Using the TPTP Language for Writing Derivations and Finite Interpretations

Geoff Sutcliffe[1], Stephan Schulz[2], Koen Claessen[3], and Allen Van Gelder[4]

[1] University of Miami, USA
geoff@cs.miami.edu
[2] Technische Universität München, Germany
schulz@eprover.org
[3] Chalmers University of Technology, Sweden
koen@chalmers.se
[4] University of California at Santa Cruz, USA
avg@cs.ucsc.edu

Abstract. One of the keys to the success of the TPTP and related projects is their consistent use of the TPTP language. The ability of the TPTP language to express solutions as well as problems, in conjunction with the simplicity of the syntax, sets it apart from other languages used in ATP. This paper provides a complete definition of the TPTP language, and describes how the language should be used to write derivations and finite interpretations.

1 Introduction

The TPTP problem library [19] is a well known standard set of test problems for first order automated theorem proving (ATP) systems. The TSTP solution library [18], the "flip side" of the TPTP, is becoming known as a resource for contemporary ATP systems' solutions to TPTP problems. The SystemOnTPTP [16] and associated software have been employed in a range of application projects, e.g., [4,21,23]. One of the keys to the success of these projects is their consistent use of the TPTP language, which enables convenient communication between different systems and researchers.

TPTP v3.0.0 introduced a new version of the TPTP language [20]. The language was designed to be suitable for writing both ATP problems and ATP solutions, to be flexible and extensible, and easily processed by both humans and computers. The entry barrier for using the TPTP language is (and has always been) very low. The syntax shares many features with Prolog, a language that is widely known in the ATP community. Indeed, with a few operator definitions, units of TPTP data can be read in Prolog using a single `read`/1 call, and written with a single `writeq`/1 call. Development, or at least prototyping, of reasoning software in Prolog is common, and Prolog compatibility eliminates the mundane task of writing IO routines for the reasoning software.

The key development from the old (pre-v3.0.0) TPTP language to the new one was the addition of features for writing solutions to ATP problems. The features were designed for writing derivations, but their flexibility makes it possible to

U. Furbach and N. Shankar (Eds.): IJCAR 2006, LNAI 4130, pp. 67–81, 2006.

write a range of DAG structures. Additionally, there are features of the language that make it possible to conveniently specify finite interpretations. This paper provides a complete definition of the TPTP language, and describes how the language should be used to write derivations and finite interpretations.

The ability of the TPTP language to express solutions as well as problems, in conjunction with the simplicity of the syntax, sets it apart from other languages used in ATP. Some languages, e.g., the LOP format [13], were designed for writing problems, and do not support writing solutions. Some languages for writing solutions are limited in scope, e.g., the PCL language [5] is limited to solutions to to equational problems, and the OpenTheory language [8] is designed only to be a computer processible form for systems that implement the HOL logic [6]. There are some general purpose languages that have features for writing derivations, e.g., Otter's `proof_object` format [11,10] and the DFG syntax [7], but none of these (that we know of) also provide support for writing finite interpretations. Mark-up languages such as OmDoc [9], OpenMath [2], and MathML [2] are quite expressive (especially for mathematical content), but their XML based format is not suitable for human processing. Overall, the TPTP language is more expressive and usable than other languages. Interoperability with other languages is supported in some cases, through translation tools.

2 The TPTP Language

The new TPTP language was first used in TPTP v3.0.0, released in November 2004. It has been taken up by a number of developers and received valuable comments and feedback. As a consequence, since that first release there have been some small, but significant, changes and extensions. The BNF definition of the language has recently been thoroughly overhauled. A principal goal has been to make it easy to translate the BNF into `lex/yacc/flex/bison` input, so that construction of parsers (in languages other than Prolog) can be a reasonably easy task. The BNF definition is in the appendix of this paper.

The TPTP language definition uses a modified BNF meta-language that separates semantic, syntactic, lexical, and character-macro rules. Syntactic rules use the standard `::=` separator, e.g.,

```
<source>      ::= <general_term>
```

When only a subset of the syntactically acceptable values for a non-terminal make semantic sense, a second rule for the non-terminal is provided using a `:==` separator, e.g.,

```
<source>      :== <dag_source> | <internal_source> | , etc.
```

Any further semantic rules that may be reached only from the right hand side of a semantic rule are also written using the `:==` separator, e.g.,

```
<dag_source>  :== <name> | <inference_record>
```

This separation of syntax from semantics eases the task of building a syntactic analyzer, as only the `::=` rules need be considered. At the same time, the

semantic rules provide the detail necessary for semantic checking. The rules that produce tokens from the lexical level use a `::-` separator, e.g.,

```
<lower_word>  ::- <lower_alpha><alpha_numeric>*
```

with the bottom level character-macros defined by regular expressions in rules using a `:::` separator, e.g.,

```
<lower_alpha> ::: [a-z]
```

The BNF is documented with comments.

The top level building blocks of TPTP files are *annotated formulae*, *include directives*, and *comments*. An annotated formula has the form:

language(*name*, *role*, *formula*, *source*, [*useful_info*]).

