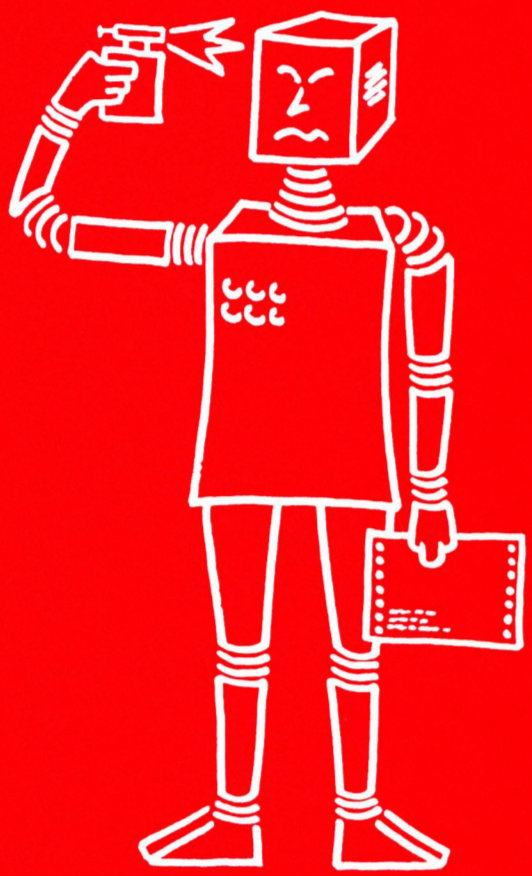
SEKI-PROJEKT

SEKI MEMO

Institut für Informatik III
Universität Bonn
Bertha-von-Suttner-Platz 6
D 5300 Bonn 1, W. Germany

Institut für Informatik I
Universität Karlsruhe
Postfach 6380
D-7500 Karlsruhe 1, W. Germany



THE MARKGRAF KARL REFUTATION PROCEDURE
PLL - A First-Order Language for an Automated Theorem Prover

Christoph Walther
Institut für Informatik 1
Universität Karlsruhe
Postfach 6380
D-7500 Karlsruhe 1

UNIVERSITÄT KARLSRUHE
FAKULTÄT FÜR INFORMATIK
Postfach 63 80, D 7500 Karlsruhe 1

Interner Bericht 35/82

THE MARKGRAF KARL REFUTATION PROCEDURE
PLL - A First-Order Language for an Automated Theorem Prover

Christoph Walther
Institut für Informatik 1
Universität Karlsruhe
Postfach 6380
D-7500 Karlsruhe 1

Abstract

The PREDICATE LOGIC LANGUAGE (PLL), a formal language in which first-order predicate logic formulas are formulated, is described. In PLL axioms and theorems are represented which are given to the MARKGRAF KARL REFUTATION PROCEDURE. Certain expressions of PLL which reflect the special facilities of this system are exhibited, viz.

- an inference mechanism based on a many-sorted calculus,

- the incorporation of special axioms into the inference mechanism, and

- the control of the inference mechanism using special derivation strategies.

Contents

1 Introduction

This is a description of the PREDICATE LOGIC LANGUAGE (PLL) which is used as the input language of the MARKGRAF KARL REFUTATION PROCEDURE, an automated theorem prover (ATP) developed at the University of Karlsruhe [BES 81, DMW 81, Ohl 82].

In PLL first-order predicate logic formulas are represented which are given as axioms or theorems (to be proved) to the ATP. Additionally PLL reflects some special facilities of the Karlsruhe ATP system, viz.

- an inference mechanism based on a many-sorted calculus,

- the incorporation of special axioms into the inference mechanism, and

- the control of the inference mechanism using special derivation strategies.

In chapter 2 we present an informal introduction to PLL, where some language examples are given to provide the reader with a general idea of which expressions can be formulated in PLL together with their intended meaning. Since the syntax of PLL is relatively simple for users familiar with predicate logic or programming languages, a careful study of these examples yields a rapid survey of PLL. In chapter 3 we define the syntax of PLL by a context free grammar. The semantic constraints (i.e. the context dependent language features) for PLL are given in chapter 4. Finally in chapter 5 we make some remarks regarding the PLL-compiler of the ATP system.

