The SMT-LIB Standard

Version 2.6

Clark Barrett    Pascal Fontaine    Cesare Tinelli

Release: 2021-05-12
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 2.6 of the SMT-LIB Standard which is a backward-compatible extension of Version 2.5.
We would like to thank the people below for their feedback, suggestions and corrections on this document. It is understood that we, the authors, are to blame for any remaining errors and omissions.

**Version 2.0**

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

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

Thanks also go to David Cok, Morgan Deters, Anders Franzén, Amit Goel, Jochen Hoenicke, and Tjark Weber for additional feedback on the standard, and to Jochen Hoenicke, Philipp Rümmer, and above all David Cok, for their careful proof-reading of earlier versions of the Version 2.0 document.

**Version 2.5**

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

Special thanks go to the following people for their thoughtful feedback and useful suggestions: Martin Brain, Adrien Champion, Jürgen Christ, David Cok, Morgan Deters, Bruno
Dutertre, Andrew Gacek, Alberto Griggio, Jochen Hoenicke, Tim King, Tianyi Liang, Aina Niemetz, Margus Veanes, and Christoph Wintersteiger. We also thank David Cok and Jürgen Christ for their very careful proof reading of this document.

Version 2.6

The main difference between Version 2.6, the current one, and Version 2.5 is the addition of algebraic datatypes to the language. Special thanks go to the following people for their feedback and useful suggestions on this extension: Guillaume Bury, David Cok, Tim King, Viktor Kuncak, Jochen Hoenicke, Pierre van de Laar, Calvin Loncaric, Andres Nötzli, Andrew Reynolds, and Cristina Serban.
Contents

Preface 2
Acknowledgments 3
Contents 5
List of Figures 8

I Introduction 9

1 General Information 10
1.1 About This Document ........................................... 10
1.1.1 Change log for Version 2.6 ................................ 10
1.1.2 Differences between Version 2.6 and Version 2.5 .......... 11
1.1.3 Change log for Version 2.5 ................................. 11
1.1.4 Differences between Version 2.5 and Version 2.0 .......... 11
1.1.5 Typographical and notational conventions ................. 13
1.2 Overview of SMT-LIB ........................................... 13
1.2.1 What is SMT-LIB? ........................................... 14
1.2.2 Main features of the SMT-LIB standard ................. 14

2 Basic Assumptions and Structure 16
2.1 Satisfiability Modulo Theories ............................... 16
2.2 Underlying Logic .............................................. 17
2.3 Background Theories .......................................... 17
2.4 Input Formulas ................................................. 18
2.5 Interface ...................................................... 18
## II Syntax

3 The SMT-LIB Language

3.1 Lexicon ................................................. 21
3.2 S-expressions .......................................... 24
3.3 Identifiers ............................................. 25
3.4 Attributes .............................................. 25
3.5 Sorts ...................................................... 26
3.5.1 Ranks ................................................. 26
3.6 Terms and Formulas ................................. 26
3.6.1 Variable Binders ................................. 28
3.6.2 Parameter Binders ............................... 29
3.6.3 Scoping of variables and parameters ............. 30
3.6.4 Well-sortedness requirements ................... 31
3.6.5 Annotations ......................................... 31
3.6.6 Term attributes ..................................... 31
3.7 Theory Declarations ................................. 32
3.7.1 Examples ............................................ 36
3.8 Logic Declarations ................................. 40
3.8.1 Examples ............................................ 42
3.9 Scripts .................................................. 44
3.9.1 Command responses ............................... 46
3.9.2 Example scripts ..................................... 48
3.9.3 SMT-LIB Benchmarks ......................... 49

## III Semantics

4 Operational Semantics of SMT-LIB ............................ 51
4.1 General Requirements .................................. 51
4.1.1 Execution Modes .................................. 52
4.1.2 Solver responses ................................... 53
4.1.3 Printing of terms and defined symbols ............. 54
4.1.4 The assertion stack ................................ 54
4.1.5 Symbol declarations and definitions ............... 55
4.1.6 In-line definitions ................................... 55
4.1.7 Solver options ....................................... 55
4.1.8 Solver information .................................. 57
4.2 Commands ............................................... 58
4.2.1 (Re)starting and terminating ................. 59
4.2.2 Modifying the assertion stack .................... 59
4.2.3 Introducing new symbols ....................... 60
4.2.4 Asserting and inspecting formulas ............... 63
4.2.5 Checking for satisfiability ....................... 64
4.2.6 Inspecting models ............................... 64
CONTENTS

4.2.7 Inspecting proofs ........................................ 66
4.2.8 Inspecting settings ..................................... 67
4.2.9 Script information ...................................... 67

5 Logical Semantics of SMT-LIB Formulas ............ 69
  5.1 The language of sorts .................................. 70
  5.2 The language of terms .................................. 71
    5.2.1 Signatures .......................................... 72
    5.2.2 Well-sorted terms .................................. 74
  5.3 Structures and Satisfiability .......................... 76
    5.3.1 The meaning of terms .............................. 78
  5.4 Theories .................................................. 80
    5.4.1 Combined Theories ................................ 80
    5.4.2 Theory declarations ............................... 81
  5.5 Logics ...................................................... 82
    5.5.1 Logic declarations ................................. 83

IV References .................................................. 85

Bibliography .................................................. 86

V Appendices .................................................. 88
  A Notes ....................................................... 89
  B Concrete Syntax .......................................... 94
  C Abstract Syntax .......................................... 101
# List of Figures

3.1 Theory declarations .................................................. 33  
3.2 The Core theory declaration ......................................... 37  
3.3 A possible theory declaration for the integer numbers .......... 38  
3.4 The ArraysEx theory declaration ................................... 39  
3.5 A possible declaration for a theory of sets and relations ....... 41  
3.6 SMT-LIB Commands .................................................... 45  
3.7 Command options ...................................................... 46  
3.8 Info flags ............................................................... 46  
3.9 Command responses ................................................... 47  
3.10 Example script (over two columns), with expected solver responses in comments 48  
3.11 Another example script (excerpt), with expected solver responses in comments 49  
4.1 Abstract view of transitions between solver execution modes. The symbol * here stands for the matching wildcard .................. 52  
5.1 Abstract syntax for sort terms ...................................... 70  
5.2 Abstract syntax for unsorted terms ................................ 71  
5.3 Well-sortedness rules for terms ...................................... 75  
5.4 Abstract syntax for theory declarations ........................... 81  
5.5 Abstract syntax for logic declarations ............................. 82
Part I

Introduction
1.1 About This Document

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

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

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

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

1.1.1 Change log for Version 2.6

Release: 2021-05-12

- Fixed misleading example of usage of as to disambiguate symbol sorts.
1.1. ABOUT THIS DOCUMENT

Release: 2021-04-02

- Now allowing reserved words in s-expression (as arguments of attributes).
- Disambiguated description of in-line definition (:named annotation).

Release: 2017-07-18

- First release of Version 2.6.

1.1.2 Differences between Version 2.6 and Version 2.5

The SMT-LIB 2 language and logic now supports user-defined algebraic datatypes. Such types can be introduced with two new commands: declare-datatype, to declare a single algebraic datatype, and declare-datatypes, to declare two or more mutually recursive datatypes (see Section 4.2.3). The language of terms now includes a new binder, match, for pattern-matching-based case analysis over datatype values (see Section 3.6.1). The variables bound by match are those that occur in patterns, also a new addition to the language. The underlying logic has also been modified to provide built-in semantics for the constructor, selector and tester symbols defined with each new algebraic datatype (see Chapter 5).

This version also makes official a number of set-info attributes used in benchmarks from the official SMT-LIB repository and specifies some requirements on their occurrence and order (see Section 3.9.3 and Section 4.2.9). Any other differences in this document are only edits to improve the presentation. Except for the addition of algebraic datatypes, which is fully backward compatible, the rest of the SMT-LIB language is unchanged.

1.1.3 Change log for Version 2.5

Release: 2015-06-28

- Clarified in Section 4.1.1 that the exit command can be issued in any mode.

Release: 2015-05-28

- First release of Version 2.5.

1.1.4 Differences between Version 2.5 and Version 2.0

Version 2.5 is an extension of Version 2.0 and, with two minor exceptions, is fully backward compatible with it. There is then no need to have separate support for 2.0 if one supports Version 2.6. The following list summarizes notable differences and extensions. The first two items are the only non-backward compatible changes.

- There is now a different set of escape sequences for string literals. It consists of a single sequence, "", used to represent the double quote character within the literal.
- The predefined option :expand-definitions has been removed because there are now no cases in which it applies.
• SMT-LIB source files are not limited to the US-ASCII format anymore and can now consist of Unicode characters. The concrete encoding is currently left unspecified, but should be a compatible 8-bit extension of the 7-bit US-ASCII set, such as UTF-8.

• We have clarified several points about the character set used by the SMT-LIB language and specified more precisely which characters are allowed in string literals, identifiers and symbols.

• We have made explicit several details on the scoping and shadowing rules for identifiers, in particulars those occurring in binders.

• Identifiers can now be indexed not just with numerals but also with symbols.

• The use of the term attribute :pattern and its related syntax for quantifier patterns has been made official.

• The solver option :interactive-mode has been renamed :produce-assertions. The old name is still accepted but its use is now deprecated.

• There is now a predefined argument, ALL, for the set-logic command which refers to the most general logic supported by the solver executing the command.

• We have introduced a notion of execution mode for a solver to better describe the restriction on when commands can be executed or options set.

• There is a new solver option :global-declarations that makes all definitions and declarations global and not removable by pop operations. Global declarations can be removed only by the new command reset.

• The new command reset brings the state of a solver to the state it had immediately after start up (resetting everything).

• The new command reset-assertions empties the assertion stack and removes all assertions. If :global-declarations is set to false, it also removes all declarations and definitions.

• The new command check-sat-assuming checks the satisfiability of the current context under an additional number of assumptions provided as input to the command. When it returns unsat, a new companion command, get-unsat-assumptions, returns the subset of input assumptions used by the solver to prove the context unsatisfiable. The latter command is enabled or disabled with the new option :produce-unsat-assumptions. The old check-sat command can now be defined, conservatively, as a special case of check-sat-assuming with an empty set of assumptions.

• The new command declare-const can now be used to declare nullary function symbols.

• The new command echo prints back on the regular output channel a string provided as input.
1.2. OVERVIEW OF SMT-LIB

- The new commands `define-fun-rec` and `define-funs-rec` respectively allow the definition of recursive functions and of sets of mutually recursive functions.

- The new command `get-model` returns a representation of a model computed by the solver in response to an invocation of the `check-sat` or `check-sat-assuming` command.

- The new `get-info` flag `:assertion-stack-levels` returns the current number of levels in the assertion stack.

- The new option `:reproducible-resource-limit` can be used to set a solver-defined resource limit that applies to each invocation of `check-sat` or `check-sat-assumptions`.

1.1.5 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 `<term>`. In the production rules, the meta-operators `::=` and `|` are used as usual in BNF. Also, as usual, the meta-operators `*_` and `+_` denote zero, respectively, one, or more repetitions of their argument. We use `*_n` and `+_n+1` instead of `*_` and `+_` when we want to indicate that multiple occurrences to the latter operators have the same number of repetitions. We use the notation `d_{dec}` (resp., `e_{hex}`) to represent the Unicode character with decimal code `d` (resp., hexadecimal code `e`). Remember that the US-ASCII character with code `d < 128` is also the Unicode character `d_{dec}`. Examples of concrete syntax expressions are provided in shaded boxes like the following.

\[
(f (- x) x)
\]

In the abstract syntax notation, which uses the same meta-operators as the concrete syntax, words in **boldface** as well as the symbols `≈`, `∃`, `∀`, 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` (as well as `t_1`, `t_2` and so on) is also used to denote individual terms.

The meta-syntax `x` denotes a sequence of the form `x_1 x_2 · · · x_n` for some `x_1, x_2, · · · , x_n` and `n ≥ 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 `T`
of interest [BSST09]. What distinguishes SMT from general automated deduction is that the background theory \( T \) need not be finitely or even first-order axiomatizable, and that specialized inference methods are used for each theory. By being theory-specific and restricting their language to certain classes of formulas (such as, typically but not exclusively, quantifier-free formulas), these specialized methods can be implemented in solvers that are more efficient in practice than general-purpose theorem provers.

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

1.2.1 What is SMT-LIB?

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

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

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

1.2.2 Main features of the SMT-LIB standard

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

Version 2.0 sought to improve the usefulness of the SMT-LIB standard by simplifying its logical language while increasing its expressiveness and flexibility. In addition, it introduced a command language for SMT solvers that expanded their SMT-LIB interface considerably, allowing users to tap the numerous functionalities that most modern SMT solvers provide.
Like Version 2.0 and later versions, Version 2.6 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.
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 $\varphi$ is satisfiable, not in general but in the context of some background theory $T$ which constrains the interpretation of the symbols used in $\varphi$. Technically, the SMT problem for $\varphi$ and $T$ is the question of whether there is a model of $T$ that makes $\varphi$ true.

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

Informally speaking, SMT-LIB calls an \textit{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. \textit{underlying logic}, e.g., first-order, modal, temporal, second-order, and so on,

2. \textit{background theory}, the theory against which satisfiability is checked,

3. \textit{input formulas}, the class of formulas the solver accepts as input, and

4. \textit{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 may be 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 commitment to each of them is described in the following.

2.2 Underlying Logic

Version 2.6 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 sort constants, to denote sorts: sorts can be denoted by sort constants like \texttt{Int} as well as sort terms like \texttt{(List (Array Int Real))}. Finally, in addition to the usual existential and universal quantifiers, the logic includes a \textit{let} binder and a \textit{match} binder analogous to constructs with the same name found in functional programming languages.

SMT-LIB’s underlying logic, henceforth \textit{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. 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 \textit{basic} theories and \textit{combined} theories. Basic theories, such as the theory of real numbers, the theory of arrays, the theory of fixed-size bit vectors and so on, are those explicitly defined in the SMT-LIB catalog. Combined theories are defined implicitly in terms of basic theories by means of a general modular combination operator. The difference between a basic theory and a combined one in SMT-LIB is essentially operational. Some SMT-LIB theories, such as the theory of finite sets with a

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

\footnote{This catalog is available, separately from this document, from the SMT-LIB website (www.smt-lib.org).}
cardinality operator, are defined as basic theories, even if they are in fact a combination of smaller theories, because they cannot be obtained by modular combination.

Theory specifications have mostly documentation purposes. They are meant to be standard references for human readers. For practicality then, the format insists that only the signature of a theory (essentially, its set of sort symbols and sorted function symbols) be specified formally—provided it is finite.\footnote{The finiteness condition can be relaxed a bit for signatures that include certain commonly used sets of constants such as the set of all numerals.} By “formally” here we mean written in a machine-readable and processable format, as opposed to written in free text, no matter how rigorously. By this definition, theories themselves are defined informally, in natural language. Some theories, such as the theory of bit vectors, have an infinite signature. For them, the signature too is specified informally in English.\footnote{By efficiency here we do not necessarily refer to worst-case time complexity, but efficiency in practice.}

2.4 Input Formulas

SMT-LIB adopts a single and general first-order (sorted) language in which to write logical formulas. It is often the case, however, that SMT applications work with formulas expressed in some particular fragment of the language. The fragment in question matters because one can often write a solver specialized on that sublanguage that is much more efficient than a solver meant for a larger sublanguage.\footnote{An extreme case of this situation occurs when satisfiability modulo a given theory $T$ is decidable for a certain fragment (quantifier-free, say) but undecidable for a larger one (full first-order, say), as for instance happens with the theory of arrays \cite{BMS06}. But a similar situation occurs even when the decidability of the satisfiability problem is preserved across various fragments. For instance, if $T$ is the theory of real numbers, the satisfiability in $T$ of full-first order formulas 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 \cite{BBC05}.}

An extreme case of this situation occurs when satisfiability modulo a given theory $T$ is decidable for a certain fragment (quantifier-free, say) but undecidable for a larger one (full first-order, say), as for instance happens with the theory of arrays \cite{BMS06}. But a similar situation occurs even when the decidability of the satisfiability problem is preserved across various fragments. For instance, if $T$ is the theory of real numbers, the satisfiability in $T$ of full-first order formulas 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 \cite{BBC05}.

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

Starting with Version 2.0, the SMT-LIB standard includes a scripting language that defines a textual interface for SMT solvers. SMT solvers implementing this interface act as interpreters of the scripting language. The language is command-based, and defines a number of input/output functionalities that go well beyond simply checking the satisfiability of an input
formula. It includes commands for setting various solver parameters, declaring new symbols, asserting and retracting formulas, checking the satisfiability of the current set of asserted formulas, inquiring about models of satisfiable sets, printing various diagnostics, and so on.
Part II
Syntax
This chapter defines and explains the concrete syntax of the SMT-LIB standard, what we comprehensively refer to as the SMT-LIB language. The SMT-LIB language has three main components: theory declarations, logic declarations, and scripts. Its syntax is similar to that of the LISP programming language. In fact, every expression in this version is a legal S-expression of Common Lisp [Ste90]. The choice of the S-expression syntax and the design of the concrete syntax was mostly driven by the goal of simplifying parsing, as opposed to facilitating human readability.(2)

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

### 3.1 Lexicon

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

SMT-LIB source files consist of Unicode characters in any 8-bit encoding, such as UTF-8, that extends the original 7-bit US-ASCII set. While not technically Unicode, the ISO 8859-1 character set is also allowed since it coincides with the first 256 characters of UTF-8.(3)

Most lexical tokens defined below are limited to US-ASCII printable characters, namely, characters 32\textsubscript{dec} to 126\textsubscript{dec}. The remaining printable characters, mostly used for non-English alphabet letters (characters 128\textsubscript{dec} and beyond), are allowed in string literals, quoted symbols, and comments.
A comment is any character sequence not contained within a string literal or a quoted symbol (see later) that begins with the semi-colon character ; and ends with the first subsequent line-breaking character, i.e., 10_{dec} or 13_{dec}. Both comments and consecutive white space characters occurring outside a string literal or a symbol (see later) are considered whitespace. The only lexical function of whitespace is to break the source text into tokens.\footnote{Which implies that the language's semantics does not depend on indentation and spacing.}

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

**White Space Characters.** A \langle white\_space\_char \rangle is one of the following characters: 9_{dec} (tab), 10_{dec} (line feed), 13_{dec} (carriage return), and 32_{dec} (space).

**Printable Characters.** A \langle printable\_char \rangle is any character from 32_{dec} to 126_{dec} (US-ASCII) and from 128_{dec} on.\footnote{Note that the space character is both a printable and a whitespace character.}

**Digits.** A \langle digit \rangle is any character from 48_{dec} to 57_{dec} (0 through 9)

**Letters.** A \langle letter \rangle is any character from 65_{dec} to 90_{dec} (English alphabet letters A through Z) and from 97_{dec} to 122_{dec} (English alphabet letters a through z).\footnote{Which implies that the language's semantics does not depend on indentation and spacing.}

**Numerals.** A \langle numeral \rangle is the digit 0 or a non-empty sequence of digits not starting with 0.

**Decimals.** A \langle decimal \rangle is a 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 .

\begin{verbatim}
#x0      #xA04
#x01Ab   #x61ff
\end{verbatim}

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

\begin{verbatim}
#b0      #b1
#b001    #b101011
\end{verbatim}

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

"this is a string literal"

""

"She said: ""Bye bye"" and left."

"this is a string literal

with a line break in it"

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

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

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

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

The syntactic category ⟨reserved⟩ denotes any reserved word.

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

\[\sim ! \odot \% \& \_ - + = < > . ? / \]

that does not start with a digit and is not a reserved word.\(^5\)

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

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

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

---

\(^5\)Note that simple symbols cannot contain non-English letters.
Symbols are case sensitive. They are used mainly as operators or identifiers. Conventionally, arithmetic characters and the like are used, individually or in combination, as operator names; in contrast, alpha-numeric symbols, possibly with punctuation characters and underscores, are used as identifiers. But, as in LISP, this usage is only recommended (for human readability), not prescribed. For additional flexibility, arbitrary sequences of whitespace and printable characters (except for \ and \) enclosed in vertical bars are also allowed as symbols. Following Common Lisp’s conventions, enclosing a simple symbol in vertical bars does not produce a new symbol. This means for instance that \texttt{abc} and \texttt{|abc|} are the same symbol.

Simple symbols starting with the character \texttt{@} or \texttt{.} are reserved for solver use.\footnote{This includes symbols such as \texttt{|@abc|} and \texttt{|.abc|} which are considered the same as \texttt{@abc} and \texttt{.abc}, respectively.} Solvers can use them respectively as identifiers for abstract values and solver-generated function symbols other than abstract values.

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

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 ⟨\textit{s_expr}⟩ defined by the production rules below.

\[
\begin{align*}
\langle\text{spec_constant}\rangle & ::= \langle\text{numeral}\rangle | \langle\text{decimal}\rangle | \langle\text{hexadecimal}\rangle | \langle\text{binary}\rangle | \langle\text{string}\rangle \\
\langle\text{s_expr}\rangle & ::= \langle\text{spec_constant}\rangle | \langle\text{symbol}\rangle | \langle\text{reserved}\rangle | \langle\text{keyword}\rangle \\
& \quad | (\langle\text{s_expr}\rangle^*)
\end{align*}
\]
Remark 1 (Meaning of special constants). Elements of the \textit{\langle spec\_constant \rangle} category do not always have the expected associated semantics in the SMT-LIB language (i.e., elements of \textit{\langle numeral \rangle} denoting integers, elements of \textit{\langle string \rangle} denoting character strings, and so on). In particular, in the \textit{\langle 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 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 \textit{\langle binary \rangle} may denote integers modulo \(n\) in one theory and binary strings in another; the elements of \textit{\langle decimal \rangle} may denote rational numbers in one theory and floating point values in another.