The *language*s currently supported are `fof` - formulae in full first order form, and `cnf` - formulae in clause normal form. The *role* gives the user semantics of the *formula*, e.g., `axiom`, `lemma`, `conjecture`, and hence defines its use in an ATP system - see the BNF for the list of recognized roles and their meaning. The logical *formula*, in either FOF or CNF, uses a consistent and easily understood notation [20] that can be seen in the BNF. The *source* describes where the formula came from, e.g., an input file or an inference. The *useful_info* is a list of arbitrary useful information, as required for user applications. The *useful_info* field is optional, and if it is not used then the *source* field becomes optional. An example of a FOF formula, supplied from a file, is:

```
fof(formula_27,axiom,
    ! [X,Y] :
      ( subclass(X,Y) <=>
        ! [U] :
          ( member(U,X) => member(U,Y) )),
    file('SET005+0.ax',subclass_defn),
    [description('Definition of subclass'), relevance(0.9)]).
```

An example of an inferred CNF formula is:

```
cnf(175,lemma,
    ( rsymProp(ib,sk_c3)
    | sk_c4 = sk_c3 ),
    inference(factor_simp,[status(thm)],[
        inference(para_into,[status(thm)],[96,78,theory(equality)])]),
    [iquote('para_into,96.2.1,78.1.1,factor_simp')]).
```

A novel feature of the TPTP language, which is employed in the representation of finite interpretations, is the recognition of interpreted predicates and functors. These come in two varieties: *defined* predicates and functors, whose interpretation is specified by the TPTP language, and *system* predicates and functors, whose interpretation is ATP system specific. Interpreted predicates and functors are syntactically different from uninterpreted predicates and functors. Defined predicates and functors either start with a `$`, or are composed of non-alphanumeric characters. System predicates and functors start with `$$`. Uninterpreted predicates and functors start with a lower case alphabetic. The

defined predicates recognized so far are `$true` and `$false`, with the obvious interpretations, and `=` and `!=` for equality and inequality. The defined functors recognized so far are "distinct object"s, written in double quotes, and numbers. A "distinct object" is interpreted as the domain element in the double quotes. Numbers are interpreted as themselves (as domain elements). A consequence of the predefined interpretations is that all different "distinct object"s and numbers are known to be unequal, e.g., `"Apple" != "Microsoft"` and `1 != 2` are implicit axioms. Such implicit axioms may be built into an ATP system, e.g., [14], or generated. System predicates and functors are used for interpreted predicates and functors that are available in particular ATP tools. The names are not controlled by the TPTP language, so they must be used with caution.

The source field of an annotated formula is most commonly a `file` record or an `inference` record. A `file` record stores the name of the file from which the annotated formula was read, and optionally the name of the annotated formula as it occurs in the file (this may be different from the name of the annotated formula itself, e.g., if the ATP system renames the annotated formulae that it reads in). An `inference` record stores three items of information about an inferred formula: the name of the inference rule provided by the ATP system, i.e., there are no standards; a list of useful information items, e.g., the semantic `status` of the formula and with respect to its parents as an SZS ontology value [20] (commonly inferred formulae are theorems of their parents, but in some cases the semantic relationship is weaker, as in Skolemization steps); and a list of the parents, which most commonly are parent annotated formula names, nested `inference` records, and `theory` records. A theory record is used when the axioms of some theory are built into the inference rule, e.g., equality axioms are built into paramodulation.

The `include` directives of the TPTP language are analogous to C's `#include` directives. An `include` directive may include an entire file, or may specify the names of the annotated formulae that are to be included from the file, thus providing a more finely grained include mechanism.

Regular comments in the TPTP language extend from a `%` character to the end of the line, or may be block comments within `/* ...*/` bracketing. System comments in the TPTP language are used for system specific annotations. They extend from a `%$$` sequence to the end of the line, or may be block comments within `/*$$ ...*/` bracketing. System comments look like regular comments, so normally they would be discarded. However, a wily user of the language can store/extract information from the comment before discarding it. System comments should identify the ATP system, followed by a :, e.g., `/*$$Otter 3.3: Demodulator */`. Comments may occur between any two tokens.

Parsing tools written in C are available for the TPTP language, conversion of the BNF into `lex`/`yacc` input is available [22], and the `tptp2X` utility distributed with the TPTP is compatible with the language.

3 Derivations

A derivation is a directed acyclic graph (DAG) whose leaf nodes are formulae from the input, whose interior nodes are formulae inferred from parent formulae, and whose root nodes are the final derived formulae. For example, a proof of a FOF theorem from some axioms, by refutation of the CNF of the axioms and negated conjecture, is a derivation whose leaf nodes are the FOF axioms and conjecture, whose internal nodes are formed from the process of clausification and then from inferences performed on the clauses, and whose root node is the *false* formula.

The information required to record a derivation is, minimally, the leaf formulae, and each inferred formula with references to its parent formulae. More detailed information that may be recorded and useful includes: the role of each formula; the name of the inference rule used in each inference step; sufficient details of each inference step to deterministically reproduce the inference; and the semantic relationships of inferred formulae with respect to their parents. The TPTP language is sufficient for recording all this, and more. A comprehensively recorded derivation provides the information required for various forms of processing, such as proof verification [17], proof visualization [15], and lemma extraction [5].

A derivation written in the TPTP language is a list of annotated formulae. Each annotated formula has a name, a role, and the logical formula. Each inferred formula has an `inference` record with the inference rule name, the semantic relationship of the formula to its parents as an SZS ontology value in a `status` record, and a list of references to its parent formulae.