2 An Introduction to PLL

2.1 Basic Concepts

In PLL the usual junctors, denoted OR, AND, IMPL, EQV and NOT, the universal quantifier ALL, and the existential quantifier EX are present. Junctors and quantifiers are given the following priorities when used in a formula without parentheses :

(1) NOT
(2) AND
(3) OR
(4) IMPL
(5) EQV
(6) ALL , EX

NOT has the highest priority. In a formula without parentheses the rightmost junctor has precedence over all junctors with the same priority left of it.

Example 2.1.1
NOT A OR B AND C is equivalent to
(NOT A) OR (B AND C) and

A IMPL B IMPL C is equivalent to
A IMPL (B IMPL C).

In PLL the sign = denotes the equality symbol, i.e. we use a first-order predicate calculus with equality. As an example for using PLL, we axiomatize a group :

Example 2.1.2

```
* AXIOMATIZATION OF A GROUP WITH EQUALITY, *
* F IS THE GROUP OPERATOR AND 1 IS THE IDENTITY ELEMENT *

ALL X,Y EX Z   F(X Y) = Z
ALL X,Y,Z      F(X F(Y Z)) = F(F(X Y) Z)
ALL X          F(1 X) = X AND F(X 1) = X
ALL X EX Y     F(X Y) = 1
```

A theorem given to the ATP could be, for instance :

```
* IDEMPOTENCY IMPLIES COMMUTATIVITY *

ALL X F(X X) = 1 IMPL (ALL X,Y F(X Y) = F(Y X) )
```

The lines enclosed in asterisks are PLL-comments. We give another axiomatization of a group:

Example 2.1.3

```
* AXIOMATIZATION OF A GROUP WITHOUT EQUALITY *
* P(X Y Z) MEANS F(X Y) = Z WHERE F IS THE *
* GROUP OPERATOR . E IS THE LEFTIDENTITY *

ALL X,Y EX Z      P(X Y Z)
ALL X,Y,Z,U,V,W P(X Y U) AND P(Y Z V) IMPL
                  ( P(X V W) EQV P(U Z W) )
ALL X             P(E X X)
ALL X EX Y        P(X Y E)
```

Now a theorem could be, for instance:

```
* LEFTIDENTITY IS RIGHTIDENTITY *

ALL X P(X E X)
```

2.2 The Many-Sorted Calculus

Let us assume we have a finite and non-empty set of sort symbols ordered by the subsort order, i.e. a partial order relation which is reflexive, antisymmetric and transitive. Variable, constant and function symbols are associated with a certain sort symbol, called the rangesort of the respective symbol. The sort of a variable or constant symbol is its rangesort and the sort of a term which is different from a variable or constant symbol is determined by the rangesort of its outermost function symbol.

All argument positions of a function or predicate symbol are associated with certain sort symbols, called the domainsorts. In the construction of the well-formed formulas of the many-sorted calculus, only those terms whose sorts are subsorts of the domainsort given for an argument position of a function or predicate symbol may fill this argument position of the function or predicate symbol.

Besides the increase of readability of axiomatizations, the usage of the information given by the range and domainsorts and by the subsort order prevents the inference mechanism of an automated theorem prover from performing useless derivations. The theoretical foundation of the many-sorted calculus which is the basis of the MARKGRAF KARL REFUTATION PROCEDURE can be found in [Wal 82].

