The SMT-LIBv2 Language and Tools: A Tutorial

David R. Cok
GrammaTech, Inc.

Version 1.1
February 13, 2011

The most recent version is available at
`http://www.grammatech.com/resources/smt/SMTLIBTutorial.pdf`.

Contents

List of Tables

Preface

A tutorial is only as useful as its subject. This tutorial owes its debt to two groups that have established SMT solvers as they are today.

The first is the distributed group of SMT solver implementors. These researchers have pushed the capabilities of SMT solvers, making them a significantly useful tool for model checking and software verification, and making great strides in solver performance over recent years. Without good SMT solvers we would have no need for a standard language, nor a tutorial.

The second and more specific acknowledgment is to Cesare Tinelli, Clark Barett and Aaron Stump, the authors of the SMT-LIB version 2 specification. In addition, Tinelli and Silvio Ranise pioneered the SMT-LIB language, and Barrett and Stump, with Leonardo DeMoura, initiated the SMT-COMP solver competition.

Version History

2011-02-13	Version 1.1	Reported typos corrected
2011-01-24	Version 1.0	First round comments incorporated
2010-12-25	Version 0.2.draft	Updated to 12/21/2010 version of the standard
2010-12-19	Version 0.1.draft	First draft for comment

Note

This document almost certainly contains errors. It may anticipate changes that are not yet reflected in the SMT-LIB standard; it may anticipate changes that do not become adopted; it may not yet contain changes that have been incorporated into the standard; and it may have misinterpreted the standard. If you notice errors, please bring them to the author's attention and they will be corrected in a future edition.

Chapter 1

Introduction

1.1 The SMT-LIB endeavor

This tutorial builds on two significant developments in automated reasoning over recent years. The first development is the considerable advance in SMT solvers. These solvers (and their siblings, SAT solvers) are essential to model checking and software verification. Some such solver is embedded as a background validity checker in most verification systems.

At the 2010 SMT workshop, 10 different provers competed to demonstrate capability and performance, with an additional 8 other groups competing in 2008 and 2009. That competition, SMT-COMP, is a direct contributor to the recent improvements in SMT solver performance. A uniformly available set of benchmark problems provided a measure of solver capability and an objective means of comparing solvers. As a result, solver performance has increased considerably[1][4] over the last several years.

The second key development is the SMT-LIB language itself. Integral to the SMT competition is having a language common across solvers in which to express benchmark problems. That is the task of the SMT-LIB language. The language was first proposed in 2003[5] as the input language for the SMT benchmark problems. However, the language was subsequently revised to meet additional needs. In particular, an important application of SMT solvers is as a backend constraint solver for software verification. In this application, a solver receives input from another tool and the driver tool needs capabilities such as asserting and retracting logical expressions or exploring the satisfying assignments produced by the solver. Those requirements led to SMT-LIB version 2, which was announced in 2010[2]. This tutorial describes the December 21, 2010, edition of that standard.

The SMT-LIB standard has the goal of advancing the theory and practice of SMT solvers by providing a common language and set of benchmarks against which to test and compare solvers.

This tutorial is created in support of that goal, but with the additional intent of encouraging wider use of SMT solvers, in application areas in which SMT solvers may currently be unfamiliar.

1.2 Purpose and Content

This tutorial is intended for two audiences. The primary audience is individuals somewhat new to SMT solvers, or at least to the particular input and output format that is SMT-LIB v.2. This tutorial will provide these readers

- a very brief introduction to some of the key concepts of logic and automated theorem proving that are needed to use SMT solvers,
- information about the context of SMT solvers, SMT-LIB and the recent significant release (v.2),
- examples and description of how SMT-LIB is used to interact with SMT solvers,
- pointers to currently available SMT solvers,
- and descriptions of some tools and test suites that may be useful to the reader.

For this audience, the tutorial intends to provide sufficient information for new users to experiment with SMT solvers using SMT-LIB and for implementors of applications using SMT solvers to effectively use SMT-LIB as the interface language.

A second audience consists of those individuals who are implementing an SMT solver that is compliant with SMT-LIB v.2. These implementors will be fully versed in SMT solvers, how they work, and the associated logical concepts. However, this tutorial's discussion of details of the SMT-LIB format, the command language, and command responses may be relevant. In addition the reference tools and set of compliance tests should be quite useful to someone intending to implement a conforming SMT solver. For this set of readers, the goal of the tutorial and tools is to provide an informal but more accessible overview of the SMT-LIB language and to provide tools that will enable easier and faster development of a solver's front-end.

The tutorial consists of these parts:

- this introductory section;
- a set of examples providing readers a quick picture of SMT-LIB scripts;
- the details of the SMT-LIB language, including syntax, sorts, declarations, expressions, and commands;

- a brief description of the built-in logics and theories;
- an introduction to some of the SMT solvers currently available;
- a list of tools available to interested SMT users and implementors.

Anyone who would like to hear about updates to this tutorial or its associated tools is encouraged to subscribe to the smt-lib@cs.nyu.edu mailing list; you can register for the mailing list at `http://www.cs.nyu.edu/mailman/listinfo/smt-lib`.

1.3 Mechanics

The document employs a few definitions and typographical conventions; these are described here.

A *conforming* SMT solver is one whose behavior obeys the SMT-LIB v.2 standard. Such a solver may do more than the standard requires, but not less. For example, it may define more options or more commands, or it may be less restrictive in interpreting commands. However, any legal SMT-LIB v2 input must be accepted without complaint and yield the defined response.

This document only describes SMT-LIB version 2; references to SMT-LIB are simply an abbreviated reference to version 2 of the language.

Verbatim characters. Text written using a monospaced font, e.g. `font`, represents character sequences that are to be interpreted verbatim. Typically, they are SMT-LIB input or output or fragments of SMT-LIB commands.

Semantic categories. This document uses character sequences such as <*string*> or <*symbol*> or <*binary*> (using italics and enclosed in angle brackets) to denote various semantic categories. For example, <*string*> represents an element of the set of character sequences denoting SMT-LIB strings: sequences of characters from a specified set enclosed in `"` characters.

Examples. The tutorial includes a number of examples. Some of them are examples of input to and output from a conforming SMT solver. These examples are typeset within a box in-line in the text. For example,

```
> (set-logic QF_UF)
success
> (set-logic QF_UF)
(error "The logic has already been set")
```

In such examples, the lines beginning with `>` show input that the user provides to the solver, as if the `>` were a prompt; the other lines are the output received from the tool. In the example

above the two (identical) set-logic commands, including the containing parentheses are user input; the two lines containing the word `success` and the error response are the output. In some cases, the input will be continued on a second line, with a second prompt; these lines are shown beginning with `...>` , as in

```
> (set-logic
...> QF_UF)
success
> (set-logic QF_UF)
(error "The logic has already been set")
```

Note that these are two separate lines of user input, not simply a long line having wrapped; that is, the user typed a new line in the middle of the command. The tutorial sometimes omits the `success` responses to save space.

Comments in the examples — the text from a semicolon to the end of the line — are not part of the input or output, but descriptive text to explain aspects of the example.

Symbols. The document uses these (infix) symbols for logical operations: $\wedge$ (conjunction), $\vee$ (disjunction), $\neg$ (logical negation), $\Rightarrow$ (logical implication), $\equiv$ (logical equivalence), $\neq$ (exclusive or - that is, logical inequivalence), $\forall$ (universal quantifier), and $\exists$ (existential quantifier).

Chapter 2

Quick Start

There are a number of components to the SMT-LIB language: lexical structure, how logical expressions are written, the command language, and the various underlying logics. These are all described in detail in the following chapters. However, they proceed from the ground up, so only at the end can a useful script be written. This approach does not appeal to the person eager to see something work. So this chapter contains several simplified examples that show the general style, but do not explain all the details. After reviewing these, the reader can simply experiment or can read the following chapters to gain a more thorough understanding of the components.

First, you need an SMT-LIB conforming solver with which to work. Unfortunately, most solvers, though they may be excellent at solving constraints, are not yet fully SMT-LIB compliant. Various options for solvers are described in chapter 5, from which you may choose. Some of them can act as SMT-LIB solvers through a Java adapter interface. The adapter interface is SMT-LIB conforming, but you still need a back-end SMT solver. Or you can simply use the adapter interface as a command and type checker, without doing any actual constraint solving. To be specific, here is the command line for the adapter+simplify solver (where "<*path*>" is replaced by an absolute file-system path to the Simplify executable on your system):

```
java -jar jSMTLIB.jar -solver simplify -exec <path>
```

The command above will respond with a prompt (>). If you like, you can type commands directly at the prompt. However, most users may prefer to edit a script in a file, both to save the script and to conveniently correct or change the script. In that case, the command would be (here <*file*> is the file-system path to the script file—either absolute or relative to the current working directory):

```
java -jar jsmtlib.jar -echo -solver simplify -exec <path> <file>
```

Basic Boolean example

Here is a first example, which declares a `Bool` value p and asks whether $(p \wedge \neg p)$ is satisfiable:

```
> (set-logic QF_UF)
success
> (declare-fun p () Bool)
success
> (assert (and p (not p)))
success
> (check-sat)
unsat
> (exit)
success
```

The response to `check-sat` is `unsat`, indicating that the formula is not satisfiable. There is no value of p for which it is true. Equivalently, its negation is always true. So $\neg(p \wedge \neg p)$ is a tautology. In SMT-LIB syntax, this is written `(not (and p (not p)))`. Note that all of the subexpressions are written in fully parenthesized, prefix style.

Setting options

Conforming SMT solvers have a number of options that change the behavior of the solver. A simple one is `:print-success`; this option controls whether `success` is returned in response to each successful command. Here is the first example, with `:print-success` turned off.

```
> (set-option :print-success false)
> (set-logic QF_UF)
> (declare-fun p () Bool)
> (assert (and p (not p)))
> (check-sat)
unsat
> (exit)
```

Integer Arithmetic

This next example contains some arithmetic. For that we use a logic that defines basic arithmetic on integers, the `QF_LIA` logic. Now we can ask, for example, if there is a solution to the pair of

equations $x + 2 * y = 20$ and $x - y = 2$.

```
> (set-logic QF_LIA)
success
> (declare-fun x () Int)
success
> (declare-fun y () Int)
success
> (assert (= (+ x (* 2 y)) 20))
success
> (assert (= (- x y) 2))
success
> (check-sat)
sat
> (exit)
success
```

This similar example does not have a solution. There is a solution if x and y are `Real`, but not if they are `Int`.

```
> (set-logic QF_LIA)
success
> (declare-fun x () Int)
success
> (declare-fun y () Int)
success
> (assert (= (+ x (* 2 y)) 20))
success
> (assert (= (- x y) 3))
success
> (check-sat)
unsat
> (exit)
success
```

Getting values

The first example of the previous section determined that the given constraint problem did indeed have a solution, but did not say what the solution is. To find the solution, one must set a couple options and then use the `get-value` command; one must also have a solver that supports report-

ing the model satisfying a constraint problem.

```
> (set-option :print-success false)
> (set-option :produce-models true)
> (set-option :interactive-mode true)
> (set-logic QF_LIA)
> (declare-fun x () Int)
> (declare-fun y () Int)
> (assert (= (+ x (* 2 y)) 20))
> (assert (= (- x y) 2))
> (check-sat)
sat
> (get-value (x y))
((x 8)(y 6))
> (exit)
```

Using scopes to explore multiple problems

Sometimes there are multiple problems one would like to explore with the same set of definitions and some of the same assertions. To do that, we use the `push` and `pop` commands.

```
> (set-option :print-success false)
> (set-logic QF_LIA)
> (declare-fun x () Int)
> (declare-fun y () Int)
> (assert (= (+ x (* 2 y)) 20))
> (push 1)
> (assert (= (- x y) 2))
> (check-sat)
sat
> (pop 1)
> (push 1)
> (assert (= (- x y) 3))
> (check-sat)
unsat
> (pop 1)
> (exit)
```

Defining new sorts

Some problems profit by defining and using new sorts.

```
> (set-option :print-success false)
> (set-logic QF_UF)
> (declare-sort A 0)
> (declare-fun a () A)
> (declare-fun b () A)
> (declare-fun c () A)
> (declare-fun d () A)
> (declare-fun e () A)
> (assert (or (= c a)(= c b)))
> (assert (or (= d a)(= d b)))
> (assert (or (= e a)(= e b)))
> (push 1)
> (distinct c d)
> (check-sat)
sat
> (pop 1)
> (push 1)
> (distinct c d e)
> (check-sat)
unsat
> (pop 1)
> (exit)
```

Getting information

Solvers can provide some identifying information, obtained with the `get-info` command:

```
> (get-info :name)
(:name "simplify")
> (exit)
success
```

Chapter 3

The SMT-LIB Language (v2)

3.1 Some logical concepts

This tutorial does not provide an introduction to logic or the details of SMT solvers. However, a reader that understands the following concepts will be able to use an SMT solver and the SMT-LIB command language more effectively. A reader interested in the formal semantics of SMT-LIB should consult chapter 4 of the SMT-LIB definition[3].

3.1.1 Satisfiability and Validity

The first concept to understand is the dual relationship between satisfiability and validity. A formula P is *valid* if P always evaluates to true for *any* assignment of appropriate values to its function symbols. A formula P is *satisfiable* if there is *some* assignment of appropriate values to its function symbols under which P evaluates to true. Validity is about finding a proof of a statement; satisfiability is about finding a solution to a set of constraints.

Consider a logical expression P with some free constants, say x and y. We can ask whether P is *valid*, that is whether it is always true for any combination of values for x and y. If P is always true, then $\neg P$ is always false, and then $\neg P$ will not have any satisfying assignment; that is, $\neg P$ is unsatisfiable. In other words,

P is valid precisely when $\neg P$ is not satisfiable (is unsatisfiable).