Example. Consider the following toy FOF problem, to prove the `conjecture` from the `axioms` (not all the axioms are needed for the proof - the extra axioms come into play when the example is used again in Section 4 to illustrate the finite interpretation format):

```
%------------------------------------------------------------------------
%----All (hu)men are created equal. John is a human. John got an F grade.
%----There is someone (a human) who got an A grade. An A grade is not
%----equal to an F grade. Grades are not human. Therefore there is a
%----human other than John.
fof(all_created_equal,axiom,(
    ! [H1,H2] : ( ( human(H1) & human(H2) ) => created_equal(H1,H2) ) )).
fof(john,axiom,(
    human(john) )).
fof(john_failed,axiom,(
    grade(john) = f )).
fof(someone_got_an_a,axiom,(
    ? [H] : ( human(H) & grade(H) = a ) )).
fof(distinct_grades,axiom,(
    a != f )).
fof(grades_not_human,axiom,(
    ! [G] : ~ human(grade(G)) )).
fof(someone_not_john,conjecture,(
    ? [H] : ( human(H) & H != john ) )).
%------------------------------------------------------------------------
```

Here is a derivation recording a proof by refutation of the CNF, adapted (removing inferences that simply copy the parent formula) from the one produced by the ATP system EP v0.91 [12]:

```
%-----------------------------------------------------------------------
fof(3,axiom,(
    grade(john) = f ),
    file('CreatedEqual.p',john_failed)).
fof(4,axiom,(
    ? [X3] : ( human(X3) & grade(X3) = a ) ),
    file('CreatedEqual.p',someone_got_an_a)).
fof(5,axiom,(
    a != f ),
    file('CreatedEqual.p',distinct_grades)).
fof(7,conjecture,(
    ? [X3] : ( human(X3) & X3 != john ) ),
    file('CreatedEqual.p',someone_not_john)).
fof(8,negated_conjecture,(
    ~ ? [X3] : ( human(X3) & X3 != john ) ),
    inference(assume_negation,[status(cth)],[7])).
cnf(14,plain,
    ( grade(john) = f ),
    inference(split_conjunct,[status(thm)],[3])).
fof(16,plain,
    ( human(esk1_0) & grade(esk1_0) = a ),
    inference(skolemize,[status(sab)],[4])).
cnf(17,plain,
    ( grade(esk1_0) = a ),
    inference(split_conjunct,[status(thm)],[16])).
cnf(18,plain,
    ( human(esk1_0) ),
    inference(split_conjunct,[status(thm)],[16])).
cnf(19,plain,
    ( a != f ),
    inference(split_conjunct,[status(thm)],[5])).
fof(22,negated_conjecture,(
    ! [X3] : ( ~ human(X3) | X3 = john ) ),
    inference(fof_nnf,[status(thm)],[8])).
cnf(24,negated_conjecture,
    ( X1 = john | ~ human(X1) ),
    inference(split_conjunct,[status(thm)],[22])).
cnf(25,negated_conjecture,
    ( john = esk1_0 ),
    inference(spm,[status(thm)],[24,18,theory(equality)])).
cnf(28,plain,
    ( f = a ),
    inference(rw,[status(thm)],[
        inference(rw,[status(thm)],[17,25,theory(equality)]),
        14,theory(equality)])).
cnf(29,plain,
    ( $false ),
    inference(sr,[status(thm)],[28,19,theory(equality)])).
%-----------------------------------------------------------------------
```

4 Finite Interpretations

A finite interpretation (or "finite model" of some identified formulae) consists of a finite *domain*, an *interpretation of functors* - a functor applied to domain elements is interpreted as a domain element, and an *interpretation of predicates* - a predicate applied to domain elements is interpreted as $true$ or $false$. The elements of the domain are known to be distinct. The interpretation of functors and predicates is total, i.e., there is an interpretation for every functor and predicate for every pattern of domain element arguments.

The TPTP language is sufficient for recording a finite interpretation. The domain, interpretation of functors, and interpretation of predicates, are written as FOF annotated formulae. A recorded interpretation provides the information required for various forms of processing, such as model verification, interpretation of formulae, and identification of isomorphic interpretations.

The domain of a finite interpretation is written in the form:

```
fof(fi_name,fi_domain,
    ! [X] : ( X = e_1 | X = e_2 | ... | X = e_n ) ).
```

where the e_i are all "distinct object"s, or all distinct integers, or all distinct constant terms. If "distinct object" or integer terms appear in the interpreted signature, then all those terms must appear in the domain. If constant terms are used they are freely chosen constant terms that do not appear in the signature being interpreted. The e_i values then provide an exhaustive list of constant terms whose interpretation form the domain (there is a bijection from the constant terms to the domain, so one may think of the constant terms directly as the domain elements). The use of "distinct object"s or integer terms for a domain is preferred over constant terms, because that takes advantage of the predefined interpretation of such terms - all such terms and corresponding domain elements are known to be distinct (see Section 2). If the domain elements are constant terms then their inequality must be explicitly stated in annotated formulae of the form:

```
fof(e_i_not_e_j,fi_domain,
    e_i != e_j ).
```

The interpretation of functors is written in the form:

```
fof(fi_name,fi_functors,
    ( f(e_1,...,e_m) = e_r
    & f(e_1,...,e_p) = e_s
      ... ) ).
```

specifying that, e.g., `f(`e_1`,...,`e_m`)` is interpreted as the domain element e_r. If "distinct object"s or integer terms appear in the interpreted signature, then those terms are necessarily interpreted as themselves and must not be interpreted in the `fi_functors`.

