Thinksheet: A Tool for Information Navigation

by

Peter Piatko

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science
New York University
September, 1998

Approved:
1_____

Professor Dennis Shasha
Research Advisor

© Peter Piatko

All Rights Reserved 1998

To my grandmother, Helen.

Acknowledgments

I wish to give my deepest thanks to my advisor and friend, Professor Dennis Shasha. He introduced me to the wonderful world of complex documents, and has given me great advice ever since.

I am especially grateful to Churngwei Chu, for supporting me all these years. She also provided excellent editoral advice for nearly every page of this thesis. I would also like to thank David Tanzer for his role as editor.

Special thanks go to all those who are now or were in the past on the Thinksheet project. This includes, but is not limited to, David Tanzer, Roman Yangarber, Dao-i Lin, and Christopher Jones. Without their help, Thinksheet would never have been fully realized.

I would also like to thank Arash Baratloo and Fangzhe Chang, who volunteered their time to keep the computers in several offices running—including the one on my desk.

Finally I would like to thank my parents, my sister Christine and my brother-in-law Richard. They provided unwavering support for my studies. Christine and Richard, having gone through the same process themselves, were able to provide invaluable advice.

Table of Contents

List of Figures

List of Tables

List of Appendices

Abstract
Thinksheet: A Tool for Information Navigation

Peter Piatko
New York University, 1998
Research Advisor: Professor Dennis Shasha

Imagine that you are a "knowledge worker" in the coming millenium. You must synthesize information and make decisions such as "Which benefits plan to use?" "What do the regulations say about this course of action?" "How does my job fit into the corporate business plan?" or even "How does this program work?" If the dream of digital libraries is to bring you all material relevant to your task, you may find yourself drowning before long. Reading is harder than talking to people who know the relevant documents and can tell you what you're interested in. That is what many current knowledge workers do, giving rise to professions such as insurance consultant, lawyer, benefits specialist, and so on.

Imagine by contrast that the documents you retrieve could be tailored precisely to your needs. That is, imagine that the document might ask you questions and produce a document filtered and organized according to those you have answered.

We have been developing software that allows writers to tailor documents to the specific needs of large groups of readers. Thinksheet combines the technologies of expert systems, spreadsheets, and database query processing to provide tailoring capabilities for complex documents. The authoring model is only slighly more complex than a spreadsheet.

This thesis discusses the conceptual model and the implementation of Thinksheet, and applications for complex documents and metadata.

Chapter 1

Introduction

Consider the following example taken from a recent bill in the United States Congress [Imm97]:

> ... Notwithstanding section 203 of the Immigration and Nationality Act, during the immigration moratorium under section 2, in lieu of the number of visas that may be allotted under section 203 of such Act—
>
> (1) the number of visas that shall be allotted to family-sponsored immigrants under section 203(a) of such Act shall be 10,000 for qualified immigrants under section 203(a)(2)(A) of such Act and zero for other family-sponsored immigrants ...

A reader unfamiliar with the law would have to find the cross references and then make sense of it all. After doing all of this searching, he may realize that this particular section is not even relevant to him, resulting in frustration and wasted time.

Such a law is an example of a *complex* document. Laws, regulations, requirements documents, and insurance plans form a small list of examples. In characterizing complex documents, we can say the following:

1. They are difficult to read because they are structurally complex.
2. Only a small portion, which is usually non-consecutive, applies to a particular reader.

Writers of complex documents do not have it any easier. Organizing, checking for inconsistencies and other factors make writing a complex document a time consuming and difficult task.

We have developed a model for reading and writing complex documents and also implemented a system based on this model. This system is called *Thinksheet*. Thinksheet combines the technologies of expert systems, and database query processing and hypermedia to provide tailoring capabilities for complex documents. Thinksheet helps the reader by tailoring the complex document to his needs. Thinksheet has an

easy authoring model that allows the writer to specify when each portion of the complex document is relevant.

In the next section we describe our motivation and informally describe the Thinksheet model. Section 1.3 discusses related work. Finally, Section 1.4 gives an outline for the rest of the thesis.

1.1 Introduction to the Thinksheet Model

Before we can provide a tool for complex documents, we have to know what one is. What makes a document complex? Research in the hypertext community gives us some idea of what makes a document complex. For example, three rules where hypertext can be useful are given in [Shn89]. They are that:

1. there is a large body of information organized into fragments,
2. the fragments relate to each other, and
3. the user needs only a small fraction at any time.

We may assume that if hypertext is "useful" for a document then the document is probably complex. Therefore the above three items might be considered as rules of thumb for judging complexity. The difficulty a reader faces in reading a complex document is finding the small fraction of document fragments that he needs. So, how does the reader retrieve the right set of fragments?

Following a database model, we can associate a set of attributes with each fragment. Retrieval would be based on a query over the attributes.

For example, suppose we had a museum guidebook. We might associate attributes for each museum. A museum might specify a Price (such has High or Low), a Location, and a Subject (Art, Science, etc.). Then a reader might specify a query to retrieve the relevant museums.

There are two potential problems with this method. First, it is well known that people have trouble specifying boolean queries [WS94]. Second, while this method seems good for a simple guidebook, it does not work well with documents with a more complicated structure, such as laws or regulations. Simply adding a bunch of attribute-value pairs to each chunk of law or regulation loses information.

We propose another method. Instead of putting attributes on each fragment, Thinksheet will put some condition on it which specifies whether it is relevant to the reader.

One way of doing this is by associating a boolean formula with each fragment. If the boolean formula is true, then the fragment is relevant. We will call this boolean formula a *precondition*.

The precondition will contain variables to be filled out by the reader. We will associate a question with each variable to present to the reader. The reader's answer to the question will be the variable's value. For example, for the variable named *Family*,

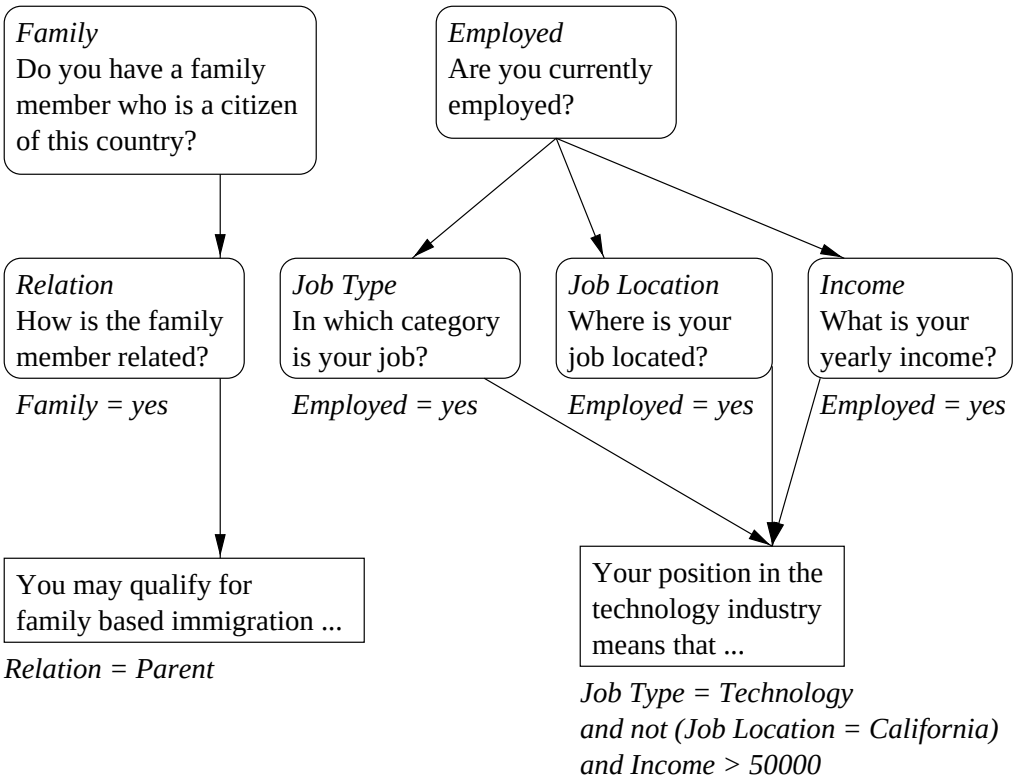


FIGURE 1.1: Immigration Law with variables and preconditions. Rounded edged boxes represent variables. Squared off boxes represent the text of the document. Variable names are the topmost words in italics. Preconditions are in italics below the boxes.

we can associate the question "Do you have a family member who is a citizen?" A particular portion of immigration law might have the precondition *Family* = *yes*. If the reader answered the question with *yes*, that portion's precondition would become true, and is therefore relevant to the reader.

Now the reader does not have to give some sort of boolean query in order to get back the relevant fragments. Instead, he answers some questions—which give variables their values—and those fragments with true preconditions (based on the values of the variables) will be considered relevant.

Boolean preconditions allow us to specify more complicated conditions for relevancy. For example, a particular piece of immigration law might be relevant if the immigrant's income is greater than $50,000 and he is employed in the technology industry but not if that job is in the state of California (where a different portion of the law might apply). This kind of condition would be hard to represent using the relational model without causing an explosion in the number of attributes.

However, the reader has to answer a lot of questions. Some of the questions themselves may not be relevant to the task at hand. To continue with the immigration example, suppose the immigrant is interested in gaining citizenship through his professional employment. He already knows that he can not fill in the variables to gain citizenship through family relations, so variables related to that condition are irrelevant. The potential citizen would prefer not to wade through all these irrelevant variables.

We can add preconditions to variables, just as we have done with fragments. Thus, the reader can fill out only those variables with true preconditions. Those with false preconditions will not be presented to him. For example, in Figure 1.1, the variables *Relation*, *Job Industry*, *Job Location* and *Income* all are dependent on either *Family* or *Employed*. The potential immigrant could simply fill in *Family* with the value *no* and any other variables related to that topic—i.e. variables with the precondition *Family* = *yes*—would disappear.

On the other hand, if the immigrant fills in *Employed* with *yes*, then the preconditions for *Job Industry*, *Job Location*, and *Income* become true. Thus, since those questions are now relevant, he can now answer them. Suppose he subsequently answers *Job Industry* with *Technology*, *Job Location* with *New Jersey* and *Income* with *60000*. Then the portion of text beginning with the phrase, "Your position in the technology industry means that ... ," will be presented to him because that portion of immigration law will be considered relevant.

The reader interaction with the system might be something like this:

1. The reader answers any of the initial questions he desires.

2. Some questions and document chunks disappear because their preconditions become false, while others are highlighted because their preconditions become true. The reader may answer more questions whose preconditions are true. This process continues until he answers all relevant questions or decides to stop.

3. Those document chunks with true precondition formulas will be presented to the reader.

Once the reader is done answering questions, the remaining document chunks represent the fragments of the reader's choice. The document has now been tailored to present only the information that is relevant to the reader's needs.

This collection of questions, document chunks and preconditions is what we call a Thinksheet. The model will be formally explained in Chapter 2, and the working system will be discussed in Chapter 3.

The contributions of this dissertation are:

1. The Thinksheet model, which is a model for complex documents.

2. A implementation based on this model, which we call Thinksheet, which can be used to read and write complex documents.

1.2 Applications

1.2.1 Immigration Law

In the summer of 1996, Emma Dickson created a Thinksheet about United States immigration law which gives advice on how to immigrate into the country based on many different criteria. At the time of the creation of this Thinksheet, Emma Dickson was a college undergraduate majoring in history. Her experience with computers had been mainly with popular business applications, such as word processors and spreadsheets. After some initial tutoring, Ms. Dickson was able to create the Thinksheet on her own. The final Thinksheet is approximately 70 nodes.

This Thinksheet breaks a summarization of the immigration law into many distinct pieces. There is a separate node holding each piece. Whether a particular piece applies to a reader is reflected in the preconditions of the node holding that piece.

The questions posed to the reader initially deal with broad categories. They are:

- Family. This category is of interest to the reader who has a relative who is already a citizen or a permanent resident.

- Employment. This category is of interest to the reader who wishes to immigrate based on his employment.

- Political. This category is of interest to the reader who wishes to be granted political asylum.

- Country of origin. The rules for immigration may also depend on the country of origin.

- Investment. This category is for readers who wish to invest money into U.S. businesses.

Each of these categories starts out with one question. For example, the Family category has the following question:

```
Do you have any relative who lives in
the United States legally?
1. Yes, my relative is a U.S. Citizen.
2. Yes, my relative is a permanent resident.
3. No.
```

If the reader picks choice `1` or `2`, then he will be asked more questions about his relatives in the states (for example, what the relationship of this relative is).

If the reader had chosen `3`, all questions about relatives would disappear because they are now irrelevant.

To make the reading result easier, another node, called *Final Advice*, consolidates the text of all the other "piece" nodes. Because of the way *Final Advice* has been created, if the reader answers none of the questions and selects this node, he will see the entire document about immigration. Answering more questions narrows down the amount of information shown at the end. However, the reader need not answer *all* of the questions presented to him. At any time, he may stop that activity and select *Final Advice*. If he feels that the information has not been narrowed down enough, he may go back and answer more questions.

1.2.2 Social Security Benefits

In the summer of 1995, Christopher Jones, a sophomore undergraduate, created a Thinksheet that calculates disability and survivorship benefits for Social Security in the United States. The entire Thinksheet is approximately 150 nodes, the largest created so far.

As the student worked on this Thinksheet, we were prompted to add new features to the system. Some of these features were additions to the user interface—e.g. making it easier to edit the text of the Thinksheet. Other features included extensions to the Thinksheet language (see Section 3.3.1).

A reader of this Thinksheet must fill in the annual income for all the years he has worked, plus information about his immediate family. However, suppose this information were stored in a database. Upon entering the reader's name, Thinksheet could fill in the relevant information.

To show that this feature could be incorporated into Thinksheet, a database of fictitious people was created, along with their income and family information. If the reader enters in one of those names, then the relevant information is filled in automatically by Thinksheet.

The resultant text is highly tailored to the individual reader. For example, pronouns are calculated based on the answers and dynamically generated. This tailoring is done through the use of the Smartfield directives (see Section 3.3.2 and Appendix C).

1.2.3 Telecommunications Requirements

In the summer of 1996, Peter Piatko worked at Bellcore and used Thinksheet for a project there which deals with the design of new telecommunications services. When designing such services, the designer must take into consideration how the new service may interact with other existing services. This requires knowledge about the requirements (and how implementations may differ from those requirements) of the services and protocols. This information is stored in a large body of requirements documentation.

In general, the designer references only a small portion or a small number of non-consecutive portions of the documentation at any one time, and finding that information can be difficult and time-consuming. Thinksheet is used as a way to guide the designer through one proprietary Bellcore requirements document, titled *SR3803: Interactions between Switch-Based and AIN Features*. Because these requirements are subject to change, the text of the document was stored in a database, which Thinksheet then extracted by constructing the proper SQL queries based on the reader's answers to questions.

Preliminary use of this application has met with promising results. Griffeth, the project director, has the following to say about Thinksheet's role [Gri97]:

> The Thinksheet database has been one of the central software components used in a project that I directed in 1996–1997 at Bellcore. It provided easy and intuitive access to the requirements involved in a particularly difficult phase of the design process for a new telecommunications service. We have found it easy to teach to new users and they find it a useful way to access information.

1.3 Related Work

1.3.1 User Interfaces

There have been a number of user interfaces developed for the purpose of navigating through large information spaces. Most of the interfaces are based on two observations:

1. Users usually are interested in details of a small portion of the information space. This section of detail is called the user's *focus*.

2. Users would like to navigate through the information space without getting "lost." Thus, they would like to know the *context* around the point which they are focusing so as to allow them to navigate to other sections of the information space in a straight-forward manner.

These *focus+context* techniques generally do not show the whole information space at once, but rather give the user details of the focus and enough portions of the rest of the information space to allow him to understand the focal point in the context of the whole.

While this technique exists for the most part for hierarchical data structures, the idea of tailoring the information space by providing details only for the focal point is similar to our idea of tailoring complex documents. However, while these graphical techniques may be used to view the overall structure of a complicated document, they can not store as complicated semantics as Thinksheet's preconditions can.

Example of such focus+context techniques include fisheye views [Fur86], graphical fisheye views [SB92, SSTR93], and Pad and Pad++ [BH94, PF93].

1.3.2 Expert Systems

Expert systems encode the knowledge of experts in a specialized subject with the aim of solving a problem or giving advice [Jac90]. Thinksheet may be thought of as trying to do something similar—we want to use the system to encode an expert's knowledge of a complex document. But Thinksheet takes a simpler approach that avoids some of the common pitfalls of expert systems.

A common problem with expert systems is that a new application of an existing expert system often requires that a whole new expert system be written [HRWL84, Ign91]. The main reason is that there is no easy separation in such a system between (its often very powerful) rules and data. Furthermore, expert systems have the added complexity that they are state-driven—sometimes inputting the same values twice will not result in the same output. They also often depend on non-deterministic techniques such as conflict resolution, making programming such systems difficult. These are reasons they have been criticized in the industrial literature [Mer94].

At this point, Thinksheet shares the rule bias of an expert system without its powerful rule language (e.g. Thinksheets don't allow recursion). In a Thinksheet application, rules are encoded into the precondition and answer formulas. As questions are answered by the user, nodes change their truth values and answers change their values. The net effect is that the set of possible queries is fixed. This works well for many applications, but does not constitute a separable query language (see Chapter 7 for future work on this, however).

1.3.3 Automatic Text Summarization

Automatic text summarization systems summarize documents through automatic or semi-automatic means, allowing the reader to use the summaries as a source of information instead of having to read whole documents. Typically these systems either use a statistical approach to extract sentences from a document, which are then used as a summary, or they use natural language processors to process a document (or group of documents) and then use a language generator to produce a summary [KR96].

These systems and Thinksheet share the common goal of providing only the relevant information to the reader, but summary generators operate on a different class of documents. Generally, they have been used to generate summaries of newspaper or magazine articles [AL97, BE97, MR95]. These papers may have complicated subject matter—i.e. they may be technical articles [TM97])—but they generally do not meet our criteria for complex documents because the structure of the articles is mostly simple and linear. These systems also do not provide the individual tailoring capability of Thinksheet, which we think is crucial for the readers of complex documents.