Alternately,

P is satisfiable if and only if $\neg P$ is not valid (is invalid).

SMT solvers find satisfying assignments (or report that there are none). To determine whether a formula P is valid, we ask an SMT solver whether $\neg P$ is satisfiable. Similarly, to determine if

$$(P \wedge Q) \Rightarrow R$$

is valid, we ask whether its negation,

$$P \wedge Q \wedge \neg R,$$

is satisfiable. If the latter is unsatisfiable, the former is valid. If $P \wedge Q \wedge \neg R$ is satisfiable, a prover is usually able to provide an assignment of values to its free symbols that makes the formula true; that assignment serves as a counterexample of why $(P \wedge Q) \Rightarrow R$ is invalid (why it is not always true). In general, if we are looking for why a formula P is not a theorem, we can look for an assignment that makes $\neg P$ true.

3.1.2 Quantified formulas and SMT solvers

The SMT-LIB language permits quantified expressions, stating that an expression is true for all values of a variable, or for at least one value of a variable. Such expressions are, of course, a natural part of first-order logic. However, most SMT solvers do not handle quantified expressions well. The presence of quantified expressions can lead to a solver being unsure whether a potential satisfying assignment is indeed a correct satisfying assignment; this phenomenon can happen because the solver cannot be sure it has instantiated the quantified expression for all cases relevant to the satisfiability problem at hand. Thus for some sets of formulae, an SMT solver may respond `unknown` to the question of whether the problem is satisfiable or not. Often in such a case the solver may report an assignment, though the `unknown` response indicates that the solver is not certain it is a truly satisfying assignment. Handling quantifiers is an area of research for SMT solvers; some logics (cf. section 4) forbid quantifiers and have faster and more certain decision procedures as a result.

3.1.3 Many-Sorted Logic

The SMT-LIB language expresses logical problems in a many-sorted first-order logic. Accordingly, each well-formed expression has a unique *sort* (known in some other contexts as a *type*). The language provides syntax and commands for defining new sorts and for expressing the sort of new symbols. The pre-defined sort `Bool` is the sort of all Boolean propositional expressions. For example, the propositional connectives `and` and `or` take arguments of sort `Bool` and have a result value of sort `Bool`; the equality predicate (`=`) takes two arguments of the same (but arbitrary) sort and has a result value of sort `Bool`. Each function symbol requires particular sorts for its arguments and has a defined result sort; in some cases, function symbols can be overloaded, but the result of a (disambiguated) function symbol applied to specific arguments of the correct sorts always has a unique sort. Expressions within SMT-LIB are correct only if they are *well-sorted*, that is, if the arguments of function symbols have the sorts expected by the function.

Some other sorts are defined by specific logics. For example, `Int` is a sort defined in the `QF_LIA` logic (and others) to denote the integers and `Real` is defined in the `AUFLIRA` logic to mean the conventional Real numbers.

3.1.4 Formulas and terms

Some automated reasoning tools make a sharp distinction between formulas and terms. Informally, formulas are subexpressions with Boolean values and are combined by top-level connectives such as conjunction and disjunction; terms have values and arguments of other sorts, such as `(x+1)`; predicates bridge the gap by taking terms as arguments and having Boolean values. Terms and predicates do not have Boolean values as arguments.

The nuisance with this distinction is that sometimes one wants to have Boolean expressions as arguments of terms or predicates. Furthermore, making the distinction requires there to be an equality defined for formulas and one for terms, and similarly for inequality and if-then-else. When applying SMT solvers to software verification programs, one has to map programming language boolean types into a logical sort different than the conventional Boolean propositional sort.

The SMT-LIB language does *not* make a distinction between formulas and terms: formulas are simply terms whose sort is `Bool`. Boolean values can be defined to be the arguments of user-defined term-like functions just as can the values of any other sort. This simplifies the logic and the use of the SMT-LIB language. However, in using non-SMT-LIB solvers (such as when converting SMT-LIB constructs to be used by such solvers) it is helpful to remember that some solvers enforce this distinction.

3.1.5 Abstract and concrete syntax

The SMT-LIB language is described using an abstract syntax, but it is written with a particular concrete representation. For example, the language is defined using an abstract lexical category <*numeral*>; the concrete syntax for an element of that category is an unsigned decimal digit sequence with no leading zeros. While that representation is natural, others are not necessarily obvious. For example, the `Int` theory defines an add operation that takes two numeric arguments. The concrete syntax for an instance of that expression is `(+ 1 2)`; but an alternate concrete syntax might just as well be `(1 + 2)` or `(add 1 2)`. This tutorial only discusses the one concrete syntax described and standardized in [2], but the reader should be aware that alternate concrete syntaxes consistent with the SMT-LIB abstract syntax could also be defined. Note that the SMT-LIB document uses the abstract syntax only to facilitate the definition of the one concrete syntax and its semantics. The abstract syntax is not part of the SMT-LIB 2 standard, only the concrete syntax is.

3.2 Character set

SMT-LIB content is written using a subset of the ASCII character set[1]. The permitted characters are

- printable characters: those from `!` (ASCII character 33 (decimal)) through `~` (ASCII character 126), namely
  - digits: `0 1 2 3 4 5 6 7 8 9`
  - letters: `a` through `z` and `A` through `Z`
  - punctuation characters:

    `~ ‘ ! @ # $ % ^ & * ( ) _ - + = { } [ ] | \ ; : ’ " , . < > ? /`
- and whitespace characters
  - space (ASCII character 32)
  - tab (ASCII character 9, often written `\t`)
  - newline (ASCII character 10, often written `\n`, which is the line termination on UNIX systems)
  - carriage return (ASCII character 13, often written `\r`; `\r\n` is the two-character line terminator on Windows systems)

Different sets of characters are permitted in different places. The different subsets of the SMT-LIB character set are defined here for reference.

- digits: the characters `0` through `9`
- alphanumeric: the characters `0` through `9`, `a` through `z`, `A` through `Z`
- whitespace: space, tab, newline, and carriage return, as described above
- line terminators: newline and carriage return
- symbol characters: alphanumerics plus any of these punctuation symbols

  `~ ! @ $ % ^ & * _ - + = < > . ? /`

  Note that this list excludes the printable characters

  `" ’ ; : | { } [ ] ( ) ‘ , \ #`
- smt-lib character: any printable or whitespace character

[1] http://www.asciitable.com/

3.3 S-expressions

All input to and output from a conforming solver (using the standard concrete syntax) is a sequence of one or more S-expressions expressed using smt-lib characters, with optional interspersed comments. Comments are all of the text beginning with a semicolon that is not part of a string literal or quoted symbol up to but not including the first succeeding line terminator character (or end-of-input); comments are only for human consumption and are completely ignored by the parser. Since comments are always followed by line termination characters, they function as separating white space.

S-expressions have a particularly simple form, making them easy to parse and process. An S-expression is either (a) a token (defined in the next subsection) or (b) a sequence of 0 or more S-expressions enclosed in a pair of left and right parentheses. Tokens must be separated from each other by whitespace; there need be no white space on either side of a left or right parenthesis.

Here are some examples. The rules about what can be a token are described in the next section.

```
abc                          ; a single token
( abcdef   7   +-* )         ; an S-expression consisting of a sequence
                             ;      of three tokens
(+ 5 (* 2 3))                ; a more complex, nested S-expression
( )                          ; an empty S-expression
( abc "a b c d") (ghi jkl)   ; two successive S-expressions, each
                             ;      containing two tokens
( ABC                        ; a comment in the middle of an S-expression
  def )                      ; the end of an S-expression spanning two lines
```

Pairs of left and right parentheses are the only structure given to a sequence of tokens. A right parenthesis without a corresponding left parenthesis is fairly easy to detect. However, a missing right parenthesis may not be detected until the end of input (e.g. end of file) is reached; an input parser may have a hard time recovering from mis-matched parentheses—match them carefully!

3.4 Tokens

Not every sequence of smt-lib characters is a valid token. There are different kinds of tokens and they play different roles in different commands. In this section we describe the various tokens defined by SMT-LIB; a summary of token types is provided in Table 3.1. For convenience in referring to tokens, we use designators such as <*numeral*> so that later we can describe the syntax of commands by, for example, `( push` <*numeral*> `)` as the general case of `(push 0)`, `(push 1)`, and so on.

Token type	Format	Regular Expression (POSIX) [a]	Examples
<*numeral*>	a sequence of digits without leading zeros	`0|([1-9][0-9]*)`	`0 1 12 340`
<*decimal*>	a <*numeral*> followed by a period followed by a sequence of digits	`(0|([1-9][0-9]*))[.]([0-9]+)`	`0.0 0.1 1.0 12.0034`
<*binary*>	the two characters #b followed by a non-empty sequence of 0 or 1	#b[01]+	`0b0 0b1 0b101010101 0b11111`
<*hex*>	the two characters #x followed by a non-empty sequence of hex digits	#x[0-9a-fA-F]+	`0xA 0x51dead 0xFeeD`
<*string*>	a string literal: a sequence of smt-lib characters (including explicit white-space characters), enclosed in " characters; \" is interpreted as " and \\ as \	`"([]!#-[^-~ \t\r\n]|\\[!-~])*"`	`"" "a\"b" "a b" " b"`
<*symbol*>	either a <*simple-symbol*> or a <*quoted-symbol*>		
<*simple-symbol*>	a non-empty sequence of symbol characters, not beginning with a digit, and not a reserved word	`[a-zA-Z~!@$%^&*_+=<>.?/-] [0-9a-zA-Z~!@$%^&*_+=<>.?/-]*`	`a b* <=`
<*quoted-symbol*>	a sequence of smt-lib characters, not including \ and |, enclosed in | characters	`|[]!-[^-{}~ \t\r\n]*|`	`|a| || |a +|`
<*keyword*>	a colon followed by a non-empty sequence of symbol characters	`:[0-9a-zA-Z~!@$%^&*_+=<>.?/-]+`	`:abc`
Reserved words	character sequences that would otherwise be symbols but are reserved for special purposes and may not be used as symbols	`(let|par|_|!|as|forall|exists| NUMERAL|DECIMAL|STRING)` *and all command names*	`let par _ assert check-sat`

Table 3.1: Token types defined in SMT-LIB

[a] In these regular expressions the `()[]-+*|.\` characters have their usual[7] special meaning, with \ being the escape character.

Although the various kinds of literals have some conventional, intuitive meanings, their semantics is actually determined by the definition of the logic being used. <*decimal*>s might represent rationals in one logic and reals in another; <*binary*>s might be bit sequences in one logic and integers in another; even <*string*>s might be interpretable as integers or sequences.

Numerals. A numeric literal is expressed as a conventional sequence of digits. **Caution:** no leading zeros are allowed: `002` is not a valid <*numeral*>. Also no leading `+` or `-` signs are allowed: `-1` is a <*symbol*>, not a <*numeral*>.

Decimal literals. A decimal literal is a <*numeral*> followed by a decimal point, followed by more digits. As with <*numeral*>s, a <*decimal*> may not have leading zeros or sign characters. There must however, be at least one digit (possibly a `0`) before and at least one after the decimal point. There is no provision for exponents.

Binary literals. These literals, prefixed by `#b`, denote a simple sequence of `0`'s and `1`'s. Which end is the most significant bit or the beginning or end of a sequence will depend on the definition of the logic.

Hex literals. These literals, prefixed by `#x`, denote a sequence of hex digits. Though an interpretation as a bit sequence or an integer is reasonable, the actual interpretation depends on the logic being used.

String literals. String literals are composed of any sequence of printable and whitespace characters (including line terminations), enclosed in double quotes (`"`), with special treatment for `"` and `\` characters. A `"` character within the string is designated by the two-character sequence `\"` and the `\\` sequence represents a single `\`. The content of a string literal is uninterpreted. Any meaning is provided by the context of its use within SMT-LIB (e.g. as the value of an option) or by a theory. No currently defined theory incorporates string literals. Note that future theories may define escape sequences that allow including any ASCII character in the string, but none are defined at present (besides the two for `\` and `"`). So `\n` within a string literal just means those two characters, not necessarily a line terminator. **Caution:** String literals begin and end with a `"` character. Since string literals may contain explicit line terminator characters, an omitted closing `"` will not be noticed until the next `"` character or until the end of input. This may be several lines later.

Reserved words. Some character sequences are reserved words in SMT-LIB. They may only be used in specific contexts and may not be used as symbols. Hence they may not be declared by the user to represent user-defined quantities. The reserved words are

```
par NUMERAL DECIMAL STRING _ !  as let forall exists
```

and all the commands of the script language. Note particularly that `_` and `!` are reserved; their use is described in section 3.7. The currently defined commands are

```
assert check-sat declare-fun declare-sort define-fun define-sort
exit get-assertions get-assignment get-info get-option get-proof
```

```
get-unsat-core get-value pop push set-info set-logic set-option .
```

Though technically there is no need to identify reserved words and disallow their use as symbols, SMT-LIB does so as a convenience for (some kinds of) parsers. Thus a **caution:** future versions of SMT-LIB might possibly define new commands, which presumably would then disallow their use as symbols, potentially invalidating old SMT-LIB scripts in which those new command names were used as ordinary symbols. One should try to avoid using character sequences as symbols that might become command names in the future!

