The SMT-LIB Standard
Version 2.5

Clark Barrett Pascal Fontaine Aaron Stump Cesare Tinelli

Release: September 27, 2014

Preface

The SMT-LIB initiative is an international effort, supported by several research groups worldwide, with the two-fold goal of producing an extensive on-line library of benchmarks and promoting the adoption of common languages and interfaces for SMT solvers. This document specifies Version 2.5 of the *SMT-LIB Standard* which is a largely backward-compatible extension of Version 2.0.

Acknowledgments

Version 2.0

Version 2.0 of the SMT-LIB standard was developed with the input of the whole SMT community and three international work groups consisting of developers and users of SMT tools: the SMT-API work group, led by A. Stump, the SMT-LOGIC work group, led by C. Tinelli, the SMT-MODELS work group, led by C. Barrett.

Particular thanks are due to the following work group members, who contributed numerous suggestions and helpful constructive criticism in person or in email discussions: Nikolaj Bjørner, Sascha Boehme, David Cok, David Deharbe, Bruno Dutertre, Pascal Fontaine, Vijay Ganesh, Alberto Griggio, Jim Grundy, Paul Jackson, Albert Oliveras, Sava Krstić, Michal Moskal, Leonardo de Moura, Philipp Rümmer, Roberto Sebastiani, and Johannes Waldmann.

Many thanks also to David Cok, Morgan Deters, Anders Franzén, Amit Goel, Jochen Hoenicke, and Tjark Weber for additional feedback on the standard, and to Jochen Hoenicke, Philipp Rümmer, and above all David Cok, for their careful proof-reading of earlier versions of this document. It is understood that we, the authors, are to blame for any remaining errors.

Version 2.5

Version 2.5, the current one, was developed again with the input of the SMT community, based on their experience with Version 2.0.

Special thanks for Version 2.5 go to the following people for their extensive feedback and useful suggestions: Martin Brain, David Cok, Jürgen Christ, Morgan Deters, Bruno Dutertre, Alberto Griggio, Jochen Hoenicke, Tianyi Liang, Margus Veanes, and Christoph Wintersteiger. [CT: More?]

Contents

List of Figures

Part I

Introduction

Chapter 1

General Information

1.1 About This Document

This document is mostly self-contained, though it assumes some familiarity with first-order logic, *aka* predicate calculus. The reader is referred to any of several textbooks on the topic [Gal86, Fit96, End01, Men09]. Previous knowledge of Version 1.2 of the SMT-LIB standard [RT06] is not necessary. In fact, Version 1.2 users are warned that this version, while largely based on Version 1.2, is *not* backward compatible with it. See the Version 2.0 document [BST10b] for a summary of the major differences.

This document provides BNF-style abstract and concrete syntax for a number of SMT-LIB languages. *Only the concrete syntax is part of the official SMT-LIB standard.* The abstract syntax is used here mainly for descriptive convenience; adherence to it is not prescribed. Implementors are free to use whatever internal structure they please for their abstract syntax trees.

New releases of the document are identified by their release date. Each new release of the same version of the SMT-LIB standard contains, by and large, only *conservative* additions and changes with respect to the standard described in the previous release, as well as improvements to the presentation. The only non-conservative changes may be error fixes.

Historical notes and explanations of the rationale of design decisions in the definition of the SMT-LIB standard are provided in Appendix A, with reference in the main text given as a superscript number enclosed in parentheses.

1.1.1 Differences with Version 2.0

Version 2.5 is an extension of Version 2.0 and, with two minor exceptions, is fully backward compatible with it. There is then no need to have separate support for 2.0 if one supports Version 2.5. The following list summarizes all differences and extensions. The first two items are the only non-backward compatible changes.

- There is now a different set of escape sequences for string literals. It consists of a single sequence, `""`, used to represent the double quote character within the literal.
- The attribute `:status`, which was predefined but unused in Version 2.0, is no longer predefined.
- We have clarified several points about the character set used by the SMT-LIB language and specified more precisely which characters are allowed in string literals, identifiers and symbols.
- The use of the term attribute `:pattern` and its related syntax for quantifier patterns has been made official.
- The command `set-info` has been renamed `meta-info`. The old name is still accepted but its use is now deprecated.
- The solver option `interactive-mode` has been renamed `produced-assertions`. The old name is still accepted but its use is now deprecated.
- The new command `reset-assertions` empties the assertion stack and remove all global assertions and declarations.
- The new command `reset` brings the state of a solver to the state it had immediately after start up.
- There is a new solver option `:global-declarations` that makes all definitions and declarations global and not removable by `pop` operations. Global declarations can be removed only by `reset-assertions` and `reset`.
- The new command `check-sat-assuming` checks the satisfiability of the current context under an additional number of assumptions provided as input. When the command returns `unsat`, a companion command, `get-unsat-assumptions`, can be used to know which assumptions were actually used to prove the context unsatisfiable. The latter command is enabled or disabled with the new option `:produce-unsat-assumptions`.
- The new command `declare-const` can be now used to declare nullary function symbols.
- The new command `echo` prints back on the regular output channel a string provided as input.
- The new command `define-fun-rec` allows the definition of recursive functions (more generally, of sets of mutually recursive functions).
- The new command `get-model` returns a representation of a model computed by the solver in response to an invocation of the `check-sat` command.

- The new `get-info` flag `:assertion-stack-levels` returns the current number of levels in the assertion stack.
- The new option `:reproducible-resource-limit` can be used to set a solver-defined resource limit that applies to each invocation of `check-sat` or `check-sat-assumptions`.

1.1.2 Typographical and notational conventions

The concrete syntax of the SMT-LIB language is defined by means of BNF-style production rules. In the concrete syntax notation, terminals are written in typewriter font, as in `false`, while syntactic categories (non-terminals) are written in slanted font and enclosed in angular brackets, as in $\langle term \rangle$. In the production rules, the meta-operator ::= and | are used as usual in BNF. Also as usual, the meta-operators $\_^*$ and $\_^+$ denote zero, respectively, one, or more repetitions of their argument. We refer to input characters by their decimal or hexadecimal code in the Extended ASCII character set. We use the notation d_{dec} (resp., e_{hex}) to represent a character from this set with decimal code d (resp., hexadecimal code e).

Examples of concrete syntax expressions are provided in shaded boxes like the following.

```
(f (- x) x)
```

In the abstract syntax notation, which uses the same meta-operators as the concrete syntax, words in **boldface** as well as the symbols $\approx, \exists, \forall$, and Π denote terminal symbols, while words in *italics* and Greek letters denote syntactic categories. For instance, x, σ are non-terminals and **Bool** is a terminal. Parentheses are meta-symbols, used just for grouping—they are not part of the abstract language. Function applications are denoted simply by juxtaposition, which is enough at the abstract level.

To simplify the notation, when there is no risk of confusion, the name of an abstract syntactic category is also used, possibly with subscripts, to denote individual elements of that category. For instance, t is the category of terms and t, together with t_1, t_2 and so on, is also used to denote individual terms.

The meta-syntax $\bar{x}$ denotes a sequence of the form $x_1 x_2 \cdots x_n$ for some $x_1, x_2, \ldots, x_n$ and $n \geq 0$.

1.2 Overview of SMT-LIB

Satisfiability Modulo Theories (SMT) is an area of automated deduction that studies methods for checking the satisfiability of first-order formulas with respect to some logical theory $\mathcal{T}$ of interest [BSST09]. What distinguishes SMT from general automated deduction is that the background theory $\mathcal{T}$ need not be finitely or even first-order axiomatizable, and that specialized inference methods are used for each theory. By being theory-specific and restricting their language to certain classes of formulas (such as, typically but not exclusively,

quantifier-free formulas), these specialized methods can be implemented in solvers that are more efficient in practice than general-purpose theorem provers.

While SMT techniques have been traditionally used to support deductive software verification, they have found applications in other areas of computer science such as, for instance, planning, model checking and automated test generation. Typical theories of interest in these applications include formalizations of various forms of arithmetic, arrays, finite sets, bit vectors, algebraic datatypes, equality with uninterpreted functions, and various combinations of these.

1.2.1 What is SMT-LIB?

SMT-LIB is an international initiative, coordinated by a subset of these authors and endorsed by a large number of research groups world-wide, aimed at facilitating research and development in SMT [BST10a]. Since its inception in 2003, the initiative has pursued these aims by focusing on the following concrete goals: provide standard rigorous descriptions of background theories used in SMT systems; develop and promote common input and output languages for SMT solvers; establish and make available to the research community a large library of benchmarks for SMT solvers.

The main motivation of the SMT-LIB initiative was the expectation that the availability of common standards and of a library of benchmarks would greatly facilitate the evaluation and the comparison of SMT systems, and advance the state of the art in the field, in the same way as, for instance, the TPTP library [Sut09] has done for theorem proving, or the SATLIB library [HS00] has done initially for propositional satisfiability. These expectations have been largely met, thanks in no small part to extensive benchmark contributions from the research community and to an annual SMT solver competition, SMT-COMP [BdMS05], based on benchmarks from the library.

At the time of this writing, the library contains more than 100,000 benchmarks and continues to grow. Formulas in SMT-LIB format are accepted by the great majority of current SMT solvers. Moreover, most published experimental work in SMT relies significantly on SMT-LIB benchmarks.

1.2.2 Main features of the SMT-LIB standard

The previous main version of the SMT-LIB standard, Version 1.2, provided a language for specifying theories, logics (see later), and benchmarks, where a benchmark was, in essence, a logical formula to be checked for satisfiability with respect to some theory.

Version 2.0 seeked to improve the usefulness of the SMT-LIB standard by simplifying its logical language while increasing its expressiveness and flexibility. In addition, it introduced a command language for SMT solvers that expanded their SMT-LIB interface considerably, allowing users to tap the numerous functionalities that most modern SMT solvers provide.

Like Version 2.0, Version 2.5 defines:

- a language for writing *terms and formulas* in a sorted (i.e., typed) version of first-order logic;
- a language for specifying *background theories* and fixing a standard vocabulary of sort, function, and predicate symbols for them;
- a language for specifying *logics*, suitably restricted classes of formulas to be checked for satisfiability with respect to a specific background theory;
- a *command* language for interacting with SMT solvers via a textual interface that allows asserting and retracting formulas, querying about their satisfiability, examining their models or their unsatisfiability proofs, and so on.

Chapter 2

Basic Assumptions and Structure

This chapter introduces the defining basic assumptions of the SMT-LIB standard and describes its overall structure.

2.1 Satisfiability Modulo Theories

The defining problem of Satisfiability Modulo Theories is checking whether a given (closed) logical formula φ is *satisfiable*, not in general but in the context of some background theory $\mathcal{T}$ which constrains the interpretation of the symbols used in φ. Technically, the SMT problem for φ and $\mathcal{T}$ is the question of whether there is a model of $\mathcal{T}$ that makes φ true.

A dual version of the SMT problem, which we could call *Validity Modulo Theories*, asks whether a formula φ is *valid* in some theory $\mathcal{T}$, that is, satisfied by every model of $\mathcal{T}$. As the name suggests, SMT-LIB focuses only on the SMT problem. However, at least for classes of formulas that are closed under logical negation, this is no restriction because the two problems are inter-reducible: a formula φ is valid in a theory $\mathcal{T}$ exactly when its negation is not satisfiable in the theory.

Informally speaking, SMT-LIB calls an *SMT solver* any software system that implements a procedure for satisfiability modulo some given theory. In general, one can distinguish among a solver's

1. *underlying logic*, e.g., first-order, modal, temporal, second-order, and so on,
2. *background theory*, the theory against which satisfiability is checked,
3. *input formulas*, the class of formulas the solver accepts as input, and
4. *interface*, the set of functionalities provided by the solver.

For instance, in a solver for linear arithmetic the underlying logic is first-order logic with equality, the background theory is the theory of real numbers, and the input language is

often limited to conjunctions of inequations between linear polynomials. The interface may be as simple as accepting a system of inequations and returning a binary response indicating whether the system is satisfiable or not. More sophisticated interfaces include the ability to return concrete solutions for satisfiable inputs, return proofs for unsatisfiable ones, allow incremental and backtrackable input, and so on.

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

2.2 Underlying Logic

Version 2.5 of the SMT-LIB format adopts as its underlying logic a version of many-sorted first-order logic with equality [Man93, Gal86, End01]. Like traditional many-sorted logic, it has sorts (i.e., basic types) and sorted terms. Unlike that logic, however, it does not have a syntactic category of formulas distinct from terms. Formulas are just sorted terms of a distinguished Boolean sort, which is interpreted as a two-element set in every SMT-LIB theory.[1] Furthermore, the SMT-LIB logic uses a language of sort terms, as opposed to just sort constants, to denote sorts: sorts can be denoted by sort constants like `Int` as well as sort terms like `(List (Array Int Real))`. Finally, in addition to the usual existential and universal quantifiers, the logic includes a *let* binder analogous to the local variable binders found in many programming languages.

SMT-LIB's underlying logic, henceforth *SMT-LIB logic*, provides the formal foundations of the SMT-LIB standard. The concrete syntax of the logic is part of the SMT-LIB language of formulas and theories, which is defined in Part II of this document. An abstract syntax for SMT-LIB logic and the logic's formal semantics are provided in Part III.

2.3 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.[2] Theories are specified in SMT-LIB independently of any benchmarks or solvers. On the other hand, each SMT-LIB script refers, indirectly, to one or more theories in the SMT-LIB catalog.

This version of the SMT-LIB standard distinguishes between *basic* theories and *combined* theories. Basic theories, such as the theory of real numbers, the theory of arrays, the theory of lists and so on, are those explicitly defined in the SMT-LIB catalog. Combined theories are defined implicitly in terms of basic theories by means of a general modular combination operator. The difference between a basic theory and a combined one in SMT-LIB is essentially operational. Some SMT-LIB theories, such as the theory of finite sets with

[1] This is similar to some formulations of classical higher-order logic, such as that of [And86].

[2] This catalog is available, separately from this document, from the SMT-LIB website (`www.smt-lib.org`).

a cardinality operator, are defined as basic theories, even if they are in fact a combination of smaller theories, because they cannot be obtained by modular combination.

Theory specifications have mostly documentation purposes. They are meant to be standard references for human readers. For practicality then, the format insists that only the *signature* of a theory (essentially, its set of sort and sorted function symbols) be specified formally—provided it is finite.[3] By "formally" here we mean written in a machine-readable and processable format, as opposed to written in free text, no matter how rigorously. By this definition, theories themselves are defined informally, in natural language. Some theories, such as the theory of bit vectors, have an infinite signature. For them, the signature too is specified informally in English.(1)

2.4 Input Formulas

SMT-LIB adopts a single and general first-order (sorted) language in which to write logical formulas. It is often the case, however, that SMT applications work with formulas expressed in some particular fragment of the language. The fragment in question matters because one can often write a solver specialized on that sublanguage that is a lot more efficient than a solver meant for a larger sublanguage.[4]

An extreme case of this situation occurs when satisfiability modulo a given theory $\mathcal{T}$ 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 [BMS06]. But a similar situation occurs even when the decidability of the satisfiability problem is preserved across various fragments. For instance, if $\mathcal{T}$ is the theory of real numbers, the satisfiability in $\mathcal{T}$ of full-first order formulas built with the symbols $\{0, 1, +, *, <, =\}$ is decidable. However, one can implement increasingly faster solvers by restricting the language respectively to quantifier-free formulas, linear equations and inequations, difference inequations (inequations of the form $x < y + n$), and inequations between variables [BBC+05].

Certain pairs of theories and input languages are very common in the field and are often conveniently considered as a single entity. In recognition of this practice, the SMT-LIB format allows one to pair together a background theory and an input language into a *sublogic*, or, more briefly, *logic*. We call these pairs (sub)logics because, intuitively, each of them defines a sublogic of SMT-LIB logic for restricting both the set of allowed models—to the models of the background theory—and the set of allowed formulas—to the formulas in the input language.

[3] The finiteness condition can be relaxed a bit for signatures that include certain commonly used sets of constants such as the set of all numerals.

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

2.5 Interface

Starting with Version 2.0, the SMT-LIB standard includes a scripting language that defines a textual interface for SMT solvers. SMT solvers implementing this interface act as interpreters of the scripting language. The language is command-based, and defines a number of input/output functionalities that go well beyond simply checking the satisfiability of an input formula. It includes commands for setting various solver parameters, declaring new symbols, asserting and retracting formulas, checking the satisfiability of the current set of asserted formulas, inquiring about models of satisfiable sets, and printing various diagnostics.

Part II

Syntax

Chapter 3

The SMT-LIB Language

This chapter defines and explains the concrete syntax of the SMT-LIB standard, what we comprehensively refer to as the *the SMT-LIB language*. The SMT-LIB language has three main components: *theory declarations*, *logic declarations*, and *scripts*. Its syntax is similar to that of the LISP programming language. In fact, every expression in this version is a legal *S-expression* of Common Lisp [Ste90]. The choice of the S-expression syntax and the design of the concrete syntax was mostly driven by the goal of simplifying parsing, as opposed to facilitating human readability.(2)

The three main components of the language are defined in this chapter by means of BNF-style production rules. The rules, with additional details, are also provided in Appendix B. The language generated by these rules is actually a superset of the SMT-LIB language. The legal expressions of the language must satisfy additional constraints, such as well-sortedness, also specified in this document.

3.1 Lexicon

The syntax rules in this chapter are given directly with respect to streams of lexical tokens from the set defined in this section. The whole set of concrete syntax rules is also available for easy reference in Appendix B.

SMT-LIB source files consist of Extended ASCII characters, specifically the ISO 8859-1 8-bit extension, also called ISO Latin 1, of the original 7-bit US-ASCII set.(3)

Most lexical tokens defined below are limited to US-ASCII printable characters, namely, characters 32_{dec} to 126_{dec}. The remaining printable characters, characters 160_{dec} to 255_{dec}, mostly used for non-English alphabet letters, are allowed in string literals, quoted symbols and comments.

A *comment* is any character sequence not contained within a string literal or a quoted symbol (see later) that begins with the semi-colon character ; and ends with the first subsequent line-breaking character, i.e., 10_{dec} or 13_{dec}. Both comments and consecutive white space characters occurring outside a string literal or a symbol (see later) are considered

whitespace. The only lexical function of whitespace is to break the source text into tokens.[1]

The lexical tokens of the language are the parenthesis characters `(` and `)`, the elements of the syntactic categories ⟨*numeral*⟩, ⟨*decimal*⟩, ⟨*hexadecimal*⟩, ⟨*binary*⟩, ⟨*string*⟩, ⟨*symbol*⟩, ⟨*keyword*⟩, as well as a number of *reserved words*, all defined below together with a few auxiliary syntactic categories.

White Space Characters. A ⟨*white_space_char*⟩ is one of the following characters: 9_{dec} (tab), 10_{dec} (line feed), 13_{dec} (carriage return), 32_{dec} (space), and 160_{dec} (non-breaking space).

Printable Characters. A ⟨*printable_char*⟩ is any character from 32_{dec} to 126_{dec} (US-ASCII) and from 160_{dec} to 255_{dec}.[2]

Digits. A ⟨*digit*⟩ is any character from 48_{dec} to 57_{dec} (`0` through `9`)

Letters. A ⟨*letter*⟩ is any character from 65_{dec} to 90_{dec} (English alphabet letters `A` through `Z`) and from 97_{dec} to 122_{dec} (English alphabet letters `a` through `z`).(4)

Numerals. A ⟨*numeral*⟩ is the digit `0` or a non-empty sequence of digits not starting with `0` .

Decimals. A ⟨*decimal*⟩ is a token of the form ⟨*numeral*⟩`.0`*⟨*numeral*⟩ .

Hexadecimals. A ⟨*hexadecimal*⟩ is a non-empty *case-insensitive* sequence of digits and letters from `A` to `F` preceded by the (case sensitive) characters `#x` .

```
#x0       #xA04
#x01Ab    #x61ff
```

Binaries. A ⟨*binary*⟩ is a non-empty sequence of the characters `0` and `1` preceded by the characters `#b` .

```
#b0       #b1
#b001     #b101011
```

[1] Which implies that the language's semantics does not depend on indentation and spacing.

[2] Note that the space and the non-breaking space characters are both printable and whitespace characters.

String literals. A ⟨*string*⟩ (literal) is any sequence of characters from ⟨*printable_char*⟩ or ⟨*white_space_char*⟩ delimited by the double quote character " (34_{dec}). The character " can itself occur *within* a string only if duplicated. In other words, after an initial " that starts a literal, a lexer should treat the sequence "" as an escape sequence denoting a single occurrence of " within the literal.

```
""    "this is a string literal"

"She said: ""No way!"" and left."

"this is a string literal

with a line break in it"
```

SMT-LIB string literals are akin to *raw strings* in certain programming languages. However, they have only one escape sequence: "". This means, for example and in contrast to most programming languages, that within a ⟨*string*⟩ the character sequences `\n`, `\012`, `\x0A`, and `\u0008` are *not* escape sequences (all denoting the new line character), but regular sequences denoting their individual characters.(5)

Reserved words. The language uses a number of reserved words, sequences of printable characters that are to be treated as individual tokens. The basic set of reserved words consists of the following:

```
BINARY  DECIMAL  HEXADECIMAL  NUMERAL  STRING

_  !  as  let  exists  forall  par
```

Additionally, each command name in the scripting language defined in Section 3.9 (`set-logic`, `set-option`, ...) is also a reserved word.(6)

Symbols. A ⟨*symbol*⟩ is either a simple symbol or a quoted symbol. A *simple symbol* is any non-empty sequence of elements of ⟨*letter*⟩ and ⟨*digit*⟩ and the characters

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

that does not start with a digit and is not a reserved word.[3]

```
+   <=   x plus ** $ <sas   <adf>

abc77 *$s&6    .kkk    .8    +34    -32
```

[3] Note that simple symbols cannot contain non-English letters.