The interpretation of predicates is written in the form:

```
fof(fi_name,fi_predicates,
    ( p(e_1,...,e_m)
    & ~ p(e_1,...,e_p)
      ... ) ).
```

specifying that, e.g., p(e_1,...,e_m) is interpreted as *true* and p(e_1,...,e_p) is interpreted as $false$. Equality is interpreted naturally by the domain, with the understanding that identical elements are equal.

Example. Consider again the FOF problem from Section 3, but with the conjecture replaced by:

```
fof(equality_lost,conjecture,(
    ! [H1,H2] :
      ( created_equal(H1,H2)
    <=> H1 = H2 ) )).
```

The resultant problem is `CounterSatisfiable`, i.e., there is a model for the axioms and negated conjecture. Here is one such model, adapted (by converting constant term domain elements to "distinct object" domain elements) from the one found by the model finding system Paradox 1.3 [3]:

```
%----------------------------------------------------------------------
fof(equality_lost,fi_domain,
    ! [X] : ( X = "a" | X = "f" | X = "john" | X = "got_a")  ).

fof(equality_lost,fi_functors,
    ( a = "a"  & f = "f"  & john = "john"
    & grade("a") = "f"    & grade("f") = "a"
    & grade("john") = "f" & grade("got_a") = "a" )  ).

fof(equality_lost,fi_predicates,
    ( human("john")                 & human("got_a")
    & ~ human("a")                  & ~ human("f")
    & ~ created_equal("a","a")      & ~ created_equal("a","f")
    & ~ created_equal("a","john")   & ~ created_equal("a","got_a")
    & ~ created_equal("f","a")      & ~ created_equal("f","f")
    & ~ created_equal("f","john")   & ~ created_equal("f","got_a")
    & ~ created_equal("john","a")   & ~ created_equal("john","f")
    & created_equal("john","john")  & created_equal("john","got_a")
    & ~ created_equal("got_a","a")  & ~ created_equal("got_a","f")
    & created_equal("got_a","john") & created_equal("got_a","got_a") ) ).
%----------------------------------------------------------------------
```

Variations, Layout, and Verification

Normally every functor and predicate is interpreted once for every pattern of domain element arguments. No functor or predicate may be interpreted more than once for an argument pattern. If a functor or predicate is not interpreted for a given argument pattern then multiple interpretations are being represented, in which that functor or predicate applied to the argument pattern is interpreted as each of the possible values (each domain element for a functor, both *true* and $false$ for a predicate).

It is recommended that interpretations follow a standard layout, as illustrated by the examples above. However, the conjuncts of functor and predicate interpretations may be separated into individual annotated formulae. Compact forms are possible using universally quantified formulae, e.g.,

```
fof(equality_lost,fi_predicates,
    ( human("john") & human("got_a")
    & ~ human("a")  & ~ human("f")
    & ! [X] : ~ created_equal("a",X)
    & ! [X] : ~ created_equal("f",X)
    & ! [X] : ~ created_equal(X,"a")
    & ! [X] : ~ created_equal(X,"f")
    & created_equal("john","john")  & created_equal("john","got_a")
    & created_equal("got_a","john") & created_equal("got_a","got_a") ) ).
```

An interpretation can be verified as a model of a set of formulae by directly evaluating each formula in the model. The TPTP format also provides an alternative approach - the interpretation is adjoined to the formulae, and a trusted model finder is then used to find a model of the combined formula set.

5 Conclusion

Standards for writing derivations and finite interpretations have been presented. These standards should be adopted by the ATP community, to increase the range of ATP tools that can be seamlessly integrated into more complex and effective reasoning systems. Increased interoperability will contribute to the usability and uptake of ATP technology in application domains.

Current work is extending the TPTP language for higher order logic [22]. When this is available, it will be used for extending the TPTP to higher order logic [1]. Future work will include the design of standards for representing infinite interpretations. As a first step, it is planned to represent Herbrand interpretations by term grammars, e.g., formulae of the form:

```
! [X,Y] : (p(X,Y) <=> ((X != a & Y != a) | (X = a & Y = a)))
```

There are decision procedures for the truth of ground atoms in the context of such formulae. Compaction of finite interpretations using normal-form theory from relational databases is also being considered.

References

1. C. Benzmüller and C. Brown. A Structured Set of Higher-Order Problems. In J. Hurd and T. Melham, editors, *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics*, LNAI 3606, pages 66–81, 2005.
2. O. Caprotti and D. Carlisle. OpenMath and MathML: Semantic Mark Up for Mathematics. *ACM Crossroads*, 6(2), 1999.
3. K. Claessen and N. Sorensson. New Techniques that Improve MACE-style Finite Model Finding. In P. Baumgartner and C. Fermueller, editors, *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*, 2003.
4. E. Denney, B. Fischer, and J. Schumann. Using Automated Theorem Provers to Certify Auto-generated Aerospace Software. In M. Rusinowitch and D. Basin, editors, *Proceedings of the 2nd International Joint Conference on Automated Reasoning*, LNAI 3097, pages 198–212, 2004.

