The SMT-LIB Standard
Version 2.7
(Draft)

Clark Barrett   Pascal Fontaine   Cesare Tinelli

Release: DRAFT

Copyright © 2015-24 Clark Barrett, Pascal Fontaine, and Cesare Tinelli.

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

Preface

The SMT-LIB initiative is an international effort, supported by several research groups world-wide, 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 2.7 of the *SMT-LIB Standard*, a backward-compatible extension of Version 2.6

Acknowledgments

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 2.6 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.

Version 2.7

The main difference between 2.7 and Version 2.6 is the inclusion of two extensions to the SMT-LIB base logic: (*i*) support for prenex polymorphism in user-defined sorts and functions, and (*ii*) an extension to higher-order logic obtained, in essence, by the introduction of a theory of functions. Aside from new functionality, which assigns special meaning to the symbols `_`, `lambda`, and `->`, Version 2.7 is fully backward-compatible with Version 2.6 and is designed to ease the transition to the upcoming Version 3, whose base logic will be a higher-order logic with dependent (and polymorphic) types. Special thanks go to Nikolaj Bjørner and Andrew Reynolds for proposing this transitional version and for their feedback and useful suggestions. We also thank Guillaume Bury and Levent Erkök for their feedback on the reference document and the standard.

Pascal Fontaine is grateful to Jasmin Blanchette for his support through his European Research Council (ERC) starting grant Matryoshka (713999). He also thanks Amazon for its support through the Amazon Research Award program (SMT: Modules, Formats, and Standards).

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 2.7 . . . 10
  - 1.1.2 Differences between Version 2.7 and Version 2.6 . . . 11
  - 1.1.3 Change log for Version 2.6 . . . 11
  - 1.1.4 Differences between Version 2.6 and Version 2.5 . . . 12
  - 1.1.5 Change log for Version 2.5 . . . 12
  - 1.1.6 Differences between Version 2.5 and Version 2.0 . . . 12
  - 1.1.7 Typographical and notational conventions . . . 14
- 1.2 Overview of SMT-LIB . . . 14
  - 1.2.1 What is SMT-LIB? . . . 15
  - 1.2.2 Main features of the SMT-LIB standard . . . 15

2 Basic Assumptions and Structure 17
- 2.1 Satisfiability Modulo Theories . . . 17
- 2.2 Underlying Logic . . . 18
- 2.3 Background Theories . . . 18

2.4 Input Formulas 19
2.5 Interface 20

II Syntax 21

3 The SMT-LIB Language 22
3.1 Lexicon 22
3.2 S-expressions 25
3.3 Identifiers 26
3.4 Attributes 26
3.5 Sorts 27
3.5.1 Ranks 27
3.6 Terms and Formulas 28
3.6.1 Variable Binders 29
3.6.2 Scoping of variables 32
3.6.3 Well-sortedness requirements 33
3.6.4 Annotations 33
3.6.5 Term attributes 33
3.7 Theory Declarations 34
3.7.1 Examples 38
3.8 Core Theory 40
3.9 Higher-order Core Theory 43
3.10 Logic Declarations 44
3.10.1 Examples 46
3.11 Scripts 46
3.11.1 Command responses 49
3.11.2 Example scripts 51
3.11.3 SMT-LIB Benchmarks 52

III Semantics 53

4 Operational Semantics of SMT-LIB 54
4.1 General Requirements 54
4.1.1 Execution Modes 55
4.1.2 Solver responses 56
4.1.3 Printing of terms and defined symbols 57
4.1.4 The assertion stack 57
4.1.5 Symbol declarations and definitions 58
4.1.6 In-line definitions 58
4.1.7 Solver options 58
4.1.8 Solver information 60
4.2 Commands 61
4.2.1 (Re)starting and terminating 62

4.2.2 Modifying the assertion stack . . . 62
4.2.3 Introducing new symbols . . . 63
4.2.4 Asserting and inspecting formulas . . . 67
4.2.5 Checking for satisfiability . . . 68
4.2.6 Inspecting models . . . 68
4.2.7 Inspecting proofs . . . 71
4.2.8 Inspecting settings . . . 71
4.2.9 Script information . . . 72

5 Logical Semantics of SMT-LIB Formulas **74**
5.1 The language of sorts . . . 75
5.2 The language of terms . . . 76
5.2.1 Signatures . . . 77
5.2.2 Well-sorted terms . . . 79
5.3 Structures and Satisfiability . . . 81
5.3.1 The meaning of terms . . . 84
5.4 Theories . . . 85
5.4.1 Combined Theories . . . 86
5.4.2 Theory declarations . . . 86
5.5 Logics . . . 87
5.5.1 Logic declarations . . . 88

IV References **90**

Bibliography **91**

V Appendices **93**

A Notes **94**

B Concrete Syntax **99**

C Abstract Syntax **106**

Index **109**

List of Figures

3.1 Theory declarations. 34
3.2 A possible theory declaration for the integer numbers. 39
3.3 The `ArraysEx` theory declaration. 40
3.4 A possible declaration for a theory of sets and relations. 41
3.5 The `Core` theory declaration. 42
3.6 The `HO-Core` theory declaration. 43
3.7 SMT-LIB Commands. 47
3.8 Command options. 48
3.9 Info flags. 48
3.10 Command responses. 50
3.11 Example script (over two columns), with expected solver responses in comments. 51
3.12 Another example script (excerpt), with expected solver responses in comments. 52

4.1 Abstract view of transitions between solver execution modes. The symbol * here stands for the matching wildcard. 55

5.1 Abstract syntax for sort terms 75
5.2 Abstract syntax for unsorted terms 76
5.3 Well-sortedness rules for terms. 80
5.4 Abstract syntax for theory declarations 87
5.5 Abstract syntax for logic declarations 88

Part I

Introduction

CHAPTER 1

General Information

1.1 About This Document

This document is mostly self-contained, though it assumes some familiarity with first-order logic, *aka* predicate calculus, and higher-order logic, *aka* simple type theory. The reader is referred to any of several textbooks on first-order logic [Gal86, Fit96, End01, Men09] and monographs on higher-order logic [And86, NS24]. Previous knowledge of Version 1.2 of the SMT-LIB standard [RT06] is not necessary.[1]

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 2.7

Release: DRAFT

- Draft release of Version 2.7.

[1] Version 2 and its subversions, while largely based on Version 1.2, are *not* backward compatible with it. See the Version 2.0 document [BST10b] for a summary of the major differences.

1.1.2 Differences between Version 2.7 and Version 2.6

The command language is extended with a new command for declaring (global) sort parameters, which can be used to assert formulas with terms of polymorphic sort in scripts. Semantically, the sort parameters are treated as implicitly universally quantified sort variables in each asserted formula. This has the effect of allowing prenex (or rank-1) polymorphism in assertions but not more general forms of polymorphism.

The term language is extended with a λ binder to construct function abstractions and a symbol for function application, `_`, intentionally equal to that for the construction of indexed symbols in Version 2.6. The semantics of the λ binder and the application symbol are provided in a new theory, of higher-order functions, which includes the binary sort constructor `->` for function space sorts. In this version, values of sort (-> τ_1 τ_2) are treated as any other value, and so can be passed to and returned from functions. However, they are not identified with functions of rank $\tau_1 \tau_2$ (such as those introduced by `declare-fun`), keeping the underlying logic of Version 2.7 first order.[2] Consistent with this extension, the command language is extended with the constant definition command `define-const`, which can be used in particular to define constants of sort (-> τ_1 τ_2).

The symbol `_` can now also be used (unambiguously) as a wildcard symbol in `match` patterns.

The functionality of `check-sat-assuming` has been extended to allow arbitrary Boolean terms to be passed as assumptions. The semantics of `get-unsat-assumptions` and `get-unsat-core` have been correspondingly generalized and clarified.

The format of answers to the `get-model` command has been further restricted to prevent forward references in models.

1.1.3 Change log for Version 2.6

Release: 2024-09-20

- Changed verbosity to have no standard default value
- Changed default for print-success to be false

Release: 2021-05-12

- Fixed misleading example of usage of `as` to disambiguate symbol sorts.

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.

[2] This identification will occur in Version 3, which will be based on higher-order logic.

1.1.4 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 3.11.3 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.

1.1.5 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.

1.1.6 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 2.5. 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.
- 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`.

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

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

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

1.2 Overview of SMT-LIB

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

While SMT techniques have been traditionally used to support deductive software verification, they have found applications in other areas of computer science such as, for in-

stance, 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 400,000 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 2.7 defines:

- a language for writing *terms and formulas* in a sorted (i.e., typed) version of first-order logic;
- a language for specifying *background theories* and fixing a standard vocabulary of sort, function, and predicate symbols for them;
- a language for specifying *logics*, suitably restricted classes of formulas to be checked for satisfiability with respect to a specific background theory;

- a *command* language for interacting with SMT solvers via a textual interface that allows asserting and retracting formulas, querying about their satisfiability, examining their models or their unsatisfiability proofs, and so on.

CHAPTER 2

Basic Assumptions and Structure

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

2.1 Satisfiability Modulo Theories

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

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

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

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

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

The most recent past version of the SMT-LIB format (Version 2.6) adopts as its underlying logic a version of many-sorted first-order logic with equality [Man93, Gal86, End01]. Like traditional many-sorted logic, it has sorts (i.e., basic types) and sorted terms. Unlike that logic, however, it does not have a syntactic category of formulas distinct from terms. Formulas are just sorted terms of a distinguished Boolean sort, which is interpreted as a two-element set in every SMT-LIB theory.[1] Furthermore, the SMT-LIB logic uses a language of sort terms, as opposed to just sort constants, to denote sorts: sorts can be denoted by sort constants like `Int` as well as sort terms like `(List (Array Int Real))`. Finally, in addition to the usual existential and universal quantifiers, the logic includes a *let* binder and a *match* binder analogous to constructs with the same name found in functional programming languages.

Version 2.7 extends the logic of Version 2.6 with a λ binder for function abstraction, creating terms with a sort the form `(->` τ_1 τ_2`)`, and an explicit application operator for values of such sort. This allows, in effect, the use of higher-order functions, while keeping the underlying logic nominally first order.

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

2.3 Background Theories

One of the goals of the SMT-LIB initiative is to clearly define a catalog of background theories, starting with a small number of popular ones, and adding new ones as solvers for them are developed.[2] Theories are specified in SMT-LIB independently of any benchmarks or solvers. On the other hand, each SMT-LIB script refers, indirectly, to one or more theories in the SMT-LIB catalog.

This version of the SMT-LIB standard distinguishes between *basic* theories and *combined* theories. Basic theories, such as the theory of real numbers, the theory of arrays, the theory

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

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

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 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.[3] By "formally" here we mean written in a machine-readable and processable format, as opposed to written in free text, no matter how rigorously. By this definition, theories themselves are defined informally, in natural language. Some theories, such as the theory of bit vectors, have an infinite signature. For them, the signature too is specified informally in English.[(1)]

2.4 Input Formulas

SMT-LIB adopts a single and general (sorted) language for writing 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.[4]

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

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

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

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

2.5 Interface

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

Part II

Syntax

CHAPTER 3

The SMT-LIB Language

This chapter defines and explains the concrete syntax of the SMT-LIB standard, what we comprehensively refer to as *the 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_{dec} to 126_{dec}. The remaining printable characters, mostly used for non-English alphabet letters (characters 128_{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.[1]

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

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

Printable Characters. A ⟨*printable_char*⟩ is any character from 32_{dec} to 126_{dec} (US-ASCII) and from 128_{dec} on.[2]

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

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

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

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

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

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

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

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

String literals. A ⟨*string*⟩ (literal) is any sequence of characters from ⟨*printable_char*⟩ or ⟨*white_space_char*⟩ delimited by the double quote character `"` (34_{dec}). The character `"` can itself occur *within* a string 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.

[1] Which implies that the language's semantics does not depend on indentation and spacing.
[2] Note that the space character is both a printable and a whitespace character.

```
"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   lambda   let   exists   forall   match   par
```

Additionally, each command name in the scripting language defined in Section 3.11 (`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

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

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

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

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

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

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

```
|this is a quoted symbol|

|so is
 this one|

||

| " can occur too|

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

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

Simple symbols starting with the character `@` or `.` are reserved for solver use.[4] Solvers can use them respectively as identifiers for abstract values and solver-generated function symbols other than abstract values.

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

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

3.2 S-expressions

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

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

[4] This includes symbols such as `|@abc|` and `|.abc|` which are considered the same as `@abc` and `.abc`, respectively.

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

3.3 Identifiers

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

⟨*index*⟩ ::= ⟨*numeral*⟩ | ⟨*symbol*⟩

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

```
plus    +       <=    Real    |John Brown|
(_ vector-add 4 5)  (_ BitVec 32)
(_ move up)  (_ move down)  (_ move left)  (_ move right)
```

We refer to identifiers from ⟨*symbol*⟩ as *simple* identifiers and to the others as *indexed* identifiers. Since identifiers are used as the names of function symbols, sort symbols, sort parameters, (term) variables and commands, we often refer to them informally as *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 user sort symbols, as described in Section 4.2.3.

3.4 Attributes

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

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

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

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

3.5 Sorts

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

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

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

```
Int                         Bool

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

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

(-> Int Real)               (-> Int (-> Int Real))
```

A *monomorphic* sort is a sort containing no parameters. A *polymorphic* sort is a sort containing zero or more parameters. For instance, if `X` and `Y` are sort parameters, and `Int` and `Array` are sort symbols, then `(Array Int Int)` is a monomorphic sort while `(Array Int Y)`, `(Array X Int)`, and `(Array X Y)` are all polymorphic sorts. Note that we treat monomorphic sorts as a special case of polymorphic ones; hence, `(Array Int Int)` is also a (trivially) polymorphic sort. Contextual information is needed in general to know whether a particular symbol occurring in a sort is a sort parameter or not. Local sort parameters are introduced by a specific binder, `par`. Global sort parameters are declared in a script by the command `declare-sort-parameter` (see Section 4.2.3).

We will write just *sort* to mean a polymorphic sort, and write instead *monomorphic sort* when we want to emphasize that the sort has no parameters. We will use σ to denote monomorphic sorts and τ to denote, more generally, polymorphic 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 monomorphic

sorts), denotes a function that takes as input n values of respective sorts $\sigma_1, \ldots, \sigma_n$, and returns a value of sort σ.

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 monomorphic sort.

3.6 Terms and Formulas

Abstractly, terms are constructed out of constant symbols in the ⟨*spec_constant*⟩ category (numerals, decimals, strings, etc.), *variables*, *function symbols*, four kinds of *binders* (introduced by the reserved words `lambda`, `let`, `forall`, `exists`, and `match`), and an annotation operator: the reserved word `!`. 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 ⟨*symbol*⟩, while a function symbol can be any ⟨*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 binders `let`, `lambda`, `forall`, `exists`, and `match`, 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 τ. Such an annotated function symbol is a *qualified identifier* of the form `(as` f τ`)`.

