The SMT-LIB Standard: Version 1.0
First Draft

Silvio Ranise* Cesare Tinelli† The SMT-LIB Working Group‡

31 May 2004

Abstract

[abstract here]

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

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

‡See the acknowledgments section for the full list of contributors.

Contents

1 Introduction

This paper defines and discusses Version 1.0 of a language standard for the Satisfiability Modulo Theories Library, or SMT-LIB for short. The main goal of the SMT-LIB initiative [2], coordinated by these authors and supported by a large number of researcher 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 modulo (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 belief 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 it 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 *satisfiability procedure* any procedure for satisfiability modulo some given theory. With satisfiability procedures one can distinguish among

1. a procedure's *underlying logic*

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

2. a procedure's *background theory*, the theory against which satisfiability is checked, and

3. a procedure's *input language*, that is, the class of formulas the procedure accepts as input.

2.1 The Logic

To start with, Version 1 of the SMT-LIB standards adopts as its logic a basic many-sorted version of classical 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 will be added in the future as needed. For instance, it is expected that version 2 of the standard will include subsorts.

In an attempt to combine the simplicity and familiarity of unsorted logic with the convenience of a sorted language, the SMT-LIB standard defines two 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 semantics are discussed 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 it 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.[2] The theory itself (i.e. its axioms) can be defined either formally or informally, 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 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.[3]

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 **??**. 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 to specific sublanguage of its benchmarks belongs to.

3 The SMT-LIB Logic and Formula 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.

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

[3] By efficiency here we do not necessary refer to worst-case time complexity, but to efficiency "in practice".

As a typed language, the SMT-LIB language 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.

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:

- 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*;

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

$x \in \mathcal{X}$,	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 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 justaposition, as this is enough at the abstract level. Parentheses are metasymbols, 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.

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

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

[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 $\mathcal{F}var(\varphi)$ of free formula variables in a formula φ is instead defined as

follows

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

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

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

3.1 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 *ite* expressions 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

$$(\varphi_0 \textbf{ implies } \varphi_1) \textbf{ and } (\textbf{not } \varphi_0 \textbf{ implies } \varphi_1)$$

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

The *let* construct for terms

This construct, which builds terms of the form

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

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 formulas of the form

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

The rationale for supporting it in SM-LIB is similar 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

$$\textbf{distinct}(t_1, \ldots, t_n)$$

with $n \geq 2$. Semantically, it is equivalent to the conjunction of all disequations of the form $\textbf{not}(t_i = t_j)$ 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 of 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

$$\forall\, x_1{:}s_1 \ldots x{:}s_k\ \varphi_0\ \alpha$$

might specify an instantiation pattern for the quantifier $\forall\, x_1{:}s_1 \ldots x{:}s_k$, 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.[4]

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

The sequence of sorts associated by Σ *to a function/predicate symbol is called the* rank *of the symbol*[5] □

[4] A similar idea is used in the ESC/Java system with the use of a special "label" predicate [**?**].

[5] As usual, the rank of a function symbol specifies the expected sort of the symbol's argument and result. Similarly for predicate symbols.

$$\frac{}{\Sigma \vdash_{\mathrm{t}} x : s} \text{ if } x : s \in \Sigma \qquad \frac{}{\Sigma \vdash_{\mathrm{t}} n : s} \text{ if } n : s \in \Sigma$$

$$\frac{\Sigma \vdash_{\mathrm{t}} t_1 : s_1 \quad \cdots \quad \Sigma \vdash_{\mathrm{t}} t_k : s_k}{\Sigma \vdash_{\mathrm{t}} f\ t_1\ \cdots\ t_k : s_{k+1}} \text{ if } f : s_1 \cdots s_{k+1} \in \Sigma$$

$$\frac{\Sigma \vdash_{\mathrm{f}} \varphi \qquad \Sigma \vdash_{\mathrm{t}} t_1 : s \qquad \Sigma \vdash_{\mathrm{t}} t_2 : s}{\Sigma \vdash_{\mathrm{t}} \textbf{ite}\ \varphi\ t_1\, t_2 : s}$$

Figure 2: Well-sortedness rules for terms

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 ambguity in the language because positional information is enough to determing during parsing if a given occurrence of a symbol is a function, predicate or sort symbol. However, 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 another 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.

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 $f \in \Sigma^{\mathrm{F}}$ 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 $\Sigma \vdash_{\mathrm{t}} t{:}s$ is derivable by the sort rules in Figure 2 for some sort $s \in \Sigma^{\mathrm{S}}$. 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 $\Sigma \vdash_{\mathrm{f}} \varphi$ is derivable by the sort rules in Figure 3.