As an example for an application of a many-sorted calculus we axiomatize sets of letters and digits and some basic operations for these sets:

Example 2.2.1

```
* DEFINITION OF THE SORTS LETTER AND DIGIT, I.E. *
* A,B,...,Z ARE CONSTANTS OF SORT LETTER AND *
* 0,1,...,9 ARE CONSTANTS OF SORT DIGIT *

TYPE A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z : LETTER
TYPE 0,9,8,7,6,5,4,3,2,1 : DIGIT
```

```
* LETTER AND DIGIT ARE SUBSORTS OF SORT SIGN *

SORT LETTER,DIGIT:SIGN

* DEFINITION OF THE EMPTY SET AND SET-MEMBERSHIP, *
* I.E. EMPTY IS A CONSTANT OF SORT SET AND MEMBER *
* IS A BINARY PREDICATE DEFINED ON (SIGN SET) *

TYPE EMPTY:SET
TYPE MEMBER(SIGN SET)
ALL X:SIGN NOT MEMBER(X EMPTY)
ALL U,V:SET U = V EQV (ALL X:SIGN MEMBER(X U) EQV MEMBER(X V) )

* DEFINITION OF SINGLETONS, I.E. *
* SINGLETON IS A FUNCTION MAPPING SIGN TO SET *

TYPE SINGLETON(SIGN):SET
ALL X,Y:SIGN MEMBER(X SINGLETON(X)) AND
             (MEMBER(Y SINGLETON(X)) IMPL X = Y)


* DEFINITION OF SET-UNION, I.E. *
* UNION IS A FUNCTION MAPPING (SET SET) TO SET *

TYPE UNION(SET SET):SET
ALL X:SIGN ALL U,V:SET (MEMBER(X U) OR MEMBER(X V) )
                        EQV MEMBER(X UNION(U V))
```

Theorems to be proved by the ATP system could be, for instance:

```
* UNION IS IDEMPOTENT AND EMPTY IS AN IDENTITY ELEMENT *

ALL X:SET UNION(X X) = X AND UNION(EMPTY X) = X

* SINGLETON IS INJECTIVE *

ALL X,Y:SIGN SINGLETON(X) = SINGLETON(Y) IMPL X = Y
```

2.3 Attributes of Functions and Predicates

Attributes are abbreviations for their defining axioms, i.e. first-order axioms which axiomatize certain properties of functions or predicates.

The effect of stating a certain attribute of a function or predicate using an attribute declaration is formally the same as giving the defining axiom to the ATP. At the moment the following attributes can be declared:

attribute declaration	defining axiom
REFLEXIVE(P)	ALL X P(X X)
IRREFLEXIVE(P)	ALL X NOT P(X X)
SYMMETRIC(P)	ALL X,Y P(X Y) IMPL P(Y X)
ASSOCIATIVE(F)	ALL X,Y,Z F(X F(Y Z)) = F(F(X Y) Z)

In the MARKGRAF KARL REFUTATION PROCEDURE the defining axioms of attributes are incorporated into the inference mechanism of the system (partially by using special unification algorithms) [Ohl 82].

Example 2.3.1

In example 2.1.2, for instance, the associativity of the group operator F could be stated by: ASSOCIATIVE(F). In example 2.2.1 we could write the following as an axiom: ASSOCIATIVE(UNION).

2.4 Special Junctors and Equality Symbols

For the junctors introduced in section 2.1, PLL offers alternative notations:

junctor	alternative notations
AND	AND: , :AND or :AND:
OR	OR: , :OR or :OR:
IMPL	IMPL: , :IMPL or :IMPL:
EQV	EQV: , :EQV or :EQV:

The equality symbol = can be alternatively denoted by := , =: or :=: .

The colon-notation of junctors and equality symbols is used to influence the sequence of deductions performed by the ATP (see [Ohl 82]). Semantically there is no difference between junctors and equality symbols written with or without colons.

3 The Syntax of PLL

The syntax of PLL is defined by the following context free grammar:

Terminal Alphabet

SORT TYPE ANY TRUE FALSE = =: := :=:
ASSOCIATIVE REFLEXIVE IRREFLEXIVE SYMMETRIC
ALL EX EQV :EQV EQV: :EQV: IMPL :IMPL IMPL: :IMPL:
OR :OR OR: :OR: AND :AND AND: :AND: NOT
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Ø 1 2 3 4 5 6 7 8 9
: ; * ! # $ & - @ + ; / .) (? " ´ > <] [

Non-terminal Alphabet

Every string enclosed in angle brackets, e.g. <term>.

<> denotes the empty word.