```
⟨qual_identifier⟩  ::=  ⟨identifier⟩ | ( as ⟨identifier⟩ ⟨sort⟩ )
⟨var_binding⟩      ::=  ( ⟨symbol⟩ ⟨term⟩ )
⟨sorted_var⟩       ::=  ( ⟨symbol⟩ ⟨sort⟩ )
⟨pattern⟩          ::=  ⟨symbol⟩ | ( ⟨symbol⟩ ⟨symbol⟩+ )
⟨match_case⟩       ::=  ( ⟨pattern⟩ ⟨term⟩ )
⟨term⟩             ::=  ⟨spec_constant⟩
                     |  ⟨qual_identifier⟩
                     |  ( ⟨qual_identifier⟩ ⟨term⟩+ )
                     |  ( let ( ⟨var_binding⟩+ ) ⟨term⟩ )
                     |  ( lambda ( ⟨sorted_var⟩+ ) ⟨term⟩ )
                     |  ( forall ( ⟨sorted_var⟩+ ) ⟨term⟩ )
                     |  ( exists ( ⟨sorted_var⟩+ ) ⟨term⟩ )
                     |  ( match ⟨term⟩ ( ⟨match_case⟩+ ) )
                     |  ( ! ⟨term⟩ ⟨attribute⟩+ )
```

Terms are members of the syntactic category ⟨*term*⟩. Note that sort terms occur within terms containing binders or qualified identifiers. These sort terms may contain sort param-

eters, which must be declared in a script by the command `declare-sort-parameter` (see Section 4.2.3).

SMT-LIB scripts can contain only *well-sorted* terms (see Section 3.6.3). Formulas in SMT-LIB are just well-sorted terms of sort `Bool` (see Section 3.8). As a consequence, there is no syntactic distinction between function and predicate symbols; the latter are simply function symbols whose result sort is `Bool`. Another consequence is that function symbols can take formulas (even quantified ones) as arguments.

```
(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))))))
```

3.6.1 Variable Binders

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

Lambda. This binder corresponds to the abstraction binder of higher-order logic. It takes a non-empty list of variables, which abbreviates a sequential nesting of lambda abstractions. Specifically, a term of the form

$$\texttt{(lambda ((}x_1\ \tau_1\texttt{) (}x_2\ \tau_2\texttt{)}\ \cdots\ \texttt{(}x_n\ \tau_n\texttt{))}\ t\texttt{)} \tag{3.1}$$

has the same semantics as the term

$$\texttt{(lambda ((}x_1\ \tau_1\texttt{)) (lambda ((}x_2\ \tau_2\texttt{)) (}\cdots\ \texttt{(lambda ((}x_n\ \tau_n\texttt{))}\ t\texttt{)}\ \cdots\texttt{)))} \tag{3.2}$$

See Section 3.8 for more details on the use of this binder.

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

$$\texttt{(forall ((}x_1\ \tau_1\texttt{) (}x_2\ \tau_2\texttt{)}\ \cdots\ \texttt{(}x_n\ \tau_n\texttt{))}\ \varphi\texttt{)} \tag{3.3}$$

has the same semantics as the formula

$$\texttt{(forall ((}x_1\ \tau_1\texttt{)) (forall ((}x_2\ \tau_2\texttt{)) (}\cdots\ \texttt{(forall ((}x_n\ \tau_n\texttt{))}\ \varphi\texttt{)}\ \cdots\texttt{)} \tag{3.4}$$

Note that the variables in the list $\texttt{((}x_1\ \tau_1\texttt{) (}x_2\ \tau_2\texttt{)}\ \cdots\ \texttt{(}x_n\ \tau_n\texttt{))}$ of (3.3) 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 `let` binder introduces and defines one or more local variables *in parallel*. Semantically, a term of the form

$$\texttt{(let ((}x_1\ t_1\texttt{)}\ \cdots\ \texttt{(}x_n\ t_n\texttt{))}\ t\texttt{)} \tag{3.5}$$

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 (3.5) must be pairwise distinct*.

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

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

$$\texttt{(match}\ t\ \texttt{((}p_1\ t_1\texttt{)}\ \cdots\ \texttt{(}p_{m+1}\ t_{m+1}\texttt{)))} \tag{3.6}$$

where t is a term of some datatype sort δ and, for each $i = 1, \ldots, m+1$, p_i is a *pattern* for δ, and t_i a term of some sort τ. A pattern p in turn is either a variable x of sort δ, the special variable `_`, a nullary constructor c of δ, or a term of the form `(`$c\ x_1\ \cdots\ x_k$`)`, where c is a constructor of δ of rank $\tau_1 \cdots \tau_k \delta$ with $k > 0$, and $x_1, \ldots, x_k$ are *distinct* variables of respective sort $\tau_1, \ldots, \tau_k$.(9) An exception is the special variable `_` which can only occur in place of one or more of the variables $x_1, \ldots, x_k$. This variable acts as a wildcard, with each occurrence matching any term of any sort, hence effectively behaving as a fresh variable.

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.(10) However, it *must* contain a pattern consisting of a variable, unless every constructor of δ occurs in one of the patterns.(11) The recommended choice for a single variable pattern is `_`; the use of other variables will be deprecated in future versions.(12)

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

```
; Axiom for list append: version 1
; List is a polymorphic 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)))
        (_ l2)))))

; Axiom for list length
(forall ((l (List Int)))
  (= (length l)
     (match l (
        (nil 0)
        ((cons _ t) (length t))))))
```

Remark 4 (No nested patterns)**.** *Nested* patterns, where the arguments of a constructor can be themselves non-variable patterns, are not allowed.(13) 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. □

Remark 5 (Double use of `_`)**.** The special symbol `_` is used both to construct index terms and as a wildcard symbol in `match` patterns. These two different uses can always be disambiguated syntactically since `_` occurs in a function symbol position in the first case (e.g., in `(_ BitVec 5)`) and in an argument position in the second case (e.g., in `(cons h _)`, `(cons _ t)`, or `(cons _ _)`).

Strictly speaking, the `match` binder is not essential since it can be defined in terms of the other binders. Specifically, expression (3.6) can be written equivalently as follows.

1. If the match statement contains one or more occurrences of `_`, each of them is replaced by a fresh variable.

2. 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 (3.6) has the same meaning as

$$
\begin{array}{l}
\texttt{(ite (}q_1\ t\texttt{) (let ((}x_{1,1}\ \texttt{(}s_{1,1}\ t\texttt{))}\ \cdots\ \texttt{(}x_{1,k_1}\ \texttt{(}s_{1,k_1}\ t\texttt{)))}\ t_1\texttt{)} \\
\quad \texttt{(ite (}q_2\ t\texttt{) (let ((}x_{2,1}\ \texttt{(}s_{2,1}\ t\texttt{))}\ \cdots\ \texttt{(}x_{2,k_2}\ \texttt{(}s_{2,k_2}\ t\texttt{)))}\ t_2\texttt{)} \\
\quad\quad \cdots \\
\quad\quad\quad \texttt{(ite (}q_m\ t\texttt{) (let ((}x_{m,1}\ \texttt{(}s_{m,1}\ t\texttt{))}\ \cdots\ \texttt{(}x_{m,k_m}\ \texttt{(}s_{m,k_m}\ t\texttt{)))}\ t_m\texttt{)} \\
\quad\quad\quad\quad \texttt{(let ((}x_{m+1,1}\ \texttt{(}s_{m+1,1}\ t\texttt{))}\ \cdots\ \texttt{(}x_{m+1,k_{m+1}}\ \texttt{(}s_{m+1,k_{m+1}}\ t\texttt{)))}\ t_{m+1}\texttt{)}\ \cdots\ \texttt{)}
\end{array}
\tag{3.7}
$$

3. When for some $i \in \{1, \ldots, m + 1\}$, the pattern p_i is a nullary constructor, the corresponding `let` subexpression in (3.7) is replaced by t_i.

4. If instead for some minimal $i \in \{1, \ldots, m + 1\}$, the pattern p_i is a variable x, then the whole ith `ite` subexpression of (3.7), if $i \leq m$, or the `let` subexpression, if $i = m + 1$, is replaced by (`let` `((`x t`))` t_i`)`.

3.6.2 Scoping of variables

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

A variable x:

- occurs free in the expression x;
- occurs free in an expression `(`e_1 `···` e_n`)` if e_1 is `par` or not a binder, and x occurs free in some e_i ($1 \leq i \leq n$);
- occurs free in an expression `(`Q `((`x_1 σ_1`)` `···` `(`x_n σ_n`))` t`)`, where Q is `lambda`, `forall`, or `exists`, if it does not occur in $\{x_1, \ldots, x_n\}$ and occurs free in t;
- occurs free in an expression `(let ((`x_1 t_1`)` `···` `(`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 `(match` t `((`p_1 t_1`)` `···` `(`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.

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

- In an expression `(`Q `((`x_1 σ_1`)` `(`x_2 σ_2`)` `···` `(`x_n σ_n`))` t`)` where Q is either the binder `lambda`, `forall`, or `exists`, or in an expression `(let ((`x_1 t_1`)` `···` `(`x_n t_n`))` t`)`, the scope of each variable in $\{x_1, \ldots, x_n\}$ is the term t.
- In an expression `(match` t `((`p_1 t_1`)` `···` `(`p_n t_n`)))`, the scope of each variable occurring in pattern p_i is the corresponding term t_i ($1 \leq i \leq n$).

Shadowing All binders follow a lexical scoping discipline, consistent with the semantics of the SMT-LIB logic, as described in Section 5.3. In particular, a bound variable will shadow any variable or user-defined function symbol with the same name from an enclosing scope. For instance, in a term like `(forall ((a Int)) (> (+ a 1) 0))`, variable `a` would shadow any user-declared function symbol called `a`. 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 6 (No shadowing of theory symbols)**.** One exception to the shadowing rule above is that binders cannot shadow *theory function or sort symbols*, that is, function or sort symbols from the declaration (see Section 3.7) of a theory included in the current logic (see Section 3.10 and Subsection 4.2.1). In other words, variables cannot have the same name as a function symbol declared in the current logic, and sort parameters (see Section 4.2.3) cannot have the same name as a sort symbol declared in the current logic.[(14)]

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

3.6.3 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 τ`)` where τ 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.

The example below illustrates the need and use of `as` for typical functions associated with lists or arrays, such as the empty list constructor and the constant array constructor.

```
; Considering cons and nil as list constructors
(cons "abc" (as nil (List String)))

; Considering a const-array function, taking a value v and
; returning an array storing v at all positions
(= a ((as const-array (Array Int Real)) 0.0))

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

3.6.4 Annotations

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

3.6.5 Term attributes

Currently there are only two predefined term attributes: `:named` and `:pattern`. The values of the `:named` attribute range over the ⟨*symbol*⟩ category. The attribute can be used in scripts to give a closed term a symbolic name, which can 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 ⟨*term*⟩ 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 φ of a quantified formula of the form

$$(Q\ ((x_1\ \sigma_1)\ \cdots\ (x_k\ \sigma_k))\ \varphi)$$

[6]That is, function symbols with more than one possible output sort for the same sequence of input sorts (see Subsection 5.2.1).

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

Figure 3.1: Theory declarations.

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

$$\begin{array}{l}(Q\ ((x_1\ \sigma_1)\ \cdots\ (x_k\ \sigma_k))\ (!\ \varphi\ \texttt{:pattern}\ (p_{1,1}\ \cdots\ p_{1,n_1}) \\ \qquad\qquad\qquad\qquad\vdots \\ \qquad\qquad\qquad\qquad \texttt{:pattern}\ (p_{m,1}\ \cdots\ p_{m,n_m})))\end{array}$$

where each $p_{i,j}$ is a binder-free term with no annotations and the same well-sortedness requirements as the formula's body.[16]

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

The intended use of these patterns is to suggest to the solver 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 *theory declarations* written in the format specified in this section. This catalog is available on the SMT-LIB web site at www.smt-

[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.

lib.org. In earlier versions of the SMT-LIB standard, a theory declaration defined both a many-sorted *signature*, i.e., a collection of sorts and sorted function symbols, and a theory with that signature. The signature was determined by the collection of individual declarations of sort symbols and function symbols with an associated *rank*—specifying the sorts of the symbol's arguments and of its result.

From Version 2.0 on, theory declarations may also declare entire families of overloaded function symbols by using ranks that contain *sort parameters*, locally scoped sort symbols of arity 0. This kind of polymorphism 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 ⟨*attribute*⟩ 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.(17)

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

A theory declaration `(theory` T α_1 $\cdots$ α_n`)` defines a *theory schema* with name T and attributes $\alpha_1, \ldots, \alpha_n$. Each instance of the schema is a theory $\mathcal{T}_\Sigma$ with an *expanded* signature Σ, containing (zero or more) additional sort and function symbols with respect to those declared in T. Theories are defined as classes of first-order structures (or *models*) of signature Σ. 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 ⟨*sort_symbol_decl*⟩. A sort symbol declaration `(`s n α_1 $\cdots$ α_m`)` declares a sort symbol s of arity n, and may additionally contain zero or more annotations, each in the form of an ⟨*attribute*⟩. In this version, there are no predefined annotations for sort declarations.

The value of a `:funs` attribute is a non-empty sequence of possibly polymorphic function symbol declarations ⟨*par_fun_symbol_decl*⟩. A (monomorphic) function symbol declaration ⟨*fun_symbol_decl*⟩ of the form `(`c σ`)`, where c is an element of ⟨*spec_constant*⟩, declares c to have sort σ. For convenience, it is possible to declare all the special constants in ⟨*numeral*⟩ to have sort σ by means of the function symbol declaration `(NUMERAL` σ`)`. This is done for instance in the theory declaration in Figure 3.2. The same can be done for the set of ⟨*decimal*⟩ and ⟨*string*⟩ constants by using `DECIMAL` and `STRING`, respectively.