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

$$\frac{}{\Sigma \vdash_{\mathrm{f}} \mathbf{true}} \qquad \frac{}{\Sigma \vdash_{\mathrm{f}} \mathbf{false}} \qquad \frac{}{\Sigma \vdash_{\mathrm{f}} \xi}$$

$$\frac{\Sigma \vdash_{\mathrm{t}} t_1 : s_1 \quad \cdots \quad \Sigma \vdash_{\mathrm{t}} t_k : s_k}{\Sigma \vdash_{\mathrm{f}} p\ t_1\ \cdots\ t_k} \text{ if } p : s_1 \cdots s_k \in \Sigma$$

$$\frac{\Sigma \vdash_{\mathrm{t}} t_1 : s_1 \quad \cdots \quad \Sigma \vdash_{\mathrm{t}} t_{k+2} : s_{k+2}}{\Sigma \vdash_{\mathrm{f}} = t_1\ \cdots\ t_{k+2}} \qquad \frac{\Sigma \vdash_{\mathrm{t}} t_1 : s_1 \quad \cdots \quad \Sigma \vdash_{\mathrm{t}} t_{k+2} : s_{k+2}}{\Sigma \vdash_{\mathrm{f}} \mathbf{distinct}\ t_1\ \cdots\ t_{k+2}}$$

$$\frac{\Sigma \vdash_{\mathrm{f}} \varphi}{\Sigma \vdash_{\mathrm{f}} \mathbf{not}\ \varphi} \qquad \frac{\Sigma \vdash_{\mathrm{f}} \varphi_1 \quad \Sigma \vdash_{\mathrm{f}} \varphi_2}{\Sigma \vdash_{\mathrm{f}} \mathbf{impl}\ \varphi_1\ \varphi_2} \qquad \frac{\Sigma \vdash_{\mathrm{f}} \varphi_0 \quad \Sigma \vdash_{\mathrm{f}} \varphi_1 \quad \Sigma \vdash_{\mathrm{f}} \varphi_2}{\Sigma \vdash_{\mathrm{f}} \mathbf{if}\ \varphi_0\ \mathbf{then}\ \varphi_1\ \mathbf{else}\ \varphi_2}$$

$$\frac{\Sigma \vdash_{\mathrm{f}} \varphi_1 \quad \cdots \quad \Sigma \vdash_{\mathrm{f}} \varphi_{k+2}}{\Sigma \vdash_{\mathrm{f}} c\ \varphi_1\ \cdots\ \varphi_{k+2}} \text{ if } c \in \{\mathbf{and}, \mathbf{or}, \mathbf{xor}, \mathbf{iff}\}$$

$$\frac{\Sigma,\ x_1{:}s_1, \ldots, x{:}s_{k+1} \vdash_{\mathrm{f}} \varphi}{\Sigma \vdash_{\mathrm{f}} \exists x_1{:}s_1 \ldots x{:}s_{k+1}\ \varphi} \qquad \frac{\Sigma,\ x_1{:}s_1, \ldots, x{:}s_{k+1} \vdash_{\mathrm{f}} \varphi}{\Sigma \vdash_{\mathrm{f}} \forall x_1{:}s_1 \ldots x{:}s_{k+1}\ \varphi}$$

$$\frac{\Sigma \vdash_{\mathrm{t}} t : s \quad \Sigma, x : s \vdash_{\mathrm{f}} \varphi}{\Sigma \vdash_{\mathrm{f}} \mathbf{let}\ x\ = t\ \mathbf{in}\ \varphi} \qquad \frac{\Sigma \vdash_{\mathrm{f}} \varphi_0 \quad \Sigma \vdash_{\mathrm{f}} \varphi_1}{\Sigma \vdash_{\mathrm{f}} \mathbf{flet}\ \xi\ = \varphi_0\ \mathbf{in}\ \varphi_1}$$

Figure 3: Well-sortedness rules for formulas

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 $\mathcal{V}ar(\varphi) = \{x_1, \ldots, x_n\}$, where each x_i is expected to have sort s_i, can be rewritten as

$$\exists\, x_1{:}s_1 \ldots x_n{:}s_n\, \varphi.$$

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.