5. J. Denzinger and S. Schulz. Recording and Analysing Knowledge-Based Distributed Deduction Processes. *Journal of Symbolic Computation*, 21:523–541, 1996.
6. M. Gordon and T. Melham. *Introduction to HOL, a Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
7. R. Hähnle, M. Kerber, and C. Weidenbach. Common Syntax of the DFG-Schwerpunktprogramm Deduction. Technical Report TR 10/96, Fakultät für Informatik, Universät Karlsruhe, Karlsruhe, Germany, 1996.
8. J. Hurd and R. Arthan. OpenTheory. http://www.cl.cam.ac.uk/ jeh1004/research/ opentheory, URL.
9. M. Kohlhase. OMDOC: Towards an Internet Standard for the Administration, Distribution, and Teaching of Mathematical Knowledge. In J.A. Campbell and E. Roanes-Lozano, editors, *Proceedings of the Artificial Intelligence and Symbolic Computation Conference, 2000*, LNCS 1930, pages 32–52, 2000.
10. W. McCune and O. Shumsky-Matlin. Ivy: A Preprocessor and Proof Checker for First-Order Logic. In M. Kaufmann, P. Manolios, and J. Strother Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, number 4 in Advances in Formal Methods, pages 265–282. Kluwer Academic Publishers, 2000.
11. W.W. McCune. Otter 3.3 Reference Manual. Technical Report ANL/MSC-TM-263, Argonne National Laboratory, Argonne, USA, 2003.
12. S. Schulz. E: A Brainiac Theorem Prover. *AI Communications*, 15(2-3):111–126, 2002.
13. S. Schulz. LOP-Syntax for Theorem Proving Applications. `http://www4.informatik.tu-muenchen.de/~schulz/WORK/lop.syntax`, URL.
14. S. Schulz and Maria Paola Bonacina. On Handling Distinct Objects in the Superposition Calculus. In B. Konev and S. Schulz, editors, *Proceedings of the 5th International Workshop on the Implementation of Logics*, pages 66–77, 2005.
15. G. Steel. Visualising First-Order Proof Search. In C. Aspinall, D. Lüth, editor, *Proceedings of User Interfaces for Theorem Provers 2005*, pages 179–189, 2005.
16. G. Sutcliffe. SystemOnTPTP. In D. McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction*, LNAI 1831, pages 406–410, 2000.
17. G. Sutcliffe. Semantic Derivation Verification. *International Journal on Artificial Intelligence Tools*, page To appear, 2006.
18. G. Sutcliffe. The TSTP Solution Library. http://www.TPTP.org/TSTP, URL.
19. G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
20. G. Sutcliffe, J. Zimmer, and S. Schulz. TSTP Data-Exchange Formats for Automated Theorem Proving Tools. In W. Zhang and V. Sorge, editors, *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, number 112 in Frontiers in Artificial Intelligence and Applications, pages 201–215. IOS Press, 2004.
21. J. Urban. MPTP - Motivation, Implementation, First Experiments. *Journal of Automated Reasoning*, 33(3-4):319–339, 2004.
22. A. Van Gelder and G. Sutcliffe. Extending the TPTP Language to Higher-Order Logic with Automated Parser Generation. In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, Lecture Notes in Artificial Intelligence, 2006.
23. J. Zimmer. A New Framework for Reasoning Agents. In V. Sorge, S. Colton, M. Fisher, and J. Gow, editors, *Proceedings of the Workshop on Agents and Automated Reasoning, 18th International Joint Conference on Artificial Intelligence*, pages 58–64, 2003.

