A computer assistant for a formal mathematical language : the construction of a user friendly assistant program to enter, check, store and view Weak Type Theory descriptions

Körvers, R.W.J.

Award date:
2002

TECHNISCHE UNIVERSITEIT EINDHOVEN

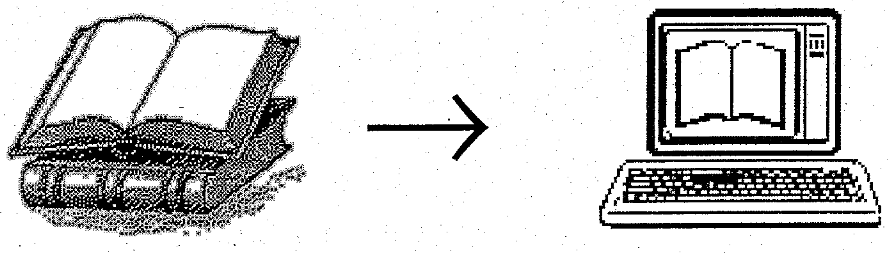
Department of Mathematics and Computing Science

MASTER'S THESIS

A computer assistant for a formal mathematical language

The construction of a user friendly assistant program
to enter, check, store and view
Weak Type Theory descriptions

by R.W.J. Körvers



Supervisor: dr. R. P. Nederpelt

Eindhoven, November 2002

Index

Foreword

This document is the master's thesis of Roel Körvers. The master's education is done at Eindhoven University of Technology. The thesis discusses the development of a computer program in which the user can enter a text in a formal mathematical language.
As formal mathematical language Weak Type Theory (WTT) is used. WTT is a formal mathematical language in which mathematical texts, that are written in a mixture of natural language and formula, can be translated. In this way a more structured and formal description of the original mathematical text is constructed.
All rules and checks of the WTT language are described in "CS Report 02-05 Weak Type Theory: A formal language for mathematics" written by R. P. Nederpelt [Ned 2002].
The computer program must check if the text entered is in accordance with the rules of the WTT language or must suggest changes to the text so that the text will be in accordance with the rules of the WTT language. The program must also keep an administration of the text entered and show it in the WTT format. Saving the text for later use must also be possible.
The program developed in the process of this master thesis is called the Weak Type Theory Assistant (WTTA).

The first chapter will explain why WTT is used and how this master thesis research is useful for WTT. This will be followed by a brief overview of WTT, including the derivation rules used for checking if a text is WTT correct. In chapter two we will discuss some restrictions and demands on the text entered as input for the program. A structure for modeling the input as well as a way for checking that structure, on syntax correctness and correctness with respect to the derivation rules, will be discussed. Furthermore this chapter will describe the core version of the WTTA program.

A user group has tested the core program and has requested all kinds of changes, these changes and their implementation in the program are described in chapter three.
The user group also requested some changes that were not implemented due to their low priority, low importance, high difficulty or lack of time. All suggested changes of the user group are described in chapter four.

Chapter five describes attention points that are important for developing a computer program for a formal mathematical language. This chapter is especially useful for readers who want to expand the current version of the WTTA program or want to develop a computer program for another formal mathematical language. The document will finish with some remarks of the author and the conclusions about the project.

The writer would like to thank the user group for their comments on the program. Without those comments the program would be a lot less user friendly, although I regret that not all suggested features could be implemented.
The user group consisted of: G. Jojgov, M. Franssen, M. Scheffer and R. Nederpelt.

Special thanks go to R. Nederpelt who supervised the entire process and pushed me into the right direction when I almost took a wrong turn. With M. Scheffer I discussed various aspects of WTT, which helped to refine some of the sections of this document. G. Jojgov made valuable comments about this thesis in an early stage, especially about the use of priority and associativity by infix constants. For assistance with the flag view of the program the author also likes to thank M. Oostdijk for granting him permission to look into the code of a program that translates Coq proofs into a flag notation of that proof.

Summary

In this master's thesis we describe the construction, of a prototype of an assistant tool that supports the user in entering a formal mathematical language into the tool. The formal mathematical language used is called Weak Type Theory (WTT).

The research started with the study of the WTT language described in CS-Report 02-05 [Ned 2002]. The language categories and the derivation rules of the WTT language had to be adapted to a form that could be used to develop a computer program. In this thesis it is argued that the adaptations made to the WTT rules were allowed, which ensures that the program still checks the WTT rules described in the report.
The main reason to slightly change the WTT rules was to achieve a clear separation between rules on the syntax of the WTT language and rules on the linguistic types of the WTT language.

The input of the program developed first, the core program, was restricted to ensure that only a small and unambiguous language could be entered. This language, called 'unsugared' language, is translated into an object structure, because checking an object structure is easier than checking an input line. This linguistic structure is checked on syntax correctness and linguistic type correctness by two small verifying programs. Small verifying programs are in accordance with the "de Bruijn criterion", which states that a verifying program should be very small; then this program can be checked by hand, giving the highest possible reliability. [Bar 2001].

This 'unsugared language', however, is not easy to read or write. Therefore a second program was developed, the final program, that accepted a more user friendly input.
Developing this program came down to the replacement of a small part of the core program. The difference between checking the input of the final program and the input of the core program is only the parsing of the input and the construction of the object structure. Working with an object structure assured that the verifying programs were not changed significantly and therefore stayed very small.

Besides the addition of the user friendly language, the final program was also extended with different ways to represent the entered text, also the user gained options to manipulate the text entered into the final program. However, still a lot of options to extend the program and to increase the usability of the program were not implemented, due to low priority, low importance or high difficulty. If more time had been available for this research, more of these options could have been implemented.

Samenvatting

Dit document beschrijft de constructie van een prototype dat de gebruiker assisteert om een formele wiskundige taal op correcte wijze in te voeren in een computer. De formele wiskundige taal die gebruikt wordt door het programma is Weak Type Theory (WTT).

Het onderzoek begon met het bestuderen van de WTT-taal die beschreven wordt in CS-Report 02-05 [Ned 2002]. De taalcategorieën en de afleidingsregels van de WTT-taal moesten worden aangepast om gebruikt te kunnen worden voor het ontwerpen van een computerprogramma.
In dit verslag wordt beargumenteerd waarom deze aanpassingen toegestaan zijn. Hierdoor kan worden verzekerd dat de regels van het rapport ook door het programma gecontroleerd worden.
De belangrijkste reden waarom de regels in het rapport werden aangepast was de wens om een duidelijke scheiding aan te brengen tussen de regels die de structuur van de taal vastleggen en de regels die de taalkundige typen van de taal bepalen.

De invoer die bij het eerst ontworpen programma, het kernprogramma, werd toegelaten, was beperkt door een kleine, ondubbelzinnige taal. Deze taal, genoemd 'ongesuikerde' taal, wordt vertaalt naar een objectenstructuur. Controleren van deze objectenstructuur is gemakkelijker dan het controleren van een regel tekst. De objectenstructuur moet voldoen aan de structuur regels en taalkundige type regels van WTT-taal, wat gecontroleerd wordt door twee kleine controleprogramma's. Op deze manier wordt aan het "de Bruijn-criterium" voldaan, dat zegt dat een controleprogramma klein moet zijn; het programma kan dan met de hand op correctheid gecontroleerd worden, wat het grootst mogelijk vertrouwen geeft [Bar 2001].

Deze 'ongesuikerde' taal is echter niet erg leesbaar en schrijfbaar. Daarom werd er een tweede programma ontworpen, het uiteindelijke programma, dat een meer gebruiksvriendelijke invoer toelaat. Het ontwerpen van dit programma kwam neer op het vervangen van een klein gedeelte van het kernprogramma.
Het uiteindelijk programma verschilt, wat het controleren van de invoer inhoudt, dan ook alleen van het kernprogramma door de manier waarop de invoer wordt gelezen en de manier waarop de objectenstructuur wordt geconstrueerd. Het gebruiken van de objectenstructuur garandeert dat de controleprogramma's niet noemenswaardig veranderen, zodat de controleprogramma's klein blijven.

Behalve het toelaten van een gebruiksvriendelijke taal, werd het uiteindelijk programma uitgebreid met verschillende manieren waarop de ingevoerde tekst kon worden weergegeven en gebruikersopties om de tekst op verschillende manieren te bewerken. Naast deze uitbreidingen zijn er ook nog andere uitbreidingen mogelijk die de gebruiksvriendelijkheid zouden hebben verhoogd.
Deze werden echter niet geïmplementeerd door een lage prioriteit, lage belangrijkheid of hoge moeilijkheid. Indien er meer tijd was geweest voor het onderzoek zouden er meer uitbreidingen geïmplementeerd zijn.

1 Introduction to Weak Type Theory

The concept of the Weak Type Theory (WTT) Language is described in CS-Report 02-05 [Ned 2002]. In this introduction we position WTT on the path between a description of a mathematical content and the formal correctness proof of that mathematical content. We will also position this research on that path and describe how the research helps to cross the path. Thereafter the structure and the linguistic types of the WTT language will be described. The linguistic types are checked by derivation rules, which are discussed in section 1.4.
Some definitions of the WTT language differ from the original WTT language, these differences are described and justified in the last section of this chapter.

1.1 Why WTT

To visualize the path from a mathematical description to the correctness proof, we take a river and its two shores. On one shore we have the (mathematical) description and on the other shore is the proof (of correctness of that mathematical description). To construct the correctness proof we need to cross the gap between the two shores (depicted by a river in figure 1.1a).

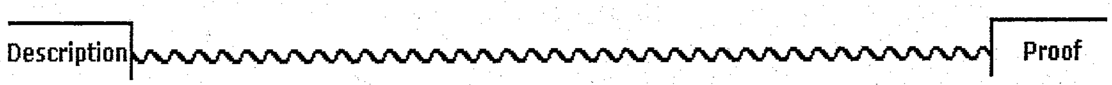


figure 1.1a

Some people, who are experts in writing correctness proofs, can easily jump across the river, in other words, give a complete correctness proof of the description. However, most people cannot construct a correctness proof that easily. By formalizing the problem, the distance between the informal description and the correctness proof is decreased. The informal description is translated in a formal language description and the task that is left is to prove the formal language description. The use of a formal language safeguards for an incomplete description, it also guarantees a basic form of correctness.

Type Theory (TT) is such a formal language. The advantage of TT is that there are a number of programs, which aid users in proving a part of the description in TT. Coq and LEGO are examples of such programs, called theorem provers. Adding Coq, LEGO and TT in our visualization from the path from description to proof results in figure 1.1b.

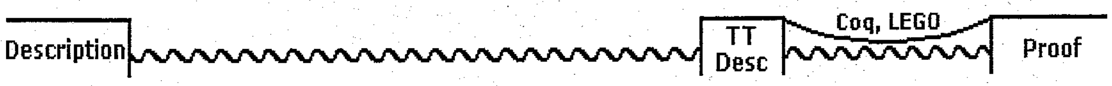


figure 1.1b

The TT description is a stepping stone between the informal description and the proof. Theorem provers are tools for proving a TT description. We depict them as a bridge between the TT description and the correctness proof. The TT description can also be proved by using pen and paper instead of using the theorem provers. This would be equivalent to jumping from the TT description to the formal proof, instead of using the bridge.

The distance between the informal description and the TT description is still very difficult to cross, but easier than proving the informal description without first formalizing to TT.
To further reduce the distance between the informal description and the TT description the idea of another stepping stone is introduced. This stepping stone is called Weak Type Theory (WTT). WTT, just like TT, formalizes the informal description but is not as strict as TT, which enables WTT to remain closer to the informal description. The WTT description is added to our path from informal description to correctness proof, which is depicted in the next figure 1.1c.

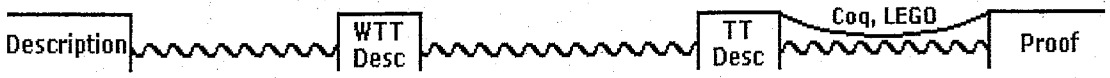


figure 1.1c

If we look at figure 1.1c we still have two gaps to cross, where we must jump to get to the next stepping stone in the path from a informal description to the formal correctness proof. This research is about the process of creating a program that helps users to translate the informal description into a WTT description.

The prototype program that was developed during this research, is the Weak Type Theory Assistant (WTTA). The WTTA program covers the gap between the informal description and the WTT translation of that description. WTTA is like Coq and LEGO a computer program that helps a user. Coq and LEGO help by proving a TT description, WTTA helps by the formalization process, therefore WTTA is also depicted as a bridge in figure 1.1d.

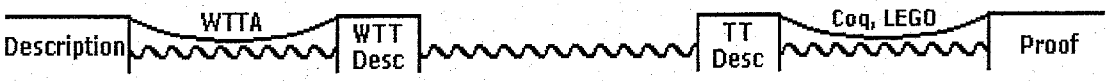


figure 1.1d

The last gap, between the WTT description and the TT description is material for future work. Probably another program must be developed that can help to translate a WTT description into a TT description. It is also possible that another stepping stone needs to be introduced, between the WTT description and the TT description.
Looking into these possibilities is outside the scope of this research.

1.2 The syntax of WTT

In this section we discuss the syntax of the WTT language by introducing different categories and different levels in which those categories can be divided. The categories and levels are depicted in table 1.2a. To clarify the syntax of the categories some examples of each category will be given. The categories discussed in this thesis differ from the categories discussed in CS-Report 02-05 [Ned 2002]. Reasons for and justification of these differences are discussed in section 1.5.

Atomic level
Every language can be divided into smaller parts, and finally into atomic parts. The natural language for example can be divided in verbs, adverbs etc. The WTT language has three atomic parts: variables, constants and binders. These three parts are represented by the first three categories of the WTT language, together these three categories form the atomic level.

<u>Variable and constant</u>
Variables and constants are used in WTT to give a name to a subject for discussion purposes. Variables represent indefinite objects of a certain type, whereas constants refer to well-defined entities.
Examples of variables are 'x', 'n', 'a', 'b', examples of constants are 'Plus(x, y)', 'Not(b)'.

The syntactical difference between constants and variables is that constants have a parameter list[1].When defining a constant, the parameter list consists of the variables on which the constant depends. When using a constant the parameters are instantiated with specific entities.
For example, 'Plus(x, y)' depends on two variables 'x' and 'y', belonging to a certain class (which has to be known on beforehand). An instantiation of 'Plus(x, y)', with 'x' and 'y' from the natural numbers class, could be 'Plus(3, 5)'. '3' and '5' take the place of 'x' and 'y'.

[1] Constants can have a parameter list of length 0, E.g. 'π' or '$\pi()$', depending on the language definition (see section 2.2).

Binder
Apart from variables and constants there are binders. A binder binds a free variable in an expression, by giving the name of that variable and its type in a subscript. A well known binder is the '$\forall$'. For example: '$\forall_{x:Nat}$ (x/1=x)'.

In WTT all binders and some constants are known in advance. These binders and constants are placed in the, so called, preface and can be used in the WTT text. Binders cannot be defined in the WTT text, but only be given in the preface. The preface only stores the information needed to check if the constant or binder is in accordance with the rules of the WTT language. For more information about the preface we refer to CS-Report 02-05 [Ned 2002].

Utterance level
Categories from the utterance level are used to express information about the categories of the atomic level. This can be done by introducing new variables or new constants, or by stating some property of a category from the atomic level. Some special variables, constants and binders can be used to state these properties.

Declaration
Variables cannot be declared in the preface but must be declared in the WTT text.
As we already saw with binders the ':' symbol is used as declaration symbol. A declaration contains, besides the declaration symbol, the variable to be declared and its set type, e.g. 'n : Nat'. Variables can also be declared as elements of the class of all sets, represented as '*s' or 'SET', or as elements of the class of all statements, represented by '*p' or 'PROP'.[2]
In WTT 'SET' and 'PROP' are used to declare variables of respectively the class of all sets and the class of all statements. For example 'U : SET', 'a : PROP'.

Definition
To define a constant, a name of the new constant, including the parameter list, is given. The constant is defined as being equivalent to a formula that, among other defined constants and binders, uses the parameters of that constant. The definition uses the ':=' symbol, to separate the constant to be defined and its equivalent formula. In this way the definition of a constant that represents the double amount of a given number can be written as : 'Double(x) := Plus(x, x)'.
To define the set of the natural numbers greater than ten we can write: 'Nat^{+10} := $Set_{n:Nat}$ ($n \geq 10$)'.

Statement
The statement category is used to state properties about categories of the atomic level. Statements are constructed by using variables, constants or binders. In general, statements are used as assumptions on variables or to state conclusions after a derivation.
For example: the expression '$n \geq 10$' is a statement, which can be an assumption on n.
'$\forall_{x:Nat}$ (x/1=x)' uses the $\forall$ binder to state a fact about all natural numbers.

[2] The declaration of variables as element of the class of all statements is missing in CS-Report 02-05 [Ned 2002]

Discourse level
The discourse level combines the categories of the utterance level in the same way as the categories of the utterance level combine the categories of the atomic level. The Discourse level is the highest level of the WTT language.

<u>Context</u>
A context is a list of declarations and statements. The statements used in the context are to be interpreted as assumptions about the variables that are declared in the declarations. The declarations and statements in the context are separated by the ',' symbol.
An example of a context could be: 'x : Nat, y : Nat, y < x', where the first two items in the context are declarations, while the third item in the context is a statement.
A context is used to introduce a list of variables, together with a number of assumptions about those variables, that are used in a line. Which brings us to the next category.

<u>Line</u>
A line is a context combined with either a definition, or a statement. A constant defined under a context always has the variables declared in that context in its parameter list. The variables in the parameter list appear in the same order as they were declared in the context.
A statement behind a context can be a theorem or an assertion holding in that context, but never an assumption.
In a line the context is separated from the definition or statement with the '⊳' symbol.
Two examples of a line are written below.
'x : Nat, y : Nat, y > 0 ⊳ Div(x, y) := Round_Down(x / y)' (context with a definition)
'x : Nat, x > 10, y : Nat, y < 10 ⊳ y < x' . (context with a statement)

<u>Book</u>
All lines constructed, as described above, are put together in a list. This list, in WTT, is called a book. Lines in the book are separated by line breaks. In the derivation rules of section 1.4, the '○' symbol is used for connecting a book to a line.

To summarize we give the following overview.

LEVEL	CATEGORY	DESCRIPTION
Atomic level	variable constant binder	The cornerstones of the WTT language
Utterance level	declaration definition statement	Combining the cornerstones to declare, define them and to state assumptions and assertions
Discourse level	context line book	Combining elements from the utterance level and other elements of the discourse level

table 1.2a

This concludes our description of the syntax of the WTT language. In the next chapter linguistic types are introduced and added to the syntax, which leads to some restrictions and other additions to the language.

1.3 The linguistic types of WTT

In this section we discuss the linguistic types (Ltypes), which are an essential feature of WTT. Restrictions on the structure of a formalized text are introduced by establishing the Ltypes of expressions. These restrictions are only on the linguistic level, hence still (very) 'weak'. However, it gives the guarantee that expressions obey (at least) some natural linguistic requirements, thus fit in the format of a language. The derivation rules describing the checks on the linguistic level are described in section 1.4.

Ltypes

The WTT language has five weak types that are added as meta-categories on top of the syntax discussed in section 1.2.

Terms	A Term is the basic type for the smallest entities. Declared variables (of certain set type) have Ltype Term. Constants defined with these variables can have Ltype Term like 'Plus(x,y)' or 'Plus(3,5)'. Even expressions starting with a binder can have Ltype Term, such as expressions beginning with the 'Σ' binder.
Sets	Collections of entities belong to the Ltype Set. A variable can have Ltype Set by declaring it as an element of the class of all sets. Constants with Ltype Set are for instance Nat, Real, while a binder as 'Set' also constructs expressions with Ltype Set.
Nouns	Nouns are used for the same purpose as nouns in natural language. Words such as 'a triangle' or 'a point' have Ltype Noun.
Adjectives	Adjectives are used for the same purpose as adjectives in natural language. Words with an obvious meaning like 'odd' or 'convergent' have Ltype Adj.
Stats	This Ltype is used for propositions. The statements discussed in section 1.2 have Ltype Stat.

The Ltypes adjective and noun

Adjectives and nouns are meant for mimicking categories present in the natural language. In this way the WTT text can be closer to the text in natural language. But to define constants like 'odd' or 'a singleton' special binders need to be introduced.

The Adj binder constructs an adjective by using a statement.

'odd' can now be defined as: 'odd := $\text{Adj}_{n:\text{Nat}}$ (n mod 2 = 1)'

The Noun binder constructs a noun by using a statement.

'a singleton' can now be defined as 'a singleton := $\text{Noun}_{V:\text{SET}}$ ($|V|=1$)

Furthermore the Abst binder is introduced to abstract a noun from a term, set or noun. This binder is used for translating texts like 'for some element of Nat', 'for some interval' or 'for some odd natural number'.
For more details and examples we refer to CS-Report 02-05 [Ned 2002]

We also have two special constants that help us in the use of nouns. If we have a set, say 'Nat', and we would like to have an arbitrary element from that set, 'a natural number'[3], then we push the set down to a noun with the '$\downarrow$' constant which constructs a noun from a set: '(Nat)$\downarrow$ = a natural number'. The other way around, a noun like 'a natural number' can be lifted to its corresponding set: '(a natural number)$\uparrow$ = Nat'.

[3] In this text we choose to consider the words 'a natural number' as a single noun.

Adjectives and nouns can, just as in natural language, be combined. Texts like 'an odd natural number' or 'a convergent series' are now also permitted in the language. This adjective noun combination can also consist of multiple adjectives followed by one noun, for example 'a positive odd natural number'. But regardless how many adjectives are added ('positive', 'odd'), the entire text says something about the noun ('a natural number'), therefore the complete adjective noun combination has Ltype noun.

This adjective-noun combination cannot be constructed by only using the categories described in the previous section. Therefore we add a category which we call 'Combination'. This combination category consists of a list of constants where all constants have Ltype Adj, except the last constant which must have Ltype Noun. The combination category is not a cornerstone of the language. It is possible to state things about 'an odd natural number' or declare a variable to be 'an odd natural number', but 'an odd natural number' on its own does not provide any information, therefore it is also not an utterance.
To complete the overview given in table 1.2a we place the combination category on a different level between the atomic and the utterance level.

The basic Ltypes of the linguistic categories
After adding this category we can determine the possible Ltypes for all the categories.

Variable
As we already described in the previous section, variables can be declared in three different ways.
1) A variable of a certain set type. Variables declared in this way have Ltype Term. They represent an element of that set type.
2) Variables declared as element of the class of all sets, these variables have Ltype Set.
3) Variables declared as element of the class of all statements, these variables have Ltype Stat.

Therefore variables can only have Ltypes Term, Set or Stat.

Constant
Constants can be given in the preface, in which case their Ltype must be given in advance, or can be defined in the WTT text. In the latter case the Ltype of the constant depends on the Ltype of the right hand side of the definition. The constant is defined as having that same Ltype.

Binder
Binders can only be defined in the preface, hence their Ltypes are already known.

Combination
Combinations always have Ltype Noun as discussed when introducing the adjective noun combination.

Declaration and definition
Declarations and definitions do not have one of the basic five Ltypes, they respectively introduce a fresh variable in the context or introduce a fresh constant in the book.

Statement
Statements always have Ltype Stat.

Context, line and book
Contexts, lines and books also have no basic Ltype in the form of Term, Set, Noun, Adj or Stat. But by looking at a text the category of that text can be determined.

Introducing extra Ltypes
The goal is to proof that a certain text is in accordance with the rules of the WTT language and therefore can be represented by a book. These rules define which checks must be performed, before a text can be judged to be correct WTT.
In order to express that a text can be represented by a correct book, comes down to dividing the text into lines and check that those lines have the form of correct WTT lines. A correct WTT line consists of a correct context combined with a correct definition or correct statement. Hence, extra Ltypes are needed to refer to correct books, lines etc..

We introduce extra Ltypes for the categories that do not have a basic Ltype.
These extra Ltypes are Decl, Def, Cont, Line, Book and they respectively represent the declaration, definition, context, line and book categories.
These categories can be checked by verifying their parts.
A book must consist of correct lines (Line), while a line must consist of a correct context (Cont), combined with a correct statement (Stat) or definition (Def). The context must consist of correct declarations (Decl) and/or Statements (Stat).

The basic Ltypes and the extra Ltypes introduced here are used in the derivation rules of section 1.4.

1.4 Linguistic type checking : derivation rules

The rules in this chapter describe which checks need to be done before a given input can be judged to be correct WTT. The rules will be described first, thereafter a mathematical representation of that rule will be given. These rules slightly vary from the derivation rules described in CS-Report 02-05 [Ned 2002], conform to the differences described in the previous section.

Introduction orthography
Our goal is to prove that a book is a correct WTT book, in order to check a book we use the derivation rules. The derivation rules make judgments on three different levels.

On the highest level they judge if a book B is a correct book
$\vdash^0$ B : **Book**
On the middle level they judge if a context Γ is correct relative to the book B
B $\vdash^1 \Gamma$: **Cont**
On the lowest level they judge if a text is WTT correct relative to the book B and the context Γ
B; $\Gamma \vdash^2$ t : **Term**, B; $\Gamma \vdash^2$ s : **Set**, B; $\Gamma \vdash^2$ n : **Noun**, B; $\Gamma \vdash^2$ a : **Adj**, B; $\Gamma \vdash^2$ S : **Stat**,
B; $\Gamma \vdash^2$ d : **Decl**, B; $\Gamma \vdash^2$ D : **Def**, B; $\Gamma \vdash^2$ L : **Line**,

These judgment levels are linked together, for the middle level we need the book to be correct and for the lowest level we need that the context is correct relative to the book and that the book is correct. These conditions are abbreviated in the derivation rules.
OK(B) stands for B is a correct book, or $\vdash^0$ B : **Book.**
OK(B; Γ) stands for B is a correct book and Γ is correct relative to book B,
or $\vdash^0$ B : **Book** and B $\vdash^1 \Gamma$: **Cont**.

Furthermore, we will use
V, C, B to respectively represent all possible variables, constants, binders.

Derivation rules

We link the derivation rules, and their descriptions, to the categories, mentioned in table 1.2a, including the combination category mentioned in section 1.3, the Ltypes adjective and noun. The order in which the derivation rules are discussed is based on the levels and categories of the WTT language.

Variable

Checking a variable to be of a certain Ltype is a judgment on the lowest level, therefore the context needs to be correct relative to the book and the book must be correct. We will not mention this again in the description of the other judgments of the lowest level, but the OK(B; Γ) check is depicted as a premisse in the derivation rules.
The entity x must be declared in a part of the context, furthermore x must be a variable.
The Ltype of x depends on how x is declared inside Γ.
x has Ltype Term if x is declared in the context as an element of a set type.
x has Ltype Set if x is declared in the context as a member of the class of all sets.
x has Ltype Stat if x is declared in the context as a member of the class of all statements.
If these premisses are correct we can conclude that x is a correct variable of certain Ltype.

$$\frac{\begin{array}{c}\text{OK}(B;\Gamma),\ x : t \text{ part of } \Gamma,\ x \in V, \\ t\textbf{: Set} \text{ or } t\textbf{ : Noun} \ / t = \text{'SET'}/t = \text{'PROP'}\end{array}}{B;\Gamma \vdash^2 x : \textbf{Term/Set/Stat}} \qquad \text{(var)}$$

This derivation rule is depicted as one rule, while in reality it represents three rules.
- The variable has Ltype Term, when the book and the context are correct and when the variable is declared, in the context, as an element of set type t, where t has Ltype Set or Ltype Noun.
- The variable has Ltype Set, when the book and the context are correct and when the variable is declared, in the context, as a member of the class of all sets.
- The variable has Ltype Stat, when the book and the context are correct and when the variable is declared, in the context, as a member of the class of all statements.

The '/' symbol is used as a abbreviation for combining these cases in one derivation rule.

Constant

Constants can be defined at two places, in the preface and in a line from the book. Therefore the constants have two derivation rules. In fact, constants have ten derivation rules, but the '/' symbol combines them into two different rules, ranging over all five basic Ltypes.
Later on the rule for constants defined in the book will be left out. This can be done, because when a constant is defined in a line, only the information needed to check if that constant is correct is stored. This is exactly the information stored in the preface, therefore we can use the rule for constant defined in the preface to verify these constants. In this overview both constant derivation rules are given.

To check a constant that is defined in the book we search the line in which the constant c was defined. c must be a constant. The constant c is defined relative to context B′ and Γ′.
The number of variables declared in context Γ′ must be the same as the number of parameters of c. All parameters must have, in the same order, the same Ltype as the variables declared in context Γ′. The right hand side from the definition of c must have the requested Ltype.

$$\frac{\begin{array}{c}\text{OK}(B;\Gamma),\ B' \circ \Gamma' \rhd c(x_1,\ldots,x_n) := t \text{ initial part of } B,\ c \in C, \\ \Gamma' \text{ declares exactly } x_1,\ldots,x_n,\ B;\Gamma \vdash^2 A_i : \textbf{W} \text{ if } B;\Gamma' \vdash^2 x_i : \textbf{W}\ (i = 1,\ldots,n), \\ B';\Gamma' \vdash^2 t : \textbf{Term/Set/Noun/Adj/Stat}\end{array}}{B;\Gamma \vdash^2 c(A_1,\ldots,A_n) : \textbf{Term/Set/Noun/Adj/Stat}} \qquad \text{(int-cons)}$$

For constants defined in the preface of book B the Ltypes of the parameters and the Ltypes of the constants themselves are known in advance. Checking the parameters and getting the Ltype of the constant is done by reading information from the preface. Reading information from the preface is done by using two functions.

- The in(x) function returns the list, of Ltypes, of the parameters of x.
- The out(x) function returns the Ltype of x itself.

The entity x can be a constant or binder that is defined in the preface.

$$\frac{\text{OK}(B;\Gamma),\ c \text{ in preface of } B,\ c \in \mathcal{C},\ \text{in}(c) = (\mathbf{T_1}, \ldots, \mathbf{T_n}),\quad B;\Gamma \vdash^2 A_i : \mathbf{T_i}\ (i = 1, \ldots, n),\ \text{out}(c) = \mathbf{T}}{B;\Gamma \vdash^2 c(A1, \ldots, An) : \mathbf{T}} \qquad \text{(ext-cons)}$$

<u>Binder</u>

Binders can only be defined in the preface, so the derivation rule for binders looks like the derivation rule for constants defined in the preface. The only difference is that binders have a declaration of a bound variable. This declaration must be checked on correctness before the parameters of the binder can be checked on correctness. The bound variable can only be used in the parameters of the binder and therefore the declaration must be correct before the parameters can be checked.

$$\frac{\text{OK}(B;\Gamma),\ b \text{ in preface of } B,\ b \in \mathcal{B},\ B;\Gamma \vdash^2 Z : \mathbf{Decl},\quad \text{in}(b) = (\mathbf{T_1}, \ldots, \mathbf{T_n}),\ B;\Gamma, Z \vdash^2 A_i : \mathbf{T_i}\ (i = 1, \ldots, n),\ \text{out}(b) = \mathbf{T}}{B;\Gamma \vdash^2 b_Z(A_1, \ldots, A_n) : \mathbf{T}} \qquad \text{(bind [4])}$$

<u>Combination</u>

The first object in the combination category must have Ltype Adj. All other entities after that first entity, can again be an adjective noun combination. From these entities again the first entity must have Ltype Adj. If only one entity remains, the adj-noun derivation rule cannot be used. The last remaining entity must be checked to have Ltype Noun, by one of the constant derivation rules. In conclusion, a combination entity consists of a list of entities, where the last entity has Ltype noun and all preceding entities have Ltype Adj.

$$\frac{\text{OK}(B;\Gamma),\ B;\Gamma \vdash^2 n : \mathbf{Noun},\ B;\Gamma \vdash^2 a : \mathbf{Adj}}{B;\Gamma \vdash^2 a\ n : \mathbf{Noun}} \qquad \text{(comb) or (adj-noun)}$$

<u>Declaration</u>

WTT has three possible declarations. One for declaring a variable and its type, one where the variable is declared as an element of the class of all sets and one where the variable is declared as an element of the class of all statements. The last two cases are very similar and are expressed in one derivation rule.

By a declaration of an x as an element of the class of all sets or statements x needs to be checked to have the form of a variable and whether the name x is not already in use by another variable.

$$\frac{\text{OK}(B;\Gamma),\ x \in \mathcal{V},\ x \text{ is a fresh variable}}{B;\Gamma \vdash^2 x : \text{'SET'} / \text{'PROP'} : \mathbf{Decl}} \qquad \text{(set/stat-decl)}$$

[4] The binder derivation rule is different from the binder rule mentioned in CS-Report 02-05 [Ned 2002]. This is done for the Lim binder which besides the range of the local variable also needs a direction and a number to which the variable is limited. For example, for a limit we have to indicate whether x is a natural or a real number, x is converging to 0 or to 1 and whether x runs to the limit from above or from below. Clearly the Lim binder has three parameters.

Checking a declaration of an x and its set type is done by again checking that x has the form of a variable and that the name x is fresh. In this declaration a check on the right hand side is also needed. The right hand side of the declaration must have Ltype Set or Noun.

$$\frac{\text{OK}(B;\Gamma),\ x \in V,\ x \text{ is a fresh variable},\ B;\Gamma \vdash^2 s/n : \textbf{Set/Noun}}{B;\Gamma \vdash^2 x : s/n : \textbf{Decl}} \qquad \text{(term-decl)}$$

Definition

To check a definition in WTT we have to check that c is a constant and that c is not yet defined in the preface or in the book. In other words c must be fresh. All variables declared in the context must be defined, in the same order, as parameters of c. The right hand side of the definition must have a valid (basic) Ltype.

$$\frac{\begin{array}{c}\text{OK}(B;\Gamma),\ c \in C,\ c \text{ is a fresh constant},\ \Gamma \text{ declares exactly } x_1,\ldots,x_n,\\ B;\Gamma \vdash^2 t/s/n/a/S : \textbf{Term/Set/Noun/Adj/Stat}\end{array}}{B;\Gamma \vdash^2 c(x_1,\ldots,x_n) := t/s/n/a/S : \textbf{Def}} \qquad \text{(int-def)}$$

Statement

As described in CS-Report 02-05 [Ned 2002], there is no derivation rule for the statement category. The derivation rules already given suffice for checking the statement category. (see CS-Report 02-05 [Ned 2002])

Context

In order to check the context of a WTT line we need a number of rules. We first need a rule that says that an empty context is correct. The check than comes down to checking the book.

$$\frac{\text{OK}(B)}{B \vdash^1 \varnothing : \textbf{Cont}} \qquad \text{(emp-cont)}$$

Furthermore a context consists of declarations or statements, so we need a rule that refers to the already defined declaration and statement rules.

$$\frac{\text{OK}(B,\Gamma),\ B;\Gamma \vdash^2 S/d : \textbf{Stat/Decl}}{B \vdash^1 \Gamma, S/d : \textbf{Cont}} \qquad \text{(assump/decl-cont)}$$

Line

A line has a similar rule. The line cannot be empty, but on the right hand side of the line we also have two choices which refer to already defined derivation rules.

$$\frac{\text{OK}(B,\Gamma),\ B;\Gamma \vdash^2 S/D : \textbf{Stat/Def}}{B \vdash^1 \Gamma \rhd S/D : \textbf{Line}} \qquad \text{(assert/def-line) or (book-ext)}$$

Book

Which brings us to the last category that needs to be checked, the book. We first need a rule that says that the empty book is correct, otherwise we could not start with a correct WTT text.

$$\frac{}{\vdash^0 \varnothing : \mathbf{Book}} \qquad \text{(emp-book)}$$

Then we want to expand the book, but we may only expand the book with a correct line. Therefore we need a derivation rule that just is a reference to the line derivation rule.

$$\frac{OK(B), B \vdash^1 L : \mathbf{Line}}{\vdash^0 B \circ L : \mathbf{Book}} \qquad \text{(line-book)}$$

Notes

- In the emp-cont derivation rule and the line-book derivation rule the context is not checked, but only the book is checked.
  - In the emp-cont derivation rule the context is empty
  - In the line-book derivation rule the context is part of the line entity.
- The emp-book derivation rule does not check the context or the book, because the book is empty and therefore there is no context.

This completes the overview of the WTT language. We discussed the structure and the linguistic typing system of WTT. CS-Report 02-05 [Ned 2002] was used as a guideline, but we deviated from the definitions in that report. Section 1.5 explains the differences and argues that these differences are allowed.

1.5 Differences with original WTT

The computer program constructed during the research described in this thesis, checks if the input entered complies to the WTT rules described in CS-Report 02-05 [Ned 2002]. However in the introduction of the WTT language we already deviated from the WTT language definition of the report[5].

The main reasons to deviate were :

I) To distinguish more clearly between linguistic types and language categories.

II) To separate declarations from contexts, and separate lines from books.

The differences between the report and this thesis can be divided into differences in the syntax definition, categories and levels, and differences in the Ltype definition, including the derivation rules. These differences will be discussed separately. It will be argued that these differences are allowed and still describe the WTT language defined in the report.

[5] In section 1.5 'the report' refers to CS-Report 02-05 [Ned 2002]

Differences in the syntax definition

In the report the utterance level is replaced by two other levels, the phrase and sentence level.

Phrase level

The phrase level in the report consists of four categories, the term, set, noun and adjective categories.

term	The term category is the collection of all variables, constants and binders of Ltype Term.
set	The set category is the collection of all variables, constants and binders of Ltype Set.
noun	The noun category is the collection of all constants and binders of Ltype Noun and all adjective noun combinations, which only can have Ltype Noun
adjective	The adjective category is the collection of all constants and binders of Ltype Adj.

With these four categories it is possible, in the report, to talk about all terms, meaning all variables, constants and binders that have Ltype Term. In fact, in the report, the section about the syntax of the phrase level refers to the corresponding sections about variables, constant and binders. Ltype checking the term category is done by calling the appropriate variable, constant or binder derivation rule.

In this thesis we try to separate the Ltypes as much as possible from the categories, therefore we did not introduce the categories of the phrase level. The entities in these categories are also part of a category on the atomic level.
The only entities not part of a category on the atomic level are the adjective noun combinations, but we introduced the combination category to represent those entities.

Sentence level

The sentence level in the report consists of two categories, the statement category and the definition category.

statement In the report statements are divided in :
- variables, constants or binders that have Ltype Stat.
- typing statements.

definition For defining new constants in a WTT book.

In this thesis we introduced two different categories, instead of dividing the statement category as is done in the report.
- The declaration category : typing statement
- The statement category : variables, constants or binders that have Ltype Stat

The definition category in the report is the same as the definition category in this thesis.

Despite the wish to separate Ltypes from the categories we still need the statement category to define the WTT language, this also explains why in the report the statement category has a different level than the term, set, noun and adjective categories.

Differences in the Ltype definition
The separation of the statement defined in the report introduces the use of the extra Ltype Decl and by more clearly separating the line category from the book the extra Ltype Line was introduced. Those Ltypes are not used in the report.

The introduction of extra Ltypes Decl and Line separates the derivation rules for context and book as described in CS-Report 02-05 [Ned 2002]. The assump/decl cont derivation rule and the line-book derivation rule refer in this thesis to the derivation rules for respectively extending the context and extending the book.
This is done to separate the derivation rule for declarations and lines from the derivation rules of contexts and books, which is also in accordance with the syntax definition.
- A context consists of declarations and statements (assump).
- A line consists of definitions and statements (assert).
- A book consists of lines.