Start Symbol <expression>

Production Rules

```
<expression>              -> <> | <comment> <expression> |
                             <sort declaration> <expression> |
                             <type declaration> <expression> |
                             <attribute declaration> <expression> |
                             <quantification> <expression>

-------------------------------------------------------------

<comment>                 -> * ´any sequence of symbols´ *

-------------------------------------------------------------

<sort declaration>        -> SORT <sort list> : <sort list>
```

```
<type declaration>      -> TYPE <constant list> : <sort symbol> |
                           TYPE <function symbol>
                             ( <sort symbols> ) : <sort symbol> |
                           TYPE <predicate symbol>
                                              ( <sort symbols> )
```

```
<attribute declaration> -> ASSOCIATIVE ( <function symbol> ) |
                           REFLEXIVE ( <predicate symbol> ) |
                           IRREFLEXIVE ( <predicate symbol> ) |
                           SYMMETRIC ( <predicate symbol> )
```

```
<quantification>        -> <equivalence> |
                           ALL <variable declaration>
                               <quantification> |
                           EX <variable declaration>
                              <quantification>

<equivalence>           -> <implication> |
                           <implication> <eqv> <equivalence>

<implication>           -> <disjunction> |
                           <disjunction> <impl> <implication>

<disjunction>           -> <conjunction> |
                           <conjunction> <or> <disjunction>

<conjunction>           -> <negation> |
                           <negation> <and> <conjunction>

<negation>              -> <atomic formula> |
                           NOT <atomic formula>

<atomic formula>        -> ( <quantification> ) | <atom>
```

```
<eqv>                    -> EQV | :EQV | EQV: | :EQV:

<impl>                   -> IMPL | :IMPL | IMPL: | :IMPL:

<or>                     -> OR | :OR | OR: | :OR:

<and>                    -> AND | :AND | AND: | :AND:

-------------------------------------------------------------

<atom>                   -> <predicate symbol> |
                            <predicate symbol> ( <terms> ) |
                            <term> <equality symbol> <term>

-------------------------------------------------------------

<variable declaration>   -> <variable list> <variable sort>

<variable sort>          -> <> | : <sort symbol>

-------------------------------------------------------------

<term>                   -> <constant symbol> | <variable symbol>|
                            <function symbol> ( <terms> )

<terms>                  -> <term> | <term> <terms>

-------------------------------------------------------------

<sort list>              -> <sort symbol> |
                            <sort symbol> , <sort list>

<sort symbols>           -> <sort symbol> |
                            <sort symbol> <sort symbols>

<constant list>          -> <constant symbol> |
                            <constant symbol> , <constant list>

<variable list>          -> <variable symbol> |
                            <variable symbol> , <variable list>
```

```
<sort symbol>           -> ANY | <identifier>

<constant symbol>       -> <identifier> | <number> | <name>

<function symbol>       -> <identifier> | <name>

<predicate symbol>      -> TRUE | FALSE | <identifier> | <name>

<equality symbol>       -> = | := | =: | :=:

<variable symbol>       -> <identifier>
```

```
<identifier>            -> <letter> | <identifier> <letter> |
                           <identifier> <digit> |
                           <identifier> <special sign>

<number>                -> <digit> | <number> <digit>

<name>                  -> <special sign> | <name> <letter> |
                           <name> <digit> | <name> <special sign>
```

```
<letter>                -> A | B | C | D | E | F | G | H | I |
                           J | K | L | M | N | O | P | Q | R |
                           S | T | U | V | W | X | Y | Z

<digit>                 -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<special sign>          -> ! | # | $ | & | * | - | + | ; | ? |
                           / | . | @ | " | ' | = | > | < | ] | [
```

The keywords SORT, TYPE, ANY etc. are member of the terminal alphabet and hence never accepted as an <identifier> (see also section 5.1). The following signs act as separator characters:

- a blank,

- the parentheses, viz.) and (,

- the colon, e.g. X:Y is the same as X : Y and

- the comma, e.g. X,Y is the same as X , Y.