3.3 Identifiers

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

\[
\text{\langle index \rangle ::= \langle numeral \rangle | \langle symbol \rangle}
\]
\[
\text{\langle identifier \rangle ::= \langle symbol \rangle | (\textunderscore\langle symbol \rangle\langle index \rangle)^+}
\]

\begin{verbatim}
  plus + <= Real | John Brown |
  (_ vector-add 4 5) (_ BitVec 32)
  (_ move up) (_ move down) (_ move left) (_ move right)
\end{verbatim}

We refer to identifiers from \textit{\langle symbol \rangle} as \textit{simple} identifiers and to the others as \textit{indexed} identifiers. Since identifiers are used as the names of function symbols, sort symbols, sort parameters, variables and commands, we often refer to them informally as \textit{names} in this document.

Remark 2 (Namespaces and shadowing of identifiers). There are several namespaces for identifiers: sorts, terms, command names, and attributes. The same identifier can occur in different namespaces with no risk of conflicts because each namespace can always be identified syntactically. Within the term namespace, bound variables can shadow one another as well as function symbols in accordance with a lexical scoping discipline described in Section 3.6. Similarly, sort parameters can shadow sort symbols, as described in Section 3.5.

3.4 Attributes

Several syntactic categories in the language contain \textit{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 \textit{\langle keyword \rangle} category. Attribute values are in general S-expressions other than keywords, although most predefined attributes use a more restricted category for their values.
3.5 Sorts

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

Syntactically, a sort symbol can be either the distinguished symbol $\text{Bool}$ or any $(\text{identifier})$. A sort parameter can be any $(\text{symbol})$ (which in turn, is an $(\text{identifier})$).

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

$$\text{Int} \text{ Bool}$$

$$(\_ \text{ BitVec} 3) \text{ (List (Array Int Real))}$$

$$(\langle \_ \text{ FixedSizeList} 4 \rangle \text{ Real}) \text{ (Set (\_ \text{ Bitvec} 3))}$$

A (non-)parametric sort is a sort containing (no) parameters. We will use $\sigma$ to denote non-parametric sorts and $\tau$ to denote possibly parametric sorts.

3.5.1 Ranks

Each function symbol in an SMT-LIB script is associated with one or more ranks, non-empty sequences of sorts. Intuitively, a function symbol $f$ with rank $\sigma_1 \cdots \sigma_n \sigma$, with none of the sorts parametric, denotes a function that takes as input $n$ values of respective sorts $\sigma_1, \ldots, \sigma_n$, and returns a value of sort $\sigma$.

In contrast, a function symbol $f$ with rank $\tau_1 \cdots \tau_n \tau$, where each sort may contain sort parameters, actually stands for a whole class of function symbols, all named $f$ and each with a rank obtained from $\tau_1 \cdots \tau_n \tau$ by instantiating in all possible ways every occurrence in $\tau_1 \cdots \tau_n \tau$ of a sort parameter with a non-parametric sort.

3.6 Terms and Formulas

Abstractly, term are constructed out of constant symbols in the $(\text{spec\_constant})$ category (numerals, decimals, strings, etc.), variables, function symbols, four kinds of binders (introduced by
the reserved words \texttt{let,forall,exists, and match}, and an annotation operator (the reserved word \texttt{!}). In its simplest form, a term is a special constant symbol, a variable, a function symbol, or the application of a function symbol to one or more terms. More complex terms include one or more binders.

Concretely, a variable can be any \texttt{⟨symbol⟩}, while a function symbol can be any \texttt{⟨identifier⟩} (i.e., a symbol or an indexed symbol). As a consequence, contextual information is needed during parsing to know whether an identifier is to be treated as a variable or a function symbol. For variables, this information is provided by the three binders which are the only mechanism to introduce variables. Function symbols, in contrast, are predefined, as explained later. Recall that every function symbol \( f \) is separately associated with one or more ranks, each specifying the sorts 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 \textit{qualified identifier} of the form \texttt{(as \( f \) \( σ \))}.

\[
\text{⟨qual_identifier⟩ ::= ⟨identifier⟩ | ( as ⟨identifier⟩ ⟨sort⟩ )}
\]
\[
\text{⟨var_binding⟩ ::= ( ⟨symbol⟩ ⟨term⟩ )}
\]
\[
\text{⟨sorted_var⟩ ::= ( ⟨symbol⟩ ⟨sort⟩ )}
\]
\[
\text{⟨pattern⟩ ::= ⟨symbol⟩ | ( ⟨symbol⟩ ⟨symbol⟩ + )}
\]
\[
\text{⟨match_case⟩ ::= ( ⟨pattern⟩ ⟨term⟩ )}
\]
\[
\text{⟨term⟩ ::= ⟨spec_constant⟩}
\]
\[
\text{⟨term⟩ ::= ⟨qual_identifier⟩}
\]
\[
\text{⟨term⟩ ::= ( ⟨qual_identifier⟩ ⟨term⟩ + )}
\]
\[
\text{⟨term⟩ ::= ( let ( ⟨var_binding⟩ + ) ⟨term⟩ )}
\]
\[
\text{⟨term⟩ ::= ( forall ( ⟨sorted_var⟩ + ) ⟨term⟩ )}
\]
\[
\text{⟨term⟩ ::= ( exists ( ⟨sorted_var⟩ + ) ⟨term⟩ )}
\]
\[
\text{⟨term⟩ ::= ( match ⟨term⟩ ( ⟨match_case⟩ + ) )}
\]
\[
\text{⟨term⟩ ::= ( ! ⟨term⟩ ⟨attribute⟩ + )}
\]

SMT-LIB scripts can contain only \textit{well-sorted} terms (see Section 3.6.4). Formulas in SMT-LIB are just well-sorted terms of sort \texttt{Bool}. As a consequence, there is no syntactic distinction between function and predicate symbols; the latter are simply function symbols whose result sort is \texttt{Bool}. Another consequence is that function symbols can take formulas (even quantified ones) as arguments.

\begin{verbatim}
(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))))))
\end{verbatim}
3.6. TERMS AND FORMULAS

3.6.1 Variable Binders

Variables are introduced by means of one of the four binders. Each binder allows the introduction of one or more variables or parameters with local scope.

**Exists and forall quantifiers.** These binders correspond to the usual universal and existential quantifiers of first-order logic, except that each variable they quantify is also associated with a sort. Both binders have a non-empty list of variables, which abbreviates a sequential nesting of quantifiers. Specifically, a formula of the form

\[ (\forall ((x_1 \sigma_1) (x_2 \sigma_2) \cdots (x_n \sigma_n)) \varphi) \]  

has the same semantics as the formula

\[ (\forall ((x_1 \sigma_1)) (\forall ((x_2 \sigma_2)) \cdots (\forall ((x_n \sigma_n)) \varphi) \cdots) \]  

Note that the variables in the list \(((x_1 \sigma_1) (x_2 \sigma_2) \cdots (x_n \sigma_n))\) of (3.1) are not required to be pairwise disjoint. However, because of the nested quantifier semantics, earlier occurrences of same variable in the list are shadowed by the last occurrence—making those earlier occurrences useless. The same argument applies to the \(\exists\) binder.

**Let.** The \texttt{let} binder introduces and defines one or more local variables \textit{in parallel}. Semantically, a term of the form

\[ (\texttt{let} ((x_1 t_1) \cdots (x_n t_n)) t) \]  

is equivalent to the term \(t[t_1/x_1, \ldots, t_n/x_n]\) obtained from \(t\) by simultaneously replacing each free occurrence of \(x_i\) in \(t\) by \(t_i\), for each \(i = 1, \ldots, n\), possibly after a suitable renaming of \(t\)'s bound variables to avoid capturing any variables in \(t_1, \ldots, t_n\). Because of the parallel semantics, the variables \(x_1, \ldots, x_n\) in (3.3) must be pairwise distinct.

**Remark 3** (No sequential version of \texttt{let}). The language does not have a sequential version of \texttt{let}. Its effect is achieved by nesting lets, as in \((\texttt{let} ((x_1 t_1)) (\texttt{let} ((x_2 t_2)) t))\).

**Match.** Similarly to pattern matching statements in functional programming languages or in certain interactive theorem provers, the \texttt{match} binder is used to perform pattern matching on values of an algebraic data type (see Section 4.2.3). It has the form

\[ (\texttt{match} t ((p_1 t_1) \cdots (p_{m+1} t_{m+1}))) \]  

where \(t\) is a term of some datatype sort \(\delta\) and, for each \(i = 1, \ldots, m + 1\), \(p_i\) is a \texttt{pattern} for \(\delta\), and \(t_i\) a term of some sort \(\sigma\). A pattern \(p\) in turn is either a variable \(x\) of sort \(\delta\), a nullary constructor \(c\) of \(\delta\), or a term of the form \((c \; x_1 \cdots \; x_k)\) where \(c\) is a constructor of \(\delta\) of rank \(\sigma_1 \cdots \sigma_k \delta\) with \(k > 0\), and \(x_1, \ldots, x_k\) are \textit{distinct} variables of respective sort \(\sigma_1, \ldots, \sigma_k\). The list \(p_1, \ldots, p_{m+1}\) may contain more than one pattern with the same constructor or more than one pattern consisting of a variable. However, it must contain a pattern consisting of a variable, unless every constructor of \(\delta\) occurs in one of the patterns.

The term \(t_i\) in (3.4) can contain free occurrences of the variables occurring in pattern \(p_i\), if any. The scope of those variables is the term \(t_i\).
3.6. TERMS AND FORMULAS

; Axiom for list append: version 1
; List is a parametric datatype
; with constructors "nil" and "cons"
;
(forall ((l1 (List Int)) (l2 (List Int)))
  (= (append l1 l2)
      (match l1
         (nil l2)
         ((cons h t) (cons h (append t l2)))))
)

; Axiom for list append: version 2
(forall ((l1 (List Int)) (l2 (List Int)))
  (= (append l1 l2)
      (match l1
         ((cons h t) (cons h (append t l2)))
         (x l2)))))  ; x is a variable

**Remark 4** (No nested patterns). *Nested* patterns, where the arguments of a constructor can be themselves non-variable patterns, are not allowed. This implies in particular that in a pattern of the form \((c s_1 \cdots s_k)\), where \(s_1, \ldots, s_k\) are symbols, those symbols are always to be parsed as variables.

Strictly speaking, the *match* binder is not essential since it can be defined in terms of the other binders. Specifically, expression (3.4) can be written equivalently as follows.

1. Suppose that for all \(i = 1, \ldots, m+1\), the pattern \(p_i\) has the form \((c_i x_{i,1} \cdots x_{i,k_i})\) with \(k_i > 0\); \(q_i\) is the associate tester for that constructor; and \(s_{i,1}, \ldots, s_{i,k_i}\) are the selectors associated, in order, with the constructor’s arguments. Then, match expression (3.4) has the same meaning as

\[
\text{ite} (q_1 t) \left( \text{let} ((x_{1,1} t) \cdots (x_{1,k_1} t)) \right) t_1 \\
\text{ite} (q_2 t) \left( \text{let} ((x_{2,1} t) \cdots (x_{2,k_2} t)) \right) t_2 \\
\cdots \\
\text{ite} (q_m t) \left( \text{let} ((x_{m,1} t) \cdots (x_{m,k_m} t)) \right) t_m \\
\left( \text{let} ((x_{m+1,1} t) \cdots (x_{m+1,k_{m+1}} t)) \right) t_{m+1} \cdots 
\]

(3.5)

2. When for some \(i \in \{1, \ldots, m+1\}\), the pattern \(p_i\) is a nullary constructor, the corresponding *let* subexpression in (3.5) is replaced by \(t_i\).

3. If instead for some minimal \(i \in \{1, \ldots, m+1\}\), the pattern \(p_i\) is a variable \(x\), then the whole \(i\)th *ite* subexpression of (3.5), if \(i \leq m\), or the *let* subexpression, if \(i = m\), is replaced by \((\text{let} ((x t)) t_i)\).

### 3.6.2 Parameter Binders

SMT-LIB allows the declaration of parametric function symbols in theory declarations (see Section 3.7) and of parametric datatypes in user scripts (see Section 4.2.3). Sort parameters are introduced with expressions of the form
where $u_1, \ldots, u_k$ are $k > 0$ pairwise distinct sort parameters and $e$ is an expression containing occurrences of $u_1, \ldots, u_k$.

### 3.6.3 Scoping of variables and parameters

The notions of free variable/parameter occurrence in an s-expression, bound variable/parameter, and (binder) scope are defined as follows.

A variable $x$:

- occurs free in the expression $x$;
- occurs free in an expression $(e_1 \cdots e_n)$ if $e_1$ is $\text{par}$ or not a binder, and $x$ occurs free in some $e_i$ ($1 \leq i \leq n$);
- occurs free in an expression $(\text{forall} (s_1 \cdots s_n) t)$ or $(\text{exists} (s_1 \cdots s_n) t)$ if it does not occur in $s_1, \ldots, s_n$ and occurs free in $t$;
- occurs free in an expression $(\text{let} ((x_1 t_1) \cdots (x_n t_n)) t)$ if (i) it occurs free in some $t_i$ ($1 \leq i \leq n$) and the corresponding $x_i$ occurs free in $t$, or (ii) it does not occur in $x_1, \ldots, x_n$ and occurs free in $t$;
- occurs free in an expression $(\text{match} t ((p_1 t_1) \cdots (p_n t_n)))$ if it occurs free in $t$ or it occurs free in some $t_i$ ($1 \leq i \leq n$) and does not occur in the corresponding $p_i$.

A parameter $u$:

- occurs free in the expression $u$;
- occurs free in an expression $(e_1 \cdots e_n)$ if $e_1$ is not $\text{par}$ and $u$ occurs free in some $e_i$ ($1 \leq i \leq n$);
- occurs free in an expression $(\text{par} (u_1 \cdots u_n) e)$ if it occurs free in $e$ and does not occur in $(u_1 \cdots u_n)$.

Each non-free, or bound, occurrence of a variable in an expression has a scope defined as follows.

- In an expression $(q ((x_1 \sigma_1) (x_2 \sigma_2) \cdots (x_n \sigma_n)) t)$ where $q$ is either $\text{forall}$ or $\text{exists}$, or in an expression $(\text{let} ((x_1 t_1) \cdots (x_n t_n)) t)$, the scope of each variable in $\{x_1, \ldots, x_n\}$ is the term $t$.
- In an expression $(\text{match} t ((p_1 t_1) \cdots (p_n t_n)))$, the scope of each variable occurring in pattern $p_i$ is the corresponding term $t_i$ ($1 \leq i \leq n$).
- In an expression $(\text{par} (u_1 \cdots u_n) e)$, the scope of each parameter in $\{u_1, \ldots, u_n\}$ is the expression $e$. 
3.6. TERMS AND FORMULAS

**Shadowing** All binders follow a lexical scoping discipline, consistently with the semantics of the SMT-LIB logic, as described in Section 5.3. In particular, a bound variable or parameter will shadow any variable or user-defined function symbol with the same name from an enclosing scope. For instance, in a function symbol declaration like `(par (Int) (f (Array Int Int) Int))`, all occurrences of `Int` are sort parameters regardless of the existence of a previously defined sort symbol `Int`.\(^{13}\) Similarly, in a `match` pattern like `(cons x nil)`, the symbol `nil` is a variable regardless of the existence of a previously defined constructor symbol `nil`.\(^5\)

**Remark 5** (No shadowing of theory symbols). One exception is that binders cannot shadow *theory function symbols*, that is, function symbols from the declaration (see Section 3.7) of a theory included in the current logic (see Section 3.8 and Subsection 4.2.1). In other words, variables and parameters cannot have the same name as a theory function symbol in the same scope.\(^{14}\)

### 3.6.4 Well-sortedness requirements

All terms of the SMT-LIB language are additionally required to be well sorted. Well-sortedness rules are presented and discussed in Section 5.2, in terms of the logic’s abstract syntax.

Except for patterns in `match` expressions, every occurrence of an *ambiguous* function symbol\(^6\) `f` in a term *must* occur as a qualified identifier of the form `(as f σ)` where `σ` is the intended output sort of that occurrence.\(^{15}\) The same requirement applies to occurrences of solver-generated constants within terms output by the solver—as the type of these constants is unknown to the user.

```
(cons "abc" (as nil (List String)))

; const-array takes a value v and
; returns an array storing v at all positions
(= a ((as const-array (Array Int Real)) 0.0))

(select (as @a1 (Array Int Int)) 3)
```

### 3.6.5 Annotations

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

### 3.6.6 Term attributes

Currently there are only two predefined term attributes: `:named` and `:pattern`. The values of the `:named` attribute range over the `(symbol)` category. The attribute can be used in scripts

\(^5\)Note that a pattern `(cons x nil)` where `nil` was a constructor would be nested, which is not allowed.

\(^6\)That is, function symbols with more than one possible output sort for the same sequence of input sorts (see Subsection 5.2.1).
to give a closed term a symbolic name, which can then be used as a proxy for the term (see Section 4.2).

\[
(=> (! (> x y) : named p1)
  (! (= x z) : named p2))
\]

The values of the :pattern attribute range over sequences of \(\langle\text{term}\rangle\) elements. The attribute is used to define instantiation patterns for quantifiers, which provide heuristic information to SMT solvers that reason about quantified formulas by quantifier instantiation. Instantiation patterns can only be used to annotate the body \(\varphi\) of a quantified formula of the form

\[
(\forall (x_1 \sigma_1) \cdots (x_k \sigma_k) \varphi)
\]

where \(\varphi\) is \text{forall} or \text{exists}, so that the resulting annotated formula has the form

\[
(\forall (x_1 \sigma_1) \cdots (x_k \sigma_k)) (\text{ ! :pattern } (p_{1,1} \cdots p_{1,n_1}) \\
\text{ :pattern } (p_{m,1} \cdots p_{m,n_m}))
\]

where each \(p_{ij}\) is a binder-free term with no annotations and the same well-sortness requirements as the formula’s body.\(^{16}\)

\[
(\forall (x_0 A) (x_1 A) (x_2 A))
  \text{ ! (=> (and (r x0 x1) (r x1 x2)) (r x0 x2))}
  \text{ :pattern ((r x0 x1) (r x1 x2))}
  \text{ :pattern ((p x0 a))}
\]

The intended use of these patterns is to suggest to the solver that it should try to find independently for each \(i = 1, \ldots, m\), a sequence \(t_{i,1} \cdots t_{i,n_i}\) of \(n_i\) ground terms that simultaneously match the pattern terms \(p_{i,1} \cdots p_{i,n_i}\).\(^7\)

### 3.7 Theory Declarations

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

From Version 2.0 on, theory declarations may also declare entire families of overloaded function symbols by using ranks that contain \textit{sort parameters}, locally scoped sort symbols of

\(^7\)The terms assigned to the variables \(x_1, \ldots, x_k\) by the simultaneous matching substitution are typically used to instantiate the body of a universally quantified formula in order to generate ground consequences of that formula.
arity 0. This kind of parametricity entails that a theory declaration generally defines a whole class of similar theories.

The syntax of theory declarations, specified in Figure 3.1, follows an attribute-value-based format. A theory declaration consists of a theory name and a list of elements.

Theory attributes with the following keywords are predefined attributes, with prescribed usage and semantics:

- `:definition`
- `:funs`
- `:funs-description`
- `:notes`
- `:sorts`
- `:sorts-description`
- `:values`

Additionally, a theory declaration can contain any number of user-defined attributes.

A theory declaration $(\text{theory } T \ a_1 \ \cdots \ a_n)$ defines a theory schema with name $T$ and attributes $a_1, \ldots, a_n$. Each instance of the schema is a theory $T_{\Sigma}$ with an expanded signature $\Sigma$, containing (zero or more) additional sort and function symbols with respect to those declared in $T$. Theories are defined as classes of first-order structures (or models) of signature $\Sigma$. See Section 5.4 for a formal definition of theories and a more detailed explanation of how a theory declaration can be instantiated to a theory. Concrete examples of instances of theory declarations are discussed later.

The value of a :sorts attribute is a non-empty sequence of sort symbol declarations $(\text{sort_symbol_decl})$. A sort symbol declaration $(s \ n \ a_1 \ \cdots \ a_m)$ declares a sort symbol $s$
of arity \( n \), and may additionally contain zero or more annotations, each in the form of an \langle attribute \rangle. In this version, there are no predefined annotations for sort declarations.

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

A (non-parametric) function symbol declaration of the form

\[
\langle f \sigma_1 \cdots \sigma_n \sigma \rangle
\]

with \( n \geq 0 \) declares a function symbol \( f \) with rank \( \sigma_1 \cdots \sigma_n \sigma \).

