0 ∧ i1 = j0 ∧ ¬(z0 = 0) ∧ i2 = i1 + z0 ∧
        i2 < j0 ∧ ¬(0 < z0) ∧ ¬(z0 < 0)} ⟩ => UNSAT


Output value:
        NOT SATISFIABLE

Counter example:
        i0 = j0 ∧ i1 = j0 ∧ ¬(z0 = 0) ∧ i2 = i1 + z0 ∧
        i2 < j0 ∧ ¬(0 < z0) ∧ ¬(z0 < 0)
```

A Appendix

A.1 TinyC Grammer EBNF (with extensions)

```
<program> ::= <statement>
            | <input> <statement>
            | <output> <statement>
            | <input> <output> <statement>

<statement> ::= "if" <paren_expr> <statement>
              | "while" <paren_expr> <statement>
              | "{" { <statement> } "}"
              | <expr> ";"
              | "assert" <paren_expr> ";"
              | ";"

<paren_expr> ::= "(" <expr> ")"

<expr> ::= <test>
         | <id> "=" <expr>
         | "!" <paren_expr> // TinyC correction

<test> ::= <sum>
         | <sum> "==" <sum>
         | <sum> "<" <sum>
         | <sum> ">" <sum>  // TinyC extension

<sum> ::= <term>
        | <sum> "+" <term>
        | <sum> "-" <term>

<term> ::= <id>
         | <int>
         | <paren_expr>

<int> ::= <num><int>
        | <num>

<num> ::= "0" | "1" | "2" | ... | "9"

<id> ::= "a" | "b" | "c" | ... | "z"

<list_of_ids> ::= <id>
                | <id> "," <list_of_ids>

<input> ::= "input" <list_of_ids> ";"

<output> ::= "output" <list_of_ids> ";"
```

A.2 Trace (Kripke Structure) file format (.tks)

The used trace file format is the same as presented in [1, p. 10]. The only difference is the possibility to extend the abstract trace by the loop boundaries in the abstract trace. The loop boundary section is started by ':' and the individual counts are separated by ','. If too few boundaries were defined the standard value 0 is assigned, if too many are presented the unused values are ignored. This extension is very useful especially for automated testing – take a look at the option `-b`. Two examples are given below, once with and once without the extension:

```
$ cat test/test_no_15_01.tks
0,4,10,13,14,15,6,11,12
--
21
--
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
...

$ cat test/test_no_15_02.tks
12,17,18,19,12,17,18,20,13,14,15,6,11,12,17,18,19,12,17,18,20:5,6
--
21
--
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
...
```

A.3 License

B References

[1] Sinn, Moritz "CAV Lab Exercises - SS2012" on the *Methods in Systems Engineering* website, 22.06.2012, `http://forsyte.at/wp-content/uploads/CAVLab_Tasks.pdf`

[2] FBK-IRST "MathSAT5, An SMT Solver for Formal Verification" on the *University of Trento* website, 20.09.2012, `http://mathsat.fbk.eu/`