A *quoted symbol* is any sequence of whitespace characters and printable characters that starts and ends with | and does not contain | or \ .(7)

```
|this is a quoted symbol|

|so is
 this one|

||

| " can occur too|

|af klj^*0asfe2(&*)&(#^$>>>?"']]984|
```

Symbols are case sensitive. They are used mainly as operators or identifiers. Conventionally, arithmetic characters and the like are used, individually or in combination, as operator names; in contrast, alpha-numeric symbols, possibly with punctuation characters and underscores, are used as identifiers. But, as in LISP, this usage is only recommended (for human readability), not prescribed. For additional flexibility, arbitrary sequences of whitespace and printable characters (except for | and \) enclosed in vertical bars are also allowed as symbols. Following Common Lisp's convention, enclosing a simple symbol in vertical bars does not produce a new symbol. This means for instance that `abc` and `|abc|` are the *same* symbol.

Keywords. A ⟨*keyword*⟩ is a token of the form :⟨*simple_symbol*⟩ . Elements of this category have a special use in the language. They are used as *attribute* names or *option* names (see later).

```
:date   :a2   :foo-bar
:<=     :56   :->
```

3.2 S-expressions

An S-expression is either a non-parenthesis token or a (possibly empty) sequence of S-expressions enclosed in parentheses. Every syntactic category of the SMT-LIB language is a specialization of the category ⟨*s_expr*⟩ defined by the production rules below.

⟨*spec_constant*⟩ ::= ⟨*numeral*⟩ | ⟨*decimal*⟩ | ⟨*hexadecimal*⟩ | ⟨*binary*⟩
| ⟨*string*⟩

⟨*s_expr*⟩ ::= ⟨*spec_constant*⟩ | ⟨*symbol*⟩ | ⟨*keyword*⟩ | (⟨*s_expr*⟩*)

Remark 1. Elements of the ⟨*spec_constant*⟩ category do not always have the expected associated semantics in the SMT-LIB language (i.e., elements of ⟨*numeral*⟩ denoting integers, elements of ⟨*string*⟩ denoting character strings, and so on). In particular, in the ⟨*term*⟩ category (defined later) they simply denote constant symbols, with no fixed, predefined semantics. Their semantics is determined locally by each SMT-LIB theory that uses them. For instance, it is possible in principle for an SMT-LIB theory of sets to use the numerals `0` and `1` to denote respectively the empty set and universal set. Similarly, the elements of ⟨*binary*⟩ may denote integers modulo n in one theory and binary strings in another; the elements of ⟨*decimal*⟩ may denote rational numbers in one theory and floating point values in another.

3.3 Identifiers

Identifiers are used mostly as function and sort symbols. When defining certain SMT-LIB theories it is convenient to have indexed identifiers as well. Instead of having a special token syntax for that, indexed identifiers are defined more systematically as the application of the reserved word `_` to a symbol and one or more *indices*. Indices can be numerals or simple symbols.[8]

⟨*index*⟩ ::= ⟨*numeral*⟩ | ⟨*simple symbol*⟩
⟨*identifier*⟩ ::= ⟨*symbol*⟩ | `(` `_` ⟨*symbol*⟩ ⟨*index*⟩$^+$ `)`

```
plus    +       <=    Real    |John Brown|
(_ vector-add 4 5)
(_ move up)  (_ move down)  (_ move left)  (_ move right)
```

There are several namespaces for identifiers (sorts, terms, commands, ...). Identifiers in different namespaces can share names with no risk of conflict because the particular namespace can always be identified syntactically. Within the term namespace, bound variables can shadow one another as well as function symbol names. Similarly, bound sort parameters can shadow one another and sort symbol names.

3.4 Attributes

Several syntactic categories in the language contain *attributes*. These are generally pairs consisting of an attribute name and an associated value, although attributes with no value are also allowed.

Attribute names belong to the ⟨*keyword*⟩ category. Attribute values are in general S-expressions other than keywords, although most predefined attributes use a more restricted category for their values.

⟨*attribute_value*⟩ ::= ⟨*spec_constant*⟩ | ⟨*symbol*⟩ | (⟨*s_expr*⟩*)
⟨*attribute*⟩ ::= ⟨*keyword*⟩ | ⟨*keyword*⟩ ⟨*attribute_value*⟩

```
:left-assoc
:status unsat
:my_attribute (humpty dumpty)
:authors "Jack and Jill"
```

3.5 Sorts

A major subset of the SMT-LIB language is the language of *well-sorted* terms, used to represent logical expressions. Such terms are typed, or *sorted* in logical terminology; that is, each is associated with a (unique) *sort*. The set of sorts consists itself of *sort terms*. In essence, a sort term is a *sort symbol*, a *sort parameter*, or a sort symbol applied to a sequence of sort terms.

Syntactically, a sort symbol can be either the distinguished symbol `Bool` or any ⟨*identifier*⟩. A sort parameter can be any ⟨*symbol*⟩ (which in turn, is an ⟨*identifier*⟩).

⟨*sort*⟩ ::= ⟨*identifier*⟩ | (⟨*identifier*⟩ ⟨*sort*⟩+)

```
Int                              Bool

(_ BitVec 3)                     (List (Array Int Real))

((_ FixedSizeList 4) Real)       (Set (_ Bitvec 3))
```

3.6 Terms and Formulas

Well-sorted terms are a subset of the set of all *terms*. The latter are constructed out of constant symbols in the ⟨*spec_constant*⟩ category (numerals, rationals, strings, etc.), *variables*, *function symbols*, three kinds of *binders*—the reserved words `let`, `forall` and `exists`—and an annotation operator—the reserved word `!` .

A variable can be any ⟨*symbol*⟩, while a function symbol can be any ⟨*identifier*⟩ (i.e., a symbol or an indexed symbol). As explained later, every function symbol f is separately associated with one or more *ranks*, each specifying the sort of f's arguments and result. To simplify sort checking, a function symbol in a term can be annotated with one of its result sorts σ. Such an annotated function symbol is a *qualified identifier* of the form `(as` f σ`)`.

Formulas are just well-sorted terms of sort `Bool`. As a consequence, there is no syntactic distinction between function and predicate symbols; the latter are simply function symbols

whose result sort is `Bool`. Another consequence is that function symbols can take formulas (even quantified ones) as arguments.

⟨*qual_identifier*⟩ ::= ⟨*identifier*⟩ | `(` `as` ⟨*identifier*⟩ ⟨*sort*⟩ `)`
⟨*var_binding*⟩ ::= `(` ⟨*symbol*⟩ ⟨*term*⟩ `)`
⟨*sorted_var*⟩ ::= `(` ⟨*symbol*⟩ ⟨*sort*⟩ `)`
⟨*term*⟩ ::= ⟨*spec_constant*⟩
| ⟨*qual_identifier*⟩
| `(` ⟨*qual_identifier*⟩ ⟨*term*⟩$^+$ `)`
| `(` `let` `(` ⟨*var_binding*⟩$^+$ `)` ⟨*term*⟩ `)`
| `(` `forall` `(` ⟨*sorted_var*⟩$^+$ `)` ⟨*term*⟩ `)`
| `(` `exists` `(` ⟨*sorted_var*⟩$^+$ `)` ⟨*term*⟩ `)`
| `(` `!` ⟨*term*⟩ ⟨*attribute*⟩$^+$ `)`

In its simplest form, a term is a special constant symbol, a variable, a function symbol, or the application of a function symbol to one or more terms. Function symbols applied to no arguments are used as constant symbols.

```
(forall ((x (List Int)) (y (List Int)))
  (= (append x y)
     (ite (= x (as nil (List Int)))
          y
          (let ((h (head x)) (t (tail x)))
            (insert h (append t y))))))
```

Binders. More complex terms include `let`, `forall` and `exists` binders. The `forall` and `exists` binders correspond to the usual existential and universal quantifiers of first-order logic, except that the variables they quantify are sorted. A `let` binder introduces and defines one or more local variables *in parallel*. Semantically, a term of the form

`(let ((`x_1 t_1`)` $\cdots$ `(`x_n t_n`)) `t`)`

is equivalent to the term obtained from t by simultaneously replacing each free occurrence of x_i in t by t_i, for each $i = 1, \ldots, n$, possibly after a suitable renaming of t's bound variables to avoid variable capturing. The language does not have a sequential version of `let`. Its effect is achieved by nesting lets, as in

`(let ((`x_1 t_1`)) (let ((`x_2 t_2`)) `t`))`

All binders follow a lexical scoping discipline, enforced by SMT-LIB logic's semantics as described in Section 5.3. Note that all variables bound by a binder are elements of the ⟨*symbol*⟩ category—they cannot be indexed identifiers.

Well-sortedness requirements. All terms of the SMT-LIB language are additionally required to be well-sorted. Well-sortedness constraints are discussed in Section 5.2 in terms of the logic's abstract syntax.

Annotations. Every term t can be optionally annotated with one or more attributes $\alpha_1, \ldots, \alpha_n$ using the wrapper expression `(!` t α_1 $\cdots$ α_n`)`. Term attributes have no logical meaning—semantically `(!` t α_1 $\cdots$ α_n`)` is equivalent to t—but they are a convenient mechanism for adding meta-logical information for SMT solvers.

Term attributes. Currently there are only two predefined term attributes: `:named` and `:pattern`.

The values of the `:named` attribute range over the ⟨*symbol*⟩ category. The attribute can be used in scripts to give a closed term a symbolic name, which can be then used as a proxy for the term (see Section 4.2).

```
(=> (! (> x y) :named p1)
    (! (= x z) :named p2))
```

The values of the `:pattern` attribute ranges over sequences of ⟨*term*⟩ elements. The attribute is used to define instantiation patterns for quantifiers, which provides heuristic information to SMT solvers that do quantifier instantiation. Instantiation patterns can be used only annotate the body φ of a quantified formula of the form

$$(Q\ ((x_1\ t_1)\ \cdots\ (x_k\ t_k))\ \varphi)$$

where Q is `forall` or `exists` so that the resulting formula has the form

$$\begin{aligned}(Q\ ((x_1\ t_1)\ \cdots\ (x_k\ t_k))\ (!\ \varphi\ &\texttt{:pattern}\ (p_{1,1}\ \cdots\ p_{1,n_1})\\ &\vdots\\ &\texttt{:pattern}\ (p_{m,1}\ \cdots\ p_{m,n_m})))\end{aligned}$$

where each $p_{i,j}$ is a binder-free term with no annotations and the same well-sortedness requirements as for the formula's body. In particular, every variable in $p_{i,j}$ that is not global to the formula must be among $x_1, \ldots, x_k$.

⟨*sort_symbol_decl*⟩ ::= (⟨*identifier*⟩ ⟨*numeral*⟩ ⟨*attribute*⟩*)

⟨*meta_spec_constant*⟩ ::= NUMERAL | DECIMAL | STRING

⟨*fun_symbol_decl*⟩ ::= (⟨*spec_constant*⟩ ⟨*sort*⟩ ⟨*attribute*⟩*)
| (⟨*meta_spec_constant*⟩ ⟨*sort*⟩ ⟨*attribute*⟩*)
| (⟨*identifier*⟩ ⟨*sort*⟩+ ⟨*attribute*⟩*)

⟨*par_fun_symbol_decl*⟩ ::= ⟨*fun_symbol_decl*⟩
| (par (⟨*symbol*⟩+)
(⟨*identifier*⟩ ⟨*sort*⟩+ ⟨*attribute*⟩*))

⟨*theory_attribute*⟩ ::= :sorts (⟨*sort_symbol*⟩+)
| :funs (⟨*par_fun_symbol_decl*⟩+)
| :sorts-description ⟨*string*⟩
| :funs-description ⟨*string*⟩
| :definition ⟨*string*⟩
| :values ⟨*string*⟩
| :notes ⟨*string*⟩
| ⟨*attribute*⟩

⟨*theory_decl*⟩ ::= (theory ⟨*symbol*⟩ ⟨*theory_attribute*⟩+)

Figure 3.1: Theory declarations.

```
(forall ((x0 A) (x1 A) (x2 A))
  (! (=> (and (r x0 x1) (r x1 x2)) (r x0 x2))
   :pattern ((r x0 x1) (r x1 x2))
   :pattern ((p x0 a))
  ))
```

The intended use of these patterns is to suggest to the solver to find independently for each $i = 1, \ldots, m$ terms that simultaneously match the pattern terms $p_{i,1} \cdots p_{i,n_i}$.

3.7 Theory Declarations

The set of SMT-LIB theories is defined by a catalog of *theory declarations* written in the format specified in this section. This catalog is available on the SMT-LIB web site at www.smt-lib.org. In the previous version of the SMT-LIB standard, a theory declaration defined both a many-sorted *signature*, i.e., a collection of sorts and sorted function symbols, and a theory with that signature. The signature was determined by the collection of individual declarations of sort symbols and function symbols with an associated *rank*—specifying the sorts of the symbol's arguments and of its result.

From Version 2.0 on, theory declarations are similar to those of Version 1.2, except that

they may declare entire families of overloaded function symbols by using ranks that contain *sort parameters*, locally scoped sort symbols of arity 0. Additionally, a theory declaration generally defines a *class* of similar theories.

The syntax of theory declarations, specified in Figure 3.1, follows an attribute-value-based format. A theory declaration consists of a theory name and a list of ⟨*attribute*⟩ elements. Theory attributes with the following predefined keywords have a prescribed usage and semantics: `:definition`, `:funs`, `:funs-description`, `:notes`, `:sorts`, `:sorts-description`, and `:values`. Additionally, a theory declaration can contain any number of user-defined attributes.(9)

Theory attributes can be *formal* or *informal* depending on whether or not their value has a formal semantics and can be processed in principle automatically. The value of an informal attribute is free text, in the form of a ⟨*string*⟩ value. For instance, the attributes `:funs` and `:sorts` are formal in the sense above, whereas `:definition`, `:funs-description` and `:sorts-description` are not.

A theory declaration `(theory` T α_1 $\cdots$ α_n`)` defines a *theory schema* with name T and attributes $\alpha_1, \ldots, \alpha_n$. Each instance of the schema is a theory $\mathcal{T}_\Sigma$ with an *expanded* signature Σ, containing (zero or more) additional sort and function symbols with respect to those declared in T. Examples of instances of theory declarations are discussed later.

The value of a `:sorts` attribute is a non-empty sequence of sort symbol declarations ⟨*sort_symbol_decl*⟩. A sort symbol declaration `(`s n α_1 $\cdots$ α_n`)` declares a sort symbol s of arity n, and may additionally contain zero or more annotations $\alpha_1, \ldots, \alpha_n$, each in the form of an ⟨*attribute*⟩. In this version, there are no predefined annotations for sort declarations.

The value of a `:funs` attribute is a non-empty sequence of possibly parametric function symbol declarations ⟨*par_fun_symbol_decl*⟩. A (non-parametric) function symbol declaration ⟨*fun_symbol_decl*⟩ of the form `(`c σ`)`, where c is an element of ⟨*spec_constant*⟩, declares c to have sort σ. For convenience, it is possible to declare all the special constants in ⟨*numeral*⟩ to have sort σ by means of the function symbol declaration `(NUMERAL` σ`)`. This is done for instance in the theory declaration in Figure 3.3. The same can be done for the set of ⟨*decimal*⟩ and ⟨*string*⟩ constants by using `DECIMAL` and `STRING`, respectively.

A (non-parametric) function symbol declaration `(`f σ_1 $\cdots$ σ_n σ`)` with $n \geq 0$ declares a function symbol f with rank $\sigma_1 \cdots \sigma_n \sigma$. Intuitively, this means that f takes as input n values of respective sort $\sigma_1, \ldots, \sigma_n$, and returns a value of sort σ. On the other hand, a parametric function symbol declaration `(par (`u_1 $\cdots$ u_k`) (`f τ_1 $\cdots$ τ_n τ`))` with $k > 0$ and $n \geq 0$, declares a whole class of function symbols, all named f and each with a rank obtained from $\tau_1 \cdots \tau_n \tau$ by instantiating each occurrence in $\tau_1 \cdots \tau_n \tau$ of the sort parameters $u_1, \ldots, u_k$ with non-parametric sorts. See Section 5.4 for more details.

As with sorts, each (parametric) function symbol declaration may additionally contain zero or more annotations $\alpha_1, \ldots, \alpha_n$, each in the form of an ⟨*attribute*⟩.

In this version, there are only 4 predefined function symbol annotations, all attributes with no value: `:chainable`, `:left-assoc`, `:right-assoc`, and `:pairwise`. The `:left-assoc` annotation can be added only to function symbol declarations of the form

$$(f\ \sigma_1\ \sigma_2\ \sigma_1)\ \text{ or }\ (\texttt{par}\ (u_1\ \cdots\ u_k)\ (f\ \tau_1\ \tau_2\ \tau_1)).$$

Then, an expression of the form $(f\ t_1\ \cdots\ t_n)$ with $n > 2$ is allowed as syntactic sugar (recursively) for $(f\ (f\ t_1\ \cdots\ t_{n-1})\ t_n)$. Similarly, the `:right-assoc` annotation can be added only to function symbol declarations of the form

$$(f\ \sigma_1\ \sigma_2\ \sigma_2)\ \text{ or }\ (\texttt{par}\ (u_1\ \cdots\ u_k)\ (f\ \tau_1\ \tau_2\ \tau_2)).$$

Then, $(f\ t_1\ \cdots\ t_n)$ with $n > 2$ is syntactic sugar for $(f\ t_1\ (f\ t_2\ \cdots\ t_n))$.

The `:chainable` and `:pairwise` annotations can be added only to function symbol declarations of the form

$$(f\ \sigma\ \sigma\ \texttt{Bool})\ \text{ or }\ (\texttt{par}\ (u_1\ \cdots\ u_k)\ (f\ \tau\ \tau\ \texttt{Bool}))$$

and are mutually exclusive. With the first annotation, $(f\ t_1\ \cdots\ t_n)$ with $n > 2$ is syntactic sugar for $(\texttt{and}\ (f\ t_1\ t_2)\ \cdots\ (f\ t_{n-1}\ t_n))$ where `and` is itself a symbol declared as `:left-assoc` in every theory (see Subsection 3.7.1); with the second, $(f\ t_1\ \cdots\ t_n)$ is syntactic sugar (recursively) for $(\texttt{and}\ (f\ t_1\ t_2)\ \cdots\ (f\ t_1\ t_n)\ (f\ t_2\ \cdots\ t_n))$.

```
(+ Real Real Real :left-assoc)

(and Bool Bool Bool :left-assoc)

(par (X) (insert X (List X) (List X) :right-assoc))

(< Real Real Bool :chainable)

(equiv Elem Elem Bool :chainable)

(par (X) (Disjoint (Set X) (Set X) Bool :pairwise))

(par (X) (distinct X X Bool :pairwise))
```

For many theories in SMT-LIB, in particular those with a finite signature, it is possible to declare all of their symbols using a finite number of sort and function symbol declarations in `:sorts` and `:funs` attributes. For others, such as for instance, the theory of bit vectors, one would need infinitely many such declarations. In those cases, sort symbols and function symbols are defined informally, in plain text, in `:sorts-description`, and `:funs-description` attributes, respectively.(10)

```
:sorts_description
 "All sort symbols of the form  (_ BitVec m)  with m > 0."
```

```
:funs_description
 "All function symbols with rank of the form

    (concat (_ BitVec i) (_ BitVec j) (_ BitVec m))

  where i,j > 0 and i + j = m."
```

The `: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.(11) For other theories, a mix of formal notation and natural language might be more appropriate. In the presence of parametric function symbol declarations, the definition must also specify the meaning of each instance of the declared symbol.(12)

The attribute `:values` is used to specify, for each sort σ, a distinguished, decidable set of ground terms of sort σ that are to be considered as *values* for σ. We will call these terms *value terms*. Intuitively, given an instance theory containing a sort σ, σ's set of value terms is a set of terms that denotes, in each countable model of the theory, all the elements of that sort. These terms might be over a signature with additional function symbols with respect to those specified in the theory declaration. Ideally, the set of value terms is minimal, which means that no two distinct terms in the set denote the same element in some model of the theory. However, this is only a recommendation, not a requirement because it is impractical, or even impossible, to satisfy it for some theories. See the next subsection for examples of value sets, and Section 5.5 for a more in-depth explanation.

The attribute `:notes` is meant to contain documentation information on the theory declaration such as authors, date, version, references, etc., although this information can also be provided with more specific, user-defined attributes.

Constraint 1 (Theory Declarations)**.** The only legal theory declarations of the SMT-LIB language are those that satisfy the following restrictions.

1. They contain exactly one instance of the `:definition` attribute[4].
2. Each sort symbol used in a `:funs` attribute is previously declared in some `:sorts` attribute.
3. The definition of the theory, however provided in the `:definition` attribute, refers only to sort and function symbols previously declared formally in `:sorts` and `:funs` attributes or informally in `:sorts-description` and `:funs-description` attributes.
4. In each parametric function symbol declaration `(par (`$u_1 \cdots u_k$`) (`$f\ \tau_1 \cdots \tau_n\ \tau$`))`, any symbol other than f that is not a previously declared sort symbol must be one of the sort parameters $u_1, \ldots, u_k$.