A parametric function symbol declaration, which has the form,

\[
\langle \text{par} (u_1 \cdots u_k) (f \tau_1 \cdots \tau_n \tau) \rangle
\]

with \( k > 0, n \geq 0 \) and \( u_1, \ldots, u_k \) all distinct, declares a function symbol \( f \) with parametric rank \( \tau_1 \cdots \tau_n \tau \). This effectively 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 in all possible ways each occurrence in \( \tau_1 \cdots \tau_n \tau \) of the sort parameters \( u_1, \ldots, u_k \) with non-parametric sorts.

As with sorts, each (parametric) function symbol declaration may additionally contain zero or more annotations \( \alpha_1, \ldots, \alpha_n \), each in the form of an \langle attribute \rangle. In this version, there are only 4 predefined function symbol annotations, all attributes with no value: \langle chainable \rangle, \langle left-assoc \rangle, \langle right-assoc \rangle, and \langle pairwise \rangle. The \langle left-assoc \rangle annotation can be added only to function symbol declarations of the form

\[
\langle f \sigma_1 \sigma_2 \sigma_1 \rangle \text{ or } \langle \text{par} (u_1 \cdots u_k) (f \tau_1 \tau_2 \tau_1) \rangle.
\]

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

\[
\langle f \sigma_1 \sigma_2 \sigma_2 \rangle \text{ or } \langle \text{par} (u_1 \cdots u_k) (f \tau_1 \tau_2 \tau_2) \rangle.
\]

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

The \langle chainable \rangle and \langle pairwise \rangle annotations can be added only to function symbol declarations of the form

\[
\langle f \sigma \sigma \text{ Bool} \rangle \text{ or } \langle \text{par} (u_1 \cdots u_k) (f \tau \tau \text{ Bool}) \rangle
\]

and are mutually exclusive. With the first annotation, \( \langle f t_1 \cdots t_n \rangle \) with \( n > 2 \) is syntactic sugar for \( \langle \text{and} (f t_1 t_2) (f t_2 t_3) \cdots (f t_{n-1} t_n) \rangle \) where \text{and} is itself a symbol declared as \langle left-assoc \rangle in every theory (see Subsection 3.7.1); with the second, \( \langle f t_1 \cdots t_n \rangle \) is syntactic sugar (recursively) for \( \langle \text{and} (f t_1 t_2) \cdots (f t_1 t_n) (f t_2 \cdots t_n) \rangle \).
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.\(^{(18)}\)

```
(\(+\) \(\text{Real} \ \text{Real} \ \text{Real} : \text{left-assoc}\))

(\(\text{and}\) \(\text{Bool} \ \text{Bool} \ \text{Bool} : \text{left-assoc}\))

(\(\text{par} (X) \ (\text{insert} \ X \ (\text{List} \ X) \ (\text{List} \ X) : \text{right-assoc})\))

(\(<\) \(\text{Real} \ \text{Real} \ \text{Bool} : \text{chainable}\))

(\(\text{equiv}\) \(\text{Elem} \ \text{Elem} \ \text{Bool} : \text{chainable}\))

(\(\text{par} (X) \ (\text{Disjoint} \ (\text{Set} \ X) \ (\text{Set} \ X) \ \text{Bool} : \text{pairwise})\))

(\(\text{par} (X) \ (\text{distinct} \ X \ X \ \text{Bool} : \text{pairwise})\))
```

The :sorts-description 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.\(^{(19)}\) For some theories, a mix of formal notation and natural language might be more appropriate. In the presence of parametric function symbol declarations, the definition must also specify the meaning of each instance of the declared symbol.\(^{(20)}\)

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

:funs_description
"All function symbols with rank of the form

\(\text{concat} \ (\_ \text{BitVec} \ i) \ (\_ \text{BitVec} \ j) \ (\_ \text{BitVec} \ m)\)

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.\(^{(19)}\) For some theories, a mix of formal notation and natural language might be more appropriate. In the presence of parametric function symbol declarations, the definition must also specify the meaning of each instance of the declared symbol.\(^{(20)}\)

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

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

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

1. They contain exactly one occurrence of the :definition and the :values attribute and any number of occurrences of other attributes.

2. Each sort symbol used in a :funs attribute is previously declared in some :sorts attribute. In each parametric function symbol declaration (par (u₁,...,uₖ) (f τ₁,...,τₙ τ)), any symbol other than f that is not a previously declared sort symbol must be one of the sort parameters u₁,...,uₖ.

3. The definition of the theory, provided in the :definition attribute, refers only to sort and function symbols previously declared formally in :sorts and :funs attributes or informally in :sorts-description and :funs-description attributes.

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

### 3.7.1 Examples

**Core theory**

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

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

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

respectively as abbreviations for the terms

\[8\] Which makes those attributes non-optional.
3.7. THEORY DECLARATIONS

(Theory Core

:sorts ( (Bool 0) )

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

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

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

Figure 3.2: The Core theory declaration.

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

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

Another instance has a single additional sort symbol U, say, of arity 0, and a (possibly
infinite) set number of function symbols with rank in U⁺. This theory corresponds to EUF,
the (one-sorted) theory of equality and uninterpreted functions (over those function symbols).
In this theory, ite has two ranks: (ite Bool Bool Bool Bool) and (ite Bool U U U). A
many-sorted version of EUF is obtained by instantiating Core with more than one nullary sort
symbol—and possibly additional function symbols over the resulting sort set.
(theory Ints
 :sorts ( (Int 0) )
 :funs ( (NUMERAL Int)
   (- Int Int) ; negation
   (- Int Int Int :left-assoc) ; subtraction
   (+ Int Int Int :left-assoc)
   (* Int Int Int :left-assoc)
   (<= Int Int Bool :chainable)
   (< Int Int Bool :chainable)
   (>= Int Int Bool :chainable)
   (> Int Int Bool :chainable) )
 :definition
 "For every expanded signature Sigma, the instance of Ints with that
 signature is the theory consisting of all Sigma-models that interpret
 - the sort Int as the set of all integers,
 - the function symbols of Ints as expected."
 :values
 "The Int values are all the numerals and all the terms of the form (- n)
 where n is a non-zero numeral."
)

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

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 $\sigma$, interprets (List $\sigma$) as the set of all lists over $\sigma$. The combined theory in that case is a theory of lists with uninterpreted functions.

**Integers**

The theory declaration of Figure 3.3 defines all theories that extend the standard theory of the (mathematical) integers with 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. The set of values for the Int sort consists of all numerals and all terms of the form (- n) where n is a numeral other than 0.
3.7. THEORY DECLARATIONS

A schematic version of the theory of functional arrays with extensionality is defined in the theory declaration \texttt{ArraysEx} in Figure 3.4. Each instance gives a theory of (arbitrarily nested) arrays. For instance, with the addition of the nullary sort symbols \texttt{Int} and \texttt{Real}, we get an instance theory whose sort set \( S \) contains, inductively, \texttt{Bool}, \texttt{Int}, \texttt{Real} and all sorts of the form \( \texttt{Array} \sigma_1 \sigma_2 \) with \( \sigma_1, \sigma_2 \in S \). This includes flat array sorts such as
\[ \texttt{(Array Int Int)}, \texttt{(Array Int Real)}, \texttt{(Array Real Int)}, \texttt{(Array Bool Int)} \]
conventional nested array sorts such as
\[ \texttt{(Array Int (Array Int Real))}, \]
as well as nested sorts such as
\[ \texttt{(Array (Array Int Real) Int)}, \texttt{(Array (Array Int Real) (Array Real Int))} \]
with an array sort in the \textit{index position} of the outer array sort.\(^{21}\)

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

\(^9\)For simplicity, the theory declaration in the figure is an abridged version of the declaration actually used in the SMT-LIB catalog.
3.8. LOGIC DECLARATIONS

Sets and Relations

A schematic many-sorted version of the theory of hereditary well-founded sets with urelements is defined in the theory declaration `SetsRelations` in Figure 3.5. Each instance gives a theory of sets of elements of the same sort. These elements can be either atomic (i.e., of a primitive sort like `Bool`), or tuples of elements, or sets themselves. For instance, with the addition of the nullary sort symbol `Int` we get an instance theory whose sort set \( S \) contains, inductively, `Bool`, `Int` and all sorts of the form \((\text{Set } \sigma)\) or \((\text{Prod } \sigma_1 \cdots \sigma_n)\) with \(\sigma, \sigma_1, \ldots, \sigma_n \in S\). In each model of the theory that interprets `Int` as the integers, we get Booleans, integers, sets of Booleans, sets of integers, sets of tuples over these sets, and so on.

Note that every sort \( \sigma \) in the signature is internalized in the theory, since the set denoted by \( \sigma \) is also denoted by the constant `univSet` of (the powerset) sort \((\text{Set } \sigma)\).

Remark 6 (Instances of parametric sorts). 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 \((\text{Array } \text{Int } \text{Real})\), 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 sorts, any sorts that are not used in a script are, for practical purposes, irrelevant. It is not a problem in theory either because scripts refer to logics, not directly to theories. And the language of a logic can 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; see www.smt-lib.org for the current catalog. Logic declarations have a similar format to theory declarations, although most of their attributes are informal.(22)

Attributes with the following predefined keywords are predefined attributes, with prescribed usage and semantics in logic declarations:

\[
\text{:theories :language :extensions :notes :values.}
\]

Additionally, as with theories, a logic declaration can contain any number of user-defined attributes.

\[
\langle\text{logic_attribute}\rangle \quad ::= \quad \text{:theories (}\langle\text{symbol}\rangle^+\rangle

\quad \mid \quad \text{:language}\langle\text{string}\rangle

\quad \mid \quad \text{:extensions}\langle\text{string}\rangle

\quad \mid \quad \text{:values}\langle\text{string}\rangle

\quad \mid \quad \text{:notes}\langle\text{string}\rangle

\quad \mid \quad \langle\text{attribute}\rangle

\langle\text{logic}\rangle \quad ::= \quad \langle\text{logic}\langle\text{symbol}\rangle\langle\text{logic_attribute}\rangle^+\rangle
\]

Note that sort parameters cannot occur in a formula.
(theory SetsRelations
  :sorts ( (Set 1) )
  :funs ( (par (X) (emptySet (Set X)))
           (par (X) (univSet (Set X)))
           (par (X) (singleton X (Set X)))
           (par (X) (union (Set X) (Set X) (Set X) :left-assoc))
           (par (X) (inters (Set X) (Set X) (Set X) :left-assoc))
           (par (X) (in X (Set X) Bool))
           (par (X) (subset (Set X) (Set X) Bool :chainable)) )
  :sorts_description
"All sort symbol declarations of the form (Prod n) with n > 1"
  :funs_description
"All function symbols with declarations of the form
  (par (X1 ... Xn) (tuple X1 ... Xn (Prod X1 ... Xn)))
  (par (X1 ... Xn) ((_ project i) (Prod X1 ... Xn) Xi))
  (par (X1 ... Xn) (prod (Set X1) ... (Set Xn) (Set (Prod X1 ... Xn))))
with n > 1 and i = 1,...,n"
  :notes
"A schematic theory of sets and relations."
  :definition
"For every expanded signature Sigma, the instance of SetsRelations
with that signature is the theory consisting of all Sigma-models that
for all sorts s, s1,..., sn, with n > 1, interpret
- (Set s) as the powerset of the set denoted by s
- (as emptySet (Set s)) as the empty set of sort (Set s)
- (as univSet (Set s)) as the set denoted by s
- (Prod s1 ... sn) as the Cartesian product of the sets denoted by s1,...,sn
- (tuple s1 ... sn (Prod s1 ... sn)) as the function that maps
  its inputs x1, ..., xn to the tuple (x1, ..., xn)
- ((_ project i) (Prod s1 ... sn) si) for i = 1, ..., n as the i-th
  projection function
- (prod (Set s1) ... (Set sn) (Set (Prod s1 ... sn))) as the function
  that maps its input sets to their Cartesian product
and interpret the other function symbols as the corresponding set operators
as expected."
)

Figure 3.5: A possible declaration for a theory of sets and relations.
A logic declaration \((\text{logic } L \ \alpha_1 \ \cdots \ \alpha_n)\) defines a logic with name \(L\) and attributes \(\alpha_1, \ldots, \alpha_n\).

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

1. They include exactly one occurrence of the :theories and the :language attribute (and any number of occurrences of other attributes).
2. The value \((T_1 \ \cdots \ T_n)\) of the :theories attribute lists names of theory schemas that have a declaration in SMT-LIB.
3. If two theory declarations among \(T_1, \ldots, T_n\) declare the same sort symbol, they give it the same arity.

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

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

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

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

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

### 3.8.1 Examples

Defining theories model-theoretically, as opposed to axiomatically as in more traditional approaches, confers great expressive power to the SMT-LIB underlying logic in spite of its restriction to a first-order syntax. Several established logics, from propositional all the way to higher-order logic, can be defined as SMT-LIB sublogics given a suitable theory. We provide a small sample below, for illustrative purposes. Again, see www.smt-lib.org for the list of the actual logics defined in the SMT-LIB standard.
3.8. LOGIC DECLARATIONS

Propositional logic

Propositional logic can be readily defined by an SMT-LIB logic declaration. The logic’s theory is the instance of the 
\texttt{Core} theory declaration whose signature adds infinitely-many function symbols of rank \texttt{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 binary decision diagrams (BDDs) as formulas, thanks to the \texttt{ite} operator of \texttt{Core}.

Quantified Boolean logic

The logic of quantified Boolean formulas (QBFs) can be defined as well. The theory is again an instance of \texttt{Core} but this time with no additional symbols at all. The language consists of (closed) quantified formulas all of whose variables are of sort \texttt{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 \texttt{QF_LIA} there. Its theory is an extension of the theory of integers and the Booleans with uninterpreted constant symbols. That is, it is the instance of the theory declaration \texttt{Ints} from Figure 3.3 whose signature adds to the symbols of \texttt{Ints} infinitely many free constants, new function symbols of rank \texttt{Int} or of rank \texttt{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 \( \ast \). Extensions of the basic language include expressions of the form \((\ast \ n \ t)\) and \((\ast \ t \ n)\), for some numeral \(n\), both of which abbreviate the term \((+ \ t \ \cdots \ t)\) with \(n\) occurrences of \(t\) (or 0 if \(n\) is 0). Also included are terms with negative integer coefficients, that is, expressions of the form \((\ast \ (- \ n) \ t)\) or \((\ast \ t \ (- \ n))\) for some numeral \(n\), both of which abbreviate the expression \((- \ (\ast \ n \ t))\).

Higher-order logic

The full language of the theory \texttt{SetsRelations} from Figure 3.5 has the same expressive power as higher-order logic or, more precisely, higher-order simple predicate logic with primitive types.\footnote{Higher-order quantification is realized by first-order quantification over sorts of the form \((\texttt{Set} \ \sigma)\). Similarly, higher-order predicate symbols can be encoded as unary first-order predicate symbols over sets. In fact, because of the presence of the predicate symbol \texttt{in}, no additional predicate symbols at all are needed: every atomic formula of the form \((p \ t)\) can be replaced by the formula \((\texttt{in} \ t \ p')\) where \(p'\) is a constant symbol of sort \((\texttt{Set} \ \texttt{(Prod} \ \sigma_1 \ \cdots \ \sigma_n \ \texttt{Bool}))\) if \(p\) has rank \((\texttt{Prod} \ \sigma_1 \ \cdots \ \sigma_n \ \texttt{Bool})\), and of sort \((\texttt{Set} \ \sigma)\) if \(p\) has rank \((\sigma \ \texttt{Bool})\).

This is a relational version of higher-order logic, as opposed to the functional versions that are more popular in computer science.}
Monadic second-order logic

This logic can be obtained by imposing restrictions on the language of the SetsRelations theory mentioned above, specifically, by disallowing the use of terms of Prod sorts (monadic restriction) or nested Set sorts (second-order restriction) while, however, allowing additional function and predicate symbols.

3.9 Scripts

Scripts are sequences of commands. In line with the LISP-like syntax, all commands look like LISP-function applications, with a command name applied to zero or more arguments. To facilitate processing, each command takes a constant number of arguments, although some of these arguments can be (parenthesis-delimited) lists of variable length. The full list of commands is provided in Figure 3.6.

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

The command set-option takes as an argument expressions of the syntactic category ⟨option⟩, which have the same form as attributes with values. Options with the predefined keywords listed in Figure 3.7 have a prescribed usage and semantics. Additional, solver-specific options are also allowed.

The command get-info takes as argument expressions of the syntactic category ⟨info_flag⟩ which are flags with the same form as keywords. The predefined flags listed in Figure 3.8 have a prescribed usage and semantics. Additional, solver-specific flags are also allowed. Examples of the latter might be, for instance, flags such as :time and :memory, referring to used resources, or :decisions, :conflicts, and :restarts, referring to typical statistics for current SMT solvers.

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

Assertion stack. Compliant solvers respond to various commands mostly by performing operations on a data structure we call the assertion stack. This is a single stack whose elements, called levels, are sets of assertions. Assertions include logical formulas (that is, terms of sort Bool), as well as declarations and definitions of sort and function symbols. Assertions are added by specific commands. By default, an assertion belongs to the most recent level at the time the corresponding command was executed. The stack starts with a first assertion level that cannot be removed. Further levels can be introduced by a push command and removed by a corresponding pop command. Popping a level from the assertion stack has the effect of undoing all assertions in it, including symbol declarations and definitions. An input option, :global-declarations, allows the user to make all symbol declarations and definitions global to the assertion stack. In other words, when that option is enabled, declarations and definitions become permanent, as opposed to being added to the assertion stack. Popping a
\( \langle \text{sort} \_\text{dec} \rangle \) ::= ( \langle \text{symbol} \rangle \langle \text{numeral} \rangle )

\( \langle \text{selector} \_\text{dec} \rangle \) ::= ( \langle \text{symbol} \rangle \langle \text{sort} \rangle )

\( \langle \text{constructor} \_\text{dec} \rangle \) ::= ( \langle \text{symbol} \rangle \langle \text{selector} \_\text{dec} \rangle^* )

\( \langle \text{datatype} \_\text{dec} \rangle \) ::= ( \langle \text{constructor} \_\text{dec} \rangle^+ ) | ( \text{par} ( \langle \text{symbol} \rangle^+ ) ( \langle \text{constructor} \_\text{dec} \rangle^+ ) )

\( \langle \text{function} \_\text{dec} \rangle \) ::= ( \langle \text{symbol} \rangle ( \langle \text{sorted} \_\text{var} \rangle^* ) \langle \text{sort} \rangle )

\( \langle \text{function} \_\text{def} \rangle \) ::= ( \langle \text{symbol} \rangle ( \langle \text{sorted} \_\text{var} \rangle^* ) \langle \text{sort} \rangle \langle \text{term} \rangle )

\( \langle \text{prop} \_\text{literal} \rangle \) ::= ( \langle \text{symbol} \rangle | ( \text{not} \langle \text{symbol} \rangle )

\( \langle \text{command} \rangle \) ::= ( \langle \text{assert} \langle \text{term} \rangle \rangle

\begin{itemize}
  \item ( \text{check-sat} )
  \item ( \text{check-sat-assuming} ( \langle \text{prop} \_\text{literal} \rangle^* ) )
  \item ( \text{declare-const} ( \langle \text{symbol} \rangle \langle \text{sort} \rangle )
  \item ( \text{declare-datatype} ( \langle \text{symbol} \rangle \langle \text{datatype} \_\text{dec} \rangle )
  \item ( \text{declare-datatypes} ( \langle \text{sort} \_\text{dec} \rangle^{n+1} ) ( \langle \text{datatype} \_\text{dec} \rangle^{n+1} ) )
  \item ( \text{declare-fun} ( \langle \text{symbol} \rangle ( \langle \text{sort} \rangle^* ) \langle \text{sort} \rangle )
  \item ( \text{declare-sort} ( \langle \text{symbol} \rangle ( \langle \text{numeral} \rangle )
  \item ( \text{define-fun} ( \langle \text{function} \_\text{def} \rangle )
  \item ( \text{define-fun-rec} ( \langle \text{function} \_\text{def} \rangle )
  \item ( \text{define-funs-rec} ( \langle \text{function} \_\text{dec} \rangle^{n+1} ) ( \langle \text{term} \rangle^{n+1} ) )
  \item ( \text{define-sort} ( \langle \text{symbol} \rangle ( \langle \text{symbol} \rangle^* ) \langle \text{sort} \rangle )
  \item ( \text{echo} ( \langle \text{string} \rangle )
  \item ( \text{exit} )
  \item ( \text{get-assertions} )
  \item ( \text{get-assignment} )
  \item ( \text{get-info} ( \langle \text{info} \_\text{flag} \rangle )
  \item ( \text{get-model} )
  \item ( \text{get-option} ( \langle \text{keyword} \rangle )
  \item ( \text{get-proof} )
  \item ( \text{get-unsat-assumptions} )
  \item ( \text{get-unsat-core} )
  \item ( \text{get-value} ( \langle \text{term} \rangle^+ ) )
  \item ( \text{pop} ( \langle \text{numeral} \rangle )
  \item ( \text{push} ( \langle \text{numeral} \rangle )
  \item ( \text{reset} )
  \item ( \text{reset-assertions} )
  \item ( \text{set-info} ( \langle \text{attribute} \rangle )
  \item ( \text{set-logic} ( \langle \text{symbol} \rangle )
  \item ( \text{set-option} ( \langle \text{option} \rangle )
\end{itemize}

\( \langle \text{script} \rangle \) ::= ( \langle \text{command} \rangle^* 

Figure 3.6: SMT-LIB Commands.
3.9. SCRIPTS

\[
\langle b\_value \rangle \ ::= \ true \mid false
\]

\[
\langle \text{option} \rangle \ ::= \ :\text{diagnostic\_output\_channel} \ \langle \text{string} \rangle \\
\quad \quad \quad \ ::= \ :\text{global\_declarations} \ (b\_value) \\
\quad \quad \quad \ ::= \ :\text{interactive\_mode} \ (b\_value) \\
\quad \quad \quad \ ::= \ :\text{print\_success} \ (b\_value) \\
\quad \quad \quad \ ::= \ :\text{produce\_assertions} \ (b\_value) \\
\quad \quad \quad \ ::= \ :\text{produce\_assignments} \ (b\_value) \\
\quad \quad \quad \ ::= \ :\text{produce\_models} \ (b\_value) \\
\quad \quad \quad \ ::= \ :\text{produce\_proofs} \ (b\_value) \\
\quad \quad \quad \ ::= \ :\text{produce\_unsat\_assumptions} \ (b\_value) \\
\quad \quad \quad \ ::= \ :\text{produce\_unsat\_cores} \ (b\_value) \\
\quad \quad \quad \ ::= \ :\text{random\_seed} \ (\text{numeral}) \\
\quad \quad \quad \ ::= \ :\text{regular\_output\_channel} \ \langle \text{string} \rangle \\
\quad \quad \quad \ ::= \ :\text{reproducible\_resource\_limit} \ (\text{numeral}) \\
\quad \quad \quad \ ::= \ :\text{verbosity} \ (\text{numeral}) \\
\quad \quad \quad \ ::= \ \langle \text{attribute} \rangle
\]

Figure 3.7: Command options.

\[
\langle \text{info\_flag} \rangle \ ::= \ :\text{all\_statistics} \mid :\text{assertion\_stack\_levels} \mid :\text{authors} \\
\quad \quad \quad \ ::= \ :\text{error\_behavior} \mid :\text{name} \mid :\text{reason\_unknown} \\
\quad \quad \quad \ ::= \ :\text{version} \mid \langle \text{keyword} \rangle
\]

Figure 3.8: Info flags.

stack level then has only the effect of removing asserted formulas (those in that level). Global declarations and definitions can be removed only by a reset command.

