

The SMT-LIB Standard

Version 2.0

DRAFT

Clark Barrett

Aaron Stump

Cesare Tinelli

February 10, 2010

Copyright © 2010 Clark Barrett, Aaron Stump and Cesare Tinelli.

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

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 2.0 of the *SMT-LIB Standard*. This is a major upgrade of the previous version, Version 1.2, which, in addition to simplifying and extending the languages of that version, includes a new command language for interfacing with SMT solvers.

Acknowledgments

Version 2.0 was developed with the input of 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 the email discussion of several drafts of the new standard: 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, Roberto Sebastiani, and Johannes Waldmann. Many thanks also to Amit Goel, Philipp Rümmer, and Tjark Weber for additional feedback.

Contents

Preface	3
Acknowledgments	4
Contents	5
List of Figures	8
I Introduction	9
1 General Information	10
1.1 About This Document	10
1.1.1 Differences with Version 1.2	10
1.1.2 Typographical and Notational Conventions	12
1.2 Overview of SMT-LIB	12
1.2.1 What is SMT-LIB?	13
1.2.2 Main features of the SMT-LIB Standard	13
2 Basic Assumptions and Structure	14
2.1 Satisfiability Modulo Theories	14
2.2 Underlying Logic	15
2.3 Background Theories	15
2.4 Input Formulas	16
2.5 Interface	16
II Syntax	18
3 The SMT-LIB Language	19
3.1 Lexicon	19
3.2 S-expressions	21
3.3 Identifiers	21

3.4	Attributes	22
3.5	Sorts	22
3.6	Terms and Formulas	23
3.7	Theory Declarations	25
3.7.1	Examples	30
3.8	Logic Declarations	31
3.8.1	Examples	33
3.9	Scripts	33
III	Semantics	40
4	SMT-LIB Logic	41
4.1	The language of sorts	42
4.2	The language of terms	43
4.2.1	Signatures	44
4.2.2	Well-sorted terms	45
4.3	Structures and Satisfiability	46
4.3.1	The meaning of terms	47
4.4	Theories	48
4.4.1	Theory Declarations	49
4.5	Logics	50
4.5.1	Logic Declarations	51
5	SMT-LIB Scripts	53
5.1	Commands	54
5.1.1	Starting and terminating	55
5.1.2	Modifying the Assertion-Set Stack	55
5.1.3	Declaring and defining new symbols	56
5.1.4	Asserting formulas and checking satisfiability	57
5.2	Solver Responses, Errors, and Other Output	59
5.3	Solver Options	60
5.4	Getting Additional Information With get-info	61
5.4.1	Standard Names for Statistics for get-info	62
5.4.2	Additional Standard Names for get-info	63
5.4.3	A Note on Benchmarks	64
IV	Appendices	65
A	Notes	66
B	Concrete Syntax	68

CONTENTS	7
----------	---

C Abstract Syntax	74
-------------------	----

V References	79
--------------	----

Bibliography	80
--------------	----

List of Figures

3.1	The Core theory declaration.	25
3.2	The Integers theory declaration.	26
3.3	The ArraysEx theory declaration.	27
3.4	Example script with solver responses in comments	38
3.5	Another example script (excerpt) with solver responses in comments	39
4.1	Abstract syntax for sort terms	42
4.2	Abstract syntax for unsorted terms	43
4.3	Well-sortedness rules for terms	45
4.4	Abstract syntax for theory declarations	50
4.5	Abstract syntax for logic declarations	51
5.1	Abstract syntax for commands	54
5.2	Abstract syntax for info responses	62

Part I

Introduction

Chapter 1

General Information

1.1 About This Document

This document is mostly self-contained, however, 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 of Version 1.2, is *not* backward compatible with it. See below 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 release 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. 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 1.2

The concrete syntax of Version 2.0 is generally simpler and leaner than that of the previous version. Moreover, SMT-LIB expressions are now a sublanguage of Common Lisp's S-expressions. Several syntactic categories, including that of benchmarks, are gone.

The two major additions are (i) a meta-level mechanism that approximates parametric sorts and polymorphic function symbols in theory declarations, and (ii) a command language

for SMT solvers that allows one, among other things, to assert and retract formulas incrementally, to define new sort and function symbols, to check the satisfiability of the asserted formulas and query their found model, if any, or ask for an unsatisfiable core otherwise.

The most notable differences with Version 1.2 are listed below.

- Sort symbols can have arity greater than 0, with sorts now denoted by structured sort terms such as `(Array Int Real)`, as opposed to just sort constants such as `IntRealArray`.
- The syntactic categories for formulas, predicate symbols and formula variables are all gone. Formulas are now terms of a distinguished Boolean sort, predicate symbols are Boolean function symbols, and formula variables are (term) variables of Boolean sort.
- The two *if-then-else* operators of Version 1.2 have been merged into a single one.
- The two *let* binders of Version 1.2 have also been merged into a single one, and extended to a *parallel-let* binder.
- Variables do not have a syntax distinct from that of function symbols anymore.
- Theory function symbols can now be overloaded arbitrarily, although in ambiguous cases their occurrences within a term must be annotated with a return sort. (User-defined function symbols cannot overload any already defined symbol.)
- Indexed identifiers are now denoted by expressions like `(_ name 3 5)`, where `_` is now a reserved operator, instead of lexemes like `name[3:5]`.
- Except for `=` and `distinct`, variadic function symbols are now disallowed. Every function symbol has a fixed arity—or a finite number of fixed arities in case of overloading. Expressions of the form `(f t1 t2 ... tn)` are allowed but only for binary theory symbols of a few specific ranks, and as syntactic sugar for expressions in which `f` is applied to two arguments only. The specific desugaring to be used is specified by an annotation in `f`'s declaration.
- The concrete syntax for term annotations has changed to `(! t α1 ... αn)` where `!` is now a reserved annotation operator, `t` is a term, and `α1 ... αn` are $n > 1$ annotations.
- Each theory declaration is now parametrized by an additional set of sort and function symbols. It stands for an infinite family of theories, each an instance of the schema, as opposed to a single theory.
- Logic declarations can refer to more than one basic theory. In that case, their background theory is a modular combination of several background theories.
- Benchmarks are superseded by scripts, sequences of commands. Version 1.2 benchmarks are converted into scripts of a very simple form. Such scripts declare a logic, (possibly) declare new sort and function symbols, assert a formula, and ask about its satisfiability.

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.

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 abstract ones, 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, \dots, 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, ground formulas), these specialized methods can be implemented into 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 are now finding 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 arithmetic, arrays, 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 these authors and endorsed by a large number of research groups world-wide, aimed at facilitating research and development in SMT [BST10]. Since its inception in 2003, the initiative has pursued these aims by focusing of 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 92,000 benchmarks and keeps growing. Formulas in SMT-LIB format are now 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 version of the 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 seeks to improve the usefulness of the SMT-LIB standard by simplifying its logical language while increasing its expressiveness and flexibility. In addition, it introduces a command language for SMT solvers that expands their SMT-LIB interface considerably, allowing users to tap the numerous functionalities that most modern SMT solvers provide.

Specifically, Version 2.0 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 constraints 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 names suggests, SMT-LIB focuses only on the SMT problem. However, at least for classes of formulas that are closed under logical negations, 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 in 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 2.0 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. Furthermore, the SMT-LIB logic uses a language of sort terms, as opposed to just sorts constant, 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.¹ 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 entirely operational. Some SMT-LIB theories, such as the theory of finite sets with 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.

¹ This catalog is available, separately from this document, from the SMT-LIB website (www.smt-lib.org).

For practicality, 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. By “formally”, in this document we mean written in a machine-readable and processable format, as opposed to written in free text, no matter how rigorously. The theory itself may be defined either formally, by means of a set of axioms, or informally, in natural language, as convenient. Some theories, such as the theory of bit vectors, have an infinite signature. For them, the signature too is specified informally in English.

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 application 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.²

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.

2.5 Interface

New to this version is 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,

² By efficiency here we do not necessarily refer to worst-case time complexity, but to efficiency in practice.

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.⁽¹⁾

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

3.1 Lexicon

The permitted characters of SMT-LIB source files are a subset of the ASCII character set. They consist of all letters, digits, whitespace characters, as well as the characters

~ ! @ # \$ % ^ & * _ - + = | \ : ; " < > . ? /

Characters between the semi-colon character ; and a line breaking character are comments—and so are to be ignored by a lexical analyzer. Non-comment text is broken into tokens by white space characters and the parenthesis characters (and). The language’s semantics is format-independent. There is no distinction between line breaks, tabs and spaces, which are all treated as white space.

The other tokens besides (and) are $\langle numeral \rangle$, $\langle rational \rangle$, $\langle hexadecimal \rangle$, $\langle binary \rangle$, $\langle string \rangle$, $\langle symbol \rangle$, and $\langle keyword \rangle$, all defined below.

Numerals. A $\langle numeral \rangle$ is a non-empty sequence of digits not starting with 0 .

Rationals. A $\langle rational \rangle$ is token of the form $\langle numeral \rangle . 0^* \langle numeral \rangle$.

Hexadecimals. A $\langle hexadecimal \rangle$ 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 $\langle binary \rangle$ is a non-empty sequence of the characters 0 and 1 preceded by the characters **#b** .

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

Strings. A $\langle string \rangle$ is ASCII character string in double quotes with C-style escaped characters: `\`, `\n`, and so on.

Symbols. A $\langle symbol \rangle$ is either a non-empty sequence of letters, digits and the characters `~ ! @ $ % ^ & * _ - + = < > . ? /` that does not start with a digit, or a sequence of printable ASCII characters, including white spaces, that starts and ends with `|` and does not otherwise contain `|` .

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

abc77 *$s&6 .kkk .8

|this is a single symbol|

||

|af klj^*(0asfsfe2(&*)&(^$>>>?"')984|
```

Note that 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 printable characters enclosed in vertical bars are also allowed as symbols.

Keywords. A $\langle \text{keyword} \rangle$ is non-empty sequence of letters, digits and the characters $\sim ! @ \$ \% \wedge \& * _ - + = < > . ? /$ preceded by the character $:$.

Elements of this category have a reserved use in the language. They are used as *attribute* names or *option* names (see later).

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

The syntax rules in this chapter are given directly with respect to streams of tokens from the set above. The whole set of concrete syntax rules is also available for fast consultation in Appendix B.

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 $\langle s_expr \rangle$ defined by the production rules below.

$$\begin{aligned} \langle \text{spec_const} \rangle &::= \langle \text{numeral} \rangle \mid \langle \text{rational} \rangle \mid \langle \text{hexadecimal} \rangle \mid \langle \text{binary} \rangle \mid \langle \text{string} \rangle \\ \langle s_expr \rangle &::= \langle \text{spec_constant} \rangle \mid \langle \text{symbol} \rangle \mid \langle \text{keyword} \rangle \mid (\langle s_expr \rangle^*) \end{aligned}$$

Remark 1. Elements of the $\langle \text{spec_const} \rangle$ category have a fixed semantics in the SMT-LIB language (with elements of $\langle \text{numeral} \rangle$ denoting integers, elements of $\langle \text{string} \rangle$ denoting character strings, and so on) only in certain syntactic categories. In contrast, in the $\langle \text{term} \rangle$ 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 $\langle \text{binary} \rangle$ may denote integers modulo n in one theory and binary strings in another; the elements of $\langle \text{rational} \rangle$ may denote rational numbers indeed in one theory and floating point values in another.

3.3 Identifiers

When defining certain SMT-LIB theories it is convenient to have indexed symbols as identifiers. Instead of having a special token syntax for that, indexed identifiers are defined more systematically as the application of the reserved symbol $_$ to a symbol and one or more *indices*, given by numerals.

$$\langle \text{identifier} \rangle ::= \langle \text{symbol} \rangle \mid (_ \langle \text{symbol} \rangle \langle \text{numeral} \rangle^+)$$

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 each other as well as function symbol names. Similarly, for bound sort parameters and sort symbols.

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 $\langle \text{keyword} \rangle$ category. Attribute values are in general S-expressions, although most predefined attributes use a more restricted category for their values.

$$\langle \text{attribute} \rangle ::= \langle \text{keyword} \rangle \mid \langle \text{keyword} \rangle \langle \text{s_expr} \rangle$$

```
: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 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 (element of) $\langle \text{identifier} \rangle$. A sort parameter can any $\langle \text{symbol} \rangle$ (which in turn, is an $\langle \text{identifier} \rangle$).

$$\langle \text{sort} \rangle ::= \text{Bool} \mid \langle \text{identifier} \rangle \mid (\langle \text{identifier} \rangle \langle \text{sort} \rangle^+)$$

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 $\langle \text{spec_const} \rangle$ category (numerals, rationals, strings, etc.), *variables*, *function symbols*, a distinguished symbol for equality and one for disequality (respectively `=` and `distinct`), three kinds of *binders*, and an annotation operator (`!`).