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 :**funs**, :**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

$$
\begin{array}{lrcl}
\text{(Fun. sym. declaration)} & D_f & ::= & f\ s^+\ \alpha \\
\text{(Pred. sym. declaration)} & D_p & ::= & p\ s^*\ \alpha \\
\text{(Attribute-value pair)} & P_T & ::= & :\mathbf{sorts} = s^+ \\
 & & | & :\mathbf{funs} = (D_f)^+ \ \mid\ :\mathbf{preds} = (D_p)^+ \\
 & & | & :\mathbf{definition} = w \ \mid\ :\mathbf{axioms} = \varphi^+ \\
 & & | & :\mathbf{extensions} = w \ \mid\ :\mathbf{notes} = w \ \mid\ \alpha \\
\text{(Theory declaration)} & D_T & ::= & \mathbf{theory}\ T\ \mathbf{begin} \\
 & & & \quad (P_T)^+ \\
 & & & \mathbf{end}
\end{array}
$$

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

Figure 4: Abstract syntax for theories

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, *the only legal theory declarations in the SMT-LIB format are those that*

1. *contain the attributes* :**sort** *and* :**definition** *(making those attributes non-optional) and*
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 declation semantics, and the automated processing of theory declations.

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**, :**funs** 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 :**funs** and :**preds** attributes are optional because a theory might lack function or predicate symbols.[7] 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 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 (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 $\underbrace{t + \cdots + t}_{n \text{ times}}$.

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

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

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

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

Figure 5: Abstract syntax for benchmarks

$$
\begin{array}{lrcl}
\text{(Attribute-value pair)} & P_S & ::= & :\textbf{theory} = T \mid :\textbf{benchmarks} = (D_b)^+ \\
 & & \mid & :\textbf{extrafuns} = (D_f)^+ \mid :\textbf{extrapreds} = (D_p)^+ \\
 & & \mid & :\textbf{language} = w \mid :\textbf{notes} = w \mid \alpha \\
\text{(Bench. set declaration)} & D_S & ::= & \textbf{benchset } S \textbf{ begin} \\
 & & & \quad (P_B)^+ \\
 & & & \textbf{end}
\end{array}
$$

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

Figure 6: Abstract syntax for benchmark sets

5 The SMT-LIB Benchmark Language

In SMT-LIB, a benchmarks 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 production 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.

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.*

The formulas φ_0 and φ_1 contained respectively in the attributes :**assumption** and :**formula** of a benchmark declaration 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, where φ_0 is taken to be just **true** in case 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.

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 is information is useful for tayloring 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 :**extrafuns** are "uninterpreted" in associated theory.[8] 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 SMT-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 3.2, 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

[8] In logic parlance, they are *free* symbols for the theory.

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 be 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.]

6 Semantics

[to do]

6.1 Model-theoretic Semantics

[to do]

6.2 Translation 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

⟨*identifier*⟩ ::= *a sequence of letters, digits, dots (.), and underscores (_), starting with a letter*

⟨*numeral*⟩ ::= *a non-empty sequence of digits*

⟨*var*⟩ ::= `?`⟨*identifier*⟩

⟨*fvar*⟩ ::= `$`⟨*identifier*⟩

⟨*attribute*⟩ ::= `:`⟨*identifier*⟩

⟨*ar_operator*⟩ ::= *an "arithmetic operator" symbol such as* $+, *, -, /$, *etc.*

⟨*rel_operator*⟩ ::= *a "relational operator" symbol such as* $<, <=$, *etc.*

⟨*fun_symb*⟩ ::= ⟨*identifier*⟩ | ⟨*ar_operator*⟩

⟨*pred_symb*⟩ ::= ⟨*identifier*⟩ | ⟨*rel_operator*⟩

⟨*sort_symb*⟩ ::= ⟨*identifier*⟩

⟨*annotation*⟩ ::= ⟨*attribute*⟩ | ⟨*attribute*⟩ ⟨*value*⟩

⟨*basic term*⟩ ::= ⟨*variable*⟩ | ⟨*numeral*⟩ | ⟨*fun_symb*⟩

⟨*an_term*⟩ ::= ⟨*basic term*⟩ | `(` ⟨*basic term*⟩ ⟨*annotation*⟩$^+$ `)`
| `(` ⟨*fun_symb*⟩ ⟨*an_term*⟩$^+$ ⟨*annotation*⟩* `)`
| `(` `ite` ⟨*an_formula*⟩ ⟨*an_term*⟩ ⟨*an_term*⟩ ⟨*annotation*⟩* `)`