Besides the extra Ltypes are not needed to verify correctness of a text. Only checks on the basic Ltypes, Term, Set, Noun, Adj and Stat, are needed to check a text on WTT correctness. Checks on extra Ltypes can be reduced to checks on the basic Ltypes as described in the next five steps:

a) In the premisses of the var, int-cons, ext-cons, bind, comb, set/stat-decl, term-decl, int-def derrivation rules, only checks on basic Ltypes appear. Except for OK(B, Γ).
b) In the premisses of the assump/decl-cont and assert/def-line derivation rules, basic Ltype checks and checks on Decl, Def Ltypes appear. Checks on Decl, Def Ltypes refer to derivation rules mentioned under (a) and therefore can be translated to checks on basic Ltypes. The OK(B, Γ) check also appears here.
c) In the premisses of the line-book derivation rule, OK(B, Γ) and a check on Ltype Line appears. A check on Ltype Line refers to the assert/def-line derivation rule mentioned in case (b).
d) OK(B, Γ) checks the context Γ under the book B and B is a correct book (or OK(B)). All entities in the context are checked by the assump/decl-cont derivation rule (see b), till there are no entities left in the context, which is checked by the emp-cont derivation rule. The emp-cont derivation checks OK(B).
e) OK(B) checks the book. All entities in the book are checked by the line-book derivation rule (see c), till there are no entities in the book, which is checked by the emp-book derivation rule. The emp-book derivation rule has no premisses.

Therefore the use of two extra Ltypes does not invalidate the way in which a text is verified to be a correct WTT book, as is demonstrated by these five steps.

Other differences in the derivation rules are:

1) The Stat Ltype in the var derivation rule, as mentioned in the second footnote.

2) The bind derivation rule, as mentioned in the fourth footnote.

3) The order in which the checks are written.
   The order in this thesis corresponds to the way the computer program will check for WTT correctness. For example a definition will be checked by first checking the left hand side of the definition and then checking the right hand side of the definition.
   Therefore, the freshness of the constant and the parameters of the constant are checked before the Ltype of the right hand side is checked.
   This order is not important in the derivation rules, because all premisses must be checked before a conclusion can be drown. The order is only important for the program.

4) Another difference is caused on account of the introduction of the term, set, noun and adjective categories in the report. In the derivation rules of the report $V^{t/\sigma/s}$ and $C^{t/\sigma/\nu/\alpha/s}$ can be uses as premisses, because variables and constants are divided in disjoint subsets, where all elements of a subset have the same Ltype.
   In this thesis we had to use that Ltypes of variables, declared in the context, and Ltypes of constants, defined in the book, can be looked up by examining, respectively, the declaration of that variable or the definition of that constant.

2 WTT on the computer

In the previous chapter we introduced WTT and explained the rules to which a text must obey, before it can be judged to be WTT correct. In this chapter we begin with restricting the input that will be used by the program. This restriction is based on a decision about how the program will be used and on the input that is possible with a keyboard.

After restricting the input we define the language which will be used by our first program, the core program. The input of the core program will be written in this language, then the input must be checked to conform to the WTT language.
The language chosen is hard to write and read, therefore it is decided that a second program will be developed, that accepts a more user friendly language.

Checking that the input line is in accordance with the WTT language is hard, therefore an object structure is introduced to represent the input. Checking that this object structure is in accordance with the WTT language is easier.

A user interface is build to enable the user to enter an input and to view all information stored by the core program. This interface is described in the final section of this chapter.

2.1 Restricting the input of the program

The interesting part of a WTT book are the lines that are part of the book. The book is just a category in which multiple lines can be stored. Therefore our program is designed to store and show lines. Lines entered in our program can be stored in a file, this file represents the book. Only lines can be entered into our program. The lines entered in the program, together, form the book. Only allowing correct lines, conform the WTT language, to be entered into the program guarantees us that the book is correct conform the WTT language.
This restricts the language we use in our program. We do not have to represent a book, it is represented by the files the program loads and saves. The correctness check for the book is done by checking all lines, when a book is loaded.

The input for a program is limited by the symbols on the keyboard, not all symbols used in the previous chapter can be typed on a keyboard. There are ways to get more symbols by using a combination of keys, but a user would like to use the program by only using the regular keys. Therefore we need to use other symbols than the symbols used in CS-Report 02-05 [Ned 2002]. We examine all symbols that can be entered on a keyboard and put them in different groups. The elements of these groups then can be used to define the different categories of the language used by our program.

Group definitions
We divided the characters of the keyboard in four groups.

letters	:	'a' .. 'z', 'A' .. 'Z'
digits	:	'0' .. '9'
symbols	:	the symbols on the keyboard. (e.g. '!', '@', '#', '$', '%', '^', '&' etc..)
separators	:	<space> and <enter> (or end of line)

Variables and constants that are defined in the text can have all kinds of names, but we restrict these names to the following groups:

word[6] : letter (letter/digit/'_')*
a letter possibly followed by letters, digits or underscores

symbol : symbol symbol*
a symbol possibly followed by symbols

number : digit digit*/(digit* '.' digit*)
a digit possibly followed by digits or possibly followed by digits, the decimal point and some more digits.

A parser is built to recognize these different groups. If the parser recognizes a separator character, one element of a group is parsed successfully.
However, not all symbols can be used in a *symbol* name, because some *symbols* are used to represent language categories or are used to separate elements of a group to enhance the readability of the input. There are even two *words* that also cannot be used as a name for a variable or a constant. The *words* 'PROP' and 'SET' are already used to respectively represent the class of all statements and the class of all sets. The *symbols* and the two *words* that can not be used as names are put together in another group.

predefined : ':', ':=', ',', '.', '|>', '(', ')', '[', ']', 'PROP', 'SET'

These *predefined* symbols and names are recognized by the parser and are not possible as names for variables or constants. A *symbol* name cannot contain a *predefined* symbol, because then the parser would not be able to decide if a *predefined* symbol or a *symbol* name was parsed.

With these restrictions we can define the name group.

names : *words* except 'PROP' and 'SET'/all allowed *symbol* names/*numbers*.

Where the allowed *symbols* names are not *predefined* symbols or contain *predefined* symbols.

The parser gets an input from the program and will return the different groups parsed.
We define *identifier* as the group that is parsed and returned by the parser.

identifier : *names*/*predefined*/end of line

The parser returns some information to the program:
The *identifier* : in other words the concatenation of some keyboard characters that form an element from a group.
The type : is it a *name*, is it a *predefined* symbol or is the end of the line reached.

The parser will also rapport the begin and end positions, with relation to the entire input, of the characters parsed. Why this is needed is discussed in section 2.3. The code of the parser and the information object returned by the parser can be found in appendix A.

The order in which the identifiers are permitted to be parsed depends on the language definition. The language definition used on the computer is described in the next section.

[6] *word*, as defined here, is usually called an identifier, but the name *identifier* is introduced differently.

2.2 Language definition

The *identifiers* introduced in section 2.1 are now combined to form a language. The input of our program must be in accordance with this language. Two languages are introduced and one of these languages is chosen. After this choice the language will be defined by putting the different *identifiers* behind each other. The language definition is depicted in table 2.2a.

Sugared language
By sugared language we mean the language used in almost all mathematical texts.
The language in these texts is written in such a way that the user can more easily read the text, as well as write the text.
Examples of sugared language are : '3 + 5', 'a; b : PROP', '5 * 3 + 2'

Unsugared language
In the unsugared language we do not permit infix constants or multiple declaration. Only prefix constants are permitted and multiple declarations must be written as different declarations.
The same examples in unsugared language are: '+(3,5)', 'a : PROP, b : PROP', '+(*(5, 3), 2)'

Choosing the language
By looking at the examples we already see that the sugared language is, for a human, easier to understand and easier to write. For a program however, the unsugared language is easier to understand. For example the infix constants. For reading '5 * 3 + 2' correctly, a program needs additional information about the priorities and the associativity of the constants.
'+(*(5, 3), 2)' can be only read in one way.
The better understanding of the input, by the program, on itself is not enough reason to choose for the unsugared language, but the need for extra information to understand the infix constants and other constructs, possible in the sugared language, adds to the code of the program that needs to read the input.

Our program has to check if the input is in accordance with the WTT language and the "de Bruijn criterion" [Bar 2001] states, that a program that verifies correctness should be small; then this program can be checked by hand, giving the highest possible reliability of the verifying program. Absolute certainty about the correctness of that part of the core program that verifies the input cannot be reached, but by keeping the program small, correctness is easier to be checked.
This is the main reason that the first program, the core program, uses the unsugared language as input.

In a later stage a second program will be designed, the final program. The final program is adapted to accept some of the options that are part of the sugared language.
A layer will be put around the core program, this layer translates the sugared language to the unsugared language, then the core program will check this unsugared language.
The part of the core program that checks the input may not be changed, otherwise the "de Bruijn criterion" [Bar 2001] would not be obeyed and the choice for the unsugared language would not be necessary.

The unsugared language is defined to match the language definition discussed in section 1.2.
The unsugared language definition is depicted in table 2.2a.

Unsugared language definition
All categories in table 1.2a and the combination category, defined in section 1.3, are represented in the language definition. The definition of the unsugared language differs from the language definition in section 1.2, because of the language restrictions discussed in section 2.1.

```
variable        : name
constant        : name '(' parameterlist ')'
binder [7]      : name '[' declaration ']' '(' parameterlist ')'

parameterlist   : ∅                                  parameters        : argument
                : argument                                             : argument ',' parameters
                : argument ',' parameters

argument        : variable
                : constant
                : binder
                : combination

combination     : constant combinationlist           combinationlist   : constant
                                                                       : constant combinationlist
declaration     : variable ':' 'PROP'
                : variable ':' 'SET'
                : variable ':' argument

definition      : constant ':=' argument

statement       : variable
                : constant
                : binder

context         : ∅                                  contexts          : declaration
                : declaration                                          : statement
                : statement                                            : declaration ',' contexts
                : declaration ',' contexts                             : statement ',' contexts
                : statement ',' contexts

line            : context '|>' definition
                : context '|>' statement
```

table 2.2a

- The extra definitions of *parameterlist* and *argument* are introduced as abbreviation, for a group or structure of categories that occurs multiple times in the language definition.

- The extra definitions of *parameters* and *contexts* are needed to ensure that after the ',' identifier always an expression is written.

- The extra definition of *combinationlist* is needed to ensure that there are at least two constants in a combination category.

- Due to the design decision of section 2.1 we do not need a definition for the book category.

[7] by a slight abuse of notation we write '*binder*' instead of '*expression beginning with a binder*'

2.3 Checking the input

The input entered in our program must be checked to obey the unsugared language syntax, depicted in table 2.2a, and therefore to conform to the WTT syntax definitions. After this syntax check, the input must also be checked to conform to the derivation rules defined in section 1.4 and therefore satisfy the WTT linguistic type rules. It can be concluded that, if the input passes these two checks, the input is in accordance with the WTT rules.

The input is parsed and the *identifiers* are returned to the program. We now have two possibilities: checking the *identifiers* or representing the *identifiers* and checking this representation. In this section we discuss these possibilities and choose one of them.

Checking the input parsed
The input of our program is parsed from left to right. The first *identifier* parsed stands at the beginning of the input.
Checking the entire input begins with checking if the input is a correct *line*.
The *identifier* representing a line is somewhere in the middle of all *identifiers* parsed.
After checking that the input is a correct *line* ,the *context* of that *line* is checked to be correct.
The *context* can be found at the beginning of the input, which corresponds to the first *identifiers* parsed.

We could change the parsing, to more closely fit the checking of the input, by for example first search the '|>' identifier, which is needed to check if the input is a correct *line*.
The '|>' identifier does not occur in other *names*, so this would be possible. However, when checking the *context* we need to search for the ',' *identifier*. This *identifier* is also used in the *parameterlist*, hence the parser cannot recognize if the ',' *identifier* represents a *context* or a *parameterlist*. Therefore the parser cannot be completely changed to follow the order in which syntax checks and linguistic type checks are verified.

Furthermore, the language definition, as described in table 2.2a, will be changed in the final program to a more sugared language. This partly sugared language must be parsed and translated to the unsugared language, to obey the "de Bruijn criterion" [Bar 2001].
This unsugared language must be parsed by the core program to be checked on the WTT rules. The input would need to be parsed at least twice, even more because the parsing cannot be changed to fit the order in which the input must be checked.
Hence this is not an universal approach the check the input.

Translating the input to an object structure and checking the object structure
Another option is, parsing the input and in the meanwhile constructing an object structure that represents the input. Therefore, checking this object structure equals to checking the input. The object structure will be modeled in such a way that checking it is easier then checking the input.

When, in the final program, the partly sugared language is introduced, only the translation of the input has to be changed. Instead of parsing unsugared input to the object structure, we parse sugared input to the object structure.

The problem with this approach is that an object structure does not know which part of the input it represents. This information is needed to report, to the user, in which part of the input an error occurred.

Therefore the core program stores the original input and the objects in the object structure store the start and end positions of that part of the input that the object represents.
The start and end positions are received from the parser. The parser returns the start and end position together with the rest of the *identifier* information.
For better representation of the input, the start and end positions are expanded to include the spaces in front and after the *names*. By including the spaces, the object structure can represent the entire input more closely.

We choose to define an object structure and translate the input to that object structure.
The object structure then is checked on syntax correctness, conform to the language definition of table 2.2a, and correctness with respect to the derivation rules defined in section 1.4.
The next two sections introduce the object structure and a way to translate the input to an object structure. Then we return to checking this object structure.

2.4 Introducing the linguistic objects

The objects that together form the object structure are named linguistic objects and represent the unsugared language definition of section 2.2. This name is chosen, because the objects are a linguistic representation of the input.

The linguistic objects represent the different categories of the WTT language, which are also used in the unsugared language definition. There are also some *predefined* symbols that need to be represented by objects, so that all linguistic objects together can represent the entire unsugared language. This section defines all linguistic objects used by the program.

LINGUISTIC OBJECT
We start defining the linguistic object structure with the LINGUISTIC OBJECT itself. Every object in the linguistic object structure extends this object and adds its own storage fields to the storage fields already present in the LINGUISTIC OBJECT.

The LINGUISTIC OBJECT has two storage fields, the begin position and the end position of the part of the input that the LINGUISTIC OBJECT represents. These two fields are sufficient for generating error messages, but for different views, introduced in section 3.2.2, we also need the text represented by the objects. This text is stored in the represents field.

ATOMIC OBJECT
The *variable, constant* and *binder* definitions occur together, in two places, in the unsugared language definition viz. in the definition of *statement* and *argument*. We introduce the ATOMIC OBJECT to represent the three LINGUISTIC OBJECTS that represent the *variable, constant* and *binder* definitions.

All these tree LINGUISTIC OBJECTS are represented by a *name* identifier and all three LINGUISTIC OBJECTS extend the ATOMIC OBJECT. Therefore we add a name field to the ATOMIC OBJECT to store the *name* identifier. By adding the name field to the ATOMIC OBJECT, the three LINGUISTIC OBJECTS automatically can use the name field.

VARIABLE OBJECT
The VARIABLE OBJECT extends the ATOMIC OBJECT. The VARIABLE OBJECT represents a *variable,* which is represented by a *name*. This *name* is stored in the name field of the ATOMIC OBJECT.

CONSTANT OBJECT

The CONSTANT OBJECT extends the ATOMIC OBJECT and represents the *constant* definition. Apart from the *name*, stored in the name field of the ATOMIC OBJECT, the CONSTANT OBJECT has a *parameters list*. For time saving reasons the CONSTANT OBJECTS also store the number of parameters in an arity field. (see section 2.8, the essential linguistic information).

The CONSTANT OBJECT does not store the parentheses or commas depicted in the language definition. These symbols can be different for every language definition and are not important for checking the input.
For example : A correct CONSTANT OBJECT can still be constructed if we choose, in the language definition, to use '{', '}' instead of '(', ')' for enclosing a *parameter list.*

The program part that constructs the LINGUISTIC OBJECTS must check these *identifiers*, that depend on the language definition. An *identifier* that must be checked will be mentioned in the corresponding definition of its LINGUISTIC OBJECT.

BINDER OBJECT

The BINDER OBJECT extends the ATOMIC OBJECT and represents the *binder* definition. The BINDER OBJECT represents a *name*, stored in the name field of the ATOMIC OBJECT. Further the BINDER OBJECT represents the *declaration* and a *parameter list*. For time saving reasons the BINDER OBJECTS also stores the number of parameters in an arity field. (see section 2.8, the essential linguistic information)

COMBINATION OBJECT

A COMBINATION OBJECT represents the *combination* definition and therefore stores a list of CONSTANT OBJECTS. Whether every object in the list is a CONSTANT OBJECT will be verified by the syntax check. For defining the COMBINATION OBJECT we say that the COMBINATION OBJECT stores a LINGUISTIC OBJECTS list.

DECLARATION OBJECT

The DECLARATION OBJECT represents the *declaration* language definition. The *declaration* is defined as the declaration *identifier* (':') a left hand side and a right hand side. These are also the fields of the DECLARARTION OBJECT. The left hand side and right hand side of the DECLARATION OBJECT are LINGUISTIC OBJECTS.
'PROP' and 'SET' *identifiers* can appear on the right hand side of the *declaration*, hence the 'PROP' and 'SET' *identifiers* must be represented by LINGUISTIC OBJECTS.

SORTS OBJECT, PROP SORT OBJECT and SET SORT OBJECT

'PROP' and 'SET' are called sorts, the sort of all stats and the sort of all sets. We define two linguistic objects representing those two sorts, they both extend the SORTS OBJECT.
- The PROP SORT OBJECT representing the 'PROP' *identifier.*
- The SET SORT OBJECT representing the 'SET' *identifier.*

DEFINITION OBJECT

The DEFINITION OBJECT represents the *definition* language definition. The *definition* is defined as the definition *identifier* (':=') a left hand side and a right hand side. These are also the fields of the DEFINITION OBJECT. The left hand side and right hand side of the DEFINITION OBJECT are LINGUISTIC OBJECTS.

CONTEXT OBJECT

The CONTEXT OBJECT represents the *context* language definition. The *context* is defined as a list of *declarations* and *statements*, separated by the ',' identifier. Instead of defining the CONTEXT OBJECT as containing a list of LINGUISTIC OBJECTS we choose to model it in the same way as the DECLARATION OBJECT and the DEFINITION OBJECT. A CONTEXT OBJECT represents the (',') identifier and has a left hand side and a right hand side, both LINGUISTIC OBJECTS. The left hand side of the CONTEXT OBJECT can be another CONTEXT OBJECT, in this way a list of *declarations* and *statements* can be build.
By modeling the *context* in this way, it is not possible to have an empty context or a context of only one declaration or statement. This is solved by allowing that the left hand side of a *line* can be empty, one *statement*, one *declaration* or a *context*.
To represent the empty context we need a LINGUISTIC OBJECT.

EMPTY CONTEXT OBJECT

This linguistic object represents the empty context.

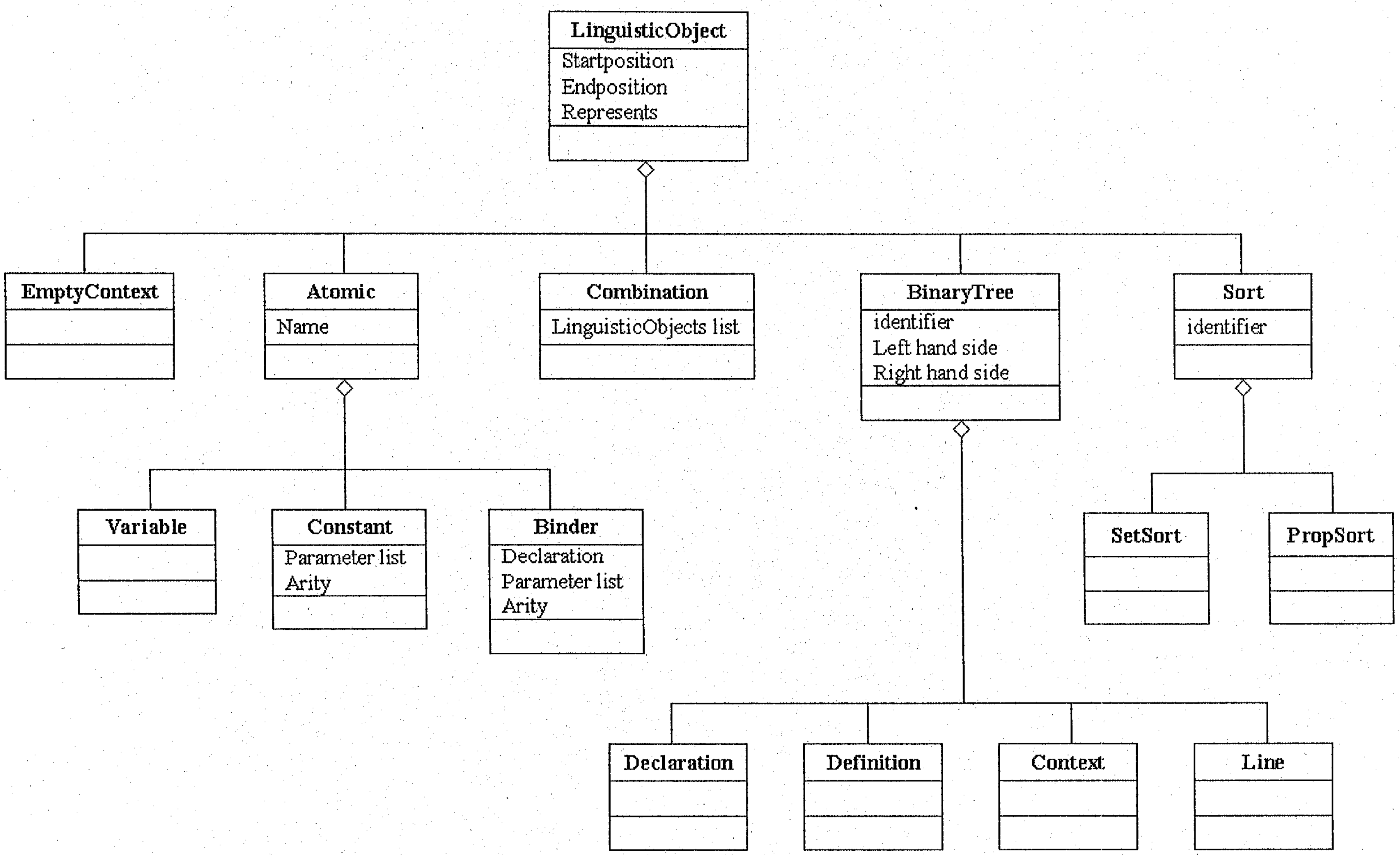
LINE OBJECT

The LINE OBJECT represents the *line* language definition. The *line* is defined as the line identifier ('|>') a left hand side and a right hand side. These are also the fields of the LINE OBJECT. The left hand side and right hand side of the LINE OBJECT are LINGUISTIC OBJECTS.

BINARY TREE OBJECT

The DECLARATION OBJECT, DEFINITION OBJECT, CONTEXT OBJECT and LINE OBJECT have the same three fields. The identifier representing the different objects, a LINGUISTIC OBJECT as left hand side and a LINGUISTIC OBJECT as right hand side.
Therefore we introduce the BINARY TREE OBJECT. The BINARY TREE OBJECT represents the DECLARATION OBJECT, DEFINITION OBJECT, CONTEXT OBJECT and LINE OBJECT. It contains an identifier field, a left hand side field and a right hand side field. The DECLARATION OBJECT, DEFINITION OBJECT, CONTEXT OBJECT and LINE OBJECT extend the binary tree object.

The LINGUISTIC OBJECTS are depicted in class diagram 2.4a



class diagram 2.4a

2.5 Constructing a linguistic object tree

The next steps, as discussed in section 2.3, are translating the input to an object structure and then checking that object structure.
We start this section with defining a six step approach, which will be used to translate the input to the linguistic object structure. Using this six step approach would mean that the input must be parsed six times. To avoid this we also introduce a parallel approach that builds the same linguistic object structure in one step. This parallel approach is used by our program and some help functions will be defined to construct the linguistic object structure in the parallel approach. The same help functions can be found in the code of this part of the program that is included in this thesis as appendix B.

After the definition of an approach, an example will visualize how that approach builds an object structure. In this example only the fields of the BINARY TREE OBJECTS are shown.
The fields of the other objects would only obscure the visualization. The structure that is found is a binary tree, hence the name BINARY TREE OBJECT. Therefore we call the structure constructed of the LINGUISTIC OBJECTS the linguistic object tree.

2.5.1 A six step approach

Constructing the linguistic object tree

1) All *name* identifiers are translated into ATOMIC OBJECTS. The ATOMIC OBJECT that has to be constructed depends on the *identifier* parsed after the *name* identifier. If that *identifier* is the '[' *identifier*, the program will construct a BINDER OBJECT. Parsing the '(' *identifier* corresponds with constructing a CONSTANT OBJECT. Any other *identifier* leads to the construction of a VARIABLE OBJECT.

   - Constructing a BINDER OBJECT includes, besides parsing the *name*, parsing the '[' *identifier*, the *declaration*, the ']' *identifier*, the'(' *identifier*, the *parameters* separated by the ',' *identifier* and the ')' *identifier*. In our example all these *identifiers* representing a *binder* are replaced by a BINDER OBJECT.
   - Constructing a CONSTANT OBJECT includes, besides parsing the *name*, parsing the '(' *identifier*, the *parameters* separated by the ',' *identifier* and the ')' *identifier*. In our example all these *identifiers* representing a *constant* are replaced by a CONSTANT OBJECT.
   - Constructing a VARIABLE OBJECT means parsing the *name* identifier. In our example the *name* identifier representing a *variable* is replaced by a VARIABLE OBJECT.

   These constructions of the ATOMIC OBJECTS correspond to the language definitions in table 2.2a. If the *identifiers* parsed do not match the required *identifiers* an error message will be generated.

2) The program constructs the PROP SORT OBJECT if a 'PROP' *identifier* is parsed and a SET SORT OBJECT if a 'SET' identifier is parsed.
   In our example these *identifiers* are replaced by their LINGUISTIC OBJECTS.

3) The ':', ':=', ',', '|>' *identifiers* that are parsed lead to the construction of BINARY TREE OBJECTS. The BINARY TREE OBJECTS constructed are respectively DECLARATION OBJECT, DEFINITION OBJECT, CONTEXT OBJECT and LINE OBJECT.
   In our example these *identifiers* are replaced by their LINGUISTIC OBJECTS.

After these three steps the entire input has been parsed and all *identifiers* have been translated to their LINGUISTIC OBJECTS. The last three steps connect the constructed LINGUISTIC OBJECTS, to build the linguistic object tree.

Examining the language definition of table 2.2a and the LINGUISTIC OBJECTS definition of section 2.4, shows that every BINARY TREE OBJECT is separated from another BINARY TREE OBJECT by only one LINGUISTIC OBJECT (not a BINARY TREE OBJECT). The only possibility is that more than one CONSTANT OBJECT is placed between two BINARY TREE OBJECTS, but in this case a COMBINATION OBJECT must be constructed to represent those CONSTANT OBJECTS.

4) If more linguistic objects occur between two BINARY TREE OBJECTS, a COMBINATION OBJECT is constructed and all LINGUISTIC OBJECTS between the two BINARY TREE OBJECTS are added as elements of the linguistic object list field of the COMBINATION OBJECT.

5) Of the BINARY TREE OBJECTS, first the DECLARATION OBJECT and the DEFINITION OBJECT are constructed. Both DECLARATION OBJECT and DEFINITION OBJECT will take the LINGUISTIC OBJECT on their left and the LINGUISTIC OBJECT on their right and add them to their respectively left hand side field and right hand side field. If this leads to a conflict or, if there is no left or right hand side, an error must be generated.

6) In the last step the CONTEXT OBJECT and the LINE OBJECT are constructed. Just like the DECLARATION OBJECT and the DEFINITION OBJECT, these objects will take the LINGUISTIC OBJECT on their left and the LINGUISTIC OBJECT on their right and add them to their respectively left hand side field and right hand side field. This step starts at the leftmost CONTEXT OBJECT or LINE OBJECT. The second CONTEXT OBJECT or LINE OBJECT takes the first CONTEXT OBJECT or LINE OBJECT as its left LINGUISTIC OBJECT. If there is no left or right hand side, an error will be generated, except for the LINE OBJECT, which may have no LINGUISTIC OBJECT to its left. In that case an EMPTY CONTEXT OBJECT is constructed as the left hand side of the LINE OBJECT. Because a correct input only has one LINE OBJECT, this LINE OBJECT will be the first LINGUISTIC OBJECT found. Only if this is the case an EMPTY CONTEXT OBJECT is constructed.

Not every input entered in our program can be translated into a linguistic object tree. However we try to give as less as possible errors while constructing the linguistic object tree. In this way the errors will be generated by the syntax and linguistic type checks that need to verify the linguistic object tree.
These errors are more precise and can be traced to the WTT rules.
Therefore for example, we will not check if only CONSTANT OBJECTS are added in the list of a COMBINATION OBJECT in step 4. The syntax check will catch and report this error.

Example:
We visualize these six steps in an example[8]:

A : SET, Forall [a : A] (Exists [b : A] (=(b,/(1(), a)))), a : nonzero() Down(A) I>
invers_times(A, a) := Iota [b : A] (=(b,/(1(), a)))

=> (Step 1: all name *identifiers* are replaced by their corresponding ATOMIC OBJECTS, including declarations and/or parameters.)

VARIABLE : SET, BINDER, VARIABLE : CONSTANT CONSTANT I> CONSTANT := BINDER

=> (Step 2: the 'PROP' and 'SET' *identifiers* are replaced and the corresponding SORT OBJECTS are constructed.)

VARIABLE : SETSORT , BINDER, VARIABLE : CONSTANT CONSTANT I> CONSTANT := BINDER

=> (Step 3: the *identifiers* representing BINARY TREE OBJECTS are parsed and the BINARY TREE OBJECTS constructed.)

VARIABLE DECLARATION SETSORT CONTEXT BINDER CONTEXT
VARIABLE DECLARATION CONSTANT CONSTANT LINE CONSTANT DEFINITION BINDER

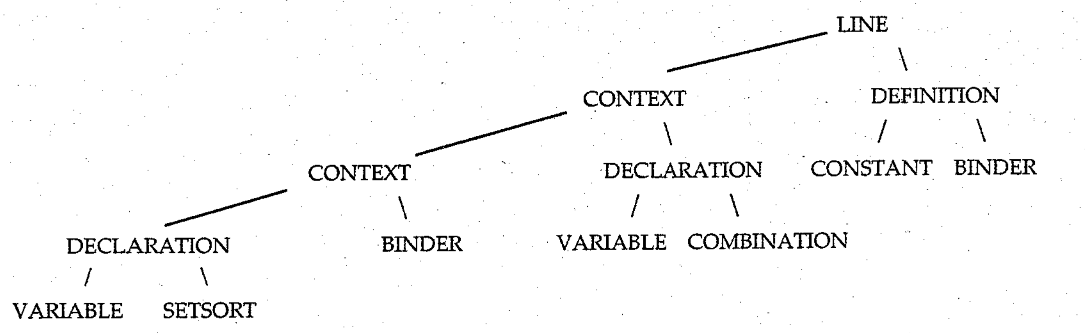
=> (Step 4: COMBINATION OBJECTS are constructed to ensure that only one LINGUISTIC OBJECT stands between two BINARY TREE OBJECTS.)

VARIABLE DECLARATION SETSORT CONTEXT BINDER CONTEXT
VARIABLE DECLARATION COMBINATION LINE CONSTANT DEFINITION BINDER

=> (Step 5: LINGUISTIC OBJECTS are added as left hand side field and right hand side field of the DECLARATION OBJECT and the DEFINITION OBJECT)

```
      DECLARATION  CONTEXT  BINDER  CONTEXT  DECLARATION       LINE      DEFINITION
        /      \                               /       \                  /      \
   VARIABLE   SETSORT                     VARIABLE  COMBINATION    CONSTANT   BINDER
```

=> (Step 6: LINGUISTIC OBJECTS are added as left hand side field and right hand side field of the CONTEXT OBJECT and the LINE OBJECT, beginning at the left side of the LINGUISTIC OBJECTS)



[8] In this example we assume that 'nonzero', '=' and '/' are defined in the preface. '1' is a number and standard defined as a constant. 'Down' represents the '↓' constant, as defined in section 1.3 (the Ltypes adjective and noun), 'Forall', 'Exists' and 'Iota' represent respectively the '∀',' ∃' and 'ι' binder.

2.5.2 A parallel approach

Constructing the linguistic object tree

Our program builds the same linguistic tree as built with the six step approach, but instead of using six steps separately, all steps are implemented in parallel. In this way the input is parsed once and then the linguistic object tree is constructed.

The construction of the linguistic object tree is done by using some help functions. The functions are constructed in such a way that always one linguistic object tree is constructed to represent the input part that already has been parsed. Subsequently if the entire input is parsed a linguistic object tree is constructed that represents the entire input.

We introduce the help functions that are used to built the linguistic object tree.

Function (1) : Function (1) is responsible for implementing the steps 1,2 and 4 of the six step approach. Function (1) is called if a *name* identifier is received from the parser. The *identifier* after the *name* identifier is examined by function (1). Depending on this *identifier* one of three other help function is called.

- If that *identifier* is a '[' function (a) is called. Function (a) constructs a BINDER OBJECT, by parsing the *identifiers* described section 2.5.1. (step 1)
- If that *identifier* is a '(' function (b) is called. Function (b) constructs a CONSTANT OBJECT, by parsing the *identifiers* described section 2.5.1. (step 1)
- Otherwise function (c) is called. Function (c) constructs a VARIABLE OBJECT, by parsing the *name* identifier.

Function (1) is also called if a 'PROP' *identifier* or a 'SET' *identifier* is received from the parser. Then the corresponding PROP SORT OBJECT or SET SORT OBJECT is constructed.

If after the construction of these LINGUISTIC OBJECTS again a *name* identifier, 'PROP' *identifier* or 'SET' *identifier* is received from the parser, a COMBINATION OBJECT is constructed. The first LINGUISTIC OBJECT constructed is added to the linguistic object list field of the COMBINATION OBJECT and a second LINGUISTIC OBJECT is constructed. This LINGUISTIC OBJECT is also added to the linguistic object list field. Adding LINGUISTIC OBJECTS to the COMBINATION OBJECT Continues until no *name* identifier, 'PROP' *identifier* or 'SET' *identifier* is received from the parser.

Function (2) : Function (2) is responsible for implementing steps 3, 5 and 6 of the six step approach. A part of this task is done by calling another help function (3), which will be discussed afterwards. Function (2) is called if a ':' *identifier*, a ':=' *identifier*, a ',' *identifier* or a '|>' *identifier* is received from the parser.
Function (2) constructs the BINARY TREE OBJECT that corresponds with the *identifier* that has been received from the parser. The LINGUISTIC OBJECT constructed, before this *identifier* was received, is added as the left hand side of the constructed BINARY TREE OBJECT. There are now two possibilities:

- When the BINARY TREE OBJECT constructed is a DECLARATION OBJECT or a DEFINITION OBJECT, function (1) is called. After a ':' *identifier* or a ':=' *identifier* an *argument*, 'PROP' *identifier* or 'SET' *identifier* is expected. (see table 2.2a) These *identifiers* are read and translated to LINGUISTIC OBJECTS by function (1). Then the LINGUISTIC OBJECT, representing the already parsed line, is added as left hand side of the constructed DECLARATION OBJECT or DEFINITION OBJECT and the LINGUISTIC OBJECT constructed after the ':' *identifier* or ':=' *identifier* is added as right hand side of the constructed BINARY TREE OBJECT.

- When the BINARY TREE OBJECT constructed is a CONTEXT OBJECT or a LINE OBJECT, function (1) and function (3) are called. After a ',' *identifier* or a '|>' *identifier* not only *statements* can be found, but also *declarations* or *definitions*. *Declarations* and *definitions* however, begin with *name* identifiers (see table 2.2a). Therefore function (1) is called to construct a LINGUISTIC OBJECT. This LINGUISTIC OBJECT is passed to function (3) as an argument. Function (3) returns the right hand side for the CONTEXT OBJECT or LINE OBJECT.

Function (3) Function (3) is called with a LINGUISTIC OBJECT as an argument.
Function (3) tries to construct a DECLARATION OBJECT or a DEFINITION OBJECT if the parser returns a ':' or ':=' *identifier*. This is done in the same way that function (2) constructs a DECLARATION OBJECT or a DEFINITION OBJECT. The only difference is that instead of using the LINGUISTIC OBJECT representing the already parsed line, the LINGUISTIC OBJECT that was given as an argument is used as the left hand side of the DECLARATION OBJECT or the DEFINITION OBJECT. If the parser returns a ',' or '|>' *identifier* function (3) will return the LINGUISTIC OBJECT that was given as an argument, because then a *statement* was parsed. Otherwise function (3) will generate an error message.

For parsing the entire input the help functions are used in the following order.
The first *identifier* is parsed. If this is a '|>' *identifier* an EMPTY CONTEXT OBJECT is created and function (2) is called to construct the LINE OBJECT.
Otherwise function (1) is called. Function (1) calls itself till no *name* identifier, 'PROP' *identifier* or 'SET' *identifier* is received from the parser. Then function (2) is called, because an *identifier* that represents a BINARY TREE OBJECT is expected. If a BINARY TREE OBJECT is constructed function (2) is called again, because the BINARY TREE OBJECT constructed is the left hand side of the next BINARY TREE OBJECT. This continues till the end of line *identifier* is parsed.

As was the case with the six step approach, a lot of input entered into the program can be parsed, while this input does not correspond with the language definition of table 2.2a. However the point is that all input corresponding to this table can be parsed. All wrong input will be filtered out by the syntax checks. Thereafter the input will be checked to correspond to the derivation rules of section 1.4.

The example
A : SET, Forall [a : A] (Exists [b : A] (=(b,/(1(), a)))), a : nonzero() Down(A) |>
invers_times(A, a) := Iota [b : A] (=(b,/(1(), a)))

=> (first identifier is parsed. A name identifier results in a call to function (1))

VARIABLE : SET, Forall [a:A] (Exists [b : A] (=(b,/(1(), a)))), a : nonzero() Down(A) |> ......

=> (function (2) is called, a ':' *identifier'* is parsed and a DECLARATION OBJECT is constructed.)

```
    DECLARATION , Forall [a:A] (Exists [b : A] ( =(b,/(1(), a)) )), a : nonzero() Down(A) |> ......
      /         \
VARIABLE     SETSORT
```

=> (function (2) is called and a ',' *identifier* is parsed. Function (1) is called and the LINGUISTIC OBJECT constructed is passed to function (3). Function (3) receives another ',' *identifier* from the parser and returns the given LINGUISTIC OBJECT to function (2). function (2) adds that LINGUISTIC OBJECT as the right hand side of the constructed CONTEXT OBJECT.)

```
                             CONTEXT, a : nonzero() Down(A) |> invers_times(A, a) := ......
                   _______/        \
        DECLARATION                BINDER
          /       \
    VARIABLE    SETSORT
```

=> (function (2) is called and the ',' *identifier* is received. Function (1) is called and the LINGUISTIC OBJECT constructed is passed to function (3). This time function (3) receives a ':' identifier, a DECLARATION OBJECT is constructed and returned to function (2).)

```
                                           CONTEXT |> invers_times(A, a) := ......
                                 _______/       \
                         CONTEXT              DECLARATION
                _______/       \               /        \
      DECLARATION              BINDER     VARIABLE  COMBINATION
        /       \
  VARIABLE    SETSORT
```

=> (function (2) is called and the '|>' *identifier* is received. Function (1) and function (3) construct the DEFINITION OBJECT and the LINE OBJECT is constructed.)

```
                                                          LINE
                                               _______/      \
                                           CONTEXT           DEFINITION
                                 _______/       \            /        \
                         CONTEXT              DECLARATION  CONSTANT   BINDER
                _______/       \               /        \
      DECLARATION              BINDER     VARIABLE  COMBINATION
        /       \
  VARIABLE    SETSORT
```

=> (The end of line *identifier* is parsed and the linguistic object tree is constructed.)