Appendix

```
%----------------------------------------------------------------------------
%----README ... this header provides important meta- and usage information
%----
%----Intended uses of the various parts of the TPTP syntax are explained
%----in the TPTP technical manual, linked from www.tptp.org.
%----
%----Four kinds of separators are used, to indicate different types of rules:
%----  ::= is used for regular grammar rules, for syntactic parsing.
%----  :== is used for semantic grammar rules. These define specific values
%----      that make semantic sense when more general syntactic rules apply.
%----  ::- is used for rules that produce tokens.
%----  ::: is used for rules that define character classes used in the
%----       construction of tokens.
%----
%----White space may occur between any two tokens. White space is not specified
%----in the grammar, but the are some restrictions to ensure that the grammar
%----is campatible with standard Prolog: a <TPTP_file> should be readable with
%----read/1.
%----
%----The syntax of comments is defined by the <comment> rule. Comments may
%----occur between any two tokens, but do not act as white space. Comments
%----will normally be discarded at the lexical level, but may be processed
%----by systems that understand them e.g., if the system comment convention
%----is followed).
%----------------------------------------------------------------------------
%----Files. Empty file is OK.
<TPTP_file>          ::= <TPTP_input>*
<TPTP_input>         ::= <annotated_formula> | <include>

%----Formula records
<annotated_formula>  ::= <fof_annotated> | <cnf_annotated>
%----Future languages may include ...  english | efof | tfof | mathml | ...
<fof_annotated>      ::= fof(<name>,<formula_role>,<fof_formula><annotations>).
<cnf_annotated>      ::= cnf(<name>,<formula_role>,<cnf_formula><annotations>).
<annotations>        ::= <null> | ,<source><optional_info>
%----In derivations the annotated formulae names must be unique, so that
%----parent references (see <inference_record>) are unambiguous.

%----Types for problems.
%----Note: The previous <source_type> from ...
%----   <formula_role> ::= <user_role>-<source>
%----... is now gone. Parsers may choose to be tolerant of it for backwards
%----compatibility.
<formula_role>       ::= <lower_word>
<formula_role>       :== axiom | hypothesis | definition | lemma | theorem |
                         conjecture | lemma_conjecture | negated_conjecture |
                         plain | fi_domain | fi_functors | fi_predicates |
                         unknown
%----"axiom"s are accepted, without proof, as a basis for proving "conjecture"s
%----and "lemma_conjecture"s in FOF problems. In CNF problems "axiom"s are
%----accepted as part of the set whose satisfiability has to be established.
%----There is no guarantee that the axioms of a problem are consistent.
%----"hypothesis"s are assumed to be true for a particular problem, and are
%----used like "axiom"s.
%----"definition"s are used to define symbols, and are used like "axiom"s.
%----"lemma"s and "theorem"s have been proven from the "axiom"s, can be used
%----like "axiom"s, but are redundant wrt the "axiom"s. "lemma" is used as the
%----role of proven "lemma_conjecture"s, and "theorem" is used as the role of
%----proven "conjecture"s, in output. A problem containing a "lemma" or
%----"theorem" that is not redundant wrt the "axiom"s is ill-formed. "theorem"s
%----are more important than "lemma"s from the user perspective.
%----"conjecture"s occur in only FOF problems, and are to all be proven from
%----the "axiom"(-like) formulae. A problem is solved only when all
%----"conjecture"s are proven.
%----"lemma_conjecture"s are expected to be provable, and may be useful to
%----prove, while proving "conjecture"s.
```

```
%----"negated_conjecture"s occur in only CNF problems, and are formed from
%----negation of a "conjecture" in a FOF to CNF conversion.
%----"plain"s have no special user semantics, and can be used like "axiom"s.
%----"fi_domain", "fi_functors", and "fi_predicates" are used to record the
%----domain, interpretation of functors, and interpretation of predicates, for
%----a finite interpretation.
%----"unknown"s have unknown role, and this is an error situation.

%----FOF formulae. All formulae must be closed.
<fof_formula>        ::= <binary_formula> | <unitary_formula>
<binary_formula>     ::= <nonassoc_binary> | <assoc_binary>
%----Only some binary connectives are associative
%----There's no precedence among binary connectives
<nonassoc_binary>    ::= <unitary_formula> <binary_connective> <unitary_formula>
<binary_connective>  ::= <=> | => | <= | <~> | ~<vline> | ~&
%----Associative connectives & and | are in <assoc_binary>
<assoc_binary>       ::= <or_formula> | <and_formula>
<or_formula>         ::= <unitary_formula> <vline> <unitary_formula>
                         <more_or_formula>*
<more_or_formula>    ::= <vline> <unitary_formula>
<and_formula>        ::= <unitary_formula> & <unitary_formula>
                         <more_and_formula>*
<more_and_formula>   ::= & <unitary_formula>
%----<unitary_formula> are in ()s or do not have a <binary_connective> at the
%----top level.
<unitary_formula>    ::= <quantified_formula> | <unary_formula> |
                         (<fof_formula>) | <atomic_formula>
<quantified_formula> ::= <quantifier> [<variable_list>] : <unitary_formula>
<quantifier>         ::= ! | ?
%----! is universal quantification and ? is existential. Syntactically, the
%----quantification is the left operand of :, and the <unitary_formula> is
%----the right operand. Although : is a binary operator syntactically, it is
%----not a <binary_connective>, and thus a <quantified_formula> is a
%----<unitary_formula>.
%----Universal   example: ! [X,Y] : ((p(X) & p(Y)) => q(X,Y)).
%----Existential example: ? [X,Y] : (p(X) & p(Y)) & ~ q(X,Y).
%----Quantifiers have higher precedence than binary connectives, so in
%----the existential example the quantifier applies to only (p(X) & p(Y)).
<variable_list>      ::= <variable> | <variable>,<variable_list>
%----Future variables may have sorts and existential counting
%----Unary connectives bind more tightly than binary