A function symbol declaration of the form

`(`f σ_1 $\cdots$ σ_n σ`)`

with $n \geq 0$ declares a *monomorphic function symbol* f with rank $\sigma_1 \cdots \sigma_n \sigma$, provided that σ and all σ_i's are monomorphic sorts.

A function symbol declaration of the form,

$$\texttt{(par (}u_1 \ \cdots \ u_k\texttt{) (}f \ \tau_1 \ \cdots \ \tau_n \ \tau\texttt{))}$$

with $k > 0$, $n \geq 0$ and $u_1, \ldots, u_k$ all distinct, declares a *polymorphic function symbol* f with rank $\tau_1 \cdots \tau_n \tau$ provided that τ and all τ_i's are polymorphic sorts over the (local) parameters $u_1, \ldots, u_k$, where `par` is a binder for these parameters. This effectively declares a *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 monomorphic sorts.

Note that a parameter will shadow any sort symbol with the same name. For instance, in a function symbol declaration like `(par (A) (f (Array A B) A))`, all occurrences of `A` are sort parameters regardless of the existence of a previously declared sort symbol `A`, whereas `B` must be a previously declared sort symbol.(18) [CB: What about shadowing of global sort parameters?]

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

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

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

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

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

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

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

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

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

(and Bool Bool Bool :left-assoc)

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

(< Real Real Bool :chainable)
```

```
(equiv Elem Elem Bool :chainable)

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

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

For many theories in SMT-LIB, in particular those with a finite signature, it is possible to declare all of their symbols using a finite number of sort and function symbol declarations in `:sorts` and `:funs` attributes. For others, such as, for instance, the theory of bit vectors, one would need infinitely many such declarations. In those cases, sort symbols and function symbols are defined informally, in plain text, in `:sorts-description`, and `:funs-description` attributes, respectively.(19)

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

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

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

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

The `:definition` attribute is meant to contain a natural language definition of the theory. While this definition is expected to be as rigorous as possible, it does not have to be a formal one.(20) For some theories, a mix of formal notation and natural language might be more appropriate. In the presence of polymorphic function symbol declarations, the definition must also specify the meaning of each instance of the declared symbol.(21)

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

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

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

1. They contain exactly one occurrence of the `:definition` and the `:values` attribute[8] 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 polymorphic function symbol declaration `(par (`$u_1 \cdots u_k$`) (`$f\ \tau_1 \cdots \tau_n\ \tau$`))`, any symbol other than f that is not a previously declared sort symbol must be one of the sort parameters $u_1, \ldots, u_k$.

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 and to define the symbols in those attributes in the same way as in `Core`. The full Core theory definition is shown in Figure 3.5. A more detailed description of this theory is provided in Section 3.8.

Integers

The theory declaration of Figure 3.2 defines all theories that extend the standard theory of the (mathematical) integers with additional *uninterpreted* sort and function symbols.[9] 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.

Arrays with extensionality

A schematic version of the theory of functional arrays with extensionality is defined in the theory declaration `ArraysEx` in Figure 3.3. Each instance gives a theory of (arbitrarily nested) arrays. For instance, with the addition of the nullary sort symbols `Int` and `Real`, we get an instance theory whose sort set S contains, inductively, `Bool`, `Int`, `Real` and all sorts of the form `(Array` $\sigma_1\ \sigma_2$`)` with $\sigma_1, \sigma_2 \in S$. This includes *flat array* sorts such as

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

[8] Which makes those attributes non-optional.

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

```
(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 3.2: A possible theory declaration for the integer numbers.

conventional *nested array* sorts such as

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

as well as nested sorts such as

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

with an array sort in the *index position* of the outer array sort.[22]

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

Sets 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 3.4. 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 `(Set` σ`)` or `(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 σ in the signature is internalized in the theory, since the set denoted by σ is also denoted by the constant `univSet` of (the powerset) sort `(Set` σ`)`.

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

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

Figure 3.3: The `ArraysEx` theory declaration.

Remark 7 (Instances of polymorphic 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 `(Array Int Real)`, without all the other nested array sorts over {`Int`, `Real`}. This, however, is not a problem 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 Core Theory

The core theory is a subtheory of all SMT-LIB theories. It introduces the basis sort and function symbols of the underlying logic, which are then automatically available in all theories. It provides the standard `Bool` sort for Boolean values. The theory also provides the usual set of Boolean connectives for building formulas, as well polymorphic function symbols for equality (`=`), disequality (`distinct`), and if-then-else (`ite`). Every theory declaration is assumed to contain implicitly the `:sorts` and `:funs` attributes of the `Core` theory declaration 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

```
(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 3.4: A possible declaration for a theory of sets and relations.

```
(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 3.5: The `Core` theory declaration.

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

```
(=> 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 Booleans with the standard Boolean operators plus `ite` and `distinct`. The set of values for the `Bool` sort is, predictably, {`true`,`false`}.

```
(theory HO-Core

 :sorts ( (-> 2 :right-assoc) )

 :funs ( (par (A B) (_ (-> A B :left-assoc) A B) ) )

 :definition
 "For every expanded signature Sigma, the instance of HO-Core with that signature
  is the theory consisting of all Sigma-models in which:
  - the sort constructor -> denotes the function space constructor;
  - for all sorts s1, s2 in Sigma,
    - (_ (-> s1 s2) s1 s2) denotes the function that
      returns the result of applying its first argument to its second."

 :values
 "For all sorts s1, s2, the set of values for the sort (-> s1 s2) consists of
  - an abstract value for each function (from a countable subset of the functions)
    of type s1 -> s2;
  - terms of the form (lambda ((x s1)) t) where t has sort s2 when every
    free occurrence of x in t has sort s1.
 "
)
```

Figure 3.6: The `HO-Core` theory declaration.

Another instance has a single additional sort symbol `U`, say, of arity 0, and a (possibly infinite) set 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.

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

3.9 Higher-order Core Theory

This version of SMT-LIB adds support for higher-order logic through the addition of a new theory, HO-Core, shown in Figure 3.6. The theory includes a binary sort constructor `->` for

sorts that denote function spaces (for example `(-> Int Real)` for the sort of functions from `Int` to `Real`) and an explicit function application operator `_`.

Note that, with this addition, the language distinguishes between *functions* of rank $\tau_1\tau_2$ and *constants* or variables of sort `(->` τ_1 τ_2`)`. For instance, the latter can be passed as an argument to other functions as any other variable but the former cannot.(23) This is no real limitation, though, because it is possible to convert from one to the other thanks to the apply operator `_` and the abstraction binder `lambda`. For instance, if `f` is a function symbol of rank $\tau_1\tau_2$, one can define a corresponding higher-order constant with the command (see 4.2.3)

`(define-const c_f (lambda ((x` τ_1`)) (f x)))` .

Conversely, if `c` is a constant of sort `(->` τ_1 τ_2`)`, the corresponding function can be defined by the command

`(define-fun f_c ((x` τ_1`))` τ_2 `(_ c x))` .

The `->` sort constructor is defined as right-associative, allowing for instance the syntax `(->` τ_1 τ_2 τ_3`)` to be used in place of the syntax `(->` τ_1 `(->` τ_2 τ_3`))`. Correspondingly, the application operator `_` is defined as left-associative, allowing the syntax `(_` t_1 t_2 t_3`)` to be used in place of the syntax `(_ (_` t_1 t_2`)` t_3`)`. Note that this also allows the *partial* application `(_ g t)` of a constant `g` of sort `(->` τ_1 τ_2 τ_3`)`, which is a term of sort `(->` τ_2 τ_3`)`.

As an additional simplification, which does not introduce ambiguities, it is possible to omit the `_` symbol altogether in applications and write `(g t)` instead of `(_ g t)`.

Remark 8. Note that the SMT-LIB logic remains first-order in syntax: it is not possible to have a function symbol g of rank $\tau_1 \cdots \tau_n$ (with $n > 1$) as an argument to another function, as in `(f g)`. The function g has to be encapsulated into a λ-abstraction first. For instance, if `f` is a function symbol of rank `(->` τ_1 τ_2`)` τ and g is a function symbol of rank $\tau_1\tau_2$, then the term `(f (lambda ((x` τ_1`)) (g x)))` is well sorted, and has sort τ.

Similarly, it is not possible to return a function of rank $\tau_1 \cdots \tau_n$ (with $n > 1$) from another function, but it is possible to return a constant of sort `(->` τ_1 $\cdots$ τ_n`)`.(24)

3.10 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.(25)

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

`:theories    :language    :extensions    :notes    :values` .

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

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

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

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 $\mathcal{T}$ of specific instances of the theory declaration schemas $T_1, \ldots, T_n$. The exact combination mechanism that yields $\mathcal{T}$ is defined formally in Section 5.5. The effect of this attribute is to declare that the logic's sort and function symbols consist of those of the combined theory $\mathcal{T}$, and that the logic's semantics is restricted to the models of $\mathcal{T}$, as specified in more detail in Section 5.5.

The `:language` attribute describes in free text the logic's *language*, a specific class of SMT-LIB formulas. This information is useful for tailoring SMT solvers to the specific sublanguage of formulas used in an input script.[26] The formulas in the logic's language are built over (a subset of) the signature of the associated theory $\mathcal{T}$, as specified in this attribute. In the context of a command script the language of a logic is implicitly expanded by `let` constructs in formulas as well as user-defined (but not user-declared) sort and function symbols. In other words, a formula φ used in a script is considered to belong to a certain logic's language iff the formula obtained from φ by replacing all let variables and all defined sort and function 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.[27]

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.

3.10.1 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.

Propositional logic

Propositional logic can be readily defined by an SMT-LIB logic declaration. The logic's theory is the instance of the `Core` theory declaration whose signature adds infinitely-many function symbols of rank `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 `ite` operator of `Core`.

Quantified Boolean logic

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

Linear integer arithmetic

Linear integer arithmetic can be defined as an SMT-LIB logic. This logic is indeed part of the official SMT-LIB catalog of logics and is called `QF_LIA` there. Its theory is an extension of the theory of integers and the Booleans with uninterpreted constant symbols. That is, it is the instance of the theory declaration `Ints` from Figure 3.2 whose signature adds to the symbols of `Ints` infinitely many *free constants*, new function symbols of rank `Int` or of rank `Bool`.

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

3.11 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

```
⟨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⟩ )
                      |  ( declare-sort-parameter ⟨symbol⟩ )
                      |  ( define-const ⟨symbol⟩ ⟨sort⟩ ⟨term⟩ )
                      |  ( 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⟩*
```

Figure 3.7: SMT-LIB Commands.

```
⟨b_value⟩   ::=  true | false

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

Figure 3.8: Command options.

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

Figure 3.9: Info flags.

of these arguments can be (parenthesis-delimited) lists of variable length. The full list of commands is provided in Figure 3.7.

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 3.8 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 3.9 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 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*.

3.11.1 Command responses

The possible responses that a solver can produce in response to commands are shown in Figure 3.10. Every response must be an instance of ⟨*general_response*⟩ which specifies generic response possibilities as well as command-specific responses for certain commands (specified by ⟨*specific_success_response*⟩). In addition, with the `:print-success` option set to `true`, a solver returns `success` after a successful command.

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.[10] The string literals `"stdout"` and `"stderr"` are reserved to refer specially to the corresponding standard process channels (as opposed to disk files with that name).

Specific Responses. Specific responses are defined, in Figure 3.10, for the following commands:

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

```
⟨error-behavior⟩          ::= immediate-exit | continued-execution
⟨reason-unknown⟩          ::= memout | incomplete | ⟨s_expr⟩
⟨model_response⟩          ::= ( define-fun ⟨function_def⟩ ) | ( define-fun-rec ⟨function_def⟩ )
                            | ( define-funs-rec ( ⟨function_dec⟩^{n+1} ) ( ⟨term⟩^{n+1} ) )
⟨info_response⟩           ::= :assertion-stack-levels ⟨numeral⟩
                            | :authors ⟨string⟩
                            | :error-behavior ⟨error-behavior⟩
                            | :name ⟨string⟩
                            | :reason-unknown ⟨reason-unknown⟩
                            | :version ⟨string⟩
                            | ⟨attribute⟩
⟨valuation_pair⟩          ::= ( ⟨term⟩ ⟨term⟩ )
⟨t_valuation_pair⟩        ::= ( ⟨symbol⟩ ⟨b_value⟩ )

⟨check_sat_response⟩            ::= sat | unsat | unknown
⟨echo_response⟩                 ::= ⟨string⟩
⟨get_assertions_response⟩       ::= ( ⟨term⟩* )
⟨get_assignment_response⟩       ::= ( ⟨t_valuation_pair⟩* )
⟨get_info_response⟩             ::= ( ⟨info_response⟩+ )
⟨get_model_response⟩            ::= ( ⟨model_response⟩* )
⟨get_option_response⟩           ::= ⟨attribute_value⟩
⟨get_proof_response⟩            ::= ⟨s_expr⟩
⟨get_unsat_assump_response⟩     ::= ( ⟨term⟩* )
⟨get_unsat_core_response⟩       ::= ( ⟨symbol⟩* )
⟨get_value_response⟩            ::= ( ⟨valuation_pair⟩+ )
⟨specific_success_response⟩     ::= ⟨check_sat_response⟩ | ⟨echo_response⟩
                                  | ⟨get_assertions_response⟩ | ⟨get_assignment_response⟩
                                  | ⟨get_info_response⟩ | ⟨get_model_response⟩
                                  | ⟨get_option_response⟩ | ⟨get_proof_response⟩
                                  | ⟨get_unsat_assumptions_response⟩
                                  | ⟨get_unsat_core_response⟩ | ⟨get_value_response⟩

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

Figure 3.10: Command responses.

```
(set-option :print-success true)
; success

(set-info :smt-lib-version 2.7)
; success

(set-logic QF_LIA)
; success

(declare-const w Int)
; success

(declare-const x Int)
; success

(declare-const y Int)
; success

(declare-const z Int)
; success

(assert (> x y))
; success

(assert (> y z))
; success

(push 1)

(assert (> z x))

(check-sat)
; unsat

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

(pop 1)

(push 1)

(check-sat)
; sat

(exit)
```

Figure 3.11: Example script (over two columns), with expected solver responses in comments.

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

See Chapter 4 for more details.

3.11.2 Example scripts

We demonstrate some allowed behavior of a hypothetical solver in response to an example script. Each command is followed by example legal output from the solver in a comment, if there is any. The script in Figure 3.11 makes two background assertions and then conducts two independent queries. The `get-info` command requests information on the search using

```
(set-info :smt-lib-version 2.7)
...
(set-option :produce-models true)

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

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

(check-sat)
; sat

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

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

```
...