2.6 Syntax checking : syntax rules

After parsing the input and constructing the linguistic object we return to checking that the tree is in accordance with the language definition of table 2.2a. In this section we will translate the language definition of table 2.2a in rules for the different LINGUISTIC OBJECTS. The syntax rules will be numbered and referred to in the next section. In that section it is described how the linguistic object tree needs to be checked to be in accordance with the syntax rules.

variable, constant, binder
variables, constants and *binders* are represented by *name* identifiers. The correctness of the *name* is checked in a later stage, see section 2.8 Ltype rules for the linguistic object tree.
The parser cannot construct the corresponding LINGUISTIC OBJECTS if the syntax is not in accordance with the language definition of table 2.2a.
We only have to check the form of the *parameterlist* of the *constant* (1); and the form of the *declaration* (2) and the form of the *parameterlist* (3) of the *binder.*

parameterlist
The syntax of the *parameterlist* is also checked in the parser. A *parameterlist* consists of *arguments* separated by the ',' symbol. Only the form of the *arguments* have to be checked.

argument
An *argument* can have the form of a *variable* (4a), *constant* (4b), *binder* (4c) or *combination* (4d).

combination
A *combination* consists of a list with, at least two (5), *constants* (6).

declaration
A *declaration* is represented by the ':' identifier (7), has a *variable* as its left hand side (8) and 'PROP' (9a), 'SET' (9b) or *argument* (9c) as its right hand side.

definition
A *definition* is represented by the ':=' identifier (10), has a *constant* as its left hand side (11) and an *argument* (12) as its right hand side.

statement
A *statement* must have the form of a *variable* (13a), *constant* (13b) or *binder* (13c).

context
A *context* can be empty (14a), a *declaration* (14b), a *statement* (14c) or a list of *declarations* and *statements* (14d). The ',' identifier is used to separate the *declarations* and *statements* in a context list (15).

line
A *line* is represented by the '|>' identifier (16), a *line* must have a *context* (17) as its left hand side. The right hand side of a *line* can be a *definition* (18a) or a *statement* (18b). Only Lines can be entered in our program (19).

Which concludes the syntax rules deduced from the language definition of table 2.2a.

2.7 Syntax checking : linguistic object tree

The language definitions of section 2.2 are represented by LINGUISTIC OBJECTS in section 2.4. These LINGUISTIC OBJECTS have now to be checked on the syntax rules defined in section 2.6.

Syntax checking the linguistic objects
We describe, in the same order used in section 2.4, how each LINGISTIC OBJECT must be checked and which syntax rules are satisfied by those checks.

LINGUISTIC OBJECT
The LINGUISTIC OBJECT has no syntax rules. The LINGUISTIC OBJECT is extended by the other objects. The syntax rules of these objects must be checked.

ATOMIC OBJECT
The ATOMIC OBJECT represents the *variable, constant* and *binder* language definitions. These language definitions are used in *arguments* and *statements.*
For *arguments* the ATOMIC OBJECT must check that the VARIABLE OBJECT, CONSTANT OBJECT or BINDER OBJECT is correct (4a, 4b, 4c)[9]
For *statements* the ATOMIC OBJECT must check that the VARIABLE OBJECT, CONSTANT OBJECT or BINDER OBJECT is correct (13a, 13b, 13c)

VARIABLE OBJECT
The name of a VARIABLE OBJECT must be correct, which is checked by the Ltype rules discussed in section 2.8.

CONSTANT OBJECT
A CONSTANT OBJECT must have a correct name, which is checked by the Ltype rules discussed in section 2.8. Only LINGUISTIC OBJECTS can be added as parameters of a CONSTANT OBJECT, it must be checked that these LINGUISTIC OBJECTS have a correct form (3). Correct parameters have the form of an *argument*, therefore the LINGUISTIC OBJECTS that represent the parameters must be ATOMIC OBJECTS (4a, 4b, 4c) or COMBINATION OBJECTS (4d).

BINDER OBJECT
A BINDER OBJECT must have a correct name, which is checked by the Ltype rules discussed in section 2.8. A BINDER OBJECT has one LINGUISTIC OBJECT that must represent the *declaration* (2), this corresponds to a DECLARATION OBJECT, and some LINGUISTIC OBJECTS that represent the parameters (3). Correct parameters have the form of an *argument*, therefore the LINGUISTIC OBJECTS that represent the parameters must be ATOMIC OBJECTS (4a, 4b, 4c) or COMBINATION OBJECTS (4d).

COMBINATION OBJECT
A COMBINATION OBJECT consists of a list of LINGUISTIC OBJECTS that represent the *constants* of the COMBINATION OBJECT. The length of the list must be at least two (5) and the LINGUISTIC OBJECTS must be CONSTANT OBJECTS (6).

DECLARATION OBJECT
The declaration identifier must be the ',' symbol (7). The left hand side of the DECLARATION OBJECT must be a VARIABLE OBJECT (8), while the right hand side can be a PROPSORT OBJECT (9a), a SETSORT OBJECT (9b) or a LINGUISTIC OBJECT that represents an *argument* (9c). An *argument* is represented by an ATOMIC OBJECT (4a, 4b, 4c) or a COMBINATION OBJECT (4d).

[9] These numbers refer to the syntax rules of section 2.6

SORTS OBJECT
The SORTS OBJECT has no syntax rules, but the PROPSORT OBJECT and the SETSORT OBJECT that extend the SORTS OBJECT must be syntax correct.

PROPSORT OBJECT
The PROPSORT OBJECT must represent the 'PROP' identifier, to validate (9a).

SETSORT OBJECT
The SETSORT OBJECT must represent the 'SET' identifier, to validate (9b).

DEFINITION OBJECT
The declaration identifier must be the ',' symbol (10). The left hand side of the DEFINITION OBJECT must be a CONSTANT OBJECT (11), while the right hand side must be a LINGUISTIC OBJECT that represents an *argument* (12). An *argument* is represented by an ATOMIC OBJECT (4a, 4b, 4c) or a COMBINATION OBJECT (4d).

CONTEXT OBJECT
As was already discussed in section 2.4, when the CONTEXT OBJECT was introduced, the CONTEXT OBJECT is modeled in another way. Therefore the syntax rules for the *context* are partly checked by the LINE OBJECT. The identifier of the CONTEXRT OBJECT is the ',' symbol (15). The CONTEXT OBJECT represents a list of DECLARATION OBJECTS and LINGUISTIC OBJECTS that represent a *statement* (14d). The LINGUISTIC OBJECT that represents a *statement* is the ATOMIC OBJECT (13a, 13b, 13c).
A list of DECLARATION OBJECTS and ATOMIC OBJECTS is constructed by allowing that the left hand side of a CONTEXT OBJECT can be another CONTEXT OBJECT. The left hand side of a CONTEXT OBJECT can also be a DECLARATION OBJECT or an ATOMIC OBJECT if the first LINGUISTIC OBJECT in the context is reached. The right hand side of the CONTEXT OBJECT can only be a DECLARATION OBJECT or an ATOMIC OBJECT.

EMPTY CONTEXT OBJECT
An EMPTY CONTEXT OBJECT may only occur on the left hand side of the LINE OBJECT (14a). This is also the only place where it is checked. In any other place an error will be generated, because the wrong LINGUISTIC OBJECT is found.

LINE OBJECT
The line identifier must be the '|>' symbol (16), because of the changes in the CONTEXT OBJECT the left hand side of the LINE OBJECT can be an EMPTY CONTEXT OBJECT (14a), a DECLARATION OBJECT (14b), a LINGUISTIC OBJECT representing a *statement* (14c) or a CONTEXT OBJECT, which represents a list of *declarations* and *statements* (14d). The LINGUISTIC OBJECT that represent a *statement* is the ATOMIC OBJECT (13a, 13b, 13c). These four LINGUISTIC OBJECTS (EMPTY CONTEXT OBJECT, DECLARATION OBJECT, ATOMIC OBJECT, CONTEXT OBJECT) represent the *context* (17).
The right hand side of the LINE OBJECT can be a DEFINITION OBJECT (18a) or a LINGUISTIC OBJECT representing a *statement* (18b), which is the ATOMIC OBJECT (13a, 13b, 13c).

BINARY TREE OBJECT
The BINARY TREE OBJECT has no syntax rules. It is extended by the DECLARATION OBJECT, DEFINITION OBJECT, CONTEXT OBJECT and LINE OBJECT.

Which only leaves syntax rule (19). The is not a check on a LINGUISTIC OBJECT, but it is a check on the linguistic object tree.

Syntax checking the linguistic object tree
The only syntax rule to be checked, concerning the linguistic object tree, is that the root of the linguistic object tree is a LINE OBJECT (19).

We discussed how all LINGUISTIC OBJECTS must be checked on syntax correctness, but these objects are part of the linguistic object tree. We want to check the linguistic object tree from left to right, which is the way the input was read. For example: a syntax error in the context must, logically, be found before a syntax error in a definition is found.
Therefore first the left hand side of a BINARY TREE OBJECT is checked and if the left hand side is syntactically correct, the right hand side of the BINARY TREE OBJECT is checked to be syntactically correct.

Note
These are the strict demands on the syntax of the LINGUISTIC OBJECTS.
Our program is built to assist the user. Therefore the check is somewhat weakened so that it also allows COMBINATION OBJECTS at the right hand side of the line. In this way a user can input LINGUISTIC OBJECTS that are not a *statement* as right hand side of the LINE OBJECT.
The program will, in the case that the right hand side of the LINE OBJECT is a *statement* or a *definition*, ask the user if the *line* must be added to the book. In every other case the program will give no error message, indicating that the given line is syntax correct, but the *line* can then not be added to the book.

The goal of this weakening of the syntax rules is that the given line can be checked to be correct in accordance with the derivation rules of section 1.4. In this way the Ltype of the LINGUISTIC OBJECT at the right hand side of the LINE OBJECT can be calculated and returned to the user.
The program than can be used to calculate the Ltype of a given input.

The program checking the syntax rules can be found in appendix C.
The next section will describe how the linguistic object tree is checked to be in accordance with the derivation rules of section 1.4.

2.8 Linguistic type checking : linguistic object tree

After checking that the linguistic object is in accordance with the language definition of table 2.2a, the linguistic object tree must be checked to be in accordance with the derivation rules of section 1.4. For these checks we need some information about the Ltypes of the constants and binders defined in the preface and some information about the variables declared in the preface.
Very important is the order in which the checks are performed because, as we already discussed in section 2.7, syntax checking the linguistic object tree. The *context* must be correct, before a *definition* can be checked. Further we will define a semi-formal, computer program based language used to describe all checks on the LINGUISTIC OBJECTS.

The Essential Ltype Information
The ext-cons derivation rule and the bind derivation rule use the in and out function of a name to get, respectively, the Ltypes of the parameters and the Ltype of the constant or binder defined in the preface. This information must be stored in our program and must be available for checking the LINGUISTIC OBJECTS.
Except from this information, we also store the Ltypes of the variables declared in the context, this is needed to simplify the var derivation rule. Instead of searching for the declaration of the variable we search the stored information and use this to verify the Ltype of the variable. Similarly, this information is also stored for constants defined in the book.
Saving the information of a constant defined in the book in the same way as the information is stored in the preface, removes the use of the int-cons derivation rule.

We define an object that stores the Essential Ltype Information (ELI OBJECT) for the variables declared in the context, the constants defined in the book and, for the constants and books defined in the preface.

The ELI OBJECT must store:

- name	: the name of the variable, constant or binder the ELI OBJECT represents.
- category	: the category of the name (`variable`, `constant` or `binder`).
- in[]	: a list of Ltypes of the parameters. A variable has no parameters.
- out	: the Ltype of the variable, constant or binder the ELI OBJECT represents.

Checking that all parameters have the correct Ltype costs time, especially if it is clear that there are more or less parameters given than required. Instead of checking that the given parameters are correct, the program can generate an error in advance if the parameters do not match. Retrieving the list of parameters from an ELI OBJECT, just to get the length of the list is inefficient, therefore an ELI OBJECT also stores the number of parameters.

- arity	: the number of parameters of the constant or binder the ELI OBJECT represents. (variables do not have parameters, hence they do no use the arity field.)

For this reason we also added an arity field to the CONSTANT OBJECT and to the BINDER OBJECT.

We can now define the preface to be a list of ELI OBJECTS, just as the variables declared in the context and the constants defined in the book are stored in a list of ELI OBJECTS.
The preface is even split in two lists of ELI OBJECTS, namely:
the static preface, that contains the binder and constant definitions, needed in every book and the dynamic preface, that contains the constant definitions, different for each book.

We introduce the following abbreviations.

Static preface	= SPRE		
Dynamic preface	= DPRE		
Preface	= SPRE $\cup$ DPRE	= PRE	(Binders and Constants in the preface)
Essential-Book	= EB		(Constants defined in the book)
Essential-Line	= ELine		(Variables declared in a line)
Essential-Global	= PRE $\cup$ EB	= EG	(All defined Binders and Constants in a book)
Essential-Local	= ELine	= EL	(All declared variables in a line)

We will give an explanation how the checks are performed on the linguistic objects, the checks are linked to the derivation rules of section 1.4.

Correctness by construction
As discussed in section 2.1, our program only accept correct lines, therefore the book is correct by construction. This requires that changes to the lines, already added to the book, may not invalidate the book.

We also want to construct the linguistic type checks in such a way that the context of a line is correct by construction. Therefore we check the linguistic object tree from left to right as discussed in section 2.7 syntax checking the linguistic object tree.
First the left most declaration or statement in the context is checked then the next declaration or statement is checked and so forth. If an error occurs in the context the Ltype checking will be terminated and no LINGUISTIC OBJECT will be checked without the certainty that all LINGUISTIC OBJECTS that are in front of it are checked on the forehand. For this reason first the left hand side of a BINARY TREE OBJECT is checked before the right hand side of the BINARY TREE OBJECT.

By this construction we can check every LINGUISTIC OBJECT in the linguistic object tree, without first checking the context and book under which that LINGUISTIC OBJECT is used. This however means that for every BINARY TREE OBJECT first the left hand side must be checked, before the right hand side is checked. The rules defined for the LINGUISTIC OBJECTS must therefore be read from top to bottom.

To keep the line correct by construction, all declarations must add a local variable to the EL, when the declaration is found correct conform to the derivation rules of section 1.4. To keep the book correct by construction, all definitions must add a global constant to the EG, when the definition is found correct conform to the derivation rules. The declarations in an expression beginning with a binder also add a variable to the EL, but this variable is removed from the EL after the entire binder is checked to conform to the derivation rules.

Introduction orthography
To depict the Ltype rules on the LINGUISTIC OBJECTS we introduce a semi-program language for different operations used in the rules and for depicting the order in which the checks of the rules must be verified. We start with a description of this semi-program language and comment on its usage.

<u>General</u>
- In the Ltype rules we will check the LINGUISTIC OBJECTS and in some places use information from an ELI OBJECT. These objects have fields as defined in section 2.4 and in this section when the Essential Ltype Information was introduced. To talk about the fields of an object we will name the object and, separated by a '.', name the field we want to use. For example: if e is an ELI OBJECT then e.name represents the name of that ELI OBJECT. The name of the CONSTANT OBJECT at the left hand side of DEFINITION OBJECT d can be represented by d.left.name.

- For the Ltype rules a similar structure as the derivation rules will be used. Here a dotted line separates the premisses from the conclusion.

- Some checks need to be performed on all items of a list. In accordance with the notation in the derivation rules, the (i = 1, . . . , n) notation is used for repeating a check for all elements in a list. This repetition can also be used to repeat another operation or multiple checks.

- The same abbreviation method used in the derivation rules of section 1.4 is used. This abbreviation method is explained in the var derivation rule of section 1.4. The abbreviation method is denoted by the '/' symbol and combines different rules about different Ltypes into one rule.

Symbols

- comparing a field with a value or a class is denoted by the '=' symbol.

- The '∈' symbol is introduced to search for an item in one of the ELI lists. The item, if found can be used in the rest of the Ltype rule, if the item is not found the Ltype check will be aborted. The '∉' symbol is introduced to check that an item does not occur in the ELI list.

- an ELI list is increased by an ELI OBJECT by adding the ELI OBJECT to the list with the use of the '∪' operator. For decreasing an ELI list the '\' operator is used.

Operations

- case: In some Ltype rules more than one LINGUISTIC OBJECT can occur. To separate between the different possible LINGUISTIC OBJECTS we introduce a case operation.
  The different cases of the case operation end by jumping to the end of the case operation, depicted by end case. The parameter of the case operation decides which case is evaluated. This decision is made by comparing a field with a certain value or a certain class.

- class: comparing if a field is from a certain class can only be done by getting the class of that field. For obtaining the class of a field we use the class operation.

- Ltypecheck: In the Ltype rules references to other Ltype rules are made. We therefore introduce the Ltypecheck operation that checks if a LINGUISTIC OBJECT has a certain Ltype. Therefore the Ltypecheck operation has two parameters, the LINGUISTIC OBJECT and its required Ltype. The Ltype used are the basic and extra Ltypes defined in section 1.3. Checking that a LINGUISTIC OBJECT has a certain Ltype is done by a Ltype rule, therefore the conclusion of an Ltype rule is represented by the Ltypecheck operation.

- check: For verifying if a field has a certain value, the check operation is defined. The difference with the '=' symbol is that if a check fails an error must be generated and the Ltype rule must be aborted.

- cast: Because the ATOMIC OBJECTS can be a VARIABLE OBJECT, a CONSTANT OBJECT or a BINDER OBJECT, a operation is defined to cast an ATOMIC OBJECT to a VARIABLE OBJECT, a CONSTANT OBJECT or a BINDER OBJECT. The cast operation, takes the ATOMIC OBJECT and the class name of the class which the ATOMIC OBJECT must be cast to. This cast operation is needed to determine which Ltype rule must be called.

- where: The where operation states which object is meant if more objects can be chosen. It also uses the compare symbol to determine which object is meant.

- assign: We define the assign operation. The assign operation assigns a value to a field of a new object or assigns an ELI list to be increased or decreased with an ELI OBJECT. The first parameter of the assign operation is the field of the new object or the ELI list to be changed. The second is the value that is assigned to the first parameter.

Ltype rules for the linguistic object tree

LINGUISTIC OBJECT
In the Ltype rules, always checks on specific LINGUISTIC OBJECTS are verified, therefore the LINGUISTIC OBJECT itself has no Ltype rule.

ATOMIC OBJECT
The ATOMIC OBJECT is used in Ltype rules, on places where VARIABLE OBJECTS, CONSTANT OBJECTS or BINDER OBJECTS can occur. Depending on which object occurs, the appropriate Ltype rule must be called.

```
case(class(obj) = VARIABLE OBJECT)
        Ltypecheck(cast(obj, VARIABLE OBJECT), Term/Set/Noun/Adj/Stat)
case(class(obj) = CONSTANT OBJECT)
        Ltypecheck(cast(obj, CONSTANT OBJECT), Term/Set/Noun/Adj/Stat)
case(class(obj) = BINDER OBJECT)
        Ltypecheck(cast(obj, BINDER OBJECT), Term/Set/Noun/Adj/Stat)
end case
-----------------------------------------------------------------------------------
        Ltypecheck(obj, Term/Set/Noun/Adj/Stat) where (class(obj) = ATOMIC OBJECT)
```

VARIABLE OBJECT
The var derivation rule states that:
The entity must be declared in the context, furthermore the entity must be a variable and may have Ltype Term, Set or Stat corresponding to the way in which the variable was declared.

For the VARIABLE OBJECT this comes down to finding an ELI object in the EL that has the same name. The EL represents the variables declared in the context, and all ELI objects in the EL are variables. Determining the Ltype of the variable is done by reading the out field of the ELI object that is constructed when the variable was declared. This Ltype is checked to be the same as the Ltype of the Ltypecheck operation below the dotted line.

```
e∈ EL where (e.name = obj.name)
check(e.out, Term/Set/Stat)
-----------------------------------------------------------------------------------
            Ltypecheck(obj, Term/Set/Stat) where (class(obj) = VARIABLE OBJECT)
```

note: Ltype rules that check whether a variable has Ltype Noun or Adj are allowed, but these checks will automatically fail. Variables can only have Ltypes Term, Set or Stat.

CONSTANT OBJECT
As discussed in this section when introducing the Essential Ltype Information object, only the ext-cons derivation rule needs to be used for checking a constant.
The entity needs to be defined in the preface of the book, the entity needs to be a constant and all parameters of the entity must have the Ltypes that are defined in the preface. The Ltype of the entity is also defined in the preface.

For the CONSTANT OBJECT this comes down to finding an ELI object in the EG that has the same name and the category of the ELI object must be `constant`. Instead of directly checking the parameters of the constant, we choose to first check the number of parameters. Then all parameters are checked. A case operation is needed to decide which Ltype rule is needed to check the parameters. This leaves checking the Ltype of the CONSTANT OBJECT.

```
e∈ EG where (e.name = obj.name)
check(e.category, constant)
check(e.arity, obj.parameters.length)
case(class(obj.parameters.i), ATOMIC OBJECT)
        Ltypecheck(cast(obj.parameters.i, ATOMIC OBJECT), e.in.i)
case(class(obj.parameters.i), COMBINATION OBJECT)
        Ltypecheck(cast(obj.parameters.i, COMBINATION OBJECT), e.in.i)
end case (i = 1, . . . , obj.parameters.length)
check(e.out, Term/Set/Noun/Adj/Stat)
--------------------------------------------------------------------------------
    Ltypecheck(obj, Term/Set/Noun/Adj/Stat) where (class(obj) = CONSTANT OBJECT)
```

BINDER OBJECT

The bind derivation rule states that:
The entity needs to be defined in the preface of the book, the entity needs to be a binder, the local declaration of the entity must be a correct declaration and all parameters of the entity must have the Ltypes that are defined in the preface. The Ltype of the entity is also defined in the preface.

For the BINDER OBJECT this comes down to finding an ELI object in the EG that has the same name and the category of the ELI object must be `binder`. The DECLARATION OBJECT of the BINDER OBJECT must be checked, before the parameters are checked in the same way as in the constant Ltype rule. Then the Ltype of the BINDER OBJECT is checked and the introduced local variable declared by the DECLARATION OBJECT is removed from the EL.

```
e∈ EG where (e.name = obj.name)
check(e.category, binder)
Ltypecheck(cast(obj.declaration, DECLARATION OBJECT), Decl)
check(e.arity, obj.parameters.length)
case(class(obj.parameters.i), ATOMIC OBJECT)
        Ltypecheck(cast(obj.parameters.i, ATOMIC OBJECT), e.in.i)
case(class(obj.parameters.i), COMBINATION OBJECT)
        Ltypecheck(cast(obj.parameters.i, COMBINATION OBJECT), e.in.i)
end case (i = 1, . . . , obj.parameters.length)
check(e.out, Term/Set/Noun/Adj/Stat)
assign(EL, EL \ [eli]) where (eli.name = obj.declaration.left.name)
--------------------------------------------------------------------------------
      Ltypecheck(obj, Term/Set/Noun/Adj/Stat) where (class(obj) = BINDER OBJECT)
```

COMBINATION OBJECT

The comb derivation rule states that:
A combination entity consists of a list of entities, where the last entity has Ltype Noun and all preceding entities have Ltype Adj.

For the COMBINATION OBJECT this comes down to checking that all elements in the LINGUISTIC OBJECT list have Ltype Adj, except the last element of the list, that must have Ltype Noun.

```
Ltypecheck(cast(obj.list.length, CONSTANT OBJECT),Noun)
Ltypecheck(cast(obj.list.i, CONSTANT OBJECT), Adj) (i = 1, . . . , obj.list.length-1)
------------------------------------------------------------------------------------
            Ltypecheck(obj, Noun) where (class(obj) = COMBINATION OBJECT)
```

note: Ltype rules that check whether a combination has Ltype Term, Set, Adj or Stat are allowed, but these checks will automatically fail. A combination always has Ltype Noun.

DECLARATION OBJECT

A declaration has two derivation rules. Both rules, set/stat-decl and term-decl derivation rules, are represented in one Ltype rule for checking the DECLARATION OBJECT. Both rules state that by a declaration of an x as an element of the class of all sets, as an element of the class of all stats or as an element of a set type, x needs to be checked to have the form of a variable and whether the name x is not already used for another variable. When x is declared as element of a set type, the set type needs to be checked to have Ltype Set or Noun.

The DECLARATION OBJECT does not have to check if the left hand side is a variable, because this is already checked by the syntax rules. Checking that the name is not already used means that there may not be an ELI OBJECT in the EG or in the EL that has the same name as the variable to be declared. A case operation decides, depending on the right hand side of the declaration whether an extra check is required. The Ltype of the declared variable is stored for usage by the Ltype rule that checks VARIABLE OBKECTS, therefore the DECLARATION OBJECT constructs an ELI OBJECT and adds it to the EL.

```
e∉ EG ∪ EL where (e.name ≠ obj.left.name)
eli where (class(eli) = ELI OBJECT)
assign(eli.name, obj.left.name)
assign(eli.category, variable)
case(class(obj.right), PROPSORT OBJECT)
        assign(eli.out, Stat)
case(class(obj.right), SETSORT OBJECT)
        assign(eli.out, Set)
case(class(obj.right), ATOMIC OBJECT)
        Ltypecheck(cast(obj.right, ATOMIC OBJECT), Set/Noun)
        assign(eli.out, Term)
case(class(d.right), COMBINATION OBJECT)
        Ltypecheck(cast(obj.right, COMBINATION OBJECT), Set/Noun)
        assign(eli.out, Term)
end case
assign (EL, EL ∪ [eli])
------------------------------------------------------------------------------------
            Ltypecheck(obj, Decl) where (class(obj) = DECLARATION OBJECT)
```

SORTS OBJECT, PROPSORT OBJECT and SETSORT OBJECT

The SORTS OBJECT , PROPSORT OBJECT and SETSORT OBJECT have no derivation rule and no specific Ltype checks.

DEFINITION OBJECT

The int-def derivation rule states that:
The left hand side of the definition must be a constant, this constant may not yet be defined in the preface or in the book. All variables declared in the context must be, in the same order, defined as parameters of that constant. The right hand side of the definition must have a valid (basic) Ltype.

The DEFINITION OBJECT does not have to check if the left hand side is a constant, because this is already checked by the syntax rules. Checking that the name is not already used means that there may not be an ELI OBJECT in the EG or in the EL that has the same name as the variable to be declared. All variables declared in the context must, in the same order, appear as parameters of the constant to be defined. A parameter already has a wrong name if it is not a VARIABLE OBJECT, because only VARIABLE OBJECTS are constructed in the context. This check is needed to determine if the linguistic object has a name field.
The Ltype of the right hand side is checked to be correct, by using that the right hand side of the DEFINITION OBNJECT is an *argument*. An *argument* has the form of an ATOMIC OBJECT or of an COMBINATION OBJECT. The Ltype of the defined constant that is defined is stored for usage by the Ltype rule that checks CONSTANT OBJECTS, therefore the DEFINITION OBJECT constructs an ELI OBJECT and adds it to the EG.

```
e∉ EG ∪ EL where (e.name = obj.left.name)
check(obj.left.parameters.length, EL.length)
check(class(obj.left.parameters.i) = VARIABLE OBJECT))                    >  (i = 1,…,EL.length)
check(cast(obj.left.parameters.i), VARIABLE OBJECT).name, EL.i.name)
eli where (class(eli) = ELI OBJECT)
assign(eli.name, obj.left.name)
assign(eli.category, constant)
assign(eli.arity, EL.length)
assign(eli.in.i, EL.i.out) (i = 1,...,EL.length)
case(class(obj.right), ATOMIC OBJECT)
        Ltypecheck(cast(obj.right, ATOMIC OBJECT), Term/Set/Noun/Adj/Stat)
case(class(d.right), COMBINATION OBJECT)
        Ltypecheck(cast(obj.right, COMBINATION OBJECT), Term/Set/Noun/Adj/Stat)
end case
assign(eli.out, Term/Set/Noun/Adj/Stat)
assign (EB, EB ∪ [eli])
-----------------------------------------------------------------------------------------
                Ltypecheck(obj, Def) where (class(obj) = DEFINITION OBJECT)
```

CONTEXT OBJECT

A context has two derivation rules. One derivation rules for the empty context, which will be checked by the EMPTY CONTEXT OBJECT. The assump/decl-cont derivation rule states:
A context consists of declarations or statements, therefore we need a rule that refers to the already defined declaration and statement rules.

For the CONTEXT OBJECT, which only represents the context list, this comes down to checking that the LINGUISTIC OBJECTS at their left hand side and right hand side have a correct Ltype. First the left hand side must be checked, to guarantee correctness by construction, the left hand side can be another CONTEXT OBJECT, a DECLARATION OBJECT or a ATOMIC OBJECT. The right hand side can be a DECLARATION OBJECT or a ATOMIC OBJECT. All objects are checked by their corresponding Ltype rule.

```
case(class(obj.left), CONTEXT OBJECT)
        Ltypecheck(obj.left, Cont)
case(class(obj.left), DECLARATION OBJECT)
        Ltypecheck(obj.left, Decl)
case(class(obj.left), ATOMIC OBJECT)
        Ltypecheck(obj.left, Stat)
end case
case(class(obj.right), DECLARATION OBJECT)
        Ltypecheck(obj.right, Decl)
case(class(obj.right), ATOMIC OBJECT)
        Ltypecheck(obj.right, Stat)
end case
```

Ltypecheck(obj, **Cont**) where (class(obj) = CONTEXT OBJECT)

EMPTY CONTEXT OBJECT

The emp-cont derivation rule only states that the book must be checked.
The book is correct by construction, therefore the EMPTY CONTEXT OBJECT is correct.

LINE OBJECT

the assert/def-line derivation rule has a similar form as the assump/decl-cont derivation rule. Therefore the Ltype rule also has a similar form. The assert/def-line derivation rule states that the left hand side of the line must be a correct context and that the right hand side of the line must be a correct definition or a correct statement.

For the LINE OBJECT this comes down to checking all objects possible as its left hand side to have the correct Ltype. There are more possible objects, due to the different modeling of the CONTEXT OBJECT. The left hand side can be an EMPTY CONTEXT OBJECT, a DECLARATION OBJECT, an ATOMIC OBJECT or a CONTEXT OBJECT. The right hand side of the LINE OBJECT can be a DEFINITION OBJECT or an ATOMIC OBJECT. All objects are checked by their corresponding Ltype rule.

```
case(class(obj.left), EMPTY CONTEXT OBJECT)

case(class(obj.left), DECLARATION OBJECT)
        Ltypecheck(obj.left, Decl)
case(class(obj.left), ATOMIC OBJECT)
        Ltypecheck(obj.left, Stat)
case(class(obj.left), CONTEXT OBJECT)
        Ltypecheck(obj.left, Cont)
end case
case(class(obj.right), DEFINITION OBJECT)
        Ltypecheck(obj.right, Def)
case(class(obj.right), ATOMIC OBJECT)
        Ltypecheck(obj.right, Stat)
end case
```

Ltypecheck(obj, **Cont**) where (class(obj) = LINE OBJECT)

BINARY TREE OBJECT

In the Ltype rules, always checks on specific BINARY TREE OBJECTS are verified, therefore the BINARY TREE OBJECT has no Ltype rule.

Notes

- Numbers are supposed to be in the preface, but one cannot add all numbers to the preface. If a CONSTANT OBJECT is a number the constant Ltype rule checks that the constant has no parameters and that the requested Ltype is Ltype Term.

- Similar to the note in section 2.7, the right hand side of the LINE OBJECT is extended to accept COMBINATION OBJECTS. The LINE OBJECT will return the Ltype of its right hand side or if that right hand side is a DEFINITION OBJECT, the Ltype of the right hand side of that DEFINITION OBJECT. The program returns this Ltype, and in case of a definition or a statement asks the user if the line should be added to the book.

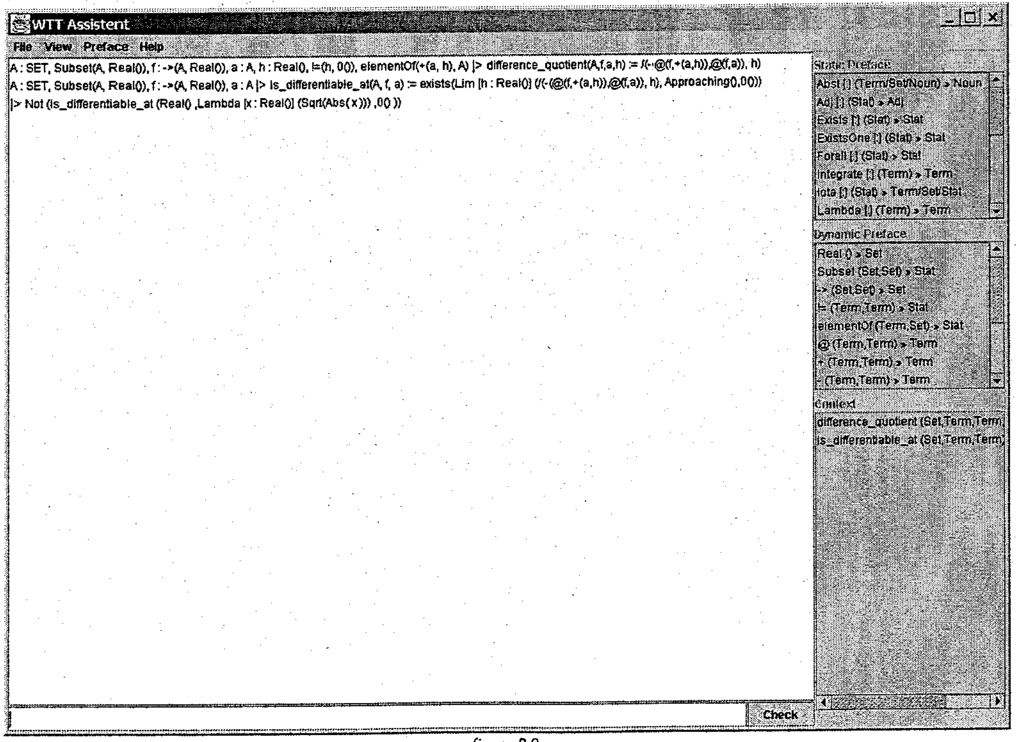
- The use of only one constant rule and the correctness by construction is a result of the translation of the WTT language to a computer implementation.

The program checking the Ltype rules can be found in appendix D.
The implementation of the core WTTA program is described in the next section

2.9 Implementing the core program

In this section we will describe the interface of the core computer program. The program uses the unsugared language defined in table 2.2a. A line written in the unsugared language can be entered into the program. The program will translate that input into a linguistic object tree. The linguistic object tree will be syntax checked and checked with the Ltype rules. The program returns a Ltype if the input is found to be correct. In case of a definition or a statement the program asks the user if the line should be added to the book.

Figure 2.9a is a screenshot of the program, showing the different parts of the program.



```
A : SET, Subset(A, Real()), f : ->(A, Real()), a : A, h : Real(), !=(h, 0()), elementOf(+(a, h), A) |> difference_quotient(A,f,a,h) := /(-(@(f,+(a,h)),@(f,a)), h)
A : SET, Subset(A, Real()), f : ->(A, Real()), a : A |> is_differentiable_at(A, f, a) := exists(Lim [h : Real()] (/(-(@(f,+(a,h)),@(f,a)), h), Approaching(),0()))
|> Not (is_differentiable_at (Real() ,Lambda [x : Real()] (Sqrt(Abs( x))) ,0() ))
```

```
Abst [] (Term/Set/Noun) > Noun
Adj [] (Stat) > Adj
Exists [] (Stat) > Stat
ExistsOne [] (Stat) > Stat
Forall [] (Stat) > Stat
Integrate [] (Term) > Term
Iota [] (Stat) > Term/Set/Stat
Lambda [] (Term) > Term
```

```
Real () > Set
Subset (Set,Set) > Stat
-> (Set,Set) > Set
!= (Term,Term) > Stat
elementOf (Term,Set) > Stat
@ (Term,Term) > Term
+ (Term,Term) > Term
- (Term,Term) > Term
```

```
difference_quotient (Set,Term,Term
is_differentiable_at (Set,Term,Term)
```

figure 2.9a

Views
Our program must show the user which constants and binders can be used in the book, at the same time it must show the user the book in which a line can be added. This results in four views that the program has to show:

1)	The static preface	(Binders and Constants for all books)
2)	The dynamic preface	(Constants for this particular book)
3)	The defined constants	(Constants defined in the book)
4)	The book	

The first three views can be hidden from the user by options in the menu bar (see figure 2.9c).

Input
The main objective of the program is to enable the user to enter lines that can be added to the book, if they are WTT correct. Therefore our program has an input field where the input can be typed. To the right of the input field a 'check' button is placed. When the button is pressed the text in the input field is checked by the program.

Dynamic preface
The dynamic preface can be different for every book, therefore the dynamic preface can be saved under a name (and is automatically saved if the program is exited) and opened.
An item can be added or deleted to or from the preface. Removing an item from the preface or opening another preface can invalidate the book. In this version of the program, removing an item and opening a preface can only be done if the book is empty.

Because the dynamic preface, in this version, is not linked to the book, the wrong preface can be opened when a book is loaded. To avoid that a book is invalid when it is loaded, the book is checked line by line when a book is opened.

All options are available in the menu bar, see figure 2.9d.

Book
The book can be loaded, saved, saved under another name or closed, and lines can be added or deleted to or from the book. Only the last line of a book can be deleted, because this can be done without invalidating the book. Adding a line to the book is done by writing a line in the input field and pressing the 'check' button.
The options for the book are depicted in figure 2.9b.

Message line
In the bottom of our program a message line is visible. The program calculates the Ltype of every input line. This Ltype is reported in the message line. The user can choose, by an option in the help menu (see figure 2.9e), if this message should also be shown by a popup message. A popup message automatically appears if the Ltype is Stat or if the line is a definition. This popup message asks the user if the line should be added to the book.

Menu bar

The menu bar has four menu's: the file menu, the view menu, the preface menu and the help menu. These four menu's are depicted, in the same order, by the following figures.

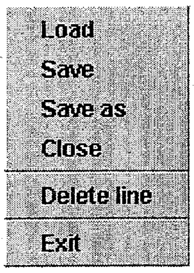


figure 2.9b

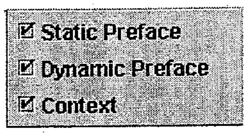


figure 2.9c

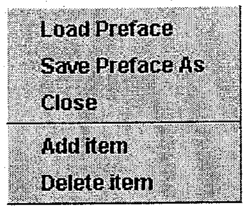


figure 2.9d

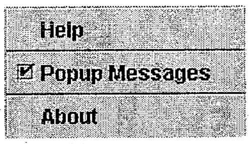


figure 2.9e

The only options not yet discussed are the 'exit' option in the file menu, that closes the program, the 'help' option in the help menu that opens a window that depicts a small help list (see figure 2.9h) and the 'about' option in the help menu that depicts a small window with some information about the program.

The menu options of the book, all options except the 'exit' option in the file menu, can also be addressed by right clicking the mouse in the book view.
The same can be done for the options of the preface, the options in the preface menu, by right clicking in the dynamic preface view.

Assistance

Some assistance is added to the program.

- A user can double click on a constant or binder defined in the preface or book, and the constant or binder name is written in the input line, including the brackets and the parameter list separators. E.g. 'Plus(,)'

- A user orders the program to check a text typed in the input field by pressing the 'check' button. The user can also press the <enter> key on the keyboard when he finishes with typing the text in the input field, this has the same effect as pressing the 'check' button.