A variable can be any $\langle \text{symbol} \rangle$, while a function symbol can be any $\langle \text{identifier} \rangle$ (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 $(\text{as } f \ \sigma)$.

In this version, formulas are 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`.

$$\begin{aligned}
 \langle \text{qual_identifier} \rangle &::= \langle \text{identifier} \rangle \mid (\text{as } \langle \text{identifier} \rangle \langle \text{sort} \rangle) \\
 \langle \text{var_binding} \rangle &::= (\langle \text{symbol} \rangle \langle \text{term} \rangle) \\
 \langle \text{sorted_var} \rangle &::= (\langle \text{symbol} \rangle \langle \text{sort} \rangle) \\
 \langle \text{term} \rangle &::= \langle \text{spec_constant} \rangle \mid \langle \text{qual_identifier} \rangle \\
 &\quad \mid (\langle \text{qual_identifier} \rangle \langle \text{term} \rangle^+) \\
 &\quad \mid (= \langle \text{term} \rangle \langle \text{term} \rangle^+) \\
 &\quad \mid (\text{distinct } \langle \text{term} \rangle \langle \text{term} \rangle^+) \\
 &\quad \mid (\text{let } (\langle \text{var_binding} \rangle^+) \langle \text{term} \rangle) \\
 &\quad \mid (\text{forall } (\langle \text{sorted_var} \rangle^+) \langle \text{term} \rangle) \\
 &\quad \mid (\text{exists } (\langle \text{sorted_var} \rangle^+) \langle \text{term} \rangle) \\
 &\quad \mid (! \langle \text{term} \rangle \langle \text{attribute} \rangle^+)
 \end{aligned}$$

```

(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)))))))

```

In its simplest form, a term is either a special constant symbol or a variable or a function symbol applied to zero or more terms.¹ The predefined function symbols `=` and `distinct` apply to two or more terms. Informally, a term like $(= t_1 \ \dots \ t_n)$ states that t_1, \dots, t_n have all the same value; a term like $(\text{distinct } t_1 \ \dots \ t_n)$ states that t_1, \dots, t_n have pairwise distinct values.

¹ Function symbols applied to no arguments are used as constant symbols.

Binders. More complex terms include also `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

$$(\text{let } ((x_1 \ t_1) \ \cdots \ (x_n \ t_n)) \ t)$$

is equivalent to the term obtained from t by simultaneously replacing each free occurrence of x_i in t by t_i , for each $i = 1, \dots, n$, 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

$$(\text{let } ((x_1 \ t_1)) \ (\text{let } ((x_2 \ t_2)) \ t))$$

All binders follow a lexical scoping discipline, enforced by SMT-LIB logic's semantics as described in Section 4.3. Note that all variables bound by a binder are elements of the $\langle \text{symbol} \rangle$ category—they cannot be indexed identifiers.

Annotations. Every term t can be optionally annotated with one or more attributes $\alpha_1, \dots, \alpha_n$ using the wrapper expression $(! \ t \ \alpha_1 \ \cdots \ \alpha_n)$. Attributes have no logical meaning—semantically $(! \ t \ \alpha_1 \ \cdots \ \alpha_n)$ is equivalent to t —but they are a convenient mechanism for adding meta-logical information for SMT solvers. Although not part of the standard yet, examples of term annotations are instantiation patterns for quantifiers, or labels for subterms.

```
(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))
  ))

(=> (! (> x y) :label $l1)
    (! (= x z) :label $l2))
```

The former contain heuristic information useful for SMT-solvers based on quantifier instantiation. The latter, could be used to facilitate input/output interactions with a solver—whereby a solver outputs a label in place of the labeled term.

Well-sortedness requirements. As mentioned, all terms of the SMT-LIB language are additionally required to be well-sorted. Well-sortedness constraints are discussed in Section 4.2 in terms of the logic's abstract syntax.


```

(theory Core
:sorts ((Bool 0))
:fun ( (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) (ite Bool 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 two-element domain of Boolean values;
- for all sorts s in Sigma, (ite Bool s s) denotes the function that
  returns its second argument or its third depending on whether
  its first argument evaluates to true or not;
- the other function symbols of Core denote the standard Boolean operators
  as expected."
)

```

Figure 3.1: The `Core` theory declaration.

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. In the previous version of the SMT-LIB standard, a theory declaration defines both a many-sorted *signature*, i.e., a collection of sorts and sorted function symbols, and a theory with that signature. The signature is 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.

In Version 2.0, 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 now generally defines a *class* of similar theories—as opposed to a single theory as in Version 1.2.

The syntax of theory declarations 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: `:axioms`, `:definition`, `:fun`, `:fun-description`, `:notes`, `:sorts`, and `:sorts-description`. Additionally, a theory declaration can contain any number of user-defined attributes.⁽²⁾

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 informal attributes is free text, in the form of a *string* value. For instance, `:fun`, `:sorts`, and

```

(theory Integers
:sorts ((Int 0))
:funcs ((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 Integers 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 Integers as expected."
)

```

Figure 3.2: The Integers theory declaration.

:axiom attributes are formal in the sense above, while :definition, :funcs-description and :sorts-description attributes are not.

$\langle \text{sort_symbol_decl} \rangle$::=	($\langle \text{identifier} \rangle$ $\langle \text{numeral} \rangle$ $\langle \text{attribute} \rangle^*$)
$\langle \text{meta_spec_constant} \rangle$::=	NUMERAL RATIONAL STRING
$\langle \text{fun_symbol_decl} \rangle$::=	($\langle \text{spec_constant} \rangle$ $\langle \text{sort} \rangle$ $\langle \text{attribute} \rangle^*$) ($\langle \text{meta_spec_constant} \rangle$ $\langle \text{sort} \rangle$ $\langle \text{attribute} \rangle^*$) ($\langle \text{identifier} \rangle$ $\langle \text{sort} \rangle^+$ $\langle \text{attribute} \rangle^*$)
$\langle \text{par_fun_symbol_decl} \rangle$::=	$\langle \text{fun_symbol_decl} \rangle$ (par ($\langle \text{symbol} \rangle^+$) ($\langle \text{identifier} \rangle$ $\langle \text{sort} \rangle^+$ $\langle \text{attribute} \rangle^*$))
$\langle \text{theory_attribute} \rangle$::=	:sorts ($\langle \text{sort_symbol} \rangle^+$) :funcs ($\langle \text{par_fun_symbol_decl} \rangle^+$) :sorts-description $\langle \text{string} \rangle$:funcs-description $\langle \text{string} \rangle$:definition $\langle \text{string} \rangle$:axioms ($\langle \text{term} \rangle^+$) :notes $\langle \text{string} \rangle$ $\langle \text{attribute} \rangle$
$\langle \text{theory_decl} \rangle$::=	(theory $\langle \text{symbol} \rangle$ $\langle \text{theory_attribute} \rangle^+$)

```

(theory ArraysEx
:sorts ((Array 2))
:fun (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))
    (implies (distinct i j) (= (select (store a i e) j) (select a j))))
- (forall ((a (Array s1 s2)) (b (Array s1 s2)))
    (implies
      (forall ((i s1)) (= (select a i) (select b i))) (= a b)))
"
)

```

Figure 3.3: The ArraysEx theory declaration.

A theory declaration $(\text{theory } T \alpha_1 \dots \alpha_n)$ defines a *theory schema* with name T and attributes $\alpha_1, \dots, \alpha_n$. Each instance of the schema is a theory \mathcal{T}_Σ 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 provided below.

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

The value of a `:fun` attribute is a non-empty sequence of possibly parametric function symbol declarations $\langle \text{par_fun_symbol_decl} \rangle$. A (non-parametric) function symbol declaration $\langle \text{fun_symbol_decl} \rangle$ of the form $(c \ \sigma)$, where c is an element of $\langle \text{spec_constant} \rangle$, declares c to have sort σ . For convenience, it is possible to declare all the special constants in $\langle \text{numeral} \rangle$ to have sort σ by means of the function symbol declaration $(\text{NUMERAL} \ \sigma)$. The same similar can be done for the set of $\langle \text{rational} \rangle$ and $\langle \text{string} \rangle$ constants by using **RATIONAL** and **STRING**, respectively.

A (non-parametric) function symbol declaration $(f \ \sigma_1 \ \dots \ \sigma_n \ \sigma)$ with $n \geq 0$ declares a function symbol f with rank $\sigma_1 \dots \sigma_n \sigma$. Intuitively, this means that f takes as input n values of respective sort $\sigma_1, \dots, \sigma_n$, and returns a value of sort σ . On the other hand, a parametric function symbol declaration $(\text{par } (u_1 \ \dots \ u_k) (f \ \tau_1 \ \dots \ \tau_n \ \tau))$ 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, \dots, u_k with non-parametric sorts. See Section 4.4 for more details.

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

In this version, there are only three predefined annotations, all attributes with no value: `:chainable`, `:left-assoc`, and `:right-assoc`. The declaration of a theory function symbol f can be annotated with

- `:left-assoc` only if gives f rank $\sigma_1 \sigma_2 \sigma_1$,
- `:right-assoc` only if gives f rank $\sigma_1 \sigma_2 \sigma_2$,
- `:chainable` only if gives f rank $\sigma \sigma \text{Bool}$.

Then, a term of the form $(f\ t_1\ \cdots\ t_n)$ is syntactic sugar respectively for:

- $(f\ \cdots\ (f\ (f\ t_1\ t_2)\ t_3)\ \cdots\ t_n)$,
- $(f\ t_1\ (f\ t_2\ \cdots\ (f\ t_{n-1}\ t_n)\ \cdots))$,
- $(\text{and}\ (f\ t_1\ t_2)\ \cdots\ (f\ t_{n-1}\ t_n))$ where `and` is itself a symbol declared as `:left-assoc` in every theory (see Subsection 3.7.1).