(check-sat)
; sat

(get-value (a))
; ( (a (as @array1 (Array Int (List Int))))
; )

(get-value ((select @array1 2)))
; (((select (as @array1 (Array Int (List Int))) 2)
;   (as @list0 (List Int))
;  )
; )

(get-value ((first @list0) (rest @list0)))
; (((first (as @list0 (List Int))) 1)
;  ((rest (as @list0 (List Int))) (as nil (List Int)))
; )
```

Figure 3.12: Another example script (excerpt), with expected solver responses in comments.

the `:all-statistics` flag.[11] The script in Figure 3.12 uses the `get-value` command to get information about a particular model of the formula that the solver has reported satisfiable.

3.11.3 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.[12]

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

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

[12]The 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

CHAPTER 4

Operational Semantics of SMT-LIB

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.

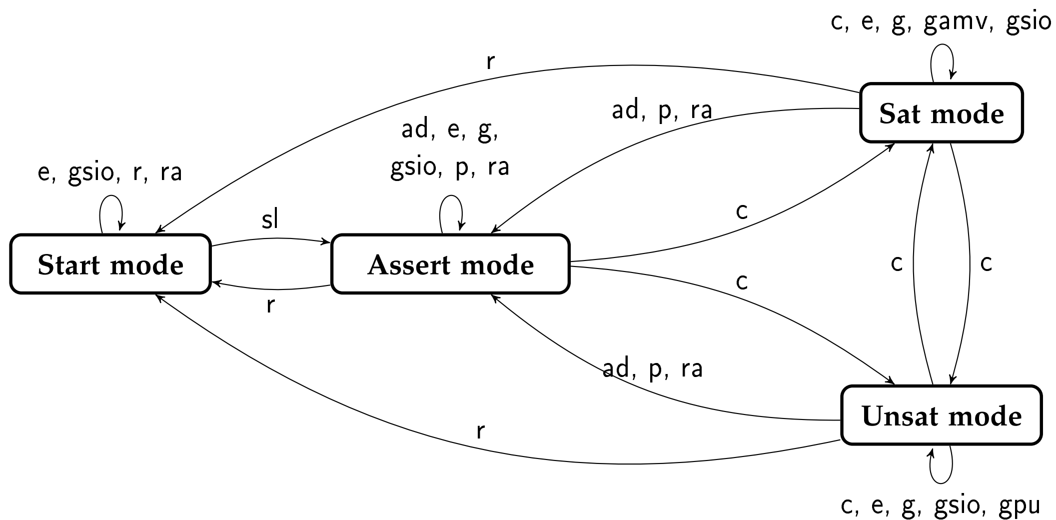
Note that the primary goal of the SMT-LIB standard is, first and foremost, to support convenient interaction with other programs, not human interaction. This has some influence on the design of the command language.

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,



Command name abbreviations:

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

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).[(28)]

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.

Errors and solver state. Solvers have two options when encountering errors. For both options, they first print an error message in the ⟨*general_response*⟩ format. Then, they may either immediately exit with a non-zero exit status, or continue accepting commands. In the second case, the solver's state remains unmodified by the error-generating command, except possibly for timing and diagnostic information. In particular, the assertion stack, discussed in Section 4.1.4, is unchanged.[(29)]

The predefined `:error-behavior` attribute can be used with the `get-info` command to check which error behavior the tool supports (see Section 4.1.8 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 `define-sort` and `define-fun` just as they are, without replacing them by the expression they are defined to be equal to. This approach generally keeps 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:

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

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:

`assert`, `declare-sort`, `declare-fun`, `declare-const`, `define-sort`, `define-fun`, `define-fun-rec`, `define-funs-rec`, `pop`, `push`, `reset`, and `reset-assertions`.

4.1.5 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.[(30)] As an alternative, declarations and definitions can all be made *global* by running the solver with the option `:global-declarations` set to `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 `reset-assertions` and can only be removed by a global reset, achieved with the `reset` command.[(31)]

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.[(32)]

4.1.6 In-line definitions

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

`(!` t_1 `:named` f_1`), ..., (!` t_n `: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

`(define-fun` f_1 `()` σ_1 t'_1`)`
⋮
`(define-fun` f_n `()` σ_n t'_n`)`
c'

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

By these semantics, each *label* f_i can occur, as a constant symbol, in any subexpression of c that comes after `(!` t_i `:named` f_i`)` in the 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 `define-fun`, labels have an additional, dedicated use in the commands `get-assignment` and `get-unsat-core` (see Section 4.2).

4.1.7 Solver options

Solver options may be set using the `set-option` command, and their current values can be obtained using the `get-option` command. If a solver does not support the setting of a particular option, for either command it should output `unsupported`.

Solver-specific option names are allowed and indeed expected. A set of standard options is presented in this subsection; refer to Figure 3.8 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: `false` support: required
: Setting this option to `true` causes the solver to print `success` as a response 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`.

`: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.(33)

`:verbosity` default: no standard default value 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(34) 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 3.10. 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 3.10) providing various statistics on the execution of the most recent check command.

No standard statistics are defined for the time being,[35] 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.[36]

`: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_unknown*⟩ giving a short reason why the solver could not successfully check satisfiability. In general, this reason can be provided by a solver-defined s-expression. Two predefined s-expressions are `memout`, for out of memory, and `incomplete`, which indicates that the solver knows it is incomplete for the class of formulas containing the most recent check query.

`:version` support: required
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.

4.2.1 (Re)starting and terminating

`(reset)` resets the solver completely to the state it had after it was started and before it started reading commands.(37)

`(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 `ALL`. With this argument, the solver sets the logic to the most general logic it supports.(38) Note that while the reaction to `(set-logic 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.(39)

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

`(declare-sort-parameter` s`)` adds *global* sort parameter s to the current signature. The command reports an error if s is a sort symbol, sort parameter, or theory symbol already present in the current signature.

`(declare-sort` s n`)` adds sort symbol s with associated arity n. It is an error if s is a sort symbol or parameter already present in the current signature.

`(define-sort` s `(`$u_1 \cdots u_n$`)` τ`)` with $n \geq 0$ adds sort symbol s with arity n and associates it with sort τ, where the u_i's are (local) sort parameters, and τ may contain any of $u_1, \ldots, u_n$, but no global sort parameters.

Subsequent well-sortedness checks must treat a sort term like `(`s $\tau_1 \cdots \tau_n$`)` as an abbreviation for the term obtained by simultaneously substituting τ_i for u_i, for $i \in \{1, \ldots, n\}$, in τ.(40)

The command reports an error if s is a sort symbol or parameter already present in the current signature or if τ is not a well-defined (polymorphic) sort with respect to the current signature extended with sort parameters $u_1, \ldots, u_n$. This restriction prohibits (meaningless) circular definitions where τ contains s. The u_i sort parameters shadow any previously user-declared sort symbol or parameter with the same name. However, they do not shadow theory sort symbols (just as bound variables are not permitted to shadow declared theory symbols). The command reports an error if a theory sort symbol is used as a sort parameter u_i.

`(declare-fun` f `(`$\tau_1 \cdots \tau_n$`)` τ`)` with $n \geq 0$ adds a new symbol f with associated rank $\tau_1 \cdots \tau_n \tau$. The command reports an error if a function symbol with name f is already present in the current signature. Note that $\tau_1 \cdots \tau_n$, and τ may contain sort parameters. In that case f is polymorphic.

`(declare-const` f τ`)` has the same effect as the command `(declare-fun` f `()` τ`)`.

`(declare-datatypes ((`δ_1 k_1`)` $\cdots$ `(`δ_n k_n`)) (`$d_1 \cdots d_n$`))` 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 `(par (`$u_1 \cdots u_k$`)` l`)` where $u_1, \ldots, u_k$ are sort parameters;(41) otherwise, it is just l. In either case, l is a (parenthesis-delimited) list of *one or more* expressions of the form

`(`c `(`s_1 τ_1`)` $\cdots$ `(`s_m τ_m`))`,

where (i) $m \geq 0$, (ii) c is a symbol, a *constructor for* δ, and (iii) for each $j = 1, \ldots, m$, s_j is a symbol, a *selector for* c, and (iv) τ_j is a sort term that contains no occurrences of $\delta_1, \ldots, \delta_n$ below its top symbol.

In the polymorphic case, the terms τ_i can contain only the sort parameters in the list $u_1, \ldots, u_k$.[4] The datatype δ must be *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 polymorphic case, the command has the effect of declaring each δ 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 monomorphic 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.[(42)] This allows the declaration of recursive and mutually recursive datatypes.[(43)]

On successfully executing this command, for each constructor c in a declared datatype δ, the solver will also automatically declare a *tester* with rank δ `Bool`. The tester's name is an indexed identifier (see Section 3.3) of the form `(_ is c)`.

```
; 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
```

[4] No previously declared sort parameter that is not shadowed by a u_i can be used.

```
(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.(44)

`(declare-datatype` δ d`)` is an abbreviation of

$$\texttt{(declare-datatypes ((}\delta\ \texttt{0)) (}d\texttt{))}$$

if δ is not parametric, and an abbreviation of

$$\texttt{(declare-datatypes ((}\delta\ k\texttt{)) (}d\texttt{))}$$

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

```
; an enumeration datatype
(declare-datatype Color ( (red) (green) (blue) ))

(declare-datatype IntList
  ( (empty)
    (insert (head Int) (tail IntList) )))

(declare-datatype List (par (E)
  ( (nil)
    (cons (car E) (cdr (List E)) ))))

(declare-datatype Option (par (X)
  ( (none)
    (some (val X) ))))

(declare-datatype Pair (par (X Y)
 ( (pair (first X) (second Y)) ))))
```

`(define-fun` f `((`x_1 τ_1`)` $\cdots$ `(`x_n τ_n`))` τ t`)` with $n \geq 0$ and t not containing f is semantically equivalent to the command sequence

`(declare-fun` f `(`$\tau_1 \cdots \tau_n$`)` τ`)`
`(assert (forall ((`x_1 τ_1`)` $\cdots$ `(`x_n τ_n`)) (= (`f $x_1 \cdots x_n$`)` t`)).`

Note that the restriction on t prohibits recursive or mutually recursive definitions, which are instead provided by `define-fun-rec` and `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 τ with respect to the current signature extended with the sort associations $(x_1 : \tau_1)$, ..., $(x_n : \tau_n)$.

(define-const f τ t) has the same effect as (define-fun f () τ t).

(define-funs-rec (d_1 ⋯ d_m) (t_1 ⋯ t_m)), where $m > 0$ and for $i = 1, \ldots, m$, d_i has the form

$$(f_i\ ((x_{i,1}\ \tau_{i,1})\ \cdots\ (x_{i,n_i}\ \tau_{i,n_i}))\ \tau_i)$$

with $n_i \geq 0$ and $f_1, \ldots, f_m$ pairwise distinct, is semantically equivalent to the command sequence

```
(declare-fun f_1 (τ_1,1 ··· τ_1,n_1) τ_1)
          ⋮
(declare-fun f_m (τ_m,1 ··· τ_m,n_m) τ_m)
(assert (forall ((x_1,1 τ_1,1) ··· (x_1,n_1 τ_1,n_1)) (= (f_1 x_1,1 ··· x_1,n_1) t_1))
                          ⋮
(assert (forall ((x_m,1 τ_m,1) ··· (x_m,n_m τ_m,n_m)) (= (f_m x_m,1 ··· x_m,n_m) t_m)) .
```

This command can be used to define multiple functions recursively, in particular, mutually recursively.(45) 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, `define-funs-rec` imposes no requirements that each f_i be terminating (a meaningless notion in our context) or even well-defined.[5] 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 τ_i with respect to the current signature extended with the sort associations $(f_1 : \tau_{1,1}\ \cdots\ \tau_{1,n_1}\tau_1)$, ..., $(f_m : \tau_{m,1}\ \cdots\ \tau_{m,n_m}\tau_m)$ and $(x_{i,1} : \tau_{i,1})$, ..., $(x_{i,n_i} : \tau_{i,n_i})$.

Note that, for each i, $\tau_{i,1}\ \cdots\ \tau_{i,n_i}$, and τ_i may contain sort parameters. In that case f_i is polymorphic.

(define-fun-rec f ((x_1 τ_1) ⋯ (x_n τ_n)) τ t) has the same effect as

$$\texttt{(define-funs-rec ((}f\ \texttt{((}x_1\ \tau_1\texttt{)}\ \cdots\ \texttt{(}x_n\ \tau_n\texttt{))}\ \tau\texttt{))}\ \texttt{(}t\texttt{))}$$

It provides a simpler syntax to define individual recursive functions.

[5] In fact, it is even possible, although certainly not desirable, to have a definition like `(define-funs-rec ((f ((x Bool)) Bool)) ((not (f x))) )`, which makes the set of formulas in the context unsatisfiable.

4.2.4 Asserting and inspecting formulas

`(assert t)` where t is a well-sorted formula (i.e., a well-sorted element of ⟨*term*⟩ of sort `Bool`), adds t to the current assertion level. The well-sortedness requirement is with respect to the current signature. If any subterm of t has a polymorphic sort τ over sort parameters $u_1, \ldots, u_n$, then the sort parameters are implicitely universally quantified in the assertion. That is, when a later check command is issued, the assertion will hold for all instances of $u_1, \ldots, u_n$ with ground sorts in the signature at that time.

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

Remark 9. Notice that the commands

```
(declare-sort-parameter A)
(assert (forall ((x A) (y A)) (= x y)))
```

have the effect of stating that every sort has a domain of size one. Consider the negation of this assertion. It should state that at least one sort has a domain of size at least two. However, the commands

```
(declare-sort-parameter A)
(assert (not (forall ((x A) (y A)) (= x y))))
```

have instead the effect of stating that *every* sort has a domain of size at least two. Properly negating the original assertion can instead be done by introducing a fresh sort symbol, as in

```
(declare-sort U 0)
(assert (not (forall ((x U) (y U)) (= x y))))
```

In general, this means that, when polymorphic sorts are involved, special care must be taken to ensure that the formula asserted matches the intent. This is especially true when checking, for example, the validity of an implication `(=> p q)` by asserting p and `(not q)`, and then checking for satisfiability.