Declared/defined symbols. Sort and function symbols introduced with a declaration or a definition cannot have a name that begins with a dot (.), as such names are reserved for solvers’ use, or with @, as such symbols are reserved for solver-defined abstract values.

3.9.1 Command responses

The possible responses that a solver can produce in response to commands are shown in Figure 3.9. Every response must be an instance of \langle general\_response \rangle which specifies generic response possibilities as well as command-specific responses for certain commands (specified by \langle specific\_success\_response \rangle). In addition, with the :print-success option set to false, a solver produces no response in cases when it would have otherwise returned success.

Regular output, including error messages, is printed on the regular output channel. Diagnostic output, including warnings and progress information, is printed on the diagnostic output channel. These may be set using set-option and the corresponding attributes: respectively, :regular-output-channel and :diagnostic-output-channel. The values of these attributes should be (double-quote delimited) file names in the format specified by the POSIX standard.\footnote{This is the usual format adopted by all Unix-based operating systems, with / used as a separator for} The string literals "stdout" and "stderr" are reserved to refer specially to the
\[(\text{error-behavior}) \ ::= \text{immediate-exit} \mid \text{continued-execution}\]
\[(\text{reason-unknown}) \ ::= \text{memout} \mid \text{incomplete} \mid (s_{\text{expr}})\]
\[(\text{model_response}) \ ::= (\text{define-fun} (\text{function_def})) \mid (\text{define-fun-rec} (\text{function_def})) \mid (\text{define-funs-rec} (\text{function_dec})^{n+1} \mid (\text{term})^{n+1})\]
\[(\text{info_response}) \ ::= :\text{assertion-stack-levels} (\text{numeral}) \mid :\text{authors} (\text{string}) \mid :\text{error-behavior} (\text{error-behavior}) \mid :\text{name} (\text{string}) \mid :\text{reason-unknown} (\text{reason-unknown}) \mid :\text{version} (\text{string}) \mid \{\text{attribute}\}\]
\[(\text{valuation_pair}) \ ::= (\text{term}) (\text{term})\]
\[(\text{t_valuation_pair}) \ ::= (\text{symbol}) (\text{b_value})\]
\[(\text{check_sat_response}) \ ::= \text{sat} \mid \text{unsat} \mid \text{unknown}\]
\[(\text{echo_response}) \ ::= \{\text{string}\}\]
\[(\text{get_assertions_response}) \ ::= (\text{term}^*)\]
\[(\text{get_assignment_response}) \ ::= (\text{t_valuation_pair}^*)\]
\[(\text{get_info_response}) \ ::= (\text{info_response}^*)\]
\[(\text{get_model_response}) \ ::= (\text{model_response}^*)\]
\[(\text{get_option_response}) \ ::= \{\text{attribute_value}\}\]
\[(\text{get_proof_response}) \ ::= \{\text{s_expr}\}\]
\[(\text{get_unsat_assumptions_response}) \ ::= (\text{symbol}^*)\]
\[(\text{get_unsat_core_response}) \ ::= (\text{symbol}^*)\]
\[(\text{get_value_response}) \ ::= (\text{valuation_pair}^*)\]
\[(\text{specific_success_response}) \ ::= (\text{check_sat_response}) \mid (\text{echo_response}) \mid (\text{get_assertions_response}) \mid (\text{get_assignment_response}) \mid (\text{get_info_response}) \mid (\text{get_model_response}) \mid (\text{get_option_response}) \mid (\text{get_proof_response}) \mid (\text{get_unsat_assumptions_response}) \mid (\text{get_unsat_core_response}) \mid (\text{get_value_response})\]
\[(\text{general_response}) \ ::= \text{success} \mid (\text{specific_success_response}) \mid \text{unsupported} \mid (\text{error} \{\text{string}\})\]

Figure 3.9: Command responses.
(set-info :smt-lib-version 2.6) ; success
(set-option :print-success false)
(push 1)
(set-logic QF_LIA) ; success
(assert (> z x))
(declare-const w Int) ; success
(declare-const x Int)
(declare-const y Int)
(declare-const z Int)
(assert (> x y)) ; success
(assert (> y z)) ; success
(1)
(1)
(check-sat) ; success
(check-sat) ; success
(pop 1)
(push 1)
(get-info :all-statistics) ; (:time 0.01 :memory 0.2)
(get-assignment)
(get-assertions)
(get-model)
(get-option)
(get-proof)
(get-unsat-assumptions)
(get-unsat-core)
(exit)

Figure 3.10: Example script (over two columns), with expected solver responses in comments.

Specific Responses. Specific responses are defined, in Figure 3.9, for the following commands:

- \(\text{check-sat-response}\) for check-sat and check-sat-assuming
- \(\text{echo-response}\) for echo,
- \(\text{get-assertions-response}\) for get-assertions,
- \(\text{get-assignment-response}\) for get-assignment,
- \(\text{get-info-response}\) for get-info,
- \(\text{get-model-response}\) for get-model,
- \(\text{get-option-response}\) for get-option,
- \(\text{get-proof-response}\) for get-proof,
- \(\text{get-unsat-assump-response}\) for get-unsat-assumptions,
- \(\text{get-unsat-core-response}\) for get-unsat-core,
- \(\text{get-value-response}\) for get-value.

See Chapter 4 for more details.

3.9.2 Example scripts

We demonstrate some allowed behavior of a hypothetical solver in response to an example script. Each command is followed by example legal output from the solver in a comment, if
(set-info :smt-lib-version 2.6) ... (check-sat)
(set-option :print-success false) (get-value (a))
(set-option :produce-models true) ; (a (as @array1 (Array Int (List Int))))
; sat

(declare-const x Int) (assert ((select @array1 2)))
(declare-const y Int) ; (((select (as @array1 (Array Int (List Int))) 2)
(declare-fun f (Int) Int) ; (as @list0 (List Int))

(assert (= (f x) (f y))) ; ; get-value (a)
(assert (not (= x y))) (check-sat) ; ; sat

(check-sat) ; ; get-value ((first @list0) (rest @list0))
; ; (x 0) ; ; get-value (as @list0 (List Int)) (as nil (List Int))
; ; (y 1) ; ;
; )

(get-value (a (Array Int (List Int))))
(declare-const a (Array Int (List Int)))

Figure 3.11: Another example script (excerpt), with expected solver responses in comments.

there is any. The script in Figure 3.10 makes two background assertions and then conducts two independent queries. The get-info command requests information on the search using the :all-statistics flag. The script in Figure 3.11 uses the get-value command to get information about a particular model of the formula that the solver has reported satisfiable.

3.9.3 SMT-LIB Benchmarks

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

Benchmarks in the official SMT-LIB repository at www.smt-lib.org must satisfy additional requirements on the meta-level information they contain and the order in which it appears. Specifically, every benchmark must use set-info to set the attributes below as follows:

- :smt-lib-version, :source, :license, and :category must be set exactly once,
- :status must be set as many times as needed so that each occurrence of the command check-sat-assuming or check-sat in the benchmark is preceded (not necessarily immediately) by a corresponding :status info.

Moreover, the set set-info call for attribute :smt-lib-version must be the very first command in the benchmark.

13Since the output of (get-info :all-statistics) is solver-specific, the response reported in the script is for illustration purposes only.
14The same call to set-info can be used to provide the status for more than one call to check-sat-assuming or check-sat, if that status is the same.
Part III

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

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

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

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

4.1 General Requirements

The command language contains commands for managing a stack of assertion levels and for making queries about them. It includes commands to:

- declare and define new sort and function symbols (declare and define commands),
- add formulas to the current assertion level,
4.1. GENERAL REQUIREMENTS

Command name abbreviations:

- $ad = \text{assert, declare-}*,$ $\text{define-}*$$
- $c = \text{check-sat}^*$
- $e = \text{echo}$
- $g = \text{get-assertions}$
- $\text{gamv} = \text{get-assignment, get-model, get-value}$
- $\text{gsio} = \text{get-info, get-option, set-info, set-option}$
- $\text{gpu} = \text{get-proof, get-unsat-}^*$
- $p = \text{pop, push}$
- $r = \text{reset}$
- $ra = \text{reset-assertions}$
- $sl = \text{set-logic}$

Figure 4.1: Abstract view of transitions between solver execution modes. The symbol * here stands for the matching wildcard.

- reset the assertion stack or the whole solver,
- push and pop assertion levels,
- check the joint satisfiability of all formulas in the assertion stack, possibly under additional assumptions (check commands),
- obtain further information following a check command (e.g., model information),
- set values for standard and solver-specific options,
- get standard and solver-specific information from the solver.

This section provides some background and general requirements on how these functionalities are to be supported. The next section, Section 4.2 provides more details on how each compliant solver is to execute each command.

4.1.1 Execution Modes

At a high-level, a compliant solver can be understood as being at all times in one of four execution modes: a start mode, an assert mode and two query modes, sat and unsat. The solver
4.1. GENERAL REQUIREMENTS

starts in start mode, moves to assert mode once a logic is set, and then moves to one of the two query modes after executing a check command. Any command other than reset that modifies the assertion stack brings the solver back from a query mode to the assert mode. The reset command takes the solver back to start mode.

The transition system in Figure 4.1 illustrates in some detail which commands can trigger which mode transitions. The set of labels for each transition describes the commands that may cause it. With the exception of exit, if a command does not appear on any transitions originating from a mode, it is not permitted in that mode. The exit command, which causes the solver to quit, can be issued in any mode.

The solver must respond with an error when given a command not permitted in the current mode. Because of its level of abstraction, the transition system diagram in Figure 4.1 does not specify the conditions under which a specific command causes a transition to one mode as opposed to another; see Section 4.2 for details on that. Similarly, the diagram does not account for the fact that some specific options can be set only in certain modes. Such restrictions are described in Section 4.1.7.

4.1.2 Solver responses

Regular output, including responses and errors, produced by compliant solvers should be written to the regular output channel. Diagnostic output, including warnings, debugging, tracing, or progress information, should be written to the diagnostic output channel. These channels may be set with the ⟨set-option⟩ command (see Section 4.1.7 below). By default they are the standard output and standard error channels, respectively.

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

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

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

Any response which is not double-quoted and not parenthesized must be followed by at least one whitespace character (for example, a new line character).\(^{25}\)

Several options described in Section 4.1.7 below affect the printing of responses, in particular by suppressing the printing of success, or by redirecting the regular or diagnostic output channels.

\(^1\)Returning the empty string is allowed but discouraged because of its uninformative content.
4.1. GENERAL REQUIREMENTS

Errors and solver state. Solvers have two options when encountering errors. For both options, they first print an error message in the \texttt{general_response} format. Then, they may either immediately exit with a non-zero exit status, or continue accepting commands. In the second case, the solver’s state remains unmodified by the error-generating command, except possibly for timing and diagnostic information. In particular, the assertion stack, discussed in Section 4.1.4, is unchanged\(^{(26)}\)

The predefined \texttt{:error-behavior} attribute can be used with the \texttt{get-info} command to check which error behavior the tool supports (see Section 4.1.8 below).

4.1.3 Printing of terms and defined symbols

Several commands request the solver to print sets of terms. While some commands, naturally, place additional semantic requirements on these sets, the general syntactic requirement is that output terms must be well-sorted with respect to the current signature (as defined below in Section 4.1.4).

All output from a compliant solver should print any symbols defined with \texttt{define-sort} and \texttt{define-fun} just as they are, without replacing them by the expression they are defined to be equal to. This approach generally keeps the output from solvers much more compact than it would be if definitions were expanded.

4.1.4 The assertion stack

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

- \texttt{assertion stack}: the single stack of sets of assertions;
- \texttt{assertion level}: an element of the assertion stack (i.e., a set of assertions);
- \texttt{context}: the union of all the assertion levels on the assertion stack together with any global declarations (see Section 4.1.5);
- \texttt{current assertion level}: the assertion level at the top of the stack (i.e., the most recent);
- \texttt{first assertion level}: the first assertion level in the stack (i.e., the least recent);
- \texttt{current signature}: the signature determined by the logic specified with the most recent \texttt{set-logic} command and by the set of sort symbols and rank associations (for function symbols) in the current context.

Initially, when the solver starts, the assertion stack consists of a single element, the first assertion level, which is empty. While new assertions can be added to this set, the set itself cannot be removed from the stack with a pop operation. The following commands modify the current context:

\texttt{assert, declare-sort, declare-fun, declare-const, define-sort, define-fun, define-fun-rec, define-funs-rec, pop, push, reset, and reset-assertions}.
Symbol declarations and definitions

A number of commands allow the declaration or definition of a function or sort symbol. By default, these declarations and definitions are added to the current assertion level when the corresponding command is executed. Popping that assertion level removes them.\(^{(27)}\) As an alternative, declarations and definitions can all be made \textit{global} by running the solver with the option \texttt{:global-declarations} set to \texttt{true}. When running with this option set, all declarations and definitions become permanent. That is, they survive any pop operations on the assertion stack as well as invocations of \texttt{reset-assertions} and can only be removed by a global reset, achieved with the \texttt{reset} command.\(^{(28)}\)

Well-sortedness checks, required for commands that use sorts or terms, are always done with respect to the current signature. It is an error to declare or define a symbol that is already in the current signature. This implies in particular that, contrary to theory function symbols, user-defined function symbols cannot be overloaded.\(^{(29)}\)

In-line definitions

Any closed subterm \(t\) occurring in the argument(s) of a command \(c\) can be optionally annotated with a \texttt{:named} attribute; that is, it can appear as \((! t :\text{named } f)\) where \(f\) is a fresh function symbol from \(\langle \text{symbol} \rangle\). For such a command \(c\), let
\[
(! t_1 :\text{named } f_1),\ldots,(! t_n :\text{named } f_n)
\]
be the enumeration of all the named subterms of \(c\) obtained by the (depth-first) left-to-right post-order traversal of \(c\). The semantics of the command \(c\) is the same as the sequence of commands
\[
\begin{align*}
\text{(define-fun } f_1 () \sigma_1 t'_1) \\
\vdots \\
\text{(define-fun } f_n () \sigma_n t'_n)
\end{align*}
\]
\(c'\)
where, for each \(i = 1,\ldots,n\), (i) \(\sigma_i\) is the sort of \(t_i\) with respect to the current signature up to the declaration of \(f_i\), (ii) \(t'_i\) is the term obtained from \(t_i\) by removing all its \texttt{:named} annotations, and (iii) \(c'\) is similarly obtained from \(c\) by removing all its \texttt{:named} annotations.

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

Solver options

Solver options may be set using the \texttt{set-option} command, and their current values can be obtained using the \texttt{get-option} command. If a solver does not support the setting of a particular option, for either command it should output \texttt{unsupported}.\(^{(28)}\)
4.1. GENERAL REQUIREMENTS

Solver-specific option names are allowed and indeed expected. A set of standard options is presented in this subsection; refer to Figure 3.7 for their format. We discuss each option below, specifying also their default values and whether or not compliant solvers are required to support them. It is understood that if a solver does not support one of the optional standard options below, it behaves as if that option was permanently set to its default value.

Some options can be set only when the solver is in start mode. We list the mode when that is the case. Attempting to set an option when the solver is not in a permitted mode should trigger an error response. Each option starting with the produce- prefix is a Boolean option that enables a specific command. When such an option is set to false, calling the corresponding command should trigger an error response.

The set of standard options is likely to be expanded or otherwise revised as further desirable common options and kinds of information across tools are identified.

:diagnostic-output-channel   default: "stderr"  support: required
   The argument is a string consisting of the name of a file to be used subsequently as the diagnostic output channel. The input value "stderr" is interpreted specially to mean the solver’s standard error channel. With other filenames, subsequent solver output is to be appended to the named file (and the file should be first created if it does not already exist).

:global-declarations   default: false  support: optional  mode: start
   If the solver supports this option, setting it to true causes all declarations and definitions to be global (permanent) as opposed to being added to the current assertion level.

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

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

:produce-assertions   default: false  support: optional  mode: start
   If the solver supports this option, setting it to true enables the get-assertions command. This option was called interactive-mode in previous versions.

:produce-assignments   default: false  support: optional  mode: start
   If supported, this enables the command get-assignment.

:produce-models   default: false  support: optional  mode: start
   If supported, this enables the commands get-value and get-model.

:produce-proofs   default: false  support: optional  mode: start
   If supported, this enables the command get-proof.

:produce-unsat-assumptions   default: false  support: optional  mode: start
   If supported, this enables the command get-unsat-assumptions.
4.1. GENERAL REQUIREMENTS

:produce-unsat-cores  default: false  support: optional  mode: start
    If supported, this enables the command get-unsat-core.

:random-seed  default: 0  support: optional  mode: start
    The argument is a numeral for the solver to use as a random seed, in case the solver
    uses (pseudo-)randomization. The default value of 0 means that the solver can use
    any random seed—possibly even a different one for each run of the same script. The
    intended use of the option is to force the solver to produce identical results whenever
    given identical input (including identical non-zero seeds) on repeated runs of the solver.

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

:reproducible-resource-limit  default: 0  support: optional
    If the solver supports this option, setting it to 0 disables it. Setting it a non-zero numeral
    $n$ will cause each subsequent check command to terminate within a bounded amount
    of time dependent on $n$. The internal implementation of this option and its relation
    to run time or other concrete resources can be solver-specific. However, it is required
    that the invocation of a check command return unknown whenever the solver is unable
    to determine the satisfiability of the formulas in the current context within the current
    resource limit. Setting a higher value of $n$ should allow more resources to be used,
    which may cause the command to return sat or unsat instead of unknown. Furthermore,
    the returned result should depend deterministically on $n$; specifically, it should be the
    same every time the solver is run with the same sequence of previous commands on the
    same machine (and with an arbitrarily long external time out). If the solver makes use
    of randomization, it may require the :random-seed option to be set to a value other than
    0 before :reproducible-resource-limit can be set to a positive value.\(^{30}\)

:verbosity  default: 0  support: optional
    The argument is a numeral controlling the level of diagnostic output produced by the
    solver. All such output should be written to the diagnostic output channel\(^{31}\) which can
    be set and later changed via the diagnostic-output-channel option. An argument of 0
    requests that no such output be produced. Higher values correspond to more verbose
    output.

4.1.8 Solver information

The format for responses to get-info commands, both for standard and solver-specific information flags, is defined by the ⟨get_info_response⟩ category in Figure 3.9. The standard get-info flags and specific formats for their corresponding responses are given next.

:all-statistics  support: optional  mode: sat, unsat
    Solvers reply with a parenthesis-delimited sequence of ⟨info_response⟩ values (see Figure 3.9) providing various statistics on the execution of the most recent check command.
4.2. COMMANDS

No standard statistics are defined for the time being, so they are all solver-specific. Executions of `get-info` with `:all-statistics` are allowed only when the solver is in `sat` or `unsat` mode.

`:assertion-stack-levels` support: optional
The response is a pair of the form `(assertion-stack-levels n)` where `n` is a numeral indicating the current number of levels in the assertion stack besides the first assertion level.

