

The SMT-LIB Standard: Version 1.0

Working Draft

Silvio Ranise* Cesare Tinelli†

Revision date: **15 September 2004**

*LORIA INRIA-Lorraine, Nancy, France, ranise@loria.fr.

†Department of Computer Science, The University of Iowa, Iowa City, IA, USA,
tinelli@cs.uiowa.edu.

Contents

0	About this Document	4
0.1	Copyright Notice	4
1	Introduction	4
2	Basic Assumptions and Structure	5
2.1	The Logic	5
2.2	The Background Theories	6
2.3	The Benchmarks Language	6
3	The SMT-LIB Logic and Language	7
3.1	The Formula Sublanguage	7
3.2	The additional constructs	11
3.3	Well-sorted formulas	13
4	The SMT-LIB Theory Language	15
5	The SMT-LIB Benchmark Language	18
6	Semantics	22
6.1	Translation Semantics	22
6.2	Model-theoretic Semantics	24
6.3	Equivalence of the two Semantics	24
7	Concrete Syntax	24
7.1	Comments	24
7.2	Terms and formulas	26
7.3	Theories	26
7.4	Benchmarks and benchmark sets	26
8	Acknowledgments	28
A	Abstract Syntax	31
B	Concrete Syntax	32
C	Examples	33

List of Figures

1	Abstract syntax for unsorted terms and formulas	8
2	Well-sortedness rules for terms	14
3	Well-sortedness rules for formulas	14
4	Abstract syntax for theories	16
5	Abstract syntax for benchmarks	18
6	Abstract syntax for benchmark sets	19
7	Translation of function declarations and terms	23
8	Translation of formulas	23
9	Concrete syntax for unsorted terms and formulas	25
10	Concrete syntax for theories	26
11	Concrete syntax for benchmarks	27
12	Concrete syntax for benchmark sets	27

0 About this Document

This paper defines and discusses Version 1.0 of a language standard for the Satisfiability Modulo Theories Library, or SMT-LIB for short.

This is an incomplete working document that will be updated on a regular basis. Each new revision of the document will be identified by its date.

It is expected that each new revision will contain only *conservative* additions and changes with respect to the previous revision, that is, it will not modify any already given definitions for the SMT-LIB standard. A new revision will only add missing definitions, or provide a more complete and improved presentation of the standard.

Place holders for missing material are denoted by text in square brackets.

To facilitate the reading of the document, explanations of the rationale of the various design decisions taken in defining the SMT-LIB standard are written in san-serif font. They can be skipped on a first reading.

0.1 Copyright Notice

Permission is granted to anyone to make or distribute verbatim copies of this document, in any medium, provided that the copyright notice and permission notice are preserved, and that the distributor grants the recipient permission for further redistribution as permitted by this notice. Modified versions may not be made.

1 Introduction

The main goal of the SMT-LIB initiative [2], coordinated by these authors and supported by a large number of research groups world-wide, is to produce a on-line library of benchmarks for *satisfiability modulo theories*. By benchmark we mean a logical formula to be checked for satisfiability with respect to (combinations of) background theories of interest. Examples of background theories typically used in computer science are real and integer arithmetic and the theories of various data structures such as lists, arrays, bit vectors and so on.

A lot of work has been done in the last few years by several research groups on building systems for satisfiability modulo theories [?]. The main motivation of the SMT-LIB initiative is the expectation that having a library of benchmarks will greatly facilitate the evaluation and the comparison of these systems, and advance the state of the art in the field, in the same way as, for instance, the TPTP library [3] has done for theorem proving, or the SATLIB library [1] has done for propositional satisfiability.

[more]

SMT-LIB consists of two main sections: one containing the specification of several background theories, and another containing benchmark sets, grouped under a number of indexes such as their corresponding background theory, the class of formulas they belong to, the type of problem they originate from and so on.

For the library to be viable and useful, SMT-LIB adopts a common standard for expressing the benchmarks, and for defining the background theories in a rigorous way—so that there is no doubt on which theories are intended.

2 Basic Assumptions and Structure

In the SMT-LIB standard input problems, i.e. logical formulas, are assumed to be checked for satisfiability, not validity.¹ In particular, given a theory T and a formula φ , the problem of interest is whether φ is *satisfiable in T* , or is *satisfiable modulo T* , that is, whether there is a model of φ that satisfies (the existential closure of) φ .

Informally speaking, SMT-LIB calls a *theory solver*, or also a *satisfiability procedure*, any procedure for satisfiability modulo some given theory. In general, with satisfiability procedures one can distinguish among

1. a procedure’s *underlying logic* (e.g., first-order, modal, temporal, second-order, and so on),
2. a procedure’s *background theory*, the theory against which satisfiability is checked, and
3. a procedure’s *input language*, the class of formulas the procedure accepts as input.

For better clarity and modularity, these three aspects are kept separate in SMT-LIB. SMT-LIB’s commitments to each of them are described in the following.

2.1 The Logic

Version 1 of the SMT-LIB standard adopts as its logic a basic many-sorted version of first-order logic with equality. This logic allows the definition of sorts and of sorted symbols but not allows no subsorts, sort constructors, terms declarations and so on. These more sophisticated features may be added in the future² but only as needed.

In an attempt to combine the simplicity and familiarity of classical, unsorted logic with the convenience of a sorted language, the SMT-LIB standard defines two

¹Note that the difference matters only for those classes of problems that are not closed under logical negation.

²For instance, it is expected that Version 2 of the standard will include subsorts.

semantics for its logic: the first one is a translation semantics into classical (unsorted) first order logic with equality ($FOL^=$), the second one is a direct algebraic semantics based on many-sorted models.

The first semantics is meant to ease the transition to a sorted framework for existing tools, expertise and results, most of which have so far have been developed in the context of classical $FOL^=$. For the purposes of SMT-LIB the two semantics are equivalent, as an SMT-LIB formula admits a (many-sorted) model in the algebraic semantics if and only if its translation into $FOL^=$ admits a (unsorted) model in $FOL^=$.

The logic and its two semantics are specified in the next sections.

2.2 The Background Theories

One of the goals of the SMT-LIB initiative is to clearly define a catalog of background theories, starting with a small number of popular ones, and adding new ones as solvers for them are developed. Theories are specified in SMT-LIB independently of any benchmarks or solvers. Each set of benchmarks then contains a reference to its own background theory.

The SMT-LIB standard will eventually distinguish between *basic* (or *component*) theories and *combined* (or *structured*) theories. Basic theories will include theories such as the theory of real numbers, the theory of arrays, the theory of lists and so on. A combined theory will be one that is defined as some kind of combination of basic theories.

The current version of SMT-LIB supports only basic theories. This means in practice that a theory composed of previously defined ones can always be defined in this version of the standard, but only from scratch, as if it was a basic theories, with no external references to its previously defined components. Support for the modular definition of structures theories will be added in future versions of the standard.

For practicality, the standard insists that only the signature of a theory be specified formally.³ The theory itself can be defined either formally, by means of a set of axioms, or informally, in natural language, as convenient.

2.3 The Benchmarks Language

The SMT-LIB standard adopts a single and general first-order (sorted) language in which to write all SMT-LIB benchmarks.