- If the input is not WTT correct an error is generated by one of the checks. The cursor is placed on the place where the error occurred. This is possible, because the linguistic objects store the positions of the input which they represent.

- To let the user define an item that is stored in the dynamic preface the window depicted in figure 2.9f is constructed:

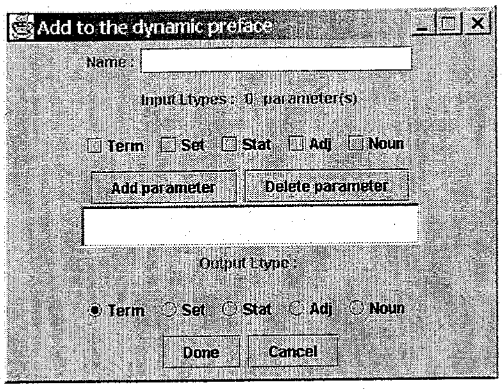
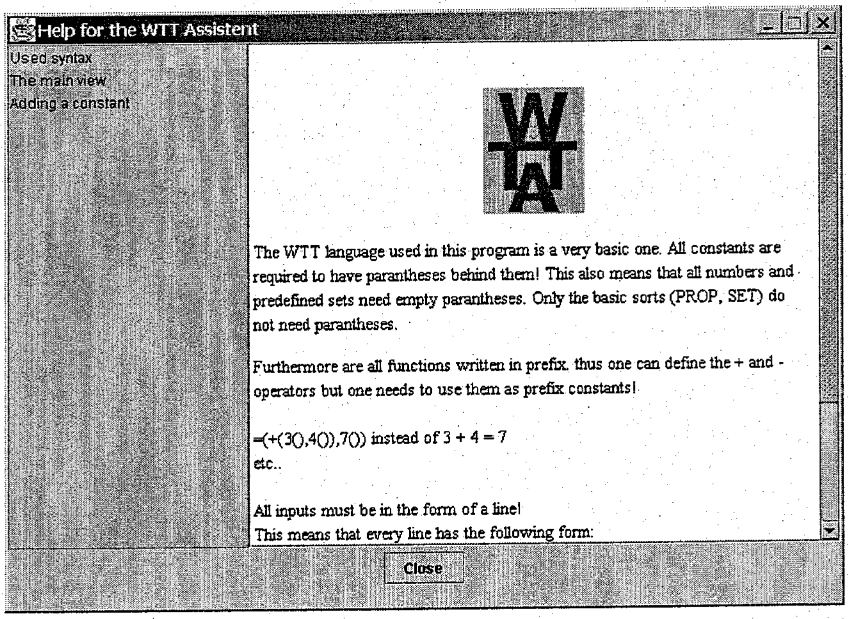


figure 2.9f

In this window the user can enter the information that will be stored in an ELI object. The user can choose to permit parameters of different Ltypes. Similar to the Abst binder which has one parameter that can have Ltype Term, Set or Noun. More information about this option can be found in chapter 6.

- A help window, shown in figure 2.9h, is available, but the help provided is small and is constricted to a page about the unsugared syntax used and two pages about the interface.



The WTT language used in this program is a very basic one. All constants are required to have parantheses behind them! This also means that all numbers and predefined sets need empty parantheses. Only the basic sorts (PROP, SET) do not need parantheses.

Furthermore are all functions written in prefix. thus one can define the + and - operators but one needs to use them as prefix constants!

=(+(3(),4()),7()) instead of 3 + 4 = 7
etc..

All inputs must be in the form of a line!
This means that every line has the following form:

figure 2.9h

Which concludes the brief overview of the core version of the program.
In the next chapter features will be added to the core program to enhance the usability and the assistance for the user. The final program will be discussed in more detail in section 3.3.

3 Extending the core program

In this chapter the implemented changes requested by the user group, who tested the core program, are discussed. The changes are divided in changes to the language and changes to the interface of the core program.
First the changes to the language and their effect on the program are discussed, then the additional functionality that had to be developed to change the interface is discussed.

The changes to the language allow the user to enter a more user friendly language into the program, while the changes to the interface allow greater manipulation and representation of the lines shown by the program.

In the last section of this chapter the new interface of the final program will be discussed.

3.1 Changes to the language

As discussed in section 2.1, the unsugared language definition changes to a more sugared language. Not all features adding sugar to the language were added. The Features added are:
1) Infix constants,
2) Extra parentheses,
3) Skipping parentheses,
3) Multiple declarations,
4) Adding comments.

The sugared language is parsed and a linguistic object tree is constructed. This linguistic object tree is checked in the same way as a linguistic object tree has been built in the basic program. Therefore the construction of the linguistic object tree must remove all sugar from the language, which is conform to the "de Bruijn criterion" [Bar 2001].

Every addition of sugar to the language requires changes to the parser and/or the program building the linguistic object tree. The language definition of the sugared language can be found in appendix E.

3.1.1 Infix constants

Infix constants are used standard in all kinds of mathematical texts. Users of the program are accustomed to reading and writing infix constants instead of prefix constants.

The main difference between prefix and infix constants is that, the order in which prefix constants are evaluated is not ambiguous, while infix constants may be ambiguous.
For Example +(2, *(3,4)) will first evaluate the * constant, before the + constant can be evaluated, however there are two ways for reading 2 + 3 * 4. First evaluating + constant before evaluating the * constant or evaluating it as described in the prefix example.

Therefore infix constants have extra information to determine which infix constant is evaluated first. This information is the priority and the associativity of the infix constant.
If two constants have the same priority the order will be decided by the associativity of the constants else the priority decides which infix constant is evaluated first.
How priority and associativity determine the order in which infix constants must be evaluated is explained when the LINGUISTIC OBJECTS, that represent the infix constants, are constructed.

Changes
Adding infix constants to the language does not only change the language, but also changes the information we need to store about the constants. This means extending the ELI OBJECTS with priority and associativity. To keep the first basic version of the program we will not change the ELI OBJECTS but define a new object, the Additional Linguistic Information object (ALI OBJECT). This object inherits every field from the ELI OBJECT, and extends those fields with two other fields, one for priority and one for associativity.
If a constant with two parameters is defined as an infix constant, the constant will get a standard priority and associativity. The user must be able to change the priority and associativity, which requires a new window for our program (see section 3.3).

The program that builds the linguistic object tree needs to know the priority and associativity of the constants already defined, in order to know which constant should be evaluated first. This constant, that is evaluated first, is represented as the top CONSTANT OBJECT, while the other constants, also represented by CONSTANT OBJECTS, are added as parameters of the top CONSTANT OBJECT.
Therefore all ELI OBJECTS that represent the constants and binders in the preface of the book are made available to the program that builds the linguistic object tree.

Parsing and building
Function (1) (see section 2.5.2) of the program that builds the linguistic object tree is changed in such a way that if after the construction of the first LINGUISTIC OBJECT a *name* identifier is found not always a COMBINATION OBJECT is constructed. The preface and already defined constants are searched for an ELI OBJECT that, has the same name as the *name* identifier.
If an ELI OBJECT is found and this ELI OBJECT is extended by an ALI OBJECT, which means that the *name* identifier found is an infix constant, a CONSTANT OBJECT is constructed instead of a COMBINATION OBJECT. This CONSTANT OBJECT represents the *name* identifier, the first LINGUISTIC OBJECT is added as its first parameter and a the third LINGUISTIC OBJECT that is constructed is added as the second parameter of the CONSTANT OBJECT.

If after the third LINGUISTIC OBJECT another *name* identifier is parsed, and this *name* identifier is again found in the preface or defined constants as an infix constant, then the priority and associativity stored in both ALI objects determine which way the third LINGUISTIC OBJECT is divided between the two CONSTANT OBJECTS that represent the infix constants.

Determining how the infix constants must be interpreted is done by two help functions.
If function (1) (see section 2.5.2) finds an infix constant and two parameters, function (i) is called. Function (i) takes three arguments, the first parameter, the infix constant and the second parameter.

Function (i) Function (i) tries to parse another infix constant and another parameter behind the already given arguments. If this does not succeed function (i) will add both parameters to the infix constant and return it .
E.g. 'p $ q' is parsed and function (i) is called to construct the infix constant.

```
                           $                no other infix constant is found.
function (i) (p, $, q) =  / \
                         p   q
```

If function (i) finds another infix constant and another parameter it uses a simple mechanism to determine how to interpret these three parameters and two infix constants as is shown in the next example.

Function (i) uses the next cases to decide how to proceed if another infix constant and another parameter are parsed behind its arguments, e.g. '# r' after 'p $ q'.
The cases in this example are based on infix constants '$' and '#'.
case a) $ higher priority than #   or $ same priority as # and $ left associative
case b) $ lower priority than #   or $ same priority as # and $ right associative

In case a the first two parameters are added to the first infix constant.
Then this infix constant, the second infix constant and the third parameter are used as arguments for a recursive call of function (i) to determine if more infix constants and parameters can be found.

```
                                              $                      '# r' found and case a
function (i) (p, $, q)  = function (i) (  /     \  , #, r)                  on '$' and '#'
                                          p     q
```

In case b function (i) is also called recursively, but this time with the second parameter, the second infix constant and the third parameter as arguments.
This will construct an infix constant for all parameters and infix constant that are behind the first infix constant.
The result of this recursive call, together with the first parameter and the first infix constant are used as argument for function (ii).

```
                                                                     '# r' found and case b
function (i) (p, $, q)  = function (ii) (p, $, function (i) (q, #, r))      on '$' and '#'
```

Function (ii) Function (ii) adds the first parameter and the first infix constant to the already constructed infix constant that is parsed to the right of it.
Function(ii) asks for the priority and associativity of the first infix constant and of the infix constant of its third argument.
Using the same cases, as described before, again a decision can be made.

In case b the first infix constant is to weak and will be put on top of the already parsed infix constant. In other words function (ii) adds the first argument and its third argument as parameters of the first infix constant and then returns it.

```
                                  $                                  case b
                      %          /  \                         on '$' and '%'
function (ii) (p, $,  /   \ )  =  p     %
                     A    B            /  \
                                      A    B
```

In case a the left parameter of the already constructed infix constant is removed from it. Function (ii) is called with the first parameter, first infix constant and this left parameter as its argument. The result of this function call is added as the new left parameter of the already constructed infix constant.

```
                      %                           %                  case a
function (ii) (p, $,  /   \ )  =                 /   \          on '$' and '%'
                     A    B     function (ii) (p, $, A)    B
```

The first and third argument are added as parameters of the infix constant if the third argument of function (ii) is not an infix constant.

```
                               $                          A is no infix constant.
function (ii) (p, $, A) =     / \
                             p   A
```

Function (i) and function (ii) construct the infix constants in such a way that the top infix constant has the weakest priority or in case of equal priority, is right associative. This holds for every parameter of that infix constant recursively.

Therefore, in function (ii), the first parameter is substituted if the infix constant to be added has stronger priority or in case of equal priority is left associative. Where inside the first parameter the infix constant is added depends on the priority and associativity of that infix constant.

This concludes the algorithm used to interpret infix constant.

Consequences of adding infix constants
If infix constants can be defined in the preface then, for similar reasons, it should also be possible to define infix constants in the book.

The problem is that the ALI OBJECT that represents the infix constant is constructed in the definition, and therefore is not known when the definition is checked.
We decide that for defining an infix constant, the constant at the left hand side of the definition *identifier* must be written in infix. Therefore the right hand side of such an infix definition consists of three *name* identifiers. For example : a div b := RoundDown(a/b).

Three name identifiers are, by function (1) (see section 2.5.2), translated to three VARIABLE OBJECTS and then put in a COMBINATION OBJECT.
When a DEFINITION OBJECT is constructed by the program and the left hand side of this DEFINITION OBJECT is a COMBINATION OBJECT, the program tries to determine if this COMBINATION OBJECT is in fact an infix definition.

If two variables are declared in the context, the LINGUISTIC OBJECT list of the COMBINATION OBJECT has three elements and the second of these LINGUISTIC OBJECTS is a VARIABLE OBJECT, then the COMBINATION OBJECT is replaced by a CONSTANT OBJECT.
The first and third LINGUISTIC OBJECT in the list of the COMBINATION OBJECT are added as parameters of the CONSTANT OBJECT. The second LINGUISTIC OBJECT in the list supplies the name of the CONSTANT OBJECT.

If it was not a infix constant the syntax or Ltype checks will generate an error, but a VARIABLE OBJECT in a COMBINATION OBJECT would also generate an error.

Another consequence of adding infix parameters is the need for parentheses to override the priority of infix constants. The introduction of parentheses is discussed in section 3.1.2

Notes: A constant defined as infix constant can still be used as prefix constant. This is done for compatibility with the core program.

Defining infix constants also changes the verifying program in a very small way.
If an infix constant is defined, an ALI OBJECT will be constructed instead of an ELI OBJECT.

3.1.2 Extra parentheses

Parentheses are used in mathematical texts to overwrite priority of infix constants. Every expression between parentheses has higher priority and must be evaluated first.

The introduction of parentheses requires no extra information, the difference is that the program must take care that all parentheses opened also must be closed.

Changes

The parentheses are meant for overruling priority of infix constants, thus we can restrict the use of extra parentheses by only permitting them for constants. Parentheses however, can also be used to increase legibility of a text. For example parentheses around the different parameters in a parameter list or around an expression beginning with a binder.
We therefore allow parentheses around *variables, constants, binders* and *combinations.*

Parsing and building

When the parser returns a '(' *identifier*, instead of a *name* identifier, a help function is called. This help function uses function (1) (see 2.5.2) to construct a LINGUISTIC OBJECT. If function (1) is finished a ')' *identifier* must be parsed. The help function that parses the parentheses does not allow parentheses around sort *identifiers*, only around *name* identifiers.
The ATOMIC OBJECT and COMBINATION OBJECT gain a special field that says if there are parentheses around the LINGUISTIC OBJECT for representation purposes.

Consequences of extra parentheses

Adding extra parentheses can increase legibility of a text, but skipping other parentheses can also add to the legibility of a text. Especially for constants that have no parameters such as the *numbers.* Skipping parentheses is discussed in the next section.

3.1.3 Skipping parentheses

Parentheses behind numbers or behind definitions of sets add to the text to be typed, but have no value. Skipping those parentheses should be permitted.

By adding infix notation, the need of parentheses for infix constants is superfluous. The priority and associativity replace the parentheses. This can also be used by unary constants. Unary constants have only one parameter. This parameter can be written behind it without the use of parentheses.

Changes

Information for constants with no parameters and information for unary constants can be saved in the already introduced ALI OBJECTS.

Constants with no parameters do not need a priority or an associativity, because they have no parameters. The use of the ALI OBJECTS only allows these constants to have no parentheses. Examples of these constants are '3' instead of '3()' or 'Nat' instead of 'Nat()'.

Unary constants always have a priority that is higher then the priority of a infix constant. This ensures that the LINGUISTIC OBJECT behind the unary constant automatically is chosen as the parameter of the unary constant.[10] The use of extra parentheses can override this priority.
For example: '– 3' instead of '–(3)' or 'Not a' instead of 'Not(a)'.

[10] A unary constant can also be postfix but only prefix unary constants are allowed here (see also chapter 4 Postfix)

Parsing and building
Constants with no parameters are parsed as normal, the information of the ALI OBJECT decides if a CONSTANT OBJECT or a VARIABLE OBJECT must be constructed. *Number* names are automatically translated to CONSTANT OBJECTS.

If a unary constant is parsed, known by the ALI OBJECT, the program will construct another ATOMIC OBJECT and add this object as the parameter of the unary constant.

Consequences of skipping parentheses
Skipping parentheses after constants with no parameters and after unary constants must also add the possibility to define constants with no parameters and unary constants, without parentheses.

A constant to be defined must have a fresh constant name. The parser will recognize this name as the name of a variable.

A CONSTANT OBJECT with no parameters is constructed, when a DEFINITION OBJECT IS constructed with a VARIABLE OBJECT as its left hand side.

For defining a unary constant the COMBINATION OBJECT, again (see infix constants), is used.
If a COMBINATION OBJECT is constructed as the left hand side of a ':=' *identifier* and only one variable is declared in the context, the COMBINATION OBJECT has two LINGUISTIC OBJECTS and the first of these LINGUISTIC OBJECTS is a variable, then this COMBINATION OBJECT is replaced by a CONSTANT OBJECT with one parameter.
The second LINGUISTIC OBJECT in the list of the COMBINATION OBJECT, is added as the parameter of the CONSTANT OBJECT and the first LINGUISTIC OBJECT in the list supplies the name of unary constant.

Defining a constant with no parameters or a unary constant means that, instead of an ELI OBJECT, an ALI OBJECT is constructed when the definition is Ltype checked.

3.1.4 Multiple declarations

In some contexts multiple variables are declared as elements of the same set type, or the variables are declared as elements of the same sort.
Instead of declaring the variables one by one, a multiple declarations form is introduced.
For example: 'a : PROP, b : PROP, c : PROP' is written as 'a; b; c : PROP'

Changes
the ';' *identifier* is added as *predefined* symbol. Adding the ';' identifier as a *predefined* symbol further restricts the names allowed for variables and constants. The *predefined* symbols are declared in the parser and will be returned to the program that constructs the binary object tree.

Parsing and constructing
If after the construction of a LINGUISTIC OBJECT by function (1) (see 2.5.2), function (2) is called and a *';' identifier* is found, a help function is called. This help function parses and constructs a part of the linguistic object tree which represents the multiple declarations.

All LINGUISTIC OBJECTS separated by *';' identifiers* are parsed and constructed by function (1). These LINGUISTIC OBJECTS are stored in a list. When the *':' identifier* is returned by the parser, function (1) is called to construct the LINGUISTIC OBJECT that represents the right hand side of the multiple declarations. The help function constructs DECLARATION OBJECTS that have as left hand side an item from the stored list and as right hand side the LINGUISTIC OBJECT constructed to represent the right hand side of the multiple declarations.

The constructed DECLARATION OBJECTS must be added to the linguistic object tree, in the same way as the unsugared translation of the multiple declarations would be added by the core program.

There are two different cases:
- The multiple declarations is the first declaration in the context.
  The linguistic object tree is constructed with the first DECLARATION OBJECT as the left hand side of a CONTEXT OBJECT and the second DECLARATION OBJECT as the right hand side of that CONTEXT OBJECT. A second CONTEXT OBJECT takes the first CONTEXT OBJECT as its left hand side and adds the third DECLARATION OBJECT as its right hand side. This is repeated until all DECLARATION OBJECTS are added to the linguistic object tree.

- The multiple declarations occur somewhere in the middle of a context.
  The LINGUISTIC OBJECT already constructed before the multiple declarations, represents the beginning of the linguistic object tree. A CONTEXT OBJECT is constructed, that takes this LINGUISTIC OBJECT as its left hand side and the first DECLARATION OBJECT as its right hand side. Again a second CONTEXT OBJECT takes this CONTEXT OBJECT as its left hand side, while the second DECLARATION OBJECT is added as the right hand side. This is repeated until all DECLARATION OBJECTS are added to the linguistic object tree.

Consequences of multiple declarations
a DECLARATION OBJECT also occurs in an expression beginning with a binder. This does not mean that the binder has more bound variables, but that the expression beginning with a binder is another expression beginning with another binder.
As an example we rewrite the sugared text ‘$\forall_{x;\, y : \mathrm{Nat}} (x + y = x -- y)$’
to its unsugared version ‘$\forall_{x : \mathrm{Nat}} (\forall_{y : \mathrm{Nat}} (x + y = x -- y))$’

Every declaration, in the multiple declarations, constructs a BINDER OBJECT that is added as the first parameter of the BINDER OBJECT that was construct in front of it. All other parameters of the binder expression are the same for each BINDER OBJECT.
For example: $\mathrm{Lim}_{x;\, y : \mathrm{Nat}} (x^2 – y^2/(x - y),0,\downarrow)$ limits both x and y to 0 from above.

3.1.5 Adding comments

The user group requested that the language was extended in such a way that comments could be added to the text, in order to be able to describe the translated problem or describe lines as 'Lemmas' or 'Theorems'. Comments can also be used for adding references to the text translated into WTT or explain certain predefined constants in the preface.

Changes

Comments added to a line do not need to be checked or represented in the linguistic object tree, therefore the parser should skip the comments. But commands should also not be allowed to appear in every place, therefore comments are treated as white spaces. One exception has been made on this rule, comments can occur in *words* to add words to a constant name. In this way constant names used can stay closer to the mathematical description they were translated from. The reason for this exception is explained in section 4 , allow words to be written inside constants.

We define an extra group that can be recognised by the parser.

comment : ' " ' (letters/digits/symbol/<space>)* ' " '

The parser recognises the ' " ' symbol and will skip every identifier until it parses another ' " ' symbol. The ' " ' symbol is added to the *predefined* group so that it can only be used to open and close a comment.

Parsing and constructing

This option does not change the way in which the linguistic object tree is built. The difference is that the parser recognizes the ' " ' *identifier,* but does not return the comment to the program that constructs the linguistic object tree. When parsing a comment in a *word* the parser will not add the comment to the *identifier* that is returned to the program, therefore this also does not change the program that construct the linguistic object tree.

Consequences of comments

Besides comments in *lines,* it should also be possible to add an entire line of comment to a book. The line should be remembered and saved, but not checked. The parser will only return the end of line *identifier,* because the comment is skipped. If an end of line *identifier* is returned while no comment is recognised an error is generated.

The final program checks if a linguistic object tree is returned, if this is not the case, the program knows that the entire line is a comment and remembers the line.

In the next section we will discuss the changes to the interface.

3.2 Changes to the interface

The comments about the interface were mainly directed at the preface and book. The user group wanted more options for the preface and the book. Options for editing a line or item and importing lines or items from other books or prefaces were suggested.
Another major change was to represent the book in different views, especially the flag view described in CS-Report 02-05 [Ned 2002].
Apart from these major changes some minor changes were made, which are described in section 3.2.3. The functionally behind the adjustments of the interface are discussed in this section. The changes to the interface itself are depicted and described in section 3.3.

3.2.1 Preface and book

In order to enhance the control the user has on the preface and the book, the user group requested the following changes.

- The user must be able to create a new book, open an existing book, close the book that is currently used by the program, save the book that is currently used and save the book that is currently used under another name.
- These same options must be possible for the preface.
- The user must be able to add a line into a book, insert a line into a book, edit a line from a book and delete a line from the book.
- These same options must be possible for the items of a preface.
- The user must also be able to append lines from another book into the book that is currently used by the program and also insert those lines into the book currently used.
- It must also be possible to add and insert items of another preface into the preface that is currently used by the program.
- A preface must be linked to a book and if the book is opened, the preface must be opened automatically.
- Show the name of the book and the name of the preface currently used by the program.

These changes extended the 'file' and 'preface' menu of the interface, depicted in figure 3.3b and figure 3.3c.

Implementation points

Invalidating the book
Deleting, editing or inserting a line in the book can invalidate the book. This also holds for deleting, editing or inserting items of the preface.

The book and/or preface are stored before such an action is performed. Then the action is performed and the entire book is checked again. If the book is invalid the stored values will be restored and the action will be cancelled. An error message is generated and the program resumes. If the book still is valid, the action is permitted and the program resumes.

Some knowledge of the WTT language is used when checking if an action can be accepted. For example: deleting a line which has a statement on the right hand side of the line identifier, has no consequences for the validity of the book. It can have consequences when proving the correctness of the book in Type Theory, but for the WTT language, removing an assertion has no consequences.

Options for lines and items
For adding, editing and inserting a line the input field of the program is used. The button after the input field shows if the line entered will be added, inserted or edited.

Adding, editing or inserting an item in the preface is done by using the window for adding items to the preface in the core program (see figure 2.9f).

Adding, inserting lines and items
For inserting or adding lines from another book, and for inserting or adding items from another preface, a new window is introduced.
This window is discussed in section 3.3 and depicted in figure 3.3f .

Linking the preface to the book

Linking the preface to the book can be done by linking one book to one preface or by letting a book have multiple prefaces. In the latter way users can re-use prefaces for multiple books. The problem is that if the preface is changed for one book, the preface is changed for all books that use that preface, which can possibly invalidate other books. Those invalidated books must then be rewritten. We therefore choose to let every book only have one preface.
If a user wants to use items already defined in another preface, the user can add them by using the therefore added functionality.
The user himself is responsible for changing the preface, but in this way the user cannot change the preface from another book and thereby, by accident, invalidate that book.

3.2.2 Representations

Apart from showing the lines that were typed by the user, it should also be possible to show the lines in accordance with the unsugared language definition (see table 2.2a) and in a flag representation as used in CS-Report 02-05 [Ned 2002].
The view menu shown in figure 3.3j can be used to change the way in which the lines are represented.

Implementation points

Storing the lines entered in the program can be done easily, but representing them in another view, comes down to storing parts of that input and printing it in another way.
Therefore we have added a represents field to the LINGUISTIC OBJECT, so that every LINGUISTIC OBJECT can store the part of the input that this object represents. Adding functions to read this field and built a textual representation of the linguistic object tree allows us to define different views in which the input can be represented.

Unsugared representation

One of these representation views is the unsugared representation. In this view the input entered is represented according to the language definition of table 2.2a. In this unsugared language representation only input in accordance with the unsugared language definitions of table 2.2a can be entered into the program, with the exception of comments. Comments can also be entered in this representation of the entered input.

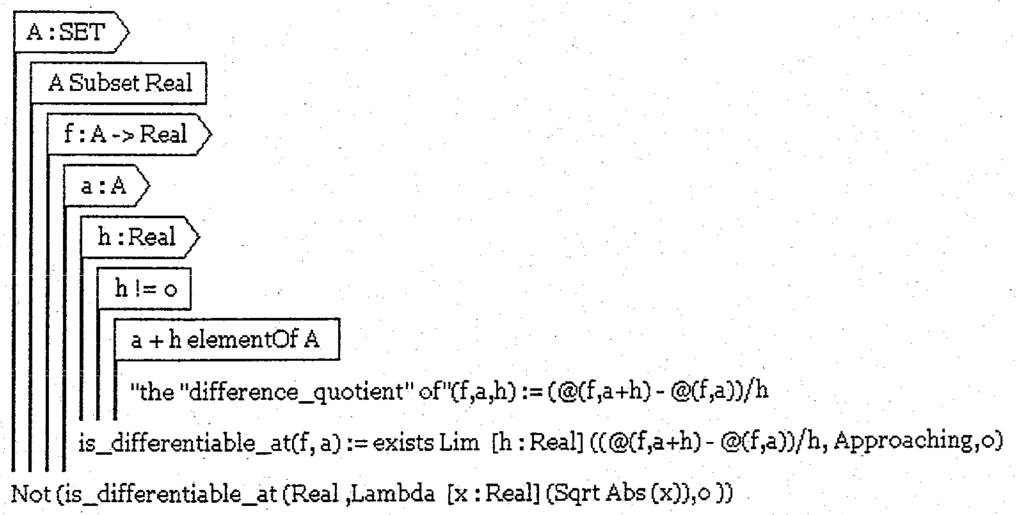
Flag representation

Another manner to view a WTT book is the flag representation. In this representation every object in the context is put in a flag and the right hand side of the line is added as text in these flags. When multiple lines use the same context, which often is the case, the flags are used for all these lines. In this way the user sees more easily the definitions and assertions, and the structure of the context under which these definitions and assertions are made.

We re-used code from M. Oostdijk, who wrote a program that showed a proof made in Coq, as a flag proof [Oost 1999], to represent the WTT book in flag representation. The use of flags in our program has nothing to do with proving a description, but is simply another way to represent the informal description in WTT.
We adepted the code of M. Oostdijk [Oost 1999] for our program and created a view for the flag representation of the WTT book.

An example of the flag representation can be seen in figure 3.2.2a.

"Example from CS-Report 02-05 'Weak Type Theory: A formal language for mathematics' section 4.3"



```
A : SET
 A Subset Real
  f : A -> Real
   a : A
    h : Real
     h != 0
      a + h elementOf A
       "the "difference_quotient" of"(f,a,h) := (@(f,a+h) - @(f,a))/h
   is_differentiable_at(f, a) := exists Lim [h : Real] ((@(f,a+h) - @(f,a))/h, Approaching,0)
Not (is_differentiable_at (Real ,Lambda [x : Real] (Sqrt Abs (x)),0 ))
```

figure 3.2.2a

Every declaration is represented by an arrow flag, while every assumption is represented by a squared flag. The program checks if the context of a line is also used by the preceding line. If the declarations or assumptions are the same, the flag is extended, else a new flag is introduced. Once a new flag is introduced all declarations and assumptions behind that new flag also introduce new flags. A flag is ended if the context is shorter than the context of the preceding line as can be seen in the example in figure 3.2.2a.

Checking if a context of different lines is the same is done by comparing the LINGUISTIC OBJECTS of both lines. An added advantage of comparing the LINGUISTIC OBJECTS is that parts of the context such as 'x : Nat, y : Nat' and 'x; y : Nat' can be judged to be equal.

Another way for comparing the context of different lines would be to enter references between contexts of different lines. This also reduces the checks needed for a line, because the referenced context is already checked and indicated as correct. However, all references to that line become invalid if the line that is referred to is deleted. Therefore the context is repeated in each line.

To help the user in re-using the context, the context will automatically be put in the input field after a line is checked and indicated to be correct. The user can apply the same context, which leads to an extension of the flags.

3.2.3 Minor changes

Further comments from the user group that were implemented are discussed in this section.

Improve help and error messages

- The help of the previous program was too small and incomplete, which was also the case with the error messages. Both help and error messages need to be rewritten.

Help and error messages have been changed and a link has been made between the error messages and the help, so the user can search for an explanation of the error messages. Language definition, syntax rules and derivation rules described in the thesis are used for the help of the program. The help interface is discussed in section 3.3 help.

Static preface, dynamic preface and defined constants in windows
- The static preface, dynamic preface and defined constants views should be resizable, without hiding one of those views. The dynamic preface tends to grow, while the static preface always has the same size.

The static preface, dynamic preface and defined constants views that were part of the main window are replaced by separate windows, as can be seen in figure 3.3a. In this way the user can resize those windows and put them on different spots, just to the liking of the user.

Print option
- Printing the WTT book, for distributing a WTT text to other people.

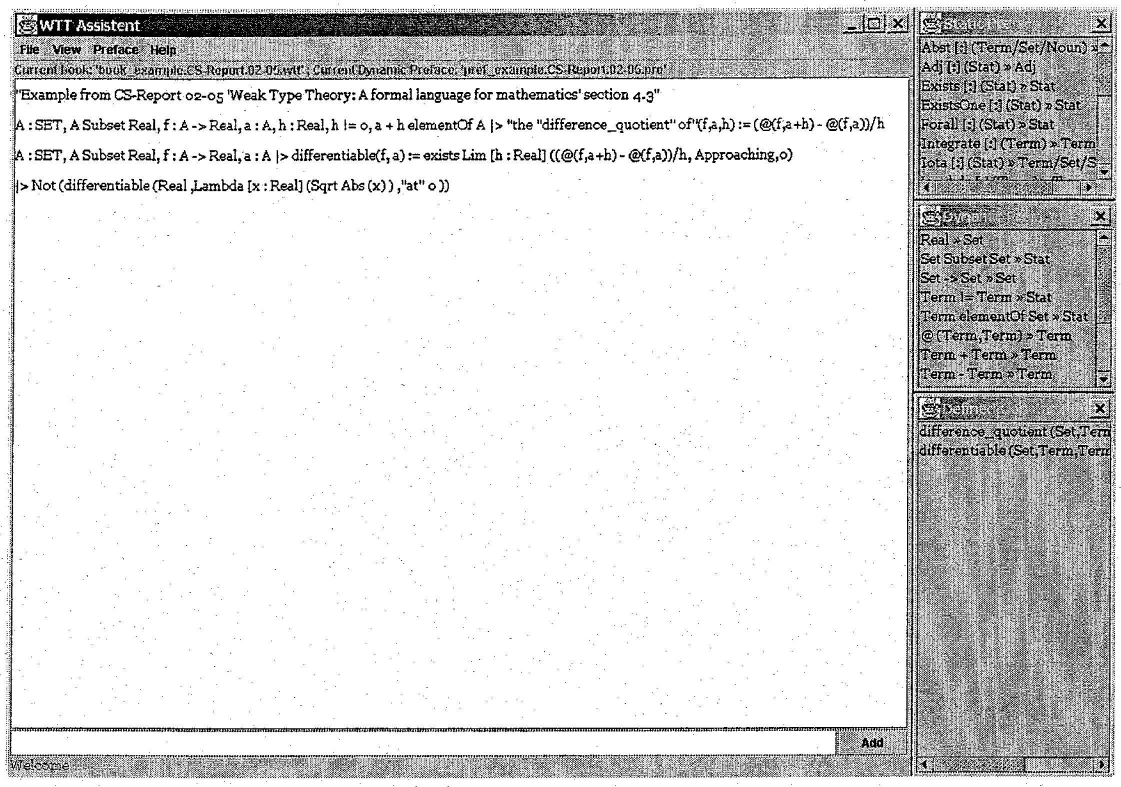
Printing the output of a program is a very common wish, there are standard classes on the internet that can be used to print a certain field of a program. One of these classes was used to enable the user to print a WTT book in the desired representation.
The 'print' option is part of the 'file' menu, depicted in figure 3.3b.

Notes
As we look at the changes described in section 3.1 and 3.2, we note that the only parts of the core program that have been changed intensively are:
- The parser part and the part that constructs the linguistic object tree.
- The user interface and the part of the program that stores the prefaces and the book.

The parts that represent the input of the program, LINGUISTIC OBJECTS, are only changed to give a better representation of the input. The Ltype checker was only extended to construct an ALI OBJECT, instead of an ELI OBJECT, if a constant was defined without parentheses.
Therefore the part of the core program that checks the input has not been significantly changed.

3.3 The final program

The new user interface is depicted in figure 3.3a.



```
Current book: 'book_example.CS-Report.02-05.wtt' ; Current Dynamic Preface: 'pref_example.CS-Report.02-06.pre'
"Example from CS-Report 02-05 'Weak Type Theory: A formal language for mathematics' section 4.3"
A : SET, A Subset Real, f : A -> Real, a : A, h : Real, h != 0, a + h elementOf A |> "the "difference_quotient" of"(f,a,h) := (@(f,a+h) - @(f,a))/h
A : SET, A Subset Real, f : A -> Real, a : A |> differentiable(f, a) := exists Lim [h : Real] ((@(f,a+h) - @(f,a))/h, Approaching,0)
|> Not (differentiable (Real ,Lambda [x : Real] (Sqrt Abs (x) ) ,"at" 0 ))
```

```
Abst [:] (Term/Set/Noun) »
Adj [:] (Stat) » Adj
Exists [:] (Stat) » Stat
ExistsOne [:] (Stat) » Stat
Forall [:] (Stat) » Stat
Integrate [:] (Term) » Term
Iota [:] (Stat) » Term/Set/S
```

```
Real » Set
Set Subset Set » Stat
Set -> Set » Set
Term != Term » Stat
Term elementOf Set » Stat
@ (Term,Term) » Term
Term + Term » Term
Term - Term » Term
```

```
difference_quotient (Set,Term
differentiable (Set,Term,Term
```

figure 3.3a

Instead of one window there are four windows and below the menu bar, a field is added where the name of the book and the name of the preface are reported.

Preface and book

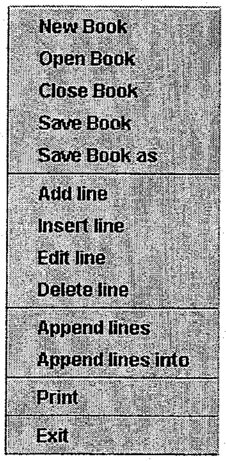


figure 3.3b

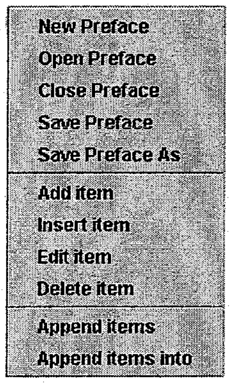


figure 3.3c

The menu is extended for all options of the book and its lines, as can be seen in the 'File' menu (see figure 3.3b) and all options of the preface and its items, as can be seen in the 'Preface' menu (see figure 3.3c).

The only difference is the 'Print' option and the 'Exit' option of the 'File' menu.
These options are used to respectively print the current book and exit the program.

Further, both book and preface have the same functionality.

Options for the line

The options for lines are visualized by the button to the right of the input field. The name of this button changes to the options that was chosen from the 'File' menu. Standard the user can add lines to the book, then the input field and the button look like figure 3.3d

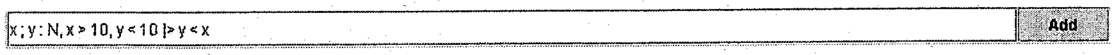

x;y:N,x>10,y<10 |> y<x

figure 3.3d

The user can click the 'insert line' option in the 'file' menu. This will change the button to 'insert' as depicted in figure 3.3e.

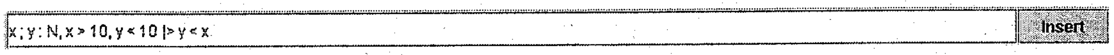

x;y:N,x>10,y<10 |> y<x

figure 3.3e

Entering an input and clicking the 'insert' button will insert the line in the book, above the selected line in the book. The user can not click the 'insert line' option if no line is selected. First typing the input line and then clicking the 'inset line' option will immediately insert the typed input above the selected line.

The same functionality is also implemented for the 'edit line' option of the 'file menu, depicted in figure 3.3f.

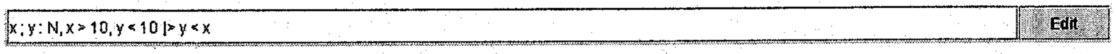
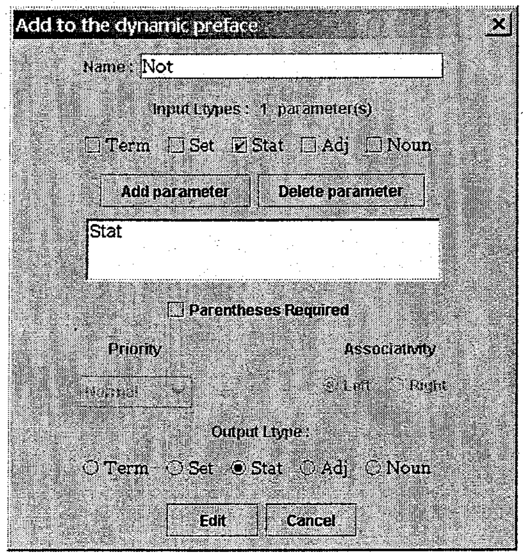

x;y:N,x>10,y<10 |> y<x

figure 3.3f

Deleting a line is done by selecting the line that must be deleted.
The input field is not needed for deleting a line.

Options for the item

The window that is constructed to define an item for the preface (see figure 2.9f) is used for visualizing the options of the items. This window changed due to introduction of the infix constants and the skipping of the parentheses to the language (see sections 3.1.1 and 3.1.3). The new window for adding items to the preface is depicted in figure 3.3g.


Add to the dynamic preface

Name: Not

Input Ltypes : 1 parameter(s)

☐ Term ☐ Set ☑ Stat ☐ Adj ☐ Noun

Stat

☐ Parentheses Required

Priority: Normal

Associativity: [✓ Left] Right

Output Ltype : Term Set [✓ Stat] Adj Noun

figure 3.3g

As long as the defined constant has two or less parameters the 'Parentheses Required' checkbox can be unchecked.