Symbols. Symbols are used for attribute values, and sort, variable, logic, theory, constant and function names. A symbol can be any sequence of symbol characters that does not start with a digit and is not a reserved word. In addition, user-defined symbols may not begin with an `@` or a period (`.`); such symbols are reserved for internal use. Some symbols consisting of punctuation characters, such as `+` and `-` have intuitively natural meanings as function symbols, though their meaning is always defined solely by the logic being used. Similarly, symbols that are English words may have obvious meanings. Good style would suggest avoiding legal but unusual symbols that mix alphanumerics and operator-like characters, such as `a<b` or `ab+-` or `-1`.

Quoted symbols. An alternate form of symbol is a sequence of any smt-lib characters except `|` and \, enclosed in beginning and ending `|` characters. This allows a wider choice of characters in forming the symbol. Note that explicit line terminators are allowed within a quoted symbol, so that a quoted symbol might span more than one line.[2] Note also that, because line terminators are allowed, a missing terminating `|` character may not be detected for several lines. Finally note that using enclosing `|`s does not change the identity of the symbol: `|abcde|` is the same symbol as `abcde`. A symbol such as `|{}|` does not have a corresponding unquoted symbol.

Keywords. A keyword consists of a `:` character followed by a sequence of symbol characters. They are used as the names of attributes, options and information flags. Good style would suggest that only alphanumeric characters (including perhaps the `-` and `_` characters) be used for user-defined keywords.

Invalid tokens. Some sequences of characters are not valid tokens:

- unclosed string literals and unclosed quoted symbols
- digit sequences or <*decimal*>-like sequences with leading zeros, such as `00`, `01`, or `00.0`. `1.000` is a valid <*decimal*>.
- decimal-like sequences with no digits after the decimal point (the sequence .123 is a symbol, but since it begins with a period, it may not be declared by the user)
- sequences of symbol characters beginning with digits, such as `0ABC`
- a # character not beginning a valid <*binary*> or <*hex*> token

[2] Quoted symbols (and strings) containing explicit line terminations can be hard to read. However, SMT-LIB was designed primarily as a machine-to-machine format and readability was not the major concern.

digit sequences with leading zeros	`00 0123`
decimals with leading zeros	`00.123 01.234`
decimals with no decimal digits	`1.`
would-be symbols that begin with digits	`1abc`
would-be symbols containing invalid characters	`abc'def`
invalid binary literals	`#b222`
invalid hex literals	`#xZZZZ`
unclosed strings	`"abc\"`
unclosed quoted symbols	`|asd`
quoted symbols with invalid characters	`|abc\def|`
keywords with invalid characters	`:abc[def`

Table 3.2: Invalid tokens

- use of printable but non-symbol characters outside of string literals or quoted symbols, such as a comma character
- purported keywords and quoted symbols that contain forbidden symbols

Table 3.2 shows some examples.

3.5 Sort and Function Declarations

SMT-LIB requires all identifiers to be declared before being used. Some symbols are declared in the selected logic and its theories. New symbols are introduced by the user in four commands:

`declare-sort` – which introduces a new sort symbol
`define-sort` – which introduces a new sort symbol as an abbreviation for a given sort expression
`declare-fun` – which introduces a new function symbol
`define-fun` – which introduces a new function symbol as an abbreviation for a given expression

These commands are described in full in subsections 3.9.3 and 3.9.4.

New sort symbols can be defined as new simple sort names or as parameterized sorts that take other sorts as parameters. Every value within an SMT-LIB logic has a specific sort; functions all have specific sort signatures that designate the sorts of the arguments and results.

New constants and function symbols are declared in a uniform way; constants are simply functions with no arguments. To declare a new function symbol, we need to specify the sorts of the

arguments and of the return value. For example, to declare a new symbolic constant value (a function with no arguments) named `x` of sort `Bool` we write:

```
(declare-fun x () Bool)
```

This statement declares a function `f` of two `Int`s, returning a `Bool`:

```
(declare-fun f (Int Int) Bool)
```

And this declares a function `ff` that returns an `Int` and takes one argument that is an `Array` of `Int` index type and `Bool` value type:

```
(declare-fun ff ( (Array Int Bool) ) Int)
```

3.6 Attributes

SMT-LIB uses attributes to attach meta-data to other syntactic entities. An attribute consists of a <*keyword*> or a <*keyword*>-value pair. As defined in section 3.4, a keyword is a `:` followed by a non-empty sequence of symbol characters. An attribute value (an <*attr-value*>) is in general any arbitrary S-expression, except that it may not be a <*keyword*>. But each individual <*keyword*> may have only a limited set of allowed values (or may not take a value at all). For example, the keyword `:left-assoc` has no value; the keyword `:print-success` may have only the values `true` and `false`. The attributes defined in SMT-LIB only have values that are <*symbol*>s, <*numeral*>s, or <*string*>s.

Attributes and attribute values are used in these circumstances:

- as option keys and values (cf. section 3.9.11)
- as information keys and values (cf. section 3.9.12)
- as part of a `!` expression (cf. section 3.7)
- to declare the various aspects of a theory or logic (cf. section 4)
- to describe associativity characteristics of functions defined in theories (cf. sections 4.1 and 3.7)
- to describe characteristics of a sort symbols defined in a theory (though no such attributes are currently defined)

3.7 Expressions

As is common in computer-parsable expression languages, SMT-LIB expressions (<*expr*>) are built recursively from a small set of basic forms. A valid expression is one of the following (the expression must also be well-sorted). Here and later in the tutorial, a + character indicates 1-or-more instances of the immediately preceding item and a $*$ indicates zero-or-more instances.

a numeric literal	<*numeral*>
a decimal literal	<*decimal*>
a string literal	<*string*>
a binary literal	<*binary*>
a hex literal	<*hex*>
an <*identifier*>, which is one of:	
a symbol:	<*symbol*>
an indexed identifier:	`( _` <*symbol*> <*numeral*>+ `)`
a *qualified-identifier*:	<*identifier*>
or	`( as` <*identifier*> <*sort*>`)`
a function application:	`(` <*qualified-identifier*> <*expr*>+`)`
a forall-expression:	`( forall ( (`<*symbol*> <*sort*>`)+ )` <*expr*>`)`
an exists-expression:	`( exists ( (`<*symbol*> <*sort*>`)+ )` <*expr*>`)`
a let-expression:	`( let ( (`<*symbol*> <*expr*>`)+ )` <*expr*>`)`
an attributed expression:	`( !` <*expr*> <*attribute*>+`)`

Literals. The constant literals are <*string*>, <*numeral*>, <*decimal*>, <*binary*>, and <*hex*>. Remember that the interpretation and sort of these literals depends on the theories that have been defined in the current solver environment (by a `set-logic` command). For example, the simplest logic, `QF_UF`, defines just the `Core` theory, in which none of the above literals are defined. <*numeral*> is used in most of the other theories; <*decimal*> is used in the `Reals` theory, and <*binary*> and <*hex*> are defined in the `Fixed_Size_BitVectors` theory.

Identifiers. Identifiers are available in all theories and are used as names of variables, parameters, functions, sorts, logics, theories, expressions, and as the values of some attributes. Identifiers are either simple symbols or indexed identifiers. The latter have the form `(_` <*symbol*> <*numeral*>+ `)` and allow the use of a family of identifiers that are indexed by one or more numerals. Indexed identifiers are only allowed in restricted settings:

- variables (in `forall`, `exists`, and `let` expressions) – symbols only
- sort and function names – theories may define indexed identifiers as sort and function names, but users may only introduce new symbols as sort and function names.
- parameters in `define-sort` and `define-fun` commands are always symbols
- logic and theory names are all pre-defined and are always symbols
- expression names (using the `!` expression syntax) are always symbols
- attribute values are sometimes symbols; since attribute values can be general S-expressions, they can possibly be indexed identifiers, but no pre-defined attributes do so

The only indexed sort identifier currently defined is `(_ BitVec` <*numeral*>`)`, the sort of bit-vectors whose length is the given numeral. For example, in the `Fixed_Size_BitVectors` theory, the constant `#b101` has sort `(_ BitVec 3)` and the hexadecimal constant `#x1A` has sort `(_ BitVec 8)`.

There is no way within (current) SMT-LIB to declare a whole family of identifiers. For example, one might like to write the following as a way of declaring a whole family of `C` identifiers to have `Int` sort

```
> (declare-fun (_ C n) () Int) ; INVALID SMT-LIB
```

to say that all identifiers of the form `(_ C` <*numeral*>`)` are `Int`s. Even more useful would be allowing the index to be used in the sorts, as in

```
> (declare-fun flip ((_ BitVec n)) (_ BitVec n)) ; INVALID
> (declare-fun (_ flip n) ((_ BitVec n)) (_ BitVec n)) ; INVALID
```

At present, however, such constructions must be pre-defined informally within theories and built in to the corresponding solvers.

Function expressions. The application of a function to a sequence of arguments is expressed simply as

`(` <*qualified-identifier*> <*expr*>`+ )` .

The syntax is uniformly prefix in style. No infix notation is used, even for conventional arithmetic operators. In SMT-LIB we write `(+ 1 2)`, not `(1 + 2)`.

There are a number of points to note:

- The function identifier must be already declared using a `declare-fun` or `define-fun` command, or have been pre-defined in the logic being used.
- The logic being used may restrict the kinds of expressions that are allowed. See section 4 for details about any particular logic.
- The number of arguments must correspond to the number of arguments declared for the function identifier. However, some theory-defined function symbols are defined to take two or more arguments. This behavior is specified by an attribute given to the function; such attributes may only be specified within theory definitions, not for user-declared functions. There are four such attributes. Here are examples of each:
  - left associative: `or` is a function defined in the `Core` theory that takes two `Bool` arguments, but can be applied to more using left associativity. `(or a b c d)` is equivalent to `(or (or (or a b) c) d)`. Other functions with the same behavior are boolean `and` and integer and real binary operations. Subtraction and division are also defined to be left-associative, so `(- a b c d)` is equivalent to `(- (- (- a b) c) d)` (in infix notation, `a-b-c-d` is equivalent to `(((a-b)-c)-d)`).
  - right associativity: The implies (`=>`) function is right associative, so `(=> a b c d)` is equivalent to `(=> a (=> b (=> c d)))`.
  - chainable: The equality and comparison (`< > <= >=`) functions are what SMT-LIB calls chainable – the operation is applied to each pair of arguments as one proceeds

down the list of arguments. `(= a b c d)` is equivalent to

```
(and (= a b) (= b c) (= c d)).
```

This is a natural definition for equality, but less-common, though convenient for comparisons.

- pairwise: all pairs of arguments are compared. The one pre-defined example is the `distinct` (not-equal) operation. `(distinct a b c d)` is

```
(and (distinct a b) (distinct a c) (distinct a d)
  (distinct b c) (distinct b d) (distinct c d))
```

.

- The expression must be well-sorted. That is, the arguments of the function must have the sorts that are specified in the definition of the function. There is no subtyping or inheritance of sorts. There is, however, a modest amount of overloading allowed within theory definitions. No overloading outside of a theory definition is permitted. Thus any `declare-fun` or `define-fun` command must introduce a completely new identifier. There are three kinds of pre-defined overloading.

  - parametric definitions. As an example, the equality function in the `Core` theory is defined to take two arguments of arbitrary but the same sort, and return a `Bool`. So = can be used, as would be naturally expected, for any user-defined sorts as well (responses are omitted):

```
> (set-logic QF_UF)
> (declare-sort A 0) ; a new sort named A
> (declare-sort B 0) ; a new sort named B
> (declare-fun a1 () A) ; a new constant of sort A
> (declare-fun a2 () A) ; another constant of sort A
> (declare-fun b1 () B) ; a new constant of sort B
> (declare-fun b2 () B) ; another constant of sort B
> (assert (= a1 a2)) ; comparing values of sort A
> (assert (= b1 b2)) ; comparing values of sort B
> (assert (= a1 b1)) ; ILLEGAL - a1 and b1 have different sorts
```

  - multiple definitions. The addition operation is not defined for arbitrary sorts, but it is defined both for `Int` and `Real` arguments, independently. So one can write (responses are omitted)

```
> (set-logic AUFLIRA)
> (assert (= (+ 1 2) 3)) ; adding two Ints
> (assert (= (+ 1.0 2.0) 3.0) ; adding two Reals
> (assert (= (+ 1 2.0) 3.0) ; ILLEGAL - mixed sorts
```

    As the last line of the example above demonstrates, there is no implicit conversion between `Int` values and `Real` values, nor between any other combination of sorts. The only connections between `Int` and `Real` values is through explicit conversion

functions such as `to_int` and `to_real` in the `Reals_Ints` theory.[3]

– **overloading on result sort.** A theory may also overload a function identifier such that two declarations of the function take the same argument sorts but return different result sorts. Suppose you declare a `List` sort that can take a sort as a parameter. You may also want a function named `emptylist` (with no arguments) that returns an empty list. But `emptylist` must return a result of a specific sort. To specify the result sort, we use a qualified identifier, as follows.

  To accomplish this example, the theory will declare

  * a `List` sort with one sort argument
  * a parametric function named `emptylist`, with sort parameter `A`, no arguments, and result sort `(List A)`

  Then that function might be used as follows (responses are omitted):

```
...
> (declare-fun listb () (List Bool)) ; listb is a List of Bool
> (declare-fun listi () (List Int)) ; listi is a List of Int
> (assert (= listb (as emptylist (List Bool))))
                          ; use the definition of emptylist
                          ; that has (List Bool) result
> (assert (= listi (as emptylist (List Int))))
                          ; use the definition of emptylist
                          ; that has (List Int) result
> (assert (= listb emptylist)); ILLEGAL - have to specify the
                               ; sort of the overloaded constant
> (assert (= listb (as emptylist (List Int)))) ; ILLEGAL -
                                               ; sort mismatch
```

Quantified expressions. The SMT-LIB language allows the two conventional quantified expressions — universal (*for all*) and existential (*there exists*). There is no syntax for the occasionally used quantifier meaning "there exists uniquely". Note that quantified expressions are not permitted in quantifier-free logics (e.g. `QF_UF`).