A sequence of blanks is always read as one blank. The signs * and = may be used in a <name>, but they have a special meaning if they are enclosed in a pair of separator characters, viz. to indicate a <comment> and to denote an <equality symbol>.

4 Semantic Constraints for PLL

In the following we state the semantic constraints (i.e. the context dependent language features) as defined for PLL. The strings in angle brackets, e.g. <term>, refer to the production rules of the PLL-grammar (see section 3).

4.1 Introduction of Symbols

4.1.1 Sort Symbols

Sort symbols are introduced by their first usage in

- a <sort declaration>, e.g. SORT LETTER,DIGIT:SIGN,ALPHABET

- a <type declaration>, e.g. TYPE A,B:BOOL ,
  TYPE MEMBER(SIGN,SET) or TYPE SINGLETON(SIGN):SET,

- a <variable declaration>, e.g. ALL Z:INT EX N:NAT ABS(Z) = N.

The direct subsort relation imposed on the set of sort symbols is a partial, irreflexive, non-transitive and finite relation such that the predefined sort symbol ANY is a direct subsort of no sort symbol and each sort symbol different from ANY is a direct subsort of at least one other sort symbol.

The subsort order imposed on the set of sort symbols is the reflexive and transitive closure of the direct subsort relation.

The sort symbols to the left of the colon in a <sort declaration> are direct subsorts of each sort symbol to the right of the colon in the <sort declaration>.

The sort symbols to the right of the colon in a <sort declaration> are direct subsorts of ANY, provided these sort symbols are introduced by this <sort declaration>.

The sort symbols which are introduced by a <type declaration> or by a <variable declaration> are direct subsorts of ANY.

Example 4.1.1.1

For the <sort declaration> given above LETTER and DIGIT are direct subsorts of SIGN and of ALPHABET, and SIGN and ALPHABET are direct subsorts of ANY. Hence LETTER, DIGIT and SIGN are subsorts of SIGN and ANY, SIGN, ALPHABET, LETTER and DIGIT are subsorts of ANY.

4.1.2 Variable Symbols

Variable symbols are introduced by a <variable declaration> in a <quantification>.

Example 4.1.2.1
ALL X,Y EX Z:S P(X Y Z)

The scope of a <variable symbol> is the <quantification> following the <variable declaration> in a <quantification>.

In its scope each <variable symbol> has as rangesort the sort symbol given by the <sort symbol> following the colon in its <variable sort> of the <variable declaration>. If no <variable sort> is present, the rangesort of the <variable symbol> is the predefined sort symbol ANY.

Example 4.1.2.2
We obtain for the expression given in example 4.1.2.1
rangesort(X) = rangesort(Y) = ANY and
rangesort(Z) = S.

In each <quantification> variable symbols are consistently renamed from left to right to resolve conflicts resulting from multiple introductions of variable symbols.

Example 4.1.2.3
ALL X,X P(X) is the same as ALL X,Y P(Y) and

ALL X (EX X P(X)) IMPL Q(X) is the same as
ALL X (EX Y P(Y)) IMPL Q(X)

4.1.3 Constant Symbols

Constant symbols are introduced by their first usage

- in a <type declaration>, e.g.TYPE -1,+1:INT

- as a <term>, e.g. ALL X P(X A) OR F(C) = D

Each constant symbol has as rangesort the <sort symbol> following the colon in the <type declaration> which introduces the <constant symbol>. The rangesort of a constant symbol which is introduced by its first usage as a <term> is ANY.

Example 4.1.3.1
For the expressions given above we find
rangesort(-1)=rangesort(+1)=INT and
rangesort(A)=rangesort(C)=rangesort(D)=ANY.

Note that in PLL variable symbols are always preceded by a quantifier and thereby can always be distinguished from constant symbols. As a consequence there is no concept of free variables in PLL.

4.1.4 Function Symbols

Function symbols are introduced by their first usage in

- a <type declaration>, e.g. TYPE ABS(INT):NAT

- an <attribute declaration>, e.g. ASSOCIATIVE(PLUS)

- a <term>, e.g. ALL X P(F(X)) OR G(X) = A.