When there are precisely two parameters, the options to choose associativity and priority are activated.

In this window the 'Edit' button also changes to 'Insert' or 'Add' depending on the option chosen for the item of the preface, just like the button to the right of the input field changes for the different options of the lines.

This window is not needed for deleting an item from the preface.

Priority and associativity

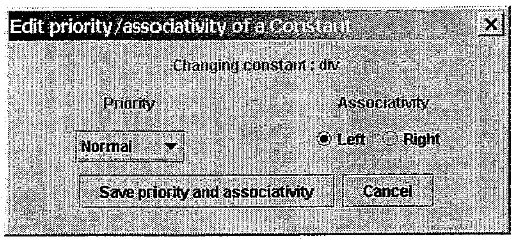

Edit priority /associativity of a Constant
Changing constant : div
Priority
Associativity
Normal
Left
Right
Save priority and associativity
Cancel

figure 3.3h

A new window, shown in figure 3,3h is constructed for changing the priority and associativity of a infix constant defined in the book. This window shows a part of the item window of figure 3.3g, but for infix constants defined in the book only priority and associativity can be changed.

Open, save book or preface

To open or save a book (or preface) the standard windows for opening and saving a file are used, depicted in figure 3.3i. Those windows enable the user to browse all files and type their names. The only difference between these windows is the title and the button to accept the file that is chosen.

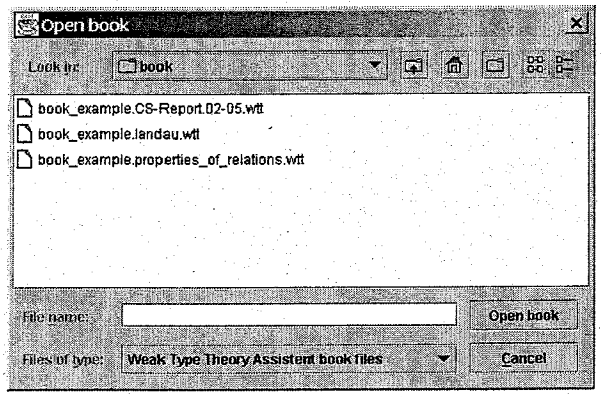

Open book
Look in:
book
book_example.CS-Report.02-05.wtt
book_example.landau.wtt
book_example.properties_of_relations.wtt
File name:
Open book
Files of type:
Weak Type Theory Assistent book files
Cancel

figure 3.3i

The top drop down box can be used to change to another drive.

There are respectively buttons for:
- Going up one directory
- Going to the home directory
- Creating a new directory
- Viewing the files with details on/off

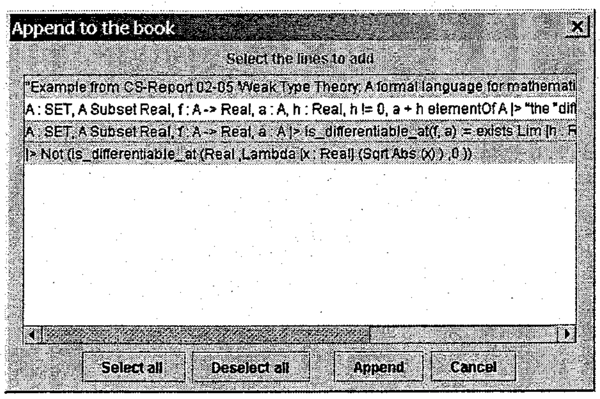
There is a list field to select a file, the name of the selected file will appear in the file name field.

The last drop down box is for the file filter. The filter determines which files are shown.

The top button, in this case the 'open book' button, is for accepting the selected file.
The 'Cancel' button returns to the program, without opening or saving a file.

Append, insert lines or items

The user can append, insert lines from other books or append, insert items from other prefaces. For example appending lines from another book. After selecting the book from which the user wants to append lines, the window depicted in figure 3.3j appears.



```
"Example from CS-Report 02-05 'Weak Type Theory: A formal language for mathemati
A : SET, A Subset Real, f : A -> Real, a : A, h : Real, h != 0, a + h elementOf A |> "the "dif
A : SET, A Subset Real, f : A -> Real, a : A |> is_differentiable_at(f, a) = exists Lim [h : R
|> Not (is_differentiable_at (Real ,Lambda [x : Real] (Sqrt Abs (x) ) ,0 ))
```

figure 3.3j

All lines are shown in the window.

The user can select or deselect all lines, or only select specific lines.

By pressing the 'Append' button the lines are added to the book.

Pressing 'Cancel' closes this window without adding the selected lines.

The same window is used for inserting lines, but then the selected lines are inserted.
A window with a different title is used to add or insert items into the preface.

Representations

Our program has three different options to represent the book, as indicated in the 'View' menu, depicted in figure 3.3k.

With the first three options the user can choose to hide or show the static preface, dynamic preface or defined constants windows.

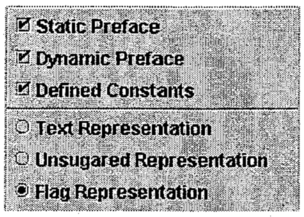


figure 3.3k

The next three options are for changing the representation of the book.

1) Text representation; shows how the user entered the line
2) Unsugared representation; shows the book in the unsugared language definitions of table 2.2a.
3) Flag representation; shows the book in the flag notation described in section 3.2.2 flag representation.

In prefix representation the program only expects input written in accordance with the unsugared language definition. The only sugaring allowed is the use of comments, because a comment is skipped by the parser and does not influences the other language definitions.

Help

Which leaves the 'help' menu, depicted in figure 3.3l:

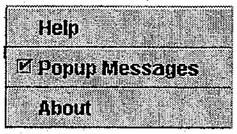
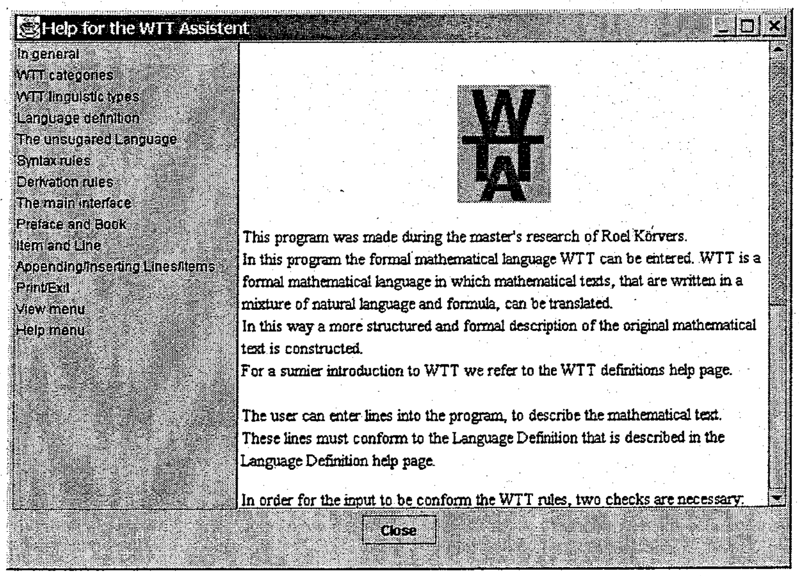


figure 3.3l

The 'Help' option loads the help pages depicted in figure 3.3m, while the 'About' option constructs a small window containing some information about the program.

Ltypes of correct lines are reported in the message field at the bottom of the main window (see figure 3.3a). If the 'Popup Messages' check box is activated, these messages are also reported in popup messages. The messages appear only in the message field if the check box is deactivated.



This program was made during the master's research of Roel Kőrvers.
In this program the formal mathematical language WTT can be entered. WTT is a formal mathematical language in which mathematical texts, that are written in a mixture of natural language and formula, can be translated.
In this way a more structured and formal description of the original mathematical text is constructed.
For a sumier introduction to WTT we refer to the WTT definitions help page.

The user can enter lines into the program, to describe the mathematical text.
These lines must conform to the Language Definition that is described in the Language Definition help page.

In order for the input to be conform the WTT rules, two checks are necessary:

figure 3.3m

Hereby the overview of the final program ends. In the following chapter some additional changes proposed by the user group are discussed. These changes are not implemented in the final program and are defined, in this report, as future user wishes.

4 User wishes

In this chapter we describe further wishes of the user group. These wishes are not implemented in the program, but a short description will be given how these wishes could be implemented. Every wish will be ranked by three numbers, these numbers represent the priority, importance and difficulty of the wish, for ranking numbers 1 to 3 are used. The ranking is made on personal assessments of the different wishes.
Number 1 means low priority, low importance or low difficulty, while number 3 means high priority, high importance or high difficulty. Number 2 is average priority, importance or difficulty.
A wish can also have priority 0 if it is a wish that is not implemented because it violates the rules of the WTT language, the design decisions made for the program or is outside the scope of this research.

Using special symbols
Instead of using 'Forall', 'Exists' use '∀', '∃'. Implementing this wish would mean a better understanding for the reader of the WTT text, because '∀', '∃' are accepted symbols. Therefore it has a high importance, but infix constants and flag view had higher priority. Lack of time prevented the implementation of this wish.

In our program these symbols could be used by adding some buttons for these special symbols. The problem however, is saving these symbols. The book and preface are saved as text files, the problem is that the special symbols cannot be saved in a text file, they are not recognized as plain text.
There are two options to fix this problem.
1) use another storage technique, so that the special symbols can be saved.
2) use another symbol that can be printed in a text file and change that symbol in the program itself when the file is opened and saved.

Priority : 2 *Importance : 3* *Difficulty : 1*

Implementing function application
In WTT the user can declare a variable to be a function that uses some parameters and returns an element. For example: 'f: Nat x Nat -> Nat', which declares a function f that takes two natural numbers and returns one natural number. Variables however can not have parameters, therefore in this stage the user must define a constant named 'func_app' (function application), in the preface.
The function application in this example takes three arguments, the function and its two parameters. Calculating the 'f' from '2' and '3' would be depicted by 'func_app(f,2,3)'.
In the sugared language this example could depicted by : 'f(2, 3)'.

For adding the possibility that function application may be written similarly to the *constant* language definition, introduces additional information that needs to be saved for variables. Variables can be declared with parameters, which can be implemented by changing the ELI OBJECT. Another parser should be written to correctly translate function applications.
Rules for checking function applications should be taken into account and the possibility to skip the parentheses by variables should also be implemented.

Priority : 2 *Importance : 3* *Difficulty : 3*

Automatic completion/automatic correction
Extend the assistance function of the program to automatically suggest constants or binder names when the user starts typing a name or correct names if they almost match a constant or binder.

To implement the automatic completion all constant and binders need to be alphabetically ordered. A popup window should appear when the user starts typing and the list of constants and binders should be scrolled to the constants or binders that start with the same characters as the user typed.

Sorting all constants and binders is difficult, although standard algorithms for sorting exist. Automatic correction needs even more difficult algorithms and rules to decide when to replace a name.
Implementing the self scrolling popup windows was also considered to be a difficult task.

Implementation of this wish would be a great expansion of the assistance function of the program, but implementing it was considered to be too difficult.

Priority : 2 *Importance : 3* *Difficulty : 3*

Reporting Ltype
The user selects a part of the text and the program gives the Ltype of the text in a popup window.

The problem is that the user can only select a part of the text in the input field. In this implementation only entire lines can be selected. Another way of representing the book could be the solution, but in this implementation the lines are put in a list and only entire items of that list can be selected. Changing the representation of the book requires a major change in the program. For example storing the lines in a text field and with mouse positions determine which part of the line is selected. Implementing this requires a way to separate the input in different parts, from which the Ltype can be calculated, and implementing mouse listeners to determine which part is selected. For information we refer to M. Franssen who used such an approach when developing 'Cocktail: A Tool for Deriving Correct Programs' [Fran 2000].

In the present stage, the user can lookup the Ltypes of the constant, binders or parameters of the constant and binders by looking them up in the static preface, dynamic preface or defined constant window.

Priority : 2 *Importance : 2* *Difficulty : 3*

Syntax highlighting
This comes down to the same wish as reporting the Ltypes of parts of the text, but in this case every Ltype is represented by a colour and every part of the text is coloured in its Ltype.
An extra feature of highlighting would be that when the line is written down the program already colours an incorrect part in a different colour

This is also not possible, because the entire line can only have one colour, this also holds for the input field. By using a different representation of the lines, as discussed in the 'Reporting Ltype' wish, this could be possible. Every part could then be modelled to have its own color.

Priority : 2 *Importance : 2* *Difficulty : 3*

Automatically take parameters when defining a constant
The changes in the definitions of constants led to a discussion with some of the persons in the user group about the probability of automatically take the variables declared in the context as the parameters of a constant to be defined. It was even suggested that some parameters could be left out so that automatically infix constants could be defined.

A parameter, however, may not be forgotten, it is a demand of the WTT language. However, this could be implemented as a sugaring of the language and would lead to the development of a different program that can calculate the value of the parameter that was omitted.
In AUTOMATH a similar request, for skipping redundant parameters, led to the development of AUT-SYNT [Ned 1994] (see also chapter 5).
Such an approach could also be used by our program.

As a compromise we already implemented another way to complete constants that are defined in the book. The user can skip typing the first parameters. The program will automatically fill in the first variables of the context till all variables of the context are represented as parameters of the constant. This is only possible if the constant is defined with parentheses. Definitions of constants without parameters, unary constants and infix constants must be done by writing all variables in the context as parameters of that constant.

Priority : 1 *Importance* : 3 *Difficulty* : 3

Converting to another format, for example Latex
Converting the WTT books to another format, so that it can be imported into another program, preferably Latex. In this way the WTT format can be more broadly used and distributed.

Converting the lines to Latex could be done more easily, but converting a tree view would be more difficult. There are special Latex libraries for depicting flags but it still would require more work.

Our inexperience with Latex and the lower priority of this wish, prevented it from being implemented.

Priority : 1 *Importance* : 3 *Difficulty* : 2

Search/replace function
A user may want to search for a constant in the preface or a formula in the book. The user can also decide to rename a constant, because its name was badly chosen. Changing the name in the preface only results in an error because the constant is used in the book. Therefore a 'replace' and 'replace all' would be necessary.

The replace options are the next logical step if a search function is implemented.

The implementation would be a simple search through all lines and/or items. There are some problems with names of constants that are also a sub-part of another constants name. Searching and especially replacing should be done with care, or else the book could become invalid.

Priority : 1 *Importance* : 3 *Difficulty* : 2

Postfix
Unary constants can be written as prefix constants, but also as postfix constants.

An option would be to add an extra field to the ALI OBJECT that stores information for a unary constant to which side of the parameter it must stand. It would also be possible to use the associativity field for this, because a unary constant does not have an associativity.

The difficulty would be with defining unary constants, because the program that constructs the LINGUISTIC OBJECTS has to decide which LINGUISTIC OBJECT is the unary constant and which LINGUISTIC OBJECT is the parameter. The name of the *variables* declared in the *context* are already stored to auto complete a parameter list of a constant that is been defined. This makes deciding which LINGUISTIC OBJECT is the parameter much easier.

Priority : 1 *Importance : 2* *Difficulty : 1*

Context Abbreviation
Many of the variables declared in the context and the assumptions on these variables are used for a number of lines, hence the use of a flag view. These contexts can grow very large and having them displayed in the text representation view is not that readable.

In the CS-Report 02-05 [Ned 2002] context abbreviation is suggested. The context is only written once and a marker is assigned to represent that context. By using the marker the entire context is used.

A possibility would be to keep a list of markers for each book. This marker then can be used in the context to link to a bigger context.

Priority : 1 *Importance : 2* *Difficulty : 2*

Allow words to be written inside constants
Constants representing adjectives or nouns need, for readability, to be differently written than the way they were defined in. For example 'an odd natural number' instead of 'odd number'.
Allowing words like 'a' and 'an' to be used in constants increases the readability.

A solution would be to let the parser skip words like 'a' and 'an', but these words could be part of other constant names. The parser would need to recognize when a certain word can be skipped.
In the final program we allow comments to be written in *words*. This is no real solution, because the ' " ' symbol must also be written in the constant name.
For example ' "an "odd" natural "number '.

Priority : 1 *Importance : 2* *Difficulty : 2*

Use multiple books in one program
In this way multiple book can be opened and the user can switch between opened books.

For now the user can start up multiple programs were each program controls its own book. The problem is that then two programs can open the same book and a change of one program is overwritten by the other program.
Opening multiple books adds to the information one program must remember. For every book a dynamic preface and a defined constant list must be stored. The static preface is the same for every book. More importantly the list of lines entered in the book must be stored.

All information stored for a book in the final program, is done by one class. This class is separated from the interface class. Opening multiple books would mean to create this class for every book that is opened. These classes could be stored in a list and a number could record which book is active.
The information class from book i, is stored on position i of the list. Activating another book will change the i and therefore automatically change the information class that is used to interact with the interface.

Priority : 1 *Importance* : 2 *Difficulty* : 2

Adding items to the preface by using the input field
An experienced user would like to add items to the preface by using the input field, instead of using the extra window constructed for this purpose. In the same way items can be edited or inserted.

An item could be given in the same way it is represented in the static preface, dynamic preface and defined constants window. An extra set of numbers could determine the priority and associativity of the infix constants. A special symbol would determine that the text in the input field is for the preface instead of for the book.

It would require another parser to parse the text for the preface.

Priority : 1 *Importance* : 2 *Difficulty* : 2

Folding/unfolding lines/flags and remembering
A mathematical description consists of different axioms, theorems etc.. When such sub-part of the description is translated in WTT a next sub-part is translated. Not every sub-part needs to be visible at all times, hiding or folding these lines or flags adds to the clarity of the view. Remembering which lines or flags were folded enables the user to continue where he left off the previous time.

Folding the flags is already possible, in the way the flag view is constructed. Again the lines can not be folded due to the way in which the line view is constructed, as discussed in the 'reporting Ltype' wish discussed in this section.

Priority : 1 *Importance* : 2 *Difficulty* : 3

Redefining constant names
In many mathematical texts the name of a constant is re-used for every axiom or theorem. First f is defined as a function used in the first axiom while f in the second axiom is defined as a completely different function.

Re-using constant names is against the rules of WTT. Defined constants should always have a fresh name. One could permit re-use but it has many complications. Lines may not be inserted or deleted as easily, because the same line inserted on another place can have a different meaning.
I. Zandleven discussed a paragraph system to avoid these problems in a paper, 'A verifying program for Automath'. This paper can be found in the book 'Selected papers on Automath' [Ned 1994].

This paragraph system could be used to add the redefinition of constants as a sugar option to the WTTA program. These redefined constants must have different names when checked by the core program.

Priority : 1 *Importance* : 2 *Difficulty* : 3

Numbering the lines
Number the lines in a book, for easy reference.

Adding line numbers for the 'Text representation' and 'Prefix representation' can be done easily by adding a counter that counts the lines in the book.
The difficulty is adding line numbers to the 'Flag representation'. Every flag and item entered must be counted and the number must be printed in front of the flag, which moves the flag more to the right.

It could be decided that some lines should not be numbered.
- should lines filled with comments be numbered.
- should every context in the tree view be numbered, or only the definitions and assertions.
This would require a more difficult method for printing the line numbers.

Priority : 1 *Importance* : 1 *Difficulty* : 1

Temporarily turn off checking
The idea is that a user translated an entire text and enters it completely, after adding the text is checked.

This goes again the design decision that all lines in the book must be correct. The user still can add the translated text line by line.

An extra window could be build, where the user can type multiple lines behind each other. These lines could then automatically, line by line, be added to the book. An error in one line would stop this process and lets the user correct the lines not yet added to the book.

Priority : 1 *Importance* :1 *Difficulty* : 1

Add statistics
From a research point of view it could be interesting to ask information about a WTT text. How many lines are there, which percent of these lines is a statement and which percent is a definition. How large is the context etc..
To find out what statistics are interesting, how to explain the outcome of the statistics and how these statistics can be represented could be a subject for future work.

Priority : 0 *Importance : 3* *Difficulty : 3*

Turn off/add sugaring options
For compatibility with the core program, lines written without sugaring can still be added to the program. A book can even be viewed in an unsugared representation.

Adding sugaring options requires a rewrite of the parser that parses the input to the linguistic object tree. Defining rules to describe these sugaring options are very difficult, because sugaring can have all kind of different forms. The easier way would be to write a parser for parsing requested sugaring and constructing the corresponding linguistic object tree.

Priority : 0 *Importance : 1* *Difficulty : 3*

Strengthen derivation rules
Add or change derivation rules to strengthen them for later versions of the WTT language.

Changing the derivation rules means changing the core of the program. The checking of the linguistic object tree should be rewritten, maybe the entire linguistic objects need to be changed. This can not be done by simply changing some rules, but needs a rewrite of a part of the program!

Priority : 0 *Importance : 1* *Difficulty : 1*

By the sheer number of wishes it can be concluded that the program is not yet ready for final use, but that is also the reason why the program is still a prototype and not yet a full version.

In the next chapter we will describe some of the attention points that should be taken into consideration by a reader who also wants to design an assistant tool for a formal mathematical language.

5 Attention points

In this chapter we will give some general attention points for developing an assistant tool for a formal mathematical language, based on the experiences during this research. The points mentioned may be of importance for future research or for others building a similar assistant tool.

Sugaring and the "de Bruijn criterion"
Most mathematical texts are full of sugared text. Readers do not notice this sugared text, because it is widely accepted. Nobody writes +(3(),4()) instead of 3 + 4, but the first form states much clearer that 3() and 4() are parameters of the + constant.

Using a program to check a mathematical language on correctness asks for a high reliability of the program. This reliability is much easier to establish by first designing a program for unsugared text. The reason is that every small addition to the language adds exceptions to the way the input of the program must be checked. These exceptions add to the code used for checking the language and as stated by the "de Bruijn criterion"
[Bar 2001], a verifying program must be a very small program, giving the highest possible reliability.

Use objects
When the input is a line that has to be checked, it is a good option to represent that line by an object structure. Objects are easier to check, because each object, if modelled correctly, has fields for every information part that has to be checked before the entire object can be considered correct. Checking these fields is easier than first searching the input line for the place where specific information is written.
Of course the input line still needs to be searched, but in this way parsing the line and checking the line is separated. This separation of concerns also allows to verify the correctness of the rules separate from the parsing of the input.

How to parse an input line can be found in 'Compilers: principles, techniques and tools' [Aho 1986] or in the 'Compilers' syllabus [Eik 1997].
Information about object modelling and data structure can be found in 'An introduction to object-oriented programming' [Bud 1997] .

Extending the language
Extending the language with sugar is more complex than we expected. Adding infix constants to the language automatically raises questions about priority, associativity and parentheses. Extending the language with multiple declarations, automatically allows binders to have multiple declarations.

Be aware of the consequences for the entire language, before adding sugaring to the language. Look carefully at the places where the unsugared language definition (table 2.2a) is changed and change the parser and linguistic object tree builder accordingly. An error message needs to be generated if the linguistic object tree cannot be built correctly.

To stress the complexity of extending the language, we look at both programs that construct the linguistic object tree, the program for the unsugared language in appendix B (21 KB) and the program for the sugared language in appendix F (71 KB), we notice that the program for the sugared language is more than 3 times bigger than the program for the unsugared language.

Do not change the checks
When adding user wishes, do not change the program part that checks the structure.
The 'de Bruijn criterion' [Bar 2001] implies that all additions to the program should be added as layers above the core program. This keeps the verifying part of the program small and concise.

For example, by adding the option that a user does not have to type all parameters when defining a constant, the program still needs to add all parameters. Leaving out a parameter, because we want 'Max' as an infix constant, has enormous consequences, as we can see in the following example:
'V : SET , ordered(V), x;y : V $\rhd$ x Max y := Iota$_{n:V}$ (((n = x) \/ (n = y)) /\ n $\geq$ x /\ n $\geq$ y)'

This definition of the 'Max' constant is not correct. The 'Max' constant as defined above is only defined for elements of the ordered set 'V', not for other sets. So 'Max' should have three parameters, 'V', 'x' and 'y', instead of only 'x' and 'y'.
A second problem is that the symbol '≥' should match with the ordered character of the set 'V', and in the given example it is unclear how that must be done.
This kind of problems led to the proposal of an AUT-SYNT extension of AUTOMATH. The AUT-SYNT extension makes it possible to suppress redundant parameters, by calculating if the type of parameters correspond with the type under which the constant was declared. For more information about AUT-SYNT we refer to the article 'Checking Landau's "Grundlagen" in the AUTOMATH System' written by L.S. van Benthem Jutting. This article can be found in the book 'Selected papers on AUTOMATH' [Ned 1994].

Representation
Choosing the way in which the WTT description will be represented must be considered carefully. The choice that the lines are represented as items in a list automatically restricts the way in which the lines can be selected and shown on the screen.
A more flexible representation could have helped on the long run, but would have added extra complexity for a simple representation in the beginning.

Representing the lines in another way would have cost a lot of time. Due to the time restrictions of this research, it was advised against by a member of the test user group.

How difficult representing the language is, became apparent when the flag view representation was implemented. Implementing the flag view was possible by using code from the Coq viewer program [Oost 1999], although it still required investigating how the program functioned, to determine the relevant classes and rewriting those classes. Adding those classes to the program was much easier, because of the object oriented way the program was written. The class that saved all information about the WTT books was separated from the interface, which enabled easy access to the functions needed for representing the different parts of the lines in the book. This again stresses the advantages of the use of objects.

Re-using other code
For implementing the print option, already existing code was used. For such common features re-using code is acceptable, especially if that feature can, independently, be integrated in the program. This is another advantage of using an object orientated programming method.

6 Remarks

This section contains some personal remarks of the author about the project in total and some detailed remarks about WTT and the programming language.

Programming language
In the entire document we only discussed some implementation possibilities, but we never discussed which programming language was used.
The programming language used is not that important and largely depends on the preferred programming language of the programmer.
Some programming languages however, are more suitable for the object oriented design used in this research.

For developing the prototype program discussed in this thesis the Java^{tm} programming language was used and the interface was built by using the standard GUI components of the java swing^{tm} set.
In spite of the extensive on-line help pages [Java], not all information about the standard classes of the programming language where that clear.
This led to some delay while constructing the flag view and also to some delay for re-using the existing code for the print option. The author estimated that implementing a print option, with no prior knowledge of the subject, would take too much time.

The final program can be downloaded from
http://www.betreft.nl/wtta/index.html [WTTA 2002]

The WTT language
The WTT language, used in this project as the formal mathematical language, still has some unclear points in its design. The derivation rules of the WTT language are proved to be correct, but some language definitions are ambiguous and unclear.

As example we use the ':' identifier, which represents the declaration symbol.
The difference between the ':' symbol and the '∈' symbol, was reason for much discussion.
In almost all sugared descriptions the '∈' symbol is used for declaring variables, while there is a big difference.
E.g. '$n \in Nat \Rightarrow n \in Real$' is true, but n can only have one type '$n : Nat$' or '$n : Real$', not both!

Furthermore, in WTT the right hand side of the declaration can be an expression with Ltype Noun. For example, 'x : a natural number'.
'x ∈ a natural number' however is impossible because, taking one element from a natural number is impossible.

Binders of the WTT language

The WTT language has a predefined set of binders, which are declared in the preface.
A complete list of all binders however cannot be found. The static preface can only be extended by hand and it is recommended that this is not done without prior knowledge about the modelling of the ELI OBJECTS. Ltypes and categories are modelled by numbers, because these are easier and quicker to compare. A binder with a wrong Ltype is not useful and an ELI OBJECT cannot be constructed with numbers that do not represent a correct category or Ltype.

Added to this is that some binders cannot be modelled by standard methods.
Two binders in particularly introduced exceptions to the design of the program.

Iota binder

The Iota binder can have three different Ltypes. Which Ltype is the correct one depends on the bounded variable of the Iota binder.

- If the bound variable is declared as an element of a set type, the Iota binder has Ltype Term.
- If the bound variable is declared as an element from the class of all sets, the Iota binder has Ltype Set.
- If the bound variable is declared as an element from the class of all statements, the Iota binder has Ltype Stat.

Abst binder

The Abst binder has one parameter, but this parameter may have three different Ltypes.
The Abst binder has Ltype Noun and is used to abstract the parameter of Ltype Term, Set or Noun to an expression of Ltype Noun.

The Abst binder is also the reason why multiple Ltypes can be selected for parameters of constants defined in the preface.
This enables the user to for example, define the '=' symbol with two parameters of multiple Ltypes. This '=' can then be used to express that two Terms are equal, but also that two Sets are equal. In this scenario however, it is also possible that the first parameter has Ltype Term and the second parameter has Ltype Set. This expression is correct following the WTT language, but to compare an expression with Ltype Term with an expression with Ltype Set, is not correct.
The user should be aware that if parameters with multiple Ltypes are used, all possible combinations of these Ltypes are found to be correct.
In the future the user probably wants a method to declare some dependency between the different parameters with multiple Ltypes. The final program generates a message box, warning the user about the use of parameters with multiple Ltypes.

As final remark the author regrets that still so many user wishes are not implemented, mainly due to the lack of time to implement those wishes.

7 Conclusions

The implementation of the prototype assistant program for the formal mathematical language WTT is, partly, succeeded. A small core program was constructed that verifies if a input text entered is conform to the rules for the WTT language described in CS-Report 02-05 [Ned 2002].

Language
The language that could be entered into the core program was as small as possible and did not allow any ambiguity in its constructions. This allowed the language to be checked by a small verify program. This is in accordance with the "de Bruijn criterion", that states that a verifying program should be a very small program; then this program can be checked by hand, giving the highest possible reliability of the verifying program [Bar 2001].

The program verifying that the input of the program, was conform to the WTT rules, consisted of two very small programs. One program verified that the input was conform to the syntax rules (13 KB), the other program verified that the input was conform to the linguistic type rules (21 KB).
Both programs were very small and could directly be linked to the rules they had to check. This strengthens the belief in the correctness of both programs, although they have not been formally verified.

Separating the WTT rules in syntax checks and linguistic type checks resulted in detailed and more precise error messages.

The drawback of using a small and unambiguous language was that reading and writing such a language was very hard. Therefore another program needed to be developed, this program however, would need to accept a more user friendly input. The program was developed as a layer on top of the verifying programs, so that the verifying programs did not have to be radically changed, and stayed as small as possible.

Although the use of a small and unambiguous language as input for the core program heavily restricts the input, it also gives a greater confidence of the correctness of the verifying parts of the program.

Objects
The input of the core program was translated in an object structure that represented the input. Instead of checking the input, the object structure was checked.
The input was parsed and by constructing an object structure, instead of checking the input, a clear separation between interpreting the input and checking the input was made.
- First the input was parsed and from this input the object structure was constructed.
- Then the constructed object structure was checked.

When the final program was developed only the parsing and object structure constructing phase had to be changed, while the checking of the object structure could be kept the same.

The use of objects ensured that the verifying parts of the program were not radically changed when the program was extended in order to accept a more user friendly input.

User friendly language
The use of objects simplified the development of a layer on top of the core program. This layer accepted a more user friendly input, but the development of the program that translated the input to the linguistic object structure that was used by the core program was still difficult.

A small extension of the language that enlarged the user friendliness, added a lot of exceptions to the way that the input had to be translated into an object structure.
For example: adding an option to accept multiple declarations added the use of multiple declarations in binders as well.
The program for constructing the linguistic object tree for the sugared language is more than three times as big as the program that constructs the linguistic object tree for the unsugared language, 71 KB versus 21 KB.

This however strengthened the believe that *parsing the input and constructing the object structure should be separated from checking the object structure.*

Representation
The input entered into our program was represented by a list of lines. In a later stage this representation was insufficient, because parts of a line needed to be selected, this was not possible in the chosen representation.
Another way of representing the lines would be to depict the different parts of a line behind each other, then every part of a line could be selected, while all parts together still were represented as a line. To implement such representation, however, would have taken too long. This time was used to add options of higher priority to the program.

The representation chosen restricts the options for manipulating the input, on the other hand the amount of time that is needed to implement a representation should also be taken into consideration.

Theoretical versus practical
The construction of a practical computer program for the theoretical WTT language, revealed a different way to interpret the WTT rules.
It led to the use of only one constant derivation rule and the distinction between declarations and statements, instead of declarations as typing statements.

Apart from finding a small omission in the document, the discussions about the WTT language also revealed unclearness about the usage of the ':' symbol and unclearness about some binders, which led to the use of another binder derivation rule in this thesis.

Trying to implement a practical version of an informal idea can only lead to a better understanding of the theoretical description, it can even produce a different view that can be used to clarify the theoretical description.

Future work
In this thesis a number of wishes are described that can increase the usability of the final program, the name 'final' therefore only refers to the fact that the program is the final version for this research.

By the sheer number of wishes it can be concluded that the program is not yet ready for final use.

Recalling the metaphor, used to describe the position of this research, we repeat that there is still work to be done, to translate a WTT description into a TT description.

Bibliography

[Aho 1986] Compilers: principles, techniques and tools
A. V. Aho, R. Sethi and J. D. Ullman
Amsterdam: Addison-Wesley, 1986

[Bar 2001] Proof-assistants using dependent type systems
H. Barendregt, J. H. Geuvers
in: 'Handbook of automated reasoning' A. Robinson, A. Voronkov eds.
Elsevier Science Publishers B.V. 2001

[Bud 1997] An introduction to object-oriented programming
T. A. Budd
Amsterdam: Addison-Wesley, 1997 (2nd Edition)

[Eik 1997] Compilers
H. M. M. ten Eikelder, R. van Geldrop, C. Hemerik
1997 (syllabus of the Compiler course given
at Eindhoven University of Technology)

[Fran 2000] Cocktail: A Tool for Deriving Correct Programs
M. Franssen
PhD. Thesis, Eindhoven University of Technology, 2000
http://www.win.tue.nl/~michaelf/compscience/cocktail/

[Java] Java on-line help pages
http://java.sun.com/j2se/1.3/docs/api/index.html

[Ned 1994] Selected papers on AUTOMATH
R. P. Nederpelt, J. H. Geuvers, R. C. de Vrijer,
L.S. van Benthem Jutting, D.T. van Daalen eds.
North-Holland, 1994
(references were made to :
a paragraph system by I. Zandleven, page 802
AUT-SYNT by L. S. van Benthem Jutting, page 299)

[Ned 2002] CS-Report 02-05 Weak Type Theory: A formal language for mathematics
R. P. Nederpelt
Computer Science report 2002 of the department of mathematics and computer science at Eindhoven University of Technology