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

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

(and Bool Bool Bool :left-assoc)

(< Real Real Bool :chainable)

(equiv A A Bool :chainable)
```

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.⁽³⁾

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

```

:fun 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.⁽⁴⁾ For theories that have a small set of non-parametric axioms, it might suffice to list the actual axioms, possibly separately in the `:axioms` attribute. For other theories, a mix of formal notation and natural language might be more appropriate. In the presence of parametric function symbols declarations, the definition must also specify the meaning of each instance of the declared symbol.

The optional 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 2 (Theory Declarations). The only legal theory declarations of the SMT-LIB language are those that satisfy the following restrictions.

1. They contain at least one instance of the `:sorts` attribute and exactly one instance of the `:definition` attribute².
2. Each sort symbol used in a `:fun` 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 `:fun` attributes or informally in `:sorts-description` and `:fun-description` attributes.
4. In each parametric function symbol declaration (`par (u1 ... uk) (f τ1 ... τn τ)`), any symbol that is not a previously declared sort symbol must be one of the sort parameters u_1, \dots, u_k .
5. The terms listed in an `:axioms` attributes are well-sorted closed formulas built with sort and function symbols previously declared in `:sorts`, `:fun`, `:sorts-description` and `:fun-description` attributes.

The `:fun` attribute is optional in a theory declaration because a theory might lack function symbols.³ The `:sorts` attribute, however, is not optional because sorted frameworks require the existence of at least one sort.⁽⁵⁾

² Which makes both attributes non-optional.

³ 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 symbols `=` and `distinct`, this version defines a basic core theory which is implicitly included in every other SMT-LIB theory.⁽⁶⁾ Concretely, every theory declaration is assumed to contain implicitly the `:sorts` and `:funcs` attributes of the `Core` theory declaration shown in Figure 3.1, 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 now the equality symbol `=` can be used in its place. The `if_then_else` connective of Version 1.2 is also absent for a similar reason.

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)`, and plays the role played by the `if_then_else` connective in Version 1.2. In other words, this is just the theory of the Booleans with the standard Boolean operators plus `ite`.

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 U^+ . This theory corresponds to *EU_F*, the (one-sorted) theory of equality and *uninterpreted functions* (over those function symbols). In this theory, `ite` has two ranks: `(ite Bool Bool Bool)` and `(ite Bool U U)`. A many-sorted version of EUF is obtained by instantiating `Core` with more than one nullary sort symbols—and zero or more additional functions 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.

Integers

The theory declaration of Figure 3.2 defines all theories that extend the standard theory of the (mathematical) integers to additional *uninterpreted* sort and function symbols. 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.

Arrays with extensionality

A schematic version of the theory of functional arrays with extensionality is defined in the theory declaration **ArraysEx** in Figure 3.3. Each instance gives a theory of (arbitrarily 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 $(\text{Array } \sigma_1 \ \sigma_2)$ with $\sigma_1, \sigma_2 \in S$. This includes *flat array* sorts such as

$(\text{Array Int Int}), (\text{Array Int Real}), (\text{Array Real Int}), (\text{Array Bool Int}),$

conventional *nested array* sorts such as

$(\text{Array Int } (\text{Array Int Real})),$

as well as nested sorts such as

$(\text{Array } (\text{Array Int Real}) \text{ Int}), (\text{Array } (\text{Array Int Real}) (\text{Array Real Int}))$

with an array sort in the *index position* of the outer array sort.⁽⁷⁾

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

Remark 3. 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 $\{\text{Int}, \text{Real}\}$.

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 sorts⁴, any theory sorts that are not used in a script are, for all purposes, irrelevant. It is not problem in theory either because scripts refer to logics, not directly to theories. And the language of a logic can be always 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*. Logic declarations have a similar format to theory declarations, although most of their attributes are informal.⁽⁸⁾

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

⁴ Note that sort parameters cannot occur in a formula.

$$\begin{aligned}
\langle \text{logic_attribute} \rangle &:= \begin{array}{l} \text{:theories } (\langle \text{symbol} \rangle^+) \\ | \\ \text{:language } \langle \text{string} \rangle \\ | \\ \text{:extensions } \langle \text{string} \rangle \\ | \\ \text{:values } \langle \text{string} \rangle \\ | \\ \text{:notes } \langle \text{string} \rangle \\ | \\ \langle \text{attribute} \rangle \end{array} \\
\langle \text{logic} \rangle &::= (\text{logic } \langle \text{symbol} \rangle \langle \text{logic_attribute} \rangle^+)
\end{aligned}$$

A logic declaration (`logic L $\alpha_1 \dots \alpha_n$`) defines a logic with name L and attributes $\alpha_1, \dots, \alpha_n$.

Constraint 4 (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, \dots, 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, \dots, T_n declare the same sort symbol, they give it the same arity.

When the value of the **:theories** attribute is $(T_1 \dots T_n)$, with $n > 0$, the logic refers a combination \mathcal{T} of specific instances of the theory declaration schemas T_1, \dots, T_n . The exact combination mechanism that yields \mathcal{T} is defined formally in Section 4.5. 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 4.5.

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.⁽⁹⁾ 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.

The optional **:extensions** attribute is meant to document any notational conventions, or syntactic sugar, allowed in the concrete syntax of formulas in this logic.⁽¹⁰⁾

The **:values** attribute is used to identify, for each sort σ in the logic's language a particular set of ground terms of sort σ that are to be considered as *values* in models of the logic. Intuitively, a set of values of sort σ for a model is a set of terms that denote all the elements of that sort in the model. These terms might be over a signature with additional function symbols with respect to those in the logic's language. See Section 4.3 for a more in depth explanation.

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 a logic declaration. The 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 quantifier 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 (with variables 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 `Integers` from Figure 3.2 whose signature adds to the symbols of `Integers` infinitely many 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 $(+ t \dots t)$ 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 $(- (* n t))$. The set of values consists of all numerals and all terms of the form $(- n)$ where n is a numeral other than 0.

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 some of which are (variable-length) lists. 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, prints a response, and repeats. Possible responses vary from a single symbol to a list of attributes, to complex expressions like proofs.

```

⟨command⟩ ::= ( set-logic ⟨symbol⟩ )
              | ( set-option ⟨option⟩ )
              | ( set-info ⟨attribute⟩ )
              | ( declare-sort ⟨symbol⟩ ⟨numeral⟩ )
              | ( define-sort ⟨symbol⟩ ( ⟨symbol⟩* ) ⟨sort⟩ )
              | ( declare-fun ⟨symbol⟩ ( ⟨sort⟩* ) ⟨sort⟩ )
              | ( define-fun ⟨symbol⟩ ( ⟨sorted_var⟩* ) ⟨sort⟩ ⟨term⟩ )
              | ( push ⟨numeral⟩ )
              | ( pop ⟨numeral⟩ )
              | ( assert ⟨term⟩ )
              | ( assert ⟨term⟩ ⟨symbol⟩ )
              | ( check-sat )
              | ( get-assertions )
              | ( get-proof )
              | ( get-unsat-core )
              | ( get-info ⟨info_name⟩ )
              | ( get-value ( ⟨term⟩+ ) )
              | ( get-model )
              | ( get-option ⟨keyword⟩ )
              | ( exit )

⟨script⟩ ::= ⟨command⟩*

```

The command **set-option** takes as argument expressions of the syntactic category $\langle option \rangle$ which have the same form as valued attributes. Options with the predefined keywords below have a prescribed usage and semantics. Additional, solver-specific options are also allowed.

```

⟨bin_value⟩ ::= true | false

⟨option⟩ ::= :print-success ⟨bin_value⟩
              | :expand-definitions ⟨bin_value⟩
              | :interactive-mode ⟨bin_value⟩
              | :timeout ⟨rational⟩
              | :timeout none
              | :regular-output-channel ⟨string⟩
              | :diagnostic-output-channel ⟨string⟩
              | :random-seed ⟨numeral⟩
              | :verbosity ⟨numeral⟩
              | ⟨attribute⟩

```

The command **get-info** takes as argument expressions of the syntactic category $\langle info_flag \rangle$ which are flags with the same form as keywords. The predefined flags below have a prescribed usage and semantics. Additional, solver-specific flags are also allowed.

```

⟨info_flag⟩ ::= :all-statistics
              | :decisions
              | :conflicts
              | :restarts
              | :time
              | :memory
              | :error-behavior
              | :name
              | :authors
              | :version
              | :status
              | :reason-unknown
              | :notes
              | ⟨keyword⟩

```

The possible responses are defined as follows.

```

⟨gen_response⟩ ::= unsupported | success | ( error ⟨string⟩ )
⟨status⟩       ::= sat | unsat | unknown
⟨proof⟩        ::= ⟨s_expr⟩
⟨cs_response⟩  ::= ⟨status⟩
⟨ga_response⟩  ::= ( ⟨term⟩* )
⟨gp_response⟩  ::= ⟨proof⟩
⟨guc_response⟩ ::= ( ⟨symbol⟩+ )
⟨gi_response⟩  ::= ( ⟨info_response⟩+ )
⟨gv_response⟩  ::= ( ⟨assignment⟩+ )
⟨assignment⟩   ::= ( assert ( = ⟨term⟩ ⟨term⟩ ) )

```

```

⟨error-behavior⟩ ::= immediate-exit | continued-execution
⟨reason-unknown⟩ ::= timeout | memout | incomplete
⟨info_response⟩ ::= :decisions ⟨numeral⟩
                  | :conflicts ⟨numeral⟩
                  | :restarts ⟨numeral⟩
                  | :time ⟨rational⟩
                  | :memory ⟨rational⟩
                  | :error-behavior ⟨error-behavior⟩
                  | :name ⟨string⟩
                  | :authors ⟨string⟩
                  | :version ⟨string⟩
                  | :status ⟨status⟩
                  | :reason-unknown ⟨reason-unknown⟩
                  | :notes ⟨string⟩
                  | ⟨keyword⟩

```

A full presentation of the semantics of these commands, presented in terms of abstract syntax, is given in Chapter 5. We briefly highlight here, however, several points, and then give an example.

Assertion-set stack. Conforming solvers respond to various commands by performing operations on a data structure called the assertion-set stack. This is a single global stack, where each element on the stack is a set of assertions. Assertions include both logical formulas (that is, terms of boolean type), as well as declarations and definitions of sort symbols and function symbols. Such declarations and definitions are thus local: popping an assertion set from the assertion-set stack removes all declarations and definitions contained in that set. This feature supports the removal of definitions and declarations, without recourse to *undefining* or shadowing, neither of which are supported or allowed.

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

Solver output. Solvers respond to commands with the responses defined above. General responses ⟨*gen_response*⟩ are used unless more specific responses are specified, for example for `get-info` (⟨*gi_response*⟩) or `get-model` (⟨*gm_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 (the `:regular-output-channel` and `:diagnostic-output-channel` attributes). The values of these attributes should be (double-

quote delimited) file names in the format specified by the POSIX standard.⁵ The strings `"stdout"` and `"stderr"` are reserved to refer specially to the corresponding standard process channels (not disk files of the same name).

Whitespace and responses. The following requirement is in effect for all responses: any response which is not double-quoted and not parenthesized should be followed by at least one whitespace character (for example, a newline). This will enable applications reading the solver's response output to know when an identifier (like `success`) has been completely printed. 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.

Remark 5. Unlike version 1.2 of the SMT-LIB format, the current specification does not have a separate syntactic category of benchmarks. Instead, declarative information is included in scripts via the `set-info` command. See Section 5.4.3 below for more on this.