[4] Which makes that attribute non-optional.

```
(theory Core

 :sorts ((Bool 0))

 :funs ((true Bool)  (false Bool)  (not Bool Bool)
        (=> Bool Bool Bool :right-assoc)  (and Bool Bool Bool :left-assoc)
        (or Bool Bool Bool :left-assoc)  (xor Bool Bool Bool :left-assoc)
        (par (A) (= A A Bool :chainable))
        (par (A) (distinct A A Bool :pairwise))
        (par (A) (ite Bool A A A))
       )

 :definition
 "For every expanded signature Sigma, the instance of Core with that signature
  is the theory consisting of all Sigma-models in which:
  - the sort Bool denotes the set {true, false} of Boolean values;
  - for all sorts s in Sigma,
    - (= s s Bool) denotes the function that
      returns true iff its two arguments are identical;
    - (distinct s s Bool) denotes the function that
      returns true iff its two arguments are not identical;
    - (ite Bool s s) denotes the function that
      returns its second argument or its third depending on whether
      its first argument is true or not;
  - the other function symbols of Core denote the standard Boolean operators
    as expected.
  "

  :values "The set of values for the sort Bool is {true, false}."
)
```

Figure 3.2: The `Core` theory declaration.

The `:funs` attribute is optional in a theory declaration because a theory might lack function symbols (although such a theory would not be not very interesting).

3.7.1 Examples

Core theory

To provide the usual set of Boolean connectives for building formulas, in addition to the predefined logical symbol `distinct`, Version 2.5 defines a basic core theory which is implicitly included in every other SMT-LIB theory.[13] Concretely, every theory declaration is assumed to contain implicitly the `:sorts` and `:funs` attributes of the `Core` theory declaration shown in Figure 3.2, and to define the symbols in those attributes in the same way as in `Core`.

Note the absence of a symbol for double implication. Such a connective is superfluous because the equality symbol `=` can be used in its place. Note how the attributes specified in the declarations of the various symbols of this theory allow one to write such expression as

- `(=> x y z)`
- `(and x y z)`
- `(= x y z)`
- `(distinct x y z)`

respectively as abbreviations of the terms

- `(=> x (=> y z))`
- `(and (and x y) z)`
- `(and (= x y) (= y z))`
- `(and (distinct x y) (distinct x z) (distinct y z))` .

The simplest instance of `Core` is the theory with no additional sort and function symbols. In that theory there is only one sort, `Bool`, and `ite` has only one rank, `(ite Bool Bool Bool Bool)`.[5] In other words, this is just the theory of the Booleans with the standard Boolean operators plus `ite`. The set of values for the `Bool` sort is, predictably, {`true`, `false`}.

Another instance has a single additional sort symbol `U`, say, of arity 0, and a (possibly infinite) set number of function symbols with rank in $\texttt{U}^+$. This theory corresponds to *EUF*, the (one-sorted) theory of equality and *uninterpreted functions* (over those function symbols). In this theory, `ite` has two ranks: `(ite Bool Bool Bool Bool)` and `(ite Bool U U U)`. A many-sorted version of EUF is obtained by instantiating `Core` with more than one nullary sort symbol—and possibly additional function symbols over the resulting sort set.

Yet another instance is the theory with an additional unary sort symbol `List` and an additional number of function symbols. This theory has infinitely many sorts: `Bool`, `(List Bool)`, `(List (List Bool))`, etc. However, by the definition of `Core`, all those sorts and function symbols are still "uninterpreted" in the theory. In essence, this theory is the same as a many-sorted version of EUF with infinitely many sorts. While not very interesting in isolation, the theory is useful in combination with a theory of lists that, for each sort σ, interprets `(List` σ`)` as the set of all lists over σ. The combined theory in that case is a theory of lists with uninterpreted functions.

```
(theory Ints

 :sorts ((Int 0))

 :funs ((NUMERAL Int)
        (- Int Int)                 ; negation
        (- Int Int Int :left-assoc) ; subtraction
        (+ Int Int Int :left-assoc)
        (* Int Int Int :left-assoc)
        (<= Int Int Bool :chainable)
        (<  Int Int Bool :chainable)
        (>= Int Int Bool :chainable)
        (>  Int Int Bool :chainable)
       )

 :definition
 "For every expanded signature Sigma, the instance of Ints with that
  signature is the theory consisting of all Sigma-models that interpret
  - the sort Int as the set of all integers,
  - the function symbols of Ints as expected.
 "

 :values
 "The Int values are all the numerals and all the terms of the form (- n)
  where n is a non-zero numeral."
)
```

Figure 3.3: A possible theory declaration for the integer numbers.

Integers

The theory declaration of Figure 3.3 defines all theories that extend the standard theory of the (mathematical) integers to additional *uninterpreted* sort and function symbols.[6] The integers theory proper is the instance with no additional symbols. More precisely, since the `Core` theory declaration is implicitly included in every theory declaration, that instance is the two-sorted theory of the integers and the Booleans. The set of values for the `Int` sorts consists of all numerals and all terms of the form `(-` n`)` where n is a numeral other than 0.

Arrays with extensionality

A schematic version of the theory of functional arrays with extensionality is defined in the theory declaration `ArraysEx` in Figure 3.4. Each instance gives a theory of (arbitrarily

[5] That `ite` operator plays the role of the `if_then_else` connective in Version 1.2.

[6] For simplicity, the theory declaration in the figure is an abridged version of the declaration actually used in the SMT-LIB catalog.

```
(theory ArraysEx
 :sorts ( (Array 2) )
 :funs ( (par (X Y) (select (Array X Y) X Y))
         (par (X Y) (store (Array X Y) X Y (Array X Y))) )
 :notes
 "A schematic version of the theory of functional arrays with extensionality."
 :definition
 "For every expanded signature Sigma, the instance of ArraysEx with that
  signature is the theory consisting of all Sigma-models that satisfy all
  axioms of the form below, for all sorts s1, s2 in Sigma:

  - (forall ((a (Array s1 s2)) (i s1) (e s2))
      (= (select (store a i e) i) e))

  - (forall ((a (Array s1 s2)) (i s1) (j s1) (e s2))
      (=> (distinct i j) (= (select (store a i e) j) (select a j))))

  - (forall ((a (Array s1 s2)) (b (Array s1 s2)))
      (=>
        (forall ((i s1)) (= (select a i) (select b i))) (= a b)))
  "
  :values
  "For all sorts s1, s2, the values of sort (Array s1 s2) are either abstract
   or have the form (store a i v) where
   - a is value of sort (Array s1 s2),
   - i is a value of sort s1, and
   - j is a value of sort s2.
  "
)
```

Figure 3.4: The `ArraysEx` theory declaration.

nested) arrays. For instance, with the addition of the nullary sort symbols `Int` and `Real`, we get an instance theory whose sort set S contains, inductively, `Bool`, `Int`, `Real` and all sorts of the form `(Array` σ_1 σ_2`)` with $\sigma_1, \sigma_2 \in S$. This includes *flat array* sorts such as

`(Array Int Int)`, `(Array Int Real)`, `(Array Real Int)`, `(Array Bool Int)`,

conventional *nested array* sorts such as

`(Array Int (Array Int Real))`,

as well as nested sorts such as

`(Array (Array Int Real) Int)`, `(Array (Array Int Real) (Array Real Int))`

with an array sort in the *index position* of the outer array sort.[(14)]

The function symbols of the theory include all symbols with name `select` and rank of the form `((Array` σ_1 σ_2`)` σ_1 σ_2`)` for all $\sigma_1, \sigma_2 \in S$. Similarly for `store`.

Sets

[CT: Add a theory of sets (of urelements) to show later that SMT-LIB's underlying logic has the expressive power of HOL.]

Remark 2. For some applications, the instantiation mechanism defined here for theory declarations will definitely over-generate. For instance, it is not possible to define by instantiation of the `ArraysEx` declaration a theory of just the arrays of sort `(Array Int Real)`, without all the other nested array sorts over {`Int`, `Real`}. This, however, is a problem neither in theory nor in practice. It is not a problem in practice because, since a script can only use formulas with non-parametric sorts[7], any theory sorts that are not used in a script are, for all purposes, irrelevant. It is not a problem in theory either because scripts refer to logics, not directly to theories. And the language of a logic can always be restricted to contain only a selected subset of the sorts in the logic's theory.

3.8 Logic Declarations

The SMT-LIB format allows the explicit definition of sublogics of its main logic— a version of many-sorted first-order logic with equality—that restrict both the main logic's syntax and semantics. A new sublogic, or simply logic, is defined in the SMT-LIB language by a *logic declaration*; see www.smt-lib.org for the current catalog. Logic declarations have a similar format to theory declarations, although most of their attributes are informal.(15)

Attributes with the following predefined keywords have a prescribed usage and semantics in logic declarations: `:theories`, `:language`, `:extensions`, `:notes`, and `:values`. Additionally, as with theories, a logic declaration can contain any number of user-defined attributes.

⟨*logic_attribute*⟩ := `:theories (` ⟨*symbol*⟩$^+$ `)`
| `:language` ⟨*string*⟩
| `:extensions` ⟨*string*⟩
| `:values` ⟨*string*⟩
| `:notes` ⟨*string*⟩
| ⟨*attribute*⟩

⟨*logic*⟩ ::= `(logic` ⟨*symbol*⟩ ⟨*logic_attribute*⟩$^+$ `)`

A logic declaration `(logic` L α_1 $\cdots$ α_n`)` defines a logic with name L and attributes $\alpha_1, \ldots, \alpha_n$.

[7] Note that sort parameters cannot occur in a formula.

Constraint 2 (Logic Declarations)**.** The only legal logic declarations in the SMT-LIB language are those that satisfy the following restrictions:

1. They include exactly one instance of the **theories** attribute and of the **language** attribute.
2. The value $T_1, \ldots, T_n$ of the **theories** attribute lists names of theory schemas that have a declaration in SMT-LIB.
3. If two theory declarations among $T_1, \ldots, T_n$ declare the same sort symbol, they give it the same arity.

When the value of the `:theories` attribute is `(`$T_1 \ \cdots \ T_n$`)`, with $n > 0$, the logic refers to a combination $\mathcal{T}$ of specific instances of the theory declaration schemas $T_1, \ldots, T_n$. The exact combination mechanism that yields $\mathcal{T}$ is defined formally in Section 5.5. The effect of this attribute is to declare that the logic's sort and function symbols consist of those of the combined theory $\mathcal{T}$, and that the logic's semantics is restricted to the models of $\mathcal{T}$, as specified in more detail in Section 5.5.

The `:language` attribute describes in free text the logic's *language*, a specific class of SMT-LIB formulas. This information is useful for tailoring SMT solvers to the specific sublanguage of formulas used in an input script.(16) The formulas in the logic's language are built over (a subset of) the signature of the associated theory $\mathcal{T}$, as specified in this attribute. In the context of a command script the language of a logic is implicitly expanded by `let` constructs in formulas as well as user-defined sort and function symbols. In other words, a formula φ used in a script is considered to belong to a certain logic's language iff the formula obtained from φ by replacing all let variables and all defined sort and function symbol by their respective definition is in the language.

The optional `:extensions` attribute is meant to document any notational conventions, or syntactic sugar, allowed in the concrete syntax of formulas in this logic.(17)

The `:values` attribute has the same use as in theory declarations but it refers to the specific theories and sorts of the logic. It is meant to complement the `:values` attribute specified in the theory declarations referred to in the `:theories` attribute.

The textual `:notes` attribute serves the same purpose as in theory declarations.

3.8.1 Examples

Propositional logic

Standard propositional logic can be readily defined by an SMT-LIB logic declaration. The logic's theory is the instance of the `Core` theory declaration whose signature adds infinitely-many function symbols of arity `Bool` (playing the role of propositional variables). The language consists of all binder-free formulas over the expanded signature. Extending the language with let binders allows a faithful encoding of BDD's as formulas, thanks to the `ite` operator of `Core`.

Quantified Boolean logic

The logic of quantified Boolean formulas (QBFs) can be defined as well. The theory is again an instance of `Core` but this time with no additional symbols at all. The language consists of (closed) quantified formulas all of whose variables are of sort `Bool`.

Linear integer arithmetic

Linear integer arithmetic can be defined as an SMT-LIB logic. This logic is indeed part of the official SMT-LIB catalog of logics and is called `QF_LIA` there. Its theory is an extension of the theory of integers and the Booleans with uninterpreted constant symbols. That is, the instance of the theory declaration `Ints` from Figure 3.3 whose signature adds to the symbols of `Ints` infinitely many *free constants*, new function symbols of rank `Int` and of rank `Bool`.

The language of the logic is made of closed quantifier-free formulas (over the theory's signature) containing only *linear atoms*, that is, atomic formulas with no occurrences of the function symbol `*`. Extensions of the basic language include expressions of the form `(*` n t`)` and `(*` t n`)`, for some numeral $n > 1$, both of which abbreviate the term `(+` t $\cdots$ t`)` with n occurrences of t. Also included are terms with negative integer coefficients, that is, expressions of the form `(* (-` n`)` t`)` or `(*` t `(-` n`))` for some numeral $n > 1$, both of which abbreviate the expression `(- (*` n t`))`.

Higher-order logic

[CT: Define a higher-order simple predicate logic based on the theory of sets.]

3.9 Scripts

Scripts are sequences of *commands*. In line with the LISP-like syntax, all commands look like LISP-function applications, with a command name applied to zero or more arguments. To facilitate processing, each command takes a constant number of arguments, although some of these arguments can be (parenthesis delimited) lists of variable length. The full list of commands is provided in Figure 3.5.

The intended use of scripts is to communicate with an SMT-solver in a *read-eval-print loop*: until a termination condition occurs, the solver reads the next command, acts on it, outputs a response, and repeats. Possible responses vary from a single symbol to a list of attributes, to complex expressions like proofs.

The command `set-option` takes as an argument expressions of the syntactic category ⟨*option*⟩, which have the same form as attributes with values. Options with the predefined keywords listed in Figure 3.6 have a prescribed usage and semantics. Additional, solver-specific options are also allowed.

The command `get-info` takes as argument expressions of the syntactic category ⟨*info_flag*⟩ which are flags with the same form as keywords. The predefined flags listed in Figure 3.7 have a prescribed usage and semantics. Additional, solver-specific flags are also allowed.

```
⟨fun_def⟩       ::= ⟨symbol⟩ ( ⟨sorted_var⟩* ) ⟨sort⟩ ⟨term⟩ )
⟨fun_rec_def⟩   ::= ( ⟨fun_def⟩ )
⟨command⟩       ::= ( assert ⟨term⟩ )
                  | ( check-sat ) | ( check-sat-assuming ( ⟨symbol⟩* ) )
                  | ( declare-const ⟨symbol⟩ ⟨sort⟩ )
                  | ( declare-fun ⟨symbol⟩ ( ⟨sort⟩* ) ⟨sort⟩ )
                  | ( declare-sort ⟨symbol⟩ ⟨numeral⟩ )
                  | ( define-fun ⟨fun_def⟩ ) | ( define-fun-rec ( ⟨fun_rec_def⟩+ ) )
                  | ( define-sort ⟨symbol⟩ ( ⟨symbol⟩* ) ⟨sort⟩ )
                  | ( echo ⟨string⟩ ) | ( exit )
                  | ( get-assertions ) | ( get-assignment )
                  | ( get-info ⟨info_flag⟩ ) | ( get-model )
                  | ( get-option ⟨keyword⟩ ) | ( get-proof )
                  | ( get-unsat-assumptions ) | ( get-unsat-core )
                  | ( get-value ( ⟨term⟩+ ) ) | ( meta-info ⟨attribute⟩ )
                  | ( pop ⟨numeral⟩ ) | ( push ⟨numeral⟩ )
                  | ( reset ) | ( reset-assertions )
                  | ( set-info ⟨attribute⟩ ) | ( set-logic ⟨symbol⟩ )
                  | ( set-option ⟨option⟩ )
⟨script⟩        ::= ⟨command⟩*
```

Figure 3.5: SMT-LIB Commands.

Examples of the latter might be, for instance, flags such as `:time` and `:memory`, referring to used resources, or `:decisions`, `:conflicts`, and `:restarts`, referring to typical statistics for SMT solvers based on some extension of the DPLL procedure.

For more on error behavior, the meanings of the various options and info names, and the semantics of the various commands, see Chapter 4. We highlight a few salient points here and provide a couple of examples.

Assertion stack. Compliant solvers respond to various commands by performing operations on a data structure we call the *assertion stack*. This is a single global stack, whose elements, called *levels*, are *sets* of *assertions*. Assertions include logical formulas (that is, terms of sort `Bool`), as well as declarations and definitions of sort and function symbols. Assertions are added by specific commands. By default an assertion is local to the level that was most recent at the time the corresponding command was executed. Popping a level from the assertion stack removes all assertions in it, including symbol declarations and definitions. An input option allows the user to make all symbol declarations and definitions *global* to the whole assertion stack. With that option enabled, declarations and definitions are always added to a global level (which cannot be popped). Popping the most recent level then has only the effect of removing asserted formulas. Declarations and definitions can be removed only by a global reset operation with the `reset` or `reset-assertions` command.

⟨*b_value*⟩ ::= `true` | `false`

⟨*option*⟩ ::= `:diagnostic-output-channel` ⟨*string*⟩
| `:expand-definitions` ⟨*b_value*⟩
| `:global-declarations` ⟨*b_value*⟩
| `:interactive-mode` ⟨*b_value*⟩
| `:print-success` ⟨*b_value*⟩
| `:produce-assertions` ⟨*b_value*⟩
| `:produce-assignments` ⟨*b_value*⟩
| `:produce-models` ⟨*b_value*⟩
| `:produce-proofs` ⟨*b_value*⟩
| `:produce-unsat-cores` ⟨*b_value*⟩
| `:random-seed` ⟨*numeral*⟩
| `:regular-output-channel` ⟨*string*⟩
| `:reproducible-resource-limit` ⟨*numeral*⟩
| `:verbosity` ⟨*numeral*⟩
| ⟨*attribute*⟩

Figure 3.6: Command options.

⟨*info_flag*⟩ ::= `:all-statistics` | `:assertion-stack-levels` | `:authors`
| `:error-behavior` | `:name` | `:reason-unknown`
| `:version` | ⟨*keyword*⟩

Figure 3.7: Info flags.

Declared/defined symbols. Sort and function symbols introduced with a declaration or a definition cannot begin with a dot (`.`), as such symbols are reserved for future use, or with `@`, as such symbols are reserved for solver-defined *abstract values*.

Benchmarks. Starting with Version 2.0 of the SMT-LIB language, there is no explicit syntactic category of benchmarks. Instead, declarative information about a script used as a benchmark is included in the script via the `meta-info` command.

3.9.1 Command responses

The possible responses that a solver can produce in response to commands are defined as follows. General responses from ⟨*general_response*⟩ are to be used except when a specific response is specified in this document. In that case, an element of ⟨*specific_success_response*⟩ is returned in place of `success`, one of the elements of ⟨*general_response*⟩.

Regular output, including error messages, is printed on the *regular output channel*; diagnostic output, including warnings or progress information, on the *diagnostic output channel*. These may be set using `set-option` and the corresponding attributes (respectively, `:regular-output-channel` and `:diagnostic-output-channel`). The values of these attributes should be (double-quote delimited) file names in the format specified by the POSIX standard.[8] The string literals `"stdout"` and `"stderr"` are reserved to refer specially to the corresponding standard process channels (as opposed to disk files with that name).

Specific Responses. Specific responses are defined, in Figure 3.8, for the following commands:

⟨*check_sat_response*⟩	for	`check-sat` and `check-sat-assuming`
⟨*echo_response*⟩	for	`echo`,
⟨*get_assertions_response*⟩	for	`get-assertions`,
⟨*get_assignment_response*⟩	for	`get-assignment`,
⟨*get_info_response*⟩	for	`get-info`,
⟨*get_model_response*⟩	for	`get-model`,
⟨*get_option_response*⟩	for	`get-option`,
⟨*get_proof_response*⟩	for	`get-proof`,
⟨*get_unsat_assump_response*⟩	for	`get-unsat-assumptions`,
⟨*get_unsat_core_response*⟩	for	`get-unsat-core`,
⟨*get_value_response*⟩	for	`get-value`.

See Chapter 4 for more details.

3.9.2 Example scripts

We demonstrate some allowed behavior of a hypothetical solver in response to an example script. Each command is followed by example legal output from the solver in a comment, if there is any. The

[8] This is the usual format adopted by all Unix-based operating systems, with `/` used as a separator for (sub)directories, etc.