1.4 Outline of Thesis

The next chapter of this thesis formally describes the Thinksheet model. Chapter 3 describes the interface to the Thinksheet system developed at NYU. Chapter 4 describes the algorithms involved when working with a Thinksheet. Chapter 5 gives an overview of the implementation of Thinksheet. Chapter 6 shows how we can use Thinksheet for another application, metadata. Finally Chapter 7 provides our conclusion and discusses future work.

Chapter 2

Thinksheet Model

This chapter describes the Thinksheet model more formally. This formal model is used in the discussion of the later chapters to prove the correctness of Thinksheet's implementation.

In the following section we give a brief overview of some of the terminology of first order logic systems. In Section 2.2 we formally define a Thinksheet as a collection of nodes and formulas. Section 2.3 deals with evaluating these formulas.

2.1 Terminology of First Order Logic

Thinksheet is based on a three-valued first order logic. As we will be discussing various aspects of these formulas, we introduce some of the terminology of first-order logic syntax [End72].

Definition 2.1 *Some terminology of first-order logic languages.*

1. Predicate *and* function *symbols are symbols in the language. Each symbol has an associated degree indicating the number of arguments to the symbol. Predicate symbols of degree zero are propositional symbols, and function symbols of degree zero are constant symbols.*

2. *A* term *is either a variable v, a constant c, or is a compound term of the form $f(t_1, \ldots, t_n)$ where $t_1, \ldots, tn$ are terms and f is a function symbol of degree n.*

3. *An* atomic formula *(or* atom *for short) is an expression of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol of degree n, and $t_1, \ldots, t_n$ are terms.*

4. *A* molecular formula *is an expression of the form $f_1 \wedge f_2$ (conjunction), $f_1 \vee f_2$ (disjunction), $\neg f$ (negation).*

First order logic languages also contain syntax for quantifiers, but this will not be discussed here.

Predicate symbols are truth value symbols, for example *true*, *false*, or *possible* or comparison operations such as $<$, $>$ and $=$. Function symbols may be, for example, arithmetic operations, such as $+$ or $-$. Constants such as 1 or 2 are considered to function symbols of degree zero. We shall use infix notation for comparison and other arithmetic operations, as shorthand for the prefix form (e.g $x+y$, as opposed to $+(x, y)$).

No particular interpretation of the predicate and function symbols is necessary for the Thinksheet model, but for the sake of discussion we will generally limit ourselves to arithmetic over the real numbers for function symbols, and corresponding comparison operations for predicate symbols.

From time to time in our discussion, we will refer to the set F as the set of all well-formed formulas in our logic language. The set C will refer to the set of all constants, i.e. all predicate and function symbols of degree zero and the set V will refer to all terms and atoms, i.e. all function and predicate calls. In general discussion the word *formula* will be informally taken to mean both terms and well-formed formulas.

2.2 The Definition of a Thinksheet

A Thinksheet consists of nodes, and precondition and answer formulas. Nodes are considered to be the variables of a Thinksheet. A precondition is a quantifier-free three-valued first order logic formula which evaluates to *true*, *false*, or *possible*. The three-valued logic is somewhat unique to our system and will be explained in section 2.3.

Definition 2.2 *A* Thinksheet T *consists of* (N, A, P) *such that:*

1. N *is a set of node symbols.*
2. $A : N \to V \cup F$ *is a many-to-one function that maps nodes to terms and formulas (called the answers of the nodes).*
3. $P : N \to F$ *is a many-to-one function that maps nodes to formulas (called the preconditions of the nodes).*

The functions A and P map nodes to the symbols representing a formula. They do not evaluate those formulas. For $n \in N$, we will call $A(n)$ the answer of node n and $P(n)$ the precondition of a node n.

P maps nodes to formulas specifying relevancy. If the precondition of a node is true, then it is considered relevant. A maps nodes to formulas representing the node's value if its precondition is true. For example, it might map the node to a symbol representing a reader's answer.

The variable symbols in precondition and answer formulas are the names of other nodes. No other variable symbols are allowed. The value of a variable is the same as the *effective value* of the node it refers to. The precise definition of effective value is given in Section 2.3.

Example 2.1 Let T be a Thinksheet constructed as follows:

1. $N = \{N_1, N_2, N_3, N_4, N_5\}$
2. A is a function defined as follows:

$$A(n) = \begin{cases} 2 & \text{if } n = N_1 \\ (N_1 + 2) & \text{if } n = N_4 \\ \textit{possible} & \text{otherwise} \end{cases}$$

3. P is a function defined as follows:

$$P(n) = \begin{cases} (N_1 = 2 \wedge N_2 = 3) & \text{if } n = N_3 \\ (N_3 = 1 \vee N_4 = 2) & \text{if } n = N_5 \\ \textit{true} & \text{otherwise} \end{cases}$$

2.3 Thinksheet's Three-valued Logic and Effective Values

Thinksheet's logic system consists of three truth symbols, *true*, *false* and *possible*. We call the formulas in our three-valued logic *extended boolean* formulas. We define a function that maps formulas to constant values. We call this the *effective value* of a formula. First we define an ordering between the logic values that will help us in our definition of effective value, and then give the definition of effective value.

Definition 2.3 *When we speak of the* maximum *or* minimum *of two truth values, we refer to following ordering:* false $<$ possible $<$ true.

Definition 2.4 (Effective Value) *The effective value function* $e : V \cup F, T \rightarrow C$, *which we will denote* $e(f, T)$, *maps a term or formula* f *in the context of some Thinksheet* $T = (N, A, P)$ *to a constant value in* C *in the following way:*

1. *If* $f = f_1 \wedge f_2$, *where* f_1 *and* f_2 *are formulas, then return the minimum of* $e(f_1, T)$ *and* $e(f_2, T)$.
2. *If* $f = f_1 \vee f_2$, *where* f_1 *and* f_2 *are formulas, then return the maximum of* $e(f_1, T)$ *and* $e(f_2, T)$.
3. *If* $f = \neg f_1$, *where* f_1 *is a formula, then return* $\neg e(f_1, T)$, *where* $\neg$ true $=$ false, $\neg$ false $=$ true *and* $\neg$ possible $=$ possible.
4. *If* f *is atom or term of the form:*

$$f_1(t_1, t_2, \ldots, t_n)$$

where f_1 is a function or predicate of degree n and $t_1, t_2, \ldots, t_n$ are terms, then return the interpretation of the function or predicate f_1 after the evaluation of the terms, meaning return

$$f_1(e(t_1,T), e(t_2,T), \ldots, e(t_n,T))$$

with the following restriction: if one or more of $t_1, \ldots t_n$ evaluate to a truth value, then return the minimum of the truth values. Otherwise return the value according to the particular interpretation of the atom's symbols.

5. *If f is a node symbol, n, then return the following:*

$$e(f,T) = \begin{cases} \text{false} & \text{if } e(P(n),T) = \text{false} \\ \text{possible} & \text{if } e(P(n),T) = \text{possible} \\ e(A(n),T) & \text{if } e(P(n),T) = \text{true} \end{cases}$$

Rule 4 pertains to function or predicate symbols applied to truth values. Rule 4 overrides any interpretation of these symbols.[1]

Example 2.2 The following are some example evaluations of extended boolean formulas involving only constant values (we leave out the Thinksheet for simplicity):

1. $e(\mathit{true} \wedge \mathit{possible}) = \mathit{possible}$ (Rule 1)
2. $e(\mathit{possible} \vee \mathit{false}) = \mathit{possible}$ (Rule 2)
3. $e(\neg(\mathit{possible} \vee \mathit{false})) = \mathit{possible}$ (Rules 2 and 3)
4. $e(\mathit{false} > 3) = \mathit{false}$ (Rule 4)

The definition of effective value in Definition 2.4 is recursive when a formula contains node symbols—in order to evaluate a formula, we must have the effective values of the nodes referenced in the formula. In order to get the effective values of the nodes, we must evaluate their precondition and answer formulas. For this reason, we disallow circular dependencies, meaning nodes that depend on each other's values. This way, when evaluating a particular formula, the recursion must stop—since the dependencies are acyclic, there must be some node(s) whose precondition(s) do not depend on any other node, and thus have no variable references in their precondition and answer formulas.

Example 2.3 Suppose we had the Thinksheet from example 2.1. Then the evaluation of the precondition of N_5 (which is $N_3 = 2 \vee N_4 = 3$) would happen this way:

[1] This means that rules 3 and 4 can lead to nonintuitive results. For example, the formula $\neg(\mathit{false} = 3)$ is not the same as $\mathit{false} \neq 3$. The first formula evaluates to *true*, while the second formula yields *false* (because any comparison with *false* still yields *false*).

1. The effective value of N_3 depends on its precondition, which is $N_1 = 2 \wedge N_2 = 3$. We evaluate this precondition in the following way:

   (a) N_1's precondition is *true*, and its answer formula is 2, so its effective value is 2. Therefore the expression $N_1 = 2$ is *true*.

   (b) N_2's precondition is also *true*, and its answer formula is *possible*, so its effective value is *possible*. By rule 4 the expression $N_2 = 3$ is *possible*.

   (c) The extended boolean expression $true \wedge possible$ is equal to *possible*.

   Therefore, N_3's effective value is *possible*.

2. The precondition of N_4 is *true*, and its answer formula is $N_1 + 2$, so its effective value is the evaluation of $N_1 + 2$, which in this case is 4. Therefore the expression $N_4 = 3$ is *false*.

3. N_5's precondition is reduced to the expression $possible \vee false$, which evaluates to *possible*.

   Therefore the value of N_5's precondition would *possible*.

2.4 A Rationale for the Effective Value of a Node

The semantics for Definition 2.4 may not make much sense in the abstract, but here is an intuitive rationale:

1. A node whose precondition is not satisfied can be discarded as *false*.

2. A node n whose precondition is *possible* reflects a state of uncertainly and therefore also has the value of *possible*.

3. A node whose precondition is *true* reflects the value of its answer — intuitively, it has the right to speak its value. However, even if the precondition is *true*, the node may not yet be answered by the reader. We reflect that condition by setting the answer formula to *possible* and thus the effective value is also *possible*.

In some sense, the values of *false* and *possible* represent meta-statements about a particular variable, as opposed to the actual value of the variable itself. For example, if a node represented the temperature outside, and the node was *false*, that does not mean that the value of the temperature outside was actually *false* (which makes no sense in any case). Rather it means that the value is unnecessary, or not applicable.

Another way to think about *false* is in the context of relational database tables. Very often, a *null* value is used in a table to represent the attribute is not applicable in that particular row. For example, suppose we had a database of researchers and the various microbes they worked on. A convenient way to store this might be a table with

the attributes, *Researcher*, *Microbe Type*, *Attributes for the Microbes*. Now it might be the case that not all of the attributes are used for a particular microbe type—for example, a bacterium does not have cilia, so any measurements about that would be inapplicable. In relational databases, we simply store *null* into those attributes that do not apply.

In Thinksheet, we would store preconditions with each of the attributes, so only the applicable attributes would be available. For example, for the attribute *Cilia Length*, we might specify the precondition *Microbe Type* = *Paramecium* to show that this measurement only deals with paramecia. Thus, when we are interested only in bacteria, *Cilia Length* is *false*, and thus not applicable. Thus corresponds to the *null* found in the relational table.

We use *possible* to distinguish between nodes which are definitely not applicable, and those nodes whose condition is uncertain or undefined. The value of *possible* can represent a node variable that is *undefined*—i.e. the node's precondition may be *true* and thus it is applicable, but no value has been assigned to it yet. This undefined state is propagated to the other nodes that depend on this node, so that they too might have the value *possible*. For these nodes, *possible* represents uncertainty, since we do not have enough information to compute the absolute truth or falsity (applicability or non-applicability) of the nodes because some of the nodes they depend on are undefined.

For example, if the precondition of a particular node is $N_1 = 3$, and N_1 is currently undefined, how do we evaluate the formula? One could argue that it should evaluate to *false* (after all, N_1 does not have the value of 3), but we take the stance that this loses information, because it would not properly take into account the undefined state of N_1. Hence the motivation of *possible* (with the hopes that the word conveys the ambiguity it is supposed to mean).

2.5 A Thinksheet Graph

We may construct a graph of dependencies of the formulas in a Thinksheet. This graph gives us the dependencies between the various nodes of the sheet.

If a formula f contains a variable that refers to node n', then n' is called a *parent* of that formula. If the formula is associated with the precondition of node n (i.e. if $P(n) = f$), then n' is called the *precondition parent* of n. Similarly, if $A(n) = f$, then n' is called the *answer parent* of n. Collectively, precondition parents and answer parents are referred to as the *parents* of a node.

Definition 2.5 *A* Thinksheet graph G_T, *for Thinksheet* $T = (N, A, P)$ *consists of* (V, E) *defined as follows:*

1. *A set of vertices,* $V = \{v_n | n \in N\}$.
2. *A set of directed edges,* $E = \{(v_1, v_2) | v_1$ *is a parent of* $v_2\}$.

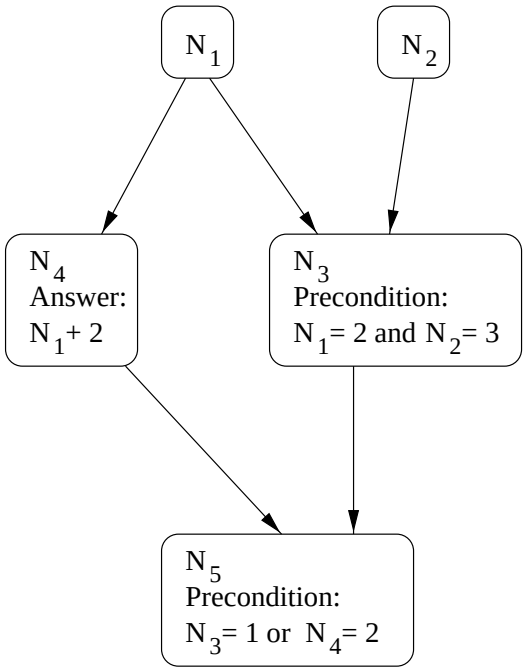


FIGURE 2.1: A example Thinksheet graph.

For example, the graph for the Thinksheet described in Example 2.1 is given in Figure 2.1.

From an interface standpoint, the leaves of the Thinksheet graph are intended to contain the information chunks (what will be referred to as *Text* nodes in Chapter 3), while the interior nodes contain the questions. From the point of view of the formal model, there is no meaning attached to any of the nodes, except that the nodes exist as variables in the system (the leaves are simply variables whose values are not used).

2.6 States and Consistent States

Definition 2.4 uses the answer and precondition formulas to map nodes to constant values, but that does not mean that these nodes explicitly take on those values. To help us separate this notion, we define the concept of a *state* to map nodes to values, and define a *consistent state* that maps nodes to their *effective values*.

Definition 2.6 *A* Thinksheet state, $S_T : N \to C$ *is a function that maps a node* n *in a Thinksheet* $T = (N, A, P)$ *to a constant value in* C.

Definition 2.7 *A state S_T is called a* consistent state *if*

$$\forall n \in N, S_T(n) = e(n, T)$$

A state of a Thinksheet may therefore map nodes to any values, but the interesting state is the one that is *consistent*. We may also refer to a consistent state as being *correct*, and any non-consistent state as *incorrect*.

The purpose of this definition of state is to mimic how a particular implementation gives values to nodes. We can then compare these values with their effective values to show the correctness of the implementation.

Example 2.4 Refering at Example 2.1 again, we may define a state S_T for Thinksheet T as follows:

$$S_T(n) = \begin{cases} 2 & \text{if } n = N_1 \\ \textit{possible} & \text{if } n = N_2 \\ \textit{possible} & \text{if } n = N_3 \\ 4 & \text{if } n = N_4 \\ \textit{possible} & \text{if } n = N_5 \end{cases}$$

In the context of the Thinksheet given in Example 2.1, S_T is a consistent state. S_T would *not* be a consistent state if, for example, $S_T(N_2) = 1$, because in the context of B_T the effective value of N_2 is not 1.

2.7 Node Domains

Our model up to now has left out the discussion of the domain of the node variables in a Thinksheet. For example, in our trial example given in Chapter 3, many of the nodes can only be answered with a 1 or a 2. Thus their domain is restricted to those two numbers.

In some cases it might be useful to explicitly specify the domain of values for the nodes in a Thinksheet. For example, if one of the questions of a Thinksheet asks for the reader's income, we expect a number of some sort, not a string. We can restrict the reader's answers by setting the domain of that node to the set of integers. If the reader mistakenly answers the question with a string (e.g. by entering her name), we will leave the result as implementation defined, but the simplest approach is to report an error.

Our approach to formalizing the notion of node domains makes note of two things. First the domain will refer to the node's effective value, not counting the meta-values of *true*, *possible* and *false*. Secondly, a node's domain may depend of the effective value of another node.

Definition 2.8 *A Thinksheet T may contain domains for each node with the addition of a function $D : N, T \rightarrow$ power set of C that maps nodes N to their domain of a set of constant values in C.*

We may overload the function D and define the domain of the entire Thinksheet as follows:

$$D(T) = \bigcup_{\forall n \in N} D(n, T)$$

We may redefine our notion of consistent states to take into account node domains by adding the extra constraint to Definition 2.7 that $S_T(n) \in D(n, T)$ in addition to being equal to the effective value.

An interesting case is if the domain of a node is the empty set. One possible interpretation of this is that the node can not be answered, since there is no consistent state in which that node has an answer. In this case, the node's effective value would be *possible* even if the its precondition was *true*, since it could never be bound to an answer. An implementation may hide the node from the reader when it discovers this, since the reader no longer has any viable way to answer this node.

Example 2.5 We create a domain function as follows for the Thinksheet described in Example 2.1.

$$D(n, T) = \begin{cases} \{1, 2\} & \text{if } n = N_1 \\ \{3, 4, 5\} & \text{if } n = N_4 \\ \{1, 2, 3\} & \text{if } n = N_3 \text{ and } e(N_1, T) < 2 \\ \{4, 5, 6\} & \text{if } n = N_3 \text{ and } e(N_1, T) \geq 2 \\ \text{Any integer} & \text{otherwise} \end{cases}$$

Note that the domain of N_4 actually encompasses more values than it can possibly take on (according to its answer in Example 2.1, its value is $N_1 + 2$. Since N_1 is 2, the only possible value for N_4 is 4). In fact, we may give a domain for N_4 in which it can take on none of the values. For example, a valid domain for N_4 could be $\{3, 5\}$. Since N_4 must have an effective value of 4 according to its answer, this leads to an inconsistency. There are many ways to interpret this inconsistency. The implementation may report an error, or, for example, ignore the answer when it does not fall into the domain, and thus N_4's effective value would be *possible*.