For more on error behavior, the meanings of the various options and info names, and the semantics of additional commands like `get-unsat-core`, please see Chapter 5.

Example Scripts

We demonstrate some allowed behavior of an imagined 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 script in Figure 3.4 makes two background assertions, and then conducts two independent queries. The `get-info` command requests information on the search. The script in Figure 3.5 uses the `get-model` command to get information about a particular model of the formula which the solver has reported satisfiable.

⁵ This is the usual format adopted by all Unix-based operating systems, with `/` used as a separator for (sub)directories, etc.

```
(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 :statistics)
; (:decisions 0 :restarts 0)

(pop 1)

(push 1)

(check-sat (= x w))
; sat

(exit)
```

Figure 3.4: Example script with 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))
; ((assert (= x 0))
;  (assert (= y 1)))

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

...

(check-sat)
; sat

(get-value (a))
; ((assert (= a @0s1)))

(get-value (select 2 @0s1))
; ((assert (= (select 2 @0s1) @2s0)))

(get-value ((first @2s0) (rest @2s0)))
; ((assert (= (first @2s0) 1))
;  (assert (= (rest @2s0) nil))
; )
```

Figure 3.5: Another example script (excerpt) with solver responses in comments

Part III

Semantics

Chapter 4

SMT-LIB Logic

As mentioned, for this version of the SMT-LIB standard, the underlying logic is still a variant of many-sorted first-order logic (FOL) with equality [Man93, Gal86, End01], although it now 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 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 mapping into this abstract syntax. We start by fixing the following sets of (abstract) symbols and values:

- an infinite set \mathcal{S} of *sort symbols* s ,
- 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 ,
- an infinite set \mathcal{A} of *attribute names* a ,
- an infinite set \mathcal{V} of *attribute values* v .

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

Figure 4.1: Abstract syntax for sort terms

- 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{LN} of *logic names* L .

It is unnecessary to require that the sets above be pairwise disjoint. We do assume, however, that \mathcal{S} contains a distinguished symbol **Bool** and \mathcal{F} contains a distinguished symbol \approx for equality.

4.1 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 6 (Sorts). For all non-empty subsets S of \mathcal{S} and all mappings $ar : S \rightarrow \mathbb{N}$, 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 $\sigma_1, \dots, \sigma_n$ are sorts, $s \in S$ and $ar(s) = n$, 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 for simplicity the arity constraints, is provided in Figure 4.1. 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.

(Attributes)	$\alpha ::= a \mid a = v$
(Non-annotated Terms)	$d ::= x \mid f t^* \mid f^\sigma t^*$ $\mid \exists (x:\sigma)^+ t \mid \forall (x:\sigma)^+ t \mid \mathbf{let} (x = t)^+ \mathbf{in} t$
(Terms)	$t ::= d \mid d \alpha^*$

Figure 4.2: Abstract syntax for unsorted terms

4.2 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 4.2. 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 $x_1, \dots, x_n \in \mathcal{X}$ and sorts $\sigma_1, \dots, \sigma_n$,

- the prefix construct $(\exists x_1:\sigma_1 \cdots x_n:\sigma_n _)$ is a *sorted existential binder (or existential quantifier)* for x_1, \dots, 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, \dots, x_n ;
- the mixfix construct $(\mathbf{let} x_1 = _ \cdots x_n = _ \mathbf{in} _)$ is an *(parallel-)let binder* for x_1, \dots, 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 and \approx . Logical connectives for negation, conjunction and so on are just function symbols of the basic theory **Core**, implicitly included in all SMT-LIB theories (refer back to 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.

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

4.2.1 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 7 (SMT-LIB Signature). An *SMT-LIB signature*, or simply a *signature*, is a tuple Σ consisting of:

- a set $\Sigma^S \subseteq \mathcal{S}$ of sort symbols containing **Bool**,
- a set $\Sigma^F \subseteq \mathcal{F}$ of function symbols containing \approx ,
- a total mapping ar from Σ^S to \mathbb{N} , with $ar(\mathbf{Bool}) = 0$,
- a partial mapping from the variables \mathcal{X} to $Sort(\Sigma) := Sort(\Sigma^S)$,²
- a left-total relation³ R from $\Sigma^F \cup \{\approx\}$ to $Sort(\Sigma)^+$ such that $(\approx, \sigma\sigma\mathbf{Bool}) \in R$ 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 **Real Real Real** (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 $\bar{\sigma}\sigma_2$. Thanks to the requirement in Definition 7 that variables have at most

² Note that $Sort(\Sigma)$ is non-empty because at least one sort in Σ^S , **Bool**, has arity 0.

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.

$$\begin{array}{c}
\frac{}{\Sigma \vdash x \alpha^* : \sigma} \quad \text{if } x:\sigma \in \Sigma \\
\\
\frac{\Sigma \vdash t_1 : \sigma_1 \quad \dots \quad \Sigma \vdash t_k : \sigma_k}{\Sigma \vdash (f \ t_1 \ \dots \ t_k) \ \alpha^* : \sigma} \quad \text{if } \begin{cases} f:\sigma_1 \dots \sigma_k \sigma \in \Sigma & \text{and} \\ f:\sigma_1 \dots \sigma_k \sigma' \notin \Sigma & \text{for all } \sigma' \neq \sigma \end{cases} \\
\\
\frac{\Sigma \vdash t_1 : \sigma_1 \quad \dots \quad \Sigma \vdash t_k : \sigma_k}{\Sigma \vdash (f^\sigma \ t_1 \ \dots \ t_k) \ \alpha^* : \sigma} \quad \text{if } \begin{cases} f:\sigma_1 \dots \sigma_k \sigma \in \Sigma & \text{and} \\ f:\sigma_1 \dots \sigma_k \sigma' \in \Sigma & \text{for some } \sigma' \neq \sigma \end{cases} \\
\\
\frac{\Sigma[x_1:\sigma_1, \dots, x_{k+1}:\sigma_{k+1}] \vdash t : \mathbf{Bool}}{\Sigma \vdash (Q x_1:\sigma_1 \ \dots \ x_{k+1}:\sigma_{k+1} \ t) \ \alpha^* : \mathbf{Bool}} \quad \text{if } Q \in \{\exists, \forall\} \\
\\
\frac{\Sigma \vdash t_1 : \sigma_1 \quad \dots \quad \Sigma \vdash t_{k+1} : \sigma_{k+1} \quad \Sigma[x_1:\sigma_1, \dots, x_{k+1}:\sigma_{k+1}] \vdash t : \sigma}{\Sigma \vdash (\mathbf{let} \ x_1 = t_1 \ \dots \ x_{k+1} = t_{k+1} \ \mathbf{in} \ t) \ \alpha^* : \sigma}
\end{array}$$

Figure 4.3: Well-sortedness rules for terms

one sort, in a signature with no ambiguous function symbols every term can have at most one sort. In contrast, with an ambiguous symbol f whose different ranks are $\bar{\sigma}\sigma_1, \dots, \bar{\sigma}\sigma_n$, 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 $\sigma_1, \dots, \sigma_n$, that is, only if it is written as $f^{\sigma_i} \bar{t}$ for some $i \in \{1, \dots, n\}$.

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 $(f, \sigma_1 \dots \sigma_n \sigma)$ in $\mathcal{F} \times \text{Sort}(\Sigma)^+$, which we write as $f:\sigma_1 \dots \sigma_n \sigma$. A *sorted variable* is a pair (x, σ) in $\mathcal{X} \times \text{Sort}(\Sigma)$, which we write as $x:\sigma$. We write $f:\sigma_1 \dots \sigma_n \sigma \in \Sigma$ and $x:\sigma \in \Sigma$ to denote that f has rank $\sigma_1 \dots \sigma_n \sigma$ 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 consider signatures that conservatively expand a given signature with additional sort and function symbols or additional ranks for Σ 's function symbols. A signature Ω is a *expansion* of a signature Σ if all of the following hold: $\Sigma^S \subseteq \Omega^S$; $\Sigma^F \subseteq \Omega^F$; the sort symbols of Σ have the same arity in Σ and in Ω ; for all $x \in \mathcal{X}$ and $\sigma \in \text{Sort}(\Sigma)$, $x:\sigma \in \Sigma$ iff $x:\sigma \in \Omega$; for all $f \in \mathcal{F}$ and $\bar{\sigma} \in \text{Sort}(\Sigma)^+$, if $f:\bar{\sigma} \in \Sigma$ then $f:\bar{\sigma} \in \Omega$. In that case, Σ is a *subsignature* of Ω .

4.2.2 Well-sorted terms

Figure 4.3 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 defined by 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 4.3 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 $\text{Sort}(\Sigma)$, the integer index k in the rules is assumed ≥ 0 . The expression $\Sigma[x_1 : \sigma_1, \dots, x_{k+1} : \sigma_{k+1}]$ denotes the signature that maps x_i to sort σ_i for $i = 1, \dots, k+1$, and coincides otherwise with Σ . The rules operate over *sorting judgments* which are triples of the form $\Sigma \vdash t : \sigma$.

Definition 8 (Well-sorted Terms). For every SMT-LIB signature Σ , a term t generated by the grammar in Figure 4.2 is *well-sorted (with respect to Σ)* if $\Sigma \vdash t : \sigma$ is derivable by the sorting rules in Figure 4.3 for some sort $\sigma \in \text{Sort}(\Sigma)$. In that case, we say that *t has, or is of, sort σ* . \square

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

Definition 9 (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**. \square

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

Constraint 10. SMT-LIB scripts consider only closed formulas, or *sentences*, closed terms of sort **Bool**.⁽¹¹⁾ \square

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, \dots, x_n of respective sort $\sigma_1, \dots, \sigma_n$, can be rewritten as

$$\exists x_1:\sigma_1 \dots x_n:\sigma_n \varphi .$$

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).

4.3 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 11 (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 $\sigma \in \text{Sort}(\Sigma)$ as subset $\sigma^{\mathbf{A}}$ of A , with $\mathbf{Bool}^{\mathbf{A}} = \mathcal{B}$,

- each $f:\sigma \in \Sigma$ 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 $\sigma \in \text{Sort}(\Sigma)$, the set $\sigma^{\mathbf{A}}$ is the *extension* of σ in \mathbf{A} .⁽¹²⁾ □

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} .

4.3.1 The meaning of terms

A *Σ -valuation* into a Σ -structure \mathbf{A} is a partial mapping v from \mathcal{X} to the set of all domain elements of \mathbf{A} such that, for all $x \in \mathcal{X}$ and $\sigma \in \text{Sort}(\Sigma)$, $v(x) \in \sigma^{\mathbf{A}}$ whenever $x : \sigma \in \Sigma$. We denote by $v[x_1 \mapsto a_1, \dots, x_n \mapsto a_n]$ the valuation that maps x_i to $a_i \in \mathbf{A}$ for $i = 1, \dots, n$ and is otherwise identical to v . Note that $v[x_1 \mapsto a_1, \dots, x_n \mapsto a_n]$ need not be a Σ -valuation⁶ but it is definitely a Σ' -valuation for some variant Σ' of Σ .⁽¹³⁾

If v is a Σ -valuation into \mathbf{A} , the pair $\mathcal{I} = (\mathbf{A}, v)$ is a *Σ -interpretation*. We write $\mathcal{I}[x_1 \mapsto a_1, \dots, x_n \mapsto a_n]$ as an abbreviation for $(\mathbf{A}, v[x_1 \mapsto a_1, \dots, x_n \mapsto a_n])$. A Σ -interpretation \mathcal{I} assigns a meaning to well-sorted Σ -terms by means of a uniquely determined (total) mapping $\llbracket _ \rrbracket^{\mathcal{I}}$ of such terms into the universe of its structure.

Definition 12. 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 , $\llbracket t \rrbracket^{\mathcal{I}}$ is defined inductively as follows.

1. $\llbracket x \rrbracket^{\mathcal{I}} = v(x)$ if x is a variable
2. $\llbracket \hat{f} \ t_1 \ \dots \ t_n \rrbracket^{\mathcal{I}} = (f:\sigma_1 \cdots \sigma_n \sigma)^{\mathbf{A}}(a_1, \dots, a_n)$ if $\begin{cases} \hat{f} = f \text{ or } \hat{f} = f^{\sigma}, \\ \Sigma \vdash t_i : \sigma_i \text{ and } a_i = \llbracket t_i \rrbracket^{\mathcal{I}} \\ \text{for } i = 1, \dots, n \end{cases}$
3. $\llbracket \text{let } x_1 = t_1 \ \dots \ x_n = t_n \text{ in } t \rrbracket^{\mathcal{I}} = \llbracket t \rrbracket^{\mathcal{I}'}$ if $\begin{cases} a_i = \llbracket t_i \rrbracket^{\mathcal{I}} \text{ for } i = 1, \dots, n, \\ \mathcal{I}' = \mathcal{I}[x_1 \mapsto a_1, \dots, x_n \mapsto a_n] \end{cases}$