It is often the case, however, that many benchmarks are expressed in a some fragment of the language of first-order logic. The particular fragment in question

³By "formal" here we will always mean written in a machine-readable and processable format, as opposed to written in free text no matter how rigorously.

does matter because one can often write a solver specialized on that fragment that is a lot more efficient than a solver meant for a larger fragment.⁴

An extreme case of this situation occurs when satisfiability modulo a given theory T is decidable for a certain fragment (quantifier-free, say) but undecidable for a larger one (full first-order, say), as for instance happens with the theory of arrays specified in Section C. But a similar situation occurs even when the decidability of the satisfiability problem is preserved across various fragments. For instance, if T is the theory of real numbers, the satisfiability in T of full-first order formulas is decidable. However, one can build increasingly faster solvers by restricting the language respectively to quantifier-free formula, linear equations and inequations, difference equations, inequations between variables, and so on [?].

As a consequence, the SMT-LIB standard makes it possible to specify for any benchmark set the specific sublanguage its benchmarks belong to.

3 The SMT-LIB Logic and Language

Because of the SMT-LIB translation semantics, for a semantical viewpoint the ultimate underlying logic for SMT-LIB is $FOL^=$, the classical (unsorted) first-order logic with equality. The *external* logic and its language are however many-sorted.

As a typed language, the SMT-LIB language for formulas and theories is intentionally limited in expressive power. In essence, the language allows one only to declare sorts (types) only by means of sort symbols, to specify the interface, or *rank* of function and predicate symbols in terms of the declared sorts, and to specify the sort of quantified variables.

In type theory terms, the language has no subtypes, no type constructors, no type quantifiers, no provisions for parametric or subsort polymorphism, and so on. Even explicit (ad-hoc) overloading of function or predicate symbols—by which a symbol could be explicitly given more than one rank—is not allowed. The idea is to provide, at least in this version, just enough expressive power to represent typical benchmarks without getting bogged down in the complexity (and higher-orderness) of type theory.

3.1 The Formula Sublanguage

The syntax of formulas in the SMT-LIB language extends the standard abstract syntax of $FOL^=$ with a construct for declaring the sort of quantified variables, plus the following non-standard constructs:

⁴By efficiency here we do not necessary refer to worst-case time complexity, but to efficiency “in practice”.

(Annotations)	$\alpha ::= a = v$
(Terms)	$u ::= x \mid n \mid f t^* \mid \mathbf{ite} \varphi t_1 t_2$
(Annot. Terms)	$t ::= u \alpha^*$
(Connectives)	$\kappa ::= \mathbf{not} \mid \mathbf{implies} \mid \mathbf{and} \mid \mathbf{or} \mid \mathbf{xor} \mid \mathbf{iff}$
(Atoms)	$A ::= \mathbf{true} \mid \mathbf{false} \mid \xi \mid p t^* \mid = t^* \mid \mathbf{distinct} t^*$
(Formulas)	$\psi ::= A \mid \kappa \varphi^+ \mid \exists (x:s)^+ \varphi_0 \mid \forall (x:s)^+ \varphi_0$ $\quad \mid \mathbf{let} x = t \mathbf{in} \varphi_0 \mid \mathbf{flet} \xi = \varphi_0 \mathbf{in} \varphi_1$ $\quad \mid \mathbf{if} \varphi_0 \mathbf{then} \varphi_1 \mathbf{else} \varphi_2$
(Annot. Formulas)	$\varphi ::= \psi \alpha^*$

$x \in \mathcal{X}$,	the set of term variables	$\xi \in \Xi$,	the set of formula variables
$n \in \mathcal{N}$,	the set of numerals	$f \in \mathcal{F}$,	the set of function symbols
$p \in \mathcal{P}$,	the set of predicate symbols	$s \in \mathcal{S}$,	the set of sort symbols
$a \in \mathcal{A}$,	the set of attributes	$v \in \mathcal{V}$,	the set of attribute values

Figure 1: Abstract syntax for unsorted terms and formulas

- an **if-then-else** construct for terms,
- an **if-then-else** logical connective,
- a **let** construct for terms,
- a **let** construct for formulas, and
- a **distinct** construct for declaring a number of values as pairwise distinct.
- an annotation mechanism for terms and formulas.

Except for the first extension, needed to support sorts, the other extensions are provided for greater convenience. We discuss each of them in the following.

An abstract grammar and syntax for a superset of the SMT-LIB language of logical formulas is defined in Figure 1 by means of production rules. This is a *superset* of the language because it contains ill sorted terms and formulas. The proper well sorted subset is defined later by means of a separate sort system.

The rules assume as given the following sets of symbols:

- an infinite set \mathcal{X} of *term variables*, standard first-order variables that can be used in place of a term;

- an infinite set Ξ of *formula variables*, second-order variables that can be used in place of a formula;
- an infinite set \mathcal{N} of *numerals*, for the natural numbers;
- an infinite set \mathcal{F} of *function symbols*;
- an infinite set \mathcal{P} of *predicate symbols*;
- an infinite set \mathcal{S} of *sort symbols*;
- an infinite set \mathcal{A} of *attribute symbols*;
- a set \mathcal{V} of attribute *values*.

It is required that \mathcal{A} be disjoint from all the other sets. The remaining sets need not be disjoint because the syntax of terms and formula allows one to infer to which set a particular symbol/value belongs.

In Figure 1 and later in the paper, boldface words denote terminal symbols and the $(_)*$ operator denotes as usual zero or more repetitions of the operand, while the $(_)+$ operator denotes one or more repetitions of the operand. Function/predicate applications are denoted simply by juxtaposition, as this is enough at the abstract level. Parentheses are metasympols, used just for grouping—they are not symbols of the abstract language. In the production rules, the letter a denotes attribute symbols, the letter v attribute values, the letter x term variables, the letter ξ formula variables, the letter n numerals, the letter f functions symbols, the letter f predicate symbols, the letter u terms, the letter t annotated terms, the letter s sort symbols, the letter ψ formulas, and the letter φ annotated formulas.

The given grammar does not distinguish between constant and function symbols (they are all defined as members of the set \mathcal{F}), and between propositional variables and predicate symbols (they are all defined as members of the set \mathcal{P}). These distinctions are really a matter of arity, which is taken care of later by the well-sortedness rules. A similar observation applies to the logical connectives, the members of κ class, and the number of arguments they are allowed take.

From now on, we will simply say term to mean a possibly annotated term, and formula to mean a possibly annotated formula.

As usual, one can speak of the free variables of a formula. Formally, The set $\mathcal{V}ar(t)$ of free term variables in a term t and the set $\mathcal{V}ar(\varphi)$ of free term variables

in a formula φ are respectively defined below by structural induction.