A more interesting case is shown for the domain for N_3. Here the domain depends on the effective value of N_1. We will see this idea put to good use in Chapter 6, when the domains of nodes become linked to the values of attributes in a table.

Chapter 3

The Thinksheet Interface

Thinksheet is an application with multiple user interfaces. Currently there exists an interface that looks like a spreadsheet, an interface via the World Wide Web and an interface through the scripting language Tcl. The spreadsheet is the primary interface to Thinksheet. However, nothing in Thinksheet or its model constrains it to the spreadsheet interface. A new type of interface may be developed using the Tcl interface.

Section 3.1 discusses the spreadsheet interface. The interface contains two major modes, called the reader and writer modes, which pertain to reading and writing complex documents respectively. These are discussed in Sections 3.2 and 3.3.

Section 3.4 compares the implementation with the formal model presented in Chapter 2. In Section 3.5 we discuss the interface through the World Wide Web and in Section 3.6 the interface through the scripting language Tcl.

3.1 The Spreadsheet Interface

The primary interface to Thinksheet closely resembles a spreadsheet. A screenshot is given in Figure 3.1.

The main portion of the Thinksheet screen is divided up into a grid of cells. Like a spreadsheet, the columns are labeled with letter values, and the rows with numbers. We will refer to cells as column letters followed by row numbers. For example, A2, refers to the cell in column A, row 2. The motivation for using an interface that looks like a spreadsheet is that people are familiar with this type of interface. Additionally, the grid provides an easy way of laying out the information. However, other types of interfaces may be developed using the Tcl/Tk scripting language (see Section 3.6).

Each non-empty cell represents a node in a Thinksheet. We partition the nodes into two types, those that contain information (e.g. hypertext) and those that contain a question to be asked to the reader. We will call these text nodes and question nodes respectively.

The non-empty cells are labeled with two lines. The first line is the title for the cell. The second line holds descriptive information about that cell. For example, in

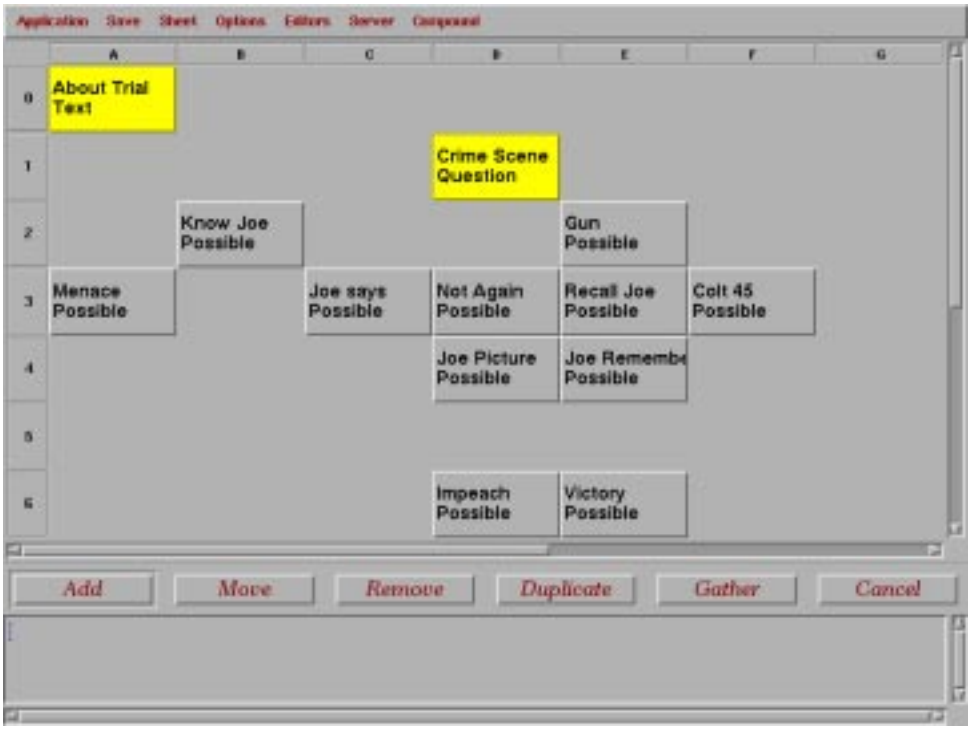


FIGURE 3.1: Screen shot of the trial strategy Thinksheet.

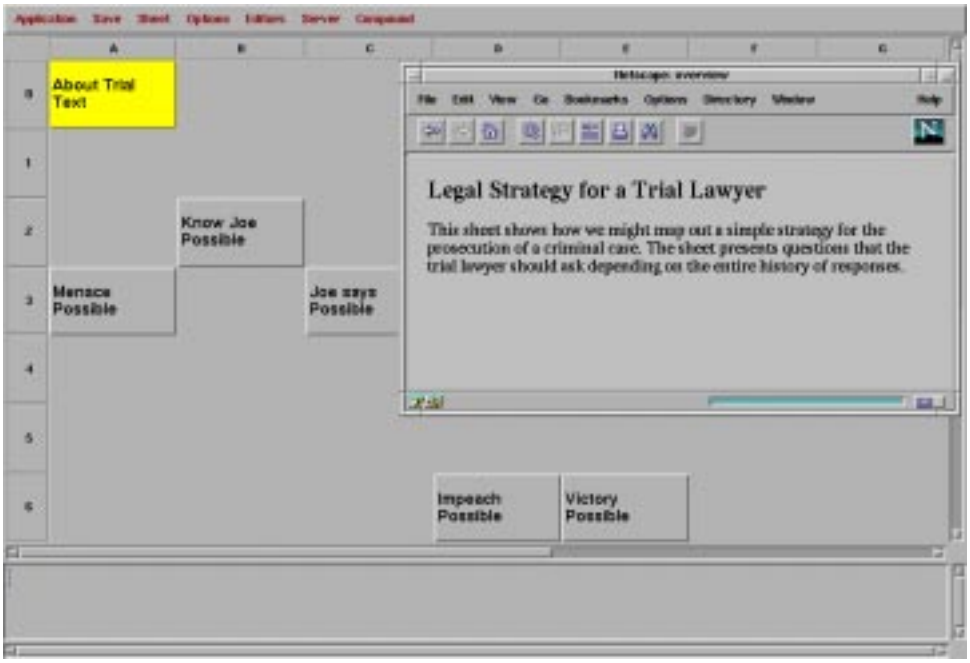


FIGURE 3.2: After clicking on the *About Trial* cell.

Figure 3.1, cell D1's title is *Crime Scene*, and its description is *Question*. The descriptive information tells us which type of cell it is—e.g. a Question cell will ask us a question.

Each cell also has a color associated with it. Cells whose preconditions are *true* are highlighted. In this example, cell A0 (*About Trial*) and cell D1 (*Crime Scene*) are both highlighted on startup. Cells which are *possible* are grayed out but still visible, and their descriptive line is *Possible*. As we will see later, cells which become *false* disappear from the reader's view.

Thinksheet has two modes, a reader mode, and a writer mode. In the reader mode, the reader answers various questions and retrieves the tailored information. In the writer mode, the writer creates the complex document by creating the question and text nodes and the preconditions for each of them.

The next section describes the interface for the reader by running through two example Thinksheets. Section 3.3 then describes the writer interface and how these two Thinksheets were created.

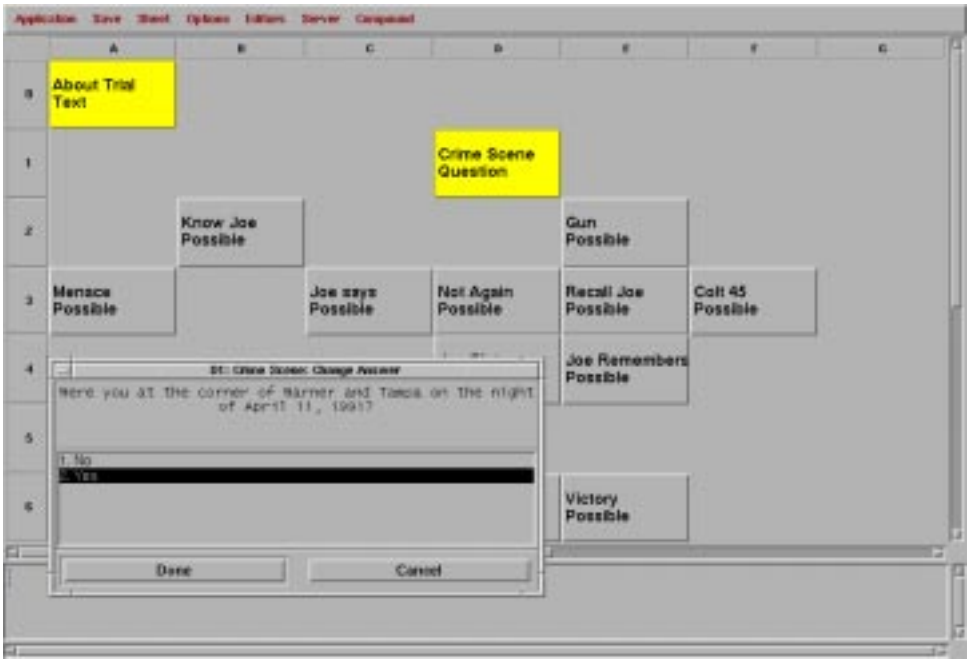


FIGURE 3.3: After clicking on the *Crime Scene* cell, the reader is presented with a question.

3.2 Reader Interface

3.2.1 A Trial Strategy

The first example maps out a strategy for the prosecution of a simple criminal case. The prosecutor wishes to question the suspect on the witness stand. Which questions he asks depends on the answers the suspect has given to previous questions. In this example, the prosecutor wants to establish that the suspect was both at the crime scene and had the murder weapon (in this case a gun).

Both the cells labeled *About Trial* and *Crime Scene* are highlighted on startup. This means that the reader should click on one of the two cells. If the reader clicks on *About Trial*, the reader will be shown a short paragraph which is a description of this Thinksheet (see Figure 3.2).

When the reader clicks on *Crime Scene*, a question is displayed. The reader is supposed to ask this question to the suspect and then input the suspect's answer (see Figure 3.3). The question states:

```
Were you at the corner of Warner and Tampa on the night of
April 11, 1991?
1. No
```

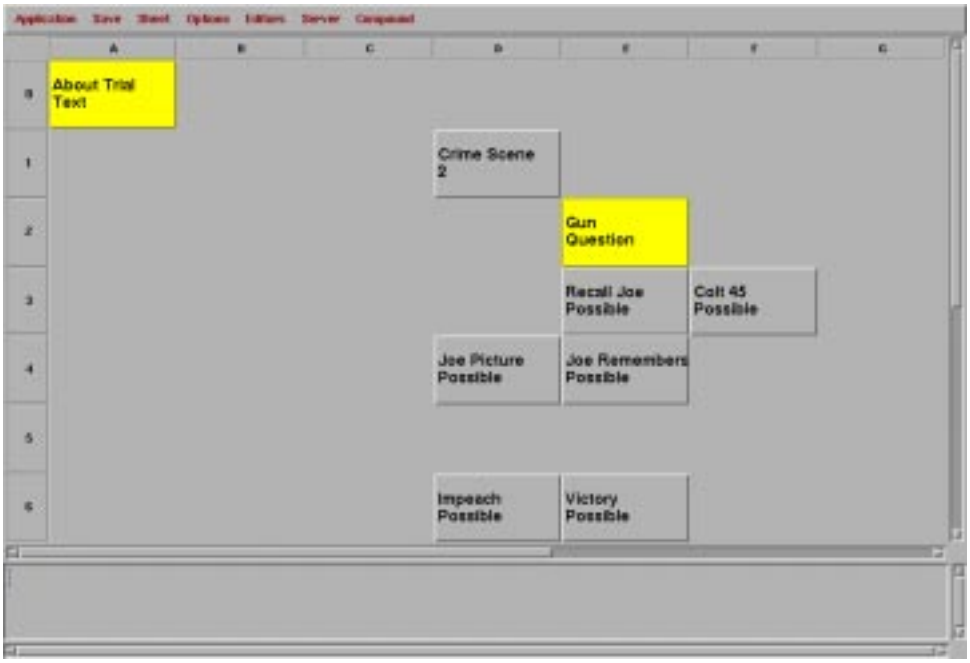


FIGURE 3.4: After answering the question for *Crime Scene*, cell E2 (labeled *Gun*) has been highlighted. Cells B2, A3, C3 and D3 have disappeared.

```
2. Yes
```

The reader (the prosecutor) should select the answer that the suspect gives him. Suppose the reader chooses *2. Yes*. The result is shown in Figure 3.4.

Notice that cell E2 has become highlighted. Its precondition has become *true* and it is the next question we should ask the suspect. Cells B2, A3, C3 and D3 have disappeared. Because of the suspect's answer the preconditions of these cells became *false* and have been removed from view.

The interaction continues like this with the prosecutor asking the suspect questions. Eventually, we may reach a state such as in Figure 3.5. In this state there are no more questions to ask and the cell *Victory* is highlighted. Clicking on this cell gives a short piece of text describing how the prosecutor has won.

This simple example highlights the basic interaction of Thinksheet. There are two types of cells—question cells and text cells. The reader answers the question cells and retrieves the relevant text from the highlighted text cells (i.e. those cells with a *true* precondition).

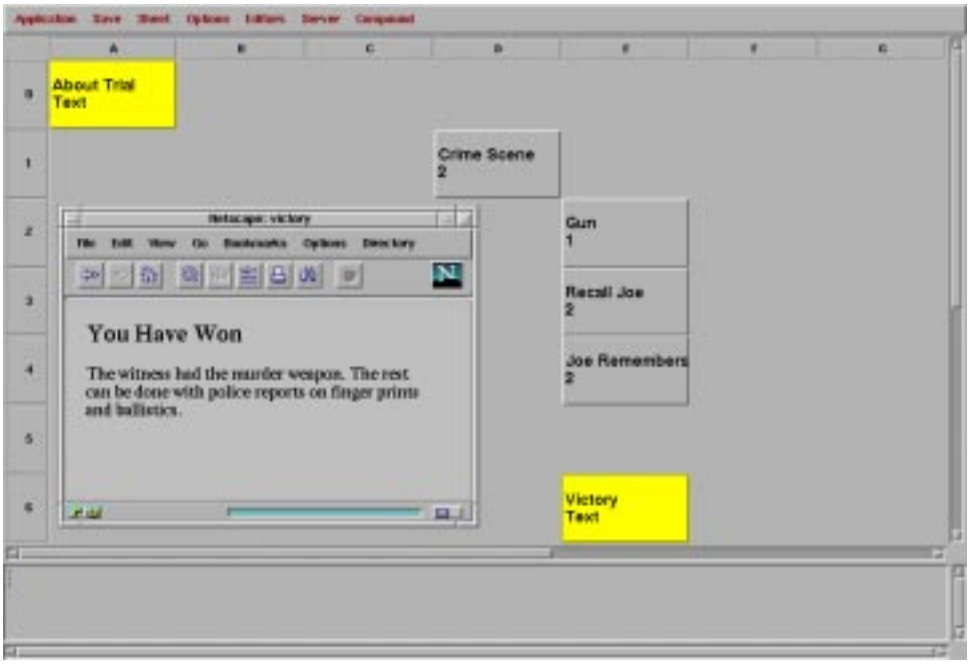


FIGURE 3.5: The result of the trial Thinksheet after answering more questions. Here the *Victory* cell is highlighted. The text of the cell is displayed on the side.

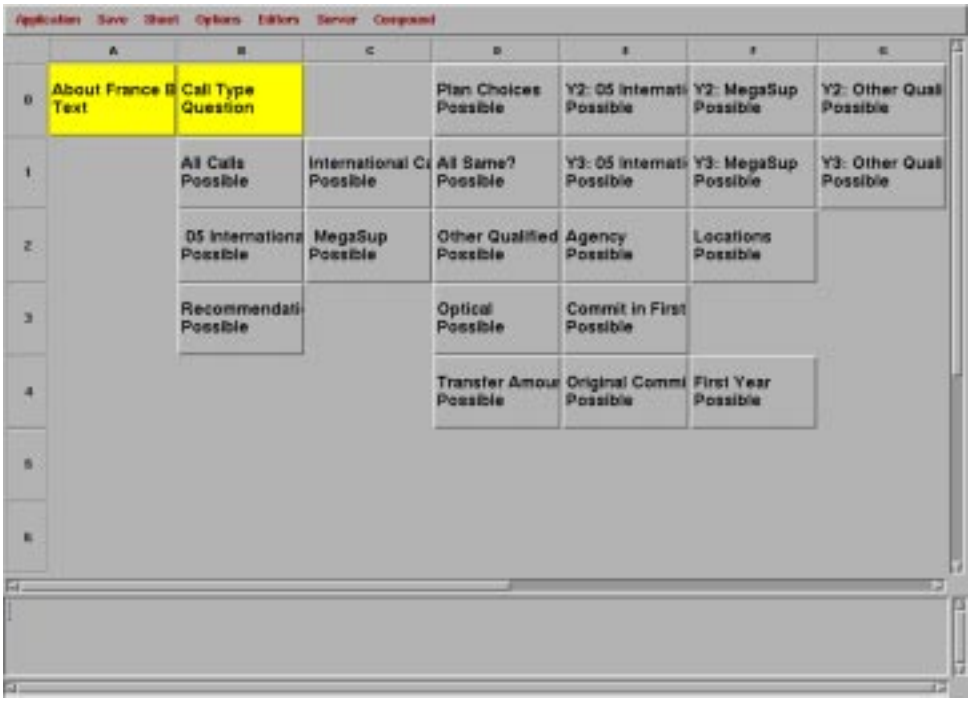


FIGURE 3.6: A telephone billing plan for toll-free calls.

```
These are mythical corporate phone plans for France.
05 calls are free to the caller but cost money to the
corporation receiving them.
They are analogous to 800 calls in the U.S.
Which Are you interested in purchasing corporate services for?

1. 05 calls (ACME SuperDuper 05 Term Plan II)
2. non-05 calls
```

FIGURE 3.7: The first question presented to the reader of the telephone billing Thinksheet.