4. $\llbracket \exists x_1:\sigma_1 \ \dots \ x_n:\sigma_n \ t \rrbracket^{\mathcal{I}} = \mathbf{true}$ iff $\llbracket t \rrbracket^{\mathcal{I}'} = \mathbf{true}$ for some $\begin{cases} (a_1, \dots, a_n) \in \sigma_1^{\mathbf{A}} \times \cdots \times \sigma_n^{\mathbf{A}}, \\ \mathcal{I}' = \mathbf{I}[x_1 \mapsto a_1, \dots, x_n \mapsto a_n] \end{cases}$

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

⁶ Because each a_i need not be in $\sigma_i^{\mathbf{A}}$ where σ_i is the sort of x_i in Σ .

5. $\llbracket \forall x_1:\sigma_1 \cdots x_n:\sigma_n t \rrbracket^{\mathcal{I}} = \mathbf{true}$ iff $\llbracket t \rrbracket^{\mathcal{I}'} = \mathbf{true}$ for all $\begin{cases} (a_1, \dots, a_n) \in \sigma_1^{\mathbf{A}} \times \cdots \times \sigma_n^{\mathbf{A}}, \\ \mathcal{I}' = \mathbf{I}[x_1 \mapsto a_1, \dots, x_n \mapsto a_n] \end{cases}$
6. $\llbracket u \alpha_1 \cdots \alpha_n \rrbracket^{\mathcal{I}} = \llbracket u \rrbracket^{\mathcal{I}}$. □

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

For a closed term t , its meaning $\llbracket t \rrbracket^{\mathcal{I}}$ 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 $\llbracket t \rrbracket^{\mathbf{A}}$ instead of $\llbracket t \rrbracket^{\mathcal{I}}$. Similarly, for sentences, we can speak directly of a *structure* satisfying or falsifying the sentence.

Definition 13 (Isomorphism). Let be \mathbf{A} and \mathbf{B} two Σ -structures with respective universes A and B . A mapping $h : A \rightarrow B$ is an *homomorphism* from \mathbf{A} to \mathbf{B} if

1. for all $\sigma \in \Sigma^S$ and $a \in \sigma^{\mathbf{A}}$,

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

$$h((f:\sigma_1 \dots \sigma_n \sigma)^{\mathbf{A}}(a_1, \dots, a_n)) = (f:\sigma_1 \dots \sigma_n \sigma)^{\mathbf{B}}(h(a_1), \dots, h(a_n)) .$$

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.

4.4 Theories

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

Definition 14 (Theory). For any signature Σ , a *Σ -theory* is a set 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 set 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* theory, 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 sorts symbols and agree both on the arity they assign to sort symbols and on the sorts they assign to variables.⁷ 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 $\bar{\sigma} \in \text{Sort}(\Sigma)^+$, $f:\bar{\sigma} \in \Sigma$ iff $f:\bar{\sigma} \in \Sigma_1$ or $f:\bar{\sigma} \in \Sigma_2$.

Definition 15 (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 $i = 1, 2$. \square

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

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

Definition 16 (Satisfiability and Entailment Modulo a Theory). For any Σ -theory \mathcal{T} , a Σ -sentence is *satisfiable in \mathcal{T}* if it is satisfied by one of \mathcal{T} 's models. A set Γ of Σ -sentences *\mathcal{T} -entails* a Σ -sentence φ , written $\Gamma \models_{\mathcal{T}} \varphi$, if every model of \mathcal{T} that satisfies all sentences in Γ satisfies φ as well.

4.4.1 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 4.4.

Given a theory declaration T , assume first that T has no **sorts-description** and **fun-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 cardinality.

The definition provided in the **definition** attribute of a T must be such that every signature like Ω above uniquely determines an instance $T[\Omega]$ of T with signature $\hat{\Omega}$ defined as follows:

1. $\hat{\Omega}^S = \Omega^S$,
2. $\hat{\Omega}^F = F \cup \Omega^F$,

⁷ 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)	$tattr ::=$ <table> <tr> <td>sorts</td> <td>$= sdec^+$</td> <td>funs</td> <td>$= pdec^+$</td> </tr> <tr> <td></td> <td>sorts-description</td> <td>$= w$</td> <td></td> </tr> <tr> <td></td> <td>funs-description</td> <td>$= w$</td> <td></td> </tr> <tr> <td></td> <td>definition</td> <td>$= w$</td> <td>axioms $= t^+$</td> </tr> <tr> <td></td> <td>notes</td> <td>$= w$</td> <td>α</td> </tr> </table>	sorts	$= sdec^+$	fun s	$= pdec^+$		sorts-description	$= w$			fun s-description	$= w$			definition	$= w$	axioms $= t^+$		notes	$= w$	α
sorts	$= sdec^+$	fun s	$= pdec^+$																		
	sorts-description	$= w$																			
	fun s-description	$= w$																			
	definition	$= w$	axioms $= t^+$																		
	notes	$= w$	α																		
(Theory declarations)	$tdec ::= \mathbf{theory} \ T \ tattr^+$																				

Figure 4.4: Abstract syntax for theory declarations

3. no variables are sorted in $\widehat{\Omega}$,⁽¹⁴⁾
4. for all $f \in \widehat{\Omega}^F$ and $\bar{\sigma} \in (\widehat{\Omega}^S)^+$, $f:\bar{\sigma} \in \widehat{\Omega}$ iff
 - (a) $f:\bar{\sigma} \in \Omega$,
 - (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 4b or 4c above. We call the sort symbols of $\widehat{\Omega}$ that are not 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 $T[\Omega]$.⁸ This terminology is justified by the following additional requirement on T .

The definition of T 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 the interpretation of $T[\Omega]$'s free symbols is a model of $T[\Omega]$ as well.⁽¹⁵⁾

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

4.5 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} ,

⁸ Note that because of overloading we talk about *ranked* function symbols being free or not, not just function symbols.

(Logic attributes) $lattr ::= \text{theories} = T^+ \mid \text{language} = w$
 $\mid \text{extensions} = w \mid \text{values} = w$
 $\mid \text{notes} = w \mid \alpha$

(Logic declarations) $ldec ::= \text{logic } L \text{ } lattr^+$

Figure 4.5: Abstract syntax for logic declarations

- restricting the set of structures to the models of \mathcal{T} , and
- 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 if it is satisfiable in \mathcal{T} .

4.5.1 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 4.5.

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

Theory. The logic's theory is the theory \mathcal{T} uniquely determined as follows. For each $i = 1, \dots, n$, 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 of S_0 sort symbols and an additional set of ranked function symbols with ranks over $Sort(S)^+$ where $S = \bigcup_{i=0, \dots, n} S_i$. Let Ω be the smallest signature with $\Omega^S = S$ containing all those ranked function symbols. Then for each $i = 1, \dots, n$, let $T_i[\Omega]$ be the instance of T_i determined by Ω as described in defined in Subsection 4.4.1. The theory of L is

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

Note that \mathcal{T} is well defined. To start, Ω is well defined because any sort symbols shared by two declarations among T_1, \dots, T_n have the same arity in them. The theories $T_1[\Omega], \dots, T_n[\Omega]$ are well defined because Ω satisfies the requirements in Subsection 4.4.1. Finally, the signatures of $T_1[\Omega], \dots, T_n[\Omega]$ 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 \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 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} should be such that, for all $\sigma \in \Sigma^S$ and all $a \in \sigma^{\mathbf{A}}$, $\llbracket t \rrbracket^{\mathbf{A}} = a$ for some $t \in V_\sigma$. For example, in a logic of the integers, the set of

values for the integer sort might consists 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 of $\sigma^{\mathbf{A}}$. For example, in a logic of rational numbers, the set of values for the rational sort might consists 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.

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 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.⁹

⁹ The concrete syntax reserves a special format for constant symbols used as abstract values (they are members of the $\langle symbol \rangle$ category that start with the character $\textcircled{\text{O}}$).

Chapter 5

SMT-LIB Scripts

To enable finer-grained interaction of SMT solvers with other tools, such as verification tools, which wish to call them, this chapter defines a Command Language for commands to and responses from SMT solvers. The calling tool issues commands in the format of the Command Language to the SMT solver via the solver's standard textual input channel. The SMT solver then responds over two textual output channels, one for regular output and one for diagnostic output. Note that the primary goal is to support convenient interaction with a calling program, not human interaction. This has some influence on the design.

There are many other commands one might wish for an SMT solver to support, beyond those adopted here. A handful of additional commands are planned for the next point release of the standard, such as a command to obtain information about the assignment of truth values to subformulas when the solver reports a formula is satisfiable. In general, it is expected that time and more experience with the needs of applications will drive the addition of further commands, in subsequent versions.

This chapter specifies the formats for commands (and their responses) to do the following things:

- managing a stack of *assertion sets*,
- defining sorts and functions in the current assertion set,
- adding assumptions to the current assertion set,
- checking satisfiability the conjunction of all assertions in the assertion-set stack,
- obtaining further information following a satisfiability check (e.g., model information),
- setting values for standard and solver-specific options,
- getting standard and solver-specific information from the solver.

This chapter, like all those in Part [III](#) of this document, gives the semantics for commands expressed in an abstract syntax. Here, we reference syntactic categories defined in previous

(Commands)	c	$::=$	set-logic L
			set-option o
			set-info α
			declare-sort $s \ n$
			define-sort $s \ u^* \ \tau$
			declare-fun $f \ \sigma^* \ \sigma$
			define-fun $f \ (x:\sigma)^* \ \sigma \ t$
			push n
			pop n
			assert t
			assert $t \ l$
			check-sat
			get-assertions
			get-value t^+
			get-model
			get-proof
			get-unsat-core
			get-info i
			get-option a
			exit
(Scripts)	scr	$::=$	c^*

Figure 5.1: Abstract syntax for commands

chapters, in particular from Chapter 4 (e.g., terms t and sorts σ). In addition to sets of abstract symbols and values fixed in Chapter 4, we add here

- an infinite set \mathcal{L} of *formula labels* l ,
- the set \mathcal{R} of *rational numbers* r .

The concrete syntax is discussed in Chapter 3 and summarized in Appendix B.

5.1 Commands

Commands have the abstract syntax given by the grammar of Figure 5.1. A script is just a sequence of commands. The commands **push**, **pop**, **declare-sort**, **define-sort**, **declare-fun**, **define-fun**, **assert**, and **check-sat** are called *assertion-set commands*, because they operate on the assertion-set stack (explained further below).

The informal semantics of scripts for an SMT solver is given in the following sections. After a few preliminary considerations, we describe how solvers conforming to this specification should respond to the commands related to the assertion-set stack, definitions, asserting

and checking satisfiability, getting evidence, setting options, and reporting additional information.

5.1.1 Starting and terminating

Setting the logic. The command

set-logic

tells the solver what logic (in the sense of Section 4.5) is being used. For simplicity, it is an error for more than one **set-logic** command to be issued to a single running instance of the solver, and it is an error for the logic name given not to correspond to an SMT-LIB logic. The logic must be set before **define-sort**, **define-fun**, **assert**, or **check-sat** are used.

Setting solver options. The command

set-option *o*

sets a solver's option to a specific value, as specified by the argument *o*. More details on predefined options and expected behavior are provided in Section 5.3.

Terminating. The command

exit

instructs the solver to exit.

5.1.2 Modifying the Assertion-Set Stack