⟨*error-behavior*⟩ ::= `immediate-exit` | `continued-execution`

⟨*reason-unknown*⟩ ::= `memout` | `incomplete` | ⟨*s_expr*⟩

⟨*model_response*⟩ ::= `(` `define-fun` ⟨*fun_def*⟩ `)`
| `(` `define-fun-rec` `(` ⟨*fun_rec_def*⟩$^+$ `)` `)`

⟨*info_response*⟩ ::= `:assertion-stack-levels` ⟨*numeral*⟩
| `:authors` ⟨*string*⟩
| `:error-behavior` ⟨*error-behavior*⟩
| `:name` ⟨*string*⟩
| `:reason-unknown` ⟨*reason-unknown*⟩
| `:version` ⟨*string*⟩
| ⟨*attribute*⟩

⟨*valuation_pair*⟩ ::= `(` ⟨*term*⟩ ⟨*term*⟩ `)`

⟨*t_valuation_pair*⟩ ::= `(` ⟨*symbol*⟩ ⟨*b_value*⟩ `)`

⟨*check_sat_response*⟩ ::= `sat` | `unsat` | `unknown`

⟨*echo_response*⟩ ::= ⟨*string*⟩

⟨*get_assertions_response*⟩ ::= `(` ⟨*term*⟩* `)`

⟨*get_assignment_response*⟩ ::= `(` ⟨*t_valuation_pair*⟩* `)`

⟨*get_info_response*⟩ ::= `(` ⟨*info_response*⟩$^+$ `)`

⟨*get_model_response*⟩ ::= `(` ⟨*model_response*⟩$^+$ `)`

⟨*get_option_response*⟩ ::= ⟨*attribute_value*⟩

⟨*get_proof_response*⟩ ::= ⟨*s_expr*⟩

⟨*get_unsat_assump_response*⟩ ::= `(` ⟨*symbol*⟩* `)`

⟨*get_unsat_core_response*⟩ ::= `(` ⟨*symbol*⟩* `)`

⟨*get_value_response*⟩ ::= `(` ⟨*valuation_pair*⟩$^+$ `)`

⟨*specific_success_response*⟩ ::= ⟨*check_sat_response*⟩ | ⟨*echo_response*⟩
| ⟨*get_assertions_response*⟩ | ⟨*get_assignment_response*⟩
| ⟨*get_info_response*⟩ | ⟨*get_model_response*⟩
| ⟨*get_info_response*⟩ | ⟨*get_model_response*⟩
| ⟨*get_option_response*⟩ | ⟨*get_proof_response*⟩
| ⟨*get_unsat_assumptions_response*⟩
| ⟨*get_unsat_core_response*⟩ | ⟨*get_value_response*⟩

⟨*general_response*⟩ ::= `success` | ⟨*specific_success_response*⟩
| `unsupported` | `(` `error` ⟨*string*⟩ `)`

Figure 3.8: Command responses.

```
(set-logic QF_LIA)
; success

(declare-fun w () Int)
; success

(declare-fun x () Int)
; success

(declare-fun y () Int)
; success

(declare-fun z () Int)
; success

(assert (> x y))
; success

(assert (> y z))
; success

(set-option :print-success false)

(push 1)

(assert (> z x))

(check-sat)
; unsat

(get-info :all-statistics)
; (:time 0.01 :memory 0.2)

(pop 1)

(push 1)

(check-sat)
; sat

(exit)
```

Figure 3.9: Example script, over two columns (i.e. commands in the first column precede those in the second column), with expected solver responses in comments.

```
...

(set-option :print-success false)

(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun f (Int) Int)

(assert (= (f x) (f y)))
(assert (not (= x y)))

(check-sat)
; sat

(get-value (x y))
; ((x 0)
;  (y 1)
; )

(declare-fun a () (Array Int (List Int)))
```

```
...

(check-sat)
; sat

(get-value (a))
; ((a @_0_1)
; )

(get-value (select 2 @_0_1))
; (((select 2 @_0_1) @_2_0)
; )

(get-value ((first @_2_0) (rest @_2_0)))
; (((first @_2_0) 1)
;  ((rest @_2_0) nil)
; )
```

Figure 3.10: Another example script (excerpt), with expected solver responses in comments. [CT: To be revised]

script in Figure 3.9 makes two background assertions, and then conducts two independent queries. The `get-info` command requests information on the search using the `:all-statistics` flag.[9] The script in Figure 3.10 uses the `get-value` command to get information about a particular model of the formula which the solver has reported satisfiable.

[9] Since output of `(get-info :all-statistics)` is solver-specific, the response reported in the script is for illustration purposes only.

Part III

Semantics

Chapter 4

Operational Semantics of SMT-LIB

This chapter specifies how a human user or a software client of an SMT-LIB compliant solver can interact with it. We do that by providing as precisely as possible an informal semantics of SMT-LIB scripts, together with additional requirements on the input/output behavior of the solver.

The expected interaction mode with a compliant solver is interactive: the user or client issues commands in the format of the Command Language to the SMT solver via the solver's standard textual input channel; the solver then responds over two textual output channels, one for regular output and one for diagnostic output. A non-interactive mode is also allowed where the solver reads commands from a scripts stored in a file. However, the solver's output behavior should be exactly the same as if the commands in the script had been sent to it one at a time.

Note that the primary goal of the SMT-LIB standard is, first and foremost, to support convenient interaction with other programs, not human interaction. This has some influence on the design of the command language.

There are other commands one might wish for an SMT solver to support beyond those adopted here. In general, it is expected that time and more experience with the needs of applications will drive the addition of further commands in later versions.

4.1 General Requirements

The command language contains commands for managing a stack of *assertion levels*, and checking various information about them. They include commands to:

- declare and define new sort and function symbols,
- add formulas to the current assertion level,
- reset the assertion stack or the whole solver (*reset* commands),
- push and pop assertion levels,
- check the joint satisfiability of all formulas in the assertion stack, possibly under additional assumptions (*check commands*),
- obtain further information following a check command (e.g., model information),
- set values for standard and solver-specific options,

- get standard and solver-specific information from the solver.

This subsection provides some background and general requirements on how these functionalities are to be supported. The next subsection describes how each compliant solver is supposed execute each command.

4.1.1 Solver responses

Regular output, including responses *and errors*, produced by compliant solvers should be written to the regular output channel. Diagnostic output, including warnings, debugging, tracing, or progress information, should be written to the diagnostic output channel. These channels may be set with the ⟨*set-option*⟩ command (see Section 4.1.6 below). By default they are the standard output and standard error channels, respectively.

Generally, a solver completes its processing in response to a command, it should print to its standard output channel a ⟨*general_response*⟩:

⟨*general_response*⟩ ::= `success` | ⟨*specific_success_response*⟩
| `unsupported` | `(` `error` ⟨*string*⟩ `)`

The value `success` is the the default response for a successful execution of a supported command. A number of commands have a more detailed response in place of `success`, discussed in Section 4.2 with those commands. The value `unsupported` should be returned if the command is not supported by the solver. An expression of the form `(error` *e*`)` should be returned for any kind of error situation (wrong command syntax, incorrect parameters, erroneous execution, and so on). The argument given to `error` is a solver-specific string containing a message that describes the problem encountered.[1]

Any response which is not double-quoted and not parenthesized must be followed by at least one whitespace character (for example, a new line character).(18)

Several options described in Section 4.1.6 below affect the printing of responses, in particular by suppressing the printing of `success`, and by redirecting the standard and the error output.

Errors and solver state. Solvers have two options when encountering errors. For both options, they first print an error message in the ⟨*general_response*⟩ format. Then, they may either immediately exit with a non-zero exit status, or continue accepting commands. In the second case, the solver's state remains unmodified by the error-generating command, except possibly for timing and diagnostic information. In particular, the assertion stack, discussed in Section 4.1.3, is unchanged.(19)

The predefined `:error-behavior` attribute can be used with the `get-info` command to check which error behavior the tool supports (see Section 4.1.7 below).

4.1.2 Printing of terms and defined symbols

Several commands request the solver to print sets of terms. While some commands, naturally, place additional semantic requirements on these sets, the general syntactic requirement is that output terms must be well-sorted with respect to the current signature.

All output from a compliant solver should print any symbols defined with `define-sort` and `define-fun` just as they are, without replacing them by the expression they are defined to be equal to. This approach generally keeps output from solvers much more compact than with definitions expanded. An option is included below (Section 4.1.6), however, to expand all (non-recursive) definitions in solver output.

[1] Returning the empty string is allowed but discouraged because of its uninformative content.

4.1.3 The assertion stack

A compliant solver maintains a stack of sets, each of which consists of *assertions*. Assertions are formulas, declarations, and definitions. We will use the following terminology with regards to this data structure:

- *assertion stack:* the single stack of sets of assertions;
- *assertion level:* an element of the assertion stack (i.e., a set of assertions);
- *context:* the union of all the assertion levels on the assertion stack;
- *current assertion level:* the assertion level at the top of the stack (i.e., the most recent);
- *global assertion level:* the first assertion level in the stack (i.e., the least recent).

Initially, when the solver starts, the assertion stack consists of a single element, the global assertion level, which is empty. While new assertions can be added to this set, the set itself cannot be removed from the stack with a pop operation.

4.1.4 Symbol declarations and definitions

A number of commands allow the declaration or definition of a function or sort symbol. By default, these declarations and definitions are added at the most recent assertion level when the corresponding command is executed. Popping that assertion level removes them.(20) As an alternative, declarations and definitions can be made all *global* by running the solver with the option `:global-declarations`. With that option, they are always added at the global assertion level. As a consequence, they survive pop operations on the assertion stack and can be removed only by a global reset, achieved with a reset command.(21)

Well-sortedness checks, required for commands that use sorts or terms, are always done with respect to the *current signature*, determined by the logic specified with the most recent ⟨*set-logic*⟩ command and by the set of sort symbols and rank associations (for function symbols) in the current context. It is an error to declare or define a symbol that is already in the current signature. This implies in particular that, contrary to theory function symbols, user-defined functions symbols cannot be overloaded.(22)

4.1.5 In-line definitions

Any closed subterm t occurring in the argument(s) of a command c can be optionally annotated with a `:named` attribute, that is, can appear as `(`t `:named` f`)` where f is a fresh function symbol from ⟨*symbol*⟩. For such a command c, let

$$\texttt{(}t_1\ \texttt{:named}\ f_1\texttt{)}, \ldots, \texttt{(}t_n\ \texttt{:named}\ f_n\texttt{)}$$

be the in-order enumeration of all the named subterms of c. The effect of those annotations is the same, and has the same requirements, as the sequence of commands

$$\begin{array}{l}\texttt{(define-fun}\ f_1\ \texttt{()}\ \sigma_1\ t'_1\texttt{)}\\ \vdots\\ \texttt{(define-fun}\ f_1\ \texttt{()}\ \sigma_n\ t'_n\texttt{)}\\ c'\end{array}$$

where, for each $i = 1, \ldots, n$, (i) σ_i is the sort of t_i with respect to the current signature up to the declaration of f_i, (ii) t'_i is the term obtained from t_i by removing all its `:named` annotations, and (iii) c' is similarly obtained from c by removing all its `:named` annotations.

By this semantics, each *label* f_i can occur, as a constant symbol, in any subexpression of c that comes after $(t_i$ `:named` $f_i)$ in the in-order traversal of c, as well as after the command c itself. The labels $f_1, \ldots, f_n$ can be used like any other user-defined nullary function symbol, with the same visibility and scoping restrictions they would have if they had been defined with the sequence of commands above. However, contrary to function symbols introduced by `:define-fun`, labels have an additional, dedicated use in the commands `:get-assignment` and `:get-unsat-core` (see Section 4.2).

4.1.6 Solver options

Solvers options may be set using the `set-option` command, and their current values can be obtained using the `get-option` command.

Solver-specific option names are allowed and indeed expected. A set of standard options is presented in this subsection; please refer to Figure 3.6 for their format. This version of the language requires solvers to recognize and reply in a standard way only to a few of them, the rest are optional. We discuss each option below, specifying also their default values and whether or not compliant solvers are required to support them. Note that some options have restrictions on when they can be set. The set of standard options is likely to be expanded or otherwise revised as further desirable common options and kinds of information across tools are identified.

The options with a name starting with `produce` are Boolean options that enable specific commands. When one of these options is set to `false`, calling the corresponding command(s) should trigger an error response from the solver.

`:diagnostic-output-channel` default: `"stderr"` support: required
: The argument should be a filename to use subsequently for the diagnostic output channel. The input value `"stderr"` is interpreted specially to mean the solver's standard error channel. For other filenames, subsequent solver output should be appended to the named file (and the file should be first created if it does not already exist).

`:expand-definitions` default: `false` support: optional
: If the solver supports this option, setting it to `true` causes all subsequent output from the solver to be printed with all definitions fully expanded. That is, subsequent output contains no symbols defined with `define-sort` or `define-fun`; instead, they contain the (full expansions of the) expressions they are defined to be equal to. The option may not be set after `set-logic` without an intervening `reset`. Note that symbols introduced with `define-fun-rec` are not expanded.[23]

`:global-declarations` default: `false` support: optional
: If the solver supports this option, setting it to `true` causes all declarations and definitions to be added at the global assertion level of the assertion stack instead of the current assertion level. It may not be set after `set-logic` without an intervening `reset`.

`:interactive-mode` default: `false` support: optional
: The old name for `produce-assertions`. Deprecated.

`:print-success` default: `true` support: required
: Setting this option to `false` causes the solver to suppress the printing of `success` in all responses to commands. Other output remains unchanged.

`:produce-assertions` default: `false` support: optional
If the solver supports this option, setting it to `true` enables the `get-assertions` command. It may not be set after `set-logic` without an intervening `reset`. This options was called `interactive-mode` in Version 2.0.

`:produce-assignments` default: `false` support: optional
If supported, this enables the command `get-assignment`. It may not be set after `set-logic` without an intervening `reset`.

`:produce-models` default: `false` support: optional
If supported, this enables the commands `get-value` and `get-model`. It may not be set after `set-logic` without an intervening `reset`.

`:produce-proofs` default: `false` support: optional
If supported, this enables the command `get-proof`. It may not be set after `set-logic` without an intervening `reset`.

`:produce-unsat-assumptions` default: `false` support: optional
If supported, this enables the command `get-unsat-assumptions`. It may not be set after `set-logic` without an intervening `reset`.

`:produce-unsat-cores` default: `false` support: optional
If supported, this enables the command `get-unsat-core`. It may not be set after `set-logic` without an intervening `reset`.

`:random-seed` default: 0 support: optional
The argument is a numeral for the solver to use as a random seed, in case the solver uses (pseudo-)randomization. The default value of 0 means that the solver can use any random seed—possibly a different one for each run of the script.

`:regular-output-channel` default: `"stdout"` support: required
The argument should be a filename to use subsequently for the regular output channel. The input value `"stdout"` is interpreted specially to mean the solver's standard output channel. For other filenames, subsequent solver output should be appended to the named file (and the file should be first created if it does not already exist).

`:reproducible-resource-limit` default: 0 support: optional
If the solver supports this option, setting it to 0 disables it. Setting it to a positive value n will cause each subsequent check command to terminate within a bounded amount of time dependent on n. The internal implementation of this option and its relation to run time or other concrete resources can be solver-specific. However, it is required that the invocation of a check command return `unknown` whenever the solver is unable to determine the satisfiability of the formulas in the context within the current resource limit. Setting a higher value of n allows more resources to be used, which may cause the command to return `sat` or `unsat` instead of `unknown`. Furthermore, the returned result depends deterministically on n; specifically, it is the same every time the solver is run with the same sequence of previous commands on the same machine (and with an arbitrarily long external time out). If the solver makes use of randomization, it may require the `:random-seed` option to be set to a value other than 0 before `:reproducible-resource-limit` can be set to a positive value.[24]

`:verbosity` default: 0 support: optional
The argument is a numeral controlling the level of diagnostic output produced by the solver.

All such output should be written to the diagnostic output channel[25] which can be set and later changed via the `diagnostic-output-channel` option. An argument of 0 requests that no such output be produced. Higher values correspond to more verbose output.

4.1.7 Solver information

The format for responses to `get-info` commands, both for standard and solver-specific information flags, is defined by ⟨*get_info_response*⟩ category in Figure 3.8. The standard `get-info` flags and specific formats for their corresponding responses are given next.

`:all-statistics` support: optional
Solvers reply with a parenthesis-delimited sequence of ⟨*info_response*⟩ values (se Figure 3.8) providing various statistics on the execution of the most recent check command. No standard statistics are defined for the time being,[26] so they are all solver-specific. Any call to `get-info` with `:all-statistics` must follow a check command and precede any intervening commands that modify the assertion stack.

`:assertion-stack-levels` support: optional
The response is a pair of the form `(:assertion-stack-levels` *n*`)` where *n* is a numeral indicating the current number of levels in the assertion stack besides the global assertion level. Intuitively, this is the same as the current number of nested push commands.[27]

`:authors` support: required
The response is a pair of the form `(:authors` *s*`)` where *s* is a string literal listing the names of the solver's authors.

`:error-behavior` support: required
The response is a pair of the form `(:error-behavior` *r*`)` where *r* is either `immediate-exit` or `continued-execution`. With `immediate-exit`, the solver is states that it will exit immediately when an error is encountered. With `continued-execution`, the solver states that it will leave the state unmodified by the erroneous command, and continue accepting and executing new commands. See Section 4.1.1 for more information.

`:name` support: required
The response is a pair of the form `(:name` *s*`)` where *s* is a string literal with the name of the solver.

`:reason-unknown` support: optional
Any call to ⟨*get-info*⟩ with `:reason-unknown` must follow a check command that replied with `unknown` and there were no intervening commands that modify the assertion stack. The response is a pair of the form `(:reason-unknown` *r*`)` where *r* is an element of ⟨*reason_unknown*⟩ giving a short reason why the solver could not successfully check satisfiability. In general, this reason can be provided by a solver-defined s-expression. Two predefined s-expressions are `memout`, for out of memory, and `incomplete`, which indicates that the solver knows it is incomplete for the class of formulas containing the most recent check query.

`:version` support: required a singleton
The response is a string literal with the version number of the solver (e.g., `"1.2"`).

4.2 Commands

The full set of commands and their expected behavior is described in this section. The description of each command may contain a number of restrictions on the command's input. Additional restrictions may be imposed on when the command can be issued. Unless otherwise specified, the solver is required to produce an error when any of those restrictions is not satisfied.

4.2.1 (Re)starting and terminating

`(reset)` resets the solver completely to the state it had after it was started and before it started reading commands.(28)

`(set-logic` *L*`)` tells the solver what logic, in the sense of Section 5.5, is being used. The argument *L* can be the name of a logic in the SMT-LIB catalog or of some other solver-specific logic. On startup or after a `reset` command, the logic must be set before any commands can be executed, with the exception of the following: `exit`, `get-info`, `get-option`, `meta-info`, `reset`, `set-info`, `set-option`. Attempting to execute any other kind of command before `set-logic` results in an error.

Typically, only one `set-logic` command is issued during the same run of a solver. Additional ones are allowed only if preceded by a `reset` command.

`(set-option` *o v*`)` sets a solver's option *o* to a specific value *v*. More details on predefined options and required behavior are provided in Section 4.1.6. In general, if a solver does not support the setting of a particular option, this command should output `:unsupported` on the regular output channel, and leave the option unchanged from its default value.

Note that the following options may not be set after `set-logic` is invoked without an intervening `reset`: `interactive-mode`, `produce-proofs`, `produce-models`, `produce-assignments`, `produce-unsat-assumptions`, `produce-unsat-cores`.(29)

`(exit)` instructs the solver to exit.

4.2.2 Modifying the assertion stack

`(push` *n*`)` pushes *n* empty assertion levels onto the assertion stack.[2] If *n* is 0, no assertion level are pushed.

`(pop` *n*`)` where *n* is smaller than the number of assertion levels in the stack, pops the top *n* assertion levels from the stack. When *n* is 0, no assertion levels are popped.[3]

`(reset-assertions)` empties the global assertion level and removes from the assertion stack all the other assertion levels.

4.2.3 Introducing new symbols

The next six commands allow one to introduce new functions symbols by providing them with a rank declaration (`declare-sort`, `declare-fun` and `declare-const`) or also with a definition (`define-sort`, `define-fun` and `define-fun-rec`). We will refer to the former as *user-declared* symbols and the latter as *user-defined* symbols.

[2] Typically, $n = 1$.
[3] Note that the global assertion level, which is not created by a `push` command, cannot be popped.

`(declare-sort` s n`)` adds sort symbol s with associated arity n to the current assertion level. It is an error to declare a sort that is already present in the assertion stack.

`(define-sort` s `(`u_1 $\cdots$ u_n`)` τ`)` with $n \geq 0$ adds sort symbol s with associated arity n to the current assertion level. Subsequent well-sortedness checks must treat a sort term like `(`s σ_1 $\cdots$ σ_n`)` as an abbreviation for the term obtained by simultaneously substituting σ_i for u_i, for $i \in \{1, \ldots, n\}$, in τ.[30]

The command reports an error if s is a sort already in the assertion stack or if τ is not a well defined parametric sort with respect to the current signature. Note that this restriction prohibits (meaningless) circular definitions where τ contains s.

`(declare-fun` f `(`σ_1 $\cdots$ σ_n`)` σ`)` with $n \geq 0$ adds adds to the current assertion level a new symbol f with associated rank $\sigma_1 \cdots \sigma_n \sigma$. The command reports an error if a function symbol with name f is already present in the assertion stack.

`(declare-const` f σ`)` abbreviates the command `(declare-fun` f `()` σ`)`.

`(define-fun` f `((`x_1 σ_1`)` $\cdots$ `(`x_n σ_n`))` σ t`)` with $n \geq 0$ and t not containing f is an abbreviation of the command sequence

```
(declare-fun f (σ_1 ··· σ_n) σ)
(assert (forall ((x_1 σ_1) ··· (x_n σ_n)) (= (f x_1 ··· x_n) t)) .
```

Note that the restriction on t prohibits recursive (or mutually recursive) definitions, which are instead provided by `define-fun-rec`. The command reports an error if the argument t is not a well-sorted term of sort σ with respect to the current signature extended with the sort associations $(x_1 : \sigma_1)$, ..., $(x_n : \sigma_n)$.