3.2.2 A Corporate Billing Plan

Our next example demonstrates another feature of Thinksheet, that of dynamic document construction based on the reader's answers. The example concerns an imaginary telephone billing plan in France. The Thinksheet attempts to find the best billing plan for the reader based on his calling patterns. Under this plan, the reader receives better discounts if he commits a certain amount of money up front. The problem is to decide how much money the reader should commit. It applies to 05 calls (which are toll free calls—much like 800 calls in the United States) and also normal telephone calls.

In its initial state the sheet has two highlighted cells, *About France Billing*, which gives a short description of the sheet, and *Call Type*, which is the first question that the reader should answer (see Figure 3.6).

The first question is presented in Figure 3.7 and asks whether we are interested in 05 service or non-05 service.

If the reader answers with the first choice (05 calls), cell D1 (*Plan Choice*) will ask how long he wishes to stay in the plan, from one to three years. Suppose he answers that question with one year. The result is shown in Figure 3.8.

Notice that three cells, B2, C2 and D2, are lit up. The reader may choose to answer any of these three cells in any order. Each cell represents how much the reader expects to spend that year on a different type of calling pattern. For example cell C2 (*Megasup*) represents calls coming from inside the European Union. Since the receiver of a 05 telephone call pays for the call, the reader must estimate the cost of the calls which originate from inside the European Union.

In Figure 3.9 we see a recommendation of what the reader should commit after answering the three questions about his usage pattern. Figure 3.10 contains some of the text of the recommendation. Note that the recommendation includes calculations based on the amounts the reader has entered. Section 3.3.2 describes how the writer of this Thinksheet used Smartfields to dynamically construct the advice to the reader.

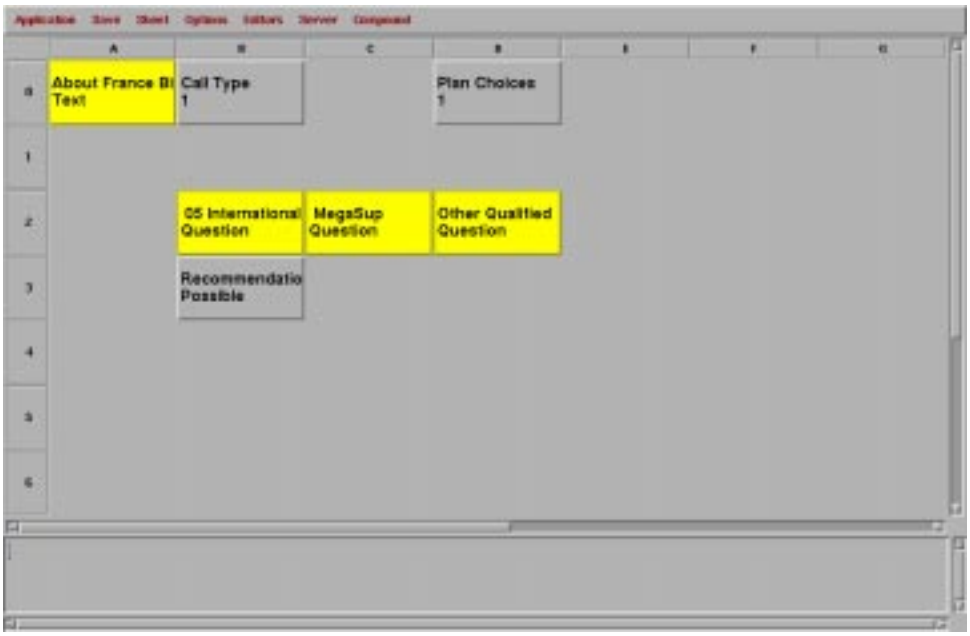


FIGURE 3.8: The billing plan Thinksheet asks for the usage pattern of the 05 calls.

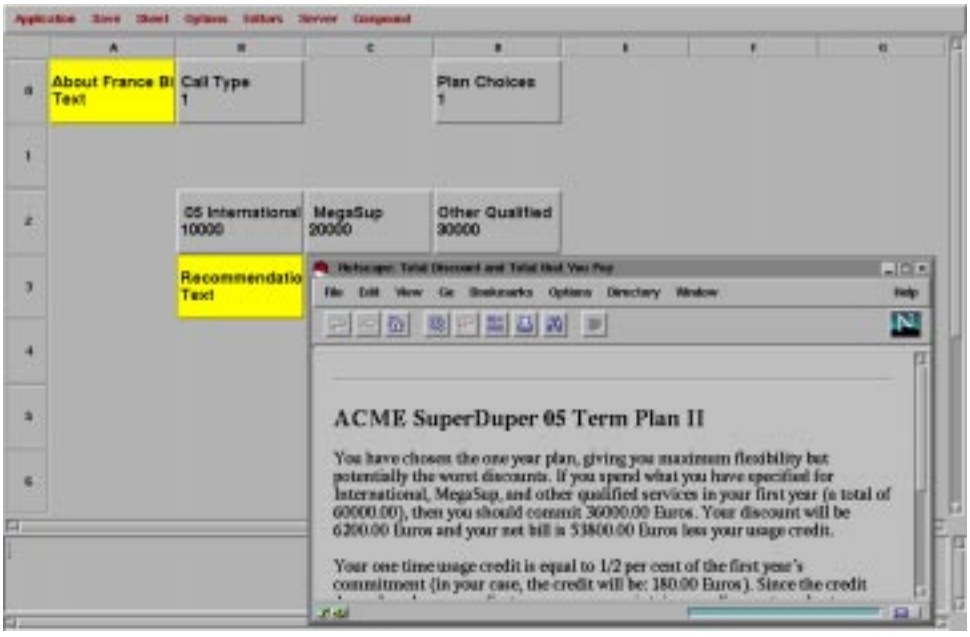


FIGURE 3.9: After entering in the usage pattern, the reader gets a recommendation for how me he should commit.

ACME SuperDuper 05 Term Plan II
You have chosen the one year plan, giving you maximum flexibility but potentially the worst discounts. If you spend what you have specified for International, MegaSup, and other qualified services in your first year (a total of 60000.00), then you should commit 36000.00 Euros. Your discount will be 6200.00 Euros and your net bill is 53800.00 Euros less your usage credit.
Your one time usage credit ...

FIGURE 3.10: The recommendation of the billing plan for the reader.

3.3 The Writer Interface

This section describes how the writer creates Thinksheets using the spreadsheet interface described in Section 3.1. Since each cell in the spreadsheet represents a node in Thinksheet, the writer creates a Thinksheet by adding cells (nodes) to the spreadsheet.

Nodes in the spreadsheet have the following fields:

- Title. This is the title of the node (such as the phrase *Crime Scene* in the trial strategy in Section 3.2.1). The title field is a Smartfield (described in Section 3.3.2).

- Question. This holds the question text for the node. This text is presented to the reader when he clicks on the node, and he is presented with the option of answering the question. The question field follows the convention that all of the text up to and including the first question mark (`(?)`) is the question to present to the reader. Every line after the question mark (if any) is considered a separate answer choice.

  The question field is also a Smartfield.

- Contents. This field contains the information that the node represents. Currently, this is only hypertext in HTML format [Wor]. When the reader clicks on this node, he will be presented with the hypertext information. The contents field is also a Smartfield.

- Answer. This is the answer field of the node as described in Chapter 2. In general this will hold the reader's answers to questions, although the writer of the Thinksheet may also fill this field with a formula.

- Precondition. This is the precondition field of the node as described in Chapter 2. This field will hold a boolean formula specifying the relevancy of the node.

The next sections describe the various features Thinksheet offers for creating complex documents. Section 3.3.1 gives an introduction to the language for writing precondition and answer formulas. Section 3.3.2 describes Smartfields. Section 3.3.3

gives an example of building a Thinksheet. Lastly, Section 3.3.4 describes a file format for creating and storing Thinksheets.

3.3.1 The Thinksheet Language

Thinksheet contains an interpreted, expression-oriented language for writing preconditions. We give an overview of the language here. A more complete description is given in Appendix B.

Base Types

The Thinksheet base types are floating point numbers and strings. From these base types, Thinksheet allows the construction of sets and ranges. The following are some examples:

1. `2.0` is a floating point number.
2. `"Hello"` is a string.
3. `[1,3,5]` is a set of numbers.
4. `1..5` is the range of floating point numbers between 1 and 5.

Nodes and Local Variables

Thinksheet expressions may reference nodes by their location on the spreadsheet. For example, `B1` references the value of the cell at location B1 on the spreadsheet.

Thinksheet also allows for local variables. These variables are local to a statement block (for example, a precondition or answer formula for a particular node). Local variables begin with an underscore (_) character, for example `_income`.

Expressions

The core of the Thinksheet language is the ability to write boolean expressions. In order to accommodate people's varying experience, we allow different syntactic conventions to express a boolean formula. For example, the following are equivalent:

1. `A0 = 3 and not (B0 = 4 or C0 = 6)`
2. `A0 = 3 && ~(B0 = 4 || C0 = 6)`
3. `A0 = 3 and not (B0 = 4 || C0 = 6)`

While the last expression is allowed in our language, the mixing of these conventions is discouraged.

Thinksheet also allows conventional arithmetic expressions using addition, subtraction, multiplication and division. It also has access to math library functions such as sin, cos, and log.

```
if (C1 < 50000) {
    _income := C1 * 0.90;
} else {
    _income := C1 * 0.85;
};
```

(a) An example `if` statement.

```
while (_i < 400 && _j < 200) {
    _sum := _sum + _i * _j;
    _i := _i + 2;
    _j := _j + 1;
};
```

(b) An example `while` statement.

```
func _ack()
{
    if ($1=0)
        return $2+1;
    if ($2=0)
        return _ack($1-1,1);
    return _ack($1-1,_ack($1,$2-1);
};
```

(c) An example function declaration.

FIGURE 3.11: An example of `if` and `while` statements and of a function declaration in the Thinksheet language.

Statements

Statements in Thinksheet allow for conditionals, loops and the assignment of local variables. All statements in Thinksheet are terminated with a semicolon (;).

Assignment statements are of the form `variable := value;`. For example:
`_income := C1 * 0.85;`
This statement assigns $C1 * 0.85$ to the local variable called `_income`.

The conditional statement looks much like the traditional `if` of the programming language C. An example is given in Figure 3.11(a). The first statement is executed only if the expression $C1 < 50000$ is *true*. Note that this expression may *possible*, especially if C1 has not been answered by the reader. In this case, and in the case that the expression is *false*, the second statement is executed.

```
Your tax bracket is:
%if% A0 > 50000 %then%
   50% and you owe %calc% A0 * 0.5 %endcalc% in taxes.
%endif%
%if% A0 <= 50000 %then%
   30% and you owe %calc% A0 * 0.3 %endcalc% in taxes.
%endif%
```

FIGURE 3.12: Example text containing Smartfield directives (directives are in bold).

Finally, loops are allowed using `while` statements. An example is given in Figure 3.11(b).

Function Declaration

Functions may be defined globally for a particular Thinksheet. An example of a function declaration is given in Figure 3.11(c). This function computes Ackermann's function. Function parameters are passed by $1, $2, ... , where the numbers refer to the parameter position.

3.3.2 Smartfields

Text nodes contain hypermedia information (currently HTML). In addition, the text may contain directives that refer to the effective values of other nodes. We call text with these directives *Smartfields*.

Smartfields are useful for including calculations based on the reader's answers into the text. An example of text containing Smartfield directives in given in Figure 3.12. In this simple example, suppose that the reader input his income for the year into cell A0. Then when the reader retrieves this text, it is processed using the directives inside the percent signs. The `%if%` statement is like the `if` of common programming languages. If the condition of the `%if%` is *true* or *possible* then the text between the `%then%` and the `%endif%` is inserted into the result. Thus, in Figure 3.12, if the reader's income is greater than $50,000, then he is in the 50 percent tax bracket. If it is less than that, then he is in the 30 percent bracket. The `%calc%` statement allows calculations based on the reader's answers. In this case, it gives a simple calculation of what the reader owes.

Inserting the Smartfields of Other Nodes

In addition to this simple processing ability, Smartfields are capable of inserting the Smartfields of other nodes using the `%insertcontents%` directive. Thus another node's Smartfield can be used like a function. An example of this is given in Figures 3.13(a)–(b). Cell A0 contains the text to be inserted. Like a function, the node's Smartfield

```
%if% $2 = 1 %then%
%calc% $1 %endcalc% is %calc% $2 %endcalc% year old.
%endif%
%if% $2 > 1 %then%
%calc% $1 %endcalc% is %calc% $2 %endcalc% years old.
%endif%
```

(a) Cell A0 holds the text to be inserted by another cell.

```
Here is a list of the children in your family:
%insertcontents% A0("Mary", 1) %endinsertcontents%
%insertcontents% A0("Joe", 7) %endinsertcontents%
```

(b) Cell A1 inserts the contents of A0 in its Smartfield.

```
Here is a list of the children in your family:
Mary is 1 year old.
Joe is 7 years old.
```

(c) The result of retrieving the contents of A1.

FIGURE 3.13: Example of the `%insertcontents%` directive.

take parameters. In this case the parameters are referenced by `$1` and `$2` as the first and second parameters of the function call.

In Figure 3.13(b), cell A1 inserts the Smartfield of A0 twice, each time passing different parameters (in this case the parameters refer to the names and ages of children). When a reader retrieves the text of cell A1, the result is shown in Figure 3.13(c).

Preambles to Allow Local Variables in Smartfields

Smartfields also have a *preambles*, which are sections in the beginning of the document to set local variables. The text inside of the preamble is actually code in the language described in Section 3.3.1. Any local variables set in the preamble may be used later in the Smartfield. The variables are local only to that particular Smartfield and may not be accessed by other Smartfields.

Figure 3.14 shows an example of the usage of preambles. In this case, we show another way to write the Smartfield in Figure 3.12. Here, instead of using the `%if%` directive to decide the tax bracket, we set two local variables inside of the preamble. Then in our text, we insert the values of those variables using the `%calc%` directive.

The complete list of commands available to Smartfields is given in Appendix C.

```
%preamble%
if (A0 > 50000) {
    _bracket := 50;
    _taxes   := A0 * 0.5;
};
if (A0 <= 50000) {
    _bracket := 30;
    _taxes   := A0 * 0.3;
};
%endpreamble%
Your tax bracket is %calc% _bracket %endcalc% percent,
and you owe %calc% _taxes %endcalc% in taxes.
```

FIGURE 3.14: Preambles allow local variables inside a Smartfield.

```
Which artist are you interested in?
%if% A0 = "Modern" %then%
Picasso
Dali
%endif%
%if% A0 = "Renaissance" %then%
Michaelangelo
Raphael
%endif%
```

FIGURE 3.15: Using a Smartfield for questions.

```
The list of employees whose salary is greater than
%calc% B2 %endcalc%.

%servercommand%
"SELECT NAME "
"FROM EMPLOYEES "
"WHERE SALARY > " B2
%endservercommand%
```

FIGURE 3.16: Example of the %servercommand% directive.

Questions and Titles as Smartfields

We have described using the Smartfields to represent the textual information of the text nodes. In fact, the titles and question fields for nodes may also be Smartfields. For questions, this means we can even modify the list of choices presented to the user on the fly. See Figure 3.15 for example.

Here cell A0 represents the art period of the reader's interests. If the reader has answered A0 with the choice "Modern" then the choices of this question are constrained to only modern artists (e.g. Picasso and Dali).

Thinksheet's Connection with Servers

Thinksheet has a general purpose method to communicate with processes such as databases, interpreters etc. The communication is through UNIX pipes [Ste90], so whatever the server prints to its standard output device will be treated as results to Thinksheet. Servers were created for communication with databases (through SQL interpreters), although they may be used to talk to other types of processes.

Servers are accessed via the %servercommand% directive in Smartfields. An example of the usage of %servercommand% is given in Figure 3.16. In this example, the server is an SQL interpreter and the query asks for a list of employees meeting a certain criterion (based on node B2). When the query is run, it will be replaced by the list of employees.

See Appendix C for the exact syntax for server commands, as well as the method for connecting the servers to Thinksheet.

3.3.3 Building a Thinksheet

To make the above discussion more concrete let us show how we built our example sheets—the prosecutor's trial strategy and the phone billing plan.

In our presentation of the trial strategy example, the initial question asks if the suspect was on a certain street corner on a specific date. Since this first question must

	A	B	C	D	E	F
1				Crime Scene		
2		Know Joe D1=1			Gun D1=2 or C3=2	
3	Menace B2=1		Joe Says B2=2 or A3=2	Not Again D1=1 and (E2=1 or F3=1)	Recall Joe D1=2 and (E2=1 or F3=1)	Colt 45 E2=2
4				Joe Picture E3=1	Joe Rememb E3=2 or D4=2	
5						
6				Impeach A3=1 or C3=1 or D3=1 ...*	Victory F3=2 or E4=2 or D3=2	

* full precondition is A3=1 or C3=1 or D3=1 or D4=1 or E4=1

FIGURE 3.17: The titles and preconditions of the trial strategy.

```
%if% d0 = 1 %then%
    You have chosen the one year plan, giving you maximum
    flexibility but potentially the worst discounts.
%endif%
%if% d0 = 2 %then%
    You have chosen the two year plan.
%endif%
%if% d0 = 3 %then%
    You have chosen the three year plan, giving you
    potentially the best discounts.
%endif%
If you spend what you have specified
for International, MegaSup, and other qualified services
in your first year (a total of
%calc% _tot %endcalc%),
then you should commit
%calc% _commit %endcalc% Euros.
```

FIGURE 3.18: The Smartfield for the final advice in the telephone billing plan.

always be asked its precondition should always be *true*. We leave the precondition empty, which is equivalent to typing in the expression *true*.