Both kinds of quantified expressions provide syntax to specify the names and sorts of parameters over which the expression quantifies. For example, the expression

```
( forall ((x Int)) (> (p x) 0) )
```

is true under the `AUFNIRA` logic and an assignment for `p` in which the sub-expression `(p x)` is positive for any `Int` value of `x`. Note the double pair of parentheses around "`x Int`". The outer pair encloses the whole list of parameters; each name-sort pair is enclosed in its own pair of parentheses. So if there are two quantification variables, we would write

[3] The AUFLIRA and AUFNIRA logics currently state that ints are implicitly converted to reals in some circumstances, but the discussion on the mailing lists concluded this should not be allowed.

```
( exists ((x Int)(y Int)) (and (> (+ x y) 0) (< (- x y) 0)))
```

The value of this example expression is `true` since there are specific integer values for `x` and `y` that are solutions to the two inequalities.

The scope of the quantification variables is limited to the sub-expression that is the target of the quantification expression. The names of the quantification variables must be <*symbol*>s, not <*identifier*>s; they may shadow previously defined variables or functions, but only do so within the target expression.

Let expressions. Let expressions provide the ability to make expressions more compact by abbreviating common sub-expressions. A let-expression has the form

`(` `let` `(` <*binding*>+ `)` <*expr*> `)`

Each <*binding*> associates a symbol and an expression; it is simply a parenthesized pair:

`(` <*symbol*> <*expr*> `)` .

The symbol is an abbreviation for its expression within the target expression of the let construct; the sort of the new symbol is the same as the sort of the expression to which it is bound. The whole let construct is entirely equivalent to replacing each new parameter by its expression in the target expression, eliminating the new symbols completely (renaming functions as needed to avoid name capture by variable bindings).

As with quantified expressions, the new symbol is in scope only within the target expression and can shadow previously declared variables and functions within that scope. Also, the new symbols are not in scope in any of the bound expressions — it is a "parallel let". For example, presuming `x` and `y` are already defined prior to these `let` expressions in the context of the `QF_LIA` logic,

```
( let ( (a (+ x y)) (b (- x y)) ) (+ a b))
```

is equivalent to

```
(+ (+ x y) (- x y))
```

.

These are both also equivalent to

```
( let ( (x (+ x y)) (y (- x y)) ) ( + x y))
```

since the `x` and `y` in the first `(+ x y)` and `(- x y)` refer to the already defined `x` and `y`, but the `x` and `y` in the last `(+ x y)` refer to the new parameters of the let expression.

Attributed expressions. Attributed expressions allow additional information to be associated with an expression. The value of the expression is not changed. Though additional kinds of attributes (cf. section 3.6) may be defined in the future or by solvers with their own extensions, within SMT-LIB, the only defined behavior within an expression is to name the (sub-)expression. The general form of the attributed expression is

`(!` <*expr*> <*attribute*>+ `)`.

Specifically, to name an expression one writes

`(!` <*expr*> `:named` <*symbol*> `)`.

The <*symbol*>is the name of the expression. The expression being named may not contain any variables (as declared in `forall`, `exists`, or `let` expressions), only function symbols.

There are two uses for named expressions.

The first is to identify which expressions are to be reported as part of the unsatisfiable core when a group of assertions are found to be unsatisfiable or as part of a `get-assignment` command if the assertions are found to be satisfiable. Refer to sections 3.9.7 and 3.9.8 for more information about these commands.

The second use is syntactic sugar for abbreviating sub-expressions. For example, writing

```
(assert (> (!  (+ x y) :named sum) 0))
```

is equivalent to the pair of commands

```
(define-fun sum () Int (+ x y))
(assert (> sum 0))
```

This implicit use of `define-fun` may only be used to introduce new function symbols that do not have arguments. Also, the sub-expression being abbreviated may not refer to any parameters of a let or quantified expression (that is, the sub-expression must be *closed*). The new symbol may be used within the same larger expression, as long as it occurs "after" its introduction. So we can write

```
(assert (>= (!  (+ x y) :named sum) sum))
```

but we cannot use `sum` before it is introduced:

```
(assert (>= sum (!  (+ x y) :named sum)))    ; ILLEGAL
```

In the latter example, there are two cases: if `sum` is already declared as a symbol before this assert command, then naming a new expression with the same name is not permitted; if `sum` is not already declared, then the first use of `sum` is illegal because the symbol is not defined.

When used only to abbreviate sub-expressions within the same expression, the let-expression is probably a better alternative. Since an expression name introduces a new function symbol, the expression name may not be the name of an already declared function.

When introducing variables or eliminating variables using substitution, always be careful of name conflicts caused by shadowing. Inadvertent use of a common name can cause a change in meaning.

3.8 Namespaces and Scopes

SMT-LIB uses <*symbol*>s and <*identifier*>s in multiple places within the language:

1. as sort names – new sort identifiers are introduced in theories and new sort symbols in `declare-sort` and `define-sort` commands
2. as function names – new function identifiers are introduced in theories and new function symbols in `declare-fun` and `define-fun` commands
3. as names of expressions (in `!` expressions), which are implicitly declarations of new function symbols – new symbols need no declaration, they are simply introduced as used (symbols only)

4. as a variable binding within a `forall`, `exists`, or `let` expression (symbols only)
5. as logic names – these are predefined (symbols only)
6. as theory names – these are predefined (symbols only)
7. as attribute values (just <*symbol*>s) – new values need no declaration, they are simply introduced as used
8. as pre-defined responses to commands (e.g. `success`) – these are defined as part of the specification of the command
9. as command names (which are technically reserved words and not symbols)

Only the function and variable symbols (categories 2, 3, and 4) are in the same namespace. All of the others may be declared or introduced independently of each other.

New sort symbols must be different from any previous sort symbol. No shadowing[4] or overloading[5] is allowed.

Function symbols may be overloaded when defined within a theory, but user-defined function symbols must be new function symbols. User-defined function symbols may not shadow or overload any existing function symbols. Variables introduced in quantified or let expressions may, however, shadow previously declared function symbols or variables; the scope of variables is only the subterm in which they are declared.

Names of expressions also introduce function names. These therefore must also be different from any already defined function names.

Logic and theory names are introduced in logic and theory definitions, respectively. Names of logics are used only in `set-logic` commands; names of theories are used only in logic definitions. Neither of these uses can be confused with other uses of symbols. However, since logics and theories have associated definition files (which may well be in the same directory), it is expected that the names of logics and theories will be distinct.

Values of attributes only appear after <*keyword*>s and in responses to `get-option` and `get-info` commands; other command responses also may contain symbols such as `success`, `unsupported`, `error`. A tool that parsed command responses may need to distinguish these, disambiguating them by the context in which they are produced.

Scopes. The only names affected by the scopes introduced with the `push` and `pop` commands are function and sort names, including the function names introduced by attributed expressions. Variables in `forall`, `exists`, and `let` expressions have a scope that extends to just the subexpression in which they are declared.

[4] A symbol *shadows* one of the same name if, within the scope of the new symbol, only the new symbol is recognized; the shadowed symbol is unavailable in expressions until the scope of the shadowing symbol is exited.

[5] A symbol *overloads* a symbol of the same name if both symbols are available within the new symbol's scope; the overloaded symbols are distinguished by, for example, the sorts of their arguments

3.9 Commands and command output

A conforming solver will support the list of commands shown in table 3.3. Each command has a set of outputs that it may produce. Those are described with each command in the following subsections. However, there are a few general points to be made.

- **Form of output.** All outputs are S-expressions. That is, the output is an individual token, such as an identifier (e.g. `success`), or a parenthesized list of tokens, such as an error message, which has the form `( error` <*string*> `)`.
- **Typical output.** Most commands will produce either `success` or an error message.
- **Error messages.** The form of an error message is defined - it is an S-expression consisting of the token `error` and a string literal; however the content of the string literal in an error message is entirely solver-specific. It may even be the empty string, as in `( error "" )`.
- **Unsupported commands.** Some commands may respond with `unsupported`. This response indicates that the command (or the command combined with the particular arguments) is optional in the SMT-LIB standard and is not supported by the given solver. Solvers are required to respond with `unsupported` to defined but optional features they do not recognize, rather than, for example, crashing or emitting an error message.
- **Identifiers vs. strings.** Some commands take <*identifier*>s as arguments, others take <*string*>s. These are different. For example `( set-logic QF_UF )` is a valid command, but `( set-logic "QF_UF" )` is not.
- **Continuing after errors.** On encountering an error, a solver has two options. It can terminate immediately or it can continue to process other commands. Immediate termination is most appropriate for batch execution, but is very user-unfriendly in interactive mode. If a solver continues after an error, it is required that the erroneous command have no effect on the state of the solver. (cf. subsection 3.9.12)

  Note: The hard distinction between always continuing and always immediately exiting in response to an error is not entirely practical. Some solvers may attempt to continue whenever possible (e.g. upon encountering syntax errors or out of place commands), but will not be able to continue (or at least preserve state) in the face of runtime errors such as lack of enough memory. More importantly, the user may wish to choose different behaviors in different circumstances, such as in interactive vs. batch execution.

A command script is a sequence of commands, typically contained in a single file. Though some variations are permitted, a typical order of the commands in a script is this sequence:

- set any options desired using `set-option`

Command name and format	when allowed
`( set-logic <`*symbol*`> )`	only once
`( declare-fun <`*symbol*`>` `(<`*sort-expr*`>*) <`*sort-expr*`>)`	after `set-logic`

`( define-fun <`*symbol*`>`
`((<`*symbol*`> <`*sort-expr*`>)*)`
`<`*sort-expr*`> <`*expr*`>)`
	after `set-logic`

`( declare-sort <`*symbol*`> <`*numeral*`>)`	after `set-logic`
`( define-sort <`*symbol*`>` `(<`*symbol*`>+) <`*expr*`> )`	after `set-logic`
`( assert <`*expr*`> )`	after `set-logic`
`( get-assertions )`	interactive mode, after `set-logic`
`( check-sat )`	after `set-logic`
`( get-proof )`	with `:produce-proofs` option set true and after `check-sat` that returns `unsat`, without intervening assertion-set commands[a]
`( get-unsat-core )`	with `:produce-unsat-cores` option set true and after `check-sat` that returns `unsat`, without intervening assertion-set commands
`( get-value <`*expr*`>+ )`	with `:produce-models` option set true and after `check-sat` that returns `sat` or `unknown`, without intervening assertion-set commands
`( get-assignment )`	with `:produce-assignments` option set true and after `check-sat` that returns `sat` or `unknown`, without intervening assertion-set commands
`( push <`*numeral*`> )`	after `set-logic`
`( pop <`*numeral*`> )`	after `set-logic`
`( get-option <`*keyword*`> )`	anytime
`( set-option <`*keyword*`> <`*attr-value*`> )`	anytime, with some options restricted to being set before a `set-logic` command
`( get-info <`*keyword*`> )`	anytime
`( set-info <`*keyword*`> <`*attr-value*`> )`	anytime
`( exit )`	anytime

Table 3.3: SMT-LIB commands

[a] an assert, push or pop, or definition or declaration command

+ indicates one-or-more repetitions, * indicates 0-or-more of the preceding item

- `set-logic` command
- declarations and definitions
- `assert` commands
- repetitions of
  - `push`
  - perhaps additional declarations and definitions
  - `assert` commands
  - `check-sat`
  - if unsatisfiable: optional calls of `get-proof` and/or `get-unsat-core`
  - if satisfiable: zero or more calls of `get-assignment` and/or `get-value`
  - `pop`
- `exit`

3.9.1 Initialization: the `set-logic` command

The `set-logic` command has the form:

`( set-logic` <*symbol*> `)`

The command has the effect of initializing the solver with the specified logic. The given <*symbol*> must correspond to a defined logic; note that the argument is a <*symbol*>, not a <*string*>. All the logics defined in SMT-LIB are presented and discussed in chapter 4; solvers may optionally support additional non-standard, solver-specific logics. An example is the `QF_UF` logic, and the corresponding example command is

```
> ( set-logic QF_UF )
success
```

All solvers should recognize the SMT-LIB defined logics, but a solver need not support all logics. Thus the `set-logic` command may have these responses:

- `success` - the solver was successfully initialized with the given logic
- `unsupported` - the solver does not support the given logic
- error - the argument is not a <*symbol*> or the named logic is not a defined logic

A `set-logic` command may be given only once in a given execution of a solver. The command must precede any definition, declaration, assert or check-sat commands. Only exit, option and info commands may precede a set-logic command. [6]

3.9.2 Termination: the `exit` command

The `exit` command has a simple form:

`( exit )`

The command should always return `success` and then the solver should terminate. There is no expectation of any persistent state of the solver. The standard does not allow simply returning without issuing a response to the command. A solver will return an error if the command is malformed or if for some reason the solver fails to cleanly terminate.

[6] The restriction to a single `set-logic` command in the life of a solver is simply for simplicity. An alternate rule (not conforming SMT-LIB) could be that a second `set-logic` command causes a reset of the solver to the new logic (perhaps retaining current option values).

3.9.3 Defining new sorts: `declare-sort` and `define-sort`

There are two commands that introduce new sort symbols: `declare-sort` declares a new sort and `define-sort` introduces a new symbol that is the abbreviation for a sort expression.