`(define-fun-rec (`d_1 $\cdots$ d_m`))` where $m > 0$, d_i has the form

$$(f_i \ ((x_{i,1} \ \sigma_{i,1}) \ \cdots \ (x_{i,n_i} \ \sigma_{i,n_i})) \ \sigma_i \ t_i),$$

and $n_i > 0$ for $i = 1, \ldots, m$, abbreviates the command sequence

```
(declare-fun f_1 (σ_{1,1} ··· σ_{1,n_1}) σ_1)
                    ⋮
(declare-fun f_m (σ_{m,1} ··· σ_{m,n_m}) σ_m)
(assert (forall ((x_{1,1} σ_{1,1}) ··· (x_{1,n} σ_{1,n_1})) (= (f_1 x_{1,1} ··· x_{1,n_1}) t_1))
                                  ⋮
(assert (forall ((x_{m,1} σ_{m,1}) ··· (x_{m,n_m} σ_{m,n_m})) (= (f_m x_{m,1} ··· x_{m,n_m}) t_m)) .
```

This command can be used to define functions recursively or, more generally, define groups of functions mutually recursively.[31] Note that the semantics of `define-fun-rec` is exactly the one provided by the assertions above. In particular, there is no requirement that each f_i be terminating (a meaningless notion in our context) or even well-defined.[4] The only requirement is on the well-sortedness of the definitions. The command reports an error if for any $i \in \{1, \ldots, m\}$ t_i is not a well sorted term of sort σ_i with respect to the current signature extended with the sort associations $(x_{i,1} : \sigma_{i,1})$, ..., $(x_{i,n_i} : \sigma_{i,n_i})$, $(f_1 : \sigma_{1,1} \ \cdots \ \sigma_{i,n_i}\sigma_1), \ldots, (f_n : \sigma_{m,1} \ \cdots \ \sigma_{m,n_m}\sigma_m)$.

[4] In fact, it is even possible, although certainly not desirable, to have definitions like `(define-fun-rec ( (f (x Bool) Bool (not (f x))) )` which makes the set of all assertions inconsistent.

4.2.4 Asserting and inspecting formulas

`(assert` t`)` where t is a closed formula (i.e., a well sorted closed term of sort `Bool`), adds t to the current assertion level. The well sortedness requirement is with respect to the current signature.

Instances of this command of the form `(assert (`t `:named` f`))`, where the asserted formula t is given a label f, have the additional effect of adding t to the formulas tracked by the commands `get-assignment` and `get-unsat-core`, as explained later.

`(get-assertions)` causes the solver to print the current set of all asserted formulas as a sequence of the form `(`$f_1 \cdots f_n$`)`. Each f_i is a formula syntactically identical, modulo whitespace, to one of the formulas entered with an `assert` command and currently in the context. Solvers are not allowed to print formulas equivalent to or derived from the asserted formulas.(32)

The command can be issued only if the `:interactive-mode` option, which is set to `false` by default, is set to `true` (see Section 4.1.6).

4.2.5 Checking for satisfiability

`(check-sat)` instructs the solver to check whether the conjunction of all the formulas in the current context is satisfiable in the logic specified by the most recent `:set-logic` command. Conceptually, it asks the solver to search for a model of the logic that satisfies all the currently asserted formulas. When it has finished attempting to do this, the solver should reply on its regular output channel (see Section 4.1.1) using the response format defined by ⟨*check_sat_response*⟩ in Figure 3.8. A `sat` response indicates that the solver has found a model, an `unsat` response that the solver has established there is no model, and an `unknown` response that the search was inconclusive—because of resource limits, solver incompleteness, and so on. If the solver reports `sat` or, optionally, if it reports `unknown`, it should then respond to `get-assignment`, `get-model`, and `get-value` commands provided that the corresponding enabling option is set to `true`. If it reports `unsat`, it must respond to `get-proof` and `get-unsat-core` commands provided that the corresponding enabling option is set to `true`.

A `check-sat` command may be followed by other `assert` and check commands, without intervening `pop` or reset commands. In that case, subsequent `assert` commands just extend the current assertion level (as it existed at the time of the `check-sat` invocation), and later check commands verify the satisfiability of the resulting context.

`(check-sat-assuming (`$a_1 \cdots a_n$`))` where $n > 0$ has the same effect of `check-sat` except that the test is performed on the set of all assertions temporarily extended with the *assumptions* $a_1, \ldots, a_n$. More precisely, it has the same effect of the following command sequence:

```
(push 1)
(assert (and a_1 ··· a_n))
(check-sat)
(pop 1)
```

with the only difference that, when `:produce-unsat-assumptions` is set to `true`, it is possible to issue the command `get-unsat-assumptions` afterwards.

The assumptions $a_1, \ldots, a_n$ can be only user-defined constants (as opposed to arbitrary Boolean terms), specifically, they must be nullary function symbols of sort `Bool` previously introduced with a define command.(33) The command's response is the same as with `check-sat`. Note that after the command executes the current context does not contain the assumptions anymore; as a consequence, it is not meaningful, and so not permitted, to call model inspection commands

such as `get-model` or `get-assignment` immediately after `check-sat-assuming`. It is, however, possible to call `get-unsat-assumptions` if the response was `unsat` and that command is enabled.

4.2.6 Inspecting models

The next three commands can be issued only following a `check-sat` command that reports `sat` or, optionally, also one that reports `unknown`, without intervening commands that modify the assertions set—which includes `check-sat-assuming`. In that case, the solver identifies a model **A** (as defined in Section 5.3) of the current logic and produces responses with respect to that model. The model **A** is required to satisfy the set of all assertions only if the most recent `check-sat` reported `sat`.(34) The internal representation of the model **A** is not exposed by the solver. Similarly to an abstract data type, the model can be inspected only through the three commands below. As a consequence, it can even be partial internally and extended as needed in response to successive invocations of some of these commands.[5]

`(get-value` $t_1 \cdots t_n$`)` where $n > 1$ and each t_i is a well sorted closed quantifier-free term, returns for each t_i a value term v_i[6] that is equivalent to t_i in current model **A** (see above). Specifically, v_i as the same sort as t_i, and t_i is interpreted as v_i in **A**. The values are returned in a sequence of pairs of the form `((`t_1 v_1`)` $\cdots$ `(`t_n v_n`))`. The terms $v_1, \ldots, v_n$ are allowed to contain symbols not in the current signature only if they are abstract values, i.e., constant symbols starting with the special character `@`.(35) Since these are solver-defined, their sort is not known to the user. Therefore, additionally, each occurrence of an abstract value a of sort σ in $v_1, \ldots, v_n$ has to be contained in a term of the form `(as` a σ`)` which makes the sort explicit.

Note that the returned abstract values are used only to express information about the current model **A**. They cannot be used in later `assert` commands since they are neither theory symbols nor user-defined ones.

There is no requirement that different permutations of the same set of `get-value` calls produce the same value for the input terms. The only requirement is that any two syntactically different values of the same sort returned by the solver have different meaning in the model.[7]

The command can be issued only if the `:produce-models` option, which is set to `false` by default, is set to `true` (see Section 4.1.6).

`(get-assignment)` can be seen as a light-weight and restricted version of `get-value` that asks for a truth assignment for a selected set of previously entered formulas.(36)

The command returns a sequence of the form `((`f_1 b_1`)` $\cdots$ `(`f_n b_n`))` with $n \geq 0$. A pair `(`f_i b_i`)` is in the returned sequence if and only if f_i is the label of a (sub)term of the form `(`t_i `:named` f_i`)` in the context, with t_i a closed term of sort `Bool`, and b_i is the value (`true` or `false`) that t_i has in the current model **A**.

The command can be issued only if the `:produce-assignments` option, which is set to `false` by default, is set to `true` (see Section 4.1.6).

[5] In that case, of course, the solver has to be sure that its partial model can be indeed extended as needed.

[6] Recall that value terms are particular ground terms defined in a logic for each sort (see Subsection 5.5.1).

[7] So, for instance, in a logic of rational numbers and values of the form `(/` m n`)` and `(/ (-` m`)` n`)` with m, n numerals, the solver cannot use both the terms `(/ 1 3)` and `(/ 2 6)` as output values for `get-value`.

`(get-model)` returns a list $(d_1 \cdots d_k)$ where $\{g_1, \ldots, g_k\}$ is the set of *all* the current user-declared function symbols, and each d_i is an expression denoting the interpretation of g_i in the current model $\mathbf{A}$. For each $f \in \{g_1, \ldots, g_k\}$ this interpretation is provided as a `define-fun` or `define-fun-rec` command having one of the following forms:(37)

- $(\texttt{define-fun}\ f\ ((x_1\ \sigma_1) \cdots (x_n\ \sigma_n))\ \sigma\ t)$
  where $n \geq 0$, t is a term not containing f, and the formula
  $$(\texttt{forall}\ ((x_1\ \sigma_1) \cdots (x_n\ \sigma_n))\ (=\ (f\ x_1 \cdots x_n)\ t))$$
  is well-sorted and satisfied by $\mathbf{A}$. The term t is expected, although not required, to be a value when f is a constant (i.e., when $n = 0$).
- $(\texttt{define-fun-rec}\ (\ (f_1\ ((x_{1,1}\ \sigma_{1,1}) \cdots (x_{1,n_1}\ \sigma_{1,n_1}))\ \sigma_1\ t_1) \cdots (f_m\ ((x_{m,1}\ \sigma_{m,1}) \cdots (x_{m,n_m}\ \sigma_{m,n_m}))\ \sigma_m\ t_m)\ ))$
  where $f \in \{f_1, \ldots, f_m\}$, $n_i > 0$ for $i = 1, \ldots, m$, and the formula
  $$\begin{array}{l}(\texttt{and}\ (\texttt{forall}\ ((x_{1,1}\ \sigma_{1,1}) \cdots (x_{1,n}\ \sigma_{1,n_1}))\\ \qquad (=\ (f_1\ x_{1,1} \cdots x_{1,n_1})\ t_1))\\ \qquad\qquad \vdots \\ \qquad (\texttt{forall}\ ((x_{m,1}\ \sigma_{m,1}) \cdots (x_{m,n_m}\ \sigma_{m,n_m}))\\ \qquad\quad (=\ (f_m\ x_{m,1} \cdots x_{m,n_m})\ t_m)))\end{array}$$
  is well-sorted and satisfied by $\mathbf{A}$.

Similarly to the response of `get-value`, the terms $t, t_1, \ldots, t_m$ above are allowed to contain symbols not in the current signature only if they are abstract values. Morevever, each occurrence of an abstract value a of sort σ in $t, t_1, \ldots, t_m$ has to be contained in a term of the form $(\texttt{as}\ a\ \sigma)$.

Later versions of the standard may impose stronger requirements on the returned definitions. For now there is only an expectation that, when possible, the solver will provide definitions that have a unique interpretation over the current signature.(38)

The command can be issued only if the `:produce-models` option, which is set to `false` by default, is set to `true` (see Section 4.1.6).

4.2.7 Inspecting proofs

The command `get-unsat-assumptions` below can be issued only following an invocation of `check-sat-assuming` that reports `unsat`, without any intervening commands other than other instances of `get-unsat-assumptions`. The other two commands can be issued only following a `check-sat` command that reports `unsat`, without intervening commands that modify the assertions set (including `check-sat-assuming`).

`(get-unsat-assumptions)` returns a subset of the assumptions from a previous `check-sat-assuming` command. More precisely, it returns a sequence $(a_1 \cdots a_n)$ where $a_1, \ldots, a_n$ are among the assumptions in the most recent `check-sat-assuming` command and are such that issuing the command $(\texttt{check-sat-assuming}\ (a_1 \cdots a_n))$ instead would have still produced an `unsat` answer. The returned sequence is not required to be minimal.(39)

The command can be issued only if the `:produce-unsat-assumptions` option, which is set to `false` by default, is set to `true` (see Section 4.1.6).

`(get-proof)` asks the solver for a proof of unsatisfiability for the set of all formulas in the current context. The solver responds by printing a refutation proof on its regular output channel. The format of the proof is solver-specific.(40) The only requirement is that, like all responses, it be a member of ⟨*s_expr*⟩.

The command can be issued only if the `:produce-proofs` option, which is set to `false` by default, is set to `true` (see Section 4.1.6).

`(get-unsat-core)` asks the solver to identify an *unsatisfiable core*, a subset of all the formulas in the current context that is unsatisfiable by itself. The solver selects from the unsatisfiable core only those formulas that have been asserted with a command of the form `(assert (`t `:named` f`))`, and returns a sequence `(`f_1 $\cdots$ f_n`)` of those labels. Unlabeled formulas in the unsatisfiable core are simply not reported.(41)

The semantics of this command's output is that the reported assertions *together with all* the unlabeled ones in the set of all assertions are jointly unsatisfiable. In practice then, not labeling assertions is useful for unsat core detection purposes only when the user is sure that the set of all unlabeled assertions is satisfiable.

The command can be issued only if the `:produce-unsat-cores` option, which is set to `false` by default, is set to `true` (see Section 4.1.6).

4.2.8 Inspecting settings

`(get-info` f`)` where f is an element of ⟨*info_flag*⟩ returns solver information as specified in Section 4.1.7.

`(get-option` o`)` returns the current value of a solver's option o as element of ⟨*attribute_value*⟩. The form of that value depends on the specific option. More details on predefined options and required behavior are provided in Section 4.1.6. As with `set-option`, if a solver does not support the setting of o, this command should output `:unsupported` on the regular output channel, and leave the option unchanged from its default value.

4.2.9 Script information

`(meta-info` a`)` where a is an element of ⟨*attribute*⟩ has no effect. Its only purpose is to allow the insertion of structured meta information into a script.(42) Typically then, a solver will just parse the command and do nothing with it except for printing a general response (`success` or an error if the argument is not an element of ⟨*attribute*⟩).

`(set-info` a`)` is the same as `(meta-info` a`)`. `set-info` is the name that `meta-info` had in Version 2.0. Its use is now deprecated.(43)

`(echo` s`)` where s is a string literal, simply prints back s as is—including the surrounding double-quotes.(44)

Chapter 5

Logical Semantics of SMT-LIB Formulas

The underlying logic of the SMT-LIB language is a variant of many-sorted first-order logic (FOL) with equality [Man93, Gal86, End01], although it incorporates some features of higher-order logics; in particular, the identification of formulas with terms of a distinguished Boolean sort, and the use of sort symbols of arity greater than 0.

These features make for a more flexible and syntactically more uniform logical language. However, while not exactly syntactic sugar, they do not change the essence of SMT-LIB logic with respect to traditional many-sorted FOL. Quantifiers are still first-order, the sort structure is flat (no subsorts), the logic's type system has no function (*arrow*) types, no type quantifiers, no dependent types, no provisions for parametric or subsort polymorphism. The only polymorphism is of the *ad-hoc* variety (a function symbol can be given more than one rank), although there is a syntactical mechanism for approximating parametric polymorphism. As a consequence, all the classical meta-theoretic results from many-sorted FOL apply to SMT-LIB logic as well.

To define SMT-LIB logic and its semantics it is convenient to work with a more abstract syntax than the concrete S-expression-based syntax of the SMT-LIB language. The formal semantics of concrete SMT-LIB expressions is then given by means of a translation into this abstract syntax. A formal definition of this translation might be provided in later releases of this document. For the time being, we will appeal to the reader's intuition and on the fact that the translation is defined as one would expect.

The translation also maps concrete predefined symbols and keywords to their abstract counterpart. To facilitate reading, usually the abstract version of a predefined concrete symbol is denoted by the symbol's name in Roman bold font (e.g., **Bool** for `Bool`). The same is done for keywords (e.g., **definition** for `:definition`).

To define our target abstract syntax we start by fixing the following sets of (abstract) symbols and values:

- an infinite set $\mathcal{S}$ of *sort symbols* s containing the symbol **Bool**,
- an infinite set $\mathcal{U}$ of *sort parameters* u,
- an infinite set $\mathcal{X}$ of *variables* x,
- an infinite set $\mathcal{F}$ of *function symbols* f containing the symbols $\approx$, $\wedge$, and $\neg$,

(Sorts)	σ	::=	$s\,\sigma^*$
(Parametric Sorts)	τ	::=	$u \mid s\,\tau^*$

Figure 5.1: Abstract syntax for sort terms

- an infinite set $\mathcal{A}$ of *attribute names* a,
- an infinite set $\mathcal{V}$ of *attribute values* v.
- the set $\mathcal{W}$ of *ASCII character strings* w.
- a two-element set $\mathcal{B} = \{\mathbf{true}, \mathbf{false}\}$ of *Boolean values* b,
- the set $\mathcal{N}$ of *natural numbers* n,
- an infinite set $\mathcal{TN}$ of *theory names* T,
- an infinite set $\mathcal{L}$ of *logic names* L.

It is unnecessary to require that the sets above be pairwise disjoint.

5.1 The language of sorts

In many-sorted logics, terms are typed, or *sorted*, and each sort is denoted by a sort symbol. In SMT-LIB logic, the language of sorts is extended from sort symbols to *sort terms* built with symbols from the set $\mathcal{S}$ above. Formally, we have the following.

Definition 1 (Sorts)**.** For all non-empty subsets S of $\mathcal{S}$ and all mappings $ar : S \to \mathbb{N}$, the set $Sort(S)$ of all *sorts* over S (with respect to ar) is defined inductively as follows:

1. every $s \in S$ with $ar(s) = 0$ is a sort;
2. If $s \in S$ and $ar(s) = n > 0$ and $\sigma_1, \ldots, \sigma_n$ are sorts, then the term $s\,\sigma_1 \cdots \sigma_n$ is a sort.

We say that $s \in S$ has (or is of) *arity* n if $ar(s) = n$. □

As an example of a sort, if **Int** and **Real** are sort symbols of arity 0, and **List** and **Array** are sort symbols of respective arity 1 and 2, then the expression **List** (**Array Int** (**List Real**)) and all of its subexpressions are sorts.

Function symbol declarations in theory declarations (defined later), use also *parametric sorts*. These are defined similarly to sorts above except that they can be built also over a further set $\mathcal{U}$ of *sort parameters*, used like sort symbols of arity 0. Similarly to the example above, if u_1, u_2 are elements of $\mathcal{U}$, the expression **List** (**Array** u_1 (**List** u_2)) and all of its subexpressions are parametric sorts.

An abstract syntax for sorts σ and parametric sorts τ, which ignores arity constraints for simplicity, is provided in Figure 5.1. Note that every sort is a parametric sort, but not vice versa. Also note that parametric sorts are used only in theory declarations; they are not part of SMT-LIB logic. In the following, we say "sort" to refer exclusively to non-parametric sorts.

$$
\begin{array}{llll}
\text{(Attributes)} & \alpha & ::= & a \mid a = v \\
\text{(Terms)} & t & ::= & x \mid f\ t^* \mid f^\sigma\ t^* \\
 & & \mid & \exists\, (x{:}\sigma)^+\ t \mid \forall\, (x{:}\sigma)^+\ t \mid \textbf{let}\ (x = t)^+\ \textbf{in}\ t \\
 & & \mid & t\ \alpha^+
\end{array}
$$

Figure 5.2: Abstract syntax for unsorted terms

5.2 The language of terms

In the abstract syntax, terms are built out of variables from $\mathcal{X}$, function symbols from $\mathcal{F}$, and a set of *binders*. The logic considers, in fact, only *well-sorted terms*, a subset of all possible terms determined by a *sorted signature*, as described below.

The set of all terms is defined by the abstract syntax rules of Figure 5.2. The rules do not distinguish between constant and function symbols (they are all members of the set $\mathcal{F}$). These distinctions are really a matter of arity, which is taken care of later by the well-sortedness rules.

For all $n \geq 0$, variables $x_1, \ldots, x_n \in \mathcal{X}$ and sorts $\sigma_1, \ldots, \sigma_n$,

- the prefix construct $\exists\, x_1{:}\sigma_1 \cdots x_n{:}\sigma_n$ _ is a *sorted existential binder (or existential quantifier)* for $x_1, \ldots, x_n$;
- the prefix construct $\forall\, x_1{:}\sigma_1 \cdots x_n{:}\sigma_n$ _ is a *sorted universal binder (or universal quantifier)* for $x_1, \ldots, x_n$;
- the mixfix construct $\textbf{let}\, x_1 = \_ \cdots x_n = \_\ \textbf{in}\ \_$ is an *(parallel-)let binder* for $x_1, \ldots, x_n$.

We speak of *bound* or *free* (occurrences of) variables in a term as usual. Terms are *closed* if they contain no free variables, and *open* otherwise. Terms are *ground* if they are variable-free.

For simplicity, the defined language does not contain any logical symbols other than quantifiers. Logical connectives for negation, conjunction and so on and the equality symbol, which we denote here by $\approx$, are just function symbols of the basic theory **Core**, implicitly included in all SMT-LIB theories (see Subsection 3.7.1).

Terms can be optionally annotated with zero or more *attributes*. Attributes have no logical meaning, but they are a convenient mechanism for adding meta-logical information, as illustrated in Section 3.6. Syntactically, an attribute is either an attribute name $a \in \mathcal{A}$ or a pair the form $a = v$ where $a \in \mathcal{A}$ and v is an attribute value in $\mathcal{V}$.[1]

Function symbols themselves may be annotated with a sort, as in f^σ. Sort annotations simplify the sorting rules of the logic, which determine the set of well-sorted terms.