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

`:error-behavior` support: required
The response is a pair of the form `(error-behavior r)` where `r` is either `immediate-exit` or `continued-execution`. A response of `immediate-exit` indicates that the solver will exit immediately when an error is encountered. A response of `continued-execution` indicates that when an error is encountered, the solver will return to the state it was in immediately before the command triggering the error, and continue accepting and executing new commands. See Section 4.1.2 for more information.

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

`:reason-unknown` support: optional mode: sat
Executions of `get-info` with `:reason-unknown` are allowed only when the solver is in sat mode following a check command whose response was `unknown`. The response is a pair of the form `(reason-unknown r)` where `r` is an element of `⟨reason_unknowm⟩` giving a short reason why the solver could not successfully check satisfiability. In general, this reason can be provided by a solver-defined s-expression. Two predefined s-expressions are `memout`, for out of memory, and `incomplete`, which indicates that the solver knows it is incomplete for the class of formulas containing the most recent check query.

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

4.2 Commands

The full set of commands and their expected behavior are described in this section. Commands may impose restrictions on their arguments as well as restrictions on when they can be issued. Unless otherwise specified, the solver is required to produce an error when any of these restrictions is violated. Figure 4.1 describes the commands permitted in each execution mode and the mode transitions each command may trigger. We specify below the conditions under which a command triggers one mode transition versus another only for commands that may trigger more than one transition.
4.2. COMMANDS

4.2.1 (Re)starting and terminating

(reset) resets the solver completely to the state it had after it was started and before it started reading commands.\(^{(34)}\)

(set-logic \(l\)) tells the solver what logic, in the sense of Section 5.5, is being used. The argument \(l\) can be the name of a logic in the SMT-LIB catalog or of some other solver-specific logic. The effect of the command is to add globally (and permanently) a declaration of each sort and function symbol in the logic.

The argument \(l\) can also be the predefined symbol \(\text{ALL}\). With this argument, the solver sets the logic to the most general logic it supports.\(^{(35)}\) Note that while the reaction to (set-logic \(\text{ALL}\)) is the same for every compliant solver, the chosen logic is solver-specific.

We refer to the logic set by the most recent set-logic command as the current logic.

(set-option \(o\) \(v\)) sets a solver’s option \(o\) to a specific value \(v\). More details on predefined options and required behavior are provided in Section 4.1.7. In general, if a solver does not support the setting of a particular option, its response to this command should be unsupported. If the option is one of the predefined ones it should also leave it unchanged from its default value. The effect of setting a supported option is immediate. In particular, for options that affect the solver’s output, such as :diagnostic-output-channel, :regular-output-channel and :print-success, the effect applies already to the output of the very command that is setting the option.

Note that some of the options defined in Section 4.1.7 may only be set in start mode.\(^{(36)}\)

(exit) instructs the solver to exit.

4.2.2 Modifying the assertion stack

(push \(n\)) pushes \(n\) empty assertion levels onto the assertion stack.\(^2\) If \(n\) is 0, no assertion levels are pushed.

(pop \(n\)) where \(n\) is smaller than the number of assertion levels in the stack, pops the \(n\) most-recent assertion levels from the stack.\(^3\) Note that the first assertion level, which is not created by a push command, cannot be popped.

(reset-assertions) removes from the assertion stack all assertion levels beyond the first one. In addition, it removes all assertions from the first assertion level. Declarations and definitions resulting from the set-logic command are unaffected (because they are global). Similarly, if the option :global-declarations has value true at the time the command is executed, then all declarations and definitions remain unaffected. Note that any information set with set-option commands is preserved in any case.

\(^2\)Typically, \(n = 1\).

\(^3\)When \(n\) is 0, no assertion levels are popped.
4.2.3 Introducing new symbols

The next seven commands allow one to introduce new sort or function symbols by providing them with a rank declaration (\texttt{declare-sort}, \texttt{declare-fun} and \texttt{declare-const}) or also with a definition (\texttt{define-sort}, \texttt{define-fun}, \texttt{define-fun-rec} and \texttt{define-funs-rec}). We refer to the former as \textit{user-declared} symbols and the latter as \textit{user-defined} symbols. Declarations and definitions are made global (permanent) or are added to the current assertion level depending on whether the \texttt{:global-declarations} option is set to \texttt{true} or not.

\texttt{(declare-sort }s\texttt{ }n\texttt{)} adds sort symbol \(s\) with associated arity \(n\). It is an error if \(s\) is a sort symbol present in the current signature.

\texttt{(define-sort }s\texttt{ }\langle u_1 \cdots u_n \rangle\texttt{ }\tau\texttt{)} with \(n \geq 0\) adds sort symbol \(s\) with associated arity \(n\). Subsequent well-sortedness checks must treat a sort term like \(\langle s \sigma_1 \cdots \sigma_n \rangle\) as an abbreviation for the term obtained by simultaneously substituting \(\sigma_i\) for \(u_i\), for \(i \in \{1, \ldots, n\}\), in \(\tau\).

The command reports an error if \(s\) is a sort symbol already present in the current signature or if \(\tau\) is not a well-defined parametric sort with respect to the current signature. This restriction prohibits (meaningless) circular definitions where \(\tau\) contains \(s\).

\texttt{(declare-fun }f\texttt{ }\langle \sigma_1 \cdots \sigma_n \rangle\texttt{ }\sigma\texttt{)} with \(n \geq 0\) adds a new symbol \(f\) with associated rank \(\sigma_1 \cdots \sigma_n \sigma\). The command reports an error if a function symbol with name \(f\) is already present in the current signature.

\texttt{(declare-const }f\texttt{ }\sigma\texttt{)} abbreviates the command \texttt{(declare-fun }f\texttt{ }\langle \rangle\texttt{ }\sigma\texttt{)}.

\texttt{(declare-datatypes }\langle (\delta_1 \ k_1) \cdots (\delta_n \ k_n) \rangle\texttt{ }\langle d_1 \cdots d_n \rangle\texttt{)} with \(n > 0\) introduces \(n\) algebraic datatypes \(\delta_1, \ldots, \delta_n\) with respective arities \(k_1, \ldots, k_n\) and declarations \(d_1, \ldots, d_n\). Let \(\delta = \delta_i\), \(k = k_i\) and \(d = d_i\) for \(i \in \{1, \ldots, n\}\). If \(k > 0\) then \(d\) is an expression of the form \texttt{(par }\langle u_1 \cdots u_k \rangle\texttt{ }l\texttt{)} where \(u_1, \ldots, u_k\) are sort parameters; otherwise, it is just \(l\). In either case, \(l\) is a (parenthesis-delimited) list of \textit{one or more} expressions of the form

\[
(c \ (s_1 \ \tau_1) \cdots \ (s_m \ \tau_m))
\]

where (i) \(m \geq 0\), (ii) \(c\) is a symbol, a \textit{constructor for }\(\delta\), and (iii) for each \(j = 1, \ldots, m\), \(s_j\) is a symbol, a \textit{selector for }\(c\), and (iv) \(\tau_j\) is a sort term that contains no occurrences of \(\delta_1, \ldots, \delta_n\) below its top symbol. In the parametric case, the terms \(\tau_i\) can contain the sort parameters \(u_1, \ldots, u_k\). The datatype \(\delta\) must be \textit{well founded} in the following inductive sense: it must have a constructor of rank \(\tau_1 \cdots \tau_m \delta\) such that \(\tau_1 \cdots \tau_m\) does not contain any of the datatypes from \(\{\delta_1, \ldots, \delta_n\}\) or, if it does contain some, they are well founded. A compliant solver must return an error in response to invocations of this command that do not satisfy all of the restrictions above.

In the parametric case, the command has the effect of declaring each \(\delta\) as a sort symbol of arity \(k\); each constructor \(c\) as a function symbol of parametric rank \(\tau_1 \cdots \tau_m \delta\); and each \(s_i\) as a function symbol of parametric rank \(\delta \tau_i\). The non-parametric case is analogous.
Note that the sort terms $\tau_1, \ldots, \tau_m$ can contain any previously defined sort symbol as well as any of the datatypes $\delta_1, \ldots, \delta_n$, as long as those datatypes are well founded. This allows the declaration of recursive and mutually recursive datatypes.

On successfully executing this command, for each constructor $c$ in a declared datatype $\delta$, the solver will also automatically declare a tester with rank $\delta \text{Bool}$. The tester’s name is an indexed identifier (see Section 3.3) of the form (_ is c).

```lisp
; an enumeration datatype
(declare-datatypes (Color 0) ( (red) (green) (blue) ))
; testers: (_ is red), (_ is green)

; integer lists with "empty" and "insert" constructors
(declare-datatypes (IntList 0) ( (empty) (insert (head Int) (tail IntList) )))
; testers: (_ is empty), (_ is insert)

; parametric lists with "nil" and "cons" constructors
(declare-datatypes (List 1) ( (par (T) ( (nil) (cons (car T) (cdr (List T)))))))

; option datatype
(declare-datatypes (Option 1) ( (par (X) ( (none) (some (val X)) ))))

; parametric pairs
(declare-datatypes (Pair 2) ( (par (X Y) ( (pair (first X) (second Y)) ))))

; two mutually recursive datatypes
(declare-datatypes (Tree 1) (TreeList 1) ( (Tree (par (X) ( (node (value X) (children (TreeList X)) ))))
; TreeList
 (par (Y) ( (empty) (insert (head (Tree Y)) (tail (TreeList Y)))))))
```

Since $\delta_1, \ldots, \delta_n$ are sort symbols, none of them can be a previously declared sort symbol. Similarly, constructors and selectors are function symbols, so none of them can be a previous declared/defined function symbol. This has the effect of also prohibiting, for instance, the use of the same constructor in different datatypes or the use of repeated instances of the same selector in the same datatype.

(declare-datatype $\delta \ d$) is an abbreviation of

\[
\text{(declare-datatypes } ((\delta \ 0)) (d))
\]
4.2. COMMANDS

if \( \delta \) is not parametric, and an abbreviation of

\[
(\text{declare-datatypes } ((\delta k)) (d))
\]

if \( d \) has the form \((\text{par } (u_1 \cdots u_k) l)\). This command provides a simpler syntax for defining a single datatype.

\[
\begin{align*}
&\text{; an enumeration datatype} \\
&(\text{declare-datatype Color ( (red) (green) (blue) )))
\end{align*}
\]

\[
\begin{align*}
&(\text{declare-datatype IntList} \\
&(\text{ (empty) } \\
&(\text{ (insert (head Int) (tail IntList) )))
\end{align*}
\]

\[
\begin{align*}
&(\text{declare-datatype List (par (E) } \\
&(\text{ (nil) } \\
&(\text{ (cons (car E) (cdr (List E))) )}))
\end{align*}
\]

\[
\begin{align*}
&(\text{declare-datatype Option (par (X) } \\
&(\text{ (none) } \\
&(\text{ (some (val X) ))})
\end{align*}
\]

\[
\begin{align*}
&(\text{declare-datatype Pair (par (X Y) } \\
&(\text{ (pair (first X) (second Y)) )})
\end{align*}
\]

\[
\begin{align*}
&(\text{define-fun } f ((x_1 \sigma_1) \cdots (x_n \sigma_n)) \sigma t) \text{ with } n \geq 0 \text{ and } t \text{ not containing } f \text{ is semantically equivalent to the command sequence }
\end{align*}
\]

\[
\begin{align*}
&(\text{declare-fun } f (\sigma_1 \cdots \sigma_n) \sigma) \\
&(\text{assert (forall } ((x_1 \sigma_1) \cdots (x_n \sigma_n)) (= (f x_1 \cdots x_n) t))
\end{align*}
\]

Note that the restriction on \( t \) prohibits recursive or mutually recursive definitions, which are instead provided by \texttt{define-fun-rec} and \texttt{define-funs-rec}. The command reports an error if a function symbol with name \( f \) is already present in the current signature or if the argument \( t \) is not a well-sorted term of sort \( \sigma \) with respect to the current signature extended with the sort associations \((x_1 : \sigma_1), \ldots, (x_n : \sigma_n)\).

\[
\begin{align*}
&(\text{define-funs-rec } (d_1 \cdots d_m) (t_1 \cdots t_m)), \text{ where } m > 0 \text{ and for } i = 1, \ldots, m, d_i \text{ has the form }
\end{align*}
\]

\[
\begin{align*}
&(f_i ((x_{i,1} \sigma_{i,1}) \cdots (x_{i,n_i} \sigma_{i,n_i})) \sigma_i)
\end{align*}
\]

with \( n_i \geq 0 \) and \( f_1, \ldots, f_m \) pairwise distinct, is semantically equivalent to the command sequence
4.2. COMMANDS

\[
(\text{declare-fun } f_1 (\sigma_{1,1} \cdots \sigma_{1,n_1}) \sigma_1) \\
\vdots \\
(\text{declare-fun } f_m (\sigma_{m,1} \cdots \sigma_{m,n_m}) \sigma_m) \\
(\text{assert } (\forall ((x_{1,1} \sigma_{1,1}) \cdots (x_{1,n_1} \sigma_{1,n_1})) \ (f_1 x_{1,1} \cdots x_{1,n_1}) t_1)) \\
\vdots \\
(\text{assert } (\forall ((x_{m,1} \sigma_{m,1}) \cdots (x_{m,n_m} \sigma_{m,n_m})) \ (f_m x_{m,1} \cdots x_{m,n_m}) t_m)).
\]

This command can be used to define multiple functions recursively, in particular, mutually recursively.\(^{41}\) Mutual recursion is possible since each term \(t_i\) can contain any applications of \(f_1, \ldots, f_m\).

Note that, according to the semantics above, \texttt{define-funs-rec} imposes no requirements that each \(f_i\) be terminating (a meaningless notion in our context) or even well-defined.\(^4\) The only requirement is on the well-sortedness of the definitions.

The command reports an error if for any \(i \in \{1, \ldots, m\}\) a function symbol with name \(f_i\) is already present in the current signature or if \(t_i\) is not a well-sorted term of sort \(\sigma_i\) with respect to the current signature extended with the sort associations \((f_1 : \sigma_{1,1} \cdots \sigma_{1,n_1} \sigma_1), \ldots, (f_m : \sigma_{m,1} \cdots \sigma_{m,n_m} \sigma_m)\) and \((x_{i,1} : \sigma_{i,1}), \ldots, (x_{i,n_i} : \sigma_{i,n_i})\).

\[
(\texttt{define-fun-rec } f \ ((x_1 \sigma_1) \cdots (x_n \sigma_n)) \sigma) \text{ is an abbreviation of } \\
(\texttt{define-funs-rec } ((f ((x_1 \sigma_1) \cdots (x_n \sigma_n)) \sigma)) \ (t))
\]

It provides a simpler syntax to define individual recursive functions.

### 4.2.4 Asserting and inspecting formulas

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

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

\(\texttt{get-assertions}\) causes the solver to print the current set of all asserted formulas as a sequence of the form \((f_1 \cdots f_n)\). Each \(f_i\) is a formula syntactically identical, modulo whitespace, to one of the formulas entered with an \texttt{assert} command and currently in the context. Solvers are not allowed to print formulas equivalent to or derived from the asserted formulas.\(^ {42}\)

The command can be issued only if the \texttt{:produce-assertions} option, which is set to \texttt{false} by default, is set to \texttt{true} (see Section 4.1.7).

---

\(^4\)In fact, it is even possible, although certainly not desirable, to have a definition like \(\texttt{define-funs-rec } ((f ((x \texttt{Bool})) \texttt{Bool})) \ (\texttt{not} (f x))\), which makes the set of formulas in the context unsatisfiable.
4.2.5 Checking for satisfiability

(check-sat) is an abbreviation of (check-sat-assuming ()).

(check-sat-assuming (a₁ ⋯ aₙ)) where n ≥ 0 and a₁, ⋯, aₙ are terms of sort $\text{Bool}$, instructs the solver to check whether the conjunction of all the formulas in the current context and the assumptions $a₁, ⋯, aₙ$ is satisfiable in the extension of the current logic with all the current user-declared and user-defined symbols. The assumptions $a₁, ⋯, aₙ$ can only have the form $p$ or (not $p$) where $p$ is a user-declared/defined Boolean constant, i.e., a nullary function symbol of sort $\text{Bool}$.

Conceptually, this command asks the solver to search for a model of the logic that satisfies all the currently asserted formulas as well as the current assumptions. When it has finished attempting to do this, the solver should reply on its regular output channel (see Section 4.1.2) using the response format defined by $\langle$check_sat_response$\rangle$ in Figure 3.9.

A sat response indicates that the solver has found a model, an unsat response that the solver has established there is no model, and an unknown response that the search was inconclusive—because of resource limits, solver incompleteness, or other reasons.

On reporting sat or unknown the solver should move to sat mode—and then respond to get-assignment, get-model, and get-value commands provided that the corresponding enabling option is set to true. On reporting unsat, it should move to unsat mode—and then respond to get-proof, get-unsat-assumptions, and get-unsat-core commands provided that the corresponding enabling option is set to true.

Regardless of how it is implemented internally, a check-sat-assuming command should preserve the current context in the sense that at the end of the command’s execution the context should be the same as it was right before the execution.

Note that a check-sat-assuming command can be issued also when the solver is already in sat or unsat mode (in this case, the context is necessarily the same as for the previous check command). However, it is possible for the solver to switch from sat to unsat mode or vice versa if the latest command has a different set of assumptions from the previous one.

4.2.6 Inspecting models

The next three commands can be issued only when the solver is in sat mode, and provide information related to the most recent check command. In that case, the solver will have identified a model $\mathbf{A}$ (as defined in Section 5.3) of the current logic, and produces responses with respect to that same model until it receives the next check command or it exits the sat mode, whichever comes first. The model $\mathbf{A}$ is required to satisfy all currently asserted formulas and current assumptions only if the most recent check command reported sat.

The internal representation of the model $\mathbf{A}$ is not exposed by the solver. Similarly to an abstract data type, the model can be inspected only through the three commands below. As a consequence, it can even be partial internally and extended as needed in response to successive invocations of some of these commands.⁵

⁵In that case, of course, the solver has to be sure that its partial model can be indeed extended as needed.
(get-value \((t_1 \cdots t_n)\)) where \(n \geq 0\) and each \(t_i\) is a well-sorted closed quantifier-free term, returns for each \(t_i\) a value term \(v_i\)\(^6\) that is equivalent to \(t_i\) in the current model \(A\) (see above). Specifically, \(v_i\) has the same sort as \(t_i\), and \(t_i\) is interpreted the same way as \(v_i\) in \(A\). The values are returned as a sequence of pairs of the form \(((t_1 v_1) \cdots (t_n v_n))\).

The terms \(v_1, \ldots, v_n\) are allowed to contain symbols not in the current signature only if they are abstract values, i.e., constant symbols starting with the special character \(\&\).\(^5\)

Since these are solver-defined, their sort is not known to the user. Therefore, additionally, each occurrence of an abstract value \(a\) of sort \(\sigma\) in \(v_1, \ldots, v_n\) has to be contained in a term of the form \((\text{as } a \sigma)\) which makes the sort explicit.

Note that the returned abstract values are used only to express information about the current model \(A\). They cannot be used in later assert commands since they are neither theory symbols nor user-defined ones. However, they can be used in later get-value commands on the same model.

There is no requirement that different permutations of the same set of get-value calls produce the same value for the input terms. The only requirement is that syntactically different values of the same sort returned by the solver have different meaning in the model.\(^7\)

The command can be issued only if the :produce-models option, which is set to false by default, is set to true (see Section 4.1.7).

(get-assignment) can be seen as a light-weight and restricted version of get-value that asks for a truth assignment for a selected set of previously entered formulas.\(^46\)

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

The command can be issued only if the :produce-assignments option, which is set to false by default, is set to true (see Section 4.1.7).

(get-model) returns a list \((d_1 \cdots d_k)\) of definitions specifying all and only the current user-declared function symbols \(\{g_1, \ldots, g_m\}\) in the current model \(A\). The interpretation of each symbol is provided in exactly one of the definition \(d_1, \ldots, d_k\). The define commands \(d_1, \ldots, d_k\) have one of the following forms: \(^{47}\)

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

  where \(n \geq 0\), \(f\) has rank \(\sigma_1 \cdots \sigma_n \sigma\), \(t\) is a term not containing \(f\), and the formula

  \[
  \text{forall } ((x_1 \sigma_1) \cdots (x_n \sigma_n)) \ (\forall \ (x_1 \cdots x_n) \ (t))
  \]

  is well-sorted and satisfied by \(A\). The term \(t\) is expected, although not required, to be a value when \(f\) is a constant (i.e., when \(n = 0\)).

---

\(^{6}\)Recall that value terms are particular ground terms defined in a logic for each sort (see Subsection 5.5.1).

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

- `(define-funs-rec ((f₁ ((x₁,₁ σ₁,₁) ⋱ (x₁,n₁ σ₁,n₁)) σ₁) ⋱ (fₚ ((xₚ,₁ σₚ,₁) ⋱ (xₚ,nₚ σₚ,nₚ)) σₚ)) (t₁ ⋱ tₚ))`
  where \( n_i > 0 \) for \( i = 1, \ldots, p \), and the formula
  
  \[
  \text{(and (forall ((x₁,₁ σ₁,₁) ⋱ (x₁,n₁ σ₁,n₁)) (= (f₁ x₁,₁ ⋱ x₁,n₁) t₁))} \\
  \quad \vdots \\
  \quad (\text{forall ((xₚ,₁ σₚ,₁) ⋱ (xₚ,nₚ σₚ,nₚ)) (= (fₚ xₚ,₁ ⋱ xₚ,nₚ) tₚ))))
  \]

  is well-sorted and satisfied by \( A \).