$$\begin{aligned}
\mathcal{V}ar(x) &= \{x\} \\
\mathcal{V}ar(n) &= \emptyset \\
\mathcal{V}ar(f\ t_1 \cdots t_k) &= \mathcal{V}ar(t_1) \cup \cdots \cup \mathcal{V}ar(t_k) \\
\mathcal{V}ar(\mathbf{ite}\ \varphi_0\ t_1\ t_2) &= \mathcal{V}ar(\varphi_0) \cup \mathcal{V}ar(t_1) \cup \mathcal{V}ar(t_2) \\
\mathcal{V}ar(u\ \alpha_1 \cdots \alpha_k) &= \mathcal{V}ar(u)
\end{aligned}$$

$$\begin{aligned}
\mathcal{V}ar(\mathbf{true}) &= \emptyset \\
\mathcal{V}ar(\mathbf{false}) &= \emptyset \\
\mathcal{V}ar(\xi) &= \emptyset \\
\mathcal{V}ar(\pi\ t_1 \cdots t_k) &= \mathcal{V}ar(t_1) \cup \cdots \cup \mathcal{V}ar(t_k) \\
&\quad \text{if } \pi \in \mathcal{P} \cup \{=, \mathbf{distinct}\} \\
\mathcal{V}ar(\kappa\ \varphi_1 \cdots \varphi_k) &= \mathcal{V}ar(\varphi_1) \cup \cdots \cup \mathcal{V}ar(\varphi_k) \\
\mathcal{V}ar(\forall\ x_1:s_1 \dots x_k:s_k\ \varphi_0) &= \mathcal{V}ar(\varphi_0) \setminus \{x_1 \dots x_k\} \\
\mathcal{V}ar(\exists\ x_1:s_1 \dots x_k:s_k\ \varphi_0) &= \mathcal{V}ar(\varphi_0) \setminus \{x_1 \dots x_k\} \\
\mathcal{V}ar(\mathbf{let}\ x = t\ \mathbf{in}\ \varphi_0) &= \mathcal{V}ar(t) \cup (\mathcal{V}ar(\varphi_0) \setminus \{x\}) \\
\mathcal{V}ar(\mathbf{flet}\ \xi = \varphi_0\ \mathbf{in}\ \varphi_1) &= \mathcal{V}ar(\varphi_0) \cup \mathcal{V}ar(\varphi_1) \\
\mathcal{V}ar(\mathbf{if}\ \varphi_0\ \mathbf{then}\ \varphi_1\ \mathbf{else}\ \varphi_2) &= \mathcal{V}ar(\varphi_0) \cup \mathcal{V}ar(\varphi_1) \cup \mathcal{V}ar(\varphi_2) \\
\mathcal{V}ar(\psi\ \alpha_1 \cdots \alpha_k) &= \mathcal{V}ar(\psi)
\end{aligned}$$

[note on the fact that with $\mathbf{let}\ x = t\ \mathbf{in}\ \varphi_0$, x is bound in φ_0 but not in t .]

The set $\mathcal{F}ar(\varphi)$ of free formula variables in a formula φ is instead defined as follows

$$\begin{aligned}
\mathcal{F}ar(\mathbf{true}) &= \emptyset \\
\mathcal{F}ar(\mathbf{false}) &= \emptyset \\
\mathcal{F}ar(\xi) &= \{\xi\} \\
\mathcal{F}ar(\pi\ t_1 \cdots t_k) &= \emptyset \text{ if } \pi \in \mathcal{P} \cup \{=, \mathbf{distinct}\} \\
\mathcal{F}ar(\kappa\ \varphi_1 \cdots \varphi_k) &= \mathcal{F}ar(\varphi_1) \cup \cdots \cup \mathcal{F}ar(\varphi_k) \\
\mathcal{F}ar(\forall\ x_1:s_1 \dots x_k:s_k\ \varphi_0) &= \mathcal{F}ar(\varphi_0) \\
\mathcal{F}ar(\exists\ x_1:s_1 \dots x_k:s_k\ \varphi_0) &= \mathcal{F}ar(\varphi_0) \\
\mathcal{F}ar(\mathbf{let}\ x = t\ \mathbf{in}\ \varphi_0) &= \mathcal{F}ar(\varphi_0) \\
\mathcal{F}ar(\mathbf{flet}\ \xi = \varphi_0\ \mathbf{in}\ \varphi_1) &= \mathcal{F}ar(\varphi_0) \cup (\mathcal{F}ar(\varphi_1) \setminus \{\xi\}) \\
\mathcal{F}ar(\mathbf{if}\ \varphi_0\ \mathbf{then}\ \varphi_1\ \mathbf{else}\ \varphi_2) &= \mathcal{F}ar(\varphi_0) \cup \mathcal{F}ar(\varphi_1) \cup \mathcal{F}ar(\varphi_2) \\
\mathcal{F}ar(\psi\ \alpha_1 \cdots \alpha_k) &= \mathcal{F}ar(\psi)
\end{aligned}$$

[note on the fact that with $\mathbf{flet}\ \xi = \varphi_0\ \mathbf{in}\ \varphi_1$, ξ is bound in φ_1 but not in φ_0 .]

A formula φ is *closed* iff $\mathcal{V}ar(\varphi) = \mathcal{F}ar(\varphi) = \emptyset$.

3.2 The additional constructs

[connecting text]

The *if-then-else* construct for terms

This construct is very common in benchmarks coming from hardware verification. Technically, it is a function symbol of arity 3, taking as first argument a formula and as second and third arguments a term. Semantically, an expression like

$$\mathbf{ite}(\varphi, t_1, t_2)$$

evaluates to the value of t_1 in every interpretation that makes φ true, and to the value of t_2 in every interpretation that makes φ false.

Although it can be defined in terms of more basic constructs, this construct provides important structural information that a solver may be able to use advantageously. More importantly, for large benchmarks, eliminating **ite** constructs can result in an unacceptable blowup in the size of the formula. For these reasons, the SMT-LIB standard supports them natively.

The *if-then-else* logical connective

This construct is also common in verification benchmarks. It is used to build a formula of the form

$$\mathbf{if} \varphi_0 \mathbf{then} \varphi_1 \mathbf{else} \varphi_2$$

which is semantically equivalent to the formula

$$(\varphi_0 \mathbf{implies} \varphi_1) \mathbf{and} (\mathbf{not} \varphi_0 \mathbf{implies} \varphi_2)$$

The SMT-LIB standard supports this *if-then-else* connective natively for similar reasons it supports the **ite** term constructor.

The *let* construct for terms

This construct builds a formula of the form

$$\mathbf{let} x = t \mathbf{in} \varphi_0$$

which is semantically equivalent to the formula obtained from φ_0 by simultaneously replacing every free occurrence of x in φ_0 by t , and renaming as necessary the bound variables of φ_0 so that the variables of t remain free. The construct is convenient for benchmark compactness as it allows one to replace multiple occurrences of the same term by a variable. It is also useful for a solver because it saves the solver the effort of recognizing the various occurrences of the same term as such.

[comment on semantics, scope and binding]

The *let* construct for formulas

This construct builds a formula of the form

$$\mathbf{flet} \ \xi = \varphi_0 \ \mathbf{in} \ \varphi_1$$

which is semantically equivalent to the formula obtained from φ_1 by simultaneously replacing every free occurrence of ξ in φ_1 by φ_0 , and renaming as necessary the bound variables of φ_1 so that the free variables of φ_0 remain free. The rationale for supporting this construct in SMT-LIB is similar to that for supporting the **let** construct for terms.

[comment on semantics, scope and binding]

The *distinct* construct

This construct as well is supported for conciseness. It is a variadic construct for building formulas of the form

$$\mathbf{distinct}(t_1, \dots, t_n)$$

with $n \geq 2$. Semantically, it is equivalent to the conjunction of all disequations of the form $\mathbf{not}(t_i = t_j)$ for $1 \leq i < j \leq n$.

The annotation mechanism for terms and formulas

Each term or formula can be annotated with a list of attribute/value pairs where each pair has the form

$$a \ v$$

where a is a symbol representing an attribute's name and v is the attribute's value. The syntax of attribute values is user-dependent and as such it is left unspecified by the SMT-LIB standard. Annotations are meant to provide extra-logical information which, while not changing the semantics of a term or formula, may be useful to an SMT solver.