Each function symbol is associated with a sort symbol for each argument position i, called its i-th domainsort, with a natural number, called its arity, and with a sort symbol, called its rangesort.

Function symbols which are introduced by a <type declaration> have as i-th domainsort the <sort symbol> given on the i-th position in the list of <sort symbols> following the <function symbol> in the <type declaration>.

Example 4.1.4.1
For the expression TYPE PRODUCT(SCALAR VECTOR):VECTOR we obtain domainsort(PRODUCT 1)=SCALAR and domainsort(PRODUCT 2)=VECTOR.

A <function symbol> which is introduced by an <attribute declaration> or by its first usage in a <term> has ANY as i-th domainsort for each argument position i.

The arity of a function symbol is given by

- the number of sort symbols in the list of <sort symbols> following the <function symbol> in the <type declaration> which introduces the <function symbol>

- 2, for a <function symbol> introduced by an <attribute declaration>

- the number of arguments on its first usage in a <term>.

Example 4.1.4.2

For the expressions given above we obtain arity(ABS) = 1 , arity(PLUS) = 2 and arity(F) = arity(G) = 1 .

The rangesort of a <function symbol> is defined by the <sort symbol> following the colon in a <type declaration>. Its rangesort is ANY if the <function symbol> is introduced by an <attribute declaration> or by its first usage in a <term>.

Example 4.1.4.3

For the expressions given in the examples above we obtain
rangesort(ABS) = NAT ,
rangesort(PRODUCT) = VECTOR, and
rangesort(PLUS) = rangesort(F) = rangesort(G) = ANY.

4.1.5 Predicate Symbols

A predicate symbol is introduced by its first usage in

- a <type declaration>, e.g. TYPE MEMBER(SIGN SET)

- an <attribute declaration>, e.g. IRREFLEXIVE(LESS)

- an <atom> , e.g. EX X,Y P(X Y) AND Q.

Each predicate symbol is associated with a natural number, called its arity, and with a sort symbol for each argument position i, called its i-th domainsort.

The arity and domainsorts of predicate symbols are determined in the same way arity and domainsorts are determined for function symbols.

Each <equality symbol> is a predefined predicate symbol whose arity is 2 and whose 1st and 2nd domainsort is ANY. They are the only predicate symbols which are written in infix notation.

TRUE and FALSE are predefined predicate symbols with arity Ø, which have the obvious meaning.

4.2 Semantically correct Expressions

In this section the numbers in angle brackets, e.g. <23>, denote error code numbers returned by the PLL-compiler of the MARKGRAF KARL REFUTATION PROCEDURE (see section 5) when given a semantically incorrect <expression> as input. Subsequently the phrase unknown symbol denotes a string of the terminal alphabet of the PLL-grammar, which has never been used before.

4.2.1 Semantically correct Sort Declarations

A <sort declaration> SORT S:T is semantically correct iff

- S and T are sort symbols or else unknown symbols (otherwise error message) <61,62,63,64> and S and T are different symbols <65> and S is a direct subsort of T or else at least one of the symbols S or T is unknown <66>.

A <sort declaration> SORT S1,...,Sm:T1,...,Tn is semantically correct iff each <sort declaration> in the sequence

SORT S1:T1 ... SORT Sm:T1 ..... SORT S1:Tn ... SORT Sm:Tn

is semantically correct.

4.2.2 Semantically correct Type Declarations

A <type declaration> T is semantically correct iff

- T is TYPE C1,...,Cn:S and S is a sort symbol or else an unknown symbol <61,62,63,64> and for all i=1...n Ci is a constant symbol with rangesort(Ci) = S <14> or Ci is an unknown symbol <11,12,16,17> or

- T is TYPE P(S1...Sn) and for all i=1...n Si is a sort symbol or else an unknown symbol <61,62,63,64> and P is a predicate symbol with arity(P)=n <34> and domainsort(P i)=Si <36> or else an unknown symbol <31,32,33,37> or

- T is TYPE F(S1...Sn):S and for all i=1...n S and Si are sort symbols or else unknown symbols <61,62,63,64> and F is a function symbol with arity(F)=n <23>, rangesort(F)=S <27> and domainsort(F i)=Si <26> or an unknown symbol <21,22,24,28>.