Sort symbols are in a different namespace than other symbols (e.g. function symbols), so their names do not conflict or shadow each other. A symbol cannot be declared or defined as a new sort symbol if there is already a sort symbol with that name. Declarations of sort symbols are removed when the scope in which they are declared is deleted by a `pop` command.

declare-sort

The `declare-sort` command has a simple form. Its arguments are the <*symbol*> that is the new sort symbol and a <*numeral*> that is the number of parameters of the new sort symbol:

`(declare-sort` <*symbol*> <*numeral*>`)`

A sort symbol with no parameters can be used as a new sort. For example, one can write

```
...
> (declare-sort MySort 0) ; new sort named MySort
success
> (declare-fun x () MySort) ; using MySort as the sort of a new constant
success
```

A sort symbol with parameters is not a sort itself, but is used to create a sort when combined with the correct number of parameters. The parameters are always other sorts. Suppose `X` and `Y` are already defined sorts. Then writing

`(declare-sort Pair 2)`

declares a new parameterized sort symbol named `Pair` that takes two parameters. The following are then three different sorts:

```
(Pair X X)
(Pair X Y)
(Pair Y Y)
```

The response to a `declare-sort` command is either

- `success`
- an error response if
  - the arguments are malformed - wrong number or wrong kinds of tokens
  - the given symbol is already declared as a sort symbol
  - the command is used before a `set-logic` command

define-sort

The `define-sort` command defines a new sort symbol that is an abbreviation for a sort expression. The new sort symbol can be parameterized, in which case the names of the parameters are specified in the command and the sort expression uses the sort parameters. The form of the command is this:

(`define-sort` <*symbol*> (<*symbol*>`*`) <*sort-expr*>)

A <*sort-expr*> is either a sort symbol or a sort symbol applied to sort arguments — one of these:

<*symbol*>

(<*symbol*> <*sort-expr*>`+`)

For example,

`(define-sort I () Int)`

defines `I` to be an abbreviation for `Int`. A use of `I` is then fully equivalent to a use of `Int`. The empty parentheses are required to indicate that `I` has no parameters.

To demonstrate a parameterized definition, assume that `Int` and `Real` are already defined sorts and that `Pair` is a sort symbol taking two parameters. The we can define sort symbol `P` with one parameter by

`(define-sort P (X) (Pair X X))`

Then `(P Int)` is equivalent to `(Pair Int Int)` and `(P Real)` is equivalent to `(Pair Real Real)`.

The <*symbol*>s used as parameters may be any <*symbol*>s (whether or not they have existing declarations or definitions), as long as they are different from each other within the `define-sort` command. They will shadow existing sort symbols of the same name.

The response to this command should be either

- `success` - in which case the new sort symbol is now part of the current scope
- error -
  - if the command is malformed
  - if the command is used before a `set-logic` command
  - if the symbols used in the definition of the new sort are themselves not yet defined
  - if the symbol being declared is already defined
  - if there are duplicate symbols used as parameters

3.9.4 Defining new function symbols and constants: `declare-fun` and `define-fun`

SMT-LIB requires all identifiers to be declared before being used. The `declare-fun` command is used to declare new symbols. Constants and functions are declared in a uniform way; constants

are simply functions with no arguments. To declare a new function symbol, we need to specify the sorts of the arguments and of the return value. The form of the command is this:

`(declare-fun` <*symbol*> `(` <*sort-expr*>`*` `)` <*sort-expr*>`)`

The list of sorts within parentheses are the sorts of the arguments; the sort that is the last argument of `declare-fun` is the sort of the result. To declare a constant, there are no sorts in the parentheses, but the parentheses are still present. Users may only introduce new <*symbol*>s; new <*identifier*>s appear only in theory definitions.

The response to this command should be either

- `success` - in which case the new function symbol is now part of the current scope
- error -
  - if the command is malformed
  - if the command is used before a `set-logic` command
  - if the symbols used in the definition of the new function symbol are themselves not yet defined (so no recursion is allowed)
  - if the symbol being declared is already defined

So to declare a new symbolic constant value (a function with no arguments) named `x` of sort `Bool` we write:

```
(declare-fun x () Bool)
```

This command declares a function `f` of two `Int`s, returning a `Bool`:

```
(declare-fun f (Int Int) Bool)
```

And this declares a function `size` that returns an `Int` and takes one argument that is an `Array` of `Int` index type and `Bool` value type:

```
(declare-fun size ( (Array Int Bool) ) Int)
```

It is also possible to declare function symbols whose sort is parametric, but this is allowed only within a theory definition and does not use the `declare-fun` syntax. More information about writing theory definitions is found in section 4.1.

The `declare-fun` command declares a new function symbol that is wholly uninterpreted. One can also declare function symbols to be equivalent to a given expression, using the `define-fun` command. These simply serve as abbreviations. For example, the command

```
(define-fun av ((p Int)(q Int)) Bool (or (> p (+ q 2)) (< p (- q 2))))
```

declares a new binary function symbol `av` taking `Int` arguments, returning a `Bool`; the value is `true` when the absolute value of the difference between the two arguments is more than 2.

The general form of the `define-fun` command is similar to that of the `declare-fun` command, with just the addition of the defining expression:

`(define-fun` <*symbol*> `(` <*sort-expr*>`*` `)` <*sort-expr*> <*expr*>`)`

The response to the command is either

- `success` - in which case the new function symbol is now part of the current scope
- error -
  - if the command is malformed
  - if the command is used before a `set-logic` command
  - if the symbols used in the definition of the new function symbol are themselves not yet defined (so no recursion is allowed)
  - if the symbol being declared is already defined
  - if the defining expression is not well-sorted

The symbol introduced by a `define-fun` command is simply an abbreviation. One could always substitute uses of the symbol by instances of the defining expression, with the parameters substituted with the actual arguments (renaming variables where necessary to avoid variable capture by forall, exists, or let expressions). The goal of using such abbreviations is to simplify expressions for the human reader and, perhaps, for the solver.

3.9.5 Asserting logical statements: the `assert` command

The `assert` command instructs the solver to assume that the stated formula is true. The form of the command is

`( assert` <*expr*>`)`

in which <*expr*> is a properly constructed, well-sorted SMT-LIB expression, whose sort is `Bool` (cf. section 3.7).

An `assert` command may not appear before a `set-logic` command, since the logic is needed to provide the definitions of the sort and function symbols used in the expression.

An `assert` command responds with either

- `success` - in which case the new assertion is now part of the solver context
- an error message -
  - if the command is malformed
  - if the command is not well-sorted, e.g. the sort of the whole expression is not `Bool`
  - if the command is misplaced (e.g. before a `set-logic` command)
  - if the expression is not allowed in the current logic

3.9.6 Checking satisfiability: the `check-sat` command

The main purpose of a SMT solver is to check the satisfiability of a set of asserted expressions. Once a series of `assert` commands have been made, the `check-sat` command will instruct the

solver to test for satisfiability. The command has a simple form:

```
( check-sat )
```

The command may respond with one of three responses:

- `sat` : the set of assertions is satisfiable. That is, there is a consistent set of assignments of values to constants and function symbols under which all asserted expressions are true. In this case, the `get-assignment` and `get-value` commands may be used to find out more about the satisfying assignment.
- `unsat` : the set of assertions is unsatisfiable. That is, there is no set of assignments of values to constants and function symbols under which all asserted expressions are true. In this case, the `get-proof` and `get-unsat-core` commands may be used to find out more about the unsatisfiable assertions.
- `unknown` : solver cannot determine for sure whether the set of assertions is satisfiable or not. There may be various reasons for this. In some cases, the solver may have run out of memory or time. In others, the solver may have found a set of assignments that appears to make all asserted expressions true, but it cannot be sure because of the presence of quantified assertions. The `:reason-unknown` info item (section 3.9.12 below) can be used to ask for more information about why the solver responds with `unknown`.

The command will respond with an error if there are any arguments to the command or the command precedes a `set-logic` command.

3.9.7 `sat` operations: `get-value` and `get-assignment`

When the `(check-sat)` operation produces a `sat` response, then we know that the set of all the current assertions is satisfiable. With some assignment of values to the free constant and function symbols, all the asserted formulae are true. There are two (optional) commands that enable determining the values that make up the assignment: `get-value` and `get-assignment`. These commands may only be used under these conditions:

- for `get-assignment`, the `:produce-assignments` option must have been set to true (which is not the default) before the `set-logic command`;
- for `get-value`, the `:produce-models` option must have been set to true (which is not the default) before the `set-logic` command;
- a `(check-sat)` command must have been given to the solver, and the most recent such command must have returned either `sat` or `unknown` (it is optional for a solver to support these commands when `unknown` is returned);

- since the most recent `(check-sat)` command, no `push`, `pop`, `assert`, `declare-fun`, `define-fun`, `declare-sort`, or `define-sort` command has been issued.

The command will respond with an error if there are any arguments to the command or the command precedes a `set-logic` command.

get-value

The `get-value` command takes one argument that is a parenthesized list of terms and responds with the values of those terms under the current assignment. Note that there may be more than one satisfying assignment; with the same set of asserted formulae, on two different occasions with the same or different ordering of terms, different values for the arguments might be produced.

The terms in the `get-value` command must be quantifier-free terms without new function (or constant) symbols; that is, the terms are ground terms using only the function (and constant) symbols already defined. More specifically, each theory defines the set of values that belong to each sort; for example, the values for the `Int` sort are <*numeral*>and `(-` <*numeral*>`)` (for a non-zero <*numeral*>). The values supplied by the `get-value` command are given in terms of these basic values for each sort, or in terms of unspecified abstract values.

The response to the command is a parenthesized list of parenthesized pairs, one pair for each term listed in the `get-value` command; each pair contains the given term and its value. The response may also be an error message if

- the command is syntactically mal-formed
- any term contains undefined symbols, is not valid for the given logic, or is otherwise not well-sorted
- there is no sufficiently recent `check-sat` command that returned `sat` or `unknown`
- the `:produce-models` option is not set to `true`

The `get-value` capability is optional. If it is not supported, then the `set-option` command applied to `:produce-models` must return `unsupported`, and the `get-value` command will respond with an error. In addition, the solver will respond `unsupported` to a `get-value` command if the most recent `check-sat` result was `unknown` and the solver does not support returning model information for `unknown` states.

Here is an example, without `success` responses, presuming the solver supports `get-value`.

```
> (set-option :produce-models true)
> (set-logic QF_LIA)
> (declare-fun x () Int)
> (declare-fun y () Int)
> (assert (= (+ x y) 9))
> (assert (= (+ (* 2 x) (* 3 y)) 22))
> (check-sat)
sat
> (get-value (x))
((x 5))
> (get-value ((- x y) (+ 2 y)))
(((- x y) 1) ((+ 2 y) 8))
```

get-assignment

The `get-assignment` command also responds with a list of assignments: the (sub-)terms whose values are returned are those with `Bool` sort that are named at the time they are asserted. A term is named using this syntax (cf. section 3.7):

`(!` *<expr>* `:named` *<symbol>*`)`

In general, the named term may be a sub-term or may be the entire asserted formula. However, the `get-assignment` command only returns the truth assignment of named terms or subterms of sort `Bool`.

The form of the `get-assignment` command is simply

`( get-assignment )`

The response to the command is a parenthesized (S-expression) list of zero or more pairs, one pair for each named asserted formula or named subterm of `Bool` sort in the current solver context; each pair contains the *<symbol>* naming a formula and its value (either `true` or `false`, as symbols). The response may also be an error message if

- the command is syntactically malformed
- there is no sufficiently recent `check-sat` command that returned `sat`
- the `:produce-assignments` option is not set to `true`

The `get-assignment` capability is optional. If it is not supported, then the `set-option` command applied to `:produce-assignments` must return `unsupported`, and the `get-assignment` command must respond with an error. In addition, the solver will respond `unsupported` to a `get-assignment` command if the most recent `check-sat` result was `unknown` and the solver does not support returning model information for `unknown` states.

Here is a simple example:

```
> (set-option :produce-assignments true)
success
> (set-logic QF_UF)
success
> (declare-fun p () Bool)
success
> (declare-fun q () Bool)
success
> (declare-fun r () Bool)
success
> (assert (not (=> (or (!  p :named P) (!  q :named Q)) (!  r :named
R))))
success
> (check-sat)
sat
> (get-assignment)
((P true)(Q true)(R false))
```

In the example, the response to `get-assignment` says that when p and q are `true` and r is `false`, then the asserted expression is `true`, that is, then $(p \vee q) \Rightarrow r$ is `false`.

3.9.8 `unsat` operations: `get-proof` and `get-unsat-core`

Proofs

When the `(check-sat)` operation produces an `unsat` response, then we know that the set of all the current assertions is unsatisfiable. Equivalently, all of the assertions but the last together imply the negation of the last assertion. This is, in fact, a common method of determining whether a set of assertions, P_i, implies a consequence, Q: assert each of the P_i, then assert $\neg Q$, and then check for satisfiability. If not satisfiable, then the implication holds.

However, in that case, we may well want more information: we would like a step-by-step proof that the implication holds. Some solvers may be able to supply such a proof. If so, it can be obtained using the `(get-proof)` command.

The `(get-proof)` capability is optional. If it is supported, then to obtain proofs, the following conditions must hold:

- the `:produce-proofs` option must be `true` (which is not the default); the option may only set before the `set-logic` command;

- a `(check-sat)` command must have been given to the solver, and the most recent such command must have returned `unsat`;
- since the most recent `(check-sat)` command, no `push`, `pop`, `assert`, `declare-fun`, `define-fun`, `declare-sort`, or `define-sort` command has been issued.

If the `get-proof` capability is not supported, then the `set-option` command applied to `:produce-proofs` must return `unsupported` and the `get-proof` command must respond with an error.