It is expected that typical annotations will provide operational information for the solver. For instance, the annotation α in a formula of the form

$$\forall x_1:s_1 \dots x_k:s_k \ \varphi_0 \ \alpha$$

might specify an instantiation pattern for the quantifier $\forall x_1:s_1 \dots x_k:s_k$, as done in the Simplify prover [?]. Or, for formulas that represent verification conditions for a program, the annotation might contain information relating the formula to the original code the formula was derived from.⁵

⁵A similar idea is used in the ESC/Java system with the use of a special “label” predicate [?].

3.3 Well-sorted formulas

The SMT-LIB language of formulas is the largest set of well-sorted formulas contained in the language generated by the rule for annotated formulas in the grammar of Figure 1.

Well-sorted formulas are defined by means of a set of sorting rules, similar in format and spirit to the kind of typing rules found in the programming languages literature. The rules are based on the following definition of (many-sorted) signature.

Definition 1 (SMT-LIB Signature) An *SMT-LIB signature* Σ is a tuple consisting of:

- a non-empty set $\Sigma^S \subseteq \mathcal{S}$ of sort symbols, a set $\Sigma^F \subseteq \mathcal{F}$ of function symbols, a set $\Sigma^P \subseteq \mathcal{P}$ of predicate symbols,
- a total mapping from the term variables \mathcal{X} to Σ^S ,
- a total mapping from Σ^F to $(\Sigma^S)^+$, the non-empty sequences of elements of Σ^S , and
- a mapping from Σ^P to $(\Sigma^S)^*$, the sequences of elements of Σ^S ,

The sequence of sorts associated by Σ to a function/predicate symbol is called the *rank* of the symbol

As usual, the rank of a function symbol specifies the expected sort of the symbol's argument and result. Similarly for predicate symbols. Note that the sets Σ^S , Σ^F and Σ^P of an STM-LIB signature are not required to be disjoint. So it is possible for a symbol to be both a function and a predicate symbol, say. This causes no ambiguity in the language because positional information is enough to determine during parsing if a given occurrence of a symbol is a function, predicate or sort symbol. However, it is not possible for a function (resp. predicate) symbol to have more than one rank.⁶ In other words, *ad hoc* overloading of function or of predicate symbols is not allowed. This restriction is imposed mainly in order to simplify parsing and well-sortedness checking of SMT-LIB formulas.

Figure 2 provides a set of rules defining well-sorted terms, while Figure 3 provides a rule set defining well-sorted formulas. The sort rules presuppose the existence of an SMT-LIB signature Σ . Strictly speaking then, the SMT-LIB language is a family of languages parametrized by Σ . As explained later, for each benchmark φ and theory T , the specific signature is jointly defined by the specification of T and that of the benchmark set containing φ .

⁶This is a consequence of the last two mappings in the definition of SMT-LIB signature.

$$\begin{array}{c}
\frac{}{\Sigma \vdash_t x : s} \quad \text{if } x : s \in \Sigma \qquad \frac{}{\Sigma \vdash_t n : s} \quad \text{if } n : s \in \Sigma \\
\\
\frac{\Sigma \vdash_t t_1 : s_1 \quad \cdots \quad \Sigma \vdash_t t_k : s_k}{\Sigma \vdash_t f t_1 \cdots t_k : s_{k+1}} \quad \text{if } f : s_1 \cdots s_{k+1} \in \Sigma \\
\\
\frac{\Sigma \vdash_f \varphi \quad \Sigma \vdash_t t_1 : s \quad \Sigma \vdash_t t_2 : s}{\Sigma \vdash_t \mathbf{ite} \varphi t_1 t_2 : s}
\end{array}$$

Figure 2: Well-sortedness rules for terms

$$\begin{array}{c}
\frac{}{\Sigma \vdash_f \mathbf{true}} \qquad \frac{}{\Sigma \vdash_f \mathbf{false}} \qquad \frac{}{\Sigma \vdash_f \xi} \\
\\
\frac{\Sigma \vdash_t t_1 : s_1 \quad \cdots \quad \Sigma \vdash_t t_k : s_k}{\Sigma \vdash_f p t_1 \cdots t_k} \quad \text{if } p : s_1 \cdots s_k \in \Sigma \\
\\
\frac{\Sigma \vdash_t t_1 : s \quad \cdots \quad \Sigma \vdash_t t_{k+2} : s}{\Sigma \vdash_f = t_1 \cdots t_{k+2}} \qquad \frac{\Sigma \vdash_t t_1 : s \quad \cdots \quad \Sigma \vdash_t t_{k+2} : s}{\Sigma \vdash_f \mathbf{distinct} t_1 \cdots t_{k+2}} \\
\\
\frac{\Sigma \vdash_f \varphi}{\Sigma \vdash_f \mathbf{not} \varphi} \qquad \frac{\Sigma \vdash_f \varphi_1 \quad \Sigma \vdash_f \varphi_2}{\Sigma \vdash_f \mathbf{impl} \varphi_1 \varphi_2} \qquad \frac{\Sigma \vdash_f \varphi_0 \quad \Sigma \vdash_f \varphi_1 \quad \Sigma \vdash_f \varphi_2}{\Sigma \vdash_f \mathbf{if} \varphi_0 \mathbf{then} \varphi_1 \mathbf{else} \varphi_2} \\
\\
\frac{\Sigma \vdash_f \varphi_1 \quad \cdots \quad \Sigma \vdash_f \varphi_{k+2}}{\Sigma \vdash_f c \varphi_1 \cdots \varphi_{k+2}} \quad \text{if } c \in \{\mathbf{and}, \mathbf{or}, \mathbf{xor}, \mathbf{iff}\} \\
\\
\frac{\Sigma, x_1 : s_1, \dots, x : s_{k+1} \vdash_f \varphi}{\Sigma \vdash_f \exists x_1 : s_1 \dots x : s_{k+1} \varphi} \qquad \frac{\Sigma, x_1 : s_1, \dots, x : s_{k+1} \vdash_f \varphi}{\Sigma \vdash_f \forall x_1 : s_1 \dots x : s_{k+1} \varphi} \\
\\
\frac{\Sigma \vdash_t t : s \quad \Sigma, x : s \vdash_f \varphi}{\Sigma \vdash_f \mathbf{let} x = t \mathbf{in} \varphi} \qquad \frac{\Sigma \vdash_f \varphi_0 \quad \Sigma \vdash_f \varphi_1}{\Sigma \vdash_f \mathbf{flet} \xi = \varphi_0 \mathbf{in} \varphi_1}
\end{array}$$

Figure 3: Well-sortedness rules for formulas

The format and the meaning of the sort rules in the two figures is pretty standard and should be largely self-explanatory. The integer index k in the rules is assumed ≥ 0 ; the notation $x:s \in \Sigma$ means that Σ maps the variable x to the sort s . The notation $f:s_1 \cdots s_{n+1} \in \Sigma$ means that $f \in \Sigma^F$ and Σ maps f to the sort sequence $s_1 \cdots s_{n+1}$ (and similarly for numerals and predicate symbols). The expression $\Sigma, x:s$ denotes the signature that maps x to the sort s and otherwise coincides with Σ .

A term t generated by the grammar in Figure 1 is *well-sorted (with respect to Σ)* if the expression $\Sigma \vdash_t t:s$ is derivable by the sort rules in Figure 2 for some sort $s \in \Sigma^S$. In that case, we say that t is of sort s .