5.2.1 Signatures

Well-sorted terms in SMT-LIB logic are terms that can be associated with a unique sort by means of a set of *sorting rules* similar to typing rules in programming languages. The rules are based on the following definition of a (many-sorted) signature.

Definition 2 (SMT-LIB Signature)**.** An *SMT-LIB signature*, or simply a *signature*, is a tuple Σ consisting of:

[1] At this abstract level, the syntax of attribute values is intentionally left unspecified.

- a set $\Sigma^{\mathrm{S}} \subseteq \mathcal{S}$ of sort symbols containing **Bool**,
- a set $\Sigma^{\mathrm{F}} \subseteq \mathcal{F}$ of function symbols containing $\approx$, $\wedge$, and $\neg$,
- a total mapping ar from Σ^{S} to $\mathbb{N}$, with $ar(\mathbf{Bool}) = 0$,
- a partial mapping from the variables $\mathcal{X}$ to $Sort(\Sigma) := Sort(\Sigma^{\mathrm{S}})$,[2]
- a left-total relation[3] R from Σ^{F} to $Sort(\Sigma)^+$ such that
  - $(\neg, \mathbf{Bool}\,\mathbf{Bool}) \in R$, $(\wedge, \mathbf{Bool}\,\mathbf{Bool}\,\mathbf{Bool}) \in R$, and
  - $(\approx, \sigma\sigma\,\mathbf{Bool}) \in R$ for all $\sigma \in Sort(\Sigma)$.

Each sort sequence associated by Σ to a function symbol f is a *rank* of f. □

The rank of a function symbol specifies, in order, the expected sort of the symbol's arguments and result. It is possible for a function symbol to be *overloaded* in a signature for being associated to more than one rank in that signature.

This form of *ad-hoc polymorphism* is entirely unrestricted: a function symbol can have completely different ranks—even varying in arity. For example, in a signature with sorts **Int** and **Real** (with the expected meaning), it is possible for the minus symbol $-$ to have all of the following ranks: **Real Real** (for unary negation over the reals), **Int Int** (for unary negation over the integers), **Real Real Real** (for binary subtraction over the reals), and **Int Int Int** (for binary subtraction over the integers).

Together with the mechanism used to declare theories (described in the next section), overloading also provides an approximate form of *parametric polymorphism* by allowing one to declare function symbols with ranks all having the *same shape*. For instance, it is possible to declare an array access symbol with rank $(\mathbf{Array}\,\sigma_1\,\sigma_2)\,\sigma_1\,\sigma_2$ for all sorts σ_1, σ_2 in a theory signature. Strictly speaking, this is still ad-hoc polymorphism because SMT-LIB logic itself does not allow parametric sorts.[4] However, it provides most of the convenience of parametric polymorphism while remaining within the confines of the standard semantics of many-sorted FOL.

A function symbol can be *ambiguous* in an SMT-LIB signature for having distinct ranks of the form $\bar{\sigma}\sigma_1$ and $\bar{\sigma}\sigma_2$ with σ_1 and σ_2 different sorts. Thanks to the requirement in Definition 2 that variables have at most one sort in a signature, in signatures with no ambiguous function symbols every term can have at most one sort. In contrast, with an ambiguous symbol f whose different ranks are $\bar{\sigma}\sigma_1, \ldots, \bar{\sigma}\sigma_n$, a term of the form $f\,\bar{t}$, where the terms $\bar{t}$ have sorts $\bar{\sigma}$, can be given a unique sort only if f is annotated with one of the result sorts $\sigma_1, \ldots, \sigma_n$, that is, only if it is written as $f^{\sigma_i}\,\bar{t}$ for some $i \in \{i, \ldots, n\}$.

In the following, we will work with *ranked* function symbols and *sorted* variables in a signature. Formally, given a signature Σ, a *ranked function symbol* is a pair $(f, \sigma_1 \cdots \sigma_n\sigma)$ in $\mathcal{F} \times Sort(\Sigma)^+$, which we write as $f{:}\sigma_1 \cdots \sigma_n\sigma$. A *sorted variable* is a pair (x, σ) in $\mathcal{X} \times Sort(\Sigma)$, which we write as $x{:}\sigma$. We write $f{:}\sigma_1 \cdots \sigma_n\sigma \in \Sigma$ and $x{:}\sigma \in \Sigma$ to denote that f has rank $\sigma_1 \cdots \sigma_n\sigma$ in Σ and x has sort σ in Σ.

A signature Σ' is *variant* of a signature Σ if it is identical to Σ possibly except for its mapping from variables to sorts. We will also consider signatures that conservatively expand a given signature with additional sort and function symbols or additional ranks for Σ's function symbols. A signature

[2] Note that $Sort(\Sigma)$ is non-empty because at least one sort in Σ^{S}, **Bool**, has arity 0. Also, recall that if S is a set of sort symbols (like Σ^{S}), then $Sort(S)$ is the set of all sorts over S.

[3] A binary relation $R \subseteq X \times Y$ is *left-total* if for each $x \in X$ there is (at least) a $y \in Y$ such that xRy.

[4] Parametric sort terms that occur in theory declarations are meta-level syntax as far as SMT-LIB logic is concerned. They are *schemas* standing for concrete sorts.

$$\frac{}{\Sigma \vdash x\ \alpha^* : \sigma} \quad \text{if } x{:}\sigma \in \Sigma$$

$$\frac{\Sigma \vdash t_1 : \sigma_1 \quad \cdots \quad \Sigma \vdash t_k : \sigma_k}{\Sigma \vdash (f\ t_1\ \cdots\ t_k)\ \alpha^* : \sigma} \quad \text{if } \begin{cases} f{:}\sigma_1 \cdots \sigma_k\sigma \in \Sigma & \text{and} \\ f{:}\sigma_1 \cdots \sigma_k\sigma' \notin \Sigma & \text{for all } \sigma' \neq \sigma \end{cases}$$

$$\frac{\Sigma \vdash t_1 : \sigma_1 \quad \cdots \quad \Sigma \vdash t_k : \sigma_k}{\Sigma \vdash (f^{\sigma}\ t_1\ \cdots\ t_k)\ \alpha^* : \sigma} \quad \text{if } \begin{cases} f{:}\sigma_1 \cdots \sigma_k\sigma \in \Sigma & \text{and} \\ f{:}\sigma_1 \cdots \sigma_k\sigma' \in \Sigma & \text{for some } \sigma' \neq \sigma \end{cases}$$

$$\frac{\Sigma[x_1{:}\sigma_1,\ \ldots,\ x_{k+1}{:}\sigma_{k+1}] \vdash t : \mathbf{Bool}}{\Sigma \vdash (Qx_1{:}\sigma_1\ \cdots\ x_{k+1}{:}\sigma_{k+1}\ t)\ \alpha^* : \mathbf{Bool}} \quad \text{if } Q \in \{\exists, \forall\}$$

$$\frac{\Sigma \vdash t_1 : \sigma_1 \quad \cdots \quad \Sigma \vdash t_{k+1} : \sigma_{k+1} \quad \Sigma[x_1{:}\sigma_1,\ \ldots,\ x_{k+1}{:}\sigma_{k+1}] \vdash t : \sigma}{\Sigma \vdash (\mathbf{let}\ x_1 = t_1\ \cdots\ x_{k+1} = t_{k+1}\ \mathbf{in}\ t)\ \alpha^* : \sigma}$$

Figure 5.3: Well-sortedness rules for terms

Ω is a *expansion* of a signature Σ if all of the following hold: $\Sigma^{\mathrm{S}} \subseteq \Omega^{\mathrm{S}}$; $\Sigma^{\mathrm{F}} \subseteq \Omega^{\mathrm{F}}$; the sort symbols of Σ have the same arity in Σ and in Ω; for all $x \in \mathcal{X}$ and $\sigma \in \mathit{Sort}(\Sigma)$, $x{:}\sigma \in \Sigma$ iff $x{:}\sigma \in \Omega$; for all $f \in \mathcal{F}$ and $\bar{\sigma} \in \mathit{Sort}(\Sigma)^+$, if $f{:}\bar{\sigma} \in \Sigma$ then $f{:}\bar{\sigma} \in \Omega$. In that case, Σ is a *subsignature* of Ω.

5.2.2 Well-sorted terms

Figure 5.3 provides a set of rules defining well-sorted terms with respect to an SMT-LIB signature Σ. Strictly speaking then, and similarly to more conventional logics, the SMT-LIB logic language is a family of languages parametrized by the signature Σ. As explained later, for each script working in the context of a background theory $\mathcal{T}$, the specific signature is jointly definedby the declaration of $\mathcal{T}$ plus any additional sort and function symbol declarations contained in the script.

The format and meaning of the sorting rules in Figure 5.3 is fairly standard and should be largely self-explanatory to readers familiar with type systems. In more detail, the letter σ (possibly primed or with subscripts) denotes sorts in $\mathit{Sort}(\Sigma)$, the integer index k in the rules is assumed ≥ 0. The expression $\Sigma[x_1 : \sigma_1, \ldots, x_{k+1} : \sigma_{k+1}]$ denotes the signature that maps x_i to sort σ_i for $i = 1, \ldots, k+1$, and coincides otherwise with Σ. Finally, α^* denotes a possibly empty sequence of attributes. The rules operate over *sorting judgments* which are triples of the form $\Sigma \vdash t : \sigma$.

Definition 3 (Well-sorted Terms)**.** For every SMT-LIB signature Σ, a term t generated by the grammar in Figure 5.2 is *well-sorted (with respect to Σ)* if $\Sigma \vdash t : \sigma$ is derivable by the sorting rules in Figure 5.3 for some sort $\sigma \in \mathit{Sort}(\Sigma)$. In that case, we say that *t has, or is of, sort σ*. □

With this definition, it is possible to show that every term has at most one sort.(45)

Definition 4 (SMT-LIB formulas)**.** For each signature Σ, the language of SMT-LIB logic is the set of all well-sorted terms wrt Σ. *Formulas* are well-sorted terms of sort **Bool**. □

In the following, we will sometimes use φ and ψ to denote formulas.

Constraint 3. SMT-LIB scripts consider only closed formulas, or *sentences*, closed terms of sort **Bool**.(46) □

There is no loss of generality in the restriction above because, as far as satisfiability is concerned, every formula φ with free variables $x_1, \ldots, x_n$ of respective sort $\sigma_1, \ldots, \sigma_n$, can be rewritten as

$$\exists\, x_1{:}\sigma_1 \ldots x_n{:}\sigma_n\ \varphi\ .$$

An alternative way to avoid free variables in scripts is to replace them by fresh constant symbols of the same sort. This is again with no loss of generality because, for satisfiability modulo theories purposes, a formula's free variables can be treated equivalently as *free symbols* (see later for a definition).

5.3 Structures and Satisfiability

The semantics of SMT-LIB is essentially the same as that of conventional many-sorted logic, relying on a similar notion of Σ-*structure*.

Definition 5 (Σ-structure)**.** Let Σ be a signature. A Σ-*structure* $\mathbf{A}$ is a pair consisting of a set A, the *universe* of $\mathbf{A}$, that includes the two-element set $\mathcal{B} = \{\mathbf{true}, \mathbf{false}\}$, and a mapping that interprets

- each $\sigma \in \mathit{Sort}(\Sigma)$ as subset $\sigma^{\mathbf{A}}$ of A, with $\mathbf{Bool}^{\mathbf{A}} = \mathcal{B}$,
- each (ranked function symbol) $f{:}\sigma \in \Sigma$ as an element $(f{:}\sigma)^{\mathbf{A}}$ of $\sigma^{\mathbf{A}}$,
- each $f{:}\sigma_1 \cdots \sigma_n\sigma \in \Sigma$ with $n > 0$ as a total function $(f{:}\sigma_1 \cdots \sigma_n\sigma)^{\mathbf{A}}$ from $\sigma_1^{\mathbf{A}} \times \cdots \times \sigma_n^{\mathbf{A}}$ to $\sigma^{\mathbf{A}}$, with $\approx{:}\sigma\sigma\,\mathbf{Bool}$ interpreted as the identity predicate over $\sigma^{\mathbf{A}}$[5].

For each $\sigma \in \mathit{Sort}(\Sigma)$, the set $\sigma^{\mathbf{A}}$ is called the *extension* of σ in $\mathbf{A}$.(47) □

Note that, as a consequence of overloading, a Σ-structure does not interpret plain function symbols but ranked function symbols. Also note that any Σ-structure is also a Σ'-structure for every variant Σ' of Σ.

If $\mathbf{B}$ is an Ω-signature with universe B and Σ is a subsignature of Ω, the *reduct* of $\mathbf{B}$ to Σ is the (unique) Σ-structure with universe B that interprets its sort and function symbols exactly as $\mathbf{B}$.

5.3.1 The meaning of terms

A *valuation* into a Σ-structure $\mathbf{A}$ is a partial mapping v from $\mathcal{X} \times \mathit{Sort}(\Sigma)$ to the set of all domain elements of $\mathbf{A}$ such that, for all $x \in \mathcal{X}$ and $\sigma \in \mathit{Sort}(\Sigma)$, $v(x{:}\sigma) \in \sigma^{\mathbf{A}}$. We denote by $v[x_1{:}\sigma_1 \mapsto a_1,\ \ldots,\ x_n{:}\sigma_n \mapsto a_n]$ the valuation that maps $x_i{:}\sigma_i$ to $a_i \in \sigma_i^{\mathbf{A}}$ for $i = 1, \ldots, n$ and is otherwise identical to v.

If v is a valuation into Σ-structure $\mathbf{A}$, the pair $\mathcal{I} = (\mathbf{A}, v)$ is a Σ-*interpretation*. We write $\mathcal{I}[x_1{:}\sigma_1 \mapsto a_1,\ \ldots,\ x_n{:}\sigma_n \mapsto a_n]$ as an abbreviation for the Σ'-interpretation

$$(\mathbf{A}', v[x_1{:}\sigma_1 \mapsto a_1,\ \ldots,\ x_n{:}\sigma_n \mapsto a_n])$$

where $\Sigma' = \Sigma[x_1{:}\sigma_1,\ \ldots,\ x_n{:}\sigma_n]$ and $\mathbf{A}'$ is just $\mathbf{A}$ but seen as a Σ'-structure.

A Σ-interpretation $\mathcal{I}$ assigns a meaning to well-sorted Σ-terms by means of a uniquely determined (total) mapping $[\![\_]\!]^{\mathcal{I}}$ of such terms into the universe of its structure.

[5] That is, for all $\sigma \in \mathit{Sort}(\Sigma)$ and all $a, b \in \sigma^{\mathbf{A}}$, $\approx^{\mathbf{A}}(a, b) = \mathbf{true}$ iff $a = b$.

Definition 6. Let Σ be an SMT-LIB signature and let $\mathcal{I} = (\mathbf{A}, v)$ be a Σ-interpretation. For every well-sorted term t of sort σ with respect to Σ, $[\![t]\!]^{\mathcal{I}}$ is defined inductively as follows.

1. $[\![x]\!]^{\mathcal{I}} = v(x{:}\sigma)$
2. $[\![\hat{f}\ t_1\ \ldots\ t_n]\!]^{\mathcal{I}} = (f{:}\sigma_1 \cdots \sigma_n\sigma)^{\mathbf{A}}(a_1, \ldots, a_n)$ if $\begin{cases} \hat{f} = f \text{ or } \hat{f} = f^{\sigma}, \\ \Omega \text{ is the signature of } \mathcal{I}, \\ \text{for } i = 1, \ldots, n \\ \quad \Omega \vdash t_i : \sigma_i \text{ and } a_i = [\![t_i]\!]^{\mathcal{I}} \end{cases}$
3. $[\![\textbf{let } x_1 = t_1 \ \cdots\ x_n = t_n \textbf{ in } t]\!]^{\mathcal{I}} = [\![t]\!]^{\mathcal{I}'}$ if $\begin{cases} \Omega \text{ is the signature of } \mathcal{I}, \\ \text{for } i = 1, \ldots, n \\ \quad \Omega \vdash t_i : \sigma_i \text{ and } a_i = [\![t_i]\!]^{\mathcal{I}}, \\ \mathcal{I}' = \mathcal{I}[x_1{:}\sigma_1 \mapsto a_1,\ \ldots,\ x_n{:}\sigma_1 \mapsto a_n] \end{cases}$
4. $[\![\exists\, x_1{:}\sigma_1 \cdots x_n{:}\sigma_n\ t]\!]^{\mathcal{I}} = \textbf{true}$
   iff $[\![t]\!]^{\mathcal{I}'} = \textbf{true}$ for some $\begin{cases} (a_1, \ldots, a_n) \in \sigma_1^{\mathbf{A}} \times \cdots \times \sigma_n^{\mathbf{A}}, \\ \mathcal{I}' = \mathcal{I}[x_1{:}\sigma_1 \mapsto a_1,\ \ldots,\ x_n{:}\sigma_1 \mapsto a_n] \end{cases}$
5. $[\![\forall\, x_1{:}\sigma_1 \cdots x_n{:}\sigma_n\ t]\!]^{\mathcal{I}} = \textbf{true}$
   iff $[\![t]\!]^{\mathcal{I}'} = \textbf{true}$ for all $\begin{cases} (a_1, \ldots, a_n) \in \sigma_1^{\mathbf{A}} \times \cdots \times \sigma_n^{\mathbf{A}}, \\ \mathcal{I}' = \mathcal{I}[x_1{:}\sigma_1 \mapsto a_1,\ \ldots,\ x_n{:}\sigma_1 \mapsto a_n] \end{cases}$
6. $[\![u\ \alpha_1\ \cdots\ \alpha_n]\!]^{\mathcal{I}} = [\![u]\!]^{\mathcal{I}}$. □

One can show that $[\![\_]\!]^{\mathcal{I}}$ is well-defined and indeed total over terms that are well-sorted with respect to $\mathcal{I}$'s signature.

A Σ-interpretation $\mathcal{I}$ *satisfies* a Σ-formula φ if $[\![\varphi]\!]^{\mathcal{I}} = \textbf{true}$, and *falsifies* it otherwise. The formula φ is *satisfiable* if there is a Σ-interpretation $\mathcal{I}$ that satisfies it, and is *unsatisfiable* otherwise.

For a closed term t, its meaning $[\![t]\!]^{\mathcal{I}}$ in an interpretation $\mathcal{I} = (\mathbf{A}, v)$ is independent of the choice of the valuation v—because the term has no free variables. For such terms then, we can write $[\![t]\!]^{\mathbf{A}}$ instead of $[\![t]\!]^{\mathcal{I}}$. Similarly, for sentences, we can speak directly of a *structure* satisfying or falsifying the sentence. A Σ-structure that satisfies a sentence is also called a *model* of the sentence.

The notion of isomorphism introduced below is needed for Definition 9, Theory Combination, in the next section.

Definition 7 (Isomorphism)**.** Let $\mathbf{A}$ and $\mathbf{B}$ be two Σ-structures with respective universes A and B. A mapping $h : A \to B$ is an *homomorphism* from $\mathbf{A}$ to $\mathbf{B}$ if

1. for all $\sigma \in \mathit{Sort}(\Sigma)$ and $a \in \sigma^{\mathbf{A}}$,
$$h(a) \in \sigma^{\mathbf{B}}\ ;$$
2. for all $f{:}\sigma_1 \ldots \sigma_n\sigma \in \Sigma$ with $n > 0$ and $(a_1, \ldots, a_n) \in \sigma_1^{\mathbf{A}} \times \cdots \times \sigma_n^{\mathbf{A}}$,
$$h((f{:}\sigma_1 \ldots \sigma_n\sigma)^{\mathbf{A}}(a_1, \ldots, a_n)) = (f{:}\sigma_1 \ldots \sigma_n\sigma)^{\mathbf{B}}(h(a_1), \ldots, h(a_n))\ .$$

A homomorphism between $\mathbf{A}$ and $\mathbf{B}$ is an *isomorphism* of $\mathbf{A}$ onto $\mathbf{B}$ if it is invertible and its inverse is a homomorphism from $\mathbf{B}$ to $\mathbf{A}$. □

Two Σ-structures $\mathbf{A}$ and $\mathbf{B}$ are *isomorphic* if there is an isomorphism from one onto the other. Isomorphic structures are interchangeable for satisfiability purposes because one satisfies a Σ-sentence if and only the other one does.

5.4 Theories

Theories are traditionally defined as sets of sentences. Alternatively, and more generally, in SMT-LIB a theory is defined as a class of structures with the same signature.

Definition 8 (Theory)**.** For any signature Σ, a *Σ-theory* is a class of Σ-structures. Each of these structures is a *model* of the theory.

Typical SMT-LIB theories consist of a single model (e.g., the integers) or of the class of all structures that satisfy some set of sentences—the *axioms* of the theory. Note that in SMT-LIB there is no requirement that the axiom set be finite or even recursive.

SMT-LIB uses both *basic* theories, obtained as instances of a theory declaration schema, and *combined* theories, obtained by combining together suitable instances of different theory schemas. The combination mechanism is defined below.

Two signatures Σ_1 and Σ_2 are *compatible* if they have exactly the same sort symbols and agree both on the arity they assign to sort symbols and on the sorts they assign to variables.[6] Two theories are *compatible* if they have compatible signatures. The *combination* $\Sigma_1 + \Sigma_2$ of two compatible signatures Σ_1 and Σ_2 is the smallest compatible signature that is an expansion of both Σ_1 and Σ_2, i.e., the unique signature Σ compatible with Σ_1 and Σ_2 such that, for all $f \in \mathcal{F}$ and $\bar{\sigma} \in \mathit{Sort}(\Sigma)^+$, $f{:}\bar{\sigma} \in \Sigma$ iff $f{:}\bar{\sigma} \in \Sigma_1$ or $f{:}\bar{\sigma} \in \Sigma_2$.