As mentioned above, the solver maintains a stack of sets, containing some locally scoped information: asserted formulas, declarations, and definitions. Some terminology related to this data structure is needed:

- *assertion-set stack*: the single global stack of sets.
- *assertion sets*: the sets which are the elements on the stack.
- *set of current assertions*: the union of all the assertion sets currently on the assertion-set stack.

Growing the stack. The command

push *n*

pushes *n* empty assertion sets onto this stack (typically, *n* will be 1).

Shrinking the stack. The command

pop n

pops the top n assertion sets from the stack. If n is greater than the current stack depth, an error results. If n is 0, no assertion sets are popped.

5.1.3 Declaring and defining new symbols

Four commands allow the declaration or definition of a function symbol or sort symbol. These declarations and definitions are local, in the sense that popping assertion sets removes them.⁽¹⁶⁾ So well-sortedness checks, required for commands like **assert**, are always done with respect to the *current signature*, which is determined by the logic specified by the **set-logic** command and by the set of sort symbols and rank associations (for function symbols) in the set of current assertions. It is an error for a declared or defined symbol to be shadowed by any other declared or defined symbols.

Declaring a sort symbol. The command

declare-sort $s\ n$

adds the association of arity n with sort symbol s to the current assertion set.

Defining a sort symbol. The command

define-sort $s\ u_1 \cdots u_n\ \tau$

adds the association of arity n with sort symbol s to the current assertion set. Also, subsequent well-sortedness checks must treat $s\ \sigma_1 \cdots \sigma_n$ as an abbreviation for the sort obtained by simultaneously substituting σ_i for u_i , for $i \in \{1, \dots, n\}$, in τ .⁽¹⁷⁾

Declaring a function symbol. The command

declare-fun $f\ \sigma_1 \cdots \sigma_n\ \sigma$

adds the association $f : \sigma_1 \cdots \sigma_n \sigma$ to the current assertion set.

Defining a function symbol. The command

define-fun $f\ (x_1:\sigma_1) \cdots (x_n:\sigma_n)\ \sigma\ t$

adds the association $f : \sigma_1 \cdots \sigma_n \sigma$ to the current assertion set. In addition, the logical semantics is as if the formula $\forall (x_1:\sigma_1 \cdots x_n:\sigma_n) (f\ x_1 \cdots x_n) \approx t$ had been also added to the assertion set.

5.1.4 Asserting formulas and checking satisfiability

Asserting formulas. The command

assert t

adds term t to the assertion set on the top of the assertion-set stack. The term is required to be a closed formula that is well sorted with respect to the current signature (see the previous section). The command

assert t l

has the same restrictions and effect except that it associates the formula t with a label l . Labels are scoped, and so can be reused in later calls to **assert** in another context following **pop** commands. They cannot be reused in the same context, similarly to the restrictions for definitions and declarations. The only exception is the special predefined label **background**, which can be reused for several assertions in the same context. Labels are used in support of the **get-unsat-core** command, as explained later in this section.

Inspecting asserted formulas. The command

get-assertions

causes the solver to print all currently asserted formula. The command is intended for interactive use. Indeed, it is an error to use this command when the solver is not in interactive mode. This mode may be set with **set-option** and the **interactive-mode** option; see Section 5.3 below. Supporting interactive mode is optional for solvers, and solvers electing not to support it may respond with **unsupported to get-assertions**. Note that conforming solvers are not allowed to print formulas equivalent to or derived from the current assertions; they must print exactly the set of current assertions. ⁽¹⁸⁾

Checking satisfiability. The command

check-sat

instructs the solver to check whether or not the conjunction of the current assertions is satisfiable in the logic specified with the **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 5.2) with a *csr* response:

(check-sat response) *csr* ::= **sat** | **unsat** | **unknown**

where **sat** indicates that the solver has found a model, **unsat** that the solver has established there is no model, and **unknown** that the search was inconclusive (because of time limits, solver incompleteness, etc.) If the solver reports **sat** or, optionally, if it reports **unknown**,

it should respond to the **get-model** and **keep-model** commands. It reports **unsat** it should respond to the **get-proof** and **get-unsat-core** commands.

A **check-sat** command may be followed by other **assert** and **check-sat** commands, without an intervening **pop**. In that case, the semantics is that subsequent **assert** commands are just extending the current assertion set (as it existed at the time of the **check-sat** command), and **check-sat** commands are checking satisfiability of the resulting set of current assertions. So subsequent **check-sat** commands are never handled with respect to the model possibly found by an earlier **check-sat** command.⁽¹⁹⁾

Requesting proofs. The command

get-proof

asks the solver for a proof of unsatisfiability for the set of current assertions. It can be issued only following a **check-sat** command which reports unsatisfiability, without intervening assertion-set commands. In this situation, the solver should respond by printing a refutation proof (on its regular output channel). As mentioned earlier, there is, as yet, no standard SMT-LIB proof format, so this proof will necessarily be in a solver-specific format. Solvers that do not support proof production should output **unsupported** (see Section 5.2 for more details on solver responses).

Requesting unsatisfiable cores. The command

get-unsat-core

asks the solver for an *unsat core*, a subset of the current assertions that the solver has determined to be unsatisfiable.⁽²⁰⁾ It can be issued only in the same situations in which **get-proof** can. For convenience, the solver does not return the formulas themselves but the labels that they have been associated with by **assert**. If any of the formulas in the unsat core are unlabeled (because asserted with the first form of the **assert** command), the solver should report an error. The role of the predefined label **background** is to designate certain assertions as part of a problem-specific set of premises that the calling application considers fixed. Such premises need not be reported by **get-unsat-core**, even if they are present in the unsat core.

Inspecting models. The command

get-value $t_1 \dots t_k$

where $k > 1$ and each t_i is a closed quantifier-free term, asks the solver to return for each t_i a value v_i that is equivalent to t_i in the current model.¹ Specifically, v_i has the same sort as t_i and $\llbracket t_i \rrbracket^{\mathbf{A}} = \llbracket v_i \rrbracket^{\mathbf{A}}$, where \mathbf{A} is the current model. The command can be issued only

¹ Recall that values are particular ground terms defined in a logic for each sort (see Subsection 4.5.1).

following a **check-sat** command which reports satisfiability, without intervening assertion-set commands. Solvers should report an error if the command is issued after the solver reports **unsat**. The values are returned in a sequence of asserts of the form **assert** $t_i \approx v_i$ for each $i = 1, \dots, k$.

Note that the solver does not need to describe its found model to the outside world in any way other than by providing values in the model for the given ground terms. In fact, the internal representation of the model may as well be partial, and may be extended as needed in response to successive **get-value** calls.⁽²¹⁾ The only requirement is that any two syntactically different values of the same sort returned by the solver should have different meaning in the model.²

The command

get-model

has the same effect as **get-value** $t_1 \dots t_k$ where t_1, \dots, t_k are all the closed quantifier-free (sub)terms occurring in the current set of assertions. It is intended for interactive use only and has the same requirements as **get-value** and **get-assertions**, combined.

5.2 Solver Responses, Errors, and Other Output

Regular output, including responses and errors, which is produced by conforming 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 **set-option** (see Section 5.3 below). By default they are the standard output and standard error channels, respectively.

When a solver completes its processing in response to a command, by default it should print to its standard output channel a general response:

(General response) $gr ::= \text{unsupported} \mid \text{success} \mid \text{error } w$

This default format applies to commands discussed below, unless otherwise noted. The string given to **error** may be empty (but in that case should still be present as "") or else an otherwise unspecified message describing the problem encountered. Tools communicating with an SMT solver thus can always determine when the solver has completed its processing in response to a command. Several options described in Section 5.3 below affect the printing of responses, in particular by suppressing the printing of **success**, and by redirecting the standard output.

Errors and solver state. This standard gives solvers two options when encountering errors. They may either print an error message in the above format and then immediately exit

² So, for instance, in a logic of rational numbers, the solver cannot use both the terms $1/3$ and $2/6$ as output values for **get-value**.

with a non-zero exit status; or else leave the state of the solver unmodified (by the command which encountered an error), and continue accepting commands. For the second option, the solver's state should remain unmodified by the error-generating command, except possible for timing and diagnostic information). In particular, the assertion-set stack, discussed just below (Section 5.1.2) is not modified.⁽²²⁾ The standard **error-behavior** keyword can be used with the **get-info** command to check which error behavior the tool supports (see Section 5.4 below).

Printing terms. Several commands below request the solver to print sets of terms. While some commands, naturally, place semantic requirements on these sets, we impose here the requirement that the term always must be well-sorted with respect to the current signature.

Printing defined symbols. All output from compliant solver should print defined symbols (both sort and function symbols) just as they are, without replacing them by the expression they are defined to equal. This approach generally keeps output from solvers much more compact than with definitions expanded. An option is included below (Section 5.3), however, to expand all definitions in solver output.

5.3 Solver Options

Solvers options may be set using the **set-option** command, and their current values obtained using **get-option**. If an option is not supported, in either case the solver should print **unsupported** on its regular output channel. For **get-option**, it otherwise just prints the current value of the option. Solver-specific option names are allowed and indeed expected. A set of standard names is catalogued below and in the next section. This Command Language specification requires solvers to recognize and reply in a standard way to a few of these names. The majority, however, solvers need not support, although in that case they should reply **unsupported**. These sets of names are likely to be expanded or otherwise revised as further desirable common options and kinds of information across tools are identified. For **set-option** commands, the following format is required for replies, for both solver-specific and standard options:

```
(Options)  o ::= print-success = b
              | expand-definitions = b
              | interactive-mode = b
              | timeout = r
              | timeout = none
              | regular-output-channel = w
              | diagnostic-output-channel = w
              | random-seed = n
              | verbosity = n
              |  $\alpha$ 
```

The current list of standard option names is given next, together with default values and whether or not the option must be supported by conforming solvers.

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

random-seed, default value 0, optional. The argument is a positive numeral for the solver to use as a random seed, in case the solver uses (pseudo-)randomization.

timeout, default value is -1 (meaning no timeout), optional. The argument is either -1 or a positive decimal number for the solver to use as the number of seconds before timing out, if the solver supports limiting its own runtime.

expand-definitions, default **false**, 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 should contain no defined symbols at all, only the (full expansions of the) expressions they are defined to equal.

interactive-mode, default **false**, optional. If the solver supports this option, setting it to **true** should enable the **get-assertions** command (described in Section 5.1.4 above), which otherwise may not be called.

verbosity, default 0, optional. The argument is a non-negative numeral controlling the level of diagnostic output written by the solver. All such output should be written to a secondary output channel, called the diagnostic output channel, to avoid confusion with the responses to commands which are written to standard output. These channels can be changed via the **regular-output-channel** and **diagnostic-output-channel** options below. An argument of 0 requests that no such output be produced. Higher values request more verbose output.

regular-output-channel, default **"stdout"**, required. The argument should be a filename to use subsequently for the output channel. The filename **"stdout"** is interpreted specially to mean the solver's standard output channel, and similarly, **"stderr"** means the solver's standard error channel.

diagnostic-output-channel, default **"stderr"**, required. The argument should be a filename to use subsequently for the diagnostic output channel.

5.4 Getting Additional Information With **get-info**

The format for responses *gir* to **get-info** commands, for both solver-specific and standard information names, is given in Figure 5.2. The different information names and more specific

(get-info response)	<i>gir</i>	::=	i^+
(Info response)	<i>i</i>	::=	decisions = n conflicts = n restarts = n time = r memory = r error-behavior = eb name = w authors = w^+ version = w status = csr reason-unknown = ru notes = w α
(Error behavior)	<i>eb</i>	::=	immediate-exit continue-execution
(Reason unknown)	<i>ru</i>	::=	timeout memout incomplete
(check-sat response)	<i>csr</i>	::=	sat unsat unknown
(get-assertions response)	<i>gar</i>	::=	t^*
(get-proof response)	<i>gpr</i>	::=	p
(get-unsat-core response)	<i>gucr</i>	::=	s^+

Figure 5.2: Abstract syntax for info responses

formats of the info responses are given next. First we discuss statistics, then some additional pieces of information.

5.4.1 Standard Names for Statistics for get-info

The **all-statistics** name may be given to **get-info** to get various solver statistics. Supporting it is optional. Solvers reply with a sequence of info responses i , giving statistics on the most recent **check-sat** command. It is required that this **check-sat** command is one without any intervening assertion-set commands. Each statistic's name below may be given as an information name in its own right to **get-info**. The format for the reply to **all-statistics** is just the aggregation of the replies for each individual statistic. A solver may reply to **all-statistics** giving possibly just some of the statistics listed next, and possibly some additional solver-specific ones.

decisions, optional. This, like many other kinds of information, is what we shall call a *singleton*: solvers reply with just one key-value pair, where the key is again the name of the piece of information. For **decisions**, the value is a non-negative numeral giving the number of decisions made during search.

conflicts, optional. This is a singleton, where the value is the number of conflicts encountered during search.

restarts, optional. This is a singleton, where the value is the number of restarts performed during search.

time, optional. This is a singleton, where the value is the running time in seconds, as a decimal number, taken during search.

memory, optional. This is a singleton, where the value is the memory in megabytes, as a decimal number, allocated during search.

5.4.2 Additional Standard Names for **get-info**

Some further standard names for **get-info** are given here. A few of these may be set by the **set-info** command. If not indicated, they may not be set with **set-info**, and attempting to do so results in an error. Support for setting values for solver-specific names with **set-info** is optional.

error-behavior, required. If the response is **immediate-exit**, the solver is stating that it will exit immediately when an error is encountered. If the response is **continued-execution**, the solver is stating it will leave the state unmodified by the erroneous command, and continue accepting and executing new commands. See Section 5.2 above for more on the motivation for these two error behaviors.

name, required. A singleton, where the value is the name of the solver.

version, required. A singleton, where the value is the version number of the solver (e.g., 1.2).

authors, required. A singleton, where the value is the list of names of the tool's authors.

status, required, may be set with **set-info**. The status of the most recent **check-sat** command or the value most recently set for **status** with **set-info**, whichever is more current.

reason-unknown, optional. If the status of the most recent **check-sat** command is **unknown**, this gives a short reason why the solver could not successfully check satisfiability, from the following options: **timeout**, for exceeding a time limit, if one is currently set; **memout**, for out of memory; or **incomplete**, if the solver knows it is incomplete for the class of formulas containing the most recent query.

notes, required, may be set with **set-info**. Any notes about the current (or, if set by **set-info**, future) set of asserted formulas.

5.4.3 A Note on Benchmarks

The previous SMT-LIB Formula Language (version 1.2) includes a format for benchmarks. The SMT-LIB initiative has collected a large number of benchmarks in this format, for a variety of different logics. These benchmarks are used in the SMT research community for standardized comparison of solvers, as well as for the SMT Competition (SMT-COMP) [BdMS07].

This document does not include a separate syntactic category for benchmarks, for two main reasons. First, there are many more kinds of behaviors allowed by this Command Language than by version 1.2 benchmarks. It is reasonable to expect that researchers will be interested in comparing solver performance across this wider set of possible behaviors. Second, benchmarks as they are defined in version 1.2 are subsumed by scripts, as we now explain.

Version 1.2 benchmarks can be viewed as scripts falling into a particular restricted class, making use of the **set-info** command to include some declarative information. The restrictions, summarized from Sections 5 and 7 of the version 1.2 specification, are as follows:

- The (single) **set-logic** command setting the benchmark's logic is the first command.
- There is exactly one **check-sat** command.
- There is at most one **set-info** command for **status**.
- The formulas in the script belong to the benchmark's logic, with any free symbols declared in the script.
- Extra symbols are declared exactly once before any use, and are part of the allowed signature expansion for the logic.
- The only other allowed commands are **assert**, **declare-sort**, **declare-fun**, and **check-sat**.

Part IV

Appendices

Appendix A

Notes

- ¹ 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.
- ² The rationale for allowing user-defined attributes is the same as in other attribute-value-based language (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 continued use of the SMT-LIB format will make certain user-defined attributes widely used. Those attributes might then be officially introduced in the format (as non-user-defined attributes) in later versions.
- ³ See the point made in Note 4.
- ⁴ 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.
- ⁵ This is no real restriction of course because, for instance, any unsorted theory can be modeled as at least one-sorted.
- ⁶ The theory declaration `Empty` in Version 1.2 of SMT-LIB is superseded by the `Core` theory declaration schema.
- ⁷ One advantage defining instances of theory declaration schemas as in this version 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.
- ⁸ The reason for informal attributes is similar to that for theory declarations.
- ⁹ The attribute is text valued because it has mostly 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.
- ¹⁰ This is useful because in common practice the syntax of a logic is often extended for convenience with syntactic sugar.

-
- 11 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.
 - 12 Distinct sorts can have non-disjoint extensions in a structure. However, whether they do that or not is irrelevant in SMT-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, every formula satisfiable in a structure where two given sorts have non-disjoint extensions is satisfiable in an isomorphic one where the sorts have disjoint extensions.
 - 13 This is technicality necessary to make the function $\llbracket \cdot \rrbracket$ well-defined. It is a consequence of the fact that while all variables are assigned a sort a signature Σ , quantifiers in a Σ -formulas can freely change the sort of their bound variables. This means, for instance that if $\exists x:\sigma \varphi$ is a formula of some signature Σ , its subformula φ is, for typing *and* semantic purposes, a formula of signature $\Sigma[x:\sigma]$.
 - 14 This requirement is for concreteness. Again, since we work with closed formulas, which internally assign sorts to variables, the sorting of variables in a signature is irrelevant.
 - 15 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 intuition is again that the definition of the class should impose no constraints on the interpretation of free sort symbols and free function symbols.
 - 16 It is desirable to have the ability to remove declarations and definitions, for example if they are no longer needed at some 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 making declarations and definitions locally scoped supports removal by popping the containing assertion set. Other approaches, such as the ability to add shadowing declarations or definitions of symbols, or to “undefine” or “undeclare” them, impose some issues: for example, how to print symbols that have been shadowed, undefined or undeclared.
 - 17 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.
 - 18 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.
 - 19 This restriction is for ease of solver implementation.
 - 20 Unsat cores are verify useful for applications because the circumscribe the source of unsatisfiability in the asserted set.
 - 21 In particular, there is not requirement that different permutations of the same set of **get-value** calls produce the same value for the input terms.
 - 22 The motivation for specifying these two modes in this document is that the first mode (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.

Appendix B

Concrete Syntax

Predefined symbols

!, \neg , =, as, assert, background, Bool, check-sat, continued-execution, declare-sort, declare-fun, define-sort, define-fun, error, exists, exit, false, forall, get-assertions, get-info, get-model, get-proof, get-unsat-core, get-value, immediate-exit, incomplete, let, logic, none, NUMERAL, memout, par, pop, push, RATIONAL, sat, success, set-logic, set-info, set-option, STRING, theory, timeout, true, unknown, unsupported, unsat.

Predefined keywords

:all-statistics, :author, :axioms, :chainable, :conflicts :decisions :definition, :error-behavior :expand-definitions, :extensions, :family, :funs, :funs-description, :interactive-mode, :language, :left-assoc, :logic, :memory :name, :notes, :print-success, :random-seed, :reason-unknown, :regular-output-channel, :restarts :right-assoc, :script, :sorts, :sorts-description, :status, :theories, :time :values, :diagnostic-output-channel, :verbosity, :version.

Tokens

$\langle \text{numeral} \rangle$	$::=$	$0 \mid \text{a non-empty sequence of digits not starting with } 0$
$\langle \text{rational} \rangle$	$::=$	$\langle \text{numeral} \rangle . 0^* \langle \text{numeral} \rangle$
$\langle \text{hexadecimal} \rangle$	$::=$	$\#x \text{ followed by a non-empty sequence of digits and letters from } A \text{ to } F, \text{ capitalized or not}$
$\langle \text{binary} \rangle$	$::=$	$\#b \text{ followed by a non-empty sequence of } 0\text{s and } 1\text{s}$
$\langle \text{string} \rangle$	$::=$	$\text{ASCII character string in double quotes with C-style escaped characters: } \backslash", \backslash n, \dots$
$\langle \text{symbol} \rangle$	$::=$	$\text{a non-empty sequence of letters, digits and the characters } + - / * = \% ? ! . \$ _ \sim \& \wedge < > @ \text{ that does not start with a digit}$ $\mid \text{a sequence of printable ASCII characters that starts and ends with } \text{ and does not otherwise contain } $
$\langle \text{keyword} \rangle$	$::=$	$: \text{ followed by a non-empty sequence of letters, digits and the characters } + - / * = \% ? ! . \$ _ \sim \& \wedge < > @$

Members of the $\langle \text{symbol} \rangle$ 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

$\langle \text{spec_const} \rangle$	$::=$	$\langle \text{numeral} \rangle \mid \langle \text{rational} \rangle \mid \langle \text{hexadecimal} \rangle \mid \langle \text{binary} \rangle \mid \langle \text{string} \rangle$
$\langle \text{s_expr} \rangle$	$::=$	$\langle \text{spec_constant} \rangle \mid \langle \text{symbol} \rangle \mid \langle \text{keyword} \rangle \mid (\langle \text{s_expr} \rangle^*)$

Identifiers

$\langle \text{identifier} \rangle$	$::=$	$\langle \text{symbol} \rangle \mid (_ \langle \text{symbol} \rangle \langle \text{numeral} \rangle^+)$
-------------------------------------	-------	--

Sorts

$\langle \text{sort} \rangle$	$::=$	$\text{Bool} \mid \langle \text{identifier} \rangle \mid (\langle \text{identifier} \rangle \langle \text{sort} \rangle^+)$
-------------------------------	-------	---

Attributes

$\langle \text{attribute} \rangle$	$::=$	$\langle \text{keyword} \rangle \mid \langle \text{keyword} \rangle \langle \text{s_expr} \rangle$
------------------------------------	-------	---

Terms