`(get-assertions)` causes the solver to print the current set of all asserted formulas as a sequence of the form $(f_1 \cdots f_n)$. Each f_i is a formula syntactically identical, modulo whitespace, to one of the formulas entered with an `assert` command and currently in the context. Solvers are not allowed to print formulas equivalent to or derived from the asserted formulas.(46)

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

4.2.5 Checking for satisfiability

`(check-sat)` has the same effect as `(check-sat-assuming ())`.

`(check-sat-assuming (`$a_1 \cdots a_n$`))` where $n \geq 0$ and $a_1, \cdots, a_n$ are terms of sort `Bool`, instructs the solver to check whether the conjunction of all the formulas in the current context *and* the *assumptions* $a_1, \cdots, a_n$ is satisfiable in the extension of the current logic with all the current user-declared and user-defined symbols. The assumptions $a_1, \ldots, a_n$ must be formulas, i.e., terms of sort `Bool`.(47)

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 ⟨*check_sat_response*⟩ in Figure 3.10. 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 **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 **A** is required to satisfy all currently asserted formulas and current assumptions only if the most recent check command reported `sat`.(48)

The internal representation of the model **A** is not exposed by the solver. Similarly to an abstract data type, the model can be inspected only through the three commands below. As a consequence, it can even be partial internally and extended as needed in response to successive invocations of some of these commands.[6]

[6] In 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 $n > 0$ and each t_i is a well-sorted closed quantifier-free term, returns for each t_i a value term v_i[7] 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 `@`.(49) Since these are solver-defined, their sort is not known to the user. Therefore, additionally, each occurrence of an abstract value a of sort σ in $v_1, \ldots, v_n$ has to be contained in a term of the form `(as` $a\ \sigma$`)` 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.[8]

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.(50)

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

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

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

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

`(get-model)` returns a list $(d_1 \ \cdots \ d_k)$ 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:[51]

- `(define-fun` f `((`x_1 σ_1`)` $\cdots$ `(`x_n σ_n`))` σ t`)`

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

  `(forall ((`x_1 σ_1`)` $\cdots$ `(`x_n σ_n`)) (= (`f 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 $n = 0$).

- `(define-funs-rec ((`f_1 `((`$x_{1,1}$ $\sigma_{1,1}$`)` $\cdots$ `(`x_{1,n_1} σ_{1,n_1}`))` σ_1`)` $\cdots$
  `(`f_p `((`$x_{p,1}$ $\sigma_{p,1}$`)` $\cdots$ `(`x_{p,n_p} σ_{p,n_p}`))` σ_p`))` `(`t_1 $\cdots$ t_p`))`

  where $n_i > 0$ for $i = 1, \ldots, p$, and the formula

  `(and (forall ((`$x_{1,1}$ $\sigma_{1,1}$`)` $\cdots$ `(`$x_{1,n}$ σ_{1,n_1}`)) (= (`f_1 $x_{1,1}$ $\cdots$ x_{1,n_1}`)` t_1`))`
  $\vdots$
  `(forall ((`$x_{p,1}$ $\sigma_{p,1}$`)` $\cdots$ `(`x_{p,n_p} σ_{p,n_p}`)) (= (`f_p $x_{p,1}$ $\cdots$ x_{p,n_p}`)` t_p`)))`

  is well-sorted and satisfied by **A**.

- `(define-fun-rec` f `((`x_1 σ_1`)` $\cdots$ `(`x_n σ_n`))` σ t`)` where $n > 0$ and the formula

  `(forall ((`x_1 σ_1`)` $\cdots$ `(`x_n σ_n`)) (= (`f x_1 $\cdots$ x_n`)` t`))`

  is well-sorted and satisfied by **A**.

A user-declared or user-defined function symbol can occur in the terms t, $t_1, \ldots t_p$ above only if it has been previously defined in the model or is one of the symbols being recursively defined.

Similarly to the response of `get-value`, the terms $t, t_1, \ldots, t_m$ above are allowed to contain symbols not in the current signature only if they are abstract values. Morevever, each occurrence of an abstract value a of sort σ in $t, t_1, \ldots, t_m$ has to be contained in a term of the form `(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.[52]

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 $(a_1 \cdots a_n)$ of the assumptions in the most recent `check-sat-assuming` command. These assumptions are such that issuing the command `(check-sat-assuming (`$a_1 \; \cdots \; a_n$`))` instead would have still produced an `unsat` response. The returned set is a space-delimited list of assumptions surrounded by parentheses and is not required to be minimal.(53)

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.(54) The only requirement is that, like all responses, it be a member of ⟨*s_expr*⟩.

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

`(get-unsat-core)` Let A be the set of assertions in the context and B the set of assumptions in the most recent call to `check-sat-assuming` (where, as defined above, a call to `check-sat` is considered a call to `check-sat-assuming` with no assumptions). Let A be further partitioned into A_n and A_u, where A_n contains assertions that were asserted with a command of the form `(assert (! `t` :named `f`))` and $A_u = A \setminus A_n$. The names f are called *labels*, and we call A_n the set of *labeled assertions* and A_u the set of *unlabeled assertions*. The result of this command is a sequence `(`$f_1 \; \cdots \; f_n$`)` of labels from the labeled assertions such that the corresponding subset A'_n of A_n has the property that $A'_n \cup A_u \cup B$ is, by itself, unsatisfiable. Furthermore, we require that if B' is the result of any call to `get-unsat-assumptions` in the same solver state, then $A'_n \cup A_u \cup B'$ is also unsatisfiable.

Note that assumptions and unlabeled formulas are never included in the result of `get-unsat-core`. However, it is possible to get a full set of unsatisfiable assertions by labeling all assertions and combining the result of `get-unsat-core` with the result of `get-unsat-assumptions`. In practice, not labeling assertions may be useful for unsat core detection purposes when the user is sure that the set of all unlabeled assertions is satisfiable. In such cases, users are often only interested in the part of the unsat core coming from additional assertions on top of this satisfiable base.

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 ⟨*info_flag*⟩ 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 ⟨*attribute_value*⟩. 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.(55)

`(set-info a)` where *a* is an element of ⟨*attribute*⟩ has no effect on the assertion stack. Its only purpose is to allow the insertion of structured meta information in a script.(56) 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 ⟨*attribute*⟩).

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., `2.7`). For benchmarks in the official repository a call to `set-info` with this attribute can occur only as the first command of a script.

`:source` 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.

`:category` 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.(57)

`:license` 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.

`:status` 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.(58)

Remark 10 (`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.(59)

CHAPTER 5

Logical Semantics of SMT-LIB Formulas

The underlying logic of the SMT-LIB language is a variant of many-sorted first-order logic (FOL) with equality [Man93, Gal86, End01], although it incorporates some syntactic and semantic 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, as well as, starting with Version 2.7, the use of functions as first-class values. The latter is achieved by the addition of a background theory of (higher-order) functions, which are modeled in the concrete syntax as values of sort `(->` τ_1 τ_2`)` and are constructible with a λ-abstraction binder. New to Version 2.7 is also the addition of prenex polymorphism in user-scripts, achieved through the declaration of sort parameters, which can then be used to make assertions containing terms of polymorphic sort.

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), and the logic's type system has no function types, no type quantifiers, no dependent types, and no provisions for subsort 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** and the **HO-Core** background theories introduced in Sections 3.8 and 3.9. 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 first-order formulas (or *axioms*).

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

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

Figure 5.1: Abstract syntax for sort terms

this document. Until then, we will appeal to the reader's intuition and rely on the fact that the translation is defined as one would expect.

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

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

- an infinite set $\mathcal{S}$ of *sort symbols* s containing the symbol **Bool**,
- an infinite set $\mathcal{U}$ of *sort parameters* u,
- an infinite set $\mathcal{X}$ of *variables* x,
- an infinite set $\mathcal{F}$ of *function symbols* f containing the symbols $\approx$, $\wedge$, and $\neg$,
- the set $\mathcal{W}$ of *Unicode character strings* w,
- a two-element set $\mathcal{B} = \{\mathbf{true}, \mathbf{false}\}$ of *Boolean values* b,
- the set $\mathcal{N}$ of *natural numbers* n,
- an infinite set $\mathcal{TN}$ of *theory names* T,
- an infinite set $\mathcal{L}$ of *logic names* L.

5.1 The language of sorts

In many-sorted logics, terms are typed, or *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 *sort terms* built with symbols from the set $\mathcal{S}$ above. Formally, we have the following.

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

1. every $u \in \mathcal{U}$ is a sort;
2. every $s \in S$ with $\mathrm{ar}(s) = 0$ is a sort;
3. If $s \in S$ and $\mathrm{ar}(s) = n > 0$ and $\tau_1, \ldots, \tau_n$ are sorts, then the term $s\,\tau_1 \cdots \tau_n$ is a sort.

$$
\begin{array}{llll}
\text{(Patterns)} & p & ::= & x \mid f\, x^* \\
\text{(Terms)} & t & ::= & x \mid f\, t^* \mid f^\tau\, t^* \mid \lambda\,(x{:}\tau)\; t \mid \exists\,(x{:}\tau)\; t \mid \forall\,(x{:}\tau)\; t \\
 & & \mid & \textbf{let}\; (x = t)^+\; \textbf{in}\; t \mid \textbf{match}\; t\; \textbf{with}\; (p \to t)^+
\end{array}
$$

Figure 5.2: Abstract syntax for unsorted terms

We say that $s \in S$ has (or is of) *arity* n if $\mathrm{ar}(s) = n$. A sort is *polymorphic* if it contains a sort parameter; it is *monomorphic* otherwise. □

As an example of a sort, if **Int** and **Real** are sort symbols of arity 0, **List** is a sort symbol of arity 1, and $\rightarrow$ and **Array** are sort symbols of arity 2, then the expression

$$\textbf{Int} \rightarrow (\textbf{List}\ (\textbf{Array}\ \textbf{Int}\ (\textbf{List}\ \textbf{Real})))$$

and all of its subexpressions are monomorphic sorts. We are using the constructor for function sorts $\rightarrow$ as an infix, right-associative operator to improve readability.

Note that function symbol declarations in theory declarations (defined later) also use polymorphic sorts. Similar to the example above, if u_1, u_2 are parameters, that is, elements of $\mathcal{U}$, the expression

$$u_1 \rightarrow u_2 \rightarrow (\textbf{List}\ (\textbf{Array}\ u_1\ (\textbf{List}\ u_2)))$$

and all of its subexpressions are polymorphic sorts.

An abstract syntax for monomorphic sorts σ and polymorphic sorts τ, which ignores arity constraints for simplicity, is provided in Figure 5.1. Note that every monomorphic sort is a polymorphic sort, but not vice versa. In the following, we just use "sort" to refer to possibly polymorphic sorts and we use "monorphic sort" for the more restricted case as needed.

5.2 The language of terms

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

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

Binders

For all $n \geq 0$, distinct variables $x, x_1, \ldots, x_n \in \mathcal{X}$ and sort τ,

- the prefix construct $\lambda\, x{:}\tau\ _$ is a *function abstraction binder* for x;
- the prefix construct $\exists\, x{:}\tau\ _$ is a *sorted existential binder (or existential quantifier)* for x;
- the prefix construct $\forall\, x{:}\tau\ _$ is a *sorted universal binder (or universal quantifier)* for x;

- the mixfix construct **let** $x_1 = _ \cdots x_n = _$ **in** $_$ is a *(parallel-)let binder* for $x_1, \ldots, x_n$.
- the mixfix construct **match** $_$ **with** $p_1 \rightarrow _ \cdots p_n \rightarrow _$ 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 3.6.2). 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

In the concrete syntax, 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. For this reason annotations do not occur in the abstract syntax.

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

5.2.1 Signatures

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

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

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

- a total mapping $\mathrm{sel}_\Sigma : \Sigma^\mathrm{C} \to (\Sigma^\mathrm{G})^*$, 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 $\mathrm{tes}_\Sigma : \Sigma^\mathrm{C} \to \Sigma^\mathrm{T}$, assigning a tester to each constructor;
- a partial mapping from $\mathcal{X}$ to $Sort(\Sigma^\mathrm{S},\mathcal{U})$, assigning a sort to some of the variables in $\mathcal{X}$;[1]
- a left-total *ranking* relation[2] R from Σ^F to $Sort(\Sigma^\mathrm{S},\mathcal{U})^+$, assigning at least one *rank* to each function symbol and such that
  1. each constructor $c \in \Sigma^\mathrm{C}$ has a rank of the form $\tau_1 \dots \tau_n \tau$ where the top symbol of τ is the sort symbol c is associated with;
  2. for all constructors $c \in \Sigma^\mathrm{C}$, selectors $g_1 \cdots g_n = \mathrm{sel}_\Sigma(c)$, and sorts $\tau_1, \dots, \tau_n, \tau \in Sort(\Sigma^\mathrm{S},\mathcal{U})$, if $(c, \tau_1 \cdots \tau_n \tau) \in R$ then $(g_i, \tau\, \tau_i) \in R$ for all $i = 1, \dots, n$;
     where
  3. for all constructors $c \in \Sigma^\mathrm{C}$ and testers $p = \mathrm{tes}_\Sigma(c)$, if $(c, \tau_1 \cdots \tau_n \tau) \in R$ then $(p, \tau\, \mathbf{Bool}) \in R$;
  4. there is no constructor $c \in \Sigma^\mathrm{C}$ such that $(c, \overline{\tau}_1 \tau), (c, \overline{\tau}_2 \tau) \in R$ for distinct $\overline{\tau}_1$ and $\overline{\tau}_2$.(60)

A sort in $Sort(\Sigma^\mathrm{S},\mathcal{U})$ is an *(algebraic) datatype* if its top symbol is assigned a non-empty set of constructors. □

Remark 11. 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.

Notation 1. In the following, we will write $Sort(\Sigma,\mathcal{U})$ as an abbreviation of $Sort(\Sigma^\mathrm{S},\mathcal{U})$, and $Sort(\Sigma)$ as the subset of all monorphic sorts in $Sort(\Sigma,\mathcal{U})$.

Definition 3 (Sort substitution and instance)**.** A *sort substitution* θ maps each sort parameter u in $\mathcal{U}$ to a sort $\theta(u)$ in $Sort(\Sigma,\mathcal{U})$. A *monomorphic substitution* is a sort substitution that maps every parameter to a monomorphic sort. A *sort instance of a polymorphic sort* τ is a sort $\theta(\tau)$ obtained by sort substitution, i.e., substituting every occurrence of all sort parameters u by the associated sort $\theta(u)$ in $Sort(\Sigma,\mathcal{U})$. A *sort instance of a polymorphic rank* $\tau_1 \cdots \tau_n \tau$ is a rank $\theta(\tau_1) \cdots \theta(\tau_n)\theta(\tau)$ for some sort substitution θ.