A formula φ generated by the grammar in Figure 1 is *well-sorted (with respect to Σ)* if the expression $\Sigma \vdash_f \varphi$ is derivable by the sort rules in Figure 3.

Definition 2 (SMT-LIB formulas) The SMT-LIB language for formulas is the set of all closed well-formed formulas.

Note that the SMT-LIB language for formulas contains only closed formulas. This is mostly a technical restriction, motivated by considerations of convenience. In fact, with a closed formula φ of a signature Σ the particular mapping of term variables to sorts defined by Σ is irrelevant. The reason is that the formula itself contains its own sort declaration for its term variables, either explicitly, for the variables bound by a quantifier, or implicitly, for the variables bound by a `let`. Using only closed formulas then simplifies the task of specifying their signature, as it becomes unnecessary to specify how the signature maps the elements of \mathcal{X} to the signature's sorts.

There is no loss of generality in allowing only closed formulas because, as far as satisfiability of formulas is concerned, every formula φ with free variables $\text{Var}(\varphi) = \{x_1, \dots, x_n\}$, where each x_i is expected to have sort s_i , can be rewritten as

$$\exists x_1:s_1 \dots x_n:s_n \varphi.$$

An alternative way to avoid free variables in benchmarks is defined in Section 5

A similar situation arises with formula variables. In fact, all free occurrences of a formula variable can be replaced by a fresh predicate symbol p declared in Σ as having empty arity.

4 The SMT-LIB Theory Language

This version of the standard considers only the specification of basic background theories. Facilities for specifying structured theories will be introduced in a later version.

(Fun. sym. declaration) $D_f ::= f s^+ \alpha$

(Pred. sym. declaration) $D_p ::= p s^* \alpha$

(Attribute-value pair) $P_T ::=$

sorts = s^+	
funcs = $(D_f)^+$	preds = $(D_p)^+$
definition = w	axioms = φ^+
extensions = w	notes = w α

(Theory declaration) $D_T ::=$

theory T begin
$(P_T)^+$
end

$T \in \mathcal{T}$, the set of theory symbols $w \in \mathcal{W}$, the set of character strings

Figure 4: Abstract syntax for theories

A *theory declaration* defines both an order-sorted signature for a theory and the theory itself. The abstract syntax for a theory declaration in the SMT-LIB format is provided in Figure 4. The syntax follows an attribute-value-based format.

In addition to the already defined symbols and syntactical categories, the production rules in Figure 4 assume as given the following sets of symbols:

- an infinite set \mathcal{W} of *character strings*, meant to contain free text;
- an infinite set \mathcal{T} of *theory symbols*, used to give a name to each theory.

In the rules, the letter w denotes strings and the letter T theory symbols.

The symbols **funcs**, **preds**, **axioms**, **extensions**, **notes**, **sorts**, and **definition** are reserved attribute symbols from \mathcal{A} . Their sets of values are as specified in the rules. In addition to the predefined attributes, a theory declaration D_T can contain an unspecified number of user defined attributes, and their values—formalized in the grammar simply as annotations α .

[note on user defined attribute: expandable and customizable format]

Note that attribute/value pairs in a theory declaration can be written in any order. However, they are subject to the restriction below.

Definition 3 (Theory Declarations) *The only legal theory declarations in the SMT-LIB format are those that*

1. contain the attributes **sort** and **definition**⁷;

⁷Which makes those attributes non-optional.

2. *do not contain repeated attribute symbols;*
3. *[restrictions on function/predicate symbols declarations: no overloading]*
4. *[more restrictions on axioms: must be of the declared signature]*

The first restriction is explained in the following. The second restriction is just to simplify later the definition of a declaration semantics, and the automated processing of theory declarations.

Some attributes, such as **definition** for instance, are *informal attributes* in the sense that their value (w) is free text. Ideally, a formal specification of the given free-text attributes would be preferable to free text in order to avoid ambiguities and misinterpretation. The choice of using free text for these attributes is motivated by practicality reasons. In fact, (i) these attributes are meant to be read by human readers, not programs, and (ii) the amount of effort needed to first devise a formal language for these attributes and then specify their values for each theory in the library does not seem justified by the current goals of SMT-LIB.

The signature of a theory is defined by the attributes **sorts**, **funcs** and **preds** in the obvious way. A declaration D_f for a function symbol f specifies the symbol's rank, and may contain additional, user-defined annotations. A typical annotation might specify that f is associative, say. While this property is expected to be specified also in the definition of the theory (or to be a consequence thereof), an explicit declaration at this point could be useful for certain solvers that treat associative symbols in a special way. Similarly, a declaration D_p for a predicate symbol p specifies the symbol's the rank and may be augmented with user-defined annotations.

This version of the format does not specify any predefined annotations for function and predicate symbols. Future versions might do so, depending on the recommendations and the feedback of the SMT-LIB user community.

The **funcs** and **preds** attributes are optional because a theory might lack function or predicate symbols.⁸ The **sorts** attribute, however, is not optional because sorted frameworks require the existence of at least one sort. This is no real limitation of course because, for instance, any unsorted theory can be always seen as at least one-sorted.

The non-optional **definition** attribute is meant to contain a natural language definition of the theory. While this definition is expected to be as rigorous as possible, it does not have to be a formal one. Some theories (like the theory of real numbers) are well known, and so just a reference to their common name might be enough. For theories that have a small set of axioms (or axiom schemas), it might be convenient to list the actual axioms. For some other theories, a mix a formal notation and informal explanation might be more appropriate.

Formal, first-order axioms that define part of or a whole theory can be additionally provided in the optional **axioms** attribute as a list of SMT-LIB formulas. In

⁸Or both, although a theory with no function and no predicate symbols is perhaps not very useful.

(Formula status)	$\sigma ::= \mathbf{sat} \mid \mathbf{unsat} \mid \mathbf{unknown}$
(Attribute-value pair)	$P_b ::= \mathbf{assumption} = \varphi \mid \mathbf{formula} = \varphi$ $\quad \mid \mathbf{status} = \sigma \mid \alpha$
(Benchmark declaration)	$D_b ::= \mathbf{benchmark} \ b \ \mathbf{begin}$ $\quad \quad \quad (P_b)^+$ $\quad \quad \quad \mathbf{end}$

$b \in \mathcal{B}$, the set of benchmark symbols

Figure 5: Abstract syntax for benchmarks

addition to being more rigorous, the formal definition of (a part of) a theory provided in the **axioms** attribute, might be useful for solvers that might not have (that part of) the theory built in, but accept theory axioms input. In essence, this is currently the case for instance for the solvers Simplify [?], CVC Lite [?], Harvey [?], and Argo-lib [?].

The optional attribute **extensions** is meant to document any notational conventions used in the listed benchmarks. This is useful because often the syntax of a theory is extended for convenience with syntactic sugar. One example of such conventions comes for instance from (the theory of) linear Presburger arithmetic, where a numeral n abbreviates the n -fold application of the successor function to 0, and the expression $n * t$ stands for the term $\underbrace{t + \dots + t}_{n \text{ times}}$.

The optional attribute **notes** is meant to contain documentation information such as authors, date, version, etc. of a specification.

5 The SMT-LIB Benchmark Language

In SMT-LIB, a benchmark is a closed SMT-LIB formula with attached additional information, specified in a *benchmark* declaration. Benchmarks are grouped into benchmark sets. A *benchmark set* declaration contains, in addition to the benchmarks themselves, a reference to their background theory, a description of the language fragment to which the benchmarks belong, and an optional specification of additional function and predicate symbols.