When the prosecutor enters in the suspect's answer of *1. No*, node B2 became highlighted. This occurs because the precondition of B2 is D1 = 1 (D1 is the node containing the initial question). The rest of the questions are set up in the same way but with different preconditions. Figure 3.17 shows the titles and preconditions of the trial Thinksheet.

The phone billing plan is set up similarly, but with the added twist that the advice is constructed using a Smartfield. Figure 3.18 shows a portion of the Smartfield used to create the advice. Note that it inserts calculations about spending amounts and commitments. The local variables `_tot` and `_commit` were computed in the preamble section of the Smartfield and are based on the reader's answers. Also note that the Smartfield uses the `%if%` directive to take into account the number of years the reader specified for the plan.

3.3.4 Report Format

Thinksheets may be created and stored in an ASCII representation that is called the Thinksheet Report Format.[1] A Thinksheet report is made up of fully delimited fields.

[1] This format is the brainchild of Dave Tanzer, and was originally referred to as "The Tanzer Format."

```
%node% D1
%title%
Crime Scene
%end title%
%question%
Were you at the corner of Warner and Tampa on the night
of April 11, 1991?
1. No
2. Yes
%end question%
%end node%

%node% B2
%title%
Know Joe
%end title%
%question%
Show a picture of key witness Joe.
Do you know this man?
1. No
2. Yes
%end question%
%precondition%
D1=1;
%end precondition%
%end node%
```

FIGURE 3.19: Two nodes of a Thinksheet represented in the report format (field delimiters are in bold).

Since it is easily readable, writers may directly modify this file when working on a Thinksheet instead of using the spreadsheet interface described in Section 3.1.

Each field in a report file is delimited by `%field%` and `%end field%`. The report in Figure 3.3.4 represents two nodes at locations D1 and B2. The nodes have the titles *Crime Scene* and *Know Joe* respectively. Each has a particular question, and node B2 has a precondition (D1 = 1). Not all the fields have to be listed for a particular node (e.g. node D1 does not have a precondition). If a particular field is not listed, then it is considered unset or blank.

A report file may be merged with a running Thinksheet. All fields that are listed are modified with the new values, while those that are not remain unchanged.

This feature can be used to set the answers to several questions in response to one

reader answer. For example, suppose we had an application that required information about the reader such as name, age, height, eye color, etc. We may consider each attribute about the reader as a separate question that he has to fill in. However, this may be very inconvenient, especially if such information is stored in a database. Instead, when the reader fills in his name, Thinksheet can take action by querying the database, creating a report form with the answers to the attribute questions filled in, and then merging that report (see Appendix C for the full example).

3.4 The Implementation and the Formal Model

This section compares the implementation we have discussed in this chapter with the model given in Chapter 2. The formal model of Thinksheet considers nodes to be variables that can be assigned values. The real implementation however has added various features to the nodes that are not in the abstract model. For example, in this chapter we have discussed nodes that hold questions, and nodes that hold text. A question for a node holds no special significance in the formal model except that it might express a domain of values for the node. For example, the choices of *1. No* and *2. Yes* limit the domain of the node to the values of one and two. Text nodes may be considered leaf nodes whose answer values are the Smartfields. Smartfields, in turn, may simply be considered to be functions that return textual strings. The difference is however, that we allow cycles through the use of the `%insertcontents%` directive. Our model does not allow cycles created by dependencies on the answer and precondition formulas of a node. For this reason, Smartfields are considered to be outside the core model—a sort of additional field added to each node in a Thinksheet.

The language given in our implementation is more complicated than the syntax described in Section 2.1. We allow local variables, user defined functions and programming constructs such as loops. For the most part, these additions have no effect on the formalism, except for one point. In Section 2.1, we said that we make no assumptions about the interpretation of the various function and predicate symbols, but it was the case that these functions and predicates affected only the effective answer of the node they were contained in. They did not have side effects. For example, functions that input or output to files are not idempotent, but store a state (in this case the position in the file).

Adding an outside state to a Thinksheet greatly complicates matters. Since a state depends on its previous state, it is, in some sense, like adding a node variable that depends on itself. We prefer to keep our Thinksheet graph acyclic because of the simplicity it allows. On the other hand, we must cope with reality, so some notion of state is allowed into our implementation. For example, we do allow the usage of system calls, and communication with outside processes. This typically takes the form of querying databases, or doing I/O with files.

We leave the definition of how such a state is updated to the implementation. In our implementation, this updating depends on our algorithm for propagation, described

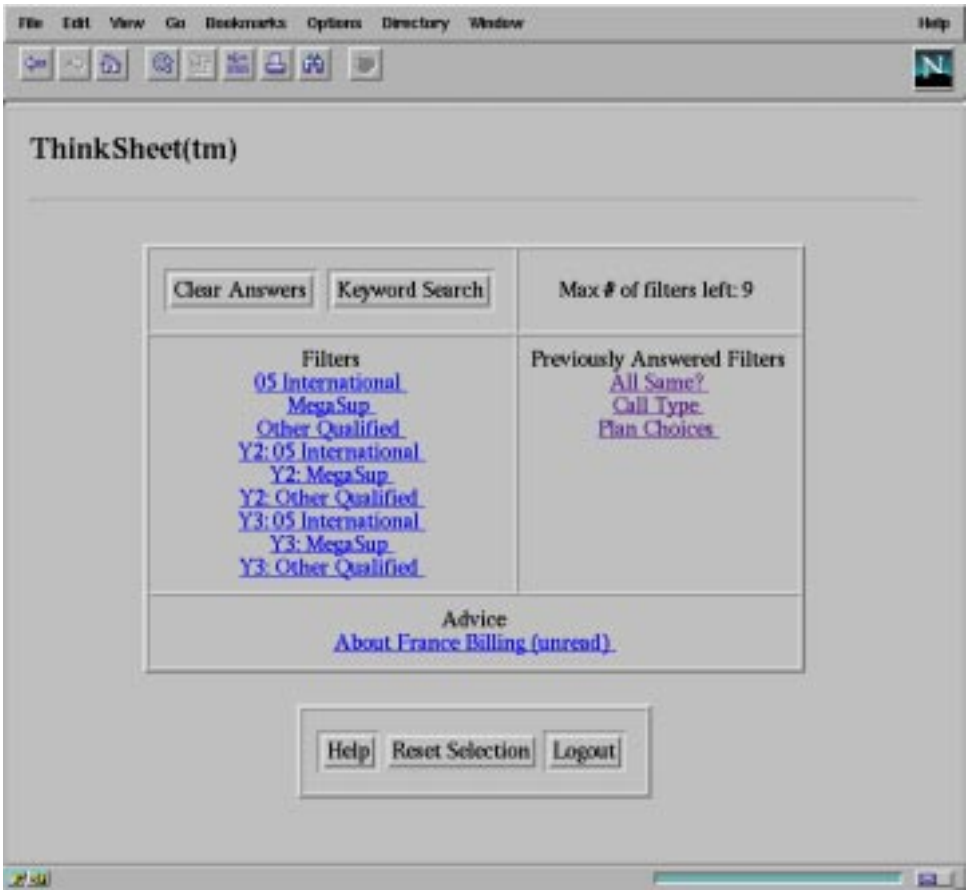


FIGURE 3.20: Screen shot the Web interface.

in Chapter 4.

3.5 The World Wide Web Interface

Thinksheet has a reader-mode-only CGI interface via the World Wide Web. The interaction between the reader and the Thinksheet is similar to the spreadsheet interface, but without the graphical niceties. The main part of the interface consists of three lists. One list displays questions that the reader can answer, another list displays the questions that the reader has already answered (so he can modify those answers if he changes his mind), and the last list displays the relevant (*true*) text titles. A screenshot of this interface is given in Figure 3.20.

As with the spreadsheet interface, the reader answers a series of questions in the list. When he has answered enough questions, he can access the relevant text, by

clicking on one of the titles.

3.6 The Tcl Interface

Tcl is a cross-platform extensible scripting language that runs on UNIX, Windows and Macintosh systems [Wel97]. Tcl has a GUI interface extension called Tk that allows the quick construction of custom GUI systems.

An extension has been made to Tcl that allows access to Thinksheets. The extension is in the form of new Tcl commands that allow the script-writer to create and modify Thinksheets. The purpose of the Tcl interface is to allow the creation of custom interfaces for Thinksheets without the complexity of C and C++ programming. For example, the World Wide Web interface described in Section 3.5 uses the Tcl interface on the server side.

This section describes the Tcl interface to Thinksheet. While some knowledge of Tcl and Tk is helpful, it is not necessary to understand this section. The full reference to the Tcl interface is given in Appendix D.

3.6.1 Loading a Thinksheet in Tcl

The command to load a Thinksheet into Tcl is as follows:

`sheetGraph <thinksheet>`

This command returns a Thinksheet object labeled `graph<n>` where `<n>` is a unique number (i.e. the first Thinksheet that is loaded is labeled `graph0`). The unique number allows us to have multiple Thinksheets loaded at the same time.

The label `graph0` represents a Thinksheet object and can be used to access information about it. It is now actually a new Tcl command, and we access the Thinksheet in the following way:

`graph0 <method>`

Here, `<method>` refers to the method that we want to apply to the object labeled with `graph0`. For example:

```
graph0 nodeList
```

returns all of the nodes residing in the Thinksheet graph.

We can set the answer to a node in the following way:

`graph0 nodeSetAnswer <id> <value>`

In this case `<id>` represents the node identifier, and `<value>` represents the new value with which to set the answer.

3.6.2 Cells in the Tcl Interface

When we change the answer of a particular node, the status of other nodes might change along with it. We would like to be notified when the status of a particular node changes. This would allow us to skip the task of scanning the entire node list for changes each time we change the answer of a node.

```
proc printNode {i g n} {
    puts "Node: '[$g nodeTitle $n]' Status: '[$g nodeStatus $n]'"
}
set graph [sheetGraph /home/thinksheet/sheets/trial]
foreach n [$graph nodeList] {
    cell $graph $n doNothing printNode doNothing
}
```

FIGURE 3.21: The `printNode` Tcl procedure.

Cells in Tcl take on this task. The command to create a cell is follows:
`cell <graph> <nodeId> <updateProc> <deleteProc>`
The various parameters have the following meaning:

1. `<graph>`: The name of the Thinksheet graph that you want to use.
2. `<nodeId>`: The node identifier of the particular node that you want the cell to watch.
3. `<updateProc>`: This is the name of a Tcl procedure that gets called when the status of the node changes.
4. `<deleteProc>`: This is the name of a Tcl procedure that gets called when the cell or corresponding node is deleted.

The `cell` command returns a cell identifier of the form `cell<n>`, (for example `cell0`). Like the creation of a Thinksheet, this return value is itself a command, and can be used to manually call the cell update procedure, by running:

```
cell0 update
```

The three procedure names passed to the cell command must refer to procedures that take three parameters. The three parameters that will be passed are: (1) the cell identifier, (2) the graph identifier, and (3) the node identifier.

Figure 3.21 gives an example where we add a cell to each node in the trial Thinksheet. When a node gets updated, the cell is instructed to run the `printNode` procedure, which prints out the title and status of the particular node.

The result of setting the answer to one of the nodes is given in Figure 3.22.

3.6.3 An Example Tcl Program

Here we describe an example program using the Tcl interface. This program answers unanswered questions with random choices until there are no more questions to answer. This type of activity might be useful if, for example, one were trying to test the integrity of a Thinksheet.

```
% graph nodeSetAnswer 1003 "1"
Node: 'Crime Scene' Status: '1'
Node: 'Know Joe' Status: 'Question'
Node: 'Recall Joe' Status: 'False'
Node: 'Joe Picture' Status: 'False'
Node: 'Joe Remembers' Status: 'False'
```

FIGURE 3.22: The `printNode` procedure is executed when a node is answered, and other nodes' statuses change.

```
set graph [sheetGraph /home/thinksheet/sheets/trial]
set newquestions {}
foreach n [$graph nodeList] {
    if ![string compare [$graph nodeStatus $n] "Question"] {
        lappend currentquestions $n
    }
    cell $graph $n doNothing cellUpdate doNothing
}

while {[llength $currentquestions] != 0} {
    foreach n $currentquestions {
        randomAnswer $graph $n
    }
    set currentquestions $newquestions
    set newquestions {}
}
```

FIGURE 3.23: The main Tcl program.

We first take a look at the main portion of the Tcl program, given in Figure 3.23. The first line loads the trial strategy Thinksheet described in Section 3.2.1. Then the variable `newquestions` is set to the empty list. This variable will hold the new questions that arise as some questions get answered.

The `foreach` loop does two things. First, if the node is a question node (which is checked by comparing its status with the string `"Question"`), then it is appended to the list `currentquestions`, which maintains the current questions to be answered. Second, it creates a cell for each node. The cell runs `cellUpdate` whenever the cell gets updated, and `doNothing` for initialization and deletion. These procedures are shown in Figure 3.24.

The `doNothing` procedure, as its name implies, does nothing. The `cellUpdate` procedure, when executed, checks to see if its node has turned into a question (by comparing its status with the string `"Question"`). If the node is a question, it is appended to the list `newquestions`.

The `while` loop in Figure 3.23 loops while there are still questions left in `currentquestions`. These questions are answered by the `randomAnswer` routine (given in Figure 3.25). The line

```
set currentquestions $newquestions
```

makes the new questions become the current questions.

We then reset the `newquestions` list and continue the loop.

The `randomAnswer` procedure gets the choice list of the node, and randomly picks one (with the help of the `random` procedure in Figure 3.26). It then extracts the number from the choice using the `extractNumber` procedure given in Figure 3.27. Then it sets the node's answer.

```
proc doNothing {i g n} {}
proc cellUpdate {i g n} {
    global newquestions
    if ![string compare [$g nodeStatus $n] "Question"] {
        lappend newquestions $n
    }
}
```

FIGURE 3.24: The `doNothing` and `cellUpdate` Tcl procedures.

```
proc randomAnswer {g n} {
    set choiceList [$g nodeAnswerChoices $n]
    set choice [lindex $choiceList [random 0 [llength $choiceList]]]
    set answer [extractNumber $choice]
    $g nodeSetAnswer $n $answer
}
```

FIGURE 3.25: The `randomAnswer` Tcl procedure.

```
proc random {low high} {
    return [expr int(rand() * ($high - $low) + $low)]
}
```

FIGURE 3.26: The `random` Tcl procedure.

```
proc extractNumber { c } {
    if [regexp {^[0-9]+} $c number] {
        return $number
    } else {
        return $c
    }
}
```

FIGURE 3.27: The `extractNumber` Tcl procedure.

Chapter 4

Algorithms for Maintaining Consistency of a Thinksheet

This chapter focuses on the various algorithms for maintaining Thinksheets. When a reader changes the answer of a particular node, that node's effective value changes. This change must be propagated to its descendents on the dependency graph. We call this process *propagation*. Section 4.1 gives an efficient algorithm for propagation.

In section 4.2 we show various optimizations that can be done for special cases. These special cases involve specific kinds of boolean formulas in the preconditions of the nodes. In these cases, we may sometimes skip the re-evaluation of the precondition even if the general algorithm for propagation calls for it. We then fit these optimizations into the general algorithm.

Section 4.4 gives a full example of propagation.

4.1 Propagation

When a node's effective value changes, the new value must propagate to all of its descendents in the Thinksheet graph. This means that its descendents may have to recalculate themselves.

The definition of effective value in Section 2.3 gives us a naive way of doing this. Start at each leaf of the Thinksheet graph (i.e. those nodes without dependent children) and calculate their effective values with a recursive call to the effective value function on the parents of that leaf.

This may be inefficient however, because many nodes will have to be recalculated that need not be. For example, suppose the precondition of node C0 is $A0 = 2 \wedge B0 = 3$. If we change the effective value of A0, then we might start recalculating the effective value of C0. By the definition of effective value, we must also calculate the effective value of B0. Now it might be the case that B0 does not depend on A0 (i.e. it is not a descendent of A0 in the dependency graph), and therefore we can just use the old

FIGURE 4.1: Using depth-first search for propagation may result in visiting a node multiple times.

effective value and skip the recalculation. But when working in this bottom-up fashion, we have no way of knowing if this is true.

Another way of doing propagation would be to do a depth first search from the node that has changed, re-evaluating each node as we visit it. However there is the disadvantage that a single node may be evaluated many times during the course of the search. This may happen if the node has multiple parents on the descendent graph (see Figure 4.1).

We present an efficient algorithm for doing propagation. In order to do this, we first define what it means when a reader "changes the answer in a node".

When a reader changes the answer or precondition of a node, the Thinksheet changes. The changes that are made reflect the difference between the previous Thinksheet and the new one. We define the difference between two Thinksheets as follows:

Definition 4.1 *The difference between two Thinksheets* $T = (N, A, P)$ *and* $T' = (N', A', P')$*, is the set defined as follows:*

$$T' - T = \{n \| n \in N' \wedge (n \notin N \vee (n \in N \wedge (A(n) \neq A'(n) \vee P(n) \neq P'(N)))\}$$

In words, $T' - T$ is the set of nodes such that either n is a "new" node in T'—i.e. $n \in N'$, but $n \notin N$—or the precondition or answer for that node has changed. We leave out the case that node n has been deleted from T'—i.e. $n \in N$, but $n \notin N'$. If node n has been deleted, then no other nodes must refer to n, otherwise T' would not be a Thinksheet (Definition 2.2 disallows references to non-existant nodes). Therefore, deleted nodes are of no consequence to propagation.

Note that in the definition, $A(n) \neq A'(n)$ and $P(n) \neq P'(n)$ refer to syntactic equality, not equality of value. For example, if $A(n) = 7 - 2$ and $A'(n) = 6 - 1$, then $A(n) \neq A'(n)$. The reason is that we have not evaluated the answer or precondition formulas. We are merely looking for changes in the symbols for the formulas.

Thinksheet's algorithm for propagation is given in Algorithm 4.1. This algorithm constructs a topological sort of the changed nodes and their descendents. Each node has a mark, either *evaluate* or *don't-evaluate*. The changed nodes are initially marked with *evaluate*, while the other nodes are marked with *don't-evaluate*.

The algorithm then iterates through the topological sort. At each node, it checks the mark. If the mark says *don't-evaluate*, the algorithm skips that node and moves onto the next one.