We will use the syntax $\{u_1 \mapsto \tau_1, \cdots u_n \mapsto \tau_n\}$ to denote a sort substitution that maps u_i to τ_i for each $i = 1, \dots n$ and maps every other sort parameter to itself.

We will work with *ranked* function symbols and *sorted* variables in a signature. Formally, given a signature Σ, a *sorted variable* is a pair (x, τ) in $\mathcal{X} \times Sort(\Sigma,\mathcal{U})$, which we write as $x{:}\tau$. We write $x{:}\tau \in \Sigma$ to denote that x has sort τ in Σ. A *ranked function symbol* is a pair $(f, \tau_1 \cdots \tau_n \tau)$ in $\mathcal{F} \times Sort(\Sigma,\mathcal{U})^+$, which we write as $f{:}\tau_1 \cdots \tau_n \tau$. We write $f{:}\tau_1 \cdots \tau_n \tau \in \Sigma$ if f has some rank $\hat{\tau}_1 \cdots \hat{\tau}_n \hat{\tau}$ in Σ, and $\tau_1 \cdots \tau_n \tau$ is an instance of $\hat{\tau}_1 \cdots \hat{\tau}_n \hat{\tau}$.

[1] Note that $Sort(\Sigma^\mathrm{S})$, the set of all sorts over Σ^S, is non-empty because at least one sort in Σ^S, **Bool**, has arity 0.
[2] A binary relation $R \subseteq X \times Y$ is *left-total* if for each $x \in X$ there is (at least) a $y \in Y$ such that xRy.

We will also consider signatures that differ from a given signature Σ only by the sort they assign to variables, as well as signatures that conservatively expand a given signature Σ with additional sort and function symbols or additional ranks for Σ's function symbols.

Definition 4 (Signature expansions)**.** A signature Ω is an *expansion* of a signature Σ if all of the following hold: $\Sigma^{S} \subseteq \Omega^{S}$; $\Sigma^{F} \subseteq \Omega^{F}$; the sort symbols of Σ have the same arity in Σ and in Ω; every sort of Σ has the same constructors in Ω that it has in Σ; every constructor of Σ has the same selectors and testers in Ω that it has in Σ; for all $x \in \mathcal{X}$ and $\tau \in \mathit{Sort}(\Sigma,\mathcal{U})$, $x{:}\tau \in \Sigma$ iff $x{:}\tau \in \Omega$; for all $f \in \mathcal{F}$ and $\overline{\tau} \in \mathit{Sort}(\Sigma,\mathcal{U})^{+}$, if $f{:}\overline{\tau} \in \Sigma$ then $f{:}\overline{\tau} \in \Omega$. In that case, Σ is a *subsignature* of Ω. □

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 *overloaded* in a signature Σ by being associated with more than one rank in Σ. 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).

A function symbol f can be *ambiguous* in an SMT-LIB signature Σ. That is the case if $f : \overline{\tau}\tau_1 \in \Sigma$ and $f : \overline{\tau}\tau_2 \in \Sigma$, where τ_1 and τ_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 like f above, a term of the form $f\,\bar{t}$, where the terms $\bar{t}$ have sorts $\overline{\tau}$, can be given a unique sort only if f is annotated with one of the result sorts τ_1, τ_2, that is, only if it is written as $f^{\tau_1}\bar{t}$ or $f^{\tau_2}\bar{t}$. As a consequence, from now on *we will assume that all ambiguous symbols are annotated as described above*.

5.2.2 Well-sorted terms

Figure 5.3 provides a set of rules defining well-sorted terms with respect to an SMT-LIB signature Σ. Strictly speaking then, and similarly to more conventional logics, the SMT-LIB logic language is a family of languages parametrized by the signature Σ. As explained later, for each script working in the context of a background theory $\mathcal{T}$, the specific signature is jointly 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 τ (possibly primed or with subscripts) denotes sorts in $\mathit{Sort}(\Sigma,\mathcal{U})$, the letter k denotes a natural number. Expressions of the form $\Sigma[x_1 : \tau_1, \ldots, x_n : \tau_n]$ denotes the signature that maps x_i to sort τ_i for $i = 1, \ldots, n$, and coincides otherwise with Σ. The rules operate over *sorting judgments*, which are triples of the form $\Sigma \vdash t : \tau$.

$$\frac{}{\Sigma \vdash x : \tau} \quad \text{if } x{:}\tau \in \Sigma$$

$$\frac{\Sigma \vdash t_1 : \tau_1 \quad \cdots \quad \Sigma \vdash t_k : \tau_k}{\Sigma \vdash (f\ t_1\ \cdots\ t_k) : \tau} \quad \text{if } \begin{cases} f{:}\tau_1 \cdots \tau_k \tau \in \Sigma, \\ f{:}\tau_1 \cdots \tau_k \tau' \notin \Sigma \ \text{ for all } \tau' \neq \tau \end{cases}$$

$$\frac{\Sigma[x : \tau_1] \vdash t_2 : \tau_2}{\Sigma \vdash (\lambda(x{:}\tau_1)\ t_2) : \tau_1 \to \tau_2}$$

$$\frac{\Sigma \vdash t_1 : \tau_1 \quad \cdots \quad \Sigma \vdash t_k : \tau_k}{\Sigma \vdash (f^{\tau}\ t_1\ \cdots\ t_k)\ : \tau} \quad \text{if } f{:}\tau_1 \cdots \tau_k \tau \in \Sigma$$

$$\frac{\Sigma[x{:}\tau]\ \vdash t : \mathbf{Bool}}{\Sigma \vdash (Qx{:}\tau\ t)\ : \mathbf{Bool}} \quad \text{if } Q \in \{\exists, \forall\}$$

$$\frac{\begin{array}{c}\Sigma \vdash t_1 : \tau_1 \quad \cdots \quad \Sigma \vdash t_{k+1} : \tau_{k+1} \\ \Sigma[x_1{:}\tau_1, \ldots, x_{k+1}{:}\tau_{k+1}]\ \vdash t : \tau\end{array}}{\Sigma \vdash (\mathbf{let}\ x_1 = t_1\ \cdots\ x_{k+1} = t_{k+1}\ \mathbf{in}\ t)\ : \tau} \quad \text{if } x_1, \ldots, x_{k+1} \text{ are all distinct}$$

$$\frac{\begin{array}{c}\Sigma \vdash t : \delta \\ \Sigma[\bar{x}_i{:}\bar{\sigma}_i]\ \vdash t_i : \tau \text{ for } i = 1, \ldots, k+1\end{array}}{\Sigma \vdash (\mathbf{match}\ t\ \mathbf{with}\ c_1\ \bar{x}_1 \to t_1\ \cdots\ c_{k+1}\ \bar{x}_{k+1} \to t_{k+1})\ : \tau} \quad \text{if } \begin{cases} \{c_1, \ldots, c_{k+1}\} = \mathrm{con}_{\Sigma}(\delta), \\ \text{for all } i = 1, \ldots, k+1 \\ \quad c_i{:}\bar{\sigma}_i\delta \in \Sigma \text{ and} \\ \quad \bar{x}_i \text{ contains no repetitions} \end{cases}$$

$$\frac{\begin{array}{c}\Sigma \vdash t : \delta \\ \Sigma[\bar{x}_i{:}\bar{\tau}_i]\ \vdash t_i : \tau \text{ for } i = 1, \ldots, k \\ \Sigma[x_i{:}\delta]\ \vdash t_i : \tau \text{ for } i = k+1, \ldots, n\end{array}}{\Sigma \vdash (\mathbf{match}\ t\ \mathbf{with}\ p_1 \to t_1\ \cdots\ p_n \to t_n)\ : \tau} \quad \text{if } \begin{cases} \{c_1, \ldots, c_k\} \subseteq \mathrm{con}_{\Sigma}(\delta) \neq \emptyset, \\ \{p_1, \ldots, p_n\} = \{c_1\ \bar{x}_1, \ldots, c_k\ \bar{x}_k\} \cup \\ \qquad\qquad \{x_{k+1}, \ldots, x_n\}, \\ k < n, \\ \text{for all } i = 1, \ldots, k \\ \quad c_i{:}\bar{\tau}_i\delta \in \Sigma \text{ and} \\ \quad \bar{x}_i \text{ contains no repetitions} \end{cases}$$

Figure 5.3: Well-sortedness rules for terms.

Definition 5 (Well-sorted Terms)**.** For every SMT-LIB signature Σ, a term t generated by the grammar in Figure 5.2 is *well-sorted (with respect to Σ)* if $\Sigma \vdash t : \tau$ is derivable by the sorting rules in Figure 5.3 for some sort $\tau \in Sort(\Sigma,\mathcal{U})$. In that case, we say that *t has, or is of, sort τ*. □

With this definition, it is possible to show that every term has at most one sort in a given signature Σ.(61) This means that we can define a total function *sort* that associates with each well-sorted term t its sort $sort(t, \Sigma)$. We can also define a function *pars* that associates to each well-sorted term t the set $pars(t, \Sigma)$ of all parameters that occur in its sort or in that of one of its subterms.

Remark 12 (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 δ 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 δ is a datatype, not just any sort, by requiring it to have a non-empty set of constructors.

Definition 6 (SMT-LIB formulas)**.** For each signature Σ, the language of SMT-LIB logic is the set of all well-sorted terms wrt Σ. *Formulas* are well-sorted terms of sort **Bool**. □

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

Constraint 3. SMT-LIB scripts consider only closed formulas, or *sentences*, i.e., closed terms of sort **Bool**.(62) □

There is no loss of generality in the restriction above because, as far as satisfiability is concerned, every formula φ with free variables $x_1, \ldots, x_n$ of respective sort $\tau_1, \ldots, \tau_n$, can be rewritten as

$$\exists\, x_1{:}\tau_1 (\ldots (\exists\, x_n{:}\tau_n\ \varphi\ ) \ldots).$$

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

5.3 Structures and Satisfiability

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

Definition 7 (Σ-structure)**.** Let Σ be a signature. A *Σ-structure* **A** is a pair consisting of a sufficently large set A, the *universe* of **A**, and a mapping that

- interprets each monomorphic sort $\sigma \in Sort(\Sigma)$ as a *non-empty* subset $\sigma^{\mathbf{A}}$ of A, which we call the *domain* of σ in $\mathbf{A}$, with $A = \bigcup_{\sigma \in Sort(\Sigma)} \sigma^{\mathbf{A}}$;[63]
- associates an element $(f{:}\sigma)^{\mathbf{A}}$ of the set $\sigma^{\mathbf{A}}$ with each (ranked function symbol) $f{:}\sigma \in \Sigma$, where σ is a monomorphic sort;
- associates a total function $(f{:}\sigma_1 \cdots \sigma_n \sigma)^{\mathbf{A}}$ from $\sigma_1^{\mathbf{A}} \times \cdots \times \sigma_n^{\mathbf{A}}$ to $\sigma^{\mathbf{A}}$ with each function symbol $f{:}\sigma_1 \cdots \sigma_n \sigma \in \Sigma$ with $n > 0$, where $\sigma_1, \ldots, \sigma_n, \sigma$ are monomorphic sorts.

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

Note that, as a consequence of overloading, a Σ-structure does not interpret plain function symbols but ranked function symbols.

Definition 8 (Absolutely free structure)**.** Let $\mathbf{A}$ be a Σ-structure with universe A and let $G \subseteq A$. Let Σ_G be the expansion of Σ obtained by adding to Σ a constant symbol c_a of sort σ for every $a \in G$ and monomorphic sort $\sigma \in \Sigma^{S}$ such that $a \in \sigma^{\mathbf{A}}$. Then, $\mathbf{A}$ is an *absolutely free structure (with generators G)* if

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

Intuitively, an absolutely free Σ-structure with a set G of generators interprets every well-sorted ground Σ_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.

For interpreting functional sorts, we will use the notion of a *map*, i.e., a functional binary relation. We will denote a map as a set of pairs of the form $a \mapsto b$.

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

Definition 9 (SMT-LIB Σ-structure)**.** An SMT-LIB structure is a Σ-structure $\mathbf{A}$ such that

1. $\mathbf{Bool}^{\mathbf{A}} = \mathcal{B} = \{\mathbf{false}, \mathbf{true}\}$ with **false** and **true** distinct;
2. for all monomorphic sorts σ_1 and σ_2, $(\sigma_1 \rightarrow \sigma_2)^{\mathbf{A}}$ is the set of all total maps from $\sigma_1^{\mathbf{A}}$ to $\sigma_2^{\mathbf{A}}$;
3. if Ω is the signature obtained from Σ by removing all of its non-constructor function symbols, the Ω-reduct of $\mathbf{A}$ is an absolutely free algebra with generators $\bigcup_{\sigma \in S} \sigma^{\mathbf{A}}$ where S collects the sorts of $Sort(\Sigma)$ that are not datatypes;

4. for all constructors $c{:}\sigma_1 \cdots \sigma_n\sigma \in \Sigma$ with $n > 0$, selectors $g_1{:}\sigma\sigma_1, \ldots, g_n{:}\sigma\sigma_n$ with $\mathrm{sel}_\Sigma(c) = g_1 \cdots g_n$, values $(v_1, \ldots, v_n) \in \sigma_1^{\mathbf{A}} \times \cdots \times \sigma_n^{\mathbf{A}}$, and $i = 1, \ldots, n$,

$$(g_i{:}\sigma\sigma_i)^{\mathbf{A}}((c{:}\sigma_1 \cdots \sigma_n\sigma)^{\mathbf{A}}(v_1, \ldots, v_n)) = v_i \;;$$

5. for all constructors $c{:}\bar{\sigma}\sigma \in \Sigma$, testers q with $\mathrm{tes}_\Sigma(c) = q$, and values $v \in \sigma^{\mathbf{A}}$, $q^{\mathbf{A}}(v) = \mathbf{true}$ iff v is in the range of $(c{:}\bar{\sigma}\sigma)^{\mathbf{A}}$.

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

Remark 13. 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, as discussed in Section 4.2.3, guarantee that sorts with constructors indeed denote algebraic datatypes as traditionally understood in the literature.