Definition 9 (Theory Combination)**.** Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two theories with compatible signatures Σ_1 and Σ_2, respectively. The *combination* $\mathcal{T}_1 + \mathcal{T}_2$ of $\mathcal{T}_1$ and $\mathcal{T}_2$ consists of all $(\Sigma_1 + \Sigma_2)$-structures whose reduct to Σ_i is isomorphic to a model of $\mathcal{T}_i$, for $i = 1, 2$. □

Over pairwise compatible signatures the signature combination operation $+$ is associative and commutative. The same is also true for the theory combination operation $+$ over compatible theories. This induces, for every $n > 1$, a unique n-ary combination $\mathcal{T}_1 + \cdots + \mathcal{T}_n$ of mutually compatible theories $\mathcal{T}_1, \ldots, \mathcal{T}_n$ in terms of nested binary combinations. *Combined* theories in SMT-LIB are exclusively theories of the form $\mathcal{T}_1 + \cdots + \mathcal{T}_n$ for some basic SMT-LIB theories $\mathcal{T}_1, \ldots, \mathcal{T}_n$.

SMT is about checking the satisfiability or the entailment of formulas *modulo* some (possibly combined) theory $\mathcal{T}$. This standard adopts the following precise formulation of such notions.

Definition 10 (Satisfiability and Entailment Modulo a Theory)**.** For any Σ-theory $\mathcal{T}$, a Σ-sentence is *satisfiable in $\mathcal{T}$* iff it is satisfied by one of $\mathcal{T}$'s models. A set Γ of Σ-sentences *$\mathcal{T}$-entails* a Σ-sentence φ, written $\Gamma \models_{\mathcal{T}} \varphi$, iff every model of $\mathcal{T}$ that satisfies all sentences in Γ satisfies φ as well.

5.4.1 Theory declarations

In SMT-LIB, basic theories are obtained as instances of theory declarations. (In contrast, combined theories are defined in logic declarations.) An abstract syntax of theory declarations is defined in Figure 5.4. The symbol Π in parametric function symbol declarations is a (universal) binder for the sort parameters—and corresponds to the symbol `par` in the concrete syntax.

To simplify the meta-notation let T denote a theory declaration with theory name T. Given such a theory declaration, assume first that T has no **sorts-description** and **funs-description** attributes, and let S and F be respectively the set of all sort symbols and all function symbols occurring in T. Let Ω be a signature *whose sort symbols include all the symbols in S, with the same arity.* The definition provided in the **definition** attribute of T must be such that every signature like Ω above uniquely determines a theory $T[\Omega]$ as an instance of T with signature $\widehat{\Omega}$ defined as follows:

[6] Observe that compatibility is an equivalence relation on signatures.

(Sort symbol declarations)	*sdec*	::=	$s\ n\ \alpha^*$
(Fun. symbol declarations)	*fdec*	::=	$f\ \sigma^+\ \alpha^*$
(Param. fun. symbol declarations)	*pdec*	::=	*fdec* $\mid\ \Pi\, u^+\ (f\ \tau^+\ \alpha^*)$
(Theory attributes)	*tattr*	::=	**sorts** $= sdec^+$ $\mid$ **funs** $= pdec^+$
		$\mid$	**sorts-description** $= w$
		$\mid$	**funs-description** $= w$
		$\mid$	**definition** $= w$ $\mid$ **values** $= t^+$
		$\mid$	**notes** $= w$ $\mid$ α
(Theory declarations)	*tdec*	::=	**theory** T $tattr^+$

Figure 5.4: Abstract syntax for theory declarations

1. $\widehat{\Omega}^{\mathrm{S}} = \Omega^{\mathrm{S}}$ and $\widehat{\Omega}^{\mathrm{F}} = F \cup \Omega^{\mathrm{F}}$,
2. no variables are sorted in $\widehat{\Omega}$,(48)
3. for all $f \in \widehat{\Omega}^{\mathrm{F}}$ and $\bar{\sigma} \in (\widehat{\Omega}^{\mathrm{S}})^+$, $f{:}\bar{\sigma} \in \widehat{\Omega}$ iff
   (a) $f{:}\bar{\sigma} \in \Omega$, or
   (b) T contains a declaration of the form $f\ \bar{\sigma}\ \bar{\alpha}$, or
   (c) T contains a declaration of the form $\Pi\ \bar{u}\ (f\ \bar{\tau}\ \bar{\alpha})$ and $\bar{\sigma}$ is an instance of $\bar{\tau}$.

We say that a ranked function symbol $f{:}\bar{\sigma}$ of $\widehat{\Omega}$ is *declared in T* if $f{:}\bar{\sigma} \in \widehat{\Omega}$ because of Point 3b or 3c above. We call the sort symbols of $\widehat{\Omega}$ that are not in S the *free sort symbols* of $T[\Omega]$. Similarly, we call the ranked function symbols of $\widehat{\Omega}$ that are not declared in T the *free function symbols* of $T[\Omega]$.[7] This terminology is justified by the following additional requirement on T.

The definition of T should be *parametric*, in this sense: it must not constrain the free symbols of any instance $T[\Omega]$ of T in any way. Technically, T must be defined so that the set of models of $T[\Omega]$ is closed under any changes in the interpretation of the free symbols. That is, every structure obtained from a model of $T[\Omega]$ by changing only the interpretation of $T[\Omega]$'s free symbols should be a model of $T[\Omega]$ as well.(49)

The case of theory declarations with **sorts-description** and **funs-description** attributes is similar.

5.5 Logics

A logic in SMT-LIB is any sublogic of the main SMT-LIB logic obtained by

- fixing a signature Σ and a Σ-theory $\mathcal{T}$,
- restricting the set of structures to the models of $\mathcal{T}$, and

[7] Note that because of overloading we talk about *ranked* function symbols being free or not, not just function symbols.

(Logic attributes)	*lattr*	::=	**theories** $= T^+$ | **language** $= w$
		|	**extensions** $= w$ | **values** $= w$
		|	**notes** $= w$ | α
(Logic declarations)	*ldec*	::=	**logic** L *lattr*$^+$

Figure 5.5: Abstract syntax for logic declarations

- restricting the set of sentences to some subset of the set of all Σ-sentences.

A *model* of a logic with theory $\mathcal{T}$ is any model of $\mathcal{T}$; a sentence is *satisfiable* in the logic iff it satisfiable in $\mathcal{T}$.

5.5.1 Logic declarations

Logics are specified by means of logic declarations. Contrary to the theory declarations, a logic declaration specifies a *single* logic, not a class of them, so we call the logic L too. An abstract syntax of theory declarations is defined in Figure 5.5.

Let L be a logic declaration whose **theories** attribute has value $T_1, \ldots, T_n$.

Theory. The logic's theory is the theory $\mathcal{T}$ uniquely determined as follows. For each $i = 1, \ldots, n$, let S_i be the set of all sort symbols occurring in T_i. The text in the **language** attribute of L may specify an additional set S_0 of sort symbols and an additional set of ranked function symbols with ranks over $Sort(S)^+$ where $S = \bigcup_{i=0,\ldots,n} S_i$. Let Ω be the smallest signature with $\Omega^{\mathrm{S}} = S$ containing all those ranked function symbols. Then for each $i = 1, \ldots, n$, let $T_i[\Omega]$ be the instance of T_i determined by Ω as described in Subsection 5.4.1. The theory of L is

$$\mathcal{T} = T_1[\Omega] + \cdots + T_n[\Omega] \ .$$

Note that $\mathcal{T}$ is well defined. To start, Ω is well defined because any sort symbols shared by two declarations among $T_1, \ldots, T_n$ have the same arity in them. The theories $T_1[\Omega], \ldots, T_n[\Omega]$ are well defined because Ω satisfies the requirements in Subsection 5.4.1. Finally, the signatures of $T_1[\Omega], \ldots, T_n[\Omega]$ are pairwise compatible because they all have the same sort symbols, each with the same arity in all of them.

Values. The **values** attribute is expected to designate for each sort σ of the logic's theory $\mathcal{T}$ a distinguished set V_σ of ground terms called *values*. The definition of V_σ should be such that every sentence satisfiable in the logic L is satisfiable in a model $\mathbf{A}$ of $\mathcal{T}$ where each element of $\sigma^{\mathbf{A}}$ is denoted by some element of V_σ. In other words, if Σ is $\mathcal{T}$'s signature, $\mathbf{A}$ is such that, for all $\sigma \in \Sigma^{\mathrm{S}}$ and all $a \in \sigma^{\mathbf{A}}$, $a = [\![t]\!]^{\mathcal{A}}$ for some $t \in V_\sigma$. For example, in a logic of the integers, the set of values for the integer sort might consist of all the terms of the form 0 or $[-]n$ where n is a non-zero numeral.

For flexibility, we do not require that V_σ be minimal. That is, it is possible for two terms of V_σ to denote the same element of $\sigma^{\mathbf{A}}$. For example, in a logic of rational numbers, the set of values for the rational sort might consist of all the terms of the form $[-]m/n$ where m is a numeral and n is a non-zero numeral. This set covers all the rationals but, in contrast with the previous example, is not minimal because, for instance, $3/2$ and $9/6$ denote the same rational.

Note that the requirements on V_σ can be always trivially satisfied by L by making sure that the signature Ω above contains a distinguished set of infinitely many additional free constant symbols

of sort σ, and defining V_σ to be that set. We call these constant symbols *abstract values*. Abstract values are useful to denote the elements of uninterpreted sorts or sorts standing for structured data types such as lists, arrays, sets and so on.[8]

[8] The concrete syntax reserves a special format for constant symbols used as abstract values: they are members of the ⟨*symbol*⟩ category that start with the character `@`.

Part IV

References

Bibliography

[And86] Peter Andrews. *An Introduction to Mathematical Logic and Type Theory: to Truth through Proof*. Academic Press, 1986.

[BBC+05] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Junttila, Peter van Rossum, Stephan Schulz, and Roberto Sebastiani. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th Int. Conf., (TACAS)*, volume 3440 of *Lecture Notes in Computer Science*, pages 317–333, 2005.

[BdMS05] Clark W. Barrett, Leonardo de Moura, and Aaron Stump. SMT-COMP: Satisfiability Modulo Theories Competition. In K. Etessami and S. Rajamani, editors, *17th International Conference on Computer Aided Verification*, pages 20–23. Springer, 2005.

[BMS06] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *In Proc. Verification, Model-Checking, and Abstract-Interpretation (VMCAI) 2006*, volume 3855 of *Lecture Notes in Computer Science*, pages 427–442. Springer-Verlag, 2006.

[BSST09] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185, chapter 26, pages 825–885. IOS Press, February 2009.

[BST10a] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[BST10b] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at `www.SMT-LIB.org`.

[End01] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 2nd edition, 2001.

[Fit96] Melvin Fitting. *First-Order Logic and Automated Theorem Proving*. Graduate Texts in Computer Science. Springer, Berlin, 2nd edition, 1996.

[Gal86] Jean Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. John Wiley & Sons Inc, 1986.

[HS00] Holger H. Hoos and Thomas Stützle. SATLIB: An Online Resource for Research on SAT. In Ian Gent, Hans van Maaren, and Toby Walsh, editors, *SAT2000: Highlights of Satisfiability Research in the year 2000*, Frontiers in Artificial Intelligence and Applications, pages 283–292. Kluwer Academic, 2000.

[Man93] María Manzano. Introduction to many-sorted logic. In *Many-sorted logic and its applications*, pages 3–86. John Wiley & Sons, Inc., 1993.

[Men09] Elliott Mendelson. *Introduction to Mathematical Logic*. Chapman & Hall, 5th edition, 2009.

[RT06] Silvio Ranise and Cesare Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006. Available at `www.SMT-LIB.org`.

[Ste90] Guy L. Steele. *Common Lisp the Language*. Digital Press, 2nd edition, 1990.

[Sut09] Geoff Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

Part V

Appendices

Appendix A

Notes

1 To define such theory signatures formally, SMT-LIB's would need to rely on a more powerful underlying logic, for instance one with dependent types.

2 Preferring ease of parsing over human readability is reasonable in this context not only because SMT-LIB benchmarks are meant to be read by solvers but also because they are produced in the first place by automated tools like verification condition generators or translators from other formats.

3 There was much internal discussion on whether to adopt the Unicode UTF-8 standard instead, which is used in most web-based applications currently. We opted for ISO 8859-1 mostly because it is a fixed character encoding and is still well supported by all operating systems. A move to UTF-8 might be made in a later version, depending on the feedback and needs of the community.

4 This syntactical category excludes the non-English letters of Extended ASCII because it is used to define identifiers, which traditionally use only English letters. Future versions might extend it to non-English letters as well.

5 This is to achieve maximum generality and independence from programming language conventions. This way, SMT-LIB theories of strings that use string literals as constant symbols have the choice to define certain string constants, such as `"\n"` and `"\012"`, as equivalent or not. If we used, say, C-style backslash-prefixed escape sequences at the SMT-LIB level, it would be impractical and possibly confusing to represent literally certain sequences of characters. For instance, with C-style conventions the literals `"\\e"` and `"\e"` would be parsed as the same two-character literal consisting of the characters $5C_{hex}$ and 65_{hex}. To represent the three literal string consisting of the characters $5C_{hex}5C_{hex}65_{hex}$, one would have to write, for instance, something like `"\\\e"` or `"\x5C\e"`.

6 Strictly speaking, command names do not need to be reserved words because of the language's namespace conventions. Having them as reserved words, however, simplifies the creation of compliant parsers with the aid of parser generators like Lex/YACC and ANTLR.

7 Backslash is disallowed in quoted symbols just for simplicity. Otherwise, for Common Lisp compatibility they would have to be treated as an escaping symbol (see Section 2.3 of [Ste90]).

8 Simple symbols have been added as indices in Version 2.5 for increased flexibility.

9 The rationale for allowing user-defined attributes is the same as in other attribute-value-based languages (such as, e.g., BibTeX). It makes the SMT-LIB format more flexible and customizable. The understanding is that user-defined attributes are allowed but need not be supported by an SMT solver for the solver to be considered *SMT-LIB compliant*. We expect, however, that with continued use of the SMT-LIB format, certain user-defined attributes will become widely used. Those attributes might then be officially adopted into the format (as non-user-defined attributes) in later versions.

10 See the point made in Note 11.

11 Ideally, it would be better if `:definition` were a formal attribute, to avoid ambiguities and misinterpretation and possibly allow automatic processing. The choice of using free text for this attribute is motivated by practicality reasons. The enormous amount of effort needed to first devise a formal language for this attribute and then specify its value for each theory in the library is not justified by the current goals of SMT-LIB. Furthermore, this attribute is meant mainly for human readers, not programs, hence a natural language, but mathematically rigorous definition, seems enough.

12 Version 1.2 allowed one to specify a finitely-axiomatizable theory formally by listing a set of axioms an `:axioms` attribute. This attribute is gone in Version 2.0, because only one or two theories in the new SMT-LIB catalog can be defined that way. The remaining ones require infinitely many axioms or axioms with quantified sort symbols which are not expressible in the language.

13 The theory declaration `Empty` in Version 1.2 of SMT-LIB is superseded by the `Core` theory declaration schema.

14 One advantage of defining instances of theory declaration schemas this way is that with one instantiation of the schema one gets a *single* theory with arbitrarily nested sorts—another example being the theory of all nested lists of integers, say, with sorts `(List Int)`, `(List (List Int))`. This is convenient in applications coming from software verification, where verification conditions can contain arbitrarily nested data types. But it is also crucial in providing a simple and powerful mechanism for theory combination, as explained later.

15 The reason for informal attributes is similar to that for theory declarations.

16 The attribute is text-valued because it is mostly for documentation purposes for the benefit of benchmark users. A natural language description of the logic's language seems therefore adequate for this purpose. Of course, it is also possible to specify the language at least partially in some formal fashion in this attribute, for instance by using BNF rules.

17 This is useful because in common practice the syntax of a logic is often extended for convenience with syntactic sugar.

18 This enables applications reading a compliant solver's response to know when an identifier (like `success`) has been completely printed and, in general, when the solver has completed processing a command. For example, this is needed if one wants to use an off-the-shelf S-expression parser (e.g., `read` in Common Lisp) to read responses.

19 The motivation for allowing these two behaviors is that the first one (exiting immediately when an error occurs) may be simpler to implement, while the latter may be more useful for applications, though it might be more burdensome to support the semantics of leaving the state unmodified by the erroneous command.

20 It is desirable to have the ability to remove declarations and definitions, for example if they are no longer needed at some point during an interaction with a solver (and so the memory required for them might be reclaimed), or if a defined symbol is to be redefined. The current approach of allowing declarations and definitions to be locally scoped supports removal by popping the containing assertion level. Other approaches, such as the ability to add shadowing declarations or definitions of symbols, or to "undefine" or "undeclare" them, present some issues: for example, how to print symbols that have been shadowed, undefined or undeclared. As a consequence, they are not supported in the language.

21 The purpose of the `:global-declarations` is to allow simpler implementations that avoid the complexity of supporting local, undoable declarations and definitions.

22 The motivation for not overloading user defined symbols is to simplify their processing by a solver. This restriction is significant only for users who want to extend the signature of the theory used by a script with a new polymorphic function symbol—i.e., one whose rank would contain parametric sorts if it was a theory symbol. For instance, users who want to declare a "reverse" function on arbitrary lists, must define a different reverse function symbol for each (concrete) list sort used in the script. This restriction might be removed in future versions.

23 The reason is simply that, in general, it might not be possible to eliminate those symbols using their `define-fun-rec` definition.

24 Note that this option is not intended to be used for comparisons between different solvers since they can implement it differently. Its purpose is simply to guarantee the reproducibility of an individual solver's results under the same external conditions.

25 This is to avoid confusion with the responses to commands, which are written to standard output.

26 Some commonly used statistics (e.g., number of restarts of a solver's propositional reasoning engine) are difficult to define precisely and generally, while the exact semantics of others, such as time and memory usage, have not being agreed upon yet by the SMT community.

27 The command is useful for interactive use, to keep track of the current number of nested `push` commands.

28 This allows the user to reset the state of the solver without paying the cost of restarting it.

29 The rationale is that a solver may need to make substantial changes to its internal configuration to provide the functionality requested by these options, and so needs to be notified in advance.

30 Strictly speaking, only sort symbols introduced with `declare-sort` expand the initial signature of theory sort symbols. Sort symbols introduced with `define-sort` do not. They do not construct *real* sorts, but *aliases* of sorts built with theory sorts symbols and previously declared sort symbols.

31 Similarly to `define-fun`, while strictly not needed, `define-fun-rec` provides a more structured way to define functions axiomatically, as opposed to introducing a new function with `declare-fun` and then proving its definition with `assert` as a quantified formula. This gives an SMT solver the opportunity to easily recognize function definitions and possibly process them specially.

32 The motivation is to enable interactive users to see easily (exactly) which assertions they have asserted, without having to keep track of that information themselves.

33 The motivation for having this command is that it corresponds to a common usage pattern with SMT solvers which can be implemented considerably more efficiently than the general stack-based mechanism needed to support `push` and `pop` commands. The restriction of assumptions to Boolean constants only is also motivated by efficiency concerns since user-defined constants are associated to a formula that has already been internally simplified at definition time.

34 SMT solvers are incomplete for certain logics, typically those that include quantified formulas. However, even when they are unable to determine whether the set of all assertions Γ is satisfiable or not, SMT solvers can typically compute a model for a set Γ' of formulas that is entailed by Γ in the logic. Interpretations in this model are often useful to a client applications even if they are not guaranteed to come from a model of Γ.

35 Abstract values are useful in `get-model` with logics containing for example the theory of arrays (see Figure 3.4). For instance, a solver may specify the content of an array `a` of sort `(Array Int Int)` at positions 0–2 by returning the expression

```
(define-fun a () (store (store (store (as @array1 (Array Int Int)) 0 0) 1 2) 2 4)) .
```

The elements of `a` with index outside the 0-2 range are left unspecified because the array `@array1` itself is left unspecified.

36 Since it focuses only on preselected, Boolean terms, `get-assignment` can be implemented much more efficiently than the more general `get-value`.

37 The rationale for providing the interpretation of a function symbol as a define command is that (*i*) the syntax of such commands is general enough to be able to express a large class of functions symbolically in the language of the current logic (possibly augmented with abstract values), and (*ii*) in principle the user could fix the provided definition for later by asserting it back to the solver as is.