- `(define-fun-rec f ((x₁ σ₁) ⋱ (xₙ σₙ)) σ t)` where \( n > 0 \) and the formula
  
  \[
  \text{(forall ((x₁ σ₁) ⋱ (xₙ σₙ)) (= (f x₁ ⋱ xₙ) t))}
  \]

  is well-sorted and satisfied by \( A \).

Similarly to the response of `get-value`, the terms \( t, t₁, \ldots, t_m \) above are allowed to contain symbols not in the current signature only if they are abstract values. Moreover, each occurrence of an abstract value \( a \) of sort \( σ \) in \( t, t₁, \ldots, t_m \) has to be contained in a term of the form \((\text{as } a \ σ)\).

Later versions of the standard may impose stronger requirements on the returned definitions. For now there is only an expectation that, when possible, the solver will provide definitions that have a unique interpretation over the current signature.\(^{(48)}\)

The command can be issued only if the `:produce-models` option, which is set to `false` by default, is set to `true` (see Section 4.1.7).

4.2.7 Inspecting proofs

The next three commands can be issued only when the solver is in `unsat` mode, and provide information related to the most recent check command (which produced an `unsat` response).

- `(get-unsat-assumptions)` returns a subset \( a₁, \ldots, a_n \) of the assumptions in the most recent `check-sat-assuming` command. These assumptions are such that issuing the command `(check-sat-assuming (a₁ ⋱ a_n))` instead would have still produced an `unsat` response. The returned sequence is not required to be minimal.\(^{(49)}\)

  The command can be issued only if the `:produce-unsat-assumptions` option, which is set to `false` by default, is set to `true` (see Section 4.1.7).

- `(get-proof)` asks the solver for a proof of unsatisfiability for the set of all formulas in the current context. The command can be issued only if the most recent check command had an empty set of assumptions. The solver responds by printing a refutation proof on its regular output channel. The format of the proof is solver-specific.\(^{(50)}\) The only requirement is that, like all responses, it be a member of \( \langle \text{s_expr} \rangle \).

  The command can be issued only if the `:produce-proofs` option, which is set to `false` by default, is set to `true` (see Section 4.1.7).
(get-unsat-core) asks the solver to identify an unsatisfiable core, a subset of all the formulas in the current context that is unsatisfiable by itself. The command can be issued only if the most recent check command had an empty set of assumptions. The solver selects from the unsatisfiable core only those formulas that have been asserted with a command of the form (assert (! t :named f)), and returns a sequence \((f_1 \cdots f_n)\) of those labels. Unlabeled formulas in the unsatisfiable core are simply not reported.\(^{(51)}\)

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

The command can be issued only if the :produce-unsat-cores option, which is set to false by default, is set to true (see Section 4.1.7).

4.2.8 Inspecting settings

(get-info f) where \(f\) is an element of \(\langle\)info_flag\(\rangle\) outputs solver information as specified in Section 4.1.8. If a solver does not support a (standard or non-standard) flag \(f\), it just outputs unsupported.

(get-option o) outputs the current value of a solver’s option \(o\) as an element of \(\langle\)attribute_value\(\rangle\). The form of that value depends on the specific option. More details on standard options and required behavior are provided in Section 4.1.7. If a solver does not support the setting of a standard option \(o\), the command outputs the option’s default value. For an unsupported non-predefined option the command outputs unsupported.

4.2.9 Script information

(echo s) where \(s\) is a string literal, simply prints back \(s\) as is—including the surrounding double-quotes.\(^{(52)}\)

(set-info a) where \(a\) is an element of \(\langle\)attribute\(\rangle\) has no effect on the assertion stack. Its only purpose is to allow the insertion of structured meta information in a script.\(^{(53)}\)

Typically then, a solver will just parse the command and do nothing with it, except for printing a response (success or an error, for instance, if the argument is not an element of \(\langle\)attribute\(\rangle\)).

There is only a small number of predefined set-info attributes, which are described below together with their possible values. These attributes are used in particular in the official SMT-LIB benchmarks at www.smt-lib.org.

:smt-lib-version possible values: a decimal.

The value of this attribute is the version of SMT-LIB used by the benchmark (e.g., 2.6). For benchmarks in the official repository a call to set-info with this attribute can occur only as the first command of a script.
4.2. COMMANDS

:sourcets possible values: a string or a quoted symbol.

The value of this attribute is a textual description of the benchmark's source, containing, as appropriate, such information as the name of person(s) who generated the benchmark; the generation date; the tool that generated it; the intended application; the solvers that were initially used or targeted to check the benchmarks; references to related publications; any other information the benchmark author deems useful.

:categoryts possible values: "crafted", "random", and "industrial".

The value "crafted" indicates that the benchmark was hand-crafted while "random" indicates that it was generated by a random process; "industrial" is reserved for everything else.\(^{54}\)

:licensets possible values: a string.

This is a description of the license under which the benchmark is distributed. It can be the actual text of the license, or the URL of a web site containing the description.

:statusits possible values: sat, unsat, and unknown.

Each occurrence of the command (set-info :status sat) (respectively, (set-info :status unsat)) indicates that the next check command in the script is expected to return sat (respectively, unsat). More precisely, the expected value of a check command in a script is the one indicated by the most recent command of the form (set-info :status v) in the script. The value unknown is used when the expected value is not known.\(^{55}\)

Remark 7 (set-info and get-info are unrelated). Contrary to what their names might suggest, set-info and get-info are not related. The first command is used to store information about a script, the second to obtain solver-specific information.\(^{56}\)
The underlying logic of the SMT-LIB language is a variant of many-sorted first-order logic (FOL) with equality [Man93, Gal86, End01], although it incorporates some syntactic features of higher-order logics: in particular, the identification of formulas with terms of a distinguished Boolean sort, and the use of sort symbols of arity greater than 0.

These features make for a more flexible and syntactically more uniform logical language. However, while not exactly syntactic sugar, they do not change the essence of SMT-LIB logic with respect to traditional many-sorted FOL. Quantifiers are still first-order, the sort structure is flat (no subsorts), the logic’s type system has no function (arrow) types, no type quantifiers, no dependent types, no provisions for parametric or subsort polymorphism. The only polymorphism is of the ad-hoc variety (a function symbol can be given more than one rank), although there is a syntactical mechanism for approximating parametric polymorphism. As a consequence, all the classical meta-theoretic results from many-sorted FOL apply to SMT-LIB logic when considered in its full generality, that is, with no restrictions on the possible models other than those imposed by the Core background theory introduced in Subsection 3.7.1. Those results still hold with recursively axiomatizable background theories, i.e., theories defined as the set of all models of a recursive set of closed formulas (or axioms). As pointed out in Section 3.7.1 the ability to use non-recursively axiomatizable theories as background theories actually gives SMT-LIB logic the expressive power of higher-order logics. However, a formal treatment of this aspect is beyond the scope of this document.

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

(Sorts) \[ \sigma ::= s \sigma^* \]

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

Figure 5.1: Abstract syntax for sort terms

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

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

- an infinite set \( S \) of \textit{sort symbols} \( s \) containing the symbol \texttt{Bool},
- an infinite set \( U \) of \textit{sort parameters} \( u \),
- an infinite set \( X \) of \textit{variables} \( x \),
- an infinite set \( F \) of \textit{function symbols} \( f \) containing the symbols \( \approx, \land, \text{ and } \neg \),
- an infinite set \( A \) of \textit{attribute names} \( a \),
- an infinite set \( V \) of \textit{attribute values} \( v \),
- the set \( W \) of \textit{Unicode character strings} \( w \),
- a two-element set \( B = \{\text{true}, \text{false}\} \) of \textit{Boolean values} \( b \),
- the set \( N \) of \textit{natural numbers} \( n \),
- an infinite set \( TN \) of \textit{theory names} \( T \),
- an infinite set \( L \) of \textit{logic names} \( L \).

5.1 The language of sorts

In many-sorted logics, terms are typed, or \textit{sorted}. Each sort, which stands for a non-empty set of elements, is denoted by a sort symbol. In SMT-LIB logic, the language of sorts is extended from sort symbols to \textit{sort terms} built with symbols from the set \( S \) above. Formally, we have the following.

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

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

We say that \( s \in S \) has (or is of) \textit{arity} \( n \) if \( ar(s) = n \).
5.2 The language of terms

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

Function symbol declarations in theory declarations (defined later) use also \textit{parametric sorts}. These are defined similarly to sorts above except that they can be built also over a further set \( \mathcal{U} \) of \textit{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 \textbf{List (Array} \( u_1 \) (List \( u_2 \)) \textbf{)} and all of its subexpressions are parametric sorts.

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

5.2 The language of terms

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

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

Binders

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

- the prefix construct \( \exists x_1: \sigma_1 \cdots x_n: \sigma_n \_ \) is a \textit{sorted existential binder (or existential quantifier)} for \( x_1, \ldots, x_n \);
- the prefix construct \( \forall x_1: \sigma_1 \cdots x_n: \sigma_n \_ \) is a \textit{sorted universal binder (or universal quantifier)} for \( x_1, \ldots, x_n \);
- the mixfix construct \( \textbf{let} \ x_1 = \_ \cdots x_n = \_ \ \textbf{in} \_ \) is a \textit{(parallel-)let binder} for \( x_1, \ldots, x_n \).
• the mixfix construct `match p₁ → _ · · · pₙ → _` is a *match binder* for the variables that occur in the pattern \( p_i \) for each \( i = 1, \ldots, n \).

Occurrences of variables in terms are defined to be *free* or *bound* as in the case of the concrete syntax; the scope of each bound variable is defined similarly as well (see Subsection 3.6.3). 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 the binders. Logical connectives for negation, conjunction and so on and the equality symbol, which we denote here by \( \approx \), are just function symbols of the basic theory *Core*, implicitly included in all SMT-LIB theories (see Subsection 3.7.1).

**Annotations**

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 \( a \in A \) or a pair the form \( a = v \) where \( a \in A \) and \( v \) is an attribute value in \( V \).

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

### 5.2.1 Signatures

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

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

- a set \( \Sigma^S \subseteq S \) of *sort* symbols containing \( \text{Bool} \);
- a set \( \Sigma^F \subseteq F \) of *function* symbols;
- a distinguished finite set \( \Sigma^C \subseteq \Sigma^F \) of *constructor* symbols;
- a distinguished finite set \( \Sigma^G \subseteq \Sigma^F \) of *selector* symbols, disjoint with \( \Sigma^C \);
- a distinguished finite set \( \Sigma^T \subseteq \Sigma^F \) of *tester* symbols, disjoint with \( \Sigma^C \) and \( \Sigma^G \), and with the same cardinality as \( \Sigma^C \);
- a total mapping \( \text{con}_C : \Sigma^S \to 2^{\Sigma^C} \), assigning a (possibly empty) set of constructors to each sort symbol;
- a total mapping \( \text{ar} : \Sigma^S \to \mathbb{N} \), assigning an arity to each sort symbol, with \( \text{ar}(\text{Bool}) = 0 \);

---

\(^1\)At this abstract level, the syntax of attribute values is intentionally left unspecified.
5.2. THE LANGUAGE OF TERMS

- a total mapping \( \text{sel}_\Sigma : \Sigma^C \to (\Sigma^G)^* \), assigning a sequence of \( n \) distinct selectors to each constructor of arity \( n \) so that no selector is assigned to more than one constructor;

- a bijective mapping \( \text{tes}_\Sigma : \Sigma^C \to \Sigma^T \), assigning a tester to each constructor;

- a partial mapping from \( \mathcal{X} \) to \( \text{Sort}(\Sigma^S) \), assigning a sort to some of the variables in \( \mathcal{X} \);\(^2\)

- a left-total \( \text{ranking} \) relation \( R \) from \( \Sigma^F \) to \( \text{Sort}(\Sigma^S)^+ \), assigning at least one \( \text{rank} \) to each function symbol and such that

1. for all constructors \( c \in \Sigma^C \), selectors \( g_1 \cdots g_n = \text{sel}_\Sigma(c) \), and sorts \( \sigma_1, \ldots, \sigma_n, \sigma \in \text{Sort}(\Sigma^S) \), if \( (c, \sigma_1 \cdots \sigma_n \sigma) \in R \) then \( (g_i, \sigma \sigma_i) \in R \) for all \( i = 1, \ldots, n \);

2. for all constructors \( c \in \Sigma^C \) and testers \( p = \text{tes}_\Sigma(c) \), if \( (c, \sigma_1 \cdots \sigma_n \sigma) \in R \) then \( (p, \sigma \text{Bool}) \in R \);

3. there is no constructor \( c \in \Sigma^C \) such that \( (c, \tilde{\sigma}_1 \sigma), (c, \tilde{\sigma}_2 \sigma) \in R \) for distinct \( \tilde{\sigma}_1 \) and \( \tilde{\sigma}_2 \).\(^{3}\)

A sort in \( \Sigma \) is an \textit{algebraic data type} if it is assigned a non-empty set of constructors. \( \square \)

Remark 8. The restrictions imposed on theory declarations and on the various commands for declaring new symbols in SMT-LIB scripts make sure that the signature defined by an SMT-LIB script is in fact a signature in the sense of Definition 2.

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

We also consider signatures that differ from a given signature \( \Sigma \) only the sort they assign to variables. We will also consider signatures that conservatively expand a given signature \( \Sigma \) with additional sort and function symbols or additional ranks for \( \Sigma \)'s function symbols.

Definition 3 (Signature variants and expansions). A signature \( \Sigma' \) is a \textit{variant} of a signature \( \Sigma \) if it is identical to \( \Sigma \) possibly except for its mapping from variables to sorts.

A signature \( \Omega \) is a \textit{expansion} of a signature \( \Sigma \) if all of the following hold: \( \Sigma^S \subseteq \Omega^S \); \( \Sigma^F \subseteq \Omega^F \); the sort symbols of \( \Sigma \) have the same arity in \( \Sigma \) and in \( \Omega \); every sort of \( \Sigma \) has the same constructors in \( \Omega \) that it has in \( \Sigma \); every constructor of \( \Sigma \) has the same selectors and testers in \( \Omega \) that it has in \( \Sigma \); 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 \( \tilde{\sigma} \in \text{Sort}(\Sigma)^+ \), if \( f : \tilde{\sigma} \in \Sigma \) then \( f : \tilde{\sigma} \in \Omega \). In that case, \( \Sigma \) is a \textit{subsignature} of \( \Omega \). \( \square \)

Overloading

The rank of a function symbol in a signature specifies, in order, the expected sort of the symbol’s arguments and result. Note that it is possible for a function symbol to be \textit{overloaded} in

\(^2\)Note that \( \text{Sort}(\Sigma^S) \), the set of all sorts over \( \Sigma^S \), is non-empty because at least one sort in \( \Sigma^S \), \text{Bool}, has arity 0.

\(^3\)A binary relation \( R \subseteq X \times Y \) is \textit{left-total} if for each \( x \in X \) there is (at least) a \( y \in Y \) such that \( x R y \).
5.2. THE LANGUAGE OF TERMS

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

Together with the mechanisms used to declare theories (described in Section 5.4) and algebraic datatypes (described in Section 4.2.3), 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 $(\text{Array} \sigma_1 \sigma_2) \sigma_1 \sigma_2$ for all sorts $\sigma_1, \sigma_2$ in a theory signature. Strictly speaking, this is still ad-hoc polymorphism because SMT-LIB logic itself does not allow parametric sorts. 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 $\bar{\sigma} \sigma_2$ where $\bar{\sigma}$ is a sequence of sorts and $\sigma_1$ and $\sigma_2$ are different sorts. Thanks to the requirement in Definition 2 that variables have exactly one sort in a signature, in signatures with no ambiguous function symbols every term can have at most one sort. In contrast, with an ambiguous symbol $f$ whose different ranks are $\bar{\sigma} \sigma_1, \ldots, \bar{\sigma} \sigma_n$, a term of the form $f \bar{t}$, where the terms $\bar{t}$ have sorts $\bar{\sigma}$, can be given a unique sort only if $f$ is annotated with one of the result sorts $\sigma_1, \ldots, \sigma_n$, that is, only if it is written as $f^{\sigma_i} \bar{t}$ for some $i \in \{i, \ldots, n\}$.

5.2.2 Well-sorted terms

Figure 5.3 provides a set of rules defining well-sorted terms with respect to an SMT-LIB signature $\Sigma$. Strictly speaking then, and similarly to more conventional logics, the SMT-LIB logic language is a family of languages parametrized by the signature $\Sigma$. 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 5.3 is fairly standard and should be largely self-explanatory to readers familiar with type systems. In more detail, the letter $\sigma$ (possibly primed or with subscripts) denotes sorts in $\text{Sort}(\Sigma)$, the letter $k$ denotes a natural number. Expressions of the form $\Sigma[x_1 : \sigma_1, \ldots, x_n : \sigma_n]$ denotes the signature that maps $x_i$ to sort $\sigma_i$ for $i = 1, \ldots, n$, and coincides otherwise with $\Sigma$. Finally, $\alpha^*$ denotes a possibly empty sequence of attributes. The rules operate over sorting judgments which are triples of the form $\Sigma \vdash t : \sigma$.

Definition 4 (Well-sorted Terms). For every SMT-LIB signature $\Sigma$, a term $t$ generated by the grammar in Figure 5.2 is well-sorted (with respect to $\Sigma$) if $\Sigma \vdash t : \sigma$ is derivable by the sorting rules in Figure 5.3 for some sort $\sigma \in \text{Sort}(\Sigma)$. In that case, we say that $t$ has, or is of, sort $\sigma$.

With this definition, it is possible to show that every term has at most one sort.\(^{[58]}\)

\(^{[58]}\)Parametric sort terms that occur in theory declarations and algebraic datatype declarations are meta-level syntax as far as SMT-LIB logic is concerned. They are schemas standing for concrete sorts.
Figure 5.3: Well-sortedness rules for terms.
Remark 9 (Match rules). The two rules for the match binder in Figure 5.3 require that the match cases be exhaustive: every constructor term of sort $\delta$ must match one of the patterns; but allow it to be redundant: the same term may match more than one pattern. Exhaustiveness is necessary to make sure each match expression is semantically well defined. The first rule deals with match expressions where no patterns consist of a variable. In that case, exhaustiveness is enforced by requiring that each constructor of the datatype appear in one of the patterns. The second rule deals with match expressions where one or more patterns consist of a variable. In that case, exhaustiveness is guaranteed simply by the presence of those variables. In both cases, the preconditions ensure that $\delta$ is a datatype, not just any sort, by requiring it to have a non-empty set of constructors.

Definition 5 (SMT-LIB formulas). For each signature $\Sigma$, the language of SMT-LIB logic is the set of all well-sorted terms wrt $\Sigma$. Formulas are well-sorted terms of sort $\text{Bool}$. In the following, we will use $\varphi$ and $\psi$ to denote formulas.

Constraint 3. SMT-LIB scripts consider only closed formulas, or sentences, closed terms of sort $\text{Bool}$. There is no loss of generality in the restriction above because, as far as satisfiability is concerned, every formula $\varphi$ with free variables $x_1, \ldots, x_n$ of respective sort $\sigma_1, \ldots, \sigma_n$, can be rewritten as

$$\exists x_1: \sigma_1 \ldots x_n: \sigma_n \varphi.$$ An alternative way to avoid free variables in scripts is to replace them by fresh constant symbols of the same sort. This is again with no loss of generality because, for satisfiability modulo theories purposes, a formula’s free variables can be treated equivalently as free symbols (see later for a definition).

5.3 Structures and Satisfiability

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

Definition 6 ($\Sigma$-structure). Let $\Sigma$ be a signature. A $\Sigma$-structure $A$ is a pair consisting of a set $A$, the universe of $A$, and a mapping that interprets

- each $\sigma \in \text{Sort}(\Sigma)$ as a non-empty subset $\sigma^A$ of $A$, which we call the domain of $\sigma$ in $A$, with $A = \bigcup_{\sigma \in \Sigma} \sigma^A; \tag{\text{60}}$

- each (ranked function symbol) $f: \sigma \in \Sigma$ as an element $(f: \sigma)^A$ of $\sigma^A$;

- each $f: \sigma_1 \cdots \sigma_n \in \Sigma$ with $n > 0$ as a total function $(f: \sigma_1 \cdots \sigma_n)^A$ from $\sigma_1^A \times \cdots \times \sigma_n^A$ to $\sigma^A$.

If $B$ is an $\Omega$-structure with universe $B$ and $\Sigma$ is a subsignature of $\Omega$, the reduct of $B$ to $\Sigma$ is the (unique) $\Sigma$-structure with universe $B$ that interprets its sort and function symbols exactly as $B$. A structure $B$ is an expansion of a $\Sigma$-structure $A$ if $A$ is the $\Sigma$-reduct of $B$. \hfill $\square$
Note that, as a consequence of overloading, a $\Sigma$-structure does not interpret plain function symbols but ranked function symbols. Also note that any $\Sigma$-structure is also a $\Sigma'$-structure for every variant $\Sigma'$ of $\Sigma$.