⟨*basic atom*⟩ ::= `true` | `false` | ⟨*fvar*⟩ | ⟨*pred_symb*⟩

⟨*an_atom*⟩ ::= ⟨*basic atom*⟩ | `(` ⟨*basic atom*⟩ ⟨*annotation*⟩$^+$ `)`
| `(` ⟨*pred_symb*⟩ ⟨*term*⟩$^+$ ⟨*annotation*⟩* `)`
| `(` `=` ⟨*term*⟩ ⟨*term*⟩$^+$ ⟨*annotation*⟩* `)`
| `(` `distinct` ⟨*term*⟩ ⟨*term*⟩$^+$ ⟨*annotation*⟩* `)`

⟨*connective*⟩ ::= `not` | `implies` | `if_then_else`
| `and` | `or` | `xor` | `iff`

⟨*quant_symb*⟩ ::= `exists` | `forall`

⟨*quant_var*⟩ ::= `(` ⟨*var*⟩ ⟨*sort_symb*⟩ `)`

⟨*an_formula*⟩ ::= ⟨*an_atom*⟩
| `(` ⟨*connective*⟩ ⟨*an_formula*⟩$^+$ ⟨*annotation*⟩* `)`
| `(` ⟨*quant_symb*⟩ ⟨*quant_var*⟩$^+$ ⟨*an_formula*⟩ ⟨*annotation*⟩* `)`
| `(` `let` `(` ⟨*var*⟩ ⟨*an_term*⟩ `)` ⟨*an_formula*⟩ ⟨*annotation*⟩* `)`
| `(` `flet` `(` ⟨*fvar*⟩ ⟨*an_formula*⟩ `)` ⟨*an_formula*⟩ ⟨*annotation*⟩* `)`

Figure 7: Concrete syntax for unsorted terms and formulas

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.[9]

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 the 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.

[is # okay as a comment symbol? or should we use one of the others? (//, %, etc.)]

7.2 Terms and formulas

The concrete syntax for SMT-LIB unsorted terms and formulas is given in Figure 7 with BNF production rules based on the abstrat syntax rules given in Figure 1.

[more?]

7.3 Theories

The concrete syntax for SMT-LIB theory declarations is given in Figure 8 with BNF production rules based on the abstrat 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 9 and Figure 10 with BNF production rules based on the abstrat syntax rules given in Figure 5 and Figure 6, respectively.

[more?]

8 Examples

[to do]

[9] The set of concrete terminal symbols includes the open and closed parenthesis symbols.

⟨*string*⟩	::=	*any sequence of characters other than double quotes (")*, *and enclosed in double quotes*
⟨*fun_symb_decl*⟩	::=	`(` ⟨*fun_symb*⟩ ⟨*sort_symb*⟩* ⟨*annotation*⟩* `)`
⟨*pred_symb_decl*⟩	::=	`(` ⟨*pred_symb*⟩ ⟨*sort_symb*⟩* ⟨*annotation*⟩* `)`
⟨*theory_name*⟩	::=	⟨*identifier*⟩
⟨*theory_attribute*⟩	:=	`:notes` ⟨*string*⟩
	|	`:sorts (` ⟨*sort_symb*⟩$^+$ `)`
	|	`:funs (` ⟨*fun_symb_decl*⟩$^+$ `)`
	|	`:preds (` ⟨*pred_symb_decl*⟩$^+$ `)`
	|	`:extensions` ⟨*string*⟩
	|	`:definition` ⟨*string*⟩
	|	`:axioms` ⟨*string*⟩
	|	⟨*annotation*⟩
⟨*theory*⟩	::=	`(` `theory` ⟨*theory_name*⟩ ⟨*theory_attribute*⟩$^+$ `)`

Figure 8: Concrete syntax for theories

⟨*status*⟩	::=	`sat` | `unsat` | `unknown`
⟨*bench_name*⟩	::=	⟨*identifier*⟩
⟨*bench_attribute*⟩	:=	`:assumption` ⟨*annot_formula*⟩
	|	`:formula` ⟨*annot_formula*⟩
	|	`:status` ⟨*status*⟩
	|	⟨*annotation*⟩
⟨*benchmark*⟩	::=	`(` `benchmark` ⟨*bench_name*⟩ ⟨*bench_attribute*⟩$^+$ `)`

Figure 9: Concrete syntax for benchmarks

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

Figure 10: Concrete syntax for benchmark sets

9 Acknowledgments

The following members of the SMT-LIB interest group, in alphabetical order, provided suggestions and comments and feedback on the SMT-LIB 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?]

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/`.