Remark 14 (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 $\mathbf{A}$. This means in particular that if $g{:}\tau\tau_i \in \Sigma$ is a selector, the function $g^{\mathbf{A}}$ returns a value even for inputs outside the range of g's constructor. Definition 9 imposes no constraints on that value, other than it belongs to $\tau_i^{\mathbf{A}}$. For instance, in a structure $\mathbf{A}$ with a sort for integer lists with constructors nil and insert and selectors head and tail for insert, the function $\mathrm{head}^{\mathbf{A}}$ maps $\mathrm{nil}^{\mathbf{A}}$ to *some* integer value. Similarly, $\mathrm{tail}^{\mathbf{A}}$ maps $\mathrm{nil}^{\mathbf{A}}$ 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 13, Theory Combination, in Section 5.4.

Definition 10 (Isomorphism)**.** Let $\mathbf{A}$ and $\mathbf{B}$ be two Σ-structures with respective universes A and B. A mapping $h : A \to B$ is a *homomorphism* from $\mathbf{A}$ to $\mathbf{B}$ if

1. for all monomorphic sorts $\sigma \in \mathit{Sort}(\Sigma)$ and $a \in \sigma^{\mathbf{A}}$,

$$h(a) \in \sigma^{\mathbf{B}} \;;$$

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

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

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

Two Σ-structures $\mathbf{A}$ and $\mathbf{B}$ are *isomorphic* if there is an isomorphism from one onto the other. Isomorphic structures are interchangeable for satisfiability purposes because one satisfies a 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 $\mathbf{A}$ is a partial mapping v from $\mathcal{X} \times Sort(\Sigma)$ to the set of all domain elements of $\mathbf{A}$ such that, for all $x \in \mathcal{X}$ and $\sigma \in Sort(\Sigma)$, $v(x{:}\sigma) \in \sigma^{\mathbf{A}}$. We denote by $v[x_1{:}\sigma_1 \mapsto a_1, \ldots, x_n{:}\sigma_n \mapsto a_n]$ the valuation that maps $x_i{:}\sigma_i$ to $a_i \in \sigma_i^{\mathbf{A}}$ for $i = 1, \ldots, n$ and is otherwise identical to v.

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

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

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

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

Definition 11 (Interpretation of terms)**.** Let Σ be an SMT-LIB signature and let $\mathcal{I}$ be a Σ-interpretation. For every well-sorted term t of sort τ with respect to Σ, $[\![t]\!]^{\mathcal{I}}$ is defined recursively as follows, where Rule 1 applies to polymorphic terms and the rest of the rules apply only to monomorphic terms.

1. $[\![t]\!]^{\mathcal{I}} = \textbf{true}$ iff $[\![\theta(t)]\!]^{\mathcal{I}} = \textbf{true}$ for all θ such that $\begin{cases} \theta = \{u_1 \mapsto \sigma_1, \ldots, u_n \mapsto \sigma_n\}, \\ \{u_1, \ldots u_n\} = pars(t, \Sigma),\ n > 0 \\ \sigma_1, \ldots, \sigma_n \in Sort(\Sigma) \end{cases}$

2. $[\![x]\!]^{\mathcal{I}} = v(x{:}\sigma)$ if $\mathcal{I} = (\mathbf{A}, v)$ and v is of sort σ

3. $[\![\hat{f}\ t_1\ \ldots\ t_n]\!]^{\mathcal{I}} = (f{:}\sigma_1 \cdots \sigma_n\sigma)^{\mathbf{A}}(a_1, \ldots, a_n)$ if $\begin{cases} \mathcal{I} = (\mathbf{A}, v) \text{ with signature } \Sigma, \\ \hat{f} = f \text{ or } \hat{f} = f^{\sigma}, \\ f \text{ is of rank } \sigma_1 \cdots \sigma_n\sigma \end{cases}$

4. $[\![\textbf{let}\ x_1 = t_1\ \cdots\ x_n = t_n\ \textbf{in}\ t]\!]^{\mathcal{I}} = [\![t]\!]^{\mathcal{I}'}$ if $\begin{cases} t_i \text{ has sort } \sigma_i \text{ and } a_i = [\![t_i]\!]^{\mathcal{I}} \text{ for } i = 1, \ldots, n, \\ \text{and } \mathcal{I}' = \mathcal{I}[x_1{:}\sigma_1 \mapsto a_1, \ldots, x_n{:}\sigma_n \mapsto a_n] \end{cases}$

5. $[\![\exists\, x_1{:}\sigma_1 \cdots x_n{:}\sigma_n\ t]\!]^{\mathcal{I}} = \textbf{true}$ iff $[\![t]\!]^{\mathcal{I}'} = \textbf{true}$ for some $\mathcal{I}'$

   such that $\begin{cases} \mathcal{I}' = \mathcal{I}[x_1{:}\sigma_1 \mapsto a_1, \ldots, x_n{:}\sigma_n \mapsto a_n], \\ (a_1, \ldots, a_n) \in \sigma_1^{\mathbf{A}} \times \cdots \times \sigma_n^{\mathbf{A}}, \\ \mathcal{I} = (\mathbf{A}, v), \end{cases}$

6. $[\![\forall\, x_1{:}\sigma_1 \cdots x_n{:}\sigma_n\ t]\!]^{\mathcal{I}} = \textbf{true}$ iff $[\![t]\!]^{\mathcal{I}'} = \textbf{true}$ for all $\mathcal{I}'$

   such that $\begin{cases} \mathcal{I}' = \mathcal{I}[x_1{:}\sigma_1 \mapsto a_1, \ldots, x_n{:}\sigma_n \mapsto a_n], \\ (a_1, \ldots, a_n) \in \sigma_1^{\mathbf{A}} \times \cdots \times \sigma_n^{\mathbf{A}}, \\ \mathcal{I} = (\mathbf{A}, v), \end{cases}$

7. $[\![\lambda(x{:}\sigma)\ t]\!]^{\mathcal{I}} = \{a \mapsto [\![t]\!]^{\mathcal{I}[x:\sigma \mapsto a]} \mid a \in \sigma^{\mathbf{A}}\}$ where $\mathcal{I} = (\mathbf{A}, v)$

8. $[\![\textbf{match}\ t\ \textbf{with}\ c \to t_0\ \ p_1 \to t_1\ \cdots\ p_n \to t_n]\!]^{\mathcal{I}} = [\![t_0]\!]^{\mathcal{I}}$

$$\text{if } \begin{cases} \mathcal{I} = (\mathbf{A}, v), \\ [\![t]\!]^{\mathcal{I}} \text{ is in the range of } c^{\mathbf{A}} \end{cases}$$

9. $[\![\textbf{match}\ t\ \textbf{with}\ (c\, x_1 \cdots x_{k+1}) \to t_0\ \ p_1 \to t_1\ \cdots\ p_n \to t_n]\!]^{\mathcal{I}} =$
$[\![\textbf{let}\ x_1 = g_1(t)\ \cdots\ x_{k+1} = g_{k+1}(t)\ \textbf{in}\ t_0]\!]^{\mathcal{I}}$

$$\text{if } \begin{cases} \mathcal{I} = (\mathbf{A}, v) \text{ with signature } \Sigma, \\ [\![t]\!]^{\mathcal{I}} \text{ is in the range of } c^{\mathbf{A}}, \\ \text{sel}_\Sigma(c) = g_1 \cdots g_{k+1} \end{cases}$$

10. $[\![\textbf{match}\ t\ \textbf{with}\ (c\, \bar{x}) \to t_0\ \ p_1 \to t_1\ \cdots\ p_n \to t_n]\!]^{\mathcal{I}} =$
$[\![\textbf{match}\ t\ \textbf{with}\ \ p_1 \to t_1\ \cdots\ p_n \to t_n\ (c\, \bar{x}) \to t_0]\!]^{\mathcal{I}}$

$$\text{if } \begin{cases} \mathcal{I} = (\mathbf{A}, v), \\ [\![t]\!]^{\mathcal{I}} \text{ is not in the range of } c^{\mathbf{A}} \end{cases}$$

11. $[\![\textbf{match}\ t\ \textbf{with}\ x_0 \to t_0\ \ p_1 \to t_1\ \cdots\ p_n \to t_n]\!]^{\mathcal{I}} = [\![\textbf{let}\ x_0 = t\ \textbf{in}\ t_0]\!]^{\mathcal{I}}$ □

Remark 15. There is no rule above for higher-order function application _, since _ is a normal (interpreted) function symbol defined in the HO-Core theory (see Figure 3.6).

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

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

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

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

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

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 Σ_1 and Σ_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.[3] Two theories are *compatible* if they have compatible signatures. The *combination* $\Sigma_1 + \Sigma_2$ of two compatible signatures Σ_1 and Σ_2 is the smallest compatible signature that is an expansion of both Σ_1 and Σ_2, i.e., the unique signature Σ compatible with Σ_1 and Σ_2 such that, for all $f \in \mathcal{F}$ and $\overline{\tau} \in Sort(\Sigma)^+$, $f{:}\overline{\tau} \in \Sigma$ iff $f{:}\overline{\tau} \in \Sigma_1$ or $f{:}\overline{\tau} \in \Sigma_2$.

Definition 13 (Theory Combination)**.** Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two theories with compatible signatures Σ_1 and Σ_2, respectively. The *combination* $\mathcal{T}_1 + \mathcal{T}_2$ of $\mathcal{T}_1$ and $\mathcal{T}_2$ consists of all $(\Sigma_1 + \Sigma_2)$-structures whose reduct to Σ_i is isomorphic to a model of $\mathcal{T}_i$, for $i = 1, 2$. □

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

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

Definition 14 (Satisfiability and Entailment Modulo a Theory)**.** For any Σ-theory $\mathcal{T}$, a Σ-sentence is *satisfiable in $\mathcal{T}$* iff it is satisfied by one of $\mathcal{T}$'s models. A set Γ of Σ-sentences *$\mathcal{T}$-entails* a Σ-sentence φ, written $\Gamma \models_{\mathcal{T}} \varphi$, iff every model of $\mathcal{T}$ that satisfies all sentences in Γ satisfies φ as well.

5.4.2 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 concrete syntax allows for further theory attributes, and sort and function declarations can also have attributes. Since they have no logical meaning, similar to term attributes, these are simplified away in the abstract syntax. Note that function symbol declarations define polymorphic functions when τ contains sort parameters.[4]

To simplify the meta-notation let T denote a theory declaration with theory name T. Given such a theory declaration, assume first that T has no **sorts-description** and **funs-description** attributes, and let S and F be respectively the set of all sort symbols and all function symbols occurring in T. Let Ω be a signature *whose set of sort symbols that are not datatypes includes all*

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

[4] Recall that sort parameters are introduced in the concrete syntax of theory declarations with the binder `par`.

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

Figure 5.4: Abstract syntax for theory declarations

the symbols in S, with the same arity. The definition provided in the **definition** attribute of T must be such that every signature like Ω above uniquely determines a theory $\widehat{T} = T[\Omega]$ as an instance of T with signature $\widehat{\Omega}$ defined as follows:

1. $\widehat{\Omega}^{\mathrm{S}} = \Omega^{\mathrm{S}}$ and $\widehat{\Omega}^{\mathrm{F}} = F \cup \Omega^{\mathrm{F}}$,
2. no variables are sorted in $\widehat{\Omega}$,(64)
3. for all $f \in \widehat{\Omega}^{\mathrm{F}}$ and $\bar{\sigma} \in (Sort(\widehat{\Omega}))^+$, $f{:}\bar{\sigma} \in \widehat{\Omega}$ iff
   (a) $f{:}\bar{\sigma} \in \Omega$, or
   (b) T contains a declaration of the form $f\ \bar{\tau}$ and $\bar{\sigma}$ is an instance of $\bar{\tau}$, with $\bar{\tau} \in (Sort(\Omega,\mathcal{U}))^+$.
4. A function symbol of $\widehat{\Omega}$ is a datatype constructor/selector/tester in $\widehat{\Omega}$ iff it is so in Ω.

We say that a ranked function symbol $f{:}\bar{\sigma}$ of $\widehat{\Omega}$ is *declared in T* if $f{:}\bar{\sigma} \in \widehat{\Omega}$ because of Point 3b above. The *free sort symbols* of $\widehat{T}$ are the sort symbols of $\widehat{\Omega}$ that are not in S and are not datatypes. Similarly, the *free function symbols* of $\widehat{T}$ are the ranked function symbols of $\widehat{\Omega}$ that are not declared in T and are not datatype constructors, selectors or testers.[5] 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.(65)

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

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

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

Figure 5.5: Abstract syntax for logic declarations

- fixing a signature Σ and a Σ-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 Σ-sentences.

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

5.5.1 Logic declarations

Logics are specified by means of logic declarations. An abstract syntax of logic declarations is defined in Figure 5.5. Contrary to the theory declarations, a logic declaration specifies a *single* logic, not a class of them, so we call the logic L too. Again, the concrete syntax allows additional logic attributes that do not impact the semantics and can be ignored here. Logics allow extensions, defining further function symbols. For the sake of simplicity, we only consider here logics without extensions, since those additional symbols could be considered to be defined in the logic's theories.

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

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

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

Note that $\mathcal{T}$ is well defined. To start, Ω is well defined because any sort symbols shared by two declarations among $T_1, \ldots, T_n$ have the same arity in them. The theories $T_1[\Omega], \ldots, T_n[\Omega]$ are well defined because Ω satisfies the requirements in Subsection 5.4.2. 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 σ of the logic's theory $\mathcal{T}$ a distinguished set V_σ of ground terms called *values*. The definition of V_σ should be such that every sentence satisfiable in the logic L is satisfiable in a model $\mathbf{A}$ of $\mathcal{T}$ where each element of $\sigma^{\mathbf{A}}$ is denoted by some element of V_σ. In other words, if Σ is $\mathcal{T}$'s signature, $\mathbf{A}$ is such that,

for all $\sigma \in Sort(\Sigma)$ and all $a \in \sigma^{\mathbf{A}}$, $a = [\![t]\!]^{\mathcal{A}}$ for some $t \in V_\sigma$. For example, in a logic of the integers, the set of values for the integer sort might consist of all the terms of the form 0 or $[-]n$ where n is a non-zero numeral.