The form of the command is simple:

```
( get-proof )
```

The response to the command is one of these:

- a proof - The form of the proof is not specified by the SMT-LIB standard, except that the concrete syntax of the result must be an S-expression.
- an error response -
  - if the command is malformed (e.g. has arguments)
  - if there is no sufficiently recent `check-sat` command that returned `unsat`
  - if the `:produce-proofs` option is not set to `true`

Here is an example of using a solver to check the truth of $((p \Rightarrow q) \wedge q \Rightarrow r)) \Rightarrow (p \Rightarrow r)$ (presuming the solver supports proof generation):

```
> (set-option :produce-proofs true)
success
> (set-logic QF_UF)
success
> (declare-fun p () Bool)
success
> (declare-fun q () Bool)
success
> (declare-fun r () Bool)
success
> (assert (=> p q))
success
> (assert (=> q r))
success
> (assert (not (=> p r)))
success
> (check-sat)
unsat ; (=> p r) is always true, given the previous assertions
> (get-proof)
Solver-dependent output
```

Unsatisfiable cores

Another interesting question about an unsatisfiable set of assertions is which assertions in that set are actually causing the contradiction that makes the set unsatisfiable. Such a set, ideally smaller than the full set, but still unsatisfiable, is called an *unsatisfiable core*. One might even want a *minimal* unsatisfiable core.[7] The `get-unsat-core` command responds with an unsatisfiable core, though the core is not necessarily minimal. In fact, it might provide the unhelpful result of listing all of the named formulae.

The command takes no arguments and so has the simple form

```
(get-unsat-core)
```

It is subject to similar restrictions as the `(get-proofs)` command. If it is supported, then to obtain an unsatisfiable core, the following conditions must hold:

- the `:produce-unsat-cores` option must be `true` (which is not the default); the option may only be set before the `set-logic` command;
- a `(check-sat)` command must have been given to the solver, and the most recent such command must have returned `unsat`;
- since the most recent `(check-sat)` command, no `push`, `pop`, `assert`, `declare-fun`, `define-fun`, `declare-sort`, or `define-sort` command has been issued.

If the `get-unsat-core` capability is not supported, then the `set-option` command applied to `:produce-unsat-cores` must return `unsupported`, and the `get-unsat-core` command must respond with an error.

An error response may occur

- if the command is malformed (e.g. has arguments)
- if there is no sufficiently recent `check-sat` command that returned `unsat`
- if the `:produce-unsat-cores` option is not set to `true`

The successful result of the command is a list of formulae, identified by their names (which are <*symbol*>s). Formulae are given names when they are asserted (for this command, names of any subterms are ignored). For example, one might issue this command to a solver:

```
(assert (!  (or p q) :named FORMULA1))
```

The exclamation point is the annotation operator, indicating that the term that is the first argument has the annotations given in the rest of the S-expression (cf. section 3.7). Note that the name of the term is a <*symbol*>, not a <*string*>. Only top-level formulae with names are listed in the

[7] It is a computationally hard problem to find minimal unsatisfiable cores; solvers that do provide unsat cores generally do not guarantee minimality.

output of the `(get-unsat-core)` command; unnamed formulae are simply not listed, whether or not they are part of the unsatisfiable core.

The rule about naming formulae serves two purposes. First, it allows the user to be selective about which formulae to track, avoiding what might be extraneous information; second, it allows an implementor to know which formulae need to be tracked, allowing potential optimizations. However, if the user forgets about the distinction between named and unnamed formulae, it may appear that the solver has produced an unsatisfiable core that is in fact satisfiable; in effect, the unsatisfiable core must be considered to include all of the unnamed formulae.

Here is a simple example (without `success` responses) that presumes the solver supports producing unsatisfiable cores:

```
> (set-option :produce-unsat-cores true)
> (set-logic QF_UF)
> (declare-fun p () Bool)
> (declare-fun q () Bool)
> (declare-fun r () Bool)
> (declare-fun s () Bool)
> (declare-fun t () Bool)
> (assert (!  (=> p q) :named PQ))
> (assert (!  (=> q r) :named QR))
> (assert (!  (=> r s) :named RS))
> (assert (!  (=> s t) :named ST))
> (assert (!  (not (=> q s)) :named NQS))
> (check-sat)
unsat
> (get-unsat-core)
(QR RS NQS)
```

The names of the formulae in the unsatisfiable core may be in any order, and may include other names, but should include at least these three.

3.9.9 Adding scope: the `push` and `pop` commands

The `push` and `pop` commands enable some scoping of sort and function declarations and of assertions. In effect, a solver maintains a single, global stack of sets of assertions. The stack is initialized with one empty assertion set; that assertion set is, initially, the top set on the stack. Declaration, definition and assert commands add new declarations, definitions and assertions to the top assertion set of the stack.

The forms of the commands are

`( push` <*numeral*>`)`

(pop <*numeral*>)

Recall that a <*numeral*> is always non-negative. If <*numeral*> is zero, the command is legal but has no effect.

A push operation adds the given number of new empty assertion sets to the top of the stack, with the last one added becoming the new top assertion set and the new recipient of declarations, definitions, and assertions. A push command with argument *n* is precisely equivalent to *n* (push 1) commands.

A pop operation removes the given number of assertion sets from the stack, beginning with the top-most, in reverse order of being pushed. It is as if the commands that placed declarations, definitions, and assertions in those now popped sets had never happened.

The check-sat and get-assertions commands always operate on the union of all of the assertion sets on the stack at the time of the command.

It is an error to pop off more assertion sets than have been added; it is not permitted to pop the assertion set with which the stack is initialized; thus there is always an assertion set to receive new declarations, definitions, and assertions. The push and pop commands are not permitted prior to a set-logic command. Note that if the argument to a pop command is too large, an error response is given, and no assertion sets are popped; this behavior is consistent with the rule that if a command responds with an error then the state of the solver is unchanged.

The push and pop commands provide a scope for declarations, definitions, assertions, and names of expressions within assertions. The commands do not affect the setting of options (set-option) or information items (set-info).

The responses to push and pop are either

- success
- an error response
  - if the command is malformed, e.g. anything other than just one argument that is a <*numeral*>
  - if the argument to a pop command is more than the current net number of pushed assertion sets
  - if the command precedes a set-logic command

A very large argument to the push command risks exhausting resources, depending on the solver implementation.

The push and pop commands are typically used to try various experiments on a set of assertions

(success responses are omitted).

```
> (set-logic QF_LIA)
> (declare-fun x () Int) ; declare some constants
> (declare-fun y () Int)
> (declare-fun z () Int)
> (push 1)
> (assert (= (+ x y) 10))
> (assert (= (+ x (* 2 y)) 20))
> (check-sat)
sat ; there is a solution
> (pop 1) ; clear the assertions
> (push 1) ; ready for another problem
> (assert (= (+ (* 3 x) y) 10))
> (assert (= (+ (* 2 x) (* 2 y)) 21))
> (check-sat)
unsat ; no solution
> (declare-fun p () Bool)
> (pop 1)
> (assert p)
( error "p is not declared") ; the declaration of p was popped
```

3.9.10 Remembering what you have done: the `get-assertions` command

The `get-assertions` command simply responds with all of the expressions in the current assertion set stack. It has this simple form:

```
( get-assertions )
```

The response is a parenthesized list of all of the asserted expressions (but not the declarations or definitions[8]). Each element of the response is an expression. The expression must be identical (except for whitespace and comments)[9] to the expression in the original assert command: no normalization of the expression, reordering of arguments, flattening of associativity, substitution of definitions, or any other rearrangement is allowed. The purpose of the command is simply to replay back to the user the current list of expressions as the user asserted them. Note that the returned list of expressions may be empty or may have just one expression; it is still parenthesized.

The command is optional. If it is not supported, the solver should respond `unsupported`.

[8]This limitation is for simplicity, but it does mean that the `get-assertions` command cannot be used to obtain and replay the entire contents of the assertion set stack.

[9]Including the original whitespace and comments is permitted, but not required.

The command is only valid if the `:interactive-mode` option is true; that option may only be set before the `set-logic` command. If this is not the case, the response to the command should be an error message. The `get-assertions` command itself may not precede a `set-logic` command.

Here is an example (`success` responses are omitted):

```
> (set-option :interactive-mode true)
> (set-logic QF_UF)
> (declare-fun p () Bool)
> (declare-fun q () Bool)
> (push 1)
> (assert (or p q))
> (push 1)
> (assert (not q))
> (get-assertions)
((or p q)
(not q)
)                    ; solvers will vary in the whitespace used
> (pop 1)
> (get-assertions)
((or p q))           ; just one assertion
> (pop 1)
> (get-assertions)
()                   ; no assertions left
```

3.9.11 Options

The option mechanism allows the user to change the behavior of the solver in some ways. Some options are required to be supported by a conforming solver and some options are pre-defined but optional; if a solver does not recognize an option it is required to respond `unsupported`, rather than issuing an error or faulting.

Setting and retrieving option values is independent of the assertion stack. The setting of an option is not changed by a `pop` command.

The `set-option` and `get-option` commands

The `set-option` command sets the value of a specified option; the `get-option` command retrieves the current setting of a given option. The values of options are in general S-expressions, but for all the pre-defined options the values are either a <*string*>, a <*numeral*>, or one of the

<*symbol*>s `true` or `false`. The two commands have this form:

`( set-option` <*keyword*> <*attr-value*> `)`
`( get-option` <*keyword*> `)`

The `set-option` command will respond with one of these responses:

`success` - the value of the option was successfully changed
`unsupported` - the option is optional and is not supported by the solver
`( error` <*string*> `)`

- if the command is malformed (e.g. incorrect number or kind of arguments)
- if the option value is inappropriate for the keyword
- if the option is being set at the wrong place in a command script (e.g. after `set-logic` when it should be before `set-logic`)

The `get-option` command may also respond with `unsupported` or an error, but normally will respond with an S-expression giving the value of the option.

For example:

```
> (get-option :print-success)
true
> (set-option :print-success false)
> (get-option :print-success)
false
> (set-option :print-success true)
success
```

Options defined by standard SMT-LIB

There are a number of options that conforming solvers are required to support and others that are optional but pre-defined.

Option name	required?	type of value	default value
:print-success	required	boolean	true
:regular-output-channel	required	string	"stdout"
:diagnostic-output-channel	required	string	"stderr"
:expand-definitions	optional	boolean	false
:interactive-mode	optional	boolean	false
:produce-proofs	optional	boolean	false
:produce-unsat-cores	optional	boolean	false
:produce-models	optional	boolean	false
:produce-assignments	optional	boolean	false
:random-seed	optional	numeral	0
:verbosity	optional	numeral	0

`:print-success` – The `:print-success` option controls whether `success` is printed after successful commands. The default is to print `success` as a response. In interactive mode, this is the more convenient setting, as the user obtains direct feedback that the command completed successfully. However, in batch mode and even as a less verbose option in interactive mode, it may be desirable to set this option false. Note that any response other than `success` is always output. The value of this option may be changed at any time, with immediate effect.

`:regular-output-channel` and `:diagnostic-output-channel` – These two options control where output from the solver to the user is sent. The `:regular-output-channel` is the destination for all defined messages from the solver: success responses, error messages, and other defined outputs from the various commands. These responses are always formed as S-expressions. The `:diagnostic-output-channel` only receives output controlled by the `:verbosity` option. The format and content of that output is specific to the solver.

The output is sent to

- the standard output of the process, if the value of the option is the string "stdout"
- the standard error output of the process, if the value of the option is the string "stderr"
- to a file if the value of the option is some other string, in which case the string must be the path to a file (relative to the current working directory, if the path is relative). A conforming solver must accept a file path in POSIX form, with forward-slash characters separating components of the path.

In this last case, a `set-option` command setting the value of one of the output channels will return an error if file designated by the path cannot be created (if it does not exist) or cannot be written to. If the file already exists, new output will be appended to the file; if the file does not exist, it will be created.

This option may be changed at any point in a script, with immediate effect.

`:expand-definitions` – This option affects the presentation of expressions from the solver. When the option is set to `true`, all symbols defined with `define-sort` or `define-fun` are substituted with their definitions; these symbols (which are just abbreviations) should not appear in the output. It may be set at any point in a script.[10] The only commands that respond with expressions are `get-assertions`, `get-value`, and `get-proof`. The response to `get-assertions` is unchanged by this option; it always responds with expressions just as the user asserted them. The expressions in a proof produced by `(get-proof)` are affected by this option. The `get-value` command responds with expression-value pairs; values never contain abbreviation symbols in any case, but the user-specified expressions whose values are generated may well — the presentation of these expressions in the command response is affected by the `:expand-definitions` option.

Here is an example with `:expand-definitions` true and false:

```
> (set-option :produce-models true)
> (set-logic QF_LIA)
> (declare-fun x () Int)
> (declare-fun y () Int)
> (define-fun diff () Int (- x y))
> (assert (= (+ x y) 9))
> (assert (= (+ (* 2 x) (* 3 y)) 22))
> (check-sat)
sat
> (set-option :expand-definitions false)
> (get-value (diff))
((diff 1))
> (set-option :expand-definitions true)
> (get-value (diff))
(((- x y) 1))
```

`:interactive-mode` – This option must be set true in order for the `get-assertions` command to be issued. It must be set before `set-logic`.

`:produce-assignments`, `:produce-models`, `:produce-proofs`, `:produce-unsat-cores` – These four options enable the `get-assignment`, `get-value`, `get-proof`, `get-unsat-core` commands, respectively. For each option-command pair, the following hold:

- it is optional for a solver to support the capability provided by the command

[10] The standard does not actually specify when it may be set.