If the marks says *evaluate*, the effective value of the node is re-evaluated. If its effective value of the node has changed, all of its children are marked *evaluate*.

The processing ends when we reach the end of the topological sort.

Input:
$T^{\text{old}} = (N, A, P)$, the old Thinksheet, $T^{\text{new}} = (N', A', P')$, the new Thinksheet. S_T^{old} (the state of the old Thinksheet T).
Output:
S_T^{new} (the state of the new Thinksheet T^{new}).
Initialization:
$R \Leftarrow (T^{\text{new}} - T^{\text{old}})$
$V \Leftarrow$ topological sort of $R \cup$ descendents of R, according to $G_{T^{\text{new}}}$.
for $n = V$.first to V.last **do**
  n.mark $\Leftarrow$ *don't-evaluate*
for all $n \in R$ **do**
  n.mark $\Leftarrow$ *evaluate*
for all $n \in N'$ **do**
  $S_T^{\text{new}}(n) \Leftarrow S_T^{\text{old}}(n)$
Loop:
for $n = V$.first to V.last **do**
  if n.mark $=$ *evaluate* **then**
    $S_T^{\text{new}}(n) \Leftarrow e(n, T^{\text{new}})$
    if $S_T^{\text{new}}(n) \neq S_T^{\text{old}}(n)$ **then**
      for all $c \in n$.children **do**
        c.mark $\Leftarrow$ *evaluate*

Algorithm 4.1: The Propagation Algorithm

In our implementation, the Thinksheet will start out in some state, S_T^{old}. Our propagation algorithm will bring the Thinksheet to state S_T^{new}. The algorithm is considered to be correct if the new state, S_T^{new}, is a consistent state.

Theorem 4.1 *If the input state, S_T^{old}, for Algorithm 4.1 is a consistent state, then the output state, S_T^{new} is also consistent.*

Proof In order to show that S_T^{new} is consistent, we must show that

$$\forall n, S_T^{\text{new}}(n) = e(n, T^{\text{new}})$$

At the end of the algorithm, we have nodes marked *evaluate* and nodes marked *don't-evaluate*. Divide them up into disjoint sets, E and $\overline{E}$, respectively.

Nodes in E are explicitly re-evaluated, so they trivially satisfy this condition.

For each $n \in \overline{E}$, it is the case that none of the parents of n have changed in their effective value (otherwise the node would be marked). Also, neither the precondition nor the answer for n has changed (otherwise it would be marked *evaluate* during the initialization). Therefore the effective value of n has not changed. Thus $S_T^{\text{new}}(n) = S_T^{\text{old}}(n) = e(n, T^{\text{new}})$. ■

Algorithm 4.1 has the properties that only those descendents that require evaluation are re-evaluated, and no node is ever evaluated more than once in a propagation.

For implementation purposes, the statement,

$$S_T^{\text{new}}(n) \Leftarrow S_T^{\text{old}}(n)$$

need not be executed, as S_T^{new} can work directly on S_T^{old}. The two are later compared, but this is done on a per-node basis, so only the old value of the particular node need be kept.

Also, note that in the statement,

$$S_T^{\text{new}}(n) \Leftarrow e(n, T^{\text{new}})$$

the call to $e(n, T^{\text{new}})$ is potentially a recursive call to the effective value function e because the precondition and answer formulas of a node may refer to other nodes (see Definition 2.4). However, we can make use of the following property. If p is a parent of node n, then when computing $e(n, T^{\text{new}})$, we can take advantage of the fact that,

$$e(p, T^{\text{new}}) = S_T^{\text{new}}(p)$$

because p will have been previously visited by the algorithm if its effective value has changed. Therefore, further recursive calls are not necessary.

4.1.1 Running Time of Propagation

[CLR90] gives a $O(N + E)$ time algorithm for doing a topological sort, where N is the number of nodes and E is the number of edges of the graph. After doing the sort, we must iterate through each of the $O(N)$ nodes and possibly re-evaluate them. For each node marked, we must re-evaluate its precondition and answer formulas. In the worst case, we must re-evaluate the preconditions and answers of all the nodes. Let F be equal to the total size of the preconditions and answer formulas in the Thinksheet. Then the worst case running time is $O(N + E + F)$.

4.2 Optimization of Boolean Formulas

The Propagation Algorithm is efficient, but in some cases we do better. When the effective value of node n changes, we mark its children for re-evaluation. When we later visit that child, one or both of the precondition or the answer formulas depend on n. It might be the case the effective value of the dependent formula(s) stay the same even though n has changed. Can we catch some of these cases without re-evaluating a potentially long formula?

For example, suppose we had nodes $\text{A1}, \ldots, \text{An}, \text{B1}, \ldots B_n$, which we will refer to as the A nodes and the B nodes respectively. The precondition of the A nodes are

true, while the precondition of the B nodes would be as follows:

$$\forall j, 1 \leq j \leq n, \text{ precondition of } Bj \text{ is } \mathrm{A1} = j \wedge \mathrm{A2} = j \wedge \cdots \wedge \mathrm{An} = j$$

Now suppose the current state of the Thinksheet is as follows: node A1 has been answered with the value 1, while all the rest of the A nodes have been left unanswered. This means that node B1 is *possible*, while the other B nodes are *false*.

Now suppose we answer node A2 with the value 2. The propagation algorithm would require us to re-evaluate all of the B nodes, although clearly only the effective value of B1 has changed (the rest remain *false*).

Let us compute the running time for this instance. The running time for the Propagation Algorithm is $O(N + E + F)$. We will break this down in terms of N. The number of descendents of A2 is $O(N)$, so the running time simplifies to $O(N + F)$. Each precondition for the B nodes is of size $O(N)$. Since there are N nodes, the running time is $O(N^2)$.

However, with a little knowledge we would know that re-evaluating the preconditions for $\mathrm{B2}, \ldots, \mathrm{Bn}$ was unnecessary. Skipping the work for those nodes would reduce the running time to $O(N)$. In the following sections we will provide a few simple checks. If one of those checks are met then we will know that even if the parent of a formula has changed its effective value, this will not change the effective value of the formula itself.

We put this check into the Propagation Algorithm in the following way. In the original algorithm, if a parent's effective value changed, we blindly marked its children. In our new algorithm we perform the checks on the precondition and/or answer formulas of each child (depending on whether the dependency is in the precondition or answer, or both). If the checks hold, we do not mark the child, even though the parent's effective value has changed. In effect, we know that the new effective value of the parent will not modify that child, so we do not need to re-evaluate it.

The algorithm to do these checks must be fast and simple enough that it is advantageous to use it (i.e. it should be much faster than just recomputing the formula). This is not as difficult as it might seem, especially when the implementation uses an interpreted language to evaluate the formulas (as ours does). In this case, the evaluation of even a relatively small formula outweighs a few simple checks.

In the following sections we provide some easy checks that can be made for extended boolean formulas. The proofs of correctness of these checks is in Appendix A. We summarize these checks in Table 4.2.

Each of check will look at the old and new effective values of the parent and the old effective value of the formula. Simply by looking at these values (and by a possible constraint on the construction of the formula itself), we can determine if the effective value of the formula remains unchanged.

Symbol	Description
T^{old}	The old Thinksheet
T^{new}	The new Thinksheet
f	The formula being checked
p	The parent of the formula that has changed its effective value

TABLE 4.1: The symbols used when discussing the optimizations of propagation.

Simple Checks			
Check	$e(p, T^{\text{old}})$	$e(p, T^{\text{new}})$	$e(f, T^{\text{old}})$
1	*possible*	not *possible*	*true* or *false*
2	not *possible*	*possible*	*possible*

Positive Simple Checks			
Check	$e(p, T^{\text{old}})$	$e(p, T^{\text{new}})$	$e(f, T^{\text{old}})$
1	not *false*	*false*	*false*
2	*false*	not *false*	*true*

TABLE 4.2: These two tables summarize the checks for simple and positive simple formulas. The table specifies the old value and new value for the parent node p , and the old effective value of the formula f. In these cases, the formula's effective value does not change.

4.2.1 Classifying Boolean Expressions

Our checks only work for quantifier-free extended boolean formulas. In particular, we leave out terms such as $A0 * 6$, since these generally do not evaluate to a truth value. This means that the optimizations will usually apply to the preconditions of nodes, and not their answers.

We first start by identifying a subclass of the formulas to be evaluated. We call any boolean formula that is a quantifier-free logic formula constructed with $\wedge$, $\vee$ and/or $\neg$ a *simple* formula. [1] A simple formula that has no negation (i.e. it doesn't use the $\neg$ symbol) is called a *positive simple* formula.

Table 4.2 summarizes the checks for simple and positive simple formulas. The next section describe the checks in more detail and give examples of where they might come into play.

[1] While this may seem to encompass everything, our implementation has a more general language that allows loops, etc.

4.2.2 Checks for Boolean Formulas

In this section we define the checks in detail, give motivation for when the checks might be useful, and give simple examples illustrating the checks.

Since these checks will be taking place in the context of propagation, we will refer to symbols taken from the Propagation Algorithm. These symbols are given in Table 4.1.

For the purposes of definition, we assume that p is the only parent of the formula that has changed value. In reality, of course, multiple parents may have changed value. If this is the case, we go through the changed parents and apply the checks to each parent. If one parent forces a re-evaluation of the formula, then we do not have to check further.

Definition 4.2 (Simple Check 1) *If* $e(p, T^{old}) =$ possible *and* $e(p, T^{new}) \neq$ possible *and* f *is simple and* $e(f, T^{old}) =$ true *or* false *then* $e(f, T^{new}) = e(f, T^{old})$.

The motivation for this check is for the case where a reader answers a previously unanswered question, in this case node p. If any child formulas of p are already *true* or *false*, then they do not need to be re-evaluated.

Example 4.1 Suppose our Thinksheet consists of three nodes, A0, B0, and C0. The precondition of C0 is $\neg(\text{A0} = 1) \wedge \text{B0} = 2$. Currently the effective value of A0 is 1, and B0 is *possible*. Therefore, the effective value of the precondition is $\neg(\mathit{true}) \wedge \mathit{possible} = \mathit{false}$.

Now suppose the effective value of B0 changes to 2. According to our check, B0's effective value changed from *possible* to 2. C0's precondition was *false* and therefore, it should not have changed. This, in fact, is the case.

Definition 4.3 (Simple Check 2) *If* $e(p, T^{new}) =$ possible *and* $e(f, T^{old}) =$ possible *then* f *is simple and* $e(f, T^{new}) = e(f, T^{old})$.

The motivation for this check is something of the opposite of Simple Check 1. Suppose that a reader removes the answer to a particular question node (removing an answer means that answer formula for the node changes to *possible*). Then those child formulas that were originally *possible* remain *possible*.

Example 4.2 Let there be three nodes A0, B0 and C0. C0's precondition is $\text{A0} = 1 \vee \text{B0} = 2$. Suppose A0's effective value is *possible* and B0's effective value is 3. In this case, the effective value of C0's precondition would be *possible*.

Now suppose B0's new effective value is *possible*. The effective value of C0's precondition does not change.

The next two checks hold just for positive simple formulas.

Definition 4.4 (Positive Simple Check 1) *If* $e(p, T^{old}) \neq$ false *and* $e(p, T^{new}) =$ false *and* $e(f, T^{old}) =$ false *and* f *is positive simple then* $e(f, T^{new}) = e(f, T^{old})$.

This check handles cases where the parent question becomes *false*. If the child formula is already *false*, it will remain that way.

Example 4.3 Suppose our Thinksheet consists of A0, B0 and C0. The precondition of C0 is A0 = 1 ∧ B0 = 2. Suppose that the effective value of C0's precondition is already *false* because the effective value of A0 is 2.

If the effective value of B0 subsequently becomes *false*, the effective value of C0's precondition will not change.

Definition 4.5 (Positive Simple Check 2) *If* $e(p, T^{old}) =$ false *and* $e(p, T^{new}) \neq$ false *and* $e(f, T^{old}) =$ true *and* f *is positive simple then* $e(f, T^{old}) = e(f, T^{new})$.

This check is like the opposite of Positive Simple Check 1. In this case the parent question goes from *false* to not *false* (possibly because of a change to an ancestor question). If the child formula is already *true*, it does not need to be re-evaluated.

Example 4.4 Suppose we have a Thinksheet with three nodes, A0, B0 and C0. The precondition of C0 is A0 = 1 ∨ B0 = 2. Suppose that precondition of node C0 is *true* because the effective value of A0 is 1 and the effective value of B0 starts out as *false*.

If the effective value of B0 changes to another value, then the effective value of C0's precondition still won't change.

4.3 The New Propagation Algorithm

Our new algorithm for propagation takes into account the optimizations discussed above. It changes one statement of the propagation algorithm. Recall the following lines from the old algorithm:

if $S_T^{\text{new}}(n) \neq S_T^{\text{old}}(n)$ **then**
  for all $c \in n$.children **do**
    c.mark ⇐ evaluate

The algorithm modifies this statement to add the checks previously discussed:

if $S_T^{\text{new}}(n) \neq S_T^{\text{old}}(n)$ **then**
  for all $c \in n$.children where c is not marked *evaluate* **do**
    if n and the precondition and/or answer of c do not fit one of the checks in Table 4.2 **then**
      c.mark ⇐ evaluate

We look at the precondition or answer of child c only if they directly depend on node n.

n	Node Position:Title	Mark
$\Longrightarrow$	E3:*Recall Joe*	evaluate
	D4:*Joe Picture*	don't-evaluate
	E4:*Joe Remembers*	don't-evaluate
	D6:*Impeach*	don't-evaluate
	E6:*Victory*	don't-evaluate

TABLE 4.3: The topological sort of the trial strategy nodes. The first column denotes the current node in the iteration of the Propagation Algorithm.

Node Position	Precondition
E3	N/A
D4	E3 = 1
E4	E3 = 2 $\vee$ D4 = 2
D6	A3 = 1 $\vee$ C3 = 1 $\vee$ D3 = 1 $\vee$ D4 = 1 $\vee$ E4 = 1
E6	F3 = 2 $\vee$ E4 = 2 $\vee$ D3 = 2

TABLE 4.4: The preconditions of the nodes in the topological sort. The first column gives the node id, while the second column gives the precondition. E3's precondition is omitted.

4.4 A Full Example of Propagation

We will run through an example propagation using the trial strategy Thinksheet from Chapter 3 (see Figure 3.17).

To begin with, we assume that three of the questions have already been answered by the reader. The state of the sheet is given in Figure 4.2. Node D1 has been answered with 2, and node E2 with 1, and node E3 with 2. The current effective values of all the nodes are also given in the figure.

Suppose that the reader decides to change the answer of node E3 to 1. We now must propagate that change through the nodes.

The initialization of the Propagation Algorithm says we must construct a topological sort of the root nodes and their descendents. Then we mark the root node as *evaluate* and all the others as *don't-evaluate*. The result is given in Table 4.3. Table 4.4 lists the preconditions of each of these nodes for easy reference.

Starting with the loop phase of the algorithm, we re-evaluate node E3 (*Recall Joe*). Since the reader modified the answer, the new effective value changes from 2 to 1.

We now loop through all of the children of E3, which are D4 and E4. For each

	A	B	C	D	E	F
1				Crime Scene 2		
2		Know Joe False			Gun 1	
3	Menace False		Joe Says False	Not Again False	Recall Joe 2	Colt 45 False
4				Joe Picture False	Joe Rememb Possible *	
5						
6				Impeach Possible	Victory Possible	

* The precondition is true, but the question is unanswered, so its effective value is possible

FIGURE 4.2: The nodes' initial effective values

n	Node Position:Title	Mark
	E3:*Recall Joe*	(done)
$\Longrightarrow$	D4:*Joe Picture*	evaluate
	E4:*Joe Remembers*	evaluate
	D6:*Impeach*	don't-evaluate
	E6:*Victory*	don't-evaluate

TABLE 4.5: The result after completing a loop of propagation

n	Node Position:Title	Mark
	E3:*Recall Joe*	(done)
	D4:*Joe Picture*	(done)
$\Longrightarrow$	E4:*Joe Remembers*	evaluate
	D6:*Impeach*	don't-evaluate
	E6:*Victory*	don't-evaluate

TABLE 4.6: The result after completing a loop of propagation

child, we use Table 4.2 to decide whether it should be marked *evaluate* or not.

The precondition of D4 is E3 = 1 and of E4 is E3 = 2 $\vee$ D4 = 2. Both are positive simple. However, the criteria in Table 4.2 are not met, so we must mark both children as evaluate. Then we set the current node to be the next node in the topological sort, which is D4 (*Joe Picture*). The result is shown in Table 4.5.

We now re-evaluate the precondition of node D4. The precondition, which is E3 = 1, becomes *true*. The effective value of D4 is *possible*, because the question is currently unanswered. Thus D4's effective value went from *false* to *possible*.

Let us look at the children of D4, which are E4 and D6. Node E4's precondition is E3 = 2 $\vee$ D4 = 2, which is positive simple. Before the reader's modification, its value was *true*. Positive Simple Check 2 tells us that since the parent went from *false* to *possible*, and the formula was *true*, we don't have to mark it. Note, however, that E4 is already marked as *evaluate*, so nothing is gained here.

Node D6's precondition is

$$\text{A3} = 1 \vee \text{C3} = 1 \vee \text{D3} = 1 \vee \text{D4} = 1 \vee \text{E4} = 1$$

This is also positive simple. Simple Check 2 tells us that since D4 became *possible* and the precondition D6 is already *possible*, we don't have to mark it for evaluation.

We then set the current node to the next node in the topological sort, E4. The result shown in table 4.6.

Node E4's precondition is E4 = 2 ∨ D4 = 2. This evaluates to *possible* (E4 = 2 is *false*, but D4's effective value is *possible*). The children of E4 are D6 and E6. Since E4's effective value has not changed (it was originally *possible* because it was unanswered), the children are not marked.

Finally, the last two nodes in the topological sort (D6 and E6), are marked *don't evaluate*, and thus are skipped. The result of the propagation is given in figure 4.3. Node D4's precondition became *true* (and its effective value *possible*), and node E4's precondition changed to *possible* (but its effective value remained unchanged).