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

Note that the requirements on V_σ can be always trivially satisfied by L by making sure that the signature Ω above contains a distinguished set of infinitely many additional free constant symbols of sort σ, and defining V_σ to be that set. We call these constant symbols *abstract values*. Abstract values are useful to denote the elements of uninterpreted sorts or sorts standing for structured datatypes such as lists, arrays, sets and so on.[6]

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 δ with the usual nil and cons constructors, the set of values for δ consists of nil and all terms of the form cons v_1 (cons v_2 $\cdots$ (cons v_n nil) $\cdots$) where $n > 0$ and $v_1, \ldots, v_n$ are (recursively) values of the same sort.

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

Part IV

References

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

[NS24] Michael Norrish and Konrad Slind. The HOL system LOGIC. Technical report, 2024. Available at `https://sourceforge.net/projects/hol`.

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

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

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

Part V

Appendices

APPENDIX A

Notes

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

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

3 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.

4 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.

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

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

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

8 Symbols were added as indices in Version 2.6 for increased flexibility.

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

10 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.

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

12 This is to avoid the situation where a typo while attempting to write a nullary constructor introduces a variable pattern, thus changing the meaning of the match term without warning or error.

13 The restriction to flat patterns is for simplicity and may be lifted in later versions.

14 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.

15 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 `as` is not required in patterns because there sort inference is needed anyway since pattern variables are not explicitly assigned a sort.

16 The reason patterns annotate the *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.

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

18 This shadowing rule follows common lexical scoping conventions. It is easy to avoid declarations like `(par (A) (f (Array A B) A))`, which are potentially confusing to human readers if `A` is a sort symbol, simply by a more judicious choice of parameter names.

19 See the point made in Note 20.

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

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

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

23 This difference will disappear in the planned Version 3 of the standard whose underlying logic will change from many-sorted logic to a version of higher-order logic with dependent types.

24 The difference between function symbols and constants of `->` sorts will disappear in Version 3 where, in current terms, there will be no function symbols but only constant symbols, possibly of sort `(->` τ_1 τ_2`)`.

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

26 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.

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

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

29 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.

30 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.

31 Setting `:global-declarations` to `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.

32 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.

33 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.

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

35 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.

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

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

38 Having `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 `ALL` in set-logic but should instead use a specific logic name.

39 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.

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

41 Using local sort parameters as opposed to globally declared sort parameters is necessary because their order is meaningful, as arguments of the sort constructor.

42 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 δ_i is known to be a sort symbol of arity k_i.

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

44 The rationale for this restriction is the same as for function symbols introduced with `declare-fun` and `define-fun`, i.e., keep parsing and type checking simple.

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

46 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.

47 The motivation for having this command is that it corresponds to a common usage pattern with SMT solvers which can be implemented considerably more efficiently than the general stack-based mechanism needed to support `push` and `pop` commands.

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

49 Abstract values are useful for reporting model values in logics containing for example the theory of arrays (see Figure 3.3). For instance, a solver may specify the content of an array `a` of sort `(Array Int Int)` at positions 0–2 by returning the expression

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

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

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

51 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.

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

$$\texttt{(define-fun-rec f ((}x_1\ \sigma_1\texttt{)}\ \cdots\ \texttt{(}x_n\ \sigma_n\texttt{))}\ \sigma\ \texttt{(f}\ x_1\ \cdots\ x_n\texttt{))}$$

for a given f of rank $\sigma_1 \cdots \sigma_n \sigma$. Similarly, for constant symbols c of a sort σ that admits abstract values, they allow a solver to return `(define-fun c () @a)` with `@a` abstract.
The reason for such weak requirements is that stronger ones are currently difficult to achieve in general because of limitations in the expressive power of some SMT-LIB theories/logics or in the computational abilities of present SMT solvers. For instance, the current theory of arrays (see Figure 3.3) does not have enough *constructor* symbols to allow one to represent an array uniquely as a value term. As shown in a previous note, one can use terms like

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

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

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

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

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

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

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

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

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

56 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.

57 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.

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

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

60 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.

61 It would have been reasonable to adopt an alternative version of the rule for well-sortedness of terms $(f^\tau \ t_1 \ \cdots \ t_k)$ with annotated function symbols f^τ, without the second conjunct of the rule's side condition. This would allow formation of terms with annotated function symbols f^τ, even when f lacked two ranks of the forms $\tau_1 \cdots \tau_k \tau$ and $\tau_1 \cdots \tau_k \tau'$, for distinct τ and τ'. 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.

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

63 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.

64 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.

65 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.

APPENDIX B

Concrete Syntax

Predefined symbols

These symbols have a predefined meaning in Version 2.7. 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 2.7.

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

Auxiliary Lexical Categories

⟨*white_space_char*⟩	::=	9_{dec} | 10_{dec} | 13_{dec} | 32_{dec}
⟨*printable_char*⟩	::=	32_{dec} | ··· | 126_{dec} | 128_{dec} | ··· | 255_{dec}
⟨*digit*⟩	::=	`0` | ··· | `9`
⟨*letter*⟩	::=	`A` | ··· | `Z` | `a` | ··· | `z`

Tokens

Reserved Words

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

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

Other tokens

`(`
`)`

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

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

Attributes

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

⟨attribute⟩ ::= ⟨keyword⟩ | ⟨keyword⟩ ⟨attribute_value⟩

Terms

⟨qual_identifier⟩ ::= ⟨identifier⟩ | (as ⟨identifier⟩ ⟨sort⟩)

⟨var_binding⟩ ::= (⟨symbol⟩ ⟨term⟩)

⟨sorted_var⟩ ::= (⟨symbol⟩ ⟨sort⟩)

⟨pattern⟩ ::= ⟨symbol⟩ | (⟨symbol⟩ ⟨symbol⟩$^+$)

⟨match_case⟩ ::= (⟨pattern⟩ ⟨term⟩)

⟨term⟩ ::= ⟨spec_constant⟩
| ⟨qual_identifier⟩
| (⟨qual_identifier⟩ ⟨term⟩$^+$)
| (let (⟨var_binding⟩$^+$) ⟨term⟩)
| (lambda (⟨sorted_var⟩$^+$) ⟨term⟩)
| (forall (⟨sorted_var⟩$^+$) ⟨term⟩)
| (exists (⟨sorted_var⟩$^+$) ⟨term⟩)
| (match ⟨term⟩ (⟨match_case⟩$^+$))
| (! ⟨term⟩ ⟨attribute⟩$^+$)

Theories

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

Logics

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

Info flags

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

Command options

⟨*b_value*⟩	::=	`true` | `false`
⟨*option*⟩	::=	`:diagnostic-output-channel` ⟨*string*⟩
	|	`:global-declarations` ⟨*b_value*⟩
	|	`:interactive-mode` ⟨*b_value*⟩
	|	`:print-success` ⟨*b_value*⟩
	|	`:produce-assertions` ⟨*b_value*⟩
	|	`:produce-assignments` ⟨*b_value*⟩
	|	`:produce-models` ⟨*b_value*⟩
	|	`:produce-proofs` ⟨*b_value*⟩
	|	`:produce-unsat-assumptions` ⟨*b_value*⟩
	|	`:produce-unsat-cores` ⟨*b_value*⟩
	|	`:random-seed` ⟨*numeral*⟩
	|	`:regular-output-channel` ⟨*string*⟩
	|	`:reproducible-resource-limit` ⟨*numeral*⟩
	|	`:verbosity` ⟨*numeral*⟩
	|	⟨*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*⟩)
| (`declare-sort-parameter` ⟨*symbol*⟩)
| (`define-const` ⟨*symbol*⟩ ⟨*sort*⟩ ⟨*term*⟩)
| (`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

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

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

⟨model_response⟩ ::= (define-fun ⟨function_def⟩) | (define-fun-rec ⟨function_def⟩)
| (define-funs-rec (⟨function_dec⟩$^{n+1}$) (⟨term⟩$^{n+1}$))

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

⟨valuation_pair⟩ ::= (⟨term⟩ ⟨term⟩)

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

⟨check_sat_response⟩ ::= sat | unsat | unknown

⟨echo_response⟩ ::= ⟨string⟩

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

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

⟨get_info_response⟩ ::= (⟨info_response⟩+)

⟨get_model_response⟩ ::= (⟨model_response⟩*)

⟨get_option_response⟩ ::= ⟨attribute_value⟩

⟨get_proof_response⟩ ::= ⟨s_expr⟩

⟨get_unsat_assump_response⟩ ::= (⟨term⟩*)

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

⟨get_value_response⟩ ::= (⟨valuation_pair⟩+)

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

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

APPENDIX C

Abstract Syntax

Common Notation

$b \in \mathcal{B}$,	the set of boolean values		
$n \in \mathcal{N}$,	the set of natural numbers	$w \in \mathcal{W}$,	the set of character strings
$s \in \mathcal{S}$,	the set of sort symbols	$u \in \mathcal{U}$,	the set of sort parameters
$f \in \mathcal{F}$,	the set of function symbols	$x \in \mathcal{X}$,	the set of variables
$T \in \mathcal{TN}$,	the set of theory names	$L \in \mathcal{L}$,	the set of logic names

Sorts

(Monomorphic Sorts) $\sigma \quad ::= \quad s\,\sigma^*$

(Polymorphic Sorts) $\tau \quad ::= \quad u \;\mid\; s\,\tau^*$

Terms

(Patterns) $p \quad ::= \quad x \;\mid\; f\,x^*$

(Terms) $t \quad ::= \quad x \;\mid\; f\,t^* \;\mid\; f^\tau\,t^* \;\mid\; \lambda\,(x{:}\tau)\;t \;\mid\; \exists\,(x{:}\tau)\;t \;\mid\; \forall\,(x{:}\tau)\;t$
$\quad\mid\quad \textbf{let}\;(x = t)^+\;\textbf{in}\;t \;\mid\; \textbf{match}\;t\;\textbf{with}\;(p \to t)^+$

Well-sorting rules for terms

$f{:}\tau_1 \cdots \tau_n \tau \in \Sigma$ iff f has some rank $\hat{\tau}_1 \cdots \hat{\tau}_n \hat{\tau}$ in Σ, and $\tau_1 \cdots \tau_n \tau$ is an instance of $\hat{\tau}_1 \cdots \hat{\tau}_n \hat{\tau}$

$$\frac{}{\Sigma \vdash x : \tau} \quad \text{if } x{:}\tau \in \Sigma$$

$$\frac{\Sigma \vdash t_1 : \tau_1 \quad \cdots \quad \Sigma \vdash t_k : \tau_k}{\Sigma \vdash (f\ t_1\ \cdots\ t_k) : \tau} \quad \text{if } \begin{cases} f{:}\tau_1 \cdots \tau_k \tau \in \Sigma, \\ f{:}\tau_1 \cdots \tau_k \tau' \notin \Sigma \ \ \text{for all } \tau' \neq \tau \end{cases}$$

$$\frac{\Sigma[x : \tau_1] \vdash t_2 : \tau_2}{\Sigma \vdash (\lambda(x{:}\tau_1)\ t_2) : \tau_1 \to \tau_2}$$

$$\frac{\Sigma \vdash t_1 : \tau_1 \quad \cdots \quad \Sigma \vdash t_k : \tau_k}{\Sigma \vdash (f^{\tau}\ t_1\ \cdots\ t_k)\ : \tau} \quad \text{if } f{:}\tau_1 \cdots \tau_k \tau \in \Sigma$$

$$\frac{\Sigma[x{:}\tau]\ \vdash t : \mathbf{Bool}}{\Sigma \vdash (Qx{:}\tau\ t)\ : \mathbf{Bool}} \quad \text{if } Q\ \in \{\exists, \forall\}$$

$$\frac{\begin{array}{c}\Sigma \vdash t_1 : \tau_1 \quad \cdots \quad \Sigma \vdash t_{k+1} : \tau_{k+1} \\ \Sigma[x_1{:}\tau_1,\ \ldots,\ x_{k+1}{:}\tau_{k+1}]\ \vdash t : \tau\end{array}}{\Sigma \vdash (\mathbf{let}\ x_1 = t_1\ \cdots\ x_{k+1} = t_{k+1}\ \mathbf{in}\ t)\ : \tau} \quad \text{if } x_1, \ldots, x_{k+1} \text{ are all distinct}$$

$$\frac{\begin{array}{c}\Sigma \vdash t : \delta \\ \Sigma[\bar{x}_i{:}\bar{\tau}_i]\ \vdash t_i : \tau \text{ for } i = 1, \ldots, k+1\end{array}}{\Sigma \vdash (\mathbf{match}\ t\ \mathbf{with}\ c_1\ \bar{x}_1 \to t_1\ \cdots\ c_{k+1}\ \bar{x}_{k+1} \to t_{k+1})\ : \tau} \quad \text{if } \begin{cases} \{c_1, \ldots, c_{k+1}\} = \mathrm{con}_\Sigma(\delta), \\ \text{for all } i = 1, \ldots, k+1 \\ \quad c_i{:}\bar{\sigma}_i\delta \in \Sigma \text{ and} \\ \quad \bar{x}_i \text{ contains no repetitions} \end{cases}$$

$$\frac{\begin{array}{c}\Sigma \vdash t : \delta \\ \Sigma[\bar{x}_i{:}\bar{\tau}_i]\ \vdash t_i : \tau \text{ for } i = 1, \ldots, k \\ \Sigma[x_i{:}\delta]\ \vdash t_i : \tau \text{ for } i = k+1, \ldots, n\end{array}}{\Sigma \vdash (\mathbf{match}\ t\ \mathbf{with}\ p_1 \to t_1\ \cdots\ p_n \to t_n)\ : \tau} \quad \text{if } \begin{cases} \{c_1, \ldots, c_k\} \subseteq \mathrm{con}_\Sigma(\delta) \neq \emptyset, \\ \{p_1, \ldots, p_n\} = \{c_1\ \bar{x}_1, \ldots, c_k\ \bar{x}_k\} \cup \\ \qquad\qquad\qquad \{x_{k+1}, \ldots, x_n\}, \\ k < n, \\ \text{for all } i = 1, \ldots, k \\ \quad c_i{:}\bar{\tau}_i\delta \in \Sigma \text{ and} \\ \quad \bar{x}_i \text{ contains no repetitions} \end{cases}$$

Theories

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

Logics

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

Index

ambiguous symbol, 79
attribute, 77

binder
  existential binder, 76
  lambda binder, 76
  let binder, 77
  universal binder, 76
bound, 77

closed term, 77

free, 77

ground term, 77

open term, 77
overloaded function symbol, 79

sentence, 81
SMT, 14
  solver, 17
SMT-LIB, 15
sort, 75
symbol
  function symbol, 75

theory
  basic, 18
  combined, 18

variable, 75