4.2.3 Semantically correct Attribute Declarations

An <attribute declaration> ASSOCIATIVE(F) is semantically correct iff

- F is a function symbol with arity(F)=2 <23>, rangesort(F) = domainsort(F 1) = domainsort(F 2) <26> or else an unknown symbol <21,22,24,28>.

The <attribute declaration> REFLEXIVE(P), IRREFLEXIVE(P) and SYMMETRIC(P) are semantically correct iff

- P is a predicate symbol with arity(P)=2 <34> and domainsort(P 1) = domainsort(P 2) <36> or else an unknown symbol <31,32,33,37>.

4.2.4 Semantically correct Terms, Atoms and Quantifications

The sort of a term t, denoted sort(t), is the rangesort of t, if t is a variable or constant symbol, or else the rangesort of the outermost function symbol of t.

A <term> T is semantically correct iff

- T is a constant symbol, a variable symbol or an unknown symbol <11,12,16,17> or

- T is F(T1...Tn) and for all i=1...n, Ti is a semantically correct term, F is a function symbol with arity(F)=n <23> and sort(Ti) is a subsort of domainsort(F i) <81> or else F is an unknown symbol <21,22,24,28>.

An <atom> A is semantically correct iff

- A is a predicate symbol with arity(A)=Ø <34> or A is an unknown symbol <31,32,33,37> or

- A is P(T1...Tn) and for all i=1...n, Ti is a semantically correct term, P is a predicate symbol with arity(P)=n <34> and sort(Ti) is a subsort of domainsort(P i) <81> or else P is an unknown symbol <31,32,33,37> or

- A is T1 =*= T2, T1 and T2 are semantically correct terms and =*= is an <equality symbol>.

A <quantification> Q is semantically correct iff

- Q is ALL X... or EX X... and X is a variable symbol or an unknown symbol <51,52,53,55> and each atom in Q is semantically correct.

5 Notes on the PLL Compiler

5.1 Errors detected by the Compiler

The PLL compiler of the ATP system checks each input for syntactic and semantic correctness. An input containing signs which are not members of the terminal alphabet is responded to by a message

+++++ SYMBOL ERROR >>> xxx IS NO ADMISSIBLE SYMBOL

where 'xxx' is a sign which is not a member of the terminal alphabet.

For a syntactically incorrect input, the compiler responds

SYNTAX ERROR >>> xxx NOT ACCEPTED
UNEXAMINED REMAINDER OF THE INPUT >>> zzz

where 'xxx' is the sign which causes syntactic incorrectness and 'zzz' is the unanalysed remainder of the given input.

For a syntactically correct but semantically incorrect input, the compiler responds

***** SEMANTIC ERROR nnn >>> message
UNEXAMINED REMAINDER OF THE INPUT >>> zzz

where 'nnn' is the semantic error code (see sections 4.2, 5.2), 'message' is an error message explaining the kind of semantic error and 'zzz' is the unanalysed remainder of the given input.

5.2 Summary of Messages on Semantic Errors

Constant Symbols - see sections 4.2.2 and 4.2.4 :

```
***** SEMANTIC ERROR 11 >>>
FUNCTION SYMBOL x USED AS CONSTANT

***** SEMANTIC ERROR 12 >>>
PREDICATE SYMBOL x USED AS CONSTANT

***** SEMANTIC ERROR 14 >>>
CONSTANT SYMBOL x HAS RANGE s AND IS USED WITH RANGE t

***** SEMANTIC ERROR 16 >>>
SORT SYMBOL x USED AS CONSTANT

***** SEMANTIC ERROR 17 >>>
VARIABLE SYMBOL x USED AS CONSTANT
```