<unary_formula>      ::= <unary_connective> <unitary_formula>
<unary_connective>   ::= ~

%----CNF formulae (variables implicitly universally quantified)
<cnf_formula>        ::= (<disjunction>) | <disjunction>
<disjunction>        ::= <literal> <more_disjunction>*
<more_disjunction>   ::= <vline> <literal>
<literal>            ::= <atomic_formula> | ~ <atomic_formula>

%----Atoms (<predicate> is not used currently)
<atomic_formula>     ::= <plain_atom> | <defined_atom> | <system_atom>
<plain_atom>         ::= <plain_term>
%----A <plain_atom> looks like a <plain_term>, but really we mean
%----  <plain_atom>         ::= <proposition> | <predicate>(<arguments>)
%----  <proposition>        ::= <atomic_word>
%----  <predicate>          ::= <atomic_word>
%----Using <plain_term> removes a reduce/reduce ambiguity in lex/yacc.
<arguments>          ::= <term> | <term>,<arguments>
<defined_atom>       ::= $true | $false |
                         <term> <defined_infix_pred> <term>
<defined_infix_pred> ::= = | !=
%----A more general formulation, which syntactically admits more defined atoms,
%----is as follows. Developers may prefer to adopt this.
%----  <defined_atom>       ::= <defined_prop> | <defined_pred>(<arguments>) |
%----                           <term> <defined_infix_pred> <term>
%----  <defined_prop>       ::= <atomic_defined_word>
%----  <defined_prop>       :== $true | $false
```

```
%----  <defined_pred>       ::= <atomic_defined_word>
%----  <defined_pred>       :==
%----Some systems still interpret equal/2 as equality. The use of equal/2
%----for other purposes is therefore discouraged. Please refrain from either
%----use. Use infix '=' for equality. Note: <term> != <term> is equivalent
%----to ~ <term> = <term>
%----More defined atoms may be added in the future.
<system_atom>        ::= <system_term>
%----<system_atom>s are used for evaluable predicates that are available
%----in particular tools. The predicate names are not controlled by the
%----TPTP syntax, so use with due care. The same is true for <system_term>s.

%----Terms
<term>               ::= <function_term> | <variable>
<function_term>      ::= <plain_term> | <defined_term> | <system_term>
<plain_term>         ::= <constant> | <functor>(<arguments>)
<constant>           ::= <atomic_word>
<functor>            ::= <atomic_word>
<defined_term>       ::= <number> | <distinct_object>
%----A more general formulation, which syntactically admits more defined terms,
%----is as follows. Developers may prefer to adopt this.
%----  <defined_term>       ::= <number> | <distinct_object> |
%----                           <defined_constant> |
%----                           <defined_functor>(<arguments>) |
%----                           <term> <defined_infix_func> <term>
%----  <defined_constant>   ::= <atomic_defined_word>
%----  <defined_constant>   :==
%----  <defined_functor>    ::= <atomic_defined_word>
%----  <defined_functor>    :==
%----  <defined_infix_func> ::=
%----System terms have system specific interpretations
<system_term>        ::= <system_constant> | <system_functor>(<arguments>)
<system_functor>     ::= <atomic_system_word>
<system_constant>    ::= <atomic_system_word>
<variable>           ::= <upper_word>

%----Formula sources
<source>             ::= <general_term>
<source>             :== <dag_source> | <internal_source> | <external_source> |
                         unknown
%----Only a <dag_source> can be a <name>, i.e., derived formulae can be
%----identified by a <name> or an <inference_record>
<dag_source>         :== <name> | <inference_record>
<inference_record>   :== inference(<inference_rule>,<useful_info>,
                             [<parent_list>])
<inference_rule>     :== <atomic_word>
%----Examples are        deduction | modus_tollens | modus_ponens | rewrite |
%                        resolution | paramodulation | factorization |
%                        cnf_conversion | cnf_refutation | ...
<parent_list>        :== <parent_info> | <parent_info>,<parent_list>
<parent_info>        :== <source><parent_details>
<parent_details>     :== :<atomic_word> | <null>
<internal_source>    :== introduced(<intro_type><optional_info>)
<intro_type>         :== definition | axiom_of_choice | tautology
%----This should be used to record the symbol being defined, or the function
%----for the axiom of choice
<external_source>    :== <file_source> | <theory> | <creator_source>
<file_source>        :== file(<file_name><file_info>)
<file_info>          :== ,<name> | <null>
<theory>             :== theory(<theory_name><optional_info>)
<theory_name>        :== equality | ac
%----More theory names may be added in the future. The <optional_info> is
%----used to store, e.g., which axioms of equality have been implicitly used,
%----e.g., theory(equality,[rst]). Standard format still to be decided.
<creator_source>     :== creator(<creator_name><optional_info>)
<creator_name>       :== <atomic_word>

%----Useful info fields
```

```
<optional_info>      ::= ,<useful_info> | <null>
<useful_info>        ::= <general_term_list>
<useful_info>        :== [] | [<info_items>]
<info_items>         :== <info_item> | <info_item>,<info_items>
<info_item>          :== <formula_item> | <inference_item> | <general_function>
%----Useful info for formula records
<formula_item>       :== <description_item> | <iquote_item>
<description_item>   :== description(<atomic_word>)
<iquote_item>        :== iquote(<atomic_word>)
%----<iquote_item>s are used for recording exactly what the system output about
%----the inference step. In the future it is planned to encode this information
%----in standardized forms as <parent_details> in each <inference_record>.
%----Useful info for inference records
<inference_item>     :== <inference_status> | <refutation>
<inference_status>   :== status(<status_value>) | <inference_info>
%----These are the status values from the SZS ontology
<status_value>       :== tau | tac | eqv | thm | sat | cax | noc | csa | cth |
                         ceq | unc | uns | sab | sam | sar | sap | csp | csr |
                         csm | csb
%----The most commonly used status values are:
%----  thm - Every model (and there are some) of the parent formulae is a
%----        model of the inferred formula. Regular logical consequences.
%----  cth - Every model (and there are some) of the parent formulae is a
%----        model of the negation of the inferred formula. Used for negation
%----        of conjectures in FOF to CNF conversion.
%----  sab - There is a bijection between the models (and there are some) of
%----        the parent formulae and models of the inferred formula. Used for
%----        Skolemization steps.
%----For the full hierarchy see the SZSOntology file distributed with the TPTP.
<inference_info>     :== <inference_rule>(<atomic_word>,<general_list>)
<refutation>         :== refutation(<file_source>)
%----Useful info for creators is just <general_function>

%----Include directives
<include>            ::= include(<file_name><formula_selection>).
<formula_selection>  ::= ,[<name_list>] | <null>
<name_list>          ::= <name> | <name>,<name_list>

%----Non-logical data
<general_term>       ::= <general_data> | <general_data>:<general_term> |
                         <general_list>
<general_data>       ::= <atomic_word> | <atomic_word>(<general_arguments>) |
                         <number> | <distinct_object>
<general_arguments>  ::= <general_term> | <general_term>,<general_arguments>
<general_list>       ::= [] | [<general_term_list>]
<general_term_list>  ::= <general_term> | <general_term>,<general_term_list>

%----General purpose
<name>               ::= <atomic_word> | <unsigned_integer>
<atomic_word>        ::= <lower_word> | <single_quoted>
%----This maybe useful in the future
%----    <atomic_defined_word> ::= <dollar_word>
<atomic_system_word> ::= <dollar_dollar_word>
<number>             ::= <real> | <signed_integer> | <unsigned_integer>
%----Numbers are always interpreted as themselves, and are thus implicitly
%----distinct if they have different values, e.g., 1 != 2 is an implicit axiom.
%----All numbers are base 10 at the moment.
<file_name>          ::= <atomic_word>
<null>               ::=

%------------------------------------------------------------------------------
%----Rules from here on down are for defining tokens (terminal symbols) of the
%----grammar, assuming they will be recognized by a lexical scanner.
%----A ::- rule defines a token, a ::: rule defines a macro that is not a
%----token. Usual regexp notation is used. Single characters are always placed
%----in []s to disable any special meanings (for uniformity this is done to
%----all characters, not only those with special meanings).
```

```
%----These are tokens that appear in the syntax rules above. No rules
%----defined here because they appear explicitly in the syntax rules.
%----Keywords:    fof cnf include
%----Punctuation: ( ) , . [ ] :
%----Operators:   ! ? ~ & | <=> => <= <~> ~| ~&
%----Predicates:  = != $true $false

<comment>            ::- <comment_line>|<comment_block>
<comment_line>       ::: [%]<printable_char>*
<comment_block>      ::: [/][*]<not_star_slash>[*][*]*[/]
<not_star_slash>     ::: ([^*]*[*][*]*[^/*])*[^*]*
%----System comments are a convention used for annotations that may used as
%----additional input to a specific system. They look like comments, but start
%----with %$$ or /*$$. A wily user of the syntax can notice the $$ and extract
%----information from the "comment" and pass that on as input to the system.
%----The specific system for which the information is intended should be
%----identified after the $$, e.g., /*$$Otter 3.3: Demodulator */
%----To extract these separately from regular comments, the rules are:
%----  <system_comment>     ::- <sys_comment_line>|<sys_comment_block>
%----  <sys_comment_line>   ::: [%]<dollar_dollar><printable_char>*
%----  <sys_comment_block>  ::: [/][*]<dollar_dollar><not_star_slash>[*][*]*[/]
%----A string that matches both <system_comment> and <comment> should be
%----recognized as <system_comment>, so put these before regular comments.

<single_quoted>      ::- [']([^\\']|[\\][']|[\\][\\])*[']
%----<single_quoted>      ::- '<printable_char>*', but ' and \ are escaped.
%----\ is used as the escape character for ' and \, i.e., if \' is  encountered
%----the ' is not the end of the <single_quoted>, and if \\ is encountered the
%----second \ is not an escape. Both characters (the escape \ and the following
%----' or \) are retained and printed on output. Behaviour is undefined if the
%----escape \ is followed by anything other than ' or \. Behaviour is undefined
%----if a non-<printable_char> is encountered. If the contents of a <single
%----quoted> constitute a <lower_word>, then the ''s should be stripped to
%----produce a <lower_word>.
<distinct_object>    ::- ["]([^\\"]|[\\]["]|[\\][\\])*["]
%----<distinct_object>    ::- "<printable_char>*", but " and \ are escaped. The
%----comments for <single_quoted> apply, with ' replaced by ".
%----Distinct objects are always interpreted as themselves, and are thus
%----implicitly distinct if they look different, e.g., "Apple" != "Microsoft"
%----is an implicit axiom.

<dollar_dollar_word> ::- <dollar_dollar><lower_word>
<upper_word>         ::- <upper_alpha><alpha_numeric>*
<lower_word>         ::- <lower_alpha><alpha_numeric>*

%----Numbers
<real>               ::- (<signed_decimal>|<unsigned_decimal>)<fraction_decimal>
<signed_integer>     ::- <sign><unsigned_integer>
<unsigned_integer>   ::- <unsigned_decimal>
<signed_decimal>     ::: <sign><unsigned_decimal>
<sign>               ::: [+-]
<unsigned_decimal>   ::: ([0]|<non_zero_numeric><numeric>*)
<fraction_decimal>   ::: [.]<numeric><numeric>*

%----Character classes
<numeric>            ::: [0-9]
<non_zero_numeric>   ::: [1-9]
<lower_alpha>        ::: [a-z]
<upper_alpha>        ::: [A-Z]
<alpha_numeric>      ::: (<lower_alpha>|<upper_alpha>|<numeric>|[_])
<dollar_dollar>      ::: [$][$]
<printable_char>     ::: .
%----<printable_char>     ::: any printable ASCII character, codes 32-126
%----<printable_char> thus includes spaces, but not tabs, newlines, bells, etc.
%----This definition does not capture that.
<vline>              ::: [|]
%--------------------------------------------------------------------------------
```