38 The current requirements on the returned definitions are rather weak. For instance, they allow a solver to return something like

(define-fun-rec ((f ((x1 σ1) ··· (xn σn)) σ (f x1 ··· xn)))

for a given f of rank $\sigma_1 \cdots \sigma_n \sigma$. Similarly, for constant symbols c of a sort σ that admits abstract values, they allow a solver to return `(define-fun c () @a)` with `@a` abstract.
The reason for such weak requirements is that stronger ones are currently difficult to achieve in general because of limitations in the expressive power of some SMT-LIB theories/logics or in the computational abilities of present SMT solvers. For instance, the current theory of arrays (see Figure 3.4) does not have enough *constructor* symbols to allow one to represent an array uniquely as a value term. As shown in a previous note, one can use terms like

```
(store (store (store (as @array1 (Array Int Int)) 0 0) 1 2) 2 4))
```

which fixes only a portion of the array. This term has infinitely many interpretations that differ on the elements at indices outside the 0–2 range. Similarly, because of the progress in automated synthesis, it is conceivable that future solvers will be able to construct a model where a user-declared function symbol f denotes the factorial function over the non-negative integers. In that case, a definition like

```
(define-fun-rec ( (f ((x Int)) Int (ite (= x 0) 1 (* x (f (- x 1))))) ))
```

would not have a unique interpretation because it does not uniquely determine the behavior of f over the negative integers. In contrast, the definition

```
(define-fun-rec ( (f ((x Int)) Int (ite (<= x 1) 1 (* x (f (- x 1))))) ) '
```

say, would determine a unique function over the whole set of integers.

39 The lax requirement is justified by the fact that the minimization problem alone is NP-hard in general. On the other hand, it allows a solver to be compliant by just returning the same sequence given to (the most recent) `check-sat-assuming`.

40 There is, as yet, no standard SMT-LIB proof format.

41 Unsatisfiable cores are useful for applications because they circumscribe the source of unsatisfiability in the asserted set. The labeling mechanism allows users to track only selected asserted formulas when they already know that the rest of the asserted formulas are jointly satisfiable.

42 This is particularly useful for scripts that are used as benchmarks, as `meta-info` can be used to store such information as authors, date, expected response of `check-sat` commands, difficulty level, and so on.

43 The name `set-info` is confusing because it suggests the command is related to `get-info`, which it is not.

44 Interjecting `echo` commands in a script can help a software client know where the solver is in the execution of the script's commands.

45 It would have been reasonable to adopt an alternative version of the rule for well-sortedness of terms $(f^\sigma\ t_1\ \cdots\ t_k)\ \alpha^*$ with annotated function symbols f^σ, without the second conjunct of the rule's side condition. This would allow formation of terms with annotated function symbols f^σ, even when f lacked two ranks of the forms $\sigma_1 \cdots \sigma_k \sigma$ and $\sigma_1 \cdots \sigma_k \sigma'$, for distinct σ and σ'. The rationale for keeping this second conjunct is that with it, function symbols are annotated when used iff they are overloaded in this way. This means that it is clear from the use of the function symbol, whether or not the annotation is required. This in turn should help to improve human comprehension of scripts written using overloaded function symbols.

46 This is mostly a technical restriction, motivated by considerations of convenience. In fact, with a closed formula φ of signature Σ the signature's mapping of variables to sorts 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** binder. 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.

47 Distinct sorts can have non-disjoint extensions in a structure. However, whether they do that or not is irrelevant in SMT-LIB logic. The reason is that the logic has no sort predicates, such as a subsort predicate, and does not allow one to equate terms of different sorts (the term $t_1 \approx t_2$ is ill-sorted unless t_1 and t_2 have the same sort). As a consequence, a formula is satisfiable in a structure where two given sorts have non-disjoint extensions iff it is satisfiable in a structure where the two sorts do have disjoint extensions.

48 This requirement is for concreteness. Again, since we work with closed formulas, which internally assign sorts to their variables, the sorting of variables in a signature is irrelevant.

49 Admittedly, this requirement on theory declarations is somewhat hand-wavy. Unfortunately, it is not possible to make it a lot more rigorous because theory declarations can use natural language to define their class of instance theories. The point is again that the definition of the class should impose no constraints on the interpretation of free sort symbols and free function symbols.

Appendix B

Concrete Syntax

Predefined symbols

These symbols have a predefined meaning in Version 2.5. Note that they are not reserved words. For instance, they could also be used in principle as user-defined sort or function symbols in scripts.

`Bool continued-execution error false immediate-exit incomplete logic memout sat success theory true unknown unsupported unsat`

Predefined keywords

These keywords have a predefined meaning in Version 2.5.

`:all-statistics :assertion-stack-levels :authors :axioms :chainable :definition :diagnostic-output-channel :error-behavior :expand-definitions :extensions :funs :funs-description :global-declarations :interactive-mode :language :left-assoc :name :named :notes :print-success :produce-assignments :produce-models :produce-proofs :produce-unsat-cores :random-seed :reason-unknown :regular-output-channel :reproducible-resource-limit :right-assoc :sorts :sorts-description :theories :values :verbosity :version`

Auxiliary Lexical Categories

⟨*white_space_char*⟩ ::= 9_{dec} | 10_{dec} | 13_{dec} | 32_{dec} | 160_{dec}

⟨*printable_char*⟩ ::= 32_{dec} | ··· | 126_{dec} | 160_{dec} | ··· | 255_{dec}

⟨*digit*⟩ ::= `0` | ··· | `9`

⟨*letter*⟩ ::= `A` | ··· | `A` | `a` | ··· | `z`

Tokens

Reserved Words

General: `!` `_` `as` `BINARY` `DECIMAL` `exists` `HEXADECIMAL` `forall` `let` `NUMERAL` `par` `STRING`

Command names: `assert` `check-sat` `check-sat-assuming` `declare-const` `declare-fun` `declare-sort` `define-fun` `define-fun-rec` `define-sort` `echo` `exit` `get-assertions` `get-assignment` `get-info` `get-model` `get-option` `get-proof` `get-unsat-assumptions` `get-unsat-core` `get-value` `meta-info` `pop` `push` `reset` `reset-assertions` `set-info` `set-logic` `set-option`

Other tokens

`(`

`)`

⟨*numeral*⟩	::=	`0` | *a non-empty sequence of digits not starting with* `0`
⟨*decimal*⟩	::=	⟨*numeral*⟩`.0`*⟨*numeral*⟩
⟨*hexadecimal*⟩	::=	`#x` *followed by a non-empty sequence of digits and letters from* `A` *to* `F` *, capitalized or not*
⟨*binary*⟩	::=	`#b` *followed by a non-empty sequence of* `0` *and* `1` *characters*
⟨*string*⟩	::=	*sequence of whitespace and printable characters in double quotes with escape sequence* `""`
⟨*simple_symbol*⟩	::=	*a non-empty sequence of letters, digits and the characters* `+ - / * = % ? ! . $ _ ~ & ^ < > @` *that does not start with a digit*
⟨*symbol*⟩	::=	⟨*simple_symbol*⟩
	|	*a sequence of whitespace and printable characters that starts and ends with* `|` *and does not contain* `\`
⟨*keyword*⟩	::=	`:`⟨*simple_symbol*⟩

Members of the ⟨*symbol*⟩ category starting with of the characters `@` and `.` are reserved for solver use. Solvers can use them respectively as identifiers for abstract values and solver generated function symbols other than abstract values.

S-expressions

⟨*spec_constant*⟩	::=	⟨*numeral*⟩ | ⟨*decimal*⟩ | ⟨*hexadecimal*⟩ | ⟨*binary*⟩
	|	⟨*string*⟩
⟨*s_expr*⟩	::=	⟨*spec_constant*⟩ | ⟨*symbol*⟩ | ⟨*keyword*⟩ | `(` ⟨*s_expr*⟩* `)`

Identifiers

⟨*index*⟩	::=	⟨*numeral*⟩ | ⟨*simple symbol*⟩
⟨*identifier*⟩	::=	⟨*symbol*⟩ | `(` `_` ⟨*symbol*⟩ ⟨*index*⟩$^+$ `)`

Sorts

⟨*sort*⟩ ::= ⟨*identifier*⟩ | `(` ⟨*identifier*⟩ ⟨*sort*⟩$^+$ `)`

Attributes

⟨*attribute_value*⟩ ::= ⟨*spec_constant*⟩ | ⟨*symbol*⟩ | `(` ⟨*s_expr*⟩* `)`
⟨*attribute*⟩ ::= ⟨*keyword*⟩ | ⟨*keyword*⟩ ⟨*attribute_value*⟩

Terms

⟨*qual_identifier*⟩ ::= ⟨*identifier*⟩ | `(` `as` ⟨*identifier*⟩ ⟨*sort*⟩ `)`
⟨*var_binding*⟩ ::= `(` ⟨*symbol*⟩ ⟨*term*⟩ `)`
⟨*sorted_var*⟩ ::= `(` ⟨*symbol*⟩ ⟨*sort*⟩ `)`
⟨*term*⟩ ::= ⟨*spec_constant*⟩
| ⟨*qual_identifier*⟩
| `(` ⟨*qual_identifier*⟩ ⟨*term*⟩$^+$ `)`
| `(` `let` `(` ⟨*var_binding*⟩$^+$ `)` ⟨*term*⟩ `)`
| `(` `forall` `(` ⟨*sorted_var*⟩$^+$ `)` ⟨*term*⟩ `)`
| `(` `exists` `(` ⟨*sorted_var*⟩$^+$ `)` ⟨*term*⟩ `)`
| `(` `!` ⟨*term*⟩ ⟨*attribute*⟩$^+$ `)`

Theories

⟨*sort_symbol_decl*⟩ ::= `(` ⟨*identifier*⟩ ⟨*numeral*⟩ ⟨*attribute*⟩* `)`
⟨*meta_spec_constant*⟩ ::= `NUMERAL` | `DECIMAL` | `STRING`
⟨*fun_symbol_decl*⟩ ::= `(` ⟨*spec_constant*⟩ ⟨*sort*⟩ ⟨*attribute*⟩* `)`
| `(` ⟨*meta_spec_constant*⟩ ⟨*sort*⟩ ⟨*attribute*⟩* `)`
| `(` ⟨*identifier*⟩ ⟨*sort*⟩$^+$ ⟨*attribute*⟩* `)`
⟨*par_fun_symbol_decl*⟩ ::= ⟨*fun_symbol_decl*⟩
| `(` `par` `(` ⟨*symbol*⟩$^+$ `)`
`(` ⟨*identifier*⟩ ⟨*sort*⟩$^+$ ⟨*attribute*⟩* `)` `)`
⟨*theory_attribute*⟩ ::= `:sorts` `(` ⟨*sort_symbol*⟩$^+$ `)`
| `:funs` `(` ⟨*par_fun_symbol_decl*⟩$^+$ `)`
| `:sorts-description` ⟨*string*⟩
| `:funs-description` ⟨*string*⟩
| `:definition` ⟨*string*⟩
| `:values` ⟨*string*⟩
| `:notes` ⟨*string*⟩
| ⟨*attribute*⟩
⟨*theory_decl*⟩ ::= `(` `theory` ⟨*symbol*⟩ ⟨*theory_attribute*⟩$^+$ `)`

Logics

⟨*logic_attribute*⟩ := `:theories (` ⟨*symbol*⟩$^+$ `)`
| `:language` ⟨*string*⟩
| `:extensions` ⟨*string*⟩
| `:values` ⟨*string*⟩
| `:notes` ⟨*string*⟩
| ⟨*attribute*⟩

⟨*logic*⟩ ::= `( logic` ⟨*symbol*⟩ ⟨*logic_attribute*⟩$^+$ `)`

Info flags

⟨*info_flag*⟩ ::= `:all-statistics` | `:assertion-stack-levels` | `:authors`
| `:error-behavior` | `:name` | `:reason-unknown`
| `:version` | ⟨*keyword*⟩

Command options

⟨*b_value*⟩ ::= `true` | `false`

⟨*option*⟩ ::= `:diagnostic-output-channel` ⟨*string*⟩
| `:expand-definitions` ⟨*b_value*⟩
| `:global-declarations` ⟨*b_value*⟩
| `:interactive-mode` ⟨*b_value*⟩
| `:print-success` ⟨*b_value*⟩
| `:produce-assertions` ⟨*b_value*⟩
| `:produce-assignments` ⟨*b_value*⟩
| `:produce-models` ⟨*b_value*⟩
| `:produce-proofs` ⟨*b_value*⟩
| `:produce-unsat-cores` ⟨*b_value*⟩
| `:random-seed` ⟨*numeral*⟩
| `:regular-output-channel` ⟨*string*⟩
| `:reproducible-resource-limit` ⟨*numeral*⟩
| `:verbosity` ⟨*numeral*⟩
| ⟨*attribute*⟩

Commands

⟨*fun_def*⟩ ::= ⟨*symbol*⟩ `(` ⟨*sorted_var*⟩* `)` ⟨*sort*⟩ ⟨*term*⟩ `)`

⟨*fun_rec_def*⟩ ::= `(` ⟨*fun_def*⟩ `)`

⟨*command*⟩ ::= `(` `assert` ⟨*term*⟩ `)`
| `(` `check-sat` `)` | `(` `check-sat-assuming` `(` ⟨*symbol*⟩* `)` `)`
| `(` `declare-const` ⟨*symbol*⟩ ⟨*sort*⟩ `)`
| `(` `declare-fun` ⟨*symbol*⟩ `(` ⟨*sort*⟩* `)` ⟨*sort*⟩ `)`
| `(` `declare-sort` ⟨*symbol*⟩ ⟨*numeral*⟩ `)`
| `(` `define-fun` ⟨*fun_def*⟩ `)` | `(` `define-fun-rec` `(` ⟨*fun_rec_def*⟩+ `)` `)`
| `(` `define-sort` ⟨*symbol*⟩ `(` ⟨*symbol*⟩* `)` ⟨*sort*⟩ `)`
| `(` `echo` ⟨*string*⟩ `)` | `(` `exit` `)`
| `(` `get-assertions` `)` | `(` `get-assignment` `)`
| `(` `get-info` ⟨*info_flag*⟩ `)` | `(` `get-model` `)`
| `(` `get-option` ⟨*keyword*⟩ `)` | `(` `get-proof` `)`
| `(` `get-unsat-assumptions` `)` | `(` `get-unsat-core` `)`
| `(` `get-value` `(` ⟨*term*⟩+ `)` `)` | `(` `meta-info` ⟨*attribute*⟩ `)`
| `(` `pop` ⟨*numeral*⟩ `)` | `(` `push` ⟨*numeral*⟩ `)`
| `(` `reset` `)` | `(` `reset-assertions` `)`
| `(` `set-info` ⟨*attribute*⟩ `)` | `(` `set-logic` ⟨*symbol*⟩ `)`
| `(` `set-option` ⟨*option*⟩ `)`

⟨*script*⟩ ::= ⟨*command*⟩*

Command responses

⟨*error-behavior*⟩ ::= `immediate-exit` | `continued-execution`

⟨*reason-unknown*⟩ ::= `memout` | `incomplete` | ⟨*s_expr*⟩

⟨*model_response*⟩ ::= `(` `define-fun` ⟨*fun_def*⟩ `)`
| `(` `define-fun-rec` `(` ⟨*fun_rec_def*⟩$^{+}$ `)` `)`

⟨*info_response*⟩ ::= `:assertion-stack-levels` ⟨*numeral*⟩
| `:authors` ⟨*string*⟩
| `:error-behavior` ⟨*error-behavior*⟩
| `:name` ⟨*string*⟩
| `:reason-unknown` ⟨*reason-unknown*⟩
| `:version` ⟨*string*⟩
| ⟨*attribute*⟩

⟨*valuation_pair*⟩ ::= `(` ⟨*term*⟩ ⟨*term*⟩ `)`

⟨*t_valuation_pair*⟩ ::= `(` ⟨*symbol*⟩ ⟨*b_value*⟩ `)`

⟨*check_sat_response*⟩ ::= `sat` | `unsat` | `unknown`

⟨*echo_response*⟩ ::= ⟨*string*⟩

⟨*get_assertions_response*⟩ ::= `(` ⟨*term*⟩* `)`

⟨*get_assignment_response*⟩ ::= `(` ⟨*t_valuation_pair*⟩* `)`

⟨*get_info_response*⟩ ::= `(` ⟨*info_response*⟩$^{+}$ `)`

⟨*get_model_response*⟩ ::= `(` ⟨*model_response*⟩$^{+}$ `)`

⟨*get_option_response*⟩ ::= ⟨*attribute_value*⟩

⟨*get_proof_response*⟩ ::= ⟨*s_expr*⟩

⟨*get_unsat_assump_response*⟩ ::= `(` ⟨*symbol*⟩* `)`

⟨*get_unsat_core_response*⟩ ::= `(` ⟨*symbol*⟩* `)`

⟨*get_value_response*⟩ ::= `(` ⟨*valuation_pair*⟩$^{+}$ `)`

⟨*specific_success_response*⟩ ::= ⟨*check_sat_response*⟩ | ⟨*echo_response*⟩
| ⟨*get_assertions_response*⟩ | ⟨*get_assignment_response*⟩
| ⟨*get_info_response*⟩ | ⟨*get_model_response*⟩
| ⟨*get_info_response*⟩ | ⟨*get_model_response*⟩
| ⟨*get_option_response*⟩ | ⟨*get_proof_response*⟩
| ⟨*get_unsat_assumptions_response*⟩
| ⟨*get_unsat_core_response*⟩ | ⟨*get_value_response*⟩

⟨*general_response*⟩ ::= `success` | ⟨*specific_success_response*⟩
| `unsupported` | `(` `error` ⟨*string*⟩ `)`

Appendix C

Abstract Syntax

Common Notation

$b \in \mathcal{B}$,	the set of boolean values	$r \in \mathcal{R}$,	the set of non-negative rational numbers
$n \in \mathcal{N}$,	the set of natural numbers	$w \in \mathcal{W}$,	the set of character strings
$s \in \mathcal{S}$,	the set of sort symbols	$u \in \mathcal{U}$,	the set of sort parameters
$f \in \mathcal{F}$,	the set of function symbols	$x \in \mathcal{X}$,	the set of variables
$a \in \mathcal{A}$,	the set of attribute names	$v \in \mathcal{V}$,	the set of attribute values
$T \in \mathcal{TN}$,	the set of theory names	$L \in \mathcal{L}$,	the set of logic names

Sorts

(Sorts) $\sigma ::= s\ \sigma^*$

(Parametric Sorts) $\tau ::= u \mid s\ \tau^*$

Terms

(Attributes) $\alpha ::= a \mid a = v$

(Terms) $t ::= x \mid f\ t^* \mid f^\sigma\ t^*$
$\mid \exists\,(x{:}\sigma)^+\ t \mid \forall\,(x{:}\sigma)^+\ t \mid \textbf{let}\ (x = t)^+\ \textbf{in}\ t$
$\mid t\ \alpha^+$

Well-sorting rules for terms

$$\frac{}{\Sigma \vdash x\ \alpha^* : \sigma} \quad \text{if } x{:}\sigma \in \Sigma$$

$$\frac{\Sigma \vdash t_1 : \sigma_1 \quad \cdots \quad \Sigma \vdash t_k : \sigma_k}{\Sigma \vdash (f\ t_1\ \cdots\ t_k)\ \alpha^* : \sigma} \quad \text{if } \begin{cases} f{:}\sigma_1 \cdots \sigma_k\sigma \in \Sigma & \text{and} \\ f{:}\sigma_1 \cdots \sigma_k\sigma' \notin \Sigma & \text{for all } \sigma' \neq \sigma \end{cases}$$

$$\frac{\Sigma \vdash t_1 : \sigma_1 \quad \cdots \quad \Sigma \vdash t_k : \sigma_k}{\Sigma \vdash (f^{\sigma}\ t_1\ \cdots\ t_k)\ \alpha^* : \sigma} \quad \text{if } \begin{cases} f{:}\sigma_1 \cdots \sigma_k\sigma \in \Sigma & \text{and} \\ f{:}\sigma_1 \cdots \sigma_k\sigma' \in \Sigma & \text{for some } \sigma' \neq \sigma \end{cases}$$

$$\frac{\Sigma[x_1{:}\sigma_1,\ \ldots,\ x_{k+1}{:}\sigma_{k+1}] \vdash t : \mathbf{Bool}}{\Sigma \vdash (Qx_1{:}\sigma_1 \cdots x_{k+1}{:}\sigma_{k+1}\ t)\ \alpha^* : \mathbf{Bool}} \quad \text{if } Q \in \{\exists, \forall\}$$

$$\frac{\Sigma \vdash t_1 : \sigma_1 \quad \cdots \quad \Sigma \vdash t_{k+1} : \sigma_{k+1} \quad \Sigma[x_1{:}\sigma_1,\ \ldots,\ x_{k+1}{:}\sigma_{k+1}] \vdash t : \sigma}{\Sigma \vdash (\mathbf{let}\ x_1 = t_1\ \cdots\ x_{k+1} = t_{k+1}\ \mathbf{in}\ t)\ \alpha^* : \sigma}$$

Theories

(Sort symbol declarations)	*sdec*	::=	$s\ n\ \alpha^*$
(Fun. symbol declarations)	*fdec*	::=	$f\ \sigma^+\ \alpha^*$
(Param. fun. symbol declarations)	*pdec*	::=	*fdec* | $\Pi\, u^+\ (f\ \tau^+\ \alpha^*)$
(Theory attributes)	*tattr*	::=	**sorts** $=$ *sdec*$^+$ | **funs** $=$ *pdec*$^+$
		|	**sorts-description** $= w$
		|	**funs-description** $= w$
		|	**definition** $= w$ | **values** $= t^+$
		|	**notes** $= w$ | α
(Theory declarations)	*tdec*	::=	**theory** T *tattr*$^+$

Logics

(Logic attributes)	*lattr*	::=	**theories** $= T^+$ | **language** $= w$
		|	**extensions** $= w$ | **values** $= w$
		|	**notes** $= w$ | α
(Logic declarations)	*ldec*	::=	**logic** L *lattr*$^+$

Index