[Oost 1999] CoqViewer program
http://www.cs.kun.nl/~martijno/imd/index.html

[WTTA 2002] The final WTTA program
http://www.betreft.nl/wtta/index.html

```
package data.parsinput2structure ;

/**
 * Title:        IdentifierInformation
 * Description:  store information about an identifier that is parsed
 *               constructed by 'Parser'
 *               read by 'WTTAUnsugaredParser' and 'WTTASugaredParser'
 * Copyright:    Copyright (c) 2002
 * Company:      Eindhoven University of Technology
 * @author:      Roel Körvers
 * @version:     1.0
 */

public class IdentifierInformation  {
  private String identifier ;  //The identifier
  private String type;         //'name' or the predefined symbol parsed
  private int start_position ; //Start position in the input
  private int end_position ;   //End position in the input
                               //The identifer represents the part of the input
                               //between start position and end position

  public IdentifierInformation () {
  }

  public void setIdentifier (String name) {
    identifier = name;
  }

  public void setType (String typename ) {
    type = typename ;
  }

  public void setStartPosition (int start) {
    start_position  = start;
  }

  public void setEndPosition (int end) {
    end_position  = end;
  }

  public String getIdentifier () {
    return identifier ;
  }

  public String getType () {
    return type;
  }

  public int getStartPosition () {
    return start_position ;
  }

  public int getEndPosition () {
    return end_position ;
  }
}
```

```
package data.parsinput2structure ;

import data.WTTAMessages.WTTAError;

/**
 * Title:        Parser
 * Description:  keyboard input parser
 *               parses identifiers and stores them in IdentInformation objects
 *               identifiers are: predefined symbols, names, symbols, numbers
 *               skips comments and spaces
 * Copyright:    Copyright (c) 2002
 * Company:      Eindhoven University of Technology
 * @author:      Roel Körvers
 * @version:     1.0
 */

public class Parser {
  private static String current_string ;
  private static char current_char ;
  private static String current_identifier ;
  private static int current_pos ;

  public static IdentifierInformation  ParsIdentifier (String input) {
    //pre : start parsing input,
    //post: return the parsed identifier
    return ParsIdentifier (input, 0);
  }

  public static IdentifierInformation  ParsIdentifier (String input, int startpos) {
    //pre : parsing input start at the given position,
    //post: return the parsed indentifier
    IdentifierInformation  answer = new IdentifierInformation ();
    String identifier = "";
    String type = "";
    current_string  = input;
    if ( startpos  < 0 || startpos  > input.length() ) {
      throw new WTTAError ("WTTA Error: Parser tried to read outside the bounds of the input!"  );
    } else
    if (startpos == input.length()) {
      answer.setIdentifier ("");
      answer.setType ("end of line");
      answer.setEndPosition (startpos);
      return answer;
    } else {
      current_pos  = startpos;
      current_char  = input.charAt (current_pos );
      if ((int)current_char  == 32) {                             // <space>
        skip_spaces ();
        answer  = ParsIdentifier (input,current_pos );
        identifier  = answer.getIdentifier ();
        type = answer.getType ();
      } else //added for comments
      if ((int)current_char  == 34) {                             // comment
        skip_comment ();
        answer  = ParsIdentifier (input,current_pos );
        identifier  = answer.getIdentifier ();
        type = answer.getType ();
      } else //end added
    if ((int)current_char == 40) {                                    // (
      identifier = "";
      identifier += current_char ;
      type = "predefined" ;
      current_pos ++;
    } else
    if ((int)current_char == 41) {                                    // )
      identifier = "";
      identifier += current_char ;
      type = "predefined" ;
      current_pos ++;
    } else
    if ((int)current_char == 44) {                                    // ,
      identifier = "";
      identifier += current_char ;
      type = "predefined" ;
      current_pos ++;
    } else
    if ((int)current_char == 58) {                                    // : or :=
      current_pos ++;
      if ( current_pos < current_string .length () ) {
        current_char = current_string .charAt (current_pos );
        if ((int)current_char == 61) {                                // :=
            identifier = ":=";
            type = "predefined" ;
            current_pos ++;
        } else                                                        // :
          identifier = ":";
          type = "predefined" ;
      } else                                                          // :
        identifier = ":";
        type = "predefined" ;
    } else //added for multi declarations
    if ((int)current_char == 59) {                                    // ;
      identifier = "";
      identifier += current_char ;
      type = "predefined" ;
      current_pos ++;
    } else //end added
    if ((int)current_char == 80) {                                     // P
      identifier = "";
      identifier += current_char ;
      current_pos ++;
      if ( current_pos < current_string .length () ) {
        current_char = current_string .charAt (current_pos );
        identifier += current_char ;
        current_pos ++;
      }
      if ( current_pos < current_string .length () ) {
        current_char = current_string .charAt (current_pos );
        identifier += current_char ;
        current_pos ++;
      }
      if ( current_pos < current_string .length () ) {
        current_char = current_string .charAt (current_pos );
        identifier += current_char ;
        current_pos ++;
      }
```

```
        if (identifier.equals("PROP")) {                          // PROP
          type = "predefined";
        } else {                                                  // word
          current_pos = current_pos - identifier.length();
          current_char = current_string.charAt(current_pos);
          identifier = get_word();
          type = "name";
        }
      } else
      if ((int)current_char == 83) {                              // S
        identifier = "";
        identifier += current_char;
        current_pos++;
        if ( current_pos < current_string.length() ) {
          current_char = current_string.charAt(current_pos);
          identifier += current_char;
          current_pos++;
        }
        if ( current_pos < current_string.length() ) {
          current_char = current_string.charAt(current_pos);
          identifier += current_char;
          current_pos++;
        }
        if (identifier.equals("SET")) {                           // SET
          type = "predefined";
        } else {                                                  // word
            current_pos = current_pos - identifier.length();
            current_char = current_string.charAt(current_pos);
            identifier = get_word();
            type = "name";
        }
      } else
      if ((int)current_char == 91) {                              // [
        identifier = "";
        identifier += current_char;
        type = "predefined";
        current_pos++;
      } else
      if ((int)current_char == 93) {                              // ]
        identifier = "";
        identifier += current_char;
        type = "predefined";
        current_pos++;
      } else
```

```
    if ((int)current_char == 124) {                                   // |
      current_pos ++;
      if ( current_pos < current_string .length () ) {
        current_char = current_string .charAt (current_pos );
        if ((int)current_char == 62) {                                // |>
            identifier = "|>";
            type = "predefined";
            current_pos ++;
        } else {                                                       // symbol
          current_pos --;
          current_char = current_string .charAt (current_pos );
          identifier = get_symbol ();
          type = "name";
        }
      } else {                                                        // symbol
        current_pos --;
        current_char = current_string .charAt (current_pos );
        identifier = get_symbol ();
        type = "name";
      }
    } else
    if  (                                                             // word
        ((int)current_char >= 65  && (int)current_char <= 90 ) ||  //A..Z
        ((int)current_char >= 97  && (int)current_char <= 122)     //a..z
      ) {
      identifier = get_word ();
      type = "name";
    } else
    if  (                                                             // symbol
        ((int)current_char == 33  || (int)current_char == 35 ) ||  //!or#
        ((int)current_char >= 36  && (int)current_char <= 39 ) ||  //$..'
        ((int)current_char == 42  || (int)current_char == 43 ) ||  //*or+
        ((int)current_char == 45  || (int)current_char == 47 ) ||  //-or/
        ((int)current_char >= 60  && (int)current_char <= 64 ) ||  //<..@
        ((int)current_char == 92  || (int)current_char == 94 ) ||  //\or^
        ((int)current_char == 95  || (int)current_char == 96 ) ||  //_or`
        ((int)current_char == 123 || (int)current_char == 125) ||  //{or}
        ((int)current_char == 126)                                  //~
      ) {
      identifier = get_symbol ();
      type = "name";
    } else
    if  (                                                             //number
         (int)current_char >= 48 && (int)current_char <= 57          //0..9
      ) {
      identifier = get_number ();
      type = "name";
    } else {                 // unknown, special keyboard input parse per symbol
      identifier = "";
      identifier += current_char ;
      type = "name";
      current_pos ++;
    }
    answer .setStartPosition (startpos );
    answer .setIdentifier (identifier );
    answer .setType (type );
    answer .setEndPosition (current_pos );
    return answer ;
  }
}
```

```
private static void skip_spaces () {
  //pre : space parsed
  //post: spaces skipped
  current_pos ++;//skip first space
  while ( current_pos < current_string .length () ) {//stop if end of line
    current_char = current_string .charAt (current_pos );
    if ((int)current_char == 32) {//skip all spaces
      current_pos ++;
    } else {
      break;
    }
  }
}

private static void skip_comment () {
  //pre : '"' parsed (opening comment)
  //post: comment skipped
  current_pos ++;//skip open comment
  while ( current_pos < current_string .length () ) {//stop if end of line
    current_char = current_string .charAt (current_pos );
    if ((int)current_char != 34) {//pars until close comment
      current_pos ++;
    } else {
      current_pos ++;//skip the close comment
      if (current_pos < current_string .length ()) {
        current_char = current_string .charAt (current_pos );
      }
      break;
    }
  }
}

private static String get_number () {
  //pre : number parsed
  //post: entire number returned
  boolean foundDecimal = false;
  String answer = "";
  answer += current_char ;//save first parsed number
  current_pos ++;
  while ( current_pos < current_string .length () ) {//stop if end of line
    current_char = current_string .charAt (current_pos );
    if ( ((int)current_char >= 48 && (int)current_char <= 57) ) {
      answer += current_char ;//add if a number is parsed
      current_pos ++;
    } else {
      if ( ! foundDecimal && (int)current_char == 46 ) {//can be decimal
        foundDecimal = true;//decimal may only occur ones
        answer += current_char ;//add decimal point and pars numbers behind it
        current_pos ++;
      } else {
        break;
      }
    }
  }
```

```
    //add all spaces and comments to numbers, symbols or words.
    //not predefined identifiers
    while ((int)current_pos < current_string.length()) {
      if ((int)current_char == 32)
        skip_spaces();
      else
      if ((int)current_char == 34)
        skip_comment();
      else break;
    }
    return answer;//return the number
  }

  private static String get_symbol() {
    //pre : symbol parsed
    //post: entire symbol returned
    String answer = "";
    answer += current_char;//save first parsed symbol
    current_pos++;
    while ( current_pos < current_string.length() ) {//stop if end of line
      current_char = current_string.charAt(current_pos);
      if  (
          ((int)current_char == 33  || (int)current_char == 35 ) || //!or#
          ((int)current_char >= 36  && (int)current_char <= 38 ) || //$..&
          ((int)current_char == 42  || (int)current_char == 43 ) || //*or+
          ((int)current_char == 45  || (int)current_char == 47 ) || //-or/
          ((int)current_char >= 60  && (int)current_char <= 64 ) || //<..@
          ((int)current_char == 92  || (int)current_char == 94 ) || //\or^
          ((int)current_char == 95  || (int)current_char == 96 ) || //_or`
          ((int)current_char == 123 || (int)current_char == 125) || //{or}
          ((int)current_char == 126)                                //~
        ) {
        answer += current_char;//add if a symbol is parsed
        current_pos++;
      } else {
        break;
      }
    }
    //add all spaces and comments to symbols, words or numbers.
    //not predefined identifiers
    while ((int)current_pos < current_string.length()) {
      if ((int)current_char == 32)
        skip_spaces();
      else
      if ((int)current_char == 34)
        skip_comment();
      else
        break;
    }
    return answer;
  }
```

```
 private static String get_word() {
   //pre : name parsed
   //post: entire word with possible comment returned
   String answer = "";
   answer += current_char ;//save first parsed letter
   current_pos ++;
   while ( current_pos < current_string .length() ) {//stop if end of line
     current_char = current_string .charAt(current_pos );
     if  (
         ((int)current_char >= 48  && (int)current_char <= 57 ) || //0..9
         ((int)current_char >= 65  && (int)current_char <= 90 ) || //A..Z
         ((int)current_char == 95 )                             || //_
         ((int)current_char >= 97  && (int)current_char <= 122) //|| //a..z
       ) {
       answer += current_char ;//add if letter, number or '_'
       current_pos ++;
     } else //added for comment in words
     if ((int)current_char == 34 ) { //begin comment
       skip_comment ();//skip comment in word
     } else //end added
       break;
   }
   //add all spaces and comments to words, numbers or symbols.
   //not predefined identifiers
   while ((int)current_pos < current_string .length()) {
     if ((int)current_char == 32)
       skip_spaces ();
     else
     if ((int)current_char == 34)
       skip_comment ();
     else
       break;
   }
   return answer;
 }

}
```

```
package data.parsinput2structure ;

import java.util.Vector;
import data.linguisticstructure .*;
import data.WTTAInformation .*;
import data.WTTAMessages .IncorrectInput ;

/**
 * Title:          WTTAUnsugared2Structure
 * Description:    Unsugared language parser
 *                 uses the 'Parser' to receive identifiers and constructs
 *                 the linguistic object tree, by combining the identifiers
 * Copyright:      Copyright (c) 2002
 * Company:        Eindhoven University of Technology
 * @author:        Roel Körvers
 * @version:       1.0
 */

public class WTTAUnsugared2Structure {
  private String total_string ;
  private IdentifierInformation  ident_info ;

  public WTTAUnsugared2Structure () {
    total_string  = "";
    ident_info  = new IdentifierInformation ();
  }

  public LinguisticObject  build(String string2Bparsed ) throws IncorrectInput {
    //pre :
    //post: given string is parsed into a liguistic object tree
    LinguisticObject  answer;
    answer = null;
    total_string  = string2Bparsed ;
    LinguisticObject  first;
    ident_info  = Parser.ParsIdentifier (total_string );//first identifier
    if ( ident_info .getType ().equals ("name" ) ||
         ident_info .getIdentifier ().equals ("PROP" ) || ident_info .getIdentifier ().equals ("SET" )) {
      first = betweenPredefined ();//first linguistic object is no tree
      answer = expectTree (first);//after this object a tree identifier is expected
    } else
    if (ident_info .getIdentifier ().equals ("|>")) {//begins with a line identifier => empty context
      answer = expectTree (new EmptyContext ());//Line with empty context is constructed
    } else
    if (ident_info .getType ().equals ("end of line" ) ) {
      if ( string2Bparsed .trim().length () != 0) {//comment
        answer = null;//no error, but the line doesnot have to be checked.
      } else {
        throw new IncorrectInput (
          "Parse error: Input may not be empty!"  ,
          ident_info .getStartPosition (),
          ident_info .getEndPosition ()
        );
      }
    } else
      throw new IncorrectInput (
        "Parse error: First identifier may not be a Predefined Identifier, except the '|>' identifier."    ,
        ident_info .getStartPosition (),
        ident_info .getEndPosition ()
      );
    return answer;
  }
```

```
private LinguisticObject  expectTree (LinguisticObject  left) throws IncorrectInput {
  //pre : binary tree identifier expected given linguistic object is left hand side
  //post: binary tree object constructed with left and right hand side
  //      recursivly constructed of all identifiers to the right
  LinguisticObject  answer;
  answer = null;
  if (ident_info.getType().equals("end of line"))
    answer = left;//this (tree) object is the entire linguistic object tree
  else {
    String tree_name;
    int tree_start_position ;
    int tree_end_position ;
    LinguisticObject  tree;
    LinguisticObject  right;
    LinguisticObject  tmp_object;
    tree = null;
    right = null;
    if ( ident_info.getIdentifier().equals(":") || ident_info.getIdentifier().equals(":=") ||
         ident_info.getIdentifier().equals(",") || ident_info.getIdentifier().equals("|>") ) {
      //binary tree identifier
      tree_name = ident_info.getIdentifier();
      tree_start_position  = ident_info.getStartPosition();
      tree_end_position  = ident_info.getEndPosition();
      nextIdentifier();// identifier after binary tree identifier should be no tree
      if ( ident_info.getType().equals("name") ||
           ident_info.getIdentifier().equals("SET") || ident_info.getIdentifier().equals("PROP") ) {
        if (tree_name.equals(":") || tree_name.equals(":="))//declaration or definition
          right = betweenPredefined();//next linguistic object should be no tree
        else
        if (tree_name.equals(",") || tree_name.equals("|>")){//context or line
          tmp_object = betweenPredefined();//next linguistic object should be no tree
          right = parseRightHandSide(tmp_object);
        }
      } else
        throw new IncorrectInput(
          "Parse error: Right hand side of '"  + tree_name +
          "' identifier may not be a Predefined Identifier."  ,
          ident_info.getStartPosition(),
          ident_info.getEndPosition()
        );
      if (tree_name.equals(":")) {//declaration
        tree = new Declaration(tree_name, left, right, tree_start_position , tree_end_position );
        tree.setRepresentation (total_string.substring(tree_start_position , tree_end_position ));
        //added for views
        answer = expectTree(tree);//recursion (expect a context or line identifier)
      } else
      if (tree_name.equals(":=")) {//definition
        tree = new Definition(tree_name, left, right, tree_start_position , tree_end_position );
        tree.setRepresentation (total_string.substring(tree_start_position , tree_end_position ));
        //added for views
        answer = expectTree(tree);//recursion (expect a context or line identifier)
      } else
      if (tree_name.equals(",")) {//context
        tree = new Context(tree_name, left, right, tree_start_position , tree_end_position );
        tree.setRepresentation (total_string.substring(tree_start_position , tree_end_position ));
        //added for views
        answer = expectTree(tree);//recursion (expect a context or line identifier)
      } else
```

```
        if (tree_name.equals("|>")) {
          tree = new Line(tree_name, left, right, tree_start_position , tree_end_position );
          tree.setRepresentation (total_string .substring(tree_start_position , tree_end_position ));
          //added for views
          answer = expectTree(tree);//recursion (expect end of line)
        }
      } else
        throw new IncorrectInput (
          "Parse error: ':', ':=', ',', '|>' or 'end of line' identifier expected"   ,
          ident_info.getStartPosition (),
          ident_info.getEndPosition ()
        );
    }
    return answer;
  }

  private LinguisticObject parseRightHandSide (LinguisticObject left) throws IncorrectInput {
    //pre : given linguistic object is left hand side od declaration or definition or statement
    //post: declaration or definition object constructed and returned
    //      or statement returned
    LinguisticObject answer = null;
    if ( ident_info.getType().equals("end of line" ) || ident_info.getIdentifier ().equals(";") ||
         ident_info.getIdentifier ().equals("|>") ) {
      answer = left;//only one linguistic object left or only one
                    //linguistic object between two context/line identifiers.
                    //Statements (right hand side of context or line)
    } else {
      //pars until context/line identifier or end of line identifier
      String tree_name;
      int tree_start_position ;
      int tree_end_position ;
      LinguisticObject tree;
      LinguisticObject right;
      if ( ident_info.getIdentifier ().equals(":") || ident_info.getIdentifier ().equals(":=") ) {
        //declaration or definition (right hand side of context or line)
        tree_name = ident_info.getIdentifier ();
        tree_start_position = ident_info.getStartPosition ();
        tree_end_position = ident_info.getEndPosition ();
        nextIdentifier ();
        if ( ident_info.getType().equals("name") ||
             ident_info.getIdentifier ().equals("SET") || ident_info.getIdentifier ().equals("PROP") )
          right = betweenPredefined ();//next linguistic object should be no tree
        else
          throw new IncorrectInput (
            "Parse error: Right hand side of '" + tree_name +
            "' identifier may not be a Predefined Identifier."  ,
            ident_info.getStartPosition (),
            ident_info.getEndPosition ()
          );
        if (tree_name.equals(":")) {//declaration
          tree = new Declaration(tree_name, left, right, tree_start_position , tree_end_position );
          tree.setRepresentation (total_string .substring(tree_start_position , tree_end_position ));
          //added for views
          answer = tree;//done with right hand side
        } else
        if (tree_name.equals(":=")) {//definition
          tree = new Definition(tree_name, left, right, tree_start_position , tree_end_position );
          tree.setRepresentation (total_string .substring(tree_start_position , tree_end_position ));
          //added for views
          answer = tree;//done with right hand side
        }
      } else
```

```
        throw new IncorrectInput (
          "Parse error: ':', ':=' or 'end of line' identifier expected"  ,
          ident_info .getStartPosition (),
          ident_info .getEndPosition ()
        );
    }
    return answer;
  }

  private void nextIdentifier () {
    ident_info  = Parser.ParsIdentifier (total_string ,ident_info .getEndPosition ());
  }

  private LinguisticObject  betweenPredefined () throws IncorrectInput {
    //pre : identifier is no predefined identifier, except PROP or SET
    //post: construct one linguistic object of all identifiers between
    //      two predefined identifiers, PROP and SET are added in
    //      that one linguistc object
    LinguisticObject  answer;
    answer = null;
    if (ident_info .getIdentifier ().equals ("SET")) {//construct SetSort
      answer  = new SetSort (ident_info .getIdentifier (),
                             ident_info .getStartPosition (),
                             ident_info .getEndPosition ());
      answer.setRepresentation (total_string .substring (ident_info .getStartPosition (),
                                                         ident_info .getEndPosition ()));
      //added for views
      nextIdentifier ();
    } else
    if (ident_info .getIdentifier ().equals ("PROP")) {//construct PropSort
      answer  = new PropSort (ident_info .getIdentifier (),
                              ident_info .getStartPosition (),
                              ident_info .getEndPosition ());
      answer.setRepresentation (total_string .substring (ident_info .getStartPosition (),
                                                         ident_info .getEndPosition ()));
      //added for views
      nextIdentifier ();
    } else
    if (ident_info .getType ().equals ("name")) {
      IdentifierInformation  name_identifier  = ident_info ;//remember the name
      nextIdentifier ();
      if ( ident_info .getIdentifier ().equals ("[") )//name is a binder name
        answer  = binder (name_identifier );
      else
      if ( ident_info .getIdentifier ().equals ("(") )//name is a constant name
        answer  = constant (name_identifier );
      else
        answer  = variable (name_identifier );//name is a variable name
    } else throw new IncorrectInput (
        "Parse error: identifier representing a name expected, but "   +
        ident_info .getIdentifier () + "found!",
        ident_info .getStartPosition (),
        ident_info .getEndPosition ()
      );
  if ( ident_info .getType ().equals ("name") ||
       ident_info .getIdentifier ().equals ("SET") || ident_info .getIdentifier ().equals ("PROP") ) {
    //no predefined symbol reached (except PROP or SET)
    //construct a Combination object
    LinguisticObject tmp_object;
    tmp_object = answer;
    IdentifierInformation name_identifier ;
    answer = new Combination ();
    ((Combination )answer).addLinguisticObject (tmp_object);
    while ( ident_info .getType ().equals ("name") ||
            ident_info .getIdentifier ().equals ("SET") || ident_info .getIdentifier ().equals ("PROP") ) {
      if (ident_info .getIdentifier ().equals ("SET")) {//construct SetSort
        tmp_object = new SetSort (ident_info .getIdentifier (),
                                  ident_info .getStartPosition (),
                                  ident_info .getEndPosition ());
        tmp_object .setRepresentation (total_string .substring (ident_info .getStartPosition (),
                                                                ident_info .getEndPosition ()));
        //added for views
        nextIdentifier ();
      } else
      if (ident_info .getIdentifier ().equals ("PROP")) {//construct PropSort
        tmp_object = new PropSort (ident_info .getIdentifier (),
                                   ident_info .getStartPosition (),
                                   ident_info .getEndPosition ());
        tmp_object .setRepresentation (total_string .substring (ident_info .getStartPosition (),
                                                                ident_info .getEndPosition ()));
        //added for views
        nextIdentifier ();
      } else {//identifier symbol = "name"
        name_identifier = ident_info;//remember the name
        nextIdentifier ();
        if ( ident_info .getIdentifier ().equals ("[") )//name is a binder name
          tmp_object = binder (name_identifier );
        else
        if ( ident_info .getIdentifier ().equals ("(") )//name is a constant name
          tmp_object = constant (name_identifier );
        else
          tmp_object = variable (name_identifier );//name is a variable name
      }
      ((Combination )answer).addLinguisticObject (tmp_object);
    }//predefined identifer reached, combination object created
  }//predefined identifier reached, no combination
  return answer;
}

private LinguisticObject variable (IdentifierInformation name_identifier ) {
  //pre : name_identifier contains the name of the variable
  //post: construct the variable object
  Variable answer;
  answer = new Variable (name_identifier .getIdentifier (),
                         name_identifier .getStartPosition (),
                         name_identifier .getEndPosition ());
  answer.setRepresentation (total_string .substring (name_identifier .getStartPosition (),
                                                     name_identifier .getEndPosition ()));
  //added for views
  return answer;
}
private LinguisticObject constant(IdentifierInformation name_identifier) throws IncorrectInput {
  //pre : name_identifier contains the name of the constant
  //post: construct the constant object
  LinguisticObject tmp_object;
  LinguisticObject answer;
  answer = new Constant(name_identifier.getIdentifier(),
                        name_identifier.getStartPosition(),
                        name_identifier.getEndPosition());
  answer.setRepresentation(total_string.substring(name_identifier.getStartPosition(),
                                                  name_identifier.getEndPosition()));

  //added for views
  if ( ident_info.getIdentifier().equals("(") ) {
    nextIdentifier();
    while ( ident_info.getType().equals("name") ||
            ident_info.getIdentifier().equals("SET") || ident_info.getIdentifier().equals("PROP") ) {
      tmp_object = betweenPredefined();//parse parameter
      if (ident_info.getIdentifier().equals(",")) {//continue while ','
        nextIdentifier();
        if (ident_info.getIdentifier().equals(")")) //forbid ',)'
          throw new IncorrectInput(
            "Parse error: Expecting parameter, but found ')' identifier" ,
            ident_info.getStartPosition(),
            ident_info.getEndPosition()
          );
      } else
      if (! ident_info.getIdentifier().equals(")"))//parantheses must be closed
        throw new IncorrectInput(
          "Parse error: ')' identifier expected" ,
          ident_info.getStartPosition(),
          ident_info.getEndPosition()
        );
      ((Constant)answer).addParameter(tmp_object);
    }
    if (ident_info.getIdentifier().equals(")")) {
      nextIdentifier();
    } else
      if (ident_info.getType().equals("end of line")) {
        throw new IncorrectInput(
          "Parse error: Could not parse a correct constant, input not complete" ,
          ident_info.getStartPosition(),
          ident_info.getEndPosition()
        );
      } else
        throw new IncorrectInput(
          "Parse error: Could not parse a correct constant, wrong identifier encountered " +
          "(" + ident_info.getIdentifier() + ")",
          ident_info.getStartPosition(),
          ident_info.getEndPosition()
        );
  } else
    throw new IncorrectInput(
      "Parse error: '(' identifier expected " ,
      ident_info.getStartPosition(),
      ident_info.getEndPosition()
    );
    //is already checked in betweenPredefined function()
  return answer;
}
```

```
private LinguisticObject binder(IdentifierInformation name_identifier ) throws IncorrectInput {
  //pre : name_identifier contains the name of the binder
  //post: construct the binder object
  LinguisticObject tmp_object;
  LinguisticObject answer;
  LinguisticObject left;
  LinguisticObject right;
  answer = new Binder(name_identifier.getIdentifier(),
                      name_identifier.getStartPosition(),
                      name_identifier.getEndPosition());
  answer.setRepresentation(total_string.substring(name_identifier.getStartPosition(),
                                                  name_identifier.getEndPosition()));
  //added for views
  if (ident_info.getIdentifier().equals("[")) {
    nextIdentifier();
    if ( ident_info.getType().equals("name") ||
         ident_info.getIdentifier().equals("SET") || ident_info.getIdentifier().equals("PROP") )
      left = betweenPredefined();//left hand side of the declaration
    else
      throw new IncorrectInput(
        "Parse error: Inside the '[' and ']' identifiers a declaration is expected"   ,
        ident_info.getStartPosition(),
        ident_info.getEndPosition()
      );
    if (ident_info.getIdentifier().equals(":")) {
      tmp_object = parseRightHandSide(left);
    } else
      throw new IncorrectInput(
        "Parse error: Inside the '[' and ']' identifiers a declaration is expected"   ,
        ident_info.getStartPosition(),
        ident_info.getEndPosition()
      );
    ((Binder)answer).setDeclaration(tmp_object);//add the constructed declaration
    if ( ident_info.getIdentifier().equals("]")) //brackets must be closed
      nextIdentifier();
    else
      throw new IncorrectInput(
        "Parse error: ']' identifier expected"  ,
        ident_info.getStartPosition(),
        ident_info.getEndPosition()
      );
    if ( ident_info.getIdentifier().equals("(") ) {
      nextIdentifier();
      while ( ident_info.getType().equals("name") ||
              ident_info.getIdentifier().equals("SET") || ident_info.getIdentifier().equals("PROP") )
        tmp_object = betweenPredefined();//parse parameter
        if (ident_info.getIdentifier().equals(",")) {//continue while ','
          nextIdentifier();
          if (ident_info.getIdentifier().equals(")")) //afvangen ,)
            throw new IncorrectInput(
              "Parse error: Expecting parameter, but found ')' identifier"  ,
              ident_info.getStartPosition(),
              ident_info.getEndPosition()
            );
        } else if (! ident_info.getIdentifier().equals(")"))//parantheses must be closed
          throw new IncorrectInput(
            "Parse error: ')' identifier expected"  ,
            ident_info.getStartPosition(),
            ident_info.getEndPosition()
          );
        ((Binder)answer).addParameter(tmp_object);
      }
```

```
        if (ident_info .getIdentifier ().equals (")")) {
          nextIdentifier ();
        } else