- if the capability is supported, then to use the command during a script, the option must be set to `true` prior to the `set-logic` command
- if the capability is not supported, then attempting to set the value of the option (to either `true` or `false`) should result in an `unsupported` response
- if the capability is not supported or the option has not been set to `true`, then the response to the command should be an error message

It is an error to attempt to set any of these options after the `set-logic` command.

`:verbosity` – The `:verbosity` option is a numeral that controls the volume of diagnostic output. When set to 0, no diagnostic output should be produced. Generally, increasingly larger values should produce increasing amounts of detail, but the amount, content, and format of the output is entirely solver-specific. The value of the option may be changed at any point in a command script. All output controlled by this option must be sent to the diagnostic-output-channel.

`:random-seed` If a solver has some element of random selection in its processing and it supports the `:random-seed` option, then it should use the value of the option to set the seed used for its internal pseudo-random choices. If the value of `:random-seed` is 0, then the solver should choose a more or less arbitrary seed according to its internal logic and the seed may be different for each execution of the solver. However, if the value of the option is a positive integer, then the solver should always make the same internal choices for the same value of the seed. Thus, repeatedly setting the seed to the same positive value should always produce the same output for the same input with the same internal search choices being made.

If `:random-seed` is not supported, the behavior of the solver is as if the value were 0.

3.9.12 Solver information

The `get-info` command provides a way to retrieve items of information about the solver being used. Info items are different than options in that options change the behavior of the solver, while info items merely provide information about the solver or about the solver state. The information may, however, be dynamic; that is, it may change depending on the current state of the solver.

There are several predefined items of solver information. In addition a solver may define other, non-standard items and a user may use arbitrary keywords to add other information items. User-contributed information items function simply as meta-data; they do not affect the solver at all — they are merely stored and retrieved.

Solver information items are identified by a <*keyword*>. A solver must respond to any request for an information item that it does not support with the `unsupported` response. The response to a `get-info` command is one of

- a parenthesized S-expression containing one or more attributes
- `unsupported` – if the solver does not support that information item
- an error response
  - if the command is malformed (e.g. has the wrong number or kind of arguments)

Note that the response to a `get-info` command is always parenthesized and contains the <*keyword*> that was given in the command; also the response may contain more than one attribute. This is a different format than produced by the `get-option` command.

SMT-LIB-defined or solver-defined information items may have a restriction on the kind of S-expression that the information item value may have. For example, the `:version` information item requires a <*string*> as its value. In general, user-defined information items may have arbitrary <*attr-value*> items as their value.

Required info keywords defined by SMT-LIB

These are the pre-defined information keywords that a conforming solver must support.

`:name` replies with the name of the solver, as a pair: `( :name` <*string*>`)`

`:authors` replies with the authors of the solver, as a pair: `( :authors` <*string*>`)` . Note that the value is a single <*string*> containing all the authors names, not multiple <*string*>s each containing one name.

`:version` replies with the version of the solver, as a pair: `( :version` <*string*>`)`

For example

```
> ( get-info :name )
( :name "TheBestSolverEver" )
> ( get-info :version )
( :version "1.0" )
> ( get-info :authors )
( :authors "David R. Cok" )
```

Optional info keywords defined by SMT-LIB

These are the pre-defined information keywords that a conforming solver must either support or respond to with `unsupported`.

`:error-behavior` replies with the error behavior of the solver. The defined values are these:

> `immediate-exit` - if the solver has this behavior, then if a command responds with an error message for any reason, the solver will output the error message and then exit, ignoring the rest of the script. This behavior is perhaps best for batch execution, but it then reports only one error at a time. A script command and type checker, as described in section 6.1.1, might be useful for finding syntax and usage errors before submitting a script to a batch process.
> `continued-execution` - if the solver has this behavior, then upon encountering an error, the error message will be output, but the command will cause no change to the state of the solver, and the solver will proceed to execute the next command in the script. Interactive execution will prefer this second behavior, since then a mistyped command will not cause the immediate exit of the solver; the user will have the chance to correct the command, not losing all previous work. However, in many cases, the user will edit a file containing a script and submit the file to the solver, rather than typing into the solver directly (just as one does not generally type directly into a compiler).

`:reason-unknown` replies with more detail about why the solver responded to a `(check-sat)` command with the reply `unknown`. The defined values of this info item are

> `memout` - meaning the solver ran out of memory and therefore could not complete its analysis
> `incomplete` - meaning the satisfying assignment found cannot be assured to be complete (cf. section 3.1.2)

This option is defined only after a `(check-sat)` command that has returned `unknown` and before any subsequent define, declare, assert, push or pop commands. Other values (e.g. `timeout`) may be standardized in the future.

`:all-statistics` replies with the values of a number of solver-specified statistics about the current state of the solver. The content is not defined by SMT-LIB (yet). This information item in particular may respond with a number of different <*keyword*>-<*attr-value*> pairs, giving different statistics. The option need be available only after a `(check-sat)` command without intervening declaration, definition, push, pop or assert commands.

Note: The `:status` information item, defined in some editions of the SMT-LIB standard, has been deprecated as an SMT-LIB-defined argument to `get-option`.

It is currently unspecified what the value of a user-defined information item is before it is set with `set-info`.

3.9.13 The set-info command

The `set-info` command may only be used with user-defined information keywords; SMT-LIB-defined information items may not be reset by the user.[11]

The form of the command is

$$(\ \texttt{set-info}\ <\textit{keyword}><\textit{S-expression}>\ ).$$

The `set-info` command replies with one of

- `success`
- `unsupported`
- `( error <string> )`

The only occasion for an error response is if the command is malformed or the option is not writable. For example

```
> ( set-info :version "1.1" )
( error "The value of :version may not be set by set-info" )
> ( set-info :zzz 1.1 )
success
> ( set-info )
( error "The set-info command requires an argument" )
> ( get-info :zzz)
(:zzz 1.1)
```

[11] The rules about the `set-info` command are under discussion. It is currently unspecified what the value of an info-item is before it is set.

Chapter 4

Logics and Theories

For the most part, an SMT-LIB user need only know about logics in order to choose the appropriate one for the problem to be solved and to know the symbols defined in that logic. Each logic (and accompanying theories) have a definition and description given in a corresponding file. Inspecting that definition can be instructive.

A theory defines a vocabulary of sorts and functions, and it associates a sort with relevant literals. For example, the `Ints` theory defines the `Int` and `Bool` sorts and declares that any <*numeral*> has sort `Int`. A logic is defined to consist of one or more theories, together with some restrictions on the kinds of expressions that may be used within that logic. For example, the `QF_LIA` logic includes both the `Core` and `Ints` theories. Those theories define many of the usual operations on `Bool` and `Int` values. However the `QF_LIA` logic does not allow quantified expressions and allows only linear arithmetic (e.g. multiplication must be by a literal — one cannot multiply two non-literals together). Logics have such restrictions because there are known decision procedures that can solve satisfiability problems in those logics.

In the following subsections we will simply summarize the salient points about the various theories and logics in SMT-LIB. The interested reader can find a wealth of literature on decision procedures relevant to the different logics. The reader may also wish to consult the text files that define the logics and theories; these are available from the SMT-LIB website[6].

Not all of a logic or theory definition is machine-interpretable. Consequently. logics are actually built-in to solvers rather than the solvers acting only on the contents of the definitions in the logic and theory files. Nevertheless the textual description serves as a standard definition of a logic or theory's symbols that all SMT-LIB solvers will uniformly implement.

4.1 Theories

4.1.1 Definition of a Theory

Theories are defined using specialized S-expressions:

(theory <*symbol*> <*attribute*>+)

where the <*symbol*> is the name of the theory and the attributes (either <*keyword*>s or <*keyword*>-<*attr-value*> pairs) give properties of the theory.

Many of the attributes are descriptive. The following are currently machine-interpretable:

- `:smt-lib-version` – whose value must currently be the SMT-LIB string `"2.0"`
- `:sorts` – whose value must be a list of sort definitions provided by the theory. For example, `Core` has `:sorts ((Bool 0))` and `Reals_Ints` has `:sorts ((Int 0) (Reals 0))`.
- `:funs` – whose value is a list of function declarations, each one of which is a list containing the function name, argument sorts, result sort, and optional attributes. Here are some examples:

```
(false Bool)
(or Bool Bool Bool :left-assoc)
(<= Int Int Bool :chainable)
```

Commonly used descriptive attributes are these (the attribute values are all <*string*>s):

- `:funs-description` – informally specified functions (in addition to those in `:funs`)
- `:sorts-description` – informally specified sorts (in addition to those in `:sorts`)
- `:definition` – an informal specification of the allowed expressions in the theory (there must be exactly one `:definition` attribute)
- `:values` – the expressions that constitute the ground values of the theory's sorts
- `:written_by` – the author of the theory
- `:date` – the initial date the theory was defined
- `:notes` – other explanatory text
- `:last-modified` – the date of the most recent modification to the theory

The SMT-LIB theories are defined in files that are available as part of the SMT-LIB distribution. Each theory is contained in a file whose name is the same as the theory, along with a `.smt2` file extension. Every theory implicitly contains the `Core` theory, which defines propositional logic.

4.1.2 `Core` theory

The `Core` theory contains the basic elements of Boolean logic. It defines

- the `Bool` sort
- the constants `true` and `false` of sort `Bool`
- the `not` operation
- the familiar left-associative functions `and` and `or` and `xor` (for conjunction and disjunction and inequality) on `Bool` values
- the right-associative function `=>`, which is implication
- the (chainable) equality function among the elements of any set of values of the same sort
- the (pairwise) inequality function (called `distinct`) among the elements of any set of values of the same sort
- the if-then-else functions (called `ite`), which take a first `Bool` argument and two additional arguments of the same but arbitrary sort

4.1.3 `Ints` theory

The `Ints` theory contains the basic elements of integer arithmetic. The following are defined:

- the `Int` sort
- all <*numeral*>s as `Int` constants
- the usual `+ - * mod div abs` functions. The `-` symbol is used both for negation (with one argument) and for subtraction (with two or more arguments, left associative). The `div` operation denotes integer division. The `mod` and `div` functions satisfy
  ```
  (= x (+ (* y (div x y)) (mod x y)))
  ```
  (in infix: `x = y * (x div y) + (x mod y)`), for integers `x` and `y`, with `y` non-zero. In addition, `(mod x y)` is non-negative and less than the absolute value of `y`.
- the usual `< > <= >=` comparison functions, returning a `Bool`.
- for each non-zero numeral `n`, there is a function `(_ divisible n)` that takes a single `Int` argument and is true precisely when `n` divides the argument.

4.1.4 `Reals` theory

There is both a theory for `Reals` and a combined `Reals_Ints` theory. The `Reals` theory contains the following:

- a sort `Real`
- both <*numeral*>s and <*decimal*>s are typed as `Real`
- the expected `+ - * /` functions
- the expected `< > <= >=` comparison functions, returning `Bool`

4.1.5 `Reals_Ints` theory

The combined theory of Reals and Ints is not quite the same as the union of the definitions from the two individual theories. In particular

- both the sorts `Int` and `Real` are defined
- <*numeral*>s are of sort `Int` and <*decimal*>s are `Real`
- the functions `+ - * div mod abs < > <= >=` and the family of `divisible` functions are defined for `Int` arguments
- the functions `+ - * / < > <= >=` are defined for `Real` arguments
- the function `to_real` that maps `Int` arguments to the corresponding `Real`
- the function `to_int` that maps a `Real` argument to the largest integer less than or equal to the argument (the floor function)
- the function `is_int` that maps a `Real` to a `Bool` and is true just for those arguments that are equal to `(to_real` *n*`)` or to `(to_real (-` *n*`))` for some numeral *n*

4.1.6 `ArraysEx` theory

The theory of Arrays defines a parameterized sort and functions to read and write elements of arrays.

- the new sort symbol `Array` that takes two sort parameters: the first is the sort of the index, the second is the sort of the value of the array elements.
- the function `select` that extracts values from the array. It is a parameterized function. Its arguments are (1) the array, of sort, say, `(Array Index Value)`, with sort parameters `Index` and `Value`, and (2) the index value, of sort `Index`. The result is a value of sort `Value`.
- the function `store` that produces a new array with a modified value at a given index. It takes three arguments: the array, of sort `(Array Index Value)`; the index, of sort `Index`; and the new value for that index, of sort `Value`. The result is a new array with that one value changed.
- two values of the same Array sort are equal if the array elements are equal for every value of the index sort.

4.1.7 `Fixed_Size_BitVectors` theory

The final pre-defined theory describes the behavior of bit-vectors. A different bit-vector sort is defined for each length of vector. Operations are defined to manipulate, combine, and extract

portions of bit-vectors. Some of the operations interpret the bit-vector as a natural number: the bit-vector is considered to be an unsigned binary representation of a non-negative integer, with the least significant bit on the right.