```

⟨qual_identifier⟩ ::= ⟨identifier⟩ | ( as ⟨identifier⟩ ⟨sort⟩ )
⟨var_binding⟩    ::= ( ⟨symbol⟩ ⟨term⟩ )
⟨sorted_var⟩     ::= ( ⟨symbol⟩ ⟨sort⟩ )
⟨term⟩           ::= ⟨spec_constant⟩ | ⟨qual_identifier⟩
                  | ( ⟨qual_identifier⟩ ⟨term⟩+ )
                  | ( = ⟨term⟩ ⟨term⟩+ )
                  | ( distinct ⟨term⟩ ⟨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 | RATIONAL | 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⟩
                  | :axioms ( ⟨term⟩+ )
                  | :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⟩+ )

```

Command options

```

⟨bin_value⟩  ::=  true | false
⟨option⟩     ::=  :print-success ⟨bin_value⟩
                  | :expand-definitions ⟨bin_value⟩
                  | :interactive-mode ⟨bin_value⟩
                  | :timeout ⟨rational⟩
                  | :timeout none
                  | :regular-output-channel ⟨string⟩
                  | :diagnostic-output-channel ⟨string⟩
                  | :random-seed ⟨numeral⟩
                  | :verbosity ⟨numeral⟩
                  | ⟨attribute⟩

```

Info flags

```

⟨info_flag⟩  ::=  :all-statistics
                  | :decisions
                  | :conflicts
                  | :restarts
                  | :time
                  | :memory
                  | :error-behavior
                  | :name
                  | :authors
                  | :version
                  | :status
                  | :reason-unknown
                  | :notes
                  | ⟨keyword⟩

```

Commands

```

⟨command⟩ ::= ( set-logic ⟨symbol⟩ )
              | ( set-option ⟨option⟩ )
              | ( set-info ⟨attribute⟩ )
              | ( declare-sort ⟨symbol⟩ ⟨numeral⟩ )
              | ( define-sort ⟨symbol⟩ ( ⟨symbol⟩* ) ⟨sort⟩ )
              | ( declare-fun ⟨symbol⟩ ( ⟨sort⟩* ) ⟨sort⟩ )
              | ( define-fun ⟨symbol⟩ ( ⟨sorted_var⟩* ) ⟨sort⟩ ⟨term⟩ )
              | ( push ⟨numeral⟩ )
              | ( pop ⟨numeral⟩ )
              | ( assert ⟨term⟩ )
              | ( assert ⟨term⟩ ⟨symbol⟩ )
              | ( check-sat )
              | ( get-assertions )
              | ( get-proof )
              | ( get-unsat-core )
              | ( get-info ⟨info_name⟩ )
              | ( get-value ( ⟨term⟩+ ) )
              | ( get-model )
              | ( get-option ⟨keyword⟩ )
              | ( exit )
⟨script⟩   ::= ⟨command⟩*

```


Command responses

```

⟨gen_response⟩ ::= unsupported | success | ( error ⟨string⟩ )
⟨status⟩       ::= sat | unsat | unknown
⟨proof⟩        ::= ⟨s_expr⟩
⟨cs_response⟩  ::= ⟨status⟩
⟨ga_response⟩  ::= ( ⟨term⟩* )
⟨gp_response⟩  ::= ⟨proof⟩
⟨guc_response⟩ ::= ( ⟨symbol⟩+ )
⟨gi_response⟩  ::= ( ⟨info_response⟩+ )
⟨gv_response⟩  ::= ( ⟨assignment⟩+ )
⟨assignment⟩   ::= ( assert ( = ⟨term⟩ ⟨term⟩ ) )
⟨error-behavior⟩ ::= immediate-exit | continued-execution
⟨reason-unknown⟩ ::= timeout | memout | incomplete
⟨info_response⟩ ::=
| :decisions ⟨numeral⟩
| :conflicts ⟨numeral⟩
| :restarts ⟨numeral⟩
| :time ⟨rational⟩
| :memory ⟨rational⟩
| :error-behavior ⟨error-behavior⟩
| :name ⟨string⟩
| :authors ⟨string⟩
| :version ⟨string⟩
| :status ⟨status⟩
| :reason-unknown ⟨reason-unknown⟩
| :notes ⟨string⟩
| ⟨keyword⟩

```

Appendix C

Abstract Syntax

Common Notation

$b \in \mathcal{B}$,	the set of booleans	$r \in \mathcal{Q}$,	the set of rationals
$n \in \mathcal{N}$,	the set of numerals	$w \in \mathcal{W}$,	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
$T \in \mathcal{T}$,	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) $\alpha ::= a \mid a = v$

(Non-annotated Terms) $d ::= x \mid f \ t^* \mid f^\sigma \ t^*$
 $\mid \exists (x:\sigma)^+ \ t \mid \forall (x:\sigma)^+ \ t \mid \mathbf{let} \ (x = t)^+ \ \mathbf{in} \ t$

(Terms) $t ::= d \mid d \ \alpha^*$

Well-sorting rules for terms

$$\begin{array}{c}
\frac{}{\Sigma \vdash x \alpha^* : \sigma} \quad \text{if } x:\sigma \in \Sigma \\
\\
\frac{\Sigma \vdash t_1 : \sigma_1 \quad \dots \quad \Sigma \vdash t_k : \sigma_k}{\Sigma \vdash (f \ t_1 \ \dots \ t_k) \ \alpha^* : \sigma} \quad \text{if } \begin{cases} f:\sigma_1 \dots \sigma_k \sigma \in \Sigma & \text{and} \\ f:\sigma_1 \dots \sigma_k \sigma' \notin \Sigma & \text{for all } \sigma' \neq \sigma \end{cases} \\
\\
\frac{\Sigma \vdash t_1 : \sigma_1 \quad \dots \quad \Sigma \vdash t_k : \sigma_k}{\Sigma \vdash (f^\sigma \ t_1 \ \dots \ t_k) \ \alpha^* : \sigma} \quad \text{if } \begin{cases} f:\sigma_1 \dots \sigma_k \sigma \in \Sigma & \text{and} \\ f:\sigma_1 \dots \sigma_k \sigma' \in \Sigma & \text{for some } \sigma' \neq \sigma \end{cases} \\
\\
\frac{\Sigma[x_1:\sigma_1, \dots, x_{k+1}:\sigma_{k+1}] \vdash t : \mathbf{Bool}}{\Sigma \vdash (Q x_1:\sigma_1 \ \dots \ x_{k+1}:\sigma_{k+1} \ t) \ \alpha^* : \mathbf{Bool}} \quad \text{if } Q \in \{\exists, \forall\} \\
\\
\frac{\Sigma \vdash t_1 : \sigma_1 \quad \dots \quad \Sigma \vdash t_{k+1} : \sigma_{k+1} \quad \Sigma[x_1:\sigma_1, \dots, x_{k+1}:\sigma_{k+1}] \vdash t : \sigma}{\Sigma \vdash (\mathbf{let} \ x_1 = t_1 \ \dots \ x_{k+1} = t_{k+1} \ \mathbf{in} \ t) \ \alpha^* : \sigma}
\end{array}$$

Theories

(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)	$tattr ::= \begin{array}{l} \mathbf{sorts} = sdec^+ \mid \mathbf{funs} = pdec^+ \\ \mid \mathbf{sorts-description} = w \\ \mid \mathbf{funs-description} = w \\ \mid \mathbf{definition} = w \mid \mathbf{axioms} = t^+ \\ \mid \mathbf{notes} = w \mid \alpha \end{array}$
(Theory declarations)	$tdec ::= \mathbf{theory} \ T \ tattr^+$

Logics

(Logic attributes) $lattr ::= \text{theories} = T^+ \mid \text{language} = w$
 $\mid \text{extensions} = w \mid \text{values} = w$
 $\mid \text{notes} = w \mid \alpha$

(Logic declarations) $ldec ::= \text{logic } L \text{ } lattr^+$

Command options and info names

(Options) $o ::= \text{print-success} = b$
 $\mid \text{expand-definitions} = b$
 $\mid \text{interactive-mode} = b$
 $\mid \text{timeout} = r$
 $\mid \text{timeout} = \text{none}$
 $\mid \text{regular-output-channel} = w$
 $\mid \text{diagnostic-output-channel} = w$
 $\mid \text{random-seed} = n$
 $\mid \text{verbosity} = n$
 $\mid \alpha$

(Info names) $i ::= \text{all-statistics}$
 $\mid \text{decisions}$
 $\mid \text{conflicts}$
 $\mid \text{restarts}$
 $\mid \text{time}$
 $\mid \text{memory}$
 $\mid \text{error-behavior}$
 $\mid \text{name}$
 $\mid \text{authors}$
 $\mid \text{version}$
 $\mid \text{status}$
 $\mid \text{reason-unknown}$
 $\mid \text{notes}$
 $\mid a$

Commands

(Commands)	c	$::=$	set-logic L set-option o set-info α declare-sort $s\ n$ define-sort $s\ u^*\ \tau$ declare-fun $f\ \sigma^*\ \sigma$ define-fun $f\ (x:\sigma)^*\ \sigma\ t$ push n pop n assert t assert $t\ l$ check-sat get-assertions get-value t^+ get-model get-proof get-unsat-core get-info i get-option a exit
(Scripts)	scr	$::=$	c^*

Command responses

(General response)	gr	$::=$	unsupported success error w
(get-info response)	gir	$::=$	i^+
(Info response)	i	$::=$	decisions = n conflicts = n restarts = n time = r memory = r error-behavior = eb name = w authors = w^+ version = w status = csr reason-unknown = ru notes = w α
(Error behavior)	eb	$::=$	immediate-exit continue-execution
(Reason unknown)	ru	$::=$	timeout memout incomplete
(check-sat response)	csr	$::=$	sat unsat unknown
(get-assertions response)	gar	$::=$	t^*
(get-proof response)	gpr	$::=$	p
(get-unsat-core response)	$gucr$	$::=$	s^+
(get-value responses)	gvr	$::=$	(assert $(t \approx t)$) $^+$

Part V

References

Bibliography

- [BBC⁺05] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. 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] C. Barrett, L. de Moura, and A. 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.
- [BdMS07] C. Barrett, L. de Moura, and A. Stump. Design and Results of the 2nd Annual Satisfiability Modulo Theories competition (SMT-COMP 2006). *Formal Methods in System Design*, 31(3):221–239, 2007.
- [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.
- [BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2010.
- [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.