          if (ident_info .getType ().equals ("end of line" )) {
            throw new IncorrectInput (
              "Parse error: Could not parse a correct binder, input not complete"    ,
              ident_info .getStartPosition (),
              ident_info .getEndPosition ()
            );
          } else
            throw new IncorrectInput (
              "Parse error: Could not parse a correct binder, wrong identifier encountered "     +
              "(" + ident_info .getIdentifier () + ")",
              ident_info .getStartPosition (),
              ident_info .getEndPosition ()
            );
      } else
        throw new IncorrectInput (
          "Parse error: '(' identifier expected "  ,
          ident_info .getStartPosition (),
          ident_info .getEndPosition ()
        );
    } else
      throw new IncorrectInput (
        "Parse error: Expecting '['" ,
        ident_info .getStartPosition (),
        ident_info .getEndPosition ()
      );
      //is already checked in betweenPredefined function()
    return answer;
  }
}
```

```
package data.checkstructure ;

import data.linguisticstructure .*;
import data.WTTAMessages .IncorrectInput ;
import java.util.Vector;

/**
 * Title:         WTTASyntaxChecker
 * Description:   Implementation that checks the syntax rules
 * Copyright:     Copyright (c) 2002
 * Company:       Eindhoven University of Technology
 * @author:       Roel Körvers
 * @version:      1.0
 */

public class WTTASyntaxChecker  {

  public WTTASyntaxChecker () {
  }

  public void syntaxCheck (LinguisticObject  obj) throws IncorrectInput  {
    //pre   :
    //post  : starts syntax checking the linguistic object tree.
    if (obj instanceof Line)
      syntaxCheckLine ((Line)obj);//only lines can be entered.
    else
      throw new IncorrectInput (
        "Incorrect syntax: The input must be a line! (syntax rule 19)"   ,
        obj.getStartPosition (),
        obj.getEndPosition ()
      );
  }

  private void syntaxCheckAtomic (Atomic obj, String message)
    //the message determines if the Atomic object represents
    //an argument (syntax rules 4a, 4b, 4c) or
    //a statement (syntax rules 13a, 13b, 13c)
    throws IncorrectInput  {
    if (obj instanceof Variable)
      syntaxCheckVariable ((Variable)obj);
    else if (obj instanceof Constant)
      syntaxCheckConstant ((Constant)obj);
    else if (obj instanceof Binder)
      syntaxCheckBinder ((Binder)obj);
    else //only Variable, Constant or Binder can extend an AtomicObject.
      throw new IncorrectInput (
        message,
        obj.getStartPosition (),
        obj.getEndPosition ()
      );
  }

  private void syntaxCheckVariable (Variable obj) {
    //Variables are represented by a name identifier,
    //otherwise they cannot be constructed.
    //Whether the name is a correct name, is checked by the LtypeChecker.
  }
```

```
private void syntaxCheckConstant (Constant obj) throws IncorrectInput {
  //Constants are represented by a name identifier,
  //otherwise they cannot be constructed.
  //Whether the name is a correct name, is checked by the LtypeChecker
  //Constants have parameters that have to be checked.
  Vector parameters;
  parameters = obj.getParameters ();
  for (int i = 0; i < parameters.size(); i++) {
    if (parameters.elementAt(i) instanceof Atomic) {
      //syntax rule 1, syntax rules 4a, 4b, 4c
      syntaxCheckAtomic ((Atomic)parameters.elementAt(i),
      "Incorrect syntax: Variable, constant or binder expected! "    +
      "(syntax rule 4a, 4b, 4c)" );
    } else if (parameters.elementAt(i) instanceof Combination) {
      //syntax rule 1, syntax rule 4d
      syntaxCheckCombination ((Combination)parameters.elementAt(i));
    } else
      throw new IncorrectInput (
        "Incorrect syntax: Parameter " +(i+1)+" is no argument! "  +
        "(syntax rule 1)" ,
        ((LinguisticObject )parameters.elementAt(i)).getStartPosition (),
        ((LinguisticObject )parameters.elementAt(i)).getEndPosition ()
      );
  }
}

private void syntaxCheckBinder (Binder obj) throws IncorrectInput {
  //Binders are represented by a name identifier,
  //otherwise they cannot be constructed.
  //Whether the name is a correct name, is checked by the LtypeChecker
  //Binders have a declaration that has to be checked.
  LinguisticObject declaration;
  declaration = obj.getDeclaration ();
  if (declaration instanceof Declaration) {
    //syntax rule 2
    syntaxCheckDeclaration ((Declaration)declaration);
  } else
    throw new IncorrectInput (
      "Incorrect syntax: Declaration expected! (syntax rule 2)"  ,
      declaration.getStartPosition (),
      declaration.getEndPosition ()
    );
  //Binders have parameters that have to be checked.
  Vector parameters;
  parameters = obj.getParameters ();
  for (int i = 0; i < parameters.size(); i++) {
    if (parameters.elementAt(i) instanceof Atomic) {
      //syntax rule 3, syntax rules 4a, 4b, 4c
      syntaxCheckAtomic ((Atomic)parameters.elementAt(i),
      "Incorrect syntax: Variable, constant or binder expected! "    +
      "(syntax rule 4a, 4b, 4c)" );
    } else if (parameters.elementAt(i) instanceof Combination) {
      //syntax rule 3, syntax rule 4d
      syntaxCheckCombination ((Combination)parameters.elementAt(i));
    } else
      throw new IncorrectInput (
        "Incorrect syntax: Parameter " +(i+1)+" is no argument! (syntax rule 3)" ,
        ((LinguisticObject )parameters.elementAt(i)).getStartPosition (),
        ((LinguisticObject )parameters.elementAt(i)).getEndPosition ()
      );
  }
}
```

```
private void syntaxCheckCombination (Combination obj) throws IncorrectInput {
  //Combinations have a list of linguistic objects, at least one.
  Vector linguistic_objects ;
  linguistic_objects  = obj.getLinguisticObjects ();
  if (linguistic_objects .size() < 2) {
    //syntax rule 5
    throw new IncorrectInput (
     "Incorrect syntax: an adjective noun combination is build from "    +
     "at least two constants! (syntax rule 5)"  ,
     obj.getStartPosition (),
     obj.getEndPosition ()
    );
  }
  //The linguistic objects must be constants.
  for (int i = 0; i < linguistic_objects .size(); i++) {
    if (linguistic_objects .elementAt (i) instanceof Constant) {
      //syntax rule 6
      syntaxCheckConstant ((Constant)linguistic_objects .elementAt (i));
    } else
      throw new IncorrectInput (
       "Incorrect syntax: Only constants are allowed in an "    +
       "adjective noun combination! (syntax rule 6)"  ,
       ((LinguisticObject )linguistic_objects .elementAt (i)).getStartPosition (),
       ((LinguisticObject )linguistic_objects .elementAt (i)).getEndPosition ()
      );
  }
}

private void syntaxCheckDeclaration (Declaration obj) throws IncorrectInput {
  //A declaration is represented by the ':' symbol
  if (! obj.getIdentifier ().equals (":")) {
    //syntax rule 7
    throw new IncorrectInput (
      "Incorrect syntax: A declaration must be represented by the "    +
      "':' symbol! (syntax rule 7)" ,
      obj.getStartPosition (),
      obj.getEndPosition ()
    );
  }
  //The left hand side of a declaration must be a variable
  LinguisticObject  left;
  left = obj.getLeftSide ();
  if (left instanceof Variable) {
    //syntax rule 8
    syntaxCheckVariable ((Variable)left);
  } else
    throw new IncorrectInput (
      "Incorrect syntax: Variable expected! (syntax rule 8)"  ,
      left.getStartPosition (),
      left.getEndPosition ()
    );
  //The right hand side of a declaration must be a sort or an argument
  LinguisticObject  right;
  right = obj.getRightSide ();
  if (right instanceof Sort) {
    //syntax rules 9a, 9b SortsObject are syntax correct by construction
  } else
  if (right instanceof Atomic) {
    //syntax rule 9c, syntax rules 4a, 4b, 4c
    syntaxCheckAtomic ((Atomic)right,
      "Incorrect syntax: Variable, constant or binder expected! "    +
      "(syntax rule 4a, 4b, 4c)" );
    } else if (right instanceof Combination) {
      //syntax rule 9c, syntax rule 4d
      syntaxCheckCombination ((Combination)right);
    } else
      throw new IncorrectInput (
        "Incorrect syntax: Incorrect rightside of a declaration! "    +
        "(synax rules 9a, 9b, 9c)" ,
        right.getStartPosition (),
        right.getEndPosition ()
      );
  }

  private void syntaxCheckDefinition (Definition obj) throws IncorrectInput  {
    //A definition is represented by the ':=' symbol
    if (! obj.getIdentifier ().equals (":=")) {
      //syntax rule 10
      throw new IncorrectInput (
        "Incorrect syntax: A definition must be represented by the "    +
        "':=' symbol! (syntax rule 10)" ,
        obj.getStartPosition (),
        obj.getEndPosition ()
      );
    }
    //The left hand side of a definition must be a constant
    LinguisticObject  left;
    left = obj.getLeftSide ();
    if (left instanceof Constant) {
      //syntax rule 11
      syntaxCheckConstant ((Constant)left);
    } else
      throw new IncorrectInput (
        "Incorrect syntax: Constant expected! (syntax rule 11)"   ,
        left.getStartPosition (),
        left.getEndPosition ()
      );
    //The right hand side of a definition must be an argument
    LinguisticObject  right;
    right = obj.getRightSide ();
    if (right instanceof Atomic) {
      //syntax rule 12, syntax rules 4a, 4b, 4c
      syntaxCheckAtomic ((Atomic)right,
        "Incorrect syntax: Variable, constant or binder expected! "    +
        "(syntax rule 4a, 4b, 4c)" );
    } else if (right instanceof Combination) {
      //syntax rule 12, syntax rule 4d
      syntaxCheckCombination ((Combination)right);
    } else
      throw new IncorrectInput (
        "Incorrect syntax: Incorrect rightside of a definition! "    +
        "(syntax rule 12)" ,
        right.getStartPosition (),
        right.getEndPosition ()
      );
  }
```

```
private void syntaxCheckContext (Context obj) throws IncorrectInput {
  //A context is represented by the ',' symbol
  //The context object only represents the context list!
  if (! obj.getIdentifier().equals(",")) {
    //syntax rule 15
    throw new IncorrectInput (
      "Incorrect syntax: A context list must be seperated by the "    +
      "',' symbol! (syntax rule 15)" ,
      obj.getStartPosition (),
      obj.getEndPosition ()
    );
  }
  //The left hand side of a context list must be a context,
  //                                              a declaration or a statement.
  LinguisticObject  left;
  left = obj.getLeftSide ();
  if (left instanceof Context) {
    //syntax rule 14d, context
    syntaxCheckContext ((Context)left);
  } else if (left instanceof Declaration) {
    //syntax rule 14d, declaration
    syntaxCheckDeclaration ((Declaration)left);
  } else if (left instanceof Atomic) {
    //syntax rule 14d, statement, syntax rules 13a, 13b, 13c
    syntaxCheckAtomic ((Atomic)left,
    "Incorrect syntax: Variable, constant or binder expected! "    +
    "(syntax rule 13a, 13b, 13c)" );
  } else
    throw new IncorrectInput (
      "Incorrect syntax: Incorrect context list (syntax rule 14d)"   ,
      left.getStartPosition (),
      left.getEndPosition ()
    );
  //The right hand side of a context list must be a declaration
  //                                              or a statement.
  LinguisticObject  right;
  right = obj.getRightSide ();
  if (right instanceof Declaration) {
    //syntax rule 14d, declaration
    syntaxCheckDeclaration ((Declaration)right);
  } else if (right instanceof Atomic) {
    //syntax rule 14d, statement, syntax rules 13a, 13b, 13c
    syntaxCheckAtomic ((Atomic)right,
    "Incorrect syntax: Variable, constant or binder expected! "    +
    "(syntax rule 13a, 13b, 13c)" );
  } else
    throw new IncorrectInput (
      "Incorrect syntax: Incorrect context list (syntax rule 14d)"   ,
      right.getStartPosition (),
      right.getEndPosition ()
    );
}
```

```
 private void syntaxCheckLine (Line obj) throws IncorrectInput  {
   //A line is represented by the '|>' symbol
   if (! obj.getIdentifier ().equals ("|>")) {
     //syntax rule 16
     throw new IncorrectInput (
       "Incorrect syntax: A line must be represented by the "    +
       "'|>' symbol! (syntax rule 16)" ,
       obj.getStartPosition (),
       obj.getEndPosition ()
     );
   }
   //The left hand side of a line must be a empty context, a declaration
   //                                     a statement or a context list.
   LinguisticObject  left;
   left = obj.getLeftSide ();
   if (left instanceof EmptyContext ) {
     //syntax rule 14a
   } else if (left instanceof Declaration ) {
     //syntax rule 14b
     syntaxCheckDeclaration ((Declaration )left);
   } else if (left instanceof Atomic ) {
     //syntax rule 14c, syntax rules 13a, 13b, 13c
     syntaxCheckAtomic ((Atomic )left,
     "Incorrect syntax: Variable, constant or binder expected! "    +
     "(syntax rule 13a, 13b, 13c)" );
   } else if (left instanceof Context ) {
     //syntax rule 14d, context list
     syntaxCheckContext ((Context )left);
   } else
       throw new IncorrectInput (
         "Incorrect syntax: Incorrect context "   +
         "(syntax rules 14a, 14b, 14c, 14d)"  ,
         left.getStartPosition (),
         left.getEndPosition ()
       );
   //context is correct; syntax rule 17
   //The right hand side of a line must be a definition or statement.
   //Extended to check and return Ltypes of all lines.
   LinguisticObject  right;
   right = obj.getRightSide ();
   if (right instanceof Definition ) {
     //syntax rule 18a
     syntaxCheckDefinition ((Definition )right);
   } else if (right instanceof Atomic ) {
     //syntax rule 18b, syntax rules 13a, 13b, 13c
     syntaxCheckAtomic ((Atomic )right,
     "Incorrect syntax: Variable, constant or binder expected! "    +
     "(syntax rule 13a, 13b, 13c)" );
   } else if (right instanceof Combination ) {//extension
     syntaxCheckCombination ((Combination )right);
   } else
     throw new IncorrectInput (
       "Incorrect syntax: Incorrect rightside of a line "    +
       "(syntax rules 18a, 18b)" ,
       right.getStartPosition (),
       right.getEndPosition ()
     );
 }
}
```

```
package data.checkstructure ;

import java.util.Vector;
import data.linguisticstructure .*;
import data.WTTAInformation .*;
import data.WTTAMessages .*;

/**
 * Title:          WTTALtypeChecker
 * Description:    Implementation that checks the Ltype rules
 *                 First the syntax rules must be checked!
 *                 else error will be generated on syntax
 * Copyright:      Copyright (c) 2002
 * Company:        Eindhoven University of Technology
 * @author:        Roel Körvers
 * @version:       1.0
 */

public class WTTALtypeChecker  {
  Object newestELI ;
  //If the right hand side, of the last line found correct, was a definition:
  //    newestELI stores the ELI object constructed in that definition
  //If the right hand side, of the last line found correct, was a statement:
  //    newestELI stores null

  Vector EG;//Essential Global, all defined constants and binders
  Vector EL;//Essential Local, all declared variables

  public WTTALtypeChecker () {
    newestELI  = null;
    EG = new Vector();
    EL = new Vector();
  }

  public Object getLatestEssentialLtypeInformation  () {
    //pre   :
    //post  : returns newestELI
    return newestELI;
  }

  public byte LtypeCheck(LinguisticObject  obj, Vector EssentialGlobal )
    throws IncorrectInput  {
    //pre   :
    //post  : starts Ltype checking the linguistic object tree.
    //        returns the Ltype found on the right hand side of the line or
    //        returns the Ltype found on the right hand side of the definition
    //          that is on the right hand side of the line
    EG = EssentialGlobal ;
    EL = new Vector();//every line has its own context
    byte answer;
    answer = 0x0;
    if (obj instanceof  Line) {
      answer = LtypeCheckLine ((Line)obj);
      //begin with Ltype rule line
    } else
       throw new WrongSyntaxInLtypeCheckError ();
       //should be checked by syntax rules
    return answer;
  }
```

```
private byte LtypeCheckAtomic (Atomic obj, byte Ltype) throws IncorrectInput {
  //determines which Ltype check must be made
  byte answer;
  answer = 0x0;
  if (obj instanceof Variable)
    answer = LtypeCheckVariable ((Variable)obj,Ltype);
  else if (obj instanceof Constant)
    answer = LtypeCheckConstant ((Constant)obj,Ltype);
  else if (obj instanceof Binder)
    answer = LtypeCheckBinder ((Binder)obj,Ltype);
  else
    throw new WrongSyntaxInLtypeCheckError ();
     //should be checked by syntax rules
  return answer;
}

private byte LtypeCheckVariable (Variable obj, byte Ltype) throws IncorrectInput {
  byte answer;
  answer = 0x0;
  //find an ELI object in EL that has the same name,
  //automatically is a variable
  for (int i = 0; i < EL.size() && (answer == 0x0); i++) {
    if ( obj.getName ().equals ( ((ELI)EL.elementAt (i)).getName () ) ) {
      answer = ((ELI)EL.elementAt (i)).getLtypeOut ();
    }
  }
  if (answer == 0x0)
    throw new IncorrectInput (
      "Incorrect Ltype: Undeclared variable! (derivation rule var)"  ,
      obj.getStartPosition (),
      obj.getEndPosition ()
    );
  //check whether the Ltype of the variable is one of the permitted Ltypes
  if ( (answer & Ltype) != answer) {
    throw new IncorrectInput (
      "Incorrect Ltype: Variable has Ltype '" +ELI.Ltype(answer)+"', but " +
      "Ltype '"+ELI.Ltype(Ltype)+"' expected! (derivation rule var)"  ,
      obj.getStartPosition (),
      obj.getEndPosition ()
    );
  }
  return answer;
}

private byte LtypeCheckConstant (Constant obj, byte Ltype)
  throws IncorrectInput {
  byte answer;
  answer = 0x0;
  //exception 'numbers': numbers have no parameters, their out Ltype is 'term'
  if ( (int)obj.getName ().charAt (0) >= 48 &&
       (int)obj.getName ().charAt (0) <= 57) {//name starts with a number
    if (obj.getArity () != 0)
      throw new IncorrectInput (
        "Incorrect Ltype: 0 parameters expected, "  +
        "but "+obj.getArity () +" parameters found! (derivation rule cons)"  ,
        obj.getStartPosition (),
        obj.getEndPosition ()
      );
    answer = ELI.TERM;
  } else {
      //find an ELI object in EG that has the same name
      ELI ELIConstant ;
      ELIConstant  = new ELI();
      for (int i = 0; i < EG.size() && ELIConstant .getName().equals(""); i++) {
        if ( obj.getName().equals( ((ELI)EG.elementAt(i)).getName() )  ) {
          ELIConstant .replaceBy((ELI)EG.elementAt(i));
        }
      }
      if (ELIConstant .getName().equals(""))
        throw new IncorrectInput (
          "Incorrect Ltype: Undefined Constant (derivation rule cons)"  ,
          obj.getStartPosition (),
          obj.getEndPosition ()
        );
      //check the category of the ELI object found
      if ( ELIConstant .getCategory () != ELI.CONSTANT )
        throw new IncorrectInput (
          "Incorrect Ltype: '"  + ELIConstant .getName() +
          "' is no constant (derivation rule cons)"  ,
          obj.getStartPosition (),
          obj.getEndPosition ()
        );
      //check the number of parameters
      if (! (obj.getArity() == ELIConstant .getArity()))
        throw new IncorrectInput (
          "Incorrect Ltype: " +ELIConstant .getArity()+" parameters expected, "  +
          "but "+obj.getArity() +" parameters found! (derivation rule cons)"  ,
          obj.getStartPosition (),
          obj.getEndPosition ()
        );
      //check that all parameters have the correct Ltype
      Vector parameters ;
      parameters  = obj.getParameters ();
      boolean parametersOk ;
      parametersOk  = true;
      for (int i = 0; i < parameters .size() && parametersOk ; i++) {
        if (parameters .elementAt(i) instanceof Atomic)
          LtypeCheckAtomic (
              (Atomic)parameters .elementAt(i), ELIConstant .getLtypeIn(i));
        else if (parameters .elementAt(i) instanceof Combination)
          LtypeCheckCombination (
              (Combination )parameters .elementAt(i), ELIConstant .getLtypeIn (i));
        else
          throw new  WrongSyntaxInLtypeCheckError ();
          //should be checked by syntax rules
      }
      answer  = ELIConstant .getLtypeOut ();
    }
    //check whether the Ltype of the constant is one of the permitted Ltypes
    if ((answer & Ltype) != answer)
      throw new IncorrectInput (
        "Incorrect Ltype: Constant has Ltype '"  +ELI.Ltype(answer)+"', but " +
        "Ltype '"+ELI.Ltype(Ltype)+"' expected! (derivation rule cons)"  ,
        obj.getStartPosition (),
        obj.getEndPosition ()
      );
    return answer;
  }
```

```
private byte LtypeCheckBinder (Binder obj, byte Ltype) throws IncorrectInput {
  //find an ELI object in EG that has the same name
  ELI ELIBinder;
  ELIBinder = new ELI();
  for (int i = 0; i < EG.size() && ELIBinder.getName().equals(""); i++) {
    if ( obj.getName().equals( ((ELI)EG.elementAt(i)).getName() ) ) {
      ELIBinder.replaceBy((ELI)EG.elementAt(i));
    }
  }
  if (ELIBinder.getName().equals(""))
    throw new IncorrectInput (
      "Incorrect Ltype: Undefined Binder (derivation rule bind)"  ,
      obj.getStartPosition (),
      obj.getEndPosition ()
    );
  //check the category of the ELI object found
  if (ELIBinder.getCategory() != ELI.BINDER)
      throw new IncorrectInput (
        "Incorrect Ltype: '" + ELIBinder.getName() +
        "' is no binder (derivation rule bind)"  ,
        obj.getStartPosition (),
        obj.getEndPosition ()
      );
  //check that the declaration is correct
  LinguisticObject declaration;
  declaration = obj.getDeclaration ();
  ELI localVariable;
  if (declaration instanceof Declaration)
    localVariable = LtypeCheckDeclaration ((Declaration)declaration);
  else
    throw new  WrongSyntaxInLtypeCheckError ();
    //should be checked by syntax rules
  //check the number of parameters
  if (! (obj.getArity() == ELIBinder.getArity()))
    throw new IncorrectInput (
      "Incorrect Ltype: " +ELIBinder.getArity()+" parameters expected, "  +
      "but "+obj.getArity() +" parameters found! (derivation rule bind)"  ,
      obj.getStartPosition (),
      obj.getEndPosition ()
    );
  //check that all parameters have the correct Ltype
  Vector parameters;
  parameters = obj.getParameters ();
  boolean parametersOk;
  parametersOk = true;
  for (int i = 0; i < parameters.size() && parametersOk; i++) {
    if (parameters.elementAt(i) instanceof Atomic)
      LtypeCheckAtomic (
          (Atomic)parameters.elementAt(i), ELIBinder.getLtypeIn(i));
    else if (parameters.elementAt(i) instanceof Combination)
      LtypeCheckCombination (
          (Combination)parameters.elementAt(i), ELIBinder.getLtypeIn(i));
    else
      throw new  WrongSyntaxInLtypeCheckError ();
      //should be checked by syntax rules
  }
```

```
    byte answer;
    answer = 0x0;
    if (ELIBinder.getLtypeOut() == ELI.TERM+ELI.SET+ELI.STAT) {
      //Iota binder has three possible out Ltypes
      //which Ltype depends on the local variable
      answer = localVariable.getLtypeOut();
    } else
      answer = ELIBinder.getLtypeOut();
    //check whether the Ltype of the binder is one of the permitted Ltypes
    if ((answer & Ltype) != answer)
      throw new IncorrectInput (
        "Incorrect Ltype: Binder has Ltype '" +ELI.Ltype(answer)+"', but " +
        "Ltype '"+ELI.Ltype(Ltype)+"' expected! (derivation rule bind)" ,
        obj.getStartPosition(),
        obj.getEndPosition()
      );
    //remove the local variable from EL
    EL.removeElement(localVariable);
    return answer;
  }

  private byte LtypeCheckCombination (Combination obj, byte Ltype)
    throws IncorrectInput {
    Vector linguistic_objects;
    linguistic_objects = obj.getLinguisticObjects();
    int amount = linguistic_objects.size();
    //Check that the last element has Ltype Noun
    if (linguistic_objects.elementAt(amount) instanceof Constant)
      LtypeCheckConstant (
            (Constant)linguistic_objects.elementAt(amount),ELI.ADJ);
    else
      throw new WrongSyntaxInLtypeCheckError();
      //should be checked by syntax rules
    //check that all other elements have Ltype Adj
    boolean elementsOk;
    elementsOk = true;
    for (int i = 0; i < (amount - 1) && elementsOk; i++) {
      if (linguistic_objects.elementAt(i) instanceof Constant)
        LtypeCheckConstant((Constant)linguistic_objects.elementAt(i),ELI.ADJ);
      else
        throw new WrongSyntaxInLtypeCheckError();
        //should be checked by syntax rules
    }
    //check whether the Ltype of the constant is one of the permitted Ltypes
    if ( (ELI.NOUN & Ltype) != ELI.NOUN)
      throw new IncorrectInput (
        "Incorrect Ltype: Combination has Ltype '" +ELI.Ltype(ELI.NOUN) +
        "', but Ltype '"+ELI.Ltype(Ltype)+"' expected! (derivation rule comb)" ,
        obj.getStartPosition(),
        obj.getEndPosition()
      );
    return ELI.NOUN;
  }
```

```
private ELI LtypeCheckDeclaration (Declaration obj) throws IncorrectInput {
  //check that the variable name is a 'fresh' name
  boolean isFresh;
  isFresh = true;
  LinguisticObject left;
  left = obj.getLeftSide();
  if (left instanceof Variable) {
    if ( (int)((Variable)left).getName().charAt(0) >= 48 &&
         (int)((Variable)left).getName().charAt(0) <= 57)
      isFresh = false;//the name is a number, therefore not fresh
      //numbers are supposted to be in the EG, but this is not possible
    for (int i = 0; i < EL.size() && isFresh; i++) {
      if ( ((ELI)EL.elementAt(i)).getName().equals(((Variable)left).getName()) )
        isFresh = false;//the name is declared locally, therefore not fresh
    }
    for (int i = 0; i < EG.size() && isFresh; i++) {
      if ( ((ELI)EG.elementAt(i)).getName().equals(((Variable)left).getName()) )
        isFresh = false;//the name is defined globally, therefore not fresh
    }
    if (! isFresh)
      throw new IncorrectInput (
        "Incorrect Ltype: The name of the variable to be declared is already used! "    +
        "(derivation rule term-decl or set/stat-decl)"  ,
        left.getStartPosition(),
        left.getEndPosition()
      );
  } else
    throw new WrongSyntaxInLtypeCheckError ();
    //should be checked by syntax rules
  String eli_name = ((Variable)left).getName();
  byte eli_category = ELI.VARIABLE;
  //check that the right hand side has a correct Ltype
  //and assign the Ltype to the declared variable.
  LinguisticObject right;
  right = obj.getRightSide();
  byte eli_out;
  if (right instanceof PropSort)
    eli_out = ELI.STAT;
  else if (right instanceof SetSort)
    eli_out = ELI.SET;
  else if (right instanceof Atomic) {
    LtypeCheckAtomic ((Atomic)right, (byte)(ELI.SET + ELI.NOUN));
    eli_out = ELI.TERM;
  } else if (right instanceof Combination) {
    LtypeCheckCombination ((Combination)right, (byte)(ELI.SET + ELI.NOUN));
    eli_out = ELI.TERM;
  } else
    throw new WrongSyntaxInLtypeCheckError ();
    //should be checked by syntax rules

  //make a new ELI and add the variable to the EL
  ELI newVariable = new ELI(eli_name, eli_category, eli_out);
  EL.addElement(newVariable);
  return newVariable;
}
```

```
private byte LtypeCheckDefinition (Definition obj) throws IncorrectInput {
  //Check that the constant name is a 'fresh' name
  ELI newConstant = new ELI();
  boolean isFresh;
  isFresh = true;
  LinguisticObject left;
  left = obj.getLeftSide();
  if (left instanceof Constant) {
    if ( (int)((Constant)left).getName().charAt(0) >= 48 &&
         (int)((Constant)left).getName().charAt(0) <= 57)
      isFresh = false;//the name is a number, therefore not fresh
      //numbers are suppposted to be in the EG, but this is not possible
    for (int i = 0; i < EL.size() && isFresh; i++) {
      if ( ((ELI)EL.elementAt(i)).getName().equals(((Constant)left).getName()) )
        isFresh = false;//the name is declared locally, therefore not fresh
    }
    for (int i = 0; i < EG.size() && isFresh; i++) {
      if ( ((ELI)EG.elementAt(i)).getName().equals(((Constant)left).getName()) )
        isFresh = false;//the name is defined globally, therefore not fresh
    }
    if (! isFresh)
      throw new IncorrectInput (
        "Incorrect Ltype: The name of the constant to be defined is already used! "    +
        "(derivation rule int-def)" ,
        left.getStartPosition(),
        left.getEndPosition()
      );
    //check the number of parameters.
    if (((Constant)left).getArity() != EL.size())
      throw new IncorrectInput (
        "Incorrect Ltype: The constant must have "  + EL.size() + " parameters " +
        "instead of " + ((Constant)left).getArity() +" parameters! (derivation rule int-def)" ,
        left.getStartPosition(),
        left.getEndPosition()
      );
    //check the names, in the same order as in context, of the parameters.
    Vector parameters;
    parameters = ((Constant)left).getParameters();
    for (int i = 0; i < EL.size(); i++) {
      if (parameters.elementAt(i) instanceof Variable) {
        if (! ((Variable)parameters.elementAt(i)).getName().equals(
              ((ELI)EL.elementAt(i)).getName() ) )
          throw new IncorrectInput (
            "Incorrect Ltype: '" +((ELI)EL.elementAt(i)).getName() +
            "' expected as parameter " + i +
            " (derivation rule int-def)" ,
            ((LinguisticObject )parameters.elementAt(i)).getStartPosition(),
            ((LinguisticObject )parameters.elementAt(i)).getEndPosition()
          );
      } else//all parameters of a constant that is defined must be
            //variables declared in the context.
        throw new IncorrectInput (
          "Incorrect Ltype: '" +((ELI)EL.elementAt(i)).getName() +
          "' expected as parameter " + i +
          " (derivation rule int-def)" ,
          ((LinguisticObject )parameters.elementAt(i)).getStartPosition(),
          ((LinguisticObject )parameters.elementAt(i)).getEndPosition()
        );
    }
  } else
    throw new WrongSyntaxInLtypeCheckError ();
    //should be checked by syntax rules
    String eli_name = ((Constant)left).getName();
    byte eli_category = ELI.CONSTANT;
    int eli_arity = EL.size();
    byte[] eli_in = new byte[eli_arity];
    for (int i = 0; i < EL.size(); i++) {
      eli_in[i] = ((ELI)EL.elementAt(i)).getLtypeOut() ;
    }
    //Check that the right hand side has a correct Ltype
    //and assign that Ltype to the defined constant.
    LinguisticObject right;
    right = obj.getRightSide();
    byte eli_out;
    if (right instanceof Atomic) {
      eli_out = LtypeCheckAtomic((Atomic)right,
                (byte)(ELI.TERM + ELI.SET + ELI.NOUN + ELI.ADJ + ELI.STAT));
    } else if (right instanceof Combination) {
      eli_out = LtypeCheckCombination((Combination)right,
                (byte)(ELI.TERM + ELI.SET + ELI.NOUN + ELI.ADJ + ELI.STAT));
    } else
      throw new WrongSyntaxInLtypeCheckError();
      //should be checked by syntax rules
    //make a new ELI or ALI for this constant and return it
    if ( ((Constant)left).isParanthesesRequired() )
      newConstant = new ELI(eli_name,eli_category,eli_arity,eli_in,eli_out);
    else
      newConstant = new ALI(eli_name,eli_category,eli_arity,eli_in,eli_out);
    //The ELI constructed may not yet be added to the EG, only
    //if the user confirms that the line must be added to the book.
    //The ELI is stored in the newestELI field and can be retrived if the line is added
    newestELI = newConstant;
    return (byte)(eli_out + ELI.DEFINITION);
    //Return the Ltype of the right hand side and add that it is a definition.
  }

  private void LtypeCheckContext(Context obj) throws IncorrectInput {
    //check the left hand side of the context list
    LinguisticObject left;
    left = obj.getLeftSide();
    if (left instanceof Declaration)              //declaration
      LtypeCheckDeclaration((Declaration)left);
    else if (left instanceof Atomic)              //statement
      LtypeCheckAtomic((Atomic)left, ELI.STAT);
    else if (left instanceof Context)             //context
      LtypeCheckContext((Context)left);
    else
      throw new WrongSyntaxInLtypeCheckError();
    //check the right hand side of the context list
    LinguisticObject right;
    right = obj.getRightSide();
    if (right instanceof Declaration)             //declaration
      LtypeCheckDeclaration((Declaration)right);
    else if (right instanceof Atomic)             //statement
      LtypeCheckAtomic((Atomic)right, ELI.STAT);
    else
      throw new WrongSyntaxInLtypeCheckError();
      //should be checked by syntax rules
  }
```

```
 private byte LtypeCheckLine (Line obj) throws IncorrectInput {
   //check the left hand side of the line
   LinguisticObject left;
   left = obj.getLeftSide();
   if (left instanceof EmptyContext) {          //empty context
     //The book is correct by construction
   } else if (left instanceof Declaration)      //declaration
     LtypeCheckDeclaration ((Declaration)left);
   else if (left instanceof Atomic)             //statement
     LtypeCheckAtomic ((Atomic)left, ELI.STAT);
   else if (left instanceof Context)            //context list
     LtypeCheckContext ((Context)left);
   else
     throw new WrongSyntaxInLtypeCheckError ();
   //check the right hand side of the line
   //Extended to check and return Ltypes of all lines.
   byte answer;
   answer = 0x0;
   LinguisticObject right;
   right = obj.getRightSide();
   if (right instanceof Atomic) {
     answer = LtypeCheckAtomic ((Atomic)right,
              (byte)(ELI.TERM + ELI.SET + ELI.NOUN + ELI.ADJ + ELI.STAT));
     //Strict would be only allowing ELI.STAT.
   } else if (right instanceof Combination) {
     answer = LtypeCheckCombination ((Combination)right,
              (byte)(ELI.TERM + ELI.SET + ELI.NOUN + ELI.ADJ + ELI.STAT));
     //only Ltype NOUN possible, but this is checked in LtypeCheckCombination
   } else if (right instanceof Definition)      //definition
     answer = LtypeCheckDefinition ((Definition)right);
   else
     throw new WrongSyntaxInLtypeCheckError ();
     //should be checked by syntax rules
   return answer;
   //the Ltype returned is the Ltype of the right hand side of the line or
   //the Ltype of the right hand side of the definition that stands at
   //the right hand side of the line. The difference can be made by the Definition bit.
 }
}
```