**Definition 7 (Absolutely free structure).** Let $A$ be a $\Sigma$-structure with universe $A$ and let $G \subseteq A$. Let $\Sigma_G$ be the expansion of $\Sigma$ obtained by adding to $\Sigma$ a constant symbol $a$ of rank $\sigma$ for every $a \in G$ and sort $\sigma \in \Sigma^S$ such that $a \in \sigma^A$. Then, $A$ is an **absolutely free structure (with generators $G$)** if

- for all $\sigma \in \Sigma^S$, $\sigma^A$ is the set of well-sorted ground terms of signature $\Sigma_G$;
- $A$ interprets every function symbol $f: \sigma_1 \cdots \sigma_n \sigma \in \Sigma_G$ as the function that maps each tuple $(t_1, \ldots, t_n) \in \sigma_1^A \times \cdots \times \sigma_n^A$ to the term $f(t_1, \ldots, t_n)$.

Intuitively, an absolutely free $\Sigma$-structure with a set $G$ of generators interprets every well-sorted ground $\Sigma_G$-term as itself. Note that the choice of generators affects the property of being absolutely free. For instance, no structure without constant symbols can be absolutely free with an empty set of generators.

The SMT-LIB logic considers only structures that interpret in a special way the sort $\text{Bool}$ and any constructor, selector, and tester symbols in their signature.

**Definition 8 (SMT-LIB $\Sigma$-structure).** An SMT-LIB structure is a $\Sigma$-structure $A$ such that

1. $\text{Bool}^A = B = \{\text{false}, \text{true}\}$ with $\text{false}$ and $\text{true}$ distinct;
2. if $\Omega$ is the signature obtained from $\Sigma$ by removing all of its non-constructor function symbols, the $\Omega$-reduct of $A$ is an absolutely free algebra with generators $\bigcup_{\sigma \in \Sigma} \sigma^A$ where $\Sigma$ collects the sorts of $\Sigma$ that are not datatypes;
3. for all constructors $c: \sigma_1 \cdots \sigma_n \sigma \in \Sigma$ with $n > 0$, selectors $g_1: \sigma_1, \ldots, g_n: \sigma_n$ with $\text{sel}_\Sigma(c) = g_1 \cdots g_n$, values $(v_1, \ldots, v_n) \in \sigma_1^A \times \cdots \times \sigma_n^A$, and $i = 1, \ldots, n$,

$$g_i^A(c^A(v_1, \ldots, v_n)) = v_i;$$
4. for all constructors $c: \sigma \in \Sigma$, testers $q$ with $\text{tes}_\Sigma(c) = q$, and values $v \in \sigma^A$,

$$q^A(v) = \text{true} \text{ iff } v \text{ is in the range of } c^A.$$  

From now on, when we say “structure” we will mean “SMT-LIB structure.”

**Remark 10.** The restrictions in SMT-LIB structures on the interpretation of constructors, selectors and testers, together with the well-foundedness restrictions on those constructs in signatures, guarantee that sorts with constructors indeed denote algebraic datatypes as traditionally understood in the literature.
Remark 11 (Partiality of selectors). As in classical first-order logic, all function symbols in a signature Σ are interpreted as total functions in a Σ-structure A. This means in particular that if \( g:σ_i \in Σ \) is a selector, the function \( g^A \) returns a value even for inputs outside the range of g’s constructor. Definition 8 imposes no constraints on that value, other than it belongs to \( σ_i^A \). For instance, in a structure A with a sort for integer lists with constructors nil and insert and selectors head and tail for insert, the function head^A maps nil^A to some integer value. Similarly, tail^A maps nil^A to some integer list. This is consistent with the general modeling of partial functions in SMT-LIB as underspecified total functions—which requires a solver to consider all possible (well-sorted) ways to make a partial function total.

The notion of isomorphism between structures introduced below is needed for Definition 12, Theory Combination, in Section 5.4.

Definition 9 (Isomorphism). Let A and B be two Σ-structures with respective universes A and B. A mapping \( h: A \rightarrow B \) is a homomorphism from A to B if

1. for all \( σ \in Sort(Σ) \) and \( a \in σ^A \),
   \[ h(a) \in σ^B \; ; \]

2. for all \( f:σ_1 \ldots σ_n \in Σ \) with \( n > 0 \) and \( (a_1, \ldots, a_n) \in σ_1^A \times \cdots \times σ_n^A \),
   \[ h((f:σ_1 \ldots σ_n)(a_1, \ldots, a_n)) = (f:σ_1 \ldots σ_n)(h(a_1), \ldots, h(a_n)) \; . \]

A homomorphism between A and B is an isomorphism of A onto B if it is invertible and its inverse is a homomorphism from B to A.

Two Σ-structures A and B are isomorphic if there is an isomorphism from one onto the other. Isomorphic structures are interchangeable for satisfiability purposes because one satisfies a set of Σ-sentences if and only if the other one does.

5.3.1 The meaning of terms

A valuation into a Σ-structure A is a partial mapping \( v \) from \( X × Sort(Σ) \) to the set of all domain elements of A such that, for all \( x \in X \) and \( σ \in Sort(Σ) \), \( v(x:σ) \in σ^A \). We denote by \( v[x_1:σ_1 \mapsto a_1, \ldots, x_n:σ_n \mapsto a_n] \) the valuation that maps \( x_i:σ_i \mapsto a_i \) for \( i = 1, \ldots, n \) and is otherwise identical to \( v \).

If \( v \) is a valuation into Σ-structure A, the pair \( I = (A, v) \) is a Σ-interpretation. We write \( I[x_1:σ_1 \mapsto a_1, \ldots, x_n:σ_n \mapsto a_n] \) as an abbreviation for the Σ'-interpretation

\[ (A', v[x_1:σ_1 \mapsto a_1, \ldots, x_n:σ_n \mapsto a_n]) \]

where \( Σ' = \{x_1:σ_1, \ldots, x_n:σ_n\} \) and A' is just A but seen as a Σ'-structure.

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

Definition 10. Let Σ be an SMT-LIB signature and let \( I \) be a Σ-interpretation. For every well-sorted term \( t \) of sort \( σ \) with respect to Σ, \([t]^I\) is defined recursively as follows.
5.3. STRUCTURES AND SATISFIABILITY

1. \([u \ a_1 \cdots a_n]^\mathcal{I} = [u]^\mathcal{I}\)
2. \([x]^\mathcal{I} = v(x;\sigma) \text{ if } \mathcal{I} = (A, v)\)
3. \([\hat{f} \ t_1 \cdots t_n]^\mathcal{I} = (f;\sigma_1 \cdots \sigma_n\sigma)^A(a_1, \ldots, a_n) \text{ if } \begin{cases} \mathcal{I} = (A, v) \text{ with signature } \Sigma, \\
\hat{f} = f \text{ or } \hat{f} = f^\sigma, \\
\Sigma \vdash t_i : \sigma_i \text{ and } a_i = [t_i]^\mathcal{I}\end{cases}\)
4. \([\text{let } x_1 = t_1 \cdots x_n = t_n \text{ in } t]^\mathcal{I} = [t]^\mathcal{I}' \text{ if } \begin{cases} \mathcal{I} \text{ has signature } \Sigma, \\
\Sigma \vdash t_i : \sigma_i \text{ and } a_i = [t_i]^\mathcal{I}, \\
\mathcal{I}' = \mathcal{I}[x_1;\sigma_1 \mapsto a_1, \ldots, x_n;\sigma_n \mapsto a_n]\end{cases}\)
5. \([\exists x_1;\sigma_1 \cdots x_n;\sigma_n \ t]^\mathcal{I} = \text{true} \iff [t]^\mathcal{I}' = \text{true} \text{ for some } \mathcal{I}'\),

\[\begin{cases} \mathcal{I}' = \mathcal{I}[x_1;\sigma_1 \mapsto a_1, \ldots, x_n;\sigma_n \mapsto a_n], \\
(a_1, \ldots, a_n) \in \sigma_1^A \times \cdots \times \sigma_n^A, \\
\mathcal{I} = (A, v)\end{cases}\]
6. \([\forall x_1;\sigma_1 \cdots x_n;\sigma_n \ t]^\mathcal{I} = \text{true} \iff [t]^\mathcal{I}' = \text{true} \text{ for all } \mathcal{I}'\),

\[\begin{cases} \mathcal{I}' = \mathcal{I}[x_1;\sigma_1 \mapsto a_1, \ldots, x_n;\sigma_n \mapsto a_n], \\
(a_1, \ldots, a_n) \in \sigma_1^A \times \cdots \times \sigma_n^A, \\
\mathcal{I} = (A, v)\end{cases}\]
7. \([\text{match } t \text{ with } c \rightarrow t_0 \ p_1 \rightarrow t_1 \cdots p_n \rightarrow t_n]^\mathcal{I} = [t_0]^\mathcal{I}\),

\[\begin{cases} \mathcal{I} = (A, v), \\
[t]^\mathcal{I} \text{ is in the range of } c^A\end{cases}\]
8. \([\text{match } t \text{ with } (c x_1 \cdots x_{k+1}) \rightarrow t_0 \ p_1 \rightarrow t_1 \cdots p_n \rightarrow t_n]^\mathcal{I} = \left[\text{let } x_1 = g_1(t) \cdots x_{k+1} = g_{k+1}(t) \text{ in } t_0\right]^\mathcal{I}\),

\[\begin{cases} \mathcal{I} = (A, v) \text{ with signature } \Sigma, \\
\Sigma \vdash c \in \text{signature } \Sigma, \\
\Sigma \vdash g_1 \cdots g_{k+1} \text{ in the range of } c^A, \\
\mathcal{I} = (A, v)\end{cases}\]
9. \([\text{match } t \text{ with } (c \bar{x}) \rightarrow t_0 \ p_1 \rightarrow t_1 \cdots p_n \rightarrow t_n]^\mathcal{I} = \left[\text{match } t \text{ with } p_1 \rightarrow t_1 \cdots p_n \rightarrow t_n (c \bar{x}) \rightarrow t_0\right]^\mathcal{I}\)

\[\begin{cases} \mathcal{I} = (A, v), \\
[t]^\mathcal{I} \text{ is not in the range of } c^A\end{cases}\]
10. \([\text{match } t \text{ with } x_0 \rightarrow t_0 \ p_1 \rightarrow t_1 \cdots p_n \rightarrow t_n]^\mathcal{I} = [\text{let } x_0 = t \text{ in } t_0]^\mathcal{I}\)
One can show that $\llbracket _ \rrbracket^T$ is well-defined, and hence total, over closed terms that are well-sorted with respect to $\mathcal{I}$’s signature.

A $\Sigma$-interpretation $\mathcal{I}$ satisfies a $\Sigma$-formula $\phi$ if $\llbracket \phi \rrbracket^\mathcal{I} = \text{true}$, and falsifies it if $\llbracket \phi \rrbracket^\mathcal{I} = \text{false}$. The formula $\phi$ is satisfiable if there is a $\Sigma$-interpretation $\mathcal{I}$ that satisfies it, and is unsatisfiable otherwise.

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

### 5.4 Theories

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

**Definition 11** (Theory). For any signature $\Sigma$, a $\Sigma$-theory is a class of $\Sigma$-structures. Each of these structures is a model of the theory.

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

### 5.4.1 Combined Theories

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

Two signatures $\Sigma_1$ and $\Sigma_2$ are compatible if they have the same sort symbols, have the same datatype constructors, selectors and testers, and they agree on both the arity they assign to sort symbols and the sorts they assign to variables. Two theories are compatible if they have compatible signatures. The combination $\Sigma_1 + \Sigma_2$ of two compatible signatures $\Sigma_1$ and $\Sigma_2$ is the smallest compatible signature that is an expansion of both $\Sigma_1$ and $\Sigma_2$, i.e., the unique signature $\Sigma$ compatible with $\Sigma_1$ and $\Sigma_2$ such that, for all $f \in F$ 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 12** (Theory Combination). Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two theories with compatible signatures $\Sigma_1$ and $\Sigma_2$, 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 $\Sigma_i$ is isomorphic to a model of $\mathcal{T}_i$, for $i = 1, 2$.

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

---

5Observe that compatibility is an equivalence relation on signatures.
5.4. THEORIES

Theories in SMT-LIB are exclusively theories of the form $T_1 + \cdots + T_n$ for some basic SMT-LIB theories $T_1, \ldots, T_n$.

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

**Definition 13** (Satisfiability and Entailment Modulo a Theory). For any $\Sigma$-theory $T$, a $\Sigma$-sentence is **satisfiable in** $T$ iff it is satisfied by one of $T$’s models. A set $\Gamma$ of $\Sigma$-sentences $T$-entails a $\Sigma$-sentence $\phi$, written $\Gamma \models_T \phi$, iff every model of $T$ that satisfies all sentences in $\Gamma$ satisfies $\phi$ as well.

### 5.4.1 Theory declarations

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

To simplify the meta-notation let $T$ denote a theory declaration with theory name $T$. Given such a theory declaration, assume first that $T$ has no sorts-description and funs-description attributes, and let $S$ and $F$ be respectively the set of all sort symbols and all function symbols occurring in $T$. Let $\Omega$ be a signature whose set of sort symbols that are not datatypes includes all the symbols in $S$, with the same arity. The definition provided in the definition attribute of $T$ must be such that every signature like $\Omega$ above uniquely determines a theory $\hat{T} = T[\Omega]$ as an instance of $T$ with signature $\hat{\Omega}$ defined as follows:

1. $\hat{\Omega}^S = \Omega^S$ and $\hat{\Omega}^F = F \cup \Omega^F$,
2. no variables are sorted in $\hat{\Omega}^{(v)}$,
3. for all $f \in \hat{\Omega}^F$ and $\bar{\sigma} \in (\hat{\Omega}^S)^+$,
(Logic attributes) \( \text{lattr} ::= \text{theories} = T^+ \mid \text{language} = w \)
\[ \mid \text{extensions} = w \mid \text{values} = w \]
\[ \mid \text{notes} = w \mid \alpha \]

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

Figure 5.5: Abstract syntax for logic declarations

\( f:\bar{\sigma} \in \hat{\Omega} \) iff

(a) \( f:\bar{\sigma} \in \Omega \), or
(b) \( T \) contains a declaration of the form \( f \bar{\sigma} \bar{\alpha} \), or
(c) \( T \) contains a declaration of the form \( \Pi \bar{u} (f \bar{\tau} \bar{\alpha}) \) and \( \bar{\sigma} \) is an instance of \( \bar{\tau} \).

4. A function symbol of \( \hat{\Omega} \) is a datatype constructor/selector/tester in \( \hat{\Omega} \) iff it is so in \( \Omega \).

We say that a ranked function symbol \( f:\bar{\sigma} \) of \( \hat{\Omega} \) is declared in \( \hat{T} \) if \( f:\bar{\sigma} \in \hat{\Omega} \) because of Point 3b or 3c above. The free sort symbols of \( \hat{T} \) are the sort symbols of \( \hat{\Omega} \) that are not in \( S \) and are not datatypes. Similarly, the free function symbols of \( \hat{T} \) are the ranked function symbols of \( \hat{\Omega} \) that are not declared in \( T \) and are not datatype constructors, selectors or testers.\(^6\) This terminology is justified by the following additional requirement on \( T \).

The definition of \( T \) must be parametric, in this sense: it must not constrain the free symbols of any instance \( T[\Omega] \) of \( T \) in any way. Technically, \( T \) must be defined so that the set of models of \( T[\Omega] \) is closed under any changes in the interpretation of the free symbols. That is, every structure obtained from a model of \( T[\Omega] \) by changing only the interpretation of \( T[\Omega] \)'s free symbols should be a model of \( T[\Omega] \) as well.\(^{62}\)

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

### 5.5 Logics

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

- fixing a signature \( \Sigma \) and a \( \Sigma \)-theory \( \mathcal{T} \),
- 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 \( \Sigma \)-sentences.

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

\(^6\)Note that because of overloading we talk about ranked function symbols being free or not, not just function symbols.
5.5. LOGICS

5.5.1 Logic declarations

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

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

**Theory.** The logic’s theory is the theory $\mathcal{T}$ uniquely determined as follows. For each $i = 1, \ldots, n$, let $S_i$ be the set of all sort symbols occurring in $T_i$. The text in the language attribute of $L$ may specify an additional set $S_0$ of sort symbols and an additional set of ranked function symbols with ranks over $\text{Sort}(S)^+$ where $S = \bigcup_{i=0}^{n} S_i$. Let $\Omega$ be the smallest signature with $\Omega^S = S$ containing all those ranked function symbols. Then for each $i = 1, \ldots, n$, let $T_i[\Omega]$ be the instance of $T_i$ determined by $\Omega$ as described in Subsection 5.4.2. The theory of $L$ is

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

Note that $\mathcal{T}$ is well defined. To start, $\Omega$ is well defined because any sort symbols shared by two declarations among $T_1, \ldots, T_n$ have the same arity in them. The theories $T_1[\Omega], \ldots, T_n[\Omega]$ are well defined because $\Omega$ satisfies the requirements in Subsection 5.4.2. Finally, the signatures of $T_1[\Omega], \ldots, T_n[\Omega]$ are pairwise compatible because they all have the same sort symbols, each with the same arity in all of them.

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

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

Note that the requirements on $V_\sigma$ can be always trivially satisfied by $L$ by making sure that the signature $\Omega$ above contains a distinguished set of infinitely many additional free constant symbols of sort $\sigma$, and defining $V_\sigma$ 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.

Recall that algebraic datatypes are not defined in theories but directly at the level of the underlying SMT-LIB logic. For each such sort, the set of values is fixed to the set of constructor terms built over values from other sorts. For example, in a parametric list datatype $\delta$ with the

---

7The concrete syntax reserves a special format for constant symbols used as abstract values: they are members of the ⟨symbol⟩ category that start with the character @.
usual nil and cons constructors, the set of values for $\delta$ consists of nil and all terms of the form
\[ \text{cons } v_1 \left( \text{cons } v_2 \cdots \left( \text{cons } v_n \text{ nil} \right) \cdots \right) \]
where $n > 0$ and $v_1, \ldots, v_n$ are (recursively) values of the same sort.
Part IV

References


Part V

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

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.

The move to the Unicode character standard was motivated by the fact that US-ASCII is is inadequate in international settings and Unicode has become the dominant standard.

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

This is to achieve maximum generality and independence from programming language conventions. This way, SMT-LIB theories of strings that use string literals as constant symbols have the choice to define certain string constants, such as "\n" and "\012", as equivalent or not. If we used, say, C-style backslash-prefixed escape sequences at the SMT-LIB level, it would be impractical and possibly confusing to represent literally certain sequences of characters. For instance, with C-style conventions the literals "\e" and "\e" would be parsed as the same two-character literal consisting of the characters 5Chex and 65hex. To represent the three-character string literal consisting of the characters 5Chex5Chex65hex, one would have to write, for instance, something like "\\e" or "\x5C\e".

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

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

Symbols have been added as indices in Version 2.6 for increased flexibility.

The sorts of pattern variables are not specified explicitly since they can be readily inferred from the datatype.

Such lists are semantically redundant in the sense that some of the cases can be dropped without affecting its meaning. This is allowed to simplify parsing.

This guarantees that the match expressions are always well defined (see Section 5.3).

The restriction to flat patterns is for simplicity and may be lifted in later versions.
This shadowing rule follows common lexical scoping conventions. It is easy to avoid declarations like \( \text{(par (Int) (f (Array Int Int) Int))} \), which are potentially confusing to human readers, simply by a more judicious choice of parameter names. In patterns like \( \text{(cons x nil)} \) are problematic.

The reason for this exception is mostly historical. Many SMT solvers apply this restriction, possibly under the assumption that shadowing a theory symbol is, more often than not, unintended and so best treated as an error.

This restriction eliminates the need for a solver to do sort inference in order to determine the sorts of terms containing applications of ambiguous function symbols. The use of \text{as} is not required in patterns because there sort inference is needed anyway since pattern variables are not explicitly assigned a sort.

The reason patterns annotate the \text{body} of a formula they refer to as opposed to the formula itself, is that, this way, they can use the body's free variables as pattern variables.

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

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

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

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

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

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

28 Setting \texttt{iglobal-declarations} to \texttt{true} can be understood as stating that declarations and definitions are not part of the assertion stack, and so resetting the stack has no impact on them. This option is convenient in applications that benefit from using push and pop for assertions but need to continue using symbols declared after a push even after the corresponding pop.

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

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

31 This is to avoid confusion with the responses to commands, which are written to the regular output channel.

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

33 This command is useful for interactive use, to keep track of the current number of nested \texttt{push} commands.

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

35 Having \texttt{ALL} is convenient for client applications that generate problems on the fly in a variety of logics supported by some specific solver without knowing in advance the specific logic for each problem. SMT-LIB scripts meant for the SMT-LIB library should not use \texttt{ALL} in set-logic but should instead use a specific logic name.

36 The rationale is that a solver may need to make substantial changes to its internal configuration to provide the functionality requested by these options, and so it needs to be notified before it starts processing assertions.

37 Strictly speaking, only sort symbols introduced with \texttt{declare-sort} expand the initial signature of theory sort symbols. Sort symbols introduced with \texttt{define-sort} do not. They do not construct \texttt{real} sorts, but \texttt{aliases} of sorts built with theory sort symbols and previously declared sort symbols.

38 The input of this command is split in two arguments precisely to facilitate the type-checking of the sort terms $\tau_1, \ldots, \tau_m$. The first argument makes sure that when it is time to parse these terms each symbol $\delta_i$ is known to be a sort symbol of arity $k_i$.