4.5 Optimization Experiments

In order to show the effectiveness of these optimizations described in Section 4.2, we present the results of two experiments. The first experiment tests how much time the optimizations can save. The second experiment tests how much overhead these optimizations impose—i.e. since the optimizations are checks, and if the checks fail, then the optimizations are just overhead.

All experiments were run on a Sun SPARC workstation. The central processor was a 143 MHz UltraSPARC with 92 megabytes of memory. The results of each experiment were obtained by taking the mean of 10 runs.

4.5.1 Experiment One

In this experiment, two series of nodes were created, $A_1, \ldots, A_n$ and $B_1, \ldots, B_n$. The preconditions of $A_1, \ldots, A_n$ are *true*. The answers of $A_1, \ldots A_{n-1}$ are *possible*, and the answer of A_n is n. For $1 \leq j \leq n$, the precondition of B_j is:

$$A_1 < j \wedge A_2 < j \wedge \cdots \wedge A_n < j$$

The answers of $B_1, \ldots, B_n$ are all *possible*.

In this initial state, the effective values of $B_1, \ldots, B_n$ are all *false* since their preconditions are *false* (because the effective value of A_n is n).

In our experiment, we time how long it takes to propagate changing the answer of A_{n-1} to $n-1$ with the optimization checks in effect and without them. When the optimizations checks are in effect, Simple Check 1 is satisfied for each of the preconditions of $B_1, \ldots, B_n$. and thus the preconditions are not re-evaluated. Without the optimizations the preconditions must be re-evaluated. The results are shown in Figure 4.4 for n equal to 50, 100, 150, and 200. Times are shown in milliseconds.

Because the size of the precondition formulas of $B_1, \ldots, B_n$ increases with the square of n, the time for propagation without optimization roughly follows that curve. For n equal to 200, the time is roughly 4 seconds.

With the optimization checks, the time for propagation is never more than 65 milliseconds, which is hardly noticeable for interactive use.

	A	B	C	D	E	F
1				Crime Scene 2		
2		Know Joe False			Gun 1	
3	Menace False		Joe Says False	Not Again False	Recall Joe 2	Colt 45 False
4				Joe Picture Possible *	Joe Rememb Possible	
5						
6				Impeach Possible	Victory Possible	

* The precondition is true, but the question is unanswered, so its effective value is possible

FIGURE 4.3: The nodes' effective values after propagation

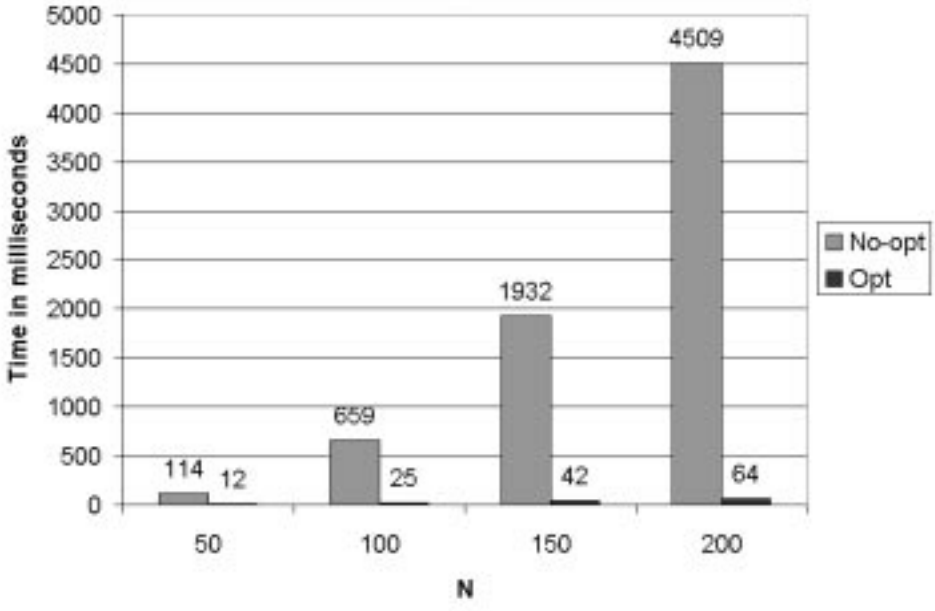


FIGURE 4.4: The results of experiment one.

4.5.2 Experiment Two

In this experiment, we again create two series of nodes, A_1 and A_2 and $B_1, \ldots, B_n$. The preconditions of A_1 and A_2 are both *true* and their answers *possible*.

For $1 \leq j \leq n$, the precondition of B_j is:

$$A_1 < j \wedge A_2 < j$$

In our experiment, we time how long it takes to propagate changing the answer of A_2 to n. In this case, the optimization checks are not satisfied, so we are measuring the overhead of the checks. The preconditions of $B_1, \ldots, B_n$ are small (involving just two terms) to test for any significant overhead in the checks.

The results are shown in Figure 4.5 for n equal to 50, 100, 150 and 200. Times are shown in milliseconds. The overhead in this case is extremely minimal—there is only a one millisecond difference when n is equal to 200.

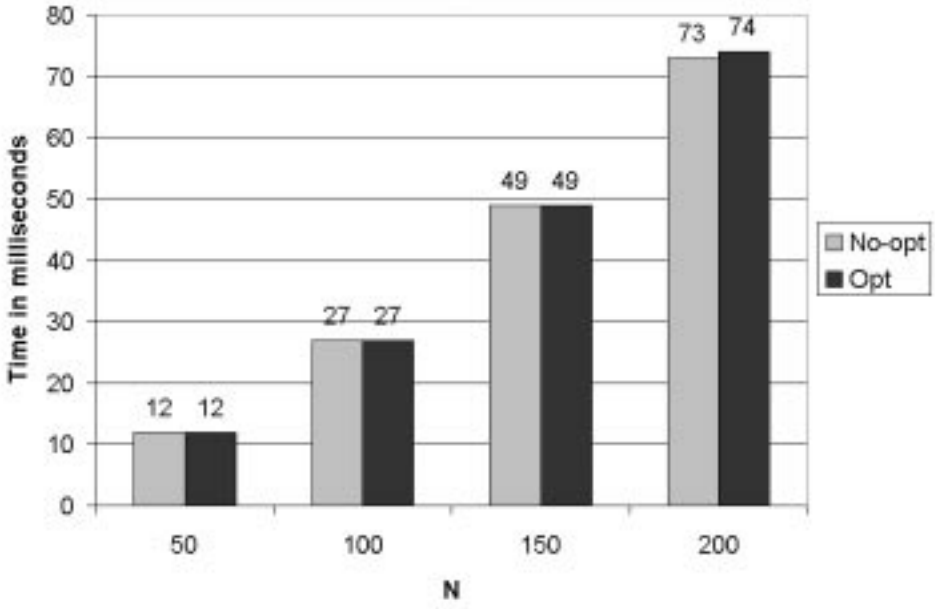


FIGURE 4.5: The results of experiment two.

Chapter 5

Thinksheet Implementation

This chapter provides an overview of the design and implementation of Thinksheet. Thinksheet has a core library designed to be attached to a user interface. The core library is designed to be as user interface independent as possible, and is discussed in Section 5.1. Section 5.2 and 5.3 discuss the implementation of two of the user interfaces to the core library, the spreadsheet and World Wide Web interfaces respectively.

5.1 The Core Thinksheet System

This section gives an overview of architecture of the core library. Appendix E contains a reference to the Application Programming Interface (API).

The core library is implemented in C++. The four main classes of the library are the `Node`, `SheetGraph`, `Parser` and `Value`.

The principle objects are called `Node`s. Each object of this class represents one Thinksheet node. A `Node` contains an answer and precondition field as specified in the model in Chapter 2, but it also contains additional information such as fields for the title, the question (for question nodes), or hypertext (for text nodes).

A `Node` also holds the list of the parent nodes of its precondition and answer fields, and the list of the children nodes that reference it in their preconditions and answers. The class has functions to set and get each of the fields. For example, the function `setAnswer` will set the answer field. There is also a function for retrieving the effective value of a `Node`.

The `SheetGraph` stores all the `Node`s of a particular Thinksheet. It has functions for saving and loading a Thinksheet to and from a file system. It also has functions to retrieve nodes that have certain characteristics. For example, one can retrieve the set of all nodes that have a *true* precondition.

The `Parser` parses and evaluates the precondition and answer formulas of the nodes. It recognizes the language described in Section 3.3.1. When it parses a formula, it returns a `Value`. A `Value` may be any of the types allowed by the Thinksheet language

(e.g. a logic value, a floating point, a string, or sets of these values). The implementation of the `Parser` class is loosely based on Hoc [KP84].

A user interface communicates with these classes when modifying or retrieving information from a Thinksheet. For example, if the interface captures a request to set the answer of a particular `Node`, it would call the `setAnswer` function on the `Node` class. When that function has finished, the interface might retrieve the effective value of that `Node`. This would return a `Value` which the interface would display to the user.

The core library contains classes that represent the interface's view of the `Node` and `SheetGraph`. They are the `Cell` class and the `SheetWindow` class respectively. When some aspect of a `Node` changes, the core library notifies the `Cell`(s) associated with the `Node`. Similarly, when a `SheetGraph` changes, the `SheetWindow` is notified. The core library contains the declaration of the `Cell` and `SheetWindow` class, but they must be implemented by the user interface.

We may associate multiple `Cell`s with a `Node` and multiple `SheetWindow`s to a `SheetGraph`, allowing multiple views of the same object. This association between a single `Node` or `SheetGraph` and its multiple views is handled by the `CellController` and `SheetWindowController` respectively.

For example, the `Cell` class might implement a button on the spreadsheet interface. If something about a `Node` changes (e.g. its effective value), then the associated `Cell` is notified, and the button is updated (e.g. it might change color).

Similarly, the `SheetWindow` class might implement the entire grid on which the `Cell`s are placed. When something in the `SheetGraph` changes (e.g. a node is added or deleted to the Thinksheet), then the `SheetWindow` is notified, and the grid is updated (e.g a `Cell` is added or removed).

The `CellController` and the `SheetWindowController` would allow the user interface to have multiple grids open at the same time. This might be useful if one wanted to have a reader mode and writer mode of the same Thinksheet open at the same time.

We present an example interaction between a user interface and the core library. This interaction represents the primary process of Thinksheet—that of the reader setting the answer to a node.

Example 5.1 This example shows the flow of control when a user sets the answer of a node. We discuss what happens at each step of the process.

1. The user interface captures the user's request to set the answer of a node N1. The interface calls the `setAnswer` function of the `Node` class (see Figure 5.1).

2. A topological sort of the node and its descendents is constructed and the node is marked with *evaluate* (see Figure 5.2). Then the Propagation Algorithm is run on the topological sort. The algorithm will iterate through the topological sort and any node that is marked *evaluate* will be re-evaluated, and then its children possibly marked *evaluate* (see Chapter 4 for details).

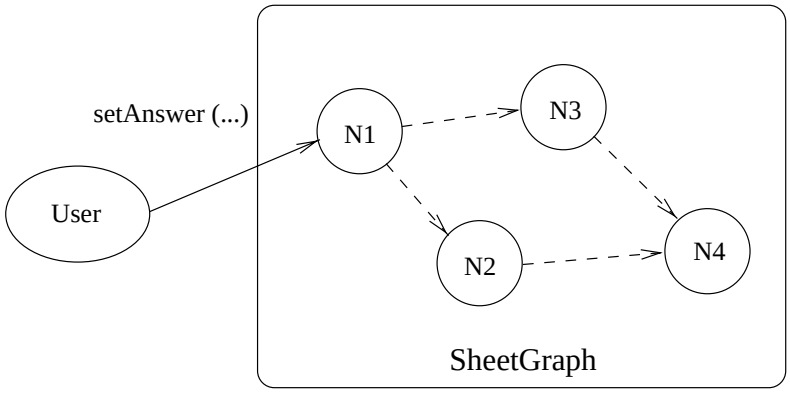


FIGURE 5.1: The user sets the answer to node N1. Solid lines represent function calls. Dashed lines represent node dependencies.

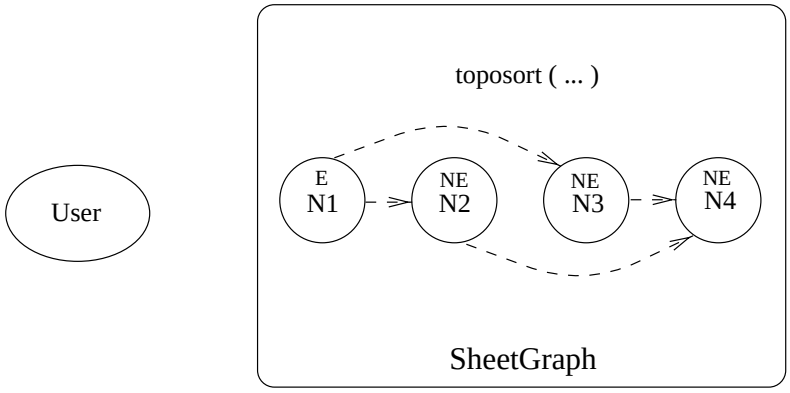


FIGURE 5.2: The nodes are topologically sorted. Those that are marked *evaluate* are labeled with an *E*. Those that aren't are labeled with an *NE*.

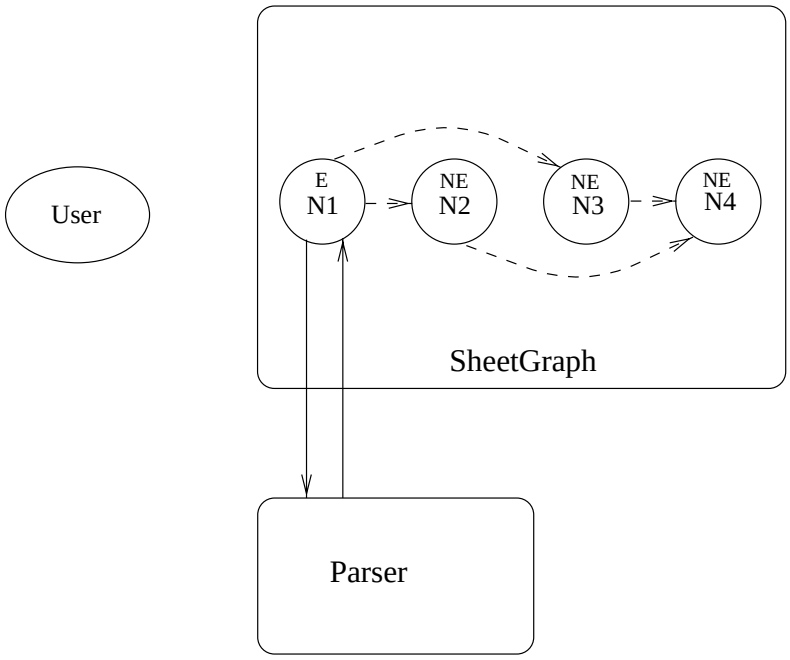


FIGURE 5.3: The nodes that are marked *evaluate* are re-evaluated by calling the `Parser`. The `Parser` returns a `Value`.

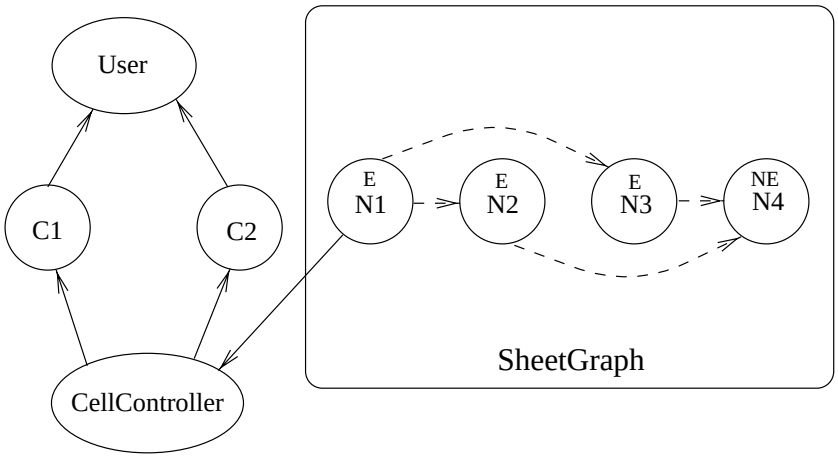


FIGURE 5.4: Node N1 updates through the `CellController` and marks its children with *evaluate*. The `CellController` notifies the two `Cells` associated with the node, C1 and C2.

```
Carbon Dioxide: %calc% A0 * $1 %endcalc%
```

(a) The Smartfield for B0, with a dependency on node A0 and $1.

```
Nitrous Oxide: %calc% A1 * $1  %endcalc%
```

(b) The Smartfield for B1, with a dependency on node A1 and $1.

```
The pollution level is:
%insertcontents% B0(0.4) %endinsertcontents%
%insertcontents% B1(0.5) %endinsertcontents%
```

(c) The Smartfield for B2, with a dependency on B0 and B1.

FIGURE 5.5: Smartfields and lazy evaluation.

3. Since the first node is marked with *evaluate*, it is re-evaluated by calling the `Parser` to interpret its precondition and/or answer formula. The `Parser` returns the values of these formulas in the form of `Value` objects. These `Value`s are used to compute the new effective value of the node (see Figure 5.3).

4. If the effective value of a node has changed, then the `CellController` for that node is notified. The `CellController`, in turn, notifies each of the `Cell`s. The `Cell`s display their results to the user (see Figure 5.4).

5.1.1 Smartfield Processing

Smartfields are text fields that contain directives to allow the dynamic creation of text based on the values of nodes in a Thinksheet (see Section 3.3.2). The directives are processed when a reader requests the text. The title, question and text fields of a `Node` are all Smartfields. This subsection discusses how Smartfields are processed.

Smartfields are implemented by a `SmartField` class in the core library. An object of this class stores the original unprocessed text containing the directives either in memory or in a file. When the user interface wants the processed text, it calls the appropriate method of the `SmartField` class, which substitutes the directives and then returns the processed text. There are also functions to view and modify the original text of the Smartfield, for when a writer is creating a Thinksheet.

Lazy Evaluation of Smartfields

Smartfields can have quite complicated directives. We would like to save time by not doing the processing when we don't have to. In order to save time, we adopt a lazy evaluation strategy for Smartfields.