- a sort `(_ BitVec` *n*`)` is defined for each non-zero numeral *n*
- the binary and hexadecimal literals are defined to have a bit-vector sort of the corresponding length
- the function `concat` is defined that combines two bit-vectors into a longer one
- a family of functions `(_ extract` *i j*`)` is defined that extracts a contiguous sub-vector from index *i* to *j* (inclusive) of a bit-vector
- unary functions `bvnot` and `bvneg`
- binary functions `bvand bvor bvadd bvmul bvudiv bvurem bvshl bvlshr`
- the binary comparison function `bvult`

4.2 Logics

4.2.1 Definition of a logic

An SMT-LIB command involving assertions is always interpreted and executed in the context of a *logic*. The relevant logic is set by the `(set-logic` <*symbol*>`)` command. A logic defines, by way of the *theories* it includes, all the sorts and function and constant symbols that make up the initial set of definitions. The logic also may restrict the expressions that are allowed. (Later subsections contain a quick overview of the SMT-LIB defined logics.) For example, the `QF_UF` (Quantifier-Free with Uninterpreted Functions) logic defines the following:

- the sort `Bool`
- the constants `true` and `false` of sort `Bool`
- the unary function `not` from `Bool` to `Bool`
- the functions `and`, `or`, `xor`, and `=>` that take two or more `Bool` arguments, producing `Bool`
- equality (`=`), pair-wise distinct (`distinct`) and if-then-else (`ite`) functions
- restricts expressions to be quantifier-free

A logic is defined using, as usual, an S-expression of a particular form:

`(` `logic` <*symbol*> <*attribute*>+ `)`

Here the <*symbol*> is the name of the logic and <*attribute*>+ indicates one or more attributes,

each of which is a <*keyword*> or <*keyword*>-<*attr-value*> pair. Some of the attributes are simply descriptive; they provide information such as the author of the logic. Other attributes are an informal part of the definition of the logic; they are not machine-interpretable and their prescriptions must be built-in to a conforming tool. Two attributes are machine-interpretable and are important to solvers:

- `:smt-lib-version` – whose value must at present be the SMT-LIB string `"2.0"`
- `:theories` – whose value must be a parenthesized list of theory names. These are the theories that make up the logic. There must be exactly one `:theories` attribute in the definition. For example, the value of `:theories` for `QF_UF` is simply `(Core)` and for `AUFLIA` is `(Ints ArraysEx)`. Every theory (and thus every logic) implicitly includes the `Core` theory.

Other commonly used, descriptive attributes are these (the attribute values are all <*string*>s):

- `:written_by` – the author
- `:date` – the date the logic was first written
- `:language` – a description of the restrictions on expressions that constitute the language of the logic
- `:notes` – additional explanatory text
- `:values` – the ground expressions of the logic
- `:extensions` – syntactic sugar that extends the language of the logic

The SMT-LIB logics are defined in files that are available as part of the SMT-LIB distribution. Each logic is contained in a file whose name is the same as the logic, along with a `.smt2` file extension. Although the currently defined logics are informally described here, the definition files are the official reference on the details of the logic.

4.2.2 Boolean logics

QF_UF is the logic of Quantifier-Free Uninterpreted Functions. It incorporates just the `Core` theory, providing the `Bool` sort and the various standard operations on Boolean values. The user may define additional sorts and uninterpreted functions. The restriction imposed by the logic is that no quantified expressions are allowed.

4.2.3 Logics with arithmetic

QF_LIA adds linear arithmetic to the `QF_UF` logic. So the `Int` sort is defined, along with the operations defined in the `Ints` theory. This logic does not permit quantified expressions. It is also

limited to *linear arithmetic*. Thus addition, negation, subtraction and comparisons are permitted; multiplication is permitted only if one multiplicand is a numeral (a constant). No additional sorts or functions are defined.

QF_NIA incorporates the `Ints` theory, but without the linearity restriction. It does not allow quantified expressions, but any of the functions defined in `Ints` are permitted. No additional sorts or functions are defined.

QF_LRA adds linear arithmetic over the reals to the `QF_UF` logic. So the `Real` sort is defined, along with the operations defined in the `Reals` theory. This logic does not permit quantified expressions and is limited to *linear arithmetic*. Thus addition, negation, subtraction, and comparisons are permitted; multiplication is permitted only if one multiplicand is a `Real` literal (either a <*numeral*> or a <*decimal*>). No additional sorts or functions are defined.

QF_AUFLIA extends the `QF_LIA` linear arithmetic logic, with the addition of arrays and arbitrary uninterpreted sorts and functions.

AUFLIA extends QF_AUFLIA by allowing quantifiers.

AUFLIRA allows quantifiers and includes `Int` and `Real` sorts, but is limited to linear arithmetic.

AUFNIRA allows quantifiers and general `Int` and `Real` arithmetic, arrays, and uninterpreted functions.

LRA simply extends `QF_LRA` by allowing quantifiers.

4.2.4 Logics for difference arithmetic

Difference arithmetics only allow either comparisons between numeric values or comparisons of a difference between two numeric values to a positive or negative numeric literal.

QF_IDL is a difference logic over `Ints` that does not allow quantifiers

QF_RDL is a difference logic over `Reals` that does not allow quantifiers

QF_UFIDL is difference logic over `Ints` that does not allow quantifiers but does permit arbitrary additional sorts and uninterpreted function symbols

4.2.5 Logics with Bit-Vectors and Arrays

QF_BV allows quantifier-free expressions, including the family of bit-vector sorts and all of the functions defined in the `Fixed_Size_BitVectors` theory (but no other new sorts or functions).

QF_AX includes Booleans, Arrays (from the `ArraysEx` theory), and arbitrary additional sorts and constants (but not functions), with quantifier-free expressions

QF_ABV includes quantifier-free expressions over Booleans, Arrays and BitVectors, with all array index and value sorts being bit-vector sorts.

QF_AUFBV extends the `QF_BV` logic with arrays, arbitrary sorts and function symbols

Chapter 5

SMT solvers

This chapter presents a brief mention of a number of SMT solvers. I will be happy to include any SMT solver that is currently maintained, under active development, and can be obtained for assessment and use. In this version of the tutorial, we simply list candidates; assessments of conformity to SMT-LIB will follow in subsequent versions. The SMT-COMP competitions were executed in Linux, so those solvers will run on Linux but may not have Windows equivalents. Some of the solvers listed may require licenses from the authors or their institutions.

- Solvers competing in the SMT-COMP 2010. That competition used SMT-LIB and these solvers were required to handle at least `assert` and `check-sat` commands in batch mode. (`http://www.smtexec.org/exec/competitors2010.php`)
  - AProVE NIA 0.2.1 : `http://aprove.informatik.rwth-aachen.de/`
  - CVC3 2.3 : `http://cs.nyu.edu/acsys/cvc3/`
  - CVC4 1.0a0 : `http://cs.nyu.edu/acsys/cvc4/`
  - MathSAT 5 : `http://mathsat4.disi.unitn.it/`
  - MiniSMT : `http://cl-informatik.uibk.ac.at/software/minismt/`
  - OpenSMT-1.0-alpha : `http://www.verify.inf.unisi.ch/opensmt`
  - simplifyingSTP : `http://sites.google.com/site/stpfastprover/`
  - SONOLAR r252 : `http://www.informatik.uni-bremen.de/~florian/sonolar/`
  - test_pmathsat 0.0.5 : relative of MathSAT
  - veriT 201007 : `http://www.verit-solver.org/`
- Other older solvers that can be executed on Windows OS and for which there are jSMTLIB adapters. These solvers are not themselves SMT-LIB conforming, but they can be used as SMT-LIB solvers through the adapters.

  - Simplify 1.5.4 (for Windows) : https://mobius.ucd.ie/repos/src/mobius.esc/escjava/trunk/ESCTools/Escjava/release/master/bin/
  - CVC3 2.2 : http://cs.nyu.edu/acsys/cvc3/
  - Yices 1.0.28 : http://yices.csl.sri.com/
  - Z3 2.11 : http://research.microsoft.com/en-us/um/redmond/projects/z3/
- Other non-SMTLIB solvers from SMTCOMP-2009 that are not subsumed by later versions (http://www.smtexec.org/exec/competitors2009.php)
  - barcelogic-QF_NIA : http://www.lsi.upc.edu/~oliveras/bclt-main.html
  - beaver-smtcomp-2009 : http://uclid.eecs.berkeley.edu/newwiki/beaver/start
  - Boolector 1.2 : http://fmv.jku.at/boolector/
  - clsat 1.0
  - SatEEn-3.5 : http://vlsi.colorado.edu/~hhkim/sateen/
  - sword-1.0 : http://www.informatik.uni-bremen.de/agra/eng/sword.php
  - Yices 2 proto : http://yices.csl.sri.com/
- Other non-SMTLIB solvers from SMTCOMP-2008 that are not subsumed by later versions (http://www.smtexec.org/exec/competitors2008.php)
  - Alt-Ergo : http://alt-ergo.lri.fr/
  - Spear : http://www.domagoj-babic.com/index.php/ResearchProjects/Spear

Details on these solvers will be added in a future edition of the tutorial.

Chapter 6

Tools

As SMT-LIB becomes more widely used, increasing numbers of supporting tools and software packages will become available. Some current tools were generated in conjunction with this tutorial; others are independently available.

6.1 Tools associated with this tutorial

6.1.1 The SMT-LIB validator

An SMT-LIB validator is available as a tool in conjunction with this tutorial. The tool is an executable jar, running on Java 1.6. As a validator, the tool reads, parses, and type-checks SMT-LIB scripts; it is intended to fully and precisely conform to SMT-LIB, except that it does not do any solving.

It is executed with the command line

```
java -jar jSMTLIB.jar <file>
```

where <*file*> is the path to the script file to check. Information about the command-line options can be obtained using

```
java -jar jSMTLIB.jar -help
```

The tool is available from `http://www.grammatech.com/resources/smt/jSMTLIB.tar`. The current version is an alpha release, since some details of the SMT-LIB language are still under discussion and some aspects of theories and logics are not yet implemented.

Client-server implementation. The validator tool can receive its input directly from a file or standard input; it can also function as a both client and server, communicating through a network

port.

6.1.2 The SMT-LIB adapters

The `jSMTLIB.jar` tool discussed in subsection 6.1.1 also contains adapters that convert standard SMT-LIB input into input specific to some non-SMT-LIB solvers. The Windows versions of Simplify, CVC3 2.2, Yices 1.0.28, and Z3 2.11 are currently supported. To use this feature, you must have an executable copy of the solver you wish to use. It is then executed by this command-line:

`java -jar jSMTLIB.jar -solver` <*solvername*> `-exec` <*path*> <*file*>

Here <*solvername*> is one of `cvc`, `simplify`, `yices`, or `z3` and <*path*> is the file system path to the corresponding executable.

The adapters are generally of alpha-release quality; they generally work, but are not necessarily robust and are missing significant features.

6.1.3 The SMT-LIB Java API

The Java validator application also serves as an API that can be programmatically driven from a Java application (or any language with a Java interface). This API is tested along with the validator application.

The `jSMTLIB.jar` file is the library for the API. The public functions have Javadoc documentation. A general user guide to the library is being written.

6.1.4 The SMT Eclipse plug-in

The validator application is a command-line tool. There is also an associated Eclipse plug-in. The internal functionality is the same as that of the command-line tool, since the plug-in is simply a GUI that drives the command-line tool through the programmatic API. The plug-in will appeal to those who prefer a GUI text editor for creating and reviewing SMT-LIB scripts. The plug-in also makes it easy to try different solvers on the same problem.

The plug-in has this functionality:

- A custom text editor associated with the `.smt2` file name suffix. The editor's syntax coloring is customized to SMT-LIB scripts.
- The SMT-LIB validator highlights errors in script files using the standard Eclipse problem mechanism and using a new problem marker that identifies SMT-LIB problems.

- The ability to view logic and theory files
- The ability to execute scripts on a chosen solver from the GUI
- A typical Eclipse preference page that permits the user to set the equivalent of command-line options using the GUI
- An Eclipse Help page

The plug-in can be installed using the standard Eclipse installation mechanism by using this web-site: `http://www.grammatech.com/resources/smt/jSMTLIB-UpdateSite`.

6.1.5 SMT validation test suite

The development of the jSMTLIB application, API, and adapters was helped by the complementary creation of a validation test suite. That suite can be used to compare the behavior of various solvers that (more or less) conform to SMT-LIBv2.

The suite consists of a number of SMT-LIB command scripts and their corresponding expected responses. A bash script drives the execution of the scripts, collection of the responses, and comparisons against the "golden" output. Since there are no standard error messages for invalid scripts, the only comparison that can be made in those cases is whether or not the solver produced an error response.

SMT solvers are under active development. Those that participated in SMT-COMP 2010, when version 2.0 of SMT-LIB had only recently been issued, typically implemented only enough of the version 2.0 syntax to be able to participate successfully in the competition. Some of these solvers are actively working to become fully conforming; other development groups will eventually do so but have other priorities for current development. Thus it was not thought useful to publish detailed evaluations of individual solvers in this document. A future edition of the tutorial will include such comments in chapter 5.

The validation suite is available from the author and will be made available by web download once some currently discussed details of SMT-LIB are resolved.

6.2 Tools from other providers

The SMT-LIB web site (`www.smt-lib.org`) lists other tools that process SMT-LIB text. The site currently lists these SMT-LIBv2 parsers:

- A C99 parser by Alberto Griggio:
  `https://es.fbk.eu/people/griggio/misc/smtlib2parser.html`
- An OCaml parser by Kyle Krchak and Aaron Stump (downloadable from the Utilities page of `www.smt-lib.org`)
- A Haskell parser by Tom Hawkins: `http://hackage.haskell.org/package/smt-lib`

Bibliography

[1] Clark Barrett, Morgan Deters, Albert Oliveras, and Aaron Stump. Design and results of the 3^{rd} annual satisfiability modulo theories competition (SMT-COMP 2007). *International Journal on Artificial Intelligence Tools (IJAIT)*, 17(4):569–606, August 2008.

[2] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, 2010.

[3] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010.

[4] Morgan Deters. `http://www.smtcomp.org/2010`.

[5] Silvio Ranise and Cesare Tinelli. The SMT-LIB Format: An Initial Proposal. In *Proceedings of the 1st Workshop on Pragmatics of Decision Procedures in Automated Reasoning (Miami Beach, USA)*. 2003.

[6] Cesare Tinelli. `http://www.smt-lib.org`.

[7] http://en.wikipedia.org/wiki/Regular_expression.

An index will be added in a future edition of the tutorial.