The abstract syntax for a benchmark and for a benchmark set declaration in the SMT-LIB format is provided in Figure 5 and Figure 6, respectively. This syntax too is attribute-value-based.

In addition to the already defined symbols and syntactical categories, the pro-

(Attribute-value pair) $P_S ::= \mathbf{theory} = T \mid \mathbf{benchmarks} = (D_b)^+$
 $\mid \mathbf{extrafuns} = (D_f)^+ \mid \mathbf{extrapreds} = (D_p)^+$
 $\mid \mathbf{language} = w \mid \mathbf{notes} = w \mid \alpha$

(Bench. set declaration) $D_S ::= \mathbf{benchset } S \mathbf{ begin}$
 $(P_B)^+$
 \mathbf{end}

$S \in \mathcal{BS}$, the set of benchmark set symbols

Figure 6: Abstract syntax for benchmark sets

duction rules in the two figures assume as given the following sets of symbols:

- an infinite set \mathcal{B} of *benchmark symbols*, used to give a name to each benchmark, and
- an infinite set \mathcal{BS} of *benchmark set symbols*, used to give a name to each benchmark set.

In the rules, the letter b denotes benchmark symbols and the letter B benchmark set symbols.

The symbols **assumption**, **formula**, **status**, **theory**, **benchmarks**, **extrafuns**, **extrapreds**, **language**, and **notes** are reserved attribute symbols from \mathcal{A} . Their sets of values are as specified in the rules. As with theories, both benchmark and benchmark set declarations can also contain user-defined attributes and their values—formalized again as annotations α .

Definition 4 (Benchmark Declarations) *The only legal benchmark declarations in the SMT-LIB format are those that*

1. *contain the attributes **formula** and **status**, and*
2. *do not contain repeated attribute symbols.*

In a benchmark declaration of the form

benchmark b **begin**
assumption = φ_0
formula = φ_1
status = σ
end

the formulas φ_0 and φ_1 together constitute the benchmark. The intended test is whether the formula φ_1 is satisfiable in the background theory *under the assumption* φ_0 ; in other words, whether the formula (**and** φ_0 φ) is satisfiable in the background theory. The assumption φ_0 is assumed to be just **true** whenever the optional attribute **assumption** is absent. [explanation of why it is useful to have explicit assumptions]

The attribute **status** of a benchmark declaration records whether the benchmark is known to be (un)satisfiable in the background theory. Knowing about the satisfiability of a benchmark is useful for debugging new solvers.

Definition 5 (Benchmark Set Declarations) *The only legal benchmark set declarations in the SMT-LIB format are those that satisfy the following restrictions:*

1. *the declaration contains the attributes **theory** and **benchmarks**;*
2. *the value of the attribute **theory** coincides with the name of a theory T for some theory declaration D_T in SMT-LIB;*
3. *the sort symbols occurring in (the value of) the attributes **extrafuns** or **extrapreds** are among the symbols listed in the attribute **sorts** of D_T ;*
4. *there are no repeated occurrences of function (resp., predicate) symbols in the attribute **extrafuns** (resp., **extrapreds**);*
5. *no symbol occurring in **extrafuns** (resp., **extrapreds**) occurs in the attribute **funs** (resp., **preds**) of D_T ;*
6. *all function (resp., predicate) symbols occurring in the attribute **benchmarks** are declared either in **extrafuns** (resp., **extra_preds**) or in the attribute **funs** (resp., **preds**) of D_T .*
7. *all formulas in the benchmarks contained in the attribute **benchmarks** are over the sort, function and predicate symbols declared in D_T , **extrafuns** and **extra_preds**.*

The optional attribute **language** of a benchmark set declaration attribute describes in free text the specific subset of SMT-LIB formulas to which the listed benchmarks belong. As explained in Section 2.3, this information is useful for tailoring solvers to the specific sublanguage of formulas used in the benchmark set. The attribute is text valued because it has mostly documentation purposes for the benefit of benchmark users. A natural language description of the sublanguage seems therefore adequate for this purpose.

The **extrafuns** attribute complements the **funs** attribute of the corresponding theory specification by declaring additional function symbols with their rank. The **extrapreds** attribute has a similar purpose, but for predicate symbols. In contrast with the symbols possibly defined in the **extensions** attribute of a theory specification, which are interpreted in terms of the symbols in the theory, the symbols in **extrafuns** and **extrapreds** are “uninterpreted” in associated theory.⁹ Uninterpreted function or predicate symbols are found often in applications of satisfiability modulo theories, typically as a consequence of Skolemization or abstraction transformations applied to more complex formulas. Hence theory solvers typically accept formulas containing uninterpreted symbols in addition to the symbols of their background theory. The **extrafuns** and **extrapreds** attributes serve to declare any uninterpreted symbols occurring in benchmarks listed in the **benchmarks** attribute. The value of **extrafuns** and **extrapreds** attributes is specified formally because, in effect, it dynamically expand the signature of the associated background theory, hence it is convenient for it to be directly readable by satisfiability procedures for that theory.

The **extrafuns** attribute is also useful for specifying benchmarks consisting of formulas with free term variables (such as quantifier-free formulas). As discussed in Section 3.3, the legal formulas of SMT-LIB do not contain free variables. One way to circumvent this restriction is to close formulas existentially. Another one is to replace each free term variable by a fresh constant symbol of the appropriate sort. In the second case, these extra constant symbols can be declared in the **extrafuns** attribute. A similar situation can occur in principle with free formula variables, which can stand for unspecified predicates. Each free occurrence of a formula variable can be replaced by a fresh predicate symbol of empty arity (a.k.a, a propositional variable). Such symbols can be declared in the **extrapreds** attribute.

Note that all function or predicates symbols occurring in the benchmarks of a benchmark set must be declared either in the corresponding theory specification or in the **extrafuns** and **extrapreds** attributes. One could think of relaxing this restriction by adopting the convention that any undeclared function or predicate symbol occurring in a benchmark is automatically considered as uninterpreted. Contrary to unsorted logics, however, this approach is not feasible in SMT-LIB because it is not possible in general to automatically infer (the sorts in) the rank of an undeclared symbol. [to add: at the many sorted level one could assume an uninterpreted sort as well and default to that sort, but that is not very satisfactory, and in any case it will be enough to achieve automated sort inference once subsorts are added.]

⁹In logic parlance, they are *free* symbols for the theory.

6 Semantics

6.1 Translation Semantics

One semantics for SMT-LIB formulas and theories is provided by a translation of formulas and their signature into formulas of $FOL^=$, classical first order logic with equality.

We define a translation operator $_^\tau$, following a well-known relativization process for translating many-sorted logics into classical logic—see for instance [?, ?]. Formally, the SMT-LIB translation semantics is defined as follows.

Definition 6 Let Σ be an SMT-LIB signature and let T be a theory of signature Σ axiomatized by a set Ax of (SMT-LIB) formulas. For every Σ -formula ψ we say that ψ is satisfiable in T iff the set

$$\{(f : s_1 \cdots s_{k+1})^\tau \mid f : s_1 \cdots s_{k+1} \in \Sigma^F\} \cup \{\varphi^\tau \mid \varphi \in Ax\} \cup \{\psi^\tau\}$$

of $FOL^=$ formulas is satisfiable (in the classical sense).