39 The various restrictions on the definition of a datatype $\delta$ are crucial since they allow the existence of standard interpretations for $\delta$. See Chapter 5 for more details.

40 The rationale for this restriction is the same as for function symbols introduced with \texttt{declare-fun} and \texttt{define-fun}, i.e., keep parsing and type checking simple.

41 Similar to \texttt{define-fun}, while strictly not needed, \texttt{define-funs-rec} provides a more structured way to define functions axiomatically, as opposed to introducing a new function with \texttt{declare-fun} and then providing its definition with \texttt{assert} and a quantified formula. This gives an SMT solver the opportunity to easily recognize function definitions and possibly process them specially.

42 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.

43 The motivation for having this command is that it corresponds to a common usage pattern with SMT solvers which can be implemented considerably more efficiently than the general stack-based mechanism needed to support \texttt{push} and \texttt{pop} commands. The restriction of assumptions to Boolean constants or negated constants only is also motivated by efficiency concerns since user-defined constants are associated with formulas that have already been internally simplified at definition time.
SMT solvers are incomplete for certain logics, typically those that include quantified formulas. However, even when they are unable to determine whether the set $\Gamma$ of all assertions and assumptions is satisfiable or not, SMT solvers can typically compute a model for a set $\Gamma'$ of formulas that is entailed by $\Gamma$ in the logic. Interpretations in this model are often useful to client applications even if they are not guaranteed to come from a model of $\Gamma$.

Abstract values are useful for reporting model values in logics containing for example the theory of arrays (see Figure 3.4). For instance, a solver may specify the content of an array $a$ of sort $\texttt{(Array Int Int)}$ at positions 0–2 by returning the expression

\[
(\text{define-fun a () (store (store (store (as @array1 (Array Int Int)) 0 0) 1 2) 2 4))}.
\]

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

Since it focuses only on preselected, Boolean terms, \texttt{get-assignment} can be implemented much more efficiently than the more general \texttt{get-value}.

The rationale for providing the interpretation of a function symbol as a define command is that (i) the syntax of such commands is general enough to be able to express a large class of functions symbolically in the language of the current logic (possibly augmented with abstract values), and (ii) in principle the user could use the provided definition as is in later interactions with the solver—as long as the original symbol's declaration is no longer in the current context as a consequence of a system restart, or a reset or a pop operation.

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

\[
(\text{define-fun-rec f ((x Int)) Int (ite (= x 0) 1 (* x (f (- x 1)))))}
\]

for a given $f$ of rank $\sigma_1 \cdots \sigma_n$. Similarly, for constant symbols $c$ of a sort $\sigma$ that admits abstract values, they allow a solver to return \texttt{(define-fun c () @a)} with @a abstract.

The reason for such weak requirements is that stronger ones are currently difficult to achieve in general because of limitations in the expressive power of some SMT-LIB theories/logics or in the computational abilities of present SMT solvers. For instance, the current theory of arrays (see Figure 3.4) does not have enough constructor symbols to allow one to represent an array uniquely as a value term. As shown in a previous note, one can use terms like

\[
(\text{store (store (store (as @array1 (Array Int Int)) 0 0) 1 2) 2 4))
\]

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

\[
(\text{define-fun-rec f ((x Int)) Int (ite (<= x 1) 1 (* x (f (- x 1)))))}
\]

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

\[
(\text{define-fun-rec f ((x Int)) Int (ite (<= x 1) 1 (* x (f (- x 1)))))}
\]

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

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

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

Unsatisfiable cores are useful for applications because they circumscribe the source of unsatisfiability in the asserted set. The labeling mechanism allows users to track only selected asserted formulas when they already know that the rest of the asserted formulas are jointly satisfiable.
Interjecting `echo` commands in a script can help a software client know where the solver is in the execution of the script's commands.

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

Note that the three possible values are strings and so need to be in quotes. The reasons for the values to be strings as opposed to symbols is historical.

Having an explicit `unknown` value is useful for comparative evaluation of solvers, for example in the SMT-COMP competition.

The reason for this unfortunate choice of names is historical. It is being kept only for backward compatibility with previous versions.

Because of this constraint, the return sort of a constructor uniquely determines the sort of its arguments. That removes the need to specify the sort of the pattern variables in a `match` expression.

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

This is mostly a technical restriction, motivated by considerations of convenience. In fact, with a closed formula \(\varphi\) of signature \(\Sigma\) 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.

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

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

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

Predefined symbols
These symbols have a predefined meaning in Version 2.6. Note that they are not reserved words. For instance, they could also be used in principle as user-defined sort or function symbols in scripts.

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

Predefined keywords
These keywords have a predefined meaning in Version 2.6.


Auxiliary Lexical Categories

\( \langle \text{white\_space\_char} \rangle \) ::= \( 9_{\text{dec}} \mid 10_{\text{dec}} \mid 13_{\text{dec}} \mid 32_{\text{dec}} \)

\( \langle \text{printable\_char} \rangle \) ::= \( 32_{\text{dec}} \mid \cdots \mid 126_{\text{dec}} \mid 128_{\text{dec}} \mid \cdots \mid 255_{\text{dec}} \)

\( \langle \text{digit} \rangle \) ::= \( 0 \mid \cdots \mid 9 \)

\( \langle \text{letter} \rangle \) ::= \( A \mid \cdots \mid Z \mid a \mid \cdots \mid z \)
Tokens

Reserved Words

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


Other tokens

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

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

S-expressions

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

Identifiers

⟨index⟩ ::= ⟨numeral⟩ | ⟨symbol⟩
⟨identifier⟩ ::= ⟨symbol⟩ | ( _ ⟨symbol⟩ ⟨index⟩ )
Sorts

\[ \langle \text{sort} \rangle ::= \langle \text{identifier} \rangle \mid (\langle \text{identifier} \rangle \langle \text{sort} \rangle^+ ) \]

Attributes

\[ \langle \text{attribute\_value} \rangle ::= \langle \text{spec\_constant} \rangle \mid \langle \text{symbol} \rangle \mid (\langle \text{s\_expr} \rangle^+ ) \]
\[ \langle \text{attribute} \rangle ::= \langle \text{keyword} \rangle \mid \langle \text{keyword} \rangle \langle \text{attribute\_value} \rangle \]

Terms

\[ \langle \text{qual\_identifier} \rangle ::= \langle \text{identifier} \rangle \mid (\text{as} \langle \text{identifier} \rangle \langle \text{sort} \rangle ) \]
\[ \langle \text{var\_binding} \rangle ::= (\langle \text{symbol} \rangle \langle \text{term} \rangle ) \]
\[ \langle \text{sorted\_var} \rangle ::= (\langle \text{symbol} \rangle \langle \text{sort} \rangle ) \]
\[ \langle \text{pattern} \rangle ::= \langle \text{symbol} \rangle \mid (\langle \text{symbol} \rangle \langle \text{symbol} \rangle^+ ) \]
\[ \langle \text{match\_case} \rangle ::= (\langle \text{pattern} \rangle \langle \text{term} \rangle ) \]
\[ \langle \text{term} \rangle ::= \langle \text{spec\_constant} \rangle \]
\[ \mid \langle \text{qual\_identifier} \rangle \]
\[ \mid (\langle \text{qual\_identifier} \rangle \langle \text{term} \rangle^+ ) \]
\[ \mid \langle \text{let} \rangle (\langle \text{var\_binding} \rangle^+ ) \langle \text{term} \rangle ) \]
\[ \mid \langle \text{forall} \rangle (\langle \text{sorted\_var} \rangle^+ ) \langle \text{term} \rangle ) \]
\[ \mid \langle \text{exists} \rangle (\langle \text{sorted\_var} \rangle^+ ) \langle \text{term} \rangle ) \]
\[ \mid \langle \text{match} \rangle \langle \text{term} \rangle (\langle \text{match\_case} \rangle^+ ) \]
\[ \mid (\langle \text{!} \rangle \langle \text{term} \rangle \langle \text{attribute} \rangle^+ ) \]
Theories

\[
\langle \text{sort} \_\text{symbol}\_\text{decl}\rangle \ := \ (\langle \text{identifier} \rangle \langle \text{numeral} \rangle \langle \text{attribute} \rangle^*)
\]

\[
\langle \text{meta}\_\text{spec}\_\text{constant}\rangle \ := \ \text{NUMERAL} \mid \text{DECIMAL} \mid \text{STRING}
\]

\[
\langle \text{fun}\_\text{symbol}\_\text{decl}\rangle \ := \ (\langle \text{spec}\_\text{constant} \rangle \langle \text{sort} \rangle \langle \text{attribute} \rangle^*)
\]

\[
\text{\mid } (\langle \text{meta}\_\text{spec}\_\text{constant} \rangle \langle \text{sort} \rangle \langle \text{attribute} \rangle^*)
\]

\[
\text{\mid } (\langle \text{identifier} \rangle \langle \text{sort} \rangle^+ \langle \text{attribute} \rangle^*)
\]

\[
\langle \text{par}\_\text{fun}\_\text{symbol}\_\text{decl}\rangle \ := \ (\langle \text{fun}\_\text{symbol}\_\text{decl} \rangle
\]

\[
\text{\mid } (\text{par} (\langle \text{symbol} \rangle^+) (\langle \text{identifier} \rangle \langle \text{sort} \rangle^+ \langle \text{attribute} \rangle^*) )
\]

\[
\langle \text{theory}\_\text{attribute}\rangle \ := \ :\text{sorts} (\langle \text{sort} \_\text{symbol}\_\text{decl}\rangle^+)
\]

\[
\text{\mid } :\text{funs} (\langle \text{par}\_\text{fun}\_\text{symbol}\_\text{decl}\rangle^+)
\]

\[
\text{\mid } :\text{sorts}\_\text{description} \langle \text{string} \rangle
\]

\[
\text{\mid } :\text{funs}\_\text{description} \langle \text{string} \rangle
\]

\[
\text{\mid } :\text{definition} \langle \text{string} \rangle
\]

\[
\text{\mid } :\text{values} \langle \text{string} \rangle
\]

\[
\text{\mid } :\text{notes} \langle \text{string} \rangle
\]

\[
\text{\mid } \langle \text{attribute} \rangle
\]

\[
\langle \text{theory}\_\text{decl}\rangle \ := \ (\langle \text{theory} \rangle \langle \text{symbol} \rangle \langle \text{theory}\_\text{attribute}\rangle^*)
\]

Logics

\[
\langle \text{logic}\_\text{attribute}\rangle \ := \ :\text{theories} (\langle \text{symbol} \rangle^*)
\]

\[
\text{\mid } :\text{language} \langle \text{string} \rangle
\]

\[
\text{\mid } :\text{extensions} \langle \text{string} \rangle
\]

\[
\text{\mid } :\text{values} \langle \text{string} \rangle
\]

\[
\text{\mid } :\text{notes} \langle \text{string} \rangle
\]

\[
\text{\mid } \langle \text{attribute} \rangle
\]

\[
\langle \text{logic}\rangle \ := \ (\langle \text{logic} \rangle \langle \text{symbol} \rangle \langle \text{logic}\_\text{attribute}\rangle^*)
\]
Info flags

\[ info\_flag \ ::= \ all\_statistics \mid assertion\_stack\_levels \mid authors \mid error\_behavior \mid name \mid reason\_unknown \mid version \mid (keyword) \]

Command options

\[ b\_value \ ::= \ true \mid false \]

\[ option \ ::= \ diagnostic\_output\_channel \ string \mid global\_declarations \ b\_value \mid interactive\_mode \ b\_value \mid print\_success \ b\_value \mid produce\_assertions \ b\_value \mid produce\_assignments \ b\_value \mid produce\_models \ b\_value \mid produce\_proofs \ b\_value \mid produce\_unsat\_assumptions \ b\_value \mid produce\_unsat\_cores \ b\_value \mid random\_seed \ numeral \mid regular\_output\_channel \ string \mid reproducible\_resource\_limit \ numeral \mid verbosity \ numeral \mid (attribute) \]
Commands

(sort_dec) ::= (⟨symbol⟩⟨numeral⟩)
(selector_dec) ::= (⟨symbol⟩⟨sort⟩)
(constructor_dec) ::= (⟨symbol⟩⟨selector_dec⟩*)
(datatype_dec) ::= (⟨constructor_dec⟩+ | (par (⟨symbol⟩+) (⟨constructor_dec⟩+ ))
(function_dec) ::= (⟨symbol⟩(⟨sorted_var⟩*)⟨sort⟩)
(function_def) ::= (⟨symbol⟩(⟨sorted_var⟩*)⟨sort⟩⟨term⟩)
(prop_literal) ::= (⟨symbol⟩ | (not ⟨symbol⟩)
(command) ::= (assert ⟨term⟩)
  | (check-sat)
  | (check-sat-assuming (⟨prop_literal⟩*)
  | (declare-const ⟨symbol⟩⟨sort⟩)
  | (declare-datatype ⟨symbol⟩⟨datatype_dec⟩)
  | (declare-datatypes (⟨sort_dec⟩n+1 )(⟨datatype_dec⟩n+1 ))
  | (declare-fun ⟨symbol⟩(⟨sort⟩*)⟨sort⟩)
  | (declare-sort ⟨symbol⟩⟨numeral⟩)
  | (define-fun ⟨function_def⟩)
  | (define-fun-rec ⟨function_def⟩)
  | (define-funs-rec (⟨function_dec⟩n+1 )(⟨term⟩n+1 ))
  | (define-sort ⟨symbol⟩(⟨symbol⟩*)⟨sort⟩)
  | (echo ⟨string⟩)
  | (exit)
  | (get-assertions)
  | (get-assignment)
  | (get-info ⟨info_flag⟩)
  | (get-model)
  | (get-option ⟨keyword⟩)
  | (get-proof)
  | (get-unsat-assumptions)
  | (get-unsat-core)
  | (get-value (⟨term⟩*)
  | (pop ⟨numeral⟩)
  | (push ⟨numeral⟩)
  | (reset)
  | (reset-assertions)
  | (set-info ⟨attribute⟩)
  | (set-logic ⟨symbol⟩)
  | (set-option ⟨option⟩)

(script) ::= (command)*
Command responses

\[
\begin{align*}
\langle \text{error-behavior} \rangle & ::= \text{immediate-exit} | \text{continued-execution} \\
\langle \text{reason-unknown} \rangle & ::= \text{memout} | \text{incomplete} | \langle s_{\text{expr}} \rangle \\
\langle \text{model-response} \rangle & ::= (\text{define-fun} \langle \text{function-def} \rangle) | (\text{define-fun-rec} \langle \text{function-def} \rangle) \\
& \quad | (\text{define-funs-rec} \langle \text{function-dec} \rangle^{n+1} \langle \text{term} \rangle^{n+1}) \\
\langle \text{info-response} \rangle & ::= \langle \text{assertion-stack-levels} \rangle \langle \text{numeral} \rangle \\
& \quad | \langle \text{authors} \rangle \langle \text{string} \rangle \\
& \quad | \langle \text{error-behavior} \rangle \langle \text{error-behavior} \rangle \\
& \quad | \langle \text{name} \rangle \langle \text{string} \rangle \\
& \quad | \langle \text{reason-unknown} \rangle \langle \text{reason-unknown} \rangle \\
& \quad | \langle \text{version} \rangle \langle \text{string} \rangle \\
& \quad | \langle \text{attribute} \rangle \\
\langle \text{valuation-pair} \rangle & ::= (\langle \text{term} \rangle \langle \text{term} \rangle) \\
\langle t_{\text{valuation-pair}} \rangle & ::= (\langle \text{symbol} \rangle \langle \text{b_value} \rangle) \\
\langle \text{check-sat-response} \rangle & ::= \text{sat} | \text{unsat} | \text{unknown} \\
\langle \text{echo-response} \rangle & ::= \langle \text{string} \rangle \\
\langle \text{get-assertions-response} \rangle & ::= (\langle \text{term} \rangle^*) \\
\langle \text{get-assignment-response} \rangle & ::= (\langle t_{\text{valuation-pair}} \rangle^*) \\
\langle \text{get-info-response} \rangle & ::= (\langle \text{info-response} \rangle^+) \\
\langle \text{get-model-response} \rangle & ::= (\langle \text{model-response} \rangle^+) \\
\langle \text{get-option-response} \rangle & ::= \langle \text{attribute-value} \rangle \\
\langle \text{get-proof-response} \rangle & ::= \langle s_{\text{expr}} \rangle \\
\langle \text{get-unsat-assump-response} \rangle & ::= (\langle \text{symbol} \rangle^*) \\
\langle \text{get-unsat-core-response} \rangle & ::= (\langle \text{symbol} \rangle^*) \\
\langle \text{get-value-response} \rangle & ::= (\langle \text{valuation-pair} \rangle^+) \\
\langle \text{specific-success-response} \rangle & ::= \langle \text{check-sat-response} \rangle | \langle \text{echo-response} \rangle \\
& \quad | \langle \text{get-assertions-response} \rangle | \langle \text{get-assignment-response} \rangle \\
& \quad | \langle \text{get-info-response} \rangle | \langle \text{get-model-response} \rangle \\
& \quad | \langle \text{get-option-response} \rangle | \langle \text{get-proof-response} \rangle \\
& \quad | \langle \text{get-unsat-assumptions-response} \rangle \\
& \quad | \langle \text{get-unsat-core-response} \rangle | \langle \text{get-value-response} \rangle \\
\langle \text{general-response} \rangle & ::= \text{success} | \langle \text{specific-success-response} \rangle \\
& \quad | \text{unsupported} | (\text{error} \langle \text{string} \rangle)
\end{align*}
\]
Abstract Syntax

Common Notation

$b \in B$, the set of boolean values
$n \in N$, the set of natural numbers
$s \in S$, the set of sort symbols
$f \in F$, the set of function symbols
$a \in A$, the set of attribute names
$T \in T\mathcal{N}$, the set of theory names

$r \in R$, the set of non-negative rational numbers
$w \in W$, the set of character strings
$u \in U$, the set of sort parameters
$x \in \mathcal{X}$, the set of variables
$v \in \mathcal{V}$, the set of attribute values
$L \in \mathcal{L}$, the set of logic names

Sorts

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

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

Terms

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

(Patterns) $p ::= x | f x^*$

(Terms) $t ::= x | f t^* | f^\sigma t^*$

$| \exists (x;\sigma)^+ t | \forall (x;\sigma)^+ t | \text{let } (x = t)^+ \text{ in } t | \text{match } t \text{ with } (p \to t)^+$
Well-sorting rules for terms

\[ \Sigma \vdash x \alpha^*: \sigma \] if \( x: \sigma \in \Sigma \)

\[ \Sigma \vdash t_1: \sigma_1 \quad \ldots \quad \Sigma \vdash t_k: \sigma_k \] if

\[ f: \sigma_1 \ldots \sigma_k \sigma \in \Sigma, \quad f: \sigma_1 \ldots \sigma_k \sigma' \notin \Sigma \quad \text{for all } \sigma' \neq \sigma \]

\[ \Sigma \vdash (f \circ t_1 \ldots t_k) \alpha^*: \sigma \] if

\[ f: \sigma_1 \ldots \sigma_k \sigma \in \Sigma, \quad f: \sigma_1 \ldots \sigma_k \sigma' \in \Sigma \quad \text{for some } \sigma' \neq \sigma \]

\[ \Sigma[x_1: \sigma_1, \ldots, x_{k+1}: \sigma_{k+1}] \vdash t: \text{Bool} \] if \( Q \in \{\exists, \forall\} \)

\[ \Sigma \vdash t_1: \sigma_1 \quad \ldots \quad \Sigma \vdash t_{k+1}: \sigma_{k+1} \]

\[ \Sigma[x_1: \sigma_1, \ldots, x_{k+1}: \sigma_{k+1}] \vdash t: \sigma \] if \( x_1, \ldots, x_{k+1} \) are all distinct

\[ \Sigma \vdash (\text{let } x_1 = t_1 \ldots x_{k+1} = t_{k+1} \text{ in } t) \alpha^*: \sigma \]

\[ \Sigma \vdash t: \delta \]

\[ \Sigma[x_i: \delta] \vdash t_i: \sigma \quad \text{for } i = 1, \ldots, k+1 \]

\[ \Sigma \vdash (\text{match } t \text{ with } c_1 x_1 \to t_1 \ldots c_{k+1} x_{k+1} \to t_{k+1}) \alpha^*: \sigma \]

\[ \Sigma \vdash t: \delta \]

\[ \Sigma[x_i: \delta] \vdash t_i: \sigma \quad \text{for } i = k+1, \ldots, n \]

\[ \Sigma \vdash (\text{match } t \text{ with } p_1 \to t_1 \ldots p_n \to t_n) \alpha^*: \sigma \]

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) \( tattr ::= \begin{array}{l} \text{sorts} = sdec^+ \mid \text{funs} = pdec^+ \\ \text{sorts-description} = w \\ \text{funs-description} = w \\ \text{definition} = w \mid \text{values} = t^+ \\ \text{notes} = w \mid \alpha \end{array} \)

(Theory declarations) \( tdec ::= \text{theory } T \ tattr^+ \)
Logics

(Logic attributes) \( lattr \) ::= the\( \text{ories} \) = \( T^+ \) | language = \( w \)
| extensions = \( w \) | values = \( w \)
| notes = \( w \) | \( \alpha \)

(Logic declarations) \( ldec \) ::= logic \( L \) \( lattr^+ \)