letters	: 'a' .. 'z', 'A' .. 'Z'		
digits	: '0' .. '9'		
symbols	: the symbols on the keyboard. (e.g. '!', '@', '#', '$', '%', '^', '&' etc..)		
separators	: <space> and <enter> (or end of line)		
word	: letter (letter/digit/'_')*		
symbol	: symbol symbols*		
number	: digit digit*/(digit* '.' digit*)		
predefined	: ':', ';', ':=', ',', '.', '|>', '(', ')', '[', ']', '"', 'PROP', 'SET'		
name	: *word/symbol/number**	without *predefined* or *symbols* containing *predefined*	
comment	: '"' (letter/digit/symbol/<space>)* '"'		
	comment can be placed in front or after, *names* or *predefined* and even inside *words*.		
variable	: *name*		
	: '(' *variable* ')'		
constant	: *name* '(' *parameterlist* ')'		
	: *argument name argument*		
	: *name argument*		
	: *name*		
	: '(' *constant* ')'		
binder	: *name* '[' *declaration* ']' '(' *parameterlist* ')'		
	: '(' *binder* ')'		
parameterlist	: ∅	*parameters*	: *argument*
	: *argument*		: *argument* ',' *parameters*
	: *argument* ',' *parameters*		
argument	: *variable*		
	: *constant*		
	: *binder*		
	: *combination*		
combination	: *constant combinationlist*	*combinationlist*	: *constant*
	: '(' *combination* ')'		: *constant combinationlist*
declaration	: *multivar* ':' 'PROP'	*multivar*	: *variable*
	: *multivar* ':' 'SET'		: *variable* ';' *multivar*
	: *multivar* ':' *argument*		
definition	: *constant* ':=' *argument*		
statement	: *variable*		
	: *constant*		
	: *binder*		
context	: ∅	*contexts*	: *declaration*
	: *declaration*		: *statement*
	: *statement*		: *declaration* ',' *contexts*
	: *declaration* ',' *contexts*		: *statement* ',' *context*
	: *statement* ',' *contexts*		
line	: *context* '|>' *definition*		
	: *context* '|>' *statement*		

```java
package data.parsinput2structure ;

import java.util.Vector;
import data.linguisticstructure .*;
import data.WTTAInformation .*;
import data.WTTAMessages .IncorrectInput ;

/**
 * Title:         WTTAUnsugared2Structure
 * Description:   Unsugared language parser
 *                uses the 'Parser' to receive identifiers and constructs
 *                the linguistic object tree, by combining the identifiers
 * Copyright:     Copyright (c) 2002
 * Company:       Eindhoven University of Technology
 * @author:       Roel Körvers
 * @version:      1.0
 */

public class WTTASugared2Structure {
  private String total_string ;
  private Vector EG;//Essential Global
  private Vector var_in_cont ;//remember the variable declared (to complete definitions)
  private ALI additional_linguistic_info ;
  private IdentifierInformation  ident_info ;

  public WTTASugared2Structure () {
    total_string  = "";
    EG = new Vector ();
    var_in_cont  = new Vector ();
    additional_linguistic_info  = null;
    ident_info  = new IdentifierInformation ();
  }

  public LinguisticObject  build(String string2Bparsed , Vector EssentialGlobal ) throws IncorrectInput {
    //pre :
    //post: given string is parsed into a liguistic object tree
    total_string  = string2Bparsed ;
    EG = EssentialGlobal ;//EG can change after every line
    var_in_cont .removeAllElements ();
    additional_linguistic_info  = null;
    ident_info  = Parser .ParsIdentifier (total_string );//first identifier
    LinguisticObject  answer;
    answer = null;
    LinguisticObject  first;
    if ( ident_info .getType ().equals ("name" ) || ident_info .getIdentifier ().equals ("(")  ||
         ident_info .getIdentifier ().equals ("PROP" ) || ident_info .getIdentifier ().equals ("SET" )) {
      first = betweenPredefined ();//first linguistic object is no tree
      if (first instanceof Variable && ident_info .getIdentifier ().equals (";" )){
        //multiple declaration case 1) input starts with a multiple declaration
        answer = expectTree ( multiDeclaration (first, null, null) );
        //no prior linguistic object tree, hence null
      } else {
        answer = expectTree (first);
      }//after this object a tree identifier is expected
    } else
    if (ident_info .getIdentifier ().equals ("|>" )) {//begins with a line identifier => empty context
      answer = expectTree (new EmptyContext ());//Line with empty context is constructed
    } else
    if (ident_info .getType ().equals ("end of line" ) ) {
      if ( string2Bparsed .trim().length() != 0) {//comment
        answer = null;//no error, but the line doesnot have to be checked.
      } else {
        throw new IncorrectInput (
          "Parse error: Input may not be empty!"  ,
          ident_info .getStartPosition (),
          ident_info .getEndPosition ()
        );
      }
    } else
      throw new IncorrectInput (
        "Parse error: First identifier may not be a Predefined Identifier, except the '|>' identifier."    ,
        ident_info .getStartPosition (),
        ident_info .getEndPosition ()
      );
    return answer;
  }

  private LinguisticObject  expectTree (LinguisticObject  left) throws IncorrectInput {
    //pre : binary tree identifier expected given linguistic object is left hand side
    //post: binary tree object constructed with left and right hand side
    //      recursivly constructed of all identifiers to the right
    LinguisticObject  answer;
    answer = null;
    if (ident_info .getType ().equals ("end of line" ))
      answer = left;//this (tree) object is the entire linguistic object tree
    else {
      String tree_name ;
      LinguisticObject  tree;
      LinguisticObject  right;
      LinguisticObject  tmp_object  = null;
      tree = null;
      right = null;
      if ( ident_info .getIdentifier ().equals (":") || ident_info .getIdentifier ().equals (":=") ||
           ident_info .getIdentifier ().equals (",") || ident_info .getIdentifier ().equals ("|>") ) {
        //binary tree identifier
        tree_name  = ident_info .getIdentifier ();
        IdentifierInformation  tree_ident  = ident_info ;
        nextIdentifier ();// identifier after binary tree identifier should be no tree
        if ( ident_info .getType ().equals ("name") || ident_info .getIdentifier ().equals ("(")
          || ident_info .getIdentifier ().equals ("SET") || ident_info .getIdentifier ().equals ("PROP") ) {
          if (tree_name .equals (":") || tree_name .equals (":="))//declaration or definition
            right  = betweenPredefined ();//next linguistic object should be no tree
          else
          if (tree_name .equals (",") || tree_name .equals ("|>")){//context or line
            tmp_object  = betweenPredefined ();//next linguistic object should be no tree
            if (tmp_object  instanceof Variable && ident_info .getIdentifier ().equals (";")){
              //multiple declaration case 2) multiple declaration inside the context
              tree_name  = ";";
            } else {
              right  = parseRightHandSide (tmp_object );
            }
          }
        } else
          throw new IncorrectInput (
            "Parse error: Right hand side of '"  + tree_name +
            "' identifier may not be a Predefined Identifier."  ,
            ident_info .getStartPosition (),
            ident_info .getEndPosition ()
          );
      if (tree_name.equals(":")) {//declaration
        tree = new Declaration(tree_ident.getIdentifier(),
                               left,
                               right,
                               tree_ident.getStartPosition(),
                               tree_ident.getEndPosition());
        tree.setRepresentation(total_string.substring(tree_ident.getStartPosition(),
                                                      tree_ident.getEndPosition()));
        //added for views
        if (left instanceof Variable)//remember the variables to autocomplete definitions
          var_in_cont.addElement(((Variable)left).getName());
        else//error, let the syntax check catch it
          var_in_cont.addElement(left.toString());
        answer = expectTree(tree);//recursion (expect a context or line identifier)
      } else
      if (tree_name.equals(":=")) {//definition
        //added for definitions with no parentheses.
        if (left instanceof Variable && var_in_cont.size() == 0) {
          //definition of constant with no parameters.
          Variable oldLeft = (Variable)left;
          left = new Constant(oldLeft.getName(),
                              oldLeft.getStartPosition(),
                              oldLeft.getEndPosition(),
                              false);
          left.setRepresentation(total_string.substring(oldLeft.getStartPosition(),
                                                        oldLeft.getEndPosition()));
          //added for views
        } else
        if ( left instanceof Combination ) {
          if ( var_in_cont.size() == 1 ) {//definition of unary constant
            Vector combi_objects = ((Combination)left).getLinguisticObjects();
            if (combi_objects.size() == 2 && combi_objects.elementAt(0) instanceof Variable) {
              left = new Constant( ((Variable)combi_objects.elementAt(0)).getName(),
                                   ((LinguisticObject)combi_objects.elementAt(0)).getStartPosition(),
                                   ((LinguisticObject)combi_objects.elementAt(0)).getEndPosition(),
                                   false);
              left.setRepresentation(total_string.substring(
                              ((LinguisticObject)combi_objects.elementAt(0)).getStartPosition(),
                              ((LinguisticObject)combi_objects.elementAt(0)).getEndPosition()));
              //added for views
              ((Constant)left).addParameter((LinguisticObject)combi_objects.elementAt(1));
            }
          } else
          if ( var_in_cont.size() == 2 ) {
            //definition of constant with 1 or 2 parameters and no parantheses
            Vector combi_objects = ((Combination)left).getLinguisticObjects();
            if (combi_objects.size() == 3 && combi_objects.elementAt(1) instanceof Variable) {
              left = new Constant( ((Variable)combi_objects.elementAt(1)).getName(),
                                   ((LinguisticObject)combi_objects.elementAt(1)).getStartPosition(),
                                   ((LinguisticObject)combi_objects.elementAt(1)).getEndPosition(),
                                   false);
              left.setRepresentation(total_string.substring(
                              ((LinguisticObject)combi_objects.elementAt(1)).getStartPosition(),
                              ((LinguisticObject)combi_objects.elementAt(1)).getEndPosition()));
              //added for views
              ((Constant)left).addParameter((LinguisticObject)combi_objects.elementAt(0));
              ((Constant)left).addParameter((LinguisticObject)combi_objects.elementAt(2));
            }
          } //else <skip> let the syntax check catch it
        }// end adding for defining without parentheses.
        //added for autocompletion of definitions.
        if ( left instanceof Constant ) {
          if ( ((Constant)left).getArity() < var_in_cont.size() ) {
            //if insufficient parameters given, add the first parameters
            Variable para_obj;
            for (int i = ( var_in_cont.size() - ((Constant)left).getArity() ) -1; i >= 0 ; i--) {
              para_obj = new Variable((String)var_in_cont.elementAt(i),0,0);
              para_obj.setRepresentation ("");
              ((Constant)left).addParameter(para_obj,0);
            }
          }
        }
        //end adding for autocompletion of definitions.
        tree = new Definition(tree_ident.getIdentifier(),
                              left,
                              right,
                              tree_ident.getStartPosition(),
                              tree_ident.getEndPosition());
        tree.setRepresentation (total_string.substring(tree_ident.getStartPosition(),
                                                       tree_ident.getEndPosition()));
        //added for views
        answer = expectTree(tree);//recursion (expect a context or line identifier)
      } else
      if (tree_name.equals(",")) {//context
        tree = new Context(tree_ident.getIdentifier(),
                           left,
                           right,
                           tree_ident.getStartPosition(),
                           tree_ident.getEndPosition());
        tree.setRepresentation (total_string.substring(tree_ident.getStartPosition(),
                                                       tree_ident.getEndPosition()));
        //added for views
        answer = expectTree(tree);//recursion (expect a context or line identifier)
      } else
      if (tree_name.equals("|>")) {
        tree = new Line(tree_ident.getIdentifier(),
                        left,
                        right,
                        tree_ident.getStartPosition(),
                        tree_ident.getEndPosition());
        tree.setRepresentation (total_string.substring(tree_ident.getStartPosition(),
                                                       tree_ident.getEndPosition()));
        //added for views
        answer = expectTree(tree);//recursion (expect end of line)
      } else
      if (tree_name.equals(";")) {//added for multiple declaration
        tree = multiDeclaration(tmp_object,left,tree_ident);
        answer = expectTree(tree);
      }
    } else
      throw new IncorrectInput (
        "Parse error: ':', ':=', ',', '|>' or 'end of line' identifier expected"   ,
        ident_info.getStartPosition(),
        ident_info.getEndPosition()
      );
  }
  return answer;
}
```

```
private LinguisticObject parseRightHandSide (LinguisticObject left) throws IncorrectInput {
  //pre : given linguistic object is left hand side od declaration or definition or statement
  //post: declaration or definition object constructed and returned
  //      or statement returned
  LinguisticObject answer;
  answer = null;
  if ( ident_info .getType ().equals ("end of line" ) ||
       ident_info .getIdentifier ().equals (",") || ident_info .getIdentifier ().equals ("|>") ) {
    answer = left;//only one linguistic object left or only one
                  //linguistic object between two context/line identifiers.
                  //Statements (right hand side of context or line)
  } else {
    //pars until context/line identifier or end of line identifier
    String tree_name ;
    int tree_start_position ;
    int tree_end_position ;
    LinguisticObject tree;
    LinguisticObject right;
    if ( ident_info .getIdentifier ().equals (":") || ident_info .getIdentifier ().equals (":=") ) {
      //declaration or definition (right hand side of context or line)
      tree_name = ident_info .getIdentifier ();
      tree_start_position = ident_info .getStartPosition ();
      tree_end_position = ident_info .getEndPosition ();
      nextIdentifier ();
      if ( ident_info .getType ().equals ("name") || ident_info .getIdentifier ().equals ("(")
        || ident_info .getIdentifier ().equals ("SET") || ident_info .getIdentifier ().equals ("PROP") )
        right = betweenPredefined ();//next linguistic object should be no tree
      else
        throw new IncorrectInput (
          "Parse error: Right hand side of '" + tree_name +
          "' identifier may not be a Predefined Identifier." ,
          ident_info .getStartPosition (),ident_info .getEndPosition ()
        );
      if (tree_name .equals (":")) {//declaration
        tree = new Declaration (tree_name , left, right, tree_start_position , tree_end_position );
        tree .setRepresentation (total_string .substring (tree_start_position ,
                                                          tree_end_position ));
        //added for views
        if (left instanceof Variable)//remember the variables to autocomplete definitions
          var_in_cont .addElement (((Variable)left).getName ());
        else//error, let the syntax check catch it
          var_in_cont .addElement (left.toString ());
        answer = tree;
      } else
      if (tree_name .equals (":=")) {//definition
        //added for definitions with no parentheses.
        if (left instanceof Variable && var_in_cont .size () == 0) {
          //definition of constant with no parameters.
          Variable oldLeft = (Variable)left;
          left = new Constant (oldLeft .getName (),
                               oldLeft .getStartPosition (),
                               oldLeft .getEndPosition (),
                               false);
          left.setRepresentation (total_string .substring (oldLeft .getStartPosition (),
                                                            oldLeft .getEndPosition ()));
          //added for views
        } else
        //added for unary and infix definitions
        if ( left instanceof Combination ) {
          if ( var_in_cont .size() == 1 ) {//definition of unary constant
            Vector combi_objects = ((Combination )left).getLinguisticObjects ();
            if (combi_objects .size() == 2 && combi_objects .elementAt (0) instanceof Variable ) {
              left = new Constant ( ((Variable )combi_objects .elementAt (0)).getName (),
                                    ((LinguisticObject )combi_objects .elementAt (0)).getStartPosition (),
                                    ((LinguisticObject )combi_objects .elementAt (0)).getEndPosition (),
                                    false);
              left.setRepresentation (total_string .substring (
                                ((LinguisticObject )combi_objects .elementAt (0)).getStartPosition (),
                                ((LinguisticObject )combi_objects .elementAt (0)).getEndPosition ()));
              //added for views
              ((Constant )left).addParameter ((LinguisticObject )combi_objects .elementAt (1));
            }
          } else
          if ( var_in_cont .size() == 2 ) {
            //definition of constant with 1 or 2 parameters and no parantheses
            Vector combi_objects = ((Combination )left).getLinguisticObjects ();
            if (combi_objects .size() == 3 && combi_objects .elementAt (1) instanceof Variable ) {
              left = new Constant ( ((Variable )combi_objects .elementAt (1)).getName (),
                                    ((LinguisticObject )combi_objects .elementAt (1)).getStartPosition (),
                                    ((LinguisticObject )combi_objects .elementAt (1)).getEndPosition (),
                                    false);
              left.setRepresentation (total_string .substring (
                                ((LinguisticObject )combi_objects .elementAt (1)).getStartPosition (),
                                ((LinguisticObject )combi_objects .elementAt (1)).getEndPosition ()));
              //added for views
              ((Constant )left).addParameter ((LinguisticObject )combi_objects .elementAt (0));
              ((Constant )left).addParameter ((LinguisticObject )combi_objects .elementAt (2));
            }
          } //else <skip> let the syntax check catch it
        }// end adding for defining without parentheses.
        //added for autocompletion of definitions.
        if ( left instanceof Constant ) {
          if ( ((Constant )left).getArity () < var_in_cont .size() ) {
            //if insufficient parameters given, add the first parameters
            Variable para_obj ;
            for (int i = ( var_in_cont .size() - ((Constant )left).getArity () ) -1; i >= 0 ; i--) {
              para_obj = new Variable ((String )var_in_cont .elementAt (i),0,0);
              para_obj .setRepresentation ("");
              ((Constant )left).addParameter (para_obj ,0);
            }
          }
        }
        //end adding for autocompletion of definitions.
        tree = new Definition (tree_name , left, right, tree_start_position , tree_end_position );
        tree.setRepresentation (total_string .substring (tree_start_position , tree_end_position ));
        //added for views
        answer = tree;
      }
    } else
      throw new IncorrectInput (
        "Parse error: ':', ':=' or 'end of line' identifier expected"   ,
        ident_info .getStartPosition (),
        ident_info .getEndPosition ()
      );
  }
  return answer;
}
```

```
private void nextIdentifier () {
  ident_info = Parser.ParsIdentifier (total_string ,ident_info .getEndPosition ());
  if (ident_info .getType ().equals ("name"))
    additional_linguistic_info  = getAdditionalLinguisticInformation (ident_info .getIdentifier ());
  else
    additional_linguistic_info  = null;
}

private ALI getAdditionalLinguisticInformation  (String ident) {
  //return ALI object if there is a ALI object with 'ident' as name
  ALI answer = null;
  for (int i = 0; i < EG.size(); i++) {
    if ( ident.equals( ((ELI)EG.elementAt (i)).getName () ) ) {
      if (EG.elementAt (i) instanceof ALI) {
        answer = (ALI)EG.elementAt (i);
      }
    }
  }
  return answer;
}

private LinguisticObject betweenPredefined () throws IncorrectInput  {
  //pre : identifier is no predefined identifier, except PROP or SET
  //post: construct one linguistic object of all identifiers between
  //      two predefined identifiers, PROP and SET are added in
  //      that one linguistc object
  LinguisticObject  answer;
  answer = null;
  answer = oneLinguisticObjectBetweenPredefined  ();//first Linguistic Object
  //more objects, case 1) infix  case 2) combination object
  //only construct a combination if it is sure that it is not infix
  LinguisticObject  tmp_object  = null;
  IdentifierInformation  name_identifier ;
  if ( ident_info .getType ().equals ("name") ) {
    if (additional_linguistic_info  != null) {//can be no parentheses
      if (additional_linguistic_info  .getArity () == 2) {//possible infix
        name_identifier  = ident_info ;
        int priority = additional_linguistic_info  .getPriority ();
        int associativity  = additional_linguistic_info  .getAssociativity ();
        nextIdentifier ();
        if ( ident_info .getIdentifier ().equals ("(") ) {
          //case 1 parentheses overruling priority
          //case 2 infix constant written as prefix
          tmp_object  = parentheses ();
          if (tmp_object  == null) {//parentheses encountered a ',' therefore parameter list
            ident_info  = name_identifier ;//restore and pars constant
            nextIdentifier ();
            tmp_object  = answer;
            answer = new Combination ();//construct the combination
            ((Combination )answer).addLinguisticObject  (tmp_object );//add first element
            tmp_object  = constant (name_identifier );//the second is a constant
            //infix written as prefix gets parentheses for correct representation
            ((Constant )tmp_object ).setParentheses  (" ( "," ) ");
            ((Combination )answer).addLinguisticObject  (tmp_object );//add second element
          } else {
```

```
          //it is infix
          Constant infix_const = new Constant(name_identifier.getIdentifier(),
                                              name_identifier.getStartPosition(),
                                              name_identifier.getEndPosition(),
                                              false);
          infix_const.setRepresentation(total_string.substring(name_identifier.getStartPosition(),
                                                               name_identifier.getEndPosition()));
          //added for views
          answer = constructInfix(answer, infix_const, tmp_object, priority, associativity);
          if ( ident_info.getType().equals("name") || ident_info.getIdentifier().equals("(")
            || ident_info.getIdentifier().equals("SET") || ident_info.getIdentifier().equals("PROP")
            //still no end, therefore combination
            tmp_object = answer;
            answer = new Combination();
            ((Combination)answer).addLinguisticObject(tmp_object);
          }
        }
      } else
      if ( ident_info.getType().equals("name") ) {
        //it is infix
        tmp_object = oneLinguisticObjectBetweenPredefined ();
        Constant infix_const = new Constant(name_identifier.getIdentifier(),
                                            name_identifier.getStartPosition(),
                                            name_identifier.getEndPosition(),
                                            false);
        infix_const.setRepresentation(total_string.substring(name_identifier.getStartPosition(),
                                                             name_identifier.getEndPosition()));
        //added for views
        answer = constructInfix(answer, infix_const, tmp_object, priority, associativity);

        if ( ident_info.getType().equals("name") || ident_info.getIdentifier().equals("(")
          || ident_info.getIdentifier().equals("SET") || ident_info.getIdentifier().equals("PROP") )
          //still no end, therefore combination
          tmp_object = answer;
          answer = new Combination();
          ((Combination)answer).addLinguisticObject(tmp_object);
        }
      } else //no valid rightside for this infix constant
        throw new IncorrectInput (
          "Parse error: Expecting right hand side for infix constant!"  ,
          ident_info.getStartPosition(),
          ident_info.getEndPosition()
        );
    } else {// it is a combination
      tmp_object = answer;
      answer = new Combination();
      ((Combination)answer).addLinguisticObject(tmp_object);
    }
  } else {// it is a combination
    tmp_object = answer;
    answer = new Combination();
    ((Combination)answer).addLinguisticObject(tmp_object);
  }
} else
if (ident_info.getIdentifier().equals("SET") || ident_info.getIdentifier().equals("PROP") ||
    ident_info.getIdentifier().equals("(")) {
    tmp_object = answer;
    answer = new Combination();
    ((Combination)answer).addLinguisticObject(tmp_object);
}
```

```
    if ( ident_info .getType ().equals ("name" ) || ident_info .getIdentifier ().equals ("(")
      || ident_info .getIdentifier ().equals ("SET") || ident_info .getIdentifier ().equals ("PROP") ) {
      //if no predefined symbol is reached, combination has been constructed and will be filled
      while ( ident_info .getType ().equals ("name") || ident_info .getIdentifier ().equals ("(")
           || ident_info .getIdentifier ().equals ("SET") || ident_info .getIdentifier ().equals ("PROP") ) {
        if (ident_info .getIdentifier ().equals ("SET")) {//construct SetSort
          tmp_object  = new SetSort (ident_info .getIdentifier (),
                                     ident_info .getStartPosition (),
                                     ident_info .getEndPosition ());
          tmp_object .setRepresentation (total_string .substring (ident_info .getStartPosition (),
                                                                  ident_info .getEndPosition ()));

          //added for views
          nextIdentifier ();
        } else
        if (ident_info .getIdentifier ().equals ("PROP")) {//construct PropSort
          tmp_object  = new PropSort (ident_info .getIdentifier (),
                                      ident_info .getStartPosition (),
                                      ident_info .getEndPosition ());
          tmp_object .setRepresentation (total_string .substring (ident_info .getStartPosition (),
                                                                  ident_info .getEndPosition ()));

          //added for views
          nextIdentifier ();
        } else
        if (ident_info .getType ().equals ("name"))  {
          name_identifier  = ident_info ;//remember the name
          if (additional_linguistic_info  != null) {
            if (additional_linguistic_info .getArity () == 2) {//left parameter is last object added
              int priority = additional_linguistic_info .getPriority ();
              int associativity  = additional_linguistic_info .getAssociativity ();
              nextIdentifier ();
              if ( ident_info .getIdentifier ().equals ("(") ) {
                //case 1 parentheses overruling priority
                //case 2 infix constant written as prefix
                tmp_object  = parentheses ();
                if (tmp_object  == null) {//parentheses encountered a ',' therefore parameter list
                  ident_info  = name_identifier ;//restore and pars constant
                  nextIdentifier ();
                  tmp_object  = constant (name_identifier );//the name is a constant
                  //infix written as prefix gets parentheses for correct representation
                  ((Constant )tmp_object ).setParentheses (" ( "," ) ");
                } else {//it is infix
                  Constant  infix_const  = new Constant (name_identifier .getIdentifier (),
                                                         name_identifier .getStartPosition (),
                                                         name_identifier .getEndPosition (),
                                                         false);
                  infix_const .setRepresentation (total_string .substring (
                                            name_identifier .getStartPosition (),
                                            name_identifier .getEndPosition ()));
                  //added for views
                  tmp_object  = constructInfix (((Combination )answer ).getAndRemoveLastLinguisticObject  (),
                                                infix_const ,
                                                tmp_object ,
                                                priority,
                                                associativity );

                }
              } else
            if ( ident_info .getType ().equals ("name" ) ) {
              //infix for sure
              tmp_object = oneLinguisticObjectBetweenPredefined ();
              Constant infix_const = new Constant (name_identifier .getIdentifier (),
                                                   name_identifier .getStartPosition (),
                                                   name_identifier .getEndPosition (),
                                                   false);
              infix_const .setRepresentation (total_string .substring (
                                         name_identifier .getStartPosition (),
                                         name_identifier .getEndPosition ()));
              //added for views
              tmp_object = constructInfix (((Combination )answer ).getAndRemoveLastLinguisticObject (),
                                            infix_const ,
                                            tmp_object ,
                                            priority ,
                                            associativity );
            } else //no valid rightside for this infix constant
              throw new IncorrectInput (
                "Parse error: Expecting right hand side for infix constant!" ,
                ident_info .getStartPosition (),
                ident_info .getEndPosition ()
              );
          } else
          if (additional_linguistic_info .getArity () == 1) {//unary constant
            tmp_object = new Constant (name_identifier .getIdentifier (),
                                       name_identifier .getStartPosition (),
                                       name_identifier .getEndPosition (),
                                       false);
            tmp_object .setRepresentation (total_string .substring (
                                     name_identifier .getStartPosition (),
                                     name_identifier .getEndPosition ()));
            //added for views
            //add next as parameter
            nextIdentifier ();
            if ( ident_info .getType ().equals ("name" ) || ident_info .getIdentifier ().equals ("(")
              || ident_info .getIdentifier ().equals ("SET")
              || ident_info .getIdentifier ().equals ("PROP" ) ) {
              //parentheses and parameterlist are the same
              LinguisticObject parameter = oneLinguisticObjectBetweenPredefined ();
              ((Constant )tmp_object ).addParameter (parameter );
            } else
              throw new IncorrectInput (
                "Parse error: Expecting parameter for the unary constant!" ,
                ident_info .getStartPosition (),
                ident_info .getEndPosition ()
              );
          } else
          if (additional_linguistic_info .getArity () == 0) {
            tmp_object = new Constant (name_identifier .getIdentifier (),
                                       name_identifier .getStartPosition (),
                                       name_identifier .getEndPosition (),
                                       false);
            tmp_object .setRepresentation (total_string .substring (
                                     name_identifier .getStartPosition (),
                                     name_identifier .getEndPosition ()));

            //added for views
            nextIdentifier ();
          }
          } else {//number with no parentheses and no ALI, prefix constant, binder or variable
            nextIdentifier ();
            if ( (int)name_identifier .getIdentifier ().charAt(0) >= 48 &&
                 (int)name_identifier .getIdentifier ().charAt(0) <= 57) {//name is a number
              tmp_object  = new Constant (name_identifier .getIdentifier (),
                                          name_identifier .getStartPosition (),
                                          name_identifier .getEndPosition (),
                                          false);
              tmp_object .setRepresentation (total_string .substring (
                                      name_identifier .getStartPosition (),
                                      name_identifier .getEndPosition ()));
              //added for views
            } else
            if ( ident_info .getIdentifier ().equals ("[")  )//name is a binder name
              tmp_object  = binder (name_identifier );
            else
            if ( ident_info .getIdentifier ().equals ("(")  )//name is a constant name
              tmp_object  = constant (name_identifier );
            else
              tmp_object  = variable (name_identifier );//name is a variable name
          }
        } else {//if (ident_info.getIdentifier().equals("("))
          tmp_object  = parentheses ();
          if (tmp_object  == null)
            throw new IncorrectInput (
              "Parse error: Expecting closing parentheses!"  ,
              ident_info .getStartPosition (),
              ident_info .getEndPosition ()
            );
        }
        ((Combination )answer ).addLinguisticObject (tmp_object );
      }//predefined identifer reached, combination object created
    }//predefined identifier reached, no combination
    return answer ;
  }

  private LinguisticObject  oneLinguisticObjectBetweenPredefined  () throws IncorrectInput {
    //pre : identifier is no predefined identifier, except PROP or SET
    //post: construct the next linguistic object
    //      called by the betweenPredefined function, used for constructing infix constants
    LinguisticObject  answer ;
    answer  = null;
    if (ident_info .getIdentifier ().equals ("SET")) {//construct SetSort
      answer  = new SetSort (ident_info .getIdentifier (),
                            ident_info .getStartPosition (),
                            ident_info .getEndPosition ());
      answer .setRepresentation (total_string .substring (ident_info .getStartPosition (),
                                                         ident_info .getEndPosition ()));
      //added for views
      nextIdentifier ();
    } else
    if (ident_info .getIdentifier ().equals ("PROP")) {//construct PropSort
      answer  = new PropSort (ident_info .getIdentifier (),
                             ident_info .getStartPosition (),
                             ident_info .getEndPosition ());
      answer .setRepresentation (total_string .substring (ident_info .getStartPosition (),
                                                         ident_info .getEndPosition ()));
      //added for views
      nextIdentifier ();
    } else
  if (ident_info .getType ().equals ("name")) {
    IdentifierInformation  name_identifier  = ident_info ;//remember the name
    if (additional_linguistic_info  != null) {
      if (additional_linguistic_info .getArity () == 1) {//unary constant
        answer  = new Constant (name_identifier .getIdentifier (),
                               name_identifier .getStartPosition (),
                               name_identifier .getEndPosition (),
                               false);
        answer .setRepresentation (total_string .substring (name_identifier .getStartPosition (),
                                                            name_identifier .getEndPosition ()));
        //added for views
        //add next as parameter
        nextIdentifier ();
        if ( ident_info .getType ().equals ("name" ) ||  ident_info .getIdentifier ().equals ("(")
          || ident_info .getIdentifier ().equals ("SET")
          || ident_info .getIdentifier ().equals ("PROP" ) ) {
          //parentheses and parameterlist are the same
          LinguisticObject  parameter  = oneLinguisticObjectBetweenPredefined  ();
          ((Constant )answer ).addParameter (parameter );
        } else
          throw new IncorrectInput (
            "Parse error: Expecting parameter for the unary constant!"   ,
            ident_info .getStartPosition (),
            ident_info .getEndPosition ()
          );
      } else
      if (additional_linguistic_info  .getArity () == 0) {
        answer  = new Constant (name_identifier .getIdentifier (),
                               name_identifier .getStartPosition (),
                               name_identifier .getEndPosition (),
                               false);
        answer .setRepresentation (total_string .substring (name_identifier .getStartPosition (),
                                                            name_identifier .getEndPosition ()));
        //added for views
        nextIdentifier ();
      } else { //did not expect infix, must be written in prefix
        answer  = constant (name_identifier );
        //infix written as prefix gets parentheses for correct representation
        ((Constant )answer ).setParentheses (" ( "," ) ");
      }
    } else {//number with no parentheses and no ALI, prefix constant, binder or variable
      nextIdentifier ();
      if ( (int)name_identifier .getIdentifier ().charAt (0) >= 48 &&
           (int)name_identifier .getIdentifier ().charAt (0) <= 57) {//name is a number
        answer  = new Constant (name_identifier .getIdentifier (),
                               name_identifier .getStartPosition (),
                               name_identifier .getEndPosition (),
                               false);
        answer .setRepresentation (total_string .substring (
                                         name_identifier .getStartPosition (),
                                         name_identifier .getEndPosition ()));
        //added for views
      } else
      if ( ident_info .getIdentifier ().equals ("[") )//name is a binder name
        answer  = binder (name_identifier );
      else
      if ( ident_info .getIdentifier ().equals ("(") )//name is a constant name
        answer  = constant (name_identifier );
      else
        answer  = variable (name_identifier );//name is a variable name
    }
  } else
```

```
    if (ident_info.getIdentifier().equals("(")) {
      answer = parentheses();
      if (answer == null)
        throw new IncorrectInput (
          "Parse error: Expecting closing parentheses!" ,
          ident_info.getStartPosition(),
          ident_info.getEndPosition()
        );
    } else throw new IncorrectInput (
        "Parse error: identifier representing a name expected, but " +
        ident_info.getIdentifier() + "found!",
        ident_info.getStartPosition(),
        ident_info.getEndPosition()
      );
    return answer;
  }

  private LinguisticObject variable(IdentifierInformation name_identifier) {
    //pre : name_identifier contains the name of the variable
    //post: construct the variable object
    Variable answer;
    answer = new Variable(name_identifier.getIdentifier(),
                          name_identifier.getStartPosition(),
                          name_identifier.getEndPosition());
    answer.setRepresentation(total_string.substring(name_identifier.getStartPosition(),
                                                    name_identifier.getEndPosition()));
    //added for views
    return answer;
  }

  private LinguisticObject constant(IdentifierInformation name_identifier) throws IncorrectInput {
    //pre : name_identifier contains the name of the constant, it is written as prefix
    //post: construct the prefix constant object
    LinguisticObject tmp_object;
    LinguisticObject answer;
    answer = new Constant(name_identifier.getIdentifier(),
                          name_identifier.getStartPosition(),
                          name_identifier.getEndPosition());
    answer.setRepresentation(total_string.substring(name_identifier.getStartPosition(),
                                                    name_identifier.getEndPosition()));
    //added for views
    if ( ident_info.getIdentifier().equals("(") ) {
      nextIdentifier();
      while ( ident_info.getType().equals("name") || ident_info.getIdentifier().equals("(")
           || ident_info.getIdentifier().equals("SET")
           || ident_info.getIdentifier().equals("PROP") ) {
        tmp_object = betweenPredefined();//parse parameter
        if (ident_info.getIdentifier().equals(",")) {//continue while ','
          nextIdentifier();
          if (ident_info.getIdentifier().equals(")")) //forbid ',)'
            throw new IncorrectInput (
              "Parse error: Expecting parameter, but found ')' identifier" ,
              ident_info.getStartPosition(),
              ident_info.getEndPosition() );
        } else
        if (! ident_info.getIdentifier().equals(")"))//parantheses must be closed
          throw new IncorrectInput (
            "Parse error: ')' identifier expected" ,
            ident_info.getStartPosition(),
            ident_info.getEndPosition() );
        ((Constant)answer).addParameter(tmp_object);
      }
      if (ident_info .getIdentifier ().equals (")")) {
        nextIdentifier ();
      } else
        if (ident_info .getType ().equals ("end of line" )) {
          throw new IncorrectInput (
            "Parse error: Could not parse a correct constant, input not complete"   ,
            ident_info .getStartPosition (),
            ident_info .getEndPosition ()
          );
        } else
          throw new IncorrectInput (
            "Parse error: Could not parse a correct constant, wrong identifier encountered "    +
            "'" + ident_info .getIdentifier () + "'",
            ident_info .getStartPosition (),
            ident_info .getEndPosition () );
    } else
      throw new IncorrectInput (
        "Parse error: '(' identifier expected "  ,
        ident_info .getStartPosition (),
        ident_info .getEndPosition ()
      );
      //is already checked in betweenPredefined function()
    return answer;
  }

  private LinguisticObject  constructInfix (LinguisticObject  left,
                                            Constant  infix_const ,
                                            LinguisticObject  right,
                                            int prio_one ,
                                            int asso_one ) throws IncorrectInput {
    //PRE: left = first parameter, infix_const = infix constant, right = second parameter
    //     priority = priority of infix constant, associativity = associatvity of infix constant
    //POST:recursive, first build all infix to right if present,
    //     then add this infix constant on the right place
    //     return the constructed infix const.
    LinguisticObject  answer;
    answer = null;
    LinguisticObject  tmp_object  = null;
    IdentifierInformation  name_identifier  = null;
    infix_const .addParameter (left);
    //check if the next identifier is an infix constant
    if ( ident_info .getType ().equals ("name") ) {
      if (additional_linguistic_info  != null) {//can be no parentheses
        if (additional_linguistic_info .getArity () == 2) {//possible infix
          name_identifier  = ident_info ;
          int prio_two  = additional_linguistic_info .getPriority ();
          int asso_two  = additional_linguistic_info .getAssociativity ();
          nextIdentifier ();
          if ( ident_info .getIdentifier ().equals ("(") ) {
            //case 1 parentheses overruling priority
            //case 2 infix constant written as prefix => end of recursion
            tmp_object  = parentheses ();
            if (tmp_object  == null) {//parentheses encountered a ',' therefore parameter list
              ident_info  = name_identifier ;//restore
              infix_const .addParameter (right);
              answer = infix_const ;
            } else {//it is infix
        if (prio_one > prio_two) {
          //right is second parameter of infix_const
          infix_const.addParameter(right);
          //recursion for second infix const
          Constant new_const = new Constant(name_identifier.getIdentifier(),
                                            name_identifier.getStartPosition(),
                                            name_identifier.getEndPosition(),
                                            false);
          new_const.setRepresentation(total_string.substring(name_identifier.getStartPosition(),
                                                             name_identifier.getEndPosition()));
          //added for views
          answer = constructInfix(infix_const,new_const,tmp_object,prio_two,asso_two);
        } else
        if (prio_one < prio_two) {
          //first construct the right hand side
          Constant new_const = new Constant(name_identifier.getIdentifier(),
                                            name_identifier.getStartPosition(),
                                            name_identifier.getEndPosition(),
                                            false);
          new_const.setRepresentation(total_string.substring(name_identifier.getStartPosition(),
                                                             name_identifier.getEndPosition()));
          //added for views
          tmp_object = constructInfix(right,new_const,tmp_object,prio_two,asso_two);
          //returned is the infix constant of the right hand side, now add infix_const
          answer = addInfixConst(infix_const,tmp_object,prio_one,asso_one);
        } else //priorities are equal
        if (asso_one == ALI.LEFTASSOCIATIVE) {
          //right is second parameter of infix_const
          infix_const.addParameter(right);
          //recursion for second infix const
          Constant new_const = new Constant(name_identifier.getIdentifier(),
                                            name_identifier.getStartPosition(),
                                            name_identifier.getEndPosition(),
                                            false);
          new_const.setRepresentation(total_string.substring(name_identifier.getStartPosition(),
                                                             name_identifier.getEndPosition()));
          //added for views
          answer = constructInfix(infix_const,new_const,tmp_object,prio_two,asso_two);
        } else {//priorities are equal and right associative
          //first construct the right hand side
          Constant new_const = new Constant(name_identifier.getIdentifier(),
                                            name_identifier.getStartPosition(),
                                            name_identifier.getEndPosition(),
                                            false);
          new_const.setRepresentation(total_string.substring(name_identifier.getStartPosition(),
                                                             name_identifier.getEndPosition()));
          //added for views
          tmp_object = constructInfix(right,new_const,tmp_object,prio_two,asso_two);
          //returned is the infix constant of the right hand side, now add infix_const
          answer = addInfixConst(infix_const,tmp_object,prio_one,asso_one);
        }
      }
    } else
    if ( ident_info.getType().equals("name") ) {
      //it is infix => recursion
      tmp_object = oneLinguisticObjectBetweenPredefined ();
```

```
        if (prio_one > prio_two) {
          //right is second parameter of infix_const
          infix_const.addParameter(right);
          //recursion for second infix const
          Constant new_const = new Constant(name_identifier.getIdentifier(),
                                            name_identifier.getStartPosition(),
                                            name_identifier.getEndPosition(),
                                            false);
          new_const.setRepresentation(total_string.substring(name_identifier.getStartPosition(),
                                                          name_identifier.getEndPosition()));
          //added for views
          answer = constructInfix(infix_const,new_const,tmp_object,prio_two,asso_two);
        } else
        if (prio_one < prio_two) {
          //first construct the right hand side
          Constant new_const = new Constant(name_identifier.getIdentifier(),
                                            name_identifier.getStartPosition(),
                                            name_identifier.getEndPosition(),
                                            false);
          new_const.setRepresentation(total_string.substring(name_identifier.getStartPosition(),
                                                          name_identifier.getEndPosition()));
          //added for views
          tmp_object = constructInfix(right,new_const,tmp_object,prio_two,asso_two);
          //returned is the infix constant of the right hand side, now add infix_const
          answer = addInfixConst(infix_const,tmp_object,prio_one,asso_one);
        } else //priorities are equal
        if (asso_one == ALI.LEFTASSOCIATIVE) {
          //right is second parameter of infix_const
          infix_const.addParameter(right);
          //recursion for second infix const
          Constant new_const = new Constant(name_identifier.getIdentifier(),
                                            name_identifier.getStartPosition(),
                                            name_identifier.getEndPosition(),
                                            false);
          new_const.setRepresentation(total_string.substring(name_identifier.getStartPosition(),
                                                          name_identifier.getEndPosition()));
          //added for views
          answer = constructInfix(infix_const,new_const,tmp_object,prio_two,asso_two);
        } else {//priorities are equal and right associative
          //first construct the right hand side
          Constant new_const = new Constant(name_identifier.getIdentifier(),
                                            name_identifier.getStartPosition(),
                                            name_identifier.getEndPosition(),
                                            false);
          new_const.setRepresentation(total_string.substring(name_identifier.getStartPosition(),
                                                          name_identifier.getEndPosition()));
          //added for views
          tmp_object = constructInfix(right,new_const,tmp_object,prio_two,asso_two);
          //returned is the infix constant of the right hand side, now add infix_const
          answer = addInfixConst(infix_const,tmp_object,prio_one,asso_one);
        }
      } else {//no infix so recursion ends
        infix_const.addParameter(right);
        answer = infix_const;
      }
    } else {//no infix so recursion ends
      infix_const.addParameter(right);
      answer = infix_const;
    }
  } else {//no infix so recursion ends
    infix_const.addParameter(right);
    answer = infix_const; }
    } else {//no infix so recursion ends
      infix_const .addParameter (right);
      answer = infix_const ;
    }
    return answer;
  }

  private Constant addInfixConst (Constant infix_one_and_left ,
                                  LinguisticObject right_hand_side ,
                                  int prio_one ,
                                  int asso_one ) {
    //PRE  prio_one and asso_one are of infix_one,
    //     infix_one must be added into the right_hand_side, then recursion
    //POST return one infix constant where all three objects are combined
    Constant answer = null;
    if (right_hand_side instanceof Constant ) {
      ALI additional_info = getAdditionalLinguisticInformation (((Constant )right_hand_side ).getName ());
      if (additional_info == null || additional_info .getArity () != 2) {
        //the rightside is no infix constant, thus stays intact
        infix_one_and_left .addParameter (right_hand_side );
        answer = infix_one_and_left ;
      } else {
        int prio_two = additional_info .getPriority ();
        int asso_two = additional_info .getAssociativity ();
        if (prio_one > prio_two ) {
          LinguisticObject tmp_object ;
          //the first parameter of the rightside belongs to infix_left.
          //Mayby even stronger, therefore recursion
          Vector store_paras_right = ((Constant )right_hand_side ).getParameters ();
          tmp_object = addInfixConst (infix_one_and_left ,
                                      (LinguisticObject )store_paras_right .elementAt (0),
                                      prio_one ,
                                      asso_one );
          //answer = right_hand_side with a new first parameter
          answer = new Constant (((Constant )right_hand_side ).getName (),
                                 ((Constant )right_hand_side ).getStartPosition (),
                                 ((Constant )right_hand_side ).getEndPosition (),
                                 false);
          answer.setRepresentation (total_string .substring (
                               ((Constant )right_hand_side ).getStartPosition (),
                               ((Constant )right_hand_side ).getEndPosition ()));
          //added for views
          answer.addParameter (tmp_object );
          answer.addParameter ((LinguisticObject )store_paras_right .elementAt (1));
        } else
        if (prio_one < prio_two ) {
          //the top of the rightside is stronger therefore it stays intact
          infix_one_and_left .addParameter (right_hand_side );
          answer = infix_one_and_left ;
        } else //priorities are equal
        if (asso_one == ALI.LEFTASSOCIATIVE ) {
          LinguisticObject tmp_object ;
          //the first parameter of the rightside belongs to infix_left.
          //Mayby even stronger, therefore recursion
          Vector store_paras_right = ((Constant )right_hand_side ).getParameters ();
          tmp_object = addInfixConst (infix_one_and_left ,
                                      (LinguisticObject )store_paras_right .elementAt (0),
                                      prio_one ,
                                      asso_one );
        //answer = right_hand_side with a new first parameter
        answer = new Constant (((Constant )right_hand_side ).getName (),
                               ((Constant )right_hand_side ).getStartPosition (),
                               ((Constant )right_hand_side ).getEndPosition (),
                               false);
        answer.setRepresentation (total_string .substring (
                             ((Constant )right_hand_side ).getStartPosition (),
                             ((Constant )right_hand_side ).getEndPosition ()));
        //added for views
        answer.addParameter (tmp_object );
        answer.addParameter ((LinguisticObject )store_paras_right .elementAt (1));
      } else {//priorities are equal and right associative
        //the top of the rightside is stronger therefore it stays intact
        infix_one_and_left .addParameter (right_hand_side );
        answer = infix_one_and_left ;
      }
    }
  } else {
    //the rightside is no infix constant, thus stays intact
    infix_one_and_left .addParameter (right_hand_side );
    answer = infix_one_and_left ;
  }
  return answer;
}

private LinguisticObject binder(IdentifierInformation name_identifier ) throws IncorrectInput {
  //pre : name_identifier contains the name of the binder
  //post: construct the binder object
  LinguisticObject tmp_object = null;
  LinguisticObject answer;
  LinguisticObject left;
  LinguisticObject right;
  boolean multiple = false;
  answer = new Binder(name_identifier .getIdentifier (),
                      name_identifier .getStartPosition (),
                      name_identifier .getEndPosition ());
  answer.setRepresentation (total_string .substring (name_identifier .getStartPosition (),
                                                      name_identifier .getEndPosition ()));
  //added for views
  if (ident_info .getIdentifier ().equals ("[")) {
    nextIdentifier ();
    if ( ident_info .getType ().equals ("name") || ident_info .getIdentifier ().equals ("(")
      || ident_info .getIdentifier ().equals ("SET") || ident_info .getIdentifier ().equals ("PROP") )
      left = betweenPredefined ();//left hand side of the declaration
    else
      throw new IncorrectInput (
        "Parse error: Inside the '[' and ']' identifiers a declaration is expected"   ,
        ident_info .getStartPosition (),
        ident_info .getEndPosition () );
    if (ident_info .getIdentifier ().equals (":")) {
      tmp_object = parseRightHandSide (left);//parseRightHandSide also extends var_in_cont
      var_in_cont .removeElementAt (var_in_cont .size()-1);
    } else
    if (ident_info .getIdentifier ().equals (";")) {//multiple binder
      //just add the context with multiple declarations as declaration and transform the binder
      //if everything is parsed.
      multiple = true;
      int tmp_size = var_in_cont .size();
      tmp_object = multiDeclaration (left,null,null);//also extends var_in_cont
      int tmp_length = var_in_cont .size() - tmp_size;
      for (int i = 0; i < tmp_length; i++)
        var_in_cont .removeElementAt (tmp_size);
      } else
        throw new IncorrectInput (
          "Parse error: Inside the '[' and ']' identifiers a declaration is expected"   ,
          ident_info .getStartPosition (),
          ident_info .getEndPosition ()
        );
      ((Binder)answer).setDeclaration (tmp_object);//add the constructed declaration
      if ( ident_info .getIdentifier ().equals ("]")) //brackets must be closed
        nextIdentifier ();
      else
        throw new IncorrectInput (
          "Parse error: ']' identifier expected"  ,
          ident_info .getStartPosition (),
          ident_info .getEndPosition ()
        );
      if ( ident_info .getIdentifier ().equals ("(") ) {
        nextIdentifier ();
        while ( ident_info .getType ().equals ("name") || ident_info .getIdentifier ().equals ("(")
             || ident_info .getIdentifier ().equals ("SET")
             || ident_info .getIdentifier ().equals ("PROP") ) {
          tmp_object = betweenPredefined ();//parse parameter
          if (ident_info .getIdentifier ().equals (",")) {//continue while ','
            nextIdentifier ();
            if (ident_info .getIdentifier ().equals (")")) //afvangen ,)
              throw new IncorrectInput (
                "Parse error: Expecting parameter, but found ')' identifier"   ,
                ident_info .getStartPosition (),
                ident_info .getEndPosition ()
              );
          } else
          if (! ident_info .getIdentifier ().equals (")"))//parantheses must be closed
            throw new IncorrectInput (
              "Parse error: ')' identifier expected"  ,
              ident_info .getStartPosition (),
              ident_info .getEndPosition ()
            );
          ((Binder)answer).addParameter (tmp_object);
        }
        if (ident_info .getIdentifier ().equals (")")) {
          nextIdentifier ();
        } else
          if (ident_info .getType ().equals ("end of line" )) {
            throw new IncorrectInput (
              "Parse error: Could not parse a correct binder, input not complete"   ,
              ident_info .getStartPosition (),
              ident_info .getEndPosition ()
            );
          } else
            throw new IncorrectInput (
              "Parse error: Could not parse a correct binder, wrong identifier encountered "    +
              "'" + ident_info .getIdentifier () + "'",
              ident_info .getStartPosition (),
              ident_info .getEndPosition ()
            );
      } else
        throw new IncorrectInput (
          "Parse error: '(' identifier expected "  ,
          ident_info .getStartPosition (),
          ident_info .getEndPosition ()
        );
    } else
```

```
      throw new IncorrectInput (
        "Parse error: Expecting '['" ,
        ident_info .getStartPosition (),
        ident_info .getEndPosition ()
      );
      //is already checked in betweenPredefined function()
    if (multiple )
      answer = multipleBinder ((Binder)answer);
    return answer;
  }

  private LinguisticObject multiDeclaration (LinguisticObject first,
                                             LinguisticObject tree,
                                             IdentifierInformation top_tree) throws IncorrectInput {
    //pars multi declarations, construct the appropriate Context and put it in the tree structure
    int startpos;
    LinguisticObject tmp_object = null;
    LinguisticObject right = null;
    LinguisticObject answer = null;
    IdentifierInformation decl_info = null;
    Vector multi_cont_ident = new Vector();
    Vector multi_vars = new Vector();
    multi_vars .addElement (first);//add the first variable
    if (first instanceof Variable)//remember the variables to autocomplete definitions
      var_in_cont .addElement (((Variable)first).getName ());
        else//error, let the syntax check catch it
      var_in_cont .addElement (first.toString ());
    while (ident_info .getIdentifier ().equals (";")) {
      //remember the identifiers, for the positions, link to the context objects
      multi_cont_ident .addElement (ident_info);
      nextIdentifier ();
      if (ident_info .getType ().equals ("name") || ident_info .getIdentifier ().equals ("(")
       || ident_info .getIdentifier ().equals ("SET") || ident_info .getIdentifier ().equals ("PROP"))
        tmp_object = betweenPredefined ();//get all other variables, until the ':' identifier
      else
        throw new IncorrectInput (
          "Parse error: Expecting expression, but found '"  + ident_info .getIdentifier () + "'",
          ident_info .getStartPosition (),
          ident_info .getEndPosition ()
        );
      multi_vars .addElement (tmp_object);//store them in a list
      if (tmp_object instanceof Variable)//remember the variables to autocomplete definitions
        var_in_cont .addElement (((Variable)tmp_object ).getName ());
          else//error, let the syntax check catch it
        var_in_cont .addElement (tmp_object .toString ());
    }
    if (ident_info .getIdentifier ().equals (":")) {
      decl_info = ident_info;
      nextIdentifier ();
      right = betweenPredefined ();//the right side of the multi declaration
      for (int i = 0; i < multi_vars .size(); i++) {
        tmp_object = new Declaration (decl_info .getIdentifier (),
                                      (LinguisticObject )multi_vars .elementAt (i),
                                      right,
                                      decl_info .getStartPosition (),
                                      decl_info .getEndPosition ());
        tmp_object .setRepresentation (total_string .substring (decl_info .getStartPosition (),
                                                                decl_info .getEndPosition ()));
        //added for views
        if (i == 0) {//the first can be on two places, depending on the multiple declaration cases
          if (tree == null) {//multi declaration in the beginning of the tree
            answer = tmp_object;
          } else {//multi declaration inside the context
            answer = new Context(top_tree.getIdentifier(),
                                 tree,tmp_object,
                                 top_tree.getStartPosition(),
                                 top_tree.getEndPosition());
            answer.setRepresentation(total_string.substring(top_tree.getStartPosition(),
                                                            top_tree.getEndPosition()));

            //added for views
          }
        } else {
          //for all other declarations in the multiple declarations
          answer = new Context(
                        ",", //multi_cont_ident.getIdentifier() = ';'
                        answer,
                        tmp_object,
                        ((IdentifierInformation)multi_cont_ident.elementAt(i-1)).getStartPosition(),
                        ((IdentifierInformation)multi_cont_ident.elementAt(i-1)).getEndPosition() );
          answer.setRepresentation(total_string.substring(
                          ((IdentifierInformation)multi_cont_ident.elementAt(i-1)).getStartPosition(),
                          ((IdentifierInformation)multi_cont_ident.elementAt(i-1)).getEndPosition()));
          //added for views
        }
      }
    } else
      throw new IncorrectInput(
        "Parse error: Expecting ':'" ,
        ident_info.getStartPosition(),
        ident_info.getEndPosition()
      );
    return answer;
  }

  private Binder multipleBinder(Binder total_binder) throws IncorrectInput {
  //PRE : The root_binder has a context with multiple declarations as its local declaration
  //POST: The binder in unsugared form
    Binder answer = null;
    LinguisticObject multi_vars = total_binder.getDeclaration();
    if (multi_vars instanceof Context) {
      LinguisticObject top_decl = ((Context)multi_vars).getRightSide();//get the last declaration
      if (top_decl instanceof Declaration) {
        //construct the last (most inner) binder
        Binder top_bind = new Binder(total_binder.getName(),
                                     total_binder.getStartPosition(),
                                     total_binder.getEndPosition());
        top_bind.setRepresentation(total_string.substring(total_binder.getStartPosition(),
                                                          total_binder.getEndPosition()));

        //added for views
        top_bind.setDeclaration(top_decl);
        //most inner binder has all parameters
        Vector paras = total_binder.getParameters();
        for (int i = 0; i < paras.size(); i++)
          top_bind.addParameter((LinguisticObject)paras.elementAt(i));
        //construct the other binders by recursion (list of declarations is one smaller)
        answer = makeBinders(((Context)multi_vars).getLeftSide(), top_bind);
      } else
      throw new IncorrectInput (
        "Parse error: Expecting declaration!" ,
        top_decl .getStartPosition (),
        top_decl .getEndPosition ()
      );
    } else
      throw new IncorrectInput (
        "Parse error: Expecting multiple declarations!" ,
        multi_vars .getStartPosition (),
        multi_vars .getEndPosition ()
      );
    return answer;
  }

  private Binder makeBinders (LinguisticObject decls, Binder bind) throws IncorrectInput {
    //PRE :
    //POST: binders constructed till no decls left
    Binder answer = null;
    if (decls instanceof Context) {//recursive until no context left
      LinguisticObject top_decl = ((Context)decls).getRightSide ();
      if (top_decl instanceof Declaration) {
        //construct the most inner binder, not yet constructed
        Binder top_bind = new Binder (bind.getName (),bind.getStartPosition (),bind.getEndPosition ());
        top_bind.setRepresentation (total_string .substring (bind.getStartPosition (),
                                                             bind.getEndPosition ()));

        //added for views
        top_bind.setDeclaration (top_decl);
        //first parameter is the previous binder constructed
        top_bind.addParameter (bind);
        //other parameters are the same as the previous binder
        Vector paras = bind.getParameters ();
        for (int i = 1; i < paras.size(); i++)
          top_bind.addParameter ((LinguisticObject )paras.elementAt (i));
        //construct the other binders by recursion (list of declarations is one smaller)
        answer = makeBinders (((Context)decls).getLeftSide (), top_bind);
      } else
      throw new IncorrectInput (
        "Parse error: Expecting declaration!" ,
        top_decl .getStartPosition (),
        top_decl .getEndPosition ()
      );
    } else
    if (decls instanceof Declaration) {//last declaration in the list
      //construct the root binder, top most binder
      Binder root_bind = new Binder (bind.getName (),bind.getStartPosition (),bind.getEndPosition ());
      root_bind .setRepresentation (total_string .substring (bind.getStartPosition (),
                                                             bind.getEndPosition ()));

      //added for views
      root_bind .setDeclaration (decls);
      //first parameter is the previous binder constructed
      root_bind .addParameter (bind);
      //other parameters are the same as the previous binder
      Vector paras = bind.getParameters ();
      for (int i = 1; i < paras.size(); i++)
        root_bind .addParameter ((LinguisticObject )paras.elementAt (i));
      //root binder, drop out of the recursion
      answer = root_bind;
    } else
```

```
      throw new IncorrectInput (
        "Parse error: Expecting multiple declarations!"  ,
        decls.getStartPosition (),
        decls.getEndPosition ()
      );
    return answer;
  }

  private LinguisticObject  parentheses () throws IncorrectInput {
  //PRE :
  //POST: parentheses parsed, representation extended to include parentheses
    LinguisticObject  answer = null;
    String open = "";
    String close = "";
    if (ident_info.getIdentifier ().equals ("(")) {
      open = total_string.substring (ident_info.getStartPosition (),ident_info.getEndPosition ());
      nextIdentifier ();
      if ( ident_info.getType ().equals ("name") || ident_info.getIdentifier ().equals ("(") )
        //we do not permit parentheses around sort objects
        answer = betweenPredefined ();
      else
        throw new IncorrectInput (
          "Parse error: Expecting expression, but found '"  +ident_info.getIdentifier ()+"'",
          ident_info.getStartPosition (),
          ident_info.getEndPosition ()
        );
      if (ident_info.getIdentifier ().equals (")")) {
        close = total_string.substring (ident_info.getStartPosition (),ident_info.getEndPosition ());
        nextIdentifier ();
      } else
        throw new IncorrectInput (
          "Parse error: Expecting ')'"  ,
          ident_info.getStartPosition (),
          ident_info.getEndPosition ()
        );
    }
    if (answer instanceof Atomic)
      ((Atomic)answer).setParentheses (open,close);
    else if (answer instanceof Combination)
      ((Combination)answer).setParentheses (open,close);
    //else it is not a name identifiers, therefore not possible.
    return answer;
  }
}
```