Function Symbols - see sections 4.2.2, 4.2.3 and 4.2.4:

```
***** SEMANTIC ERROR 21 >>>
CONSTANT SYMBOL x USED AS FUNCTION

***** SEMANTIC ERROR 22 >>>
VARIABLE SYMBOL x USED AS FUNCTION

***** SEMANTIC ERROR 23 >>>
m-ARY FUNCTION SYMBOL x USED WITH n ARGUMENTS

***** SEMANTIC ERROR 24 >>>
PREDICATE SYMBOL x USED AS FUNCTION

***** SEMANTIC ERROR 26 >>>
FUNCTION SYMBOL x (t1...tM) -> s APPLIED TO (s1...sN)

***** SEMANTIC ERROR 27 >>>
FUNCTION SYMBOL x (t1...tM) -> s USED WITH RANGE t
```

***** SEMANTIC ERROR 28 >>>
SORT SYMBOL x USED AS FUNCTION

Predicate Symbols - see sections 4.2.2, 4.2.3 and 4.2.4:

***** SEMANTIC ERROR 31 >>>
CONSTANT SYMBOL x USED AS PREDICATE

***** SEMANTIC ERROR 32 >>>
VARIABLE SYMBOL x USED AS PREDICATE

***** SEMANTIC ERROR 33 >>>
FUNCTION SYMBOL x USED AS PREDICATE

***** SEMANTIC ERROR 34 >>>
m-ARY PREDICATE SYMBOL x USED WITH n ARGUMENTS

***** SEMANTIC ERROR 36 >>>
PREDICATE SYMBOL x (t1...tM) APPLIED TO (s1...sN)

***** SEMANTIC ERROR 37 >>>
SORT SYMBOL x USED AS PREDICATE

Variable Symbols - see section 4.2.4:

***** SEMANTIC ERROR 51 >>>
CONSTANT SYMBOL x USED AS VARIABLE

***** SEMANTIC ERROR 52 >>>
FUNCTION SYMBOL x USED AS VARIABLE

***** SEMANTIC ERROR 53 >>>
PREDICATE SYMBOL x USED AS VARIABLE

***** SEMANTIC ERROR 55 >>>
SORT SYMBOL x USED AS VARIABLE

Sort Symbols - see sections 4.2.1 and 4.2.2

***** SEMANTIC ERROR 61 >>>
CONSTANT SYMBOL x USED AS SORT

***** SEMANTIC ERROR 62 >>>
VARIABLE SYMBOL x USED AS SORT

***** SEMANTIC ERROR 63 >>>
FUNCTION SYMBOL x USED AS SORT

***** SEMANTIC ERROR 64 >>>
PREDICATE SYMBOL x USED AS SORT

***** SEMANTIC ERROR 65 >>>
ATTEMPT TO ESTABLISH SORT SYMBOL x
AS A DIRECT SUBSORT OF ITSELF

***** SEMANTIC ERROR 66 >>>
SORT SYMBOL x IS NO DIRECT SUBSORT OF y

Terms and Atoms - see section 4.2.4

***** SEMANTIC ERROR 81 >>>
#n - ARGUMENT OF x HAS SORT s BUT IS USED WITH AN INCOMPATIBLE
t - SORT ARGUMENT

Acknowledgement

I would like to thank my colleagues S. Biundo, K.-H. Bläsius, N. Eisinger, A. Herold, D. Hutter, H.J. Ohlbach and J. Siekmann for their support during the preparation of this report.

References

[BES 81] Bläsius, K., Eisinger, N., Siekmann, J., Smolka, G., Herold, A., and C. Walther
The Markgraf Karl Refutation Procedure.
Proc. of the 7th International Joint Conference on Artificial Intelligence (1981)

[DMW 81] Dilger, W., Müller, J., and W. Womann
Einführung in die Markgraf Karl Refutation Procedure.
Interner Bericht 40/81, Fachbereich Informatik, Universität Kaiserslautern (1981)

[Ohl 82] Ohlbach, H.J.
The Markgraf Carl Refutation Procedure -
The Logic Engine
Interner Bericht 24/82, Institut für Informatik 1, Universität Karlsruhe (1982)

[Wal 82] Walther, C.
A Many-Sorted Calculus Based on Resolution and Paramodulation.
Interner Bericht 34/82, Institut für Informatik 1, Universität Karlsruhe (1982)