For example, Figures 5.5(a)–(c) show three Smartfields. In this example, the Smartfield of B0 depends on the current value of node A0 and the parameter, $1. Likewise, the Smartfield of node B1 depends on the current value of A1 and $1.

The first time we request the the text of node B2, all three Smartfields must be processed. However, suppose the A0's effective value changes, but A1's stays the same, and then we re-request the text of node B2. In this case the Smartfield for B1 need not go through the processing step again because its text will not change regardless of how we change A0.

We would like to catch these and similar cases, and only process Smartfields when we have to. A Smartfield needs to reprocessed when something it depends on changes. The "things" that a Smartfield may depend on are:

1. Nodes—for `%calc%` directives, etc.
2. Other Smartfields—for `%insertcontents%`.
3. Parameters (such as $1)—for Smartfields that have been called by `%insertcontents%`.

We call these objects the *parents* of a Smartfield. If any of them have changed, then the Smartfield must be reprocessed.

Because of a Smartfield's ability to insert other Smartfields, we must create a form of Smartfield propagation. For example, the contents of B2 (Figure 5.5(c)) doesn't immediately depend on node A0. But this Smartfield does depend on the contents of B0, which in turn depends on node A0. So changing the value of node A0 means that we must reprocess the contents of B2. Therefore, a Smartfield graph must be created. Unlike a Thinksheet graph, a Smartfield graph may have cycles (because a Smartfield may recursively insert the contents of itself).

5.2 Implementation of the Spreadsheet Interface

The spreadsheet GUI described in Section 3.1 is implemented in C++ and uses Motif, which is a library that contains graphical user interface components such as buttons, scrollbars, etc. [Hel91]. We use and extend a C++ framework built around the Motif library [You92].

The framework represents each GUI component as a C++ class. For example, a GUI button might be implemented as an object of the `Button` class. Applications extend from this framework in order to customize the components.

Additionally the framework provides a structure for defining commands to be executed when the user interacts with the graphical components. All commands extend from the `Cmd` class provided by the framework. A command is associated with one of the framework's GUI component classes. For example, a particular type of `Cmd` might

be associated with a `Button`. When that `Button` is clicked, the `Cmd` is executed (by calling its `execute` function).

The spreadsheet interface maps objects of the `Cell` class and objects of the `SheetWindow` class to graphical components on the screen. An object of the `SheetWindow` class represents the entire spreadsheet window that is presented to the user. An object of the `Cell` class represents one button in the spreadsheet grid. Thus a button maps to one node of a Thinksheet, and a grid maps to the entire Thinksheet.

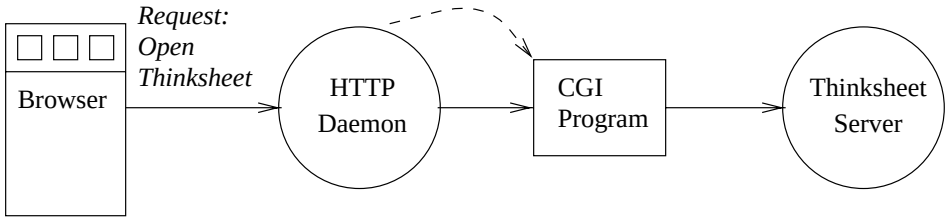
When a user clicks on a `Cell` the action taken depends on whether the interface is in the reader mode or the writer mode. Associated with each `Cell` is an object of the `ClickCmd` class which executes the appropriate action. For example, if the Thinksheet is in reader mode and the reader has clicked on a question cell, the `ClickCmd` object will open up a dialog window with the question and an entry to allow the reader to input his answer. If the Thinksheet is in writer mode, the `ClickCmd` object would open a dialog box allowing the user to edit the fields of the `Node` associated with the `Cell`.

5.3 Implementation of the CGI World Wide Web Interface

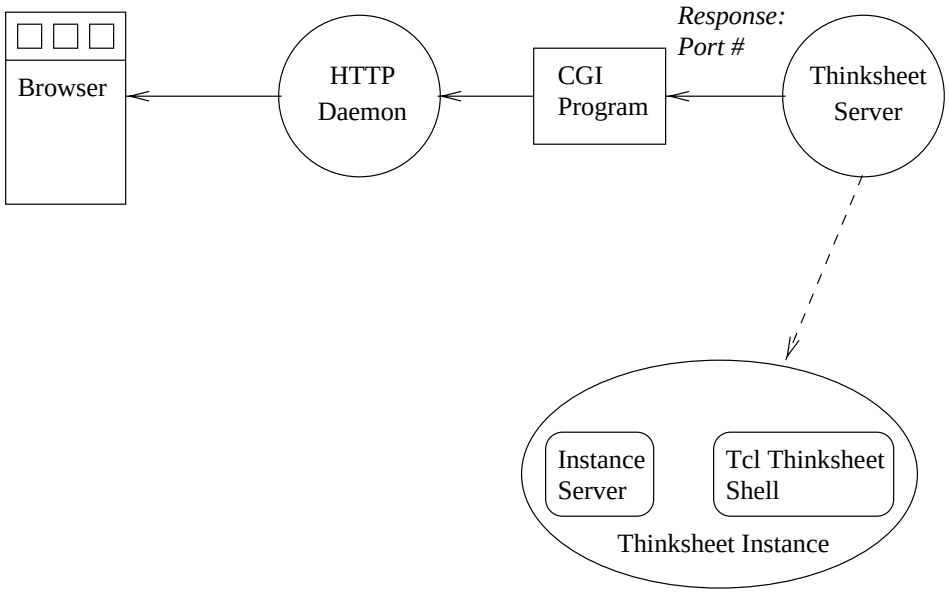
The implementation of the CGI World Wide Web interface can be broken down into several processes. They are:

1. The *browser*. This is the client from which the user starts the interaction.
2. The *HTTP daemon*. The is the Web server that handles requests in the HTTP protocol.
3. The *Thinksheet server*. This server handles requests to start a Thinksheet.
4. A *Thinksheet instance*. An instance of a running Thinksheet. In the current implementation, an single instance is actually broken down into two processes:
   (a) An *Instance server*. The server listens on a socket for requests. It forwards the requests to:
   (b) A *Tcl Thinksheet shell*. This shell receives requests from an instance server and sends the results back.

When a browser client wants to start a Thinksheet session, it typically sends a CGI request to the residing HTTP daemon, which executes the appropriate CGI program to contact the Thinksheet server (see Figure 5.6(a)). The Thinksheet server forks off a new Thinksheet instance, and sends back a port number to the browser (see Figure 5.6(b)). The port number identifies the UNIX socket the Instance server will listen to for requests.



(a) The client request



(b) The Thinksheet server response

FIGURE 5.6: The initial interaction when starting a Thinksheet in the WWW interface.

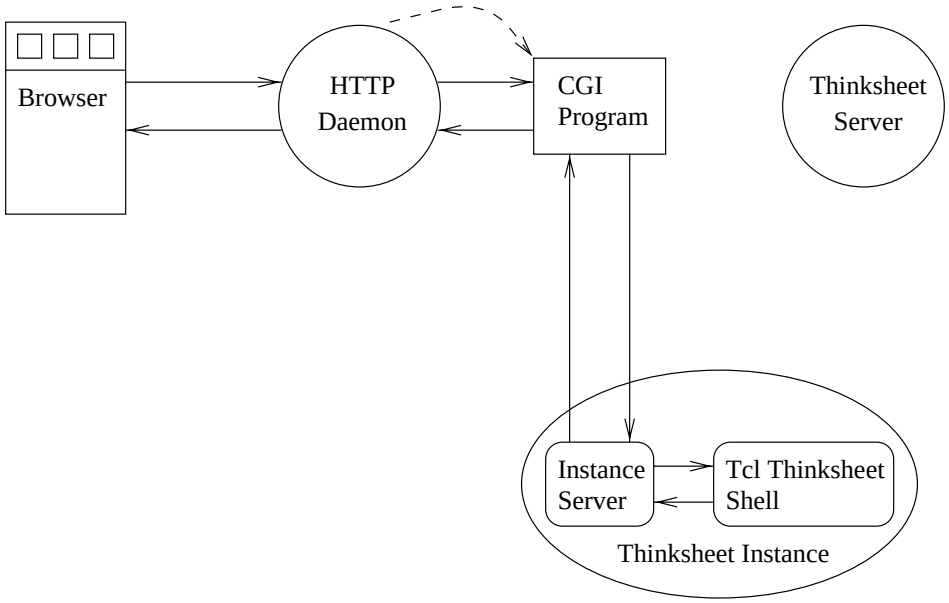


FIGURE 5.7: The interaction between the Browser and the Thinksheet instance.

After this, the interaction is between the Browser and the Thinksheet instance. The browser sends a CGI request to the HTTP daemon. Once again, the daemon executes the appropriate CGI program. One of the parameters that the browser sends in its request is the port number the Thinksheet instance is listening to. The CGI program contacts the Thinksheet instance using this port number. The CGI program then sends the instance's reply back to the browser.

The Thinksheet server and a particular Thinksheet instance may be contacted via any program that can connect to UNIX sockets, so these tools are not limited to interactions between HTTP daemons and browsers (although that was their original design goal).

Chapter 6

Thinksheet and Metadata

This chapter discusses using Thinksheet as an application to store metadata. Thinksheet is an appropriate tool for storing certain types of metadata. Its underlying data representation is semi-structured, but it also has access to relational tables.

The primary use of Thinksheet would be to store metadata about how to issue correct queries. We would like to do what a human database consultant might do, as exemplified by the following imaginary dialogue:

Researcher: "I want to know the relationship between salmon births and mud erosion."

Consultant: "Which region, year, season, type of salmon ... ?"

Researcher: "The Columbia River headwaters, 1970-1999, ... "

Consultant: "Ok. Go to table XYZ and issue the third query on the menu specifying salmon under Species and mud erosion under Environmental Cause."

As attractive as this scenario is to the researcher, it suffers from the limited number of such consultants and the fact that even the best consultant may not be aware of every new piece of data that enters the system.

The goal therefore is to create a Thinksheet that acts as a limited consultant to answer the researcher's queries on how to query the data. Possibilities include using Thinksheet to store canned queries, useful joins, etc.

6.1 A Metadata Model for Thinksheet

Thinksheet can serve as a kind of expert system over the data, helping the researcher find the right table, and even helping to form queries over the table once the data is found.

In Chapter 2 we discussed the usefulness of the meta-values of *false* and *possible* to avoid giving values to inapplicable attributes. This general principle will be useful in defining a metadata structure over databases.

6.1.1 Tables as Values for Nodes

In our examples of Thinksheets so far, node values have generally been either strings or numbers. However, there is nothing in the model given in Chapter 2 that disallows other value types. Of particular interest in the realm of metadata would be to have nodes representing tables. In this case, instead of a node being bound to the value of a string or a number, its value is bound to a table—i.e. its answer formula is a table.

We consider the domain of node to be a table, and the various operations on these tables may be the ones given by the relational algebra—e.g. σ (select), π (project), etc. [Ull88]. We may also include some optional predicates on tables—e.g. a predicate to test whether a table includes a particular tuple.

For example, table T may represent a table of students at university X. Node N may have as its answer formula the selection of students in the table in T that have a GPA greater than 3.5.

If the reader of this Thinksheet were not interested in university X then precondition of N might be *false*, and the reader would never see the result. The advantage of this approach is that the underlying data can now be stored in any common relational database system.

6.1.2 Querying a Table

Once we can access tables in a Thinksheet, we would like to establish ways for readers to query them. We present here one way to construct these queries. It is, of course, not the only way to do things, but it is general enough to apply to a wide variety of cases, and illustrates the principles well.

Our method is to create a node representing each attribute of the table. The reader will form his query by answering these nodes. We also create a node which represents the query. We set the answer formula of this node to the query string. We will call this node the *result* node because it will contain the result of the reader's query. This result will be in the form of the table (as discussed in the previous section).

The attribute nodes may be answered with values belonging in the attributes domain. For example, if a node represented the attribute *State*, it may take on the value of any subset of {Alabama, ... , Wyoming}. Then we create a *result* node which selects from the table those rows which satisfy the reader's answers. For example, if the reader had selected {New Jersey, New York} as the states he was interested in, the result node would hold only those rows corresponding to New Jersey and New York.

To generalize this notion, suppose we had a table T with attributes $A_1, \ldots, A_k$. Let $\mathcal{P}(S)$ equal the *power set* of a set S. Then, through a slight abuse of notation, we call $\mathcal{P}(\pi_A T)$ the power set of attribute A in table T (meaning the set of all subsets of the set of possible values of A). Then we create nodes $N_1, \ldots N_k$ in a Thinksheet S such for all j, $1 \leq j \leq k$:

$$D(N_j, S) = \mathcal{P}(\pi_{A_j} T) \tag{6.1}$$

where $D(N_j, S)$ represents the domain function as described in Section 2.7.

Thus the answer of node N_j can be any subset of the values of attribute A_j. The answer formula of the result node would be the following:

$$\sigma_{\bigwedge\limits_{1 \le i \le k} A_i \in N_i} T$$

where,

$$\bigwedge_{1 \le i \le k} A_i \in N_i \; = \; A_1 \in N_1 \; \wedge \; A_2 \in N_2 \; \wedge \; \ldots \; \wedge \; A_k \in N_k$$

However, we have the slight problem that the selection formula follows our three-valued logic. For example, if the reader has not answered a particular node, N_i, then its effective value would be *possible*. This would mean that the selection formula would contain the atom $A_i \in$ *possible*, which means that for some tuples the entire formula might evaluate to *possible*

Most database systems do not understand our three-valued logic, and therefore do not understand the meaning of *possible*. When the selection formula evaluates to *possible* for a particular tuple, we must decide whether to include it in the result or not. We will follow the convention that the tuple is included, under the premise that it *might* be what the reader wants and therefore should be in the result.

Also note that a particular tuple might have *Null* values for certain attributes. However, this has no effect on our selection formula (conventionally, any comparison with *Null* would yield *false*).

Example 6.1 Suppose we had the simple two column table as follows:

A_1	A_2
1	2
2	4
3	6

If we wanted to issue queries on the table, we could create two nodes, N_1 and N_2 to represent attributes A_1 and A_2. The domain for N_1 would be, for example:

$$D(N_1, S) = \mathcal{P}(\pi_{A_1} T)$$

Our result node, N_R would have as its answer field the following formula:

$$\sigma_{A_1 \in N_1 \wedge A_2 \in N_1} T$$

If the effective value of node N_1 was $\{1, 2\}$, and of N_2 was *possible* ,then the resultant effective value for N_R is the evaluation of the following formula:

$$\sigma_{A_1 \in \{1,2\} \wedge A_2 \in possible} T$$

The result is the following table.

A_1	A_2
1	2
2	4

6.1.3 Mutual Restriction

Unfortunately, mapping multiple nodes to attributes results in a loss of information. We may pick values for $N_1, \ldots, N_k$ such that no row in T satisfies these constraints. Thus the result node would be empty.

Without prior knowledge of the table, we would have no knowledge of which values are "good" in the sense that they actually select some rows.

One solution around this problem is the concept we call *mutual restriction*. This means that the domain of a particular node is restricted to those values which correspond to already answered nodes. For example, if we had two attributes, *State* and *Capital*, and the *State* node was already answered with {New Jersey, New York}, then the domain of the *Capital* node would be restricted to the power set of {Trenton, Albany}

We generalize this idea by saying that the domain of each node is set in the following way. For all j, $1 \leq j \leq k$:

$$D(N_j, S) = \mathcal{P}\left(\pi_{A_j}\sigma_{\bigwedge\limits_{\substack{1\leq i\leq k\\ i\neq j}} A_i \in N_i} T\right) \tag{6.2}$$

The difference between this equation and equation 6.1 is that here the domain is restricted by the choices of the other attributes, (this is represented by the $\bigwedge_{1\leq i\leq k, i\neq j} A_i \in N_i$ in the selection.)

Example 6.2 Suppose we had the same Thinksheet from Example 6.1. Now suppose that node N_1 has been answered with {1, 2}. Using mutual restriction, the domain for node N_2 would be:

$$\mathcal{P}(\pi_{A_2}\sigma_{A_1\in\{1,2\}}T)$$

In this case, the resultant domain is {{}, {2}, {4}, {2, 4}}.

Example 6.3 Table 6.1 contains some sample data about movies. Each row contains information about the movie title, its rating, year of release, and genre. The movies listed are a small sample of movies from the 1980's. The information in this table was taken from The Internet Movie Database [Int].

To represent this in Thinksheet, assume that M represents the table in Table 6.1. We then create a node for each attribute, N_T, N_R, N_Y, and N_G. Initially, when the

Title	Rating	Year	Genre
Attack of the Killer Tomatoes	PG	1980	Horror
Beastmaster	PG	1982	Fantasy
Commando	R	1985	Action
The Road Warrior	R	1981	Sci-Fi
Yor, the Hunter from the Future	PG	1982	Sci-Fi

TABLE 6.1: A table with movie data.

nodes are unanswered, the domain of each node is a projection of the corresponding attribute. For example, the domain of N_G is $\mathcal{P}(\{\text{Action, Fantasy, Horror, Sci-Fi}\})$.

Mutual restriction restricts these domains as the reader answers the nodes. For example, the domain of the node representing Genre would be as follows,

$$D(N_G, M) = \mathcal{P}(\pi_{\text{Genre}}\sigma_{\text{Title}=N_T \wedge \text{Rating}=N_R \wedge \text{Year}=N_Y} M)$$

If the reader answers node N_G with {Fantasy, Action}, then the new domain of N_T would be $\mathcal{P}(\{\textit{Highlander II}, \textit{Commando}\})$.

Preconditions on Tables

So far, we have shown how to incorporate tables into the model of Thinksheet, but we have not used any of Thinksheet's power to help expressiveness.

One way to add information is to use preconditions on tables to express when the tables themselves are relevant. For example, movies fall within the realm of entertainment. In Example 6.3 we presented a table with some movie information. We might have tables for other types of entertainment, for example museums, or theater. We can imagine create a node N_E which asks *Which type of entertainment are you interested in?* If the reader is interested in museums, we don't want to present information about the movies. We can therefore make the precondition