The signature Σ and the axioms Ax in the definition above are meant to be elicited from the specification of a benchmark set containing the formula ψ , and from the specification of the benchmark set’s background theory. [to add: more details on ow to elicit Σ].

A fine point to note about our translation semantics is that it effectively requires the given background theory to be defined axiomatically—so that one can identify the set Ax needed in Definition 6.

The translation operator $_^\tau$ is defined in Figure 7 for function symbol declarations and terms, and in Figure 8 for formulas. The translation into $FOL^=$ use a conventional syntax for $FOL^=$ sentences, with the exclusive or connective denoted by \oplus . On terms, the operator $_^\tau$ is defined only for terms that do not contain **ite** expressions. **ite**’s are eliminated from terms as shown in Figure 8. It is easy to show that $_^\tau$ is well defined.

In Figure 8, φ' denotes a *rectified* version of φ obtained by renaming apart all quantifiers—and their bound variables—into fresh variables, while $\varphi'\{x \mapsto t\}$ denotes the formula obtained from φ' by simultaneously replacing every unbound occurrence of the variable x in φ' by the term t .¹⁰ Similarly for $\varphi'\{\xi \mapsto \varphi_0\}$. The expression $A[\mathbf{ite} \varphi t_1 t_2]$ denotes an atomic formula A containing an occurrence of the term $(\mathbf{ite} \varphi t_1 t_2)$, while $A[t_i]$ ($i = 1, 2$) denotes the formula obtained from $A[\mathbf{ite} \varphi t_1 t_2]$ be replacing that occurrence of $(\mathbf{ite} \varphi t_1 t_2)$ with t_i .

¹⁰This is to avoid that the (free) variables of t become bound after the substitution.

Function declarations

$$(f : s_1 \cdots s_{k+1})^\tau = \forall x_1, \dots, x_n. (s_1(x_1) \wedge \cdots \wedge s_k(x_n) \Rightarrow s_{k+1}(f(x_1, \dots, x_k)))$$

ite-free Terms

$$\begin{aligned} x^\tau &= x \\ n^\tau &= n \\ (f \ t_1 \cdots t_k)^\tau &= f(t_1^\tau, \dots, t_k^\tau) \\ (u \ \alpha_1 \cdots \alpha_k)^\tau &= u^\tau \end{aligned}$$

Figure 7: Translation of function declarations and terms

Formulas

$$\begin{aligned} \mathbf{false}^\tau &= \perp \\ \mathbf{true}^\tau &= \neg \perp \\ (\pi \ t_1 \cdots t_k)^\tau &= \pi(t_1^\tau, \dots, t_k^\tau) \\ &\quad \text{if } \pi \in \mathcal{P} \cup \{=, \mathbf{distinct}\} \text{ and } t_1, \dots, t_k \text{ are } \mathbf{ite}\text{-free} \\ (A[\mathbf{ite} \ \varphi \ t_1 \ t_2])^\tau &= (\varphi^\tau \Rightarrow (A[t_1])^\tau) \wedge (\neg \varphi^\tau \Rightarrow (A[t_2])^\tau) \\ (\mathbf{not} \ \varphi)^\tau &= \neg \varphi^\tau \\ (\mathbf{implies} \ \varphi_1 \ \varphi_2)^\tau &= \varphi_1^\tau \Rightarrow \varphi_2^\tau \\ (\mathbf{and} \ \varphi_1 \cdots \varphi_k)^\tau &= \varphi_1^\tau \wedge \cdots \wedge \varphi_k^\tau \\ (\mathbf{or} \ \varphi_1 \cdots \varphi_k)^\tau &= \varphi_1^\tau \vee \cdots \vee \varphi_k^\tau \\ (\mathbf{xor} \ \varphi_1 \cdots \varphi_k)^\tau &= \varphi_1^\tau \oplus \cdots \oplus \varphi_k^\tau \\ (\mathbf{iff} \ \varphi_1 \cdots \varphi_k)^\tau &= \varphi_1^\tau \Leftrightarrow \cdots \Leftrightarrow \varphi_k^\tau \\ (\mathbf{if} \ \varphi_0 \ \mathbf{then} \ \varphi_1 \ \mathbf{else} \ \varphi_2)^\tau &= (\varphi_0^\tau \Rightarrow \varphi_1^\tau) \wedge (\neg \varphi_0^\tau \Rightarrow \varphi_2^\tau) \\ (\forall x_1:s_1 \dots x_k:s_k \ \varphi_0)^\tau &= \forall x_1, \dots, x_k. (s_1(x) \wedge \cdots \wedge s_k(x_k) \Rightarrow \varphi_0^\tau) \\ (\exists x_1:s_1 \dots x_k:s_k \ \varphi_0)^\tau &= \exists x_1, \dots, x_k. (s_1(x) \wedge \cdots \wedge s_k(x_k) \wedge \varphi_0^\tau) \\ (\mathbf{let} \ x = t \ \mathbf{in} \ \varphi)^\tau &= (\varphi' \{x \mapsto t\})^\tau \\ (\mathbf{flet} \ \xi = \varphi_0 \ \mathbf{in} \ \varphi)^\tau &= (\varphi' \{\xi \mapsto \varphi_0\})^\tau \\ (\psi \ \alpha_1 \cdots \alpha_k)^\tau &= \psi^\tau \end{aligned}$$

Figure 8: Translation of formulas

6.2 Model-theoretic Semantics

[to do]

6.3 Equivalence of the two Semantics

[to do]

7 Concrete Syntax

This section defines and explains the concrete syntax of the whole STM-LIB language. The adopted syntax is attribute-based and Lisp-like. Its design was driven more by the goal of simplifying parsing, than that of facilitating readability by humans. Preferring ease of parsing over human readability is reasonable in this context because it is expected not only that SMT-LIB benchmarks will be typically read by solvers but also that, by and large, they will be produced in the first place by automated tools like verification condition generators or translators from other formats. An alternative concrete syntax, more readable for human users, and a translation from the current concrete syntax might be defined in a later version of the standard.

In BNF-style production rules that define the concrete syntax, terminal symbols are denoted by sequences of characters in *typewriter font*. Syntactical categories are denoted by text in angular braces and *slanted font*. For simplicity, white space symbols are not modeled in the rules. It is understood though that, as usual, every two successive terminals in an actual expression must be separated by one or more white space characters, unless one of them is a parenthesis.¹¹

As with the abstract syntax, the production rules of the concrete syntax define a superset of the legal expressions. The subset of legal expression is of course the one that satisfies the same constraints defined for the abstract syntax.

7.1 Comments

Source files containing SMT-LIB expressions may contain *comments* in the sense of programming languages. In SMT-LIB, a comment is a sequence of characters that start with the character `;` and is terminated by a new line character. The choice of `;` as the comment character is simply for consistency with the Lisp-like flavor of the concrete syntax.

¹¹The set of concrete terminal symbols includes the open and closed parenthesis symbols.

