Formal specification of Minimal Type Theory

This is the formal YACC BNF specification for Minimal Type Theory (MTT). MTT was created by augmenting the syntax of First Order Logic (FOL) to specify Higher Order Logic (HOL) expressions using FOL syntax.

This second-order sentence ∀P∀x (Px ∨ ¬Px) says that for every formula P, and every individual x, either Px is true or not(Px) is true (this is the principle of bivalence).[1]

Here it is encoded using first order logic syntax and the definition operator:
(a) S1 := ∀x (Px ∨ ¬Px)
(b) ∀P(S1)
Every instance of the left-hand-side of a definition is to be expanded into its right-hand-side.
Thus the above two lines specify: ∀P(∀x (Px ∨ ¬Px)).

MTT can also correctly parse second order logic expressions directly: ∀P∀x (P(x) ∨ ~P(x))

```
<sentence_4  token="FOR_ALL">
 <sentence_4  token="IDENTIFIER"  value="P"/>
 <sentence_4  token="FOR_ALL">
  <sentence_4  token="IDENTIFIER"  value="x"/>
  <sentence_13  token="OR">
   <atomic_sentence_1  token="IDENTIFIER"  value="P">
    <term_2  token="IDENTIFIER"  value="x"/>
   </atomic_sentence_1>
   <sentence_2  token="NOT">
    <atomic_sentence_1  token="IDENTIFIER"  value="P">
     <term_2  token="IDENTIFIER"  value="x"/>
    </atomic_sentence_1>
   </sentence_2>
  </sentence_13>
 </sentence_4>
</sentence_4>
```

Another key use of the definition operator is to properly formalize self-referential expressions.
“This sentence is not true.” would be formalized as:
LP := ~True(LP) which expands into: ~True(~True(~True(~True(...)))) infinite recursion.

“This sentence is not provable.” would be formalized as:
G := ~Provable(G) which expands into: ~Provable(~Provable(~Provable(...))) infinite recursion

That macro expansion results in infinite recursion is documented in this article:
3.10.5 Self-Referential Macros
A self-referential macro is one whose name appears in its definition. Recall that all macro definitions are rescanned for more macros to replace. If the self-reference were considered a use of the macro, it would produce an infinitely large expansion. ...

Following the ordinary rules, each reference to foo will expand into (4 + foo); then this will be rescanned and will expand into (4 + (4 + foo)); and so on until the computer runs out of memory.
https://gcc.gnu.org/onlinedocs/cpp/Self-Referential-Macros.html

MTT is intended to be used as a universal Tarski meta-language including a meta-language to itself. Because MTT has its own provability operator: ″⊢″ provability can be analyzed directly within the deductive inference model instead indirectly through diagonalization. This allows us to see exactly why an expression of language can be neither proved nor disproved, details that diagonalization cannot provide. **All of the symbolic logic operators retain their conventional semantic meaning from mathematical logic.**

[1] Second-order logic, https://en.wikipedia.org/w/index.php?title=Second-order_logic&oldid=979950180

```
%left  IDENTIFIER      //  Letter+ (Letter | Digit)*  // Letter includes UTF-8
%left  SUBSET_OF       //  ⊆
%left  ELEMENT_OF      //  ∈
%left  FOR_ALL         //  ∀
%left  THERE_EXISTS    //  ∃
%left  IMPLIES         //  →
%left  PROVES          //  ⊢
%left  IFF             //  ↔
%left  AND             //  ∧
%left  OR              //  ∨
%left  NOT             //  ~
%left  ASSIGN_ALIAS    // := (definition operator) x := y means x is defined to be another name for y
                       //  LHS is assigned as an alias name for the RHS (macro substitution)
%%                     //  An alias named expression is treated syntactically as a propositional
                       //  variable in the next higher level of logic specifying HOL using FOL syntax.
                       //  This alias name is then treated semantically as if it was macro expanded.
definition
      : sentence
      | IDENTIFIER ASSIGN_ALIAS sentence                    // Enhancement to FOL
      ;

sentence
      : atomic_sentence
      | '~' sentence %prec NOT
      | '(' sentence ')'
      | sentence   IMPLIES      sentence
      | sentence   IFF          sentence
      | sentence   AND          sentence
      | sentence   OR           sentence
      | quantifier IDENTIFIER   sentence
      | quantifier IDENTIFIER   type_of IDENTIFIER sentence  // Enhancement to FOL
      | sentence   PROVES       sentence                    // Enhancement to FOL
      ;

atomic_sentence
      : IDENTIFIER '(' term_list ')' // ATOMIC PREDICATE
      | IDENTIFIER                   // SENTENTIAL VARIABLE  // Enhancement to FOL
      ;

term
      : IDENTIFIER '(' term_list ')' // FUNCTION
      | IDENTIFIER                   // CONSTANT or VARIABLE
      ;

term_list
      : term_list ',' term
      | term
      ;

type_of
     : ELEMENT_OF                    // Enhancement to FOL
     | SUBSET_OF                     // Enhancement to FOL
     ;

quantifier
     : THERE_EXISTS
     | FOR_ALL
     ;
```

Copyright 2017, 2018, 2019 Pete Olcott

The above MTT grammar is based on the following grammar for first order logic:
https://groups.google.com/forum/#!original/comp.compilers/Qalayu9h3xw/gaTrXbbRd7AJ

From: Gene <gene....@gmail.com>
Newsgroups: comp.compilers
Subject: Re: First Order Logic (FOL) parsing
Date: Fri, 10 Dec 2010 19:27:14 -0800 (PST)
Organization: Compilers Central

```
It should certainly be easy to parse any reasonable grammar for FOL
with a LALR(1) or LL(1) parser.  For example if you translate the
description at http://www.sju.edu/~jhodgson/ugai/1order.html to Bison,
you get something like this

%token PREDICATE FUNCTION CONSTANT VARIABLE
%token IMPLIES AND OR IFF THERE_EXISTS FORALL
%left '='
%left VARIABLE
%left IMPLIES
%left IFF
%left AND
%left OR
%left NOT
%%

sentence
        : atomic_sentence
        | sentence IMPLIES sentence
        | sentence IFF sentence
        | sentence AND sentence
        | sentence OR sentence
        | quantifier VARIABLE sentence
        | '~' sentence %prec NOT
        | '(' sentence ')'
        ;
atomic_sentence
        : PREDICATE '(' term_list ')'
        | term '=' term
        ;
term
        : FUNCTION '(' term_list ')'
        | CONSTANT
        | VARIABLE
        ;
term_list
        : term_list term
        | term
        ;
quantifier
        : THERE_EXISTS
        | FORALL
        ;
```