$\langle \text{identifier} \rangle$::=	<i>a sequence of letters, digits, dots (.), and underscores (-), starting with a letter</i>
$\langle \text{numeral} \rangle$::=	<i>a non-empty sequence of digits</i>
$\langle \text{var} \rangle$::=	? $\langle \text{identifier} \rangle$
$\langle \text{fvar} \rangle$::=	$\$ \langle \text{identifier} \rangle$
$\langle \text{attribute} \rangle$::=	: $\langle \text{identifier} \rangle$
$\langle \text{ar_symb} \rangle$::=	<i>a non-empty sequence of the characters: =, <, >, &, @, #, +, -, *, /, %, , ~</i>
$\langle \text{fun_symb} \rangle$::=	$\langle \text{identifier} \rangle$ $\langle \text{ar_symb} \rangle$
$\langle \text{pred_symb} \rangle$::=	$\langle \text{identifier} \rangle$ $\langle \text{ar_symb} \rangle$ distinct
$\langle \text{sort_symb} \rangle$::=	$\langle \text{identifier} \rangle$
$\langle \text{annotation} \rangle$::=	$\langle \text{attribute} \rangle$ $\langle \text{attribute} \rangle \langle \text{value} \rangle$
$\langle \text{base_term} \rangle$::=	$\langle \text{variable} \rangle$ $\langle \text{numeral} \rangle$ $\langle \text{identifier} \rangle$
$\langle \text{an_term} \rangle$::=	$\langle \text{base_term} \rangle$ ($\langle \text{base_term} \rangle \langle \text{annotation} \rangle^+ $) ($\langle \text{fun_symb} \rangle \langle \text{an_term} \rangle^+ \langle \text{annotation} \rangle^* $) (ite $\langle \text{an_formula} \rangle \langle \text{an_term} \rangle \langle \text{an_term} \rangle \langle \text{annotation} \rangle^* $)
$\langle \text{prop_atom} \rangle$::=	true false $\langle \text{fvar} \rangle$ $\langle \text{identifier} \rangle$
$\langle \text{an_atom} \rangle$::=	$\langle \text{prop_atom} \rangle$ ($\langle \text{prop_atom} \rangle \langle \text{annotation} \rangle^+ $) ($\langle \text{pred_symb} \rangle \langle \text{an_term} \rangle^+ \langle \text{annotation} \rangle^* $)
$\langle \text{connective} \rangle$::=	not implies if_then_else and or xor iff
$\langle \text{quant_symb} \rangle$::=	exists forall
$\langle \text{quant_var} \rangle$::=	($\langle \text{var} \rangle \langle \text{sort_symb} \rangle$)
$\langle \text{an_formula} \rangle$::=	$\langle \text{an_atom} \rangle$ ($\langle \text{connective} \rangle \langle \text{an_formula} \rangle^+ \langle \text{annotation} \rangle^* $) ($\langle \text{quant_symb} \rangle \langle \text{quant_var} \rangle^+ \langle \text{an_formula} \rangle \langle \text{annotation} \rangle^* $) (let ($\langle \text{var} \rangle \langle \text{an_term} \rangle$) $\langle \text{an_formula} \rangle \langle \text{annotation} \rangle^* $) (flet ($\langle \text{fvar} \rangle \langle \text{an_formula} \rangle$) $\langle \text{an_formula} \rangle \langle \text{annotation} \rangle^* $)

Figure 9: Concrete syntax for unsorted terms and formulas

$\langle string \rangle$	$::=$	<i>any sequence of characters other than double quotes ("), and enclosed in double quotes</i>
$\langle fun_symb_decl \rangle$	$::=$	$(\langle fun_symb \rangle \langle sort_symb \rangle^+ \langle annotation \rangle^*)$
$\langle pred_symb_decl \rangle$	$::=$	$(\langle pred_symb \rangle \langle sort_symb \rangle^* \langle annotation \rangle^*)$
$\langle theory_name \rangle$	$::=$	$\langle identifier \rangle$
$\langle theory_attribute \rangle$	$::=$	$:notes \langle string \rangle$ $ $ $:sorts (\langle sort_symb \rangle^+)$ $ $ $:funs (\langle fun_symb_decl \rangle^+)$ $ $ $:preds (\langle pred_symb_decl \rangle^+)$ $ $ $:extensions \langle string \rangle$ $ $ $:definition \langle string \rangle$ $ $ $:axioms \langle string \rangle$ $ $ $\langle annotation \rangle$
$\langle theory \rangle$	$::=$	$(theory \langle theory_name \rangle \langle theory_attribute \rangle^+)$

Figure 10: Concrete syntax for theories

7.2 Terms and formulas

The concrete syntax for SMT-LIB unsorted terms and formulas is given in Figure 9 with BNF production rules based on the abstract syntax rules given in Figure 1.

[more]

7.3 Theories

The concrete syntax for SMT-LIB theory declarations is given in Figure 10 with BNF production rules based on the abstract syntax rules given in Figure 4.

[more]

7.4 Benchmarks and benchmark sets

The concrete syntax for SMT-LIB benchmark and benchmark set declarations is given in Figure 11 and Figure 12 with BNF production rules based on the abstract syntax rules given in Figure 5 and Figure 6, respectively.

[more]

```

⟨status⟩           ::=  sat | unsat | unknown
⟨bench_name⟩      ::=  ⟨identifier⟩
⟨bench_attribute⟩ ::=  :assumption ⟨an_formula⟩
                    |    :formula  ⟨an_formula⟩
                    |    :status   ⟨status⟩
                    |    ⟨annotation⟩
⟨benchmark⟩       ::=  ( benchmark ⟨bench_name⟩ ⟨bench_attribute⟩+ )

```

Figure 11: Concrete syntax for benchmarks

```

⟨benchset_name⟩   ::=  ⟨identifier⟩
⟨benchset_attribute⟩ ::=  :theory  ⟨theory_name⟩
                    |    :benchmarks ⟨benchmark⟩+
                    |    :extrafuns  ⟨fun_symb_decl⟩+
                    |    :extrapreds  ⟨pred_symb_decl⟩+
                    |    :language  ⟨string⟩
                    |    :notes     ⟨string⟩
                    |    ⟨annotation⟩
⟨benchset⟩        ::=  ( benchset ⟨benchset_name⟩ ⟨benchset_attribute⟩+ )

```

Figure 12: Concrete syntax for benchmark sets

8 Acknowledgments

The SMT-LIB initiative was established in response to a call by Alessandro Armando at the FroCoS 2002 business meeting for a library of benchmarks for the SMT community.

The following people (in alphabetical order) have provided suggestions, comments and feedback on the SMT-LIB initiative and format: Peter Andrews, Alessandro Armando, Clark Barrett, Sergey Berenzin, Joseph Kiniry, Sava Krstic, Predrag Janičić, Shuvendu Lahiri, José Meseguer, Greg Nelson, Harald Ruess, Geoff Satcliffe, James Saxe, Roberto Sebastiani, Natarajan Shankar, and Aaron Stump. [did we miss anybody?]

The following people have provided suggestions, comments and feedback on this document: Mike DeCoster.

References

- [1] Holger Hoos and Thomas Stützle. SATLIB–The Satisfiability Library. Web site at: <http://www.satlib.org/>.
- [2] Silvio Ranise and Cesare Tinelli. SMT-LIB–The Satisfiability Modulo Theories Library. Web site at: <http://combination.cs.uiowa.edu/smtlib/>.
- [3] Geoff Sutcliffe and Christian Suttner. The TPTP Problem Library for Automated Theorem Proving. Web site at: <http://www.cs.miami.edu/tptp/>.

Index

$FOL^=$, 6

formula
 closed, 10

rank, 7

satisfiability
 modulo theories, 4
 procedure, 5
SMT-LIB, 4

theory
 basic, 6
 combined, 6
 component, 6
 solver, 5
 structured, 6

variables
 formula variables, 9
 term variables, 8

A Abstract Syntax

[to do]

B Concrete Syntax

[to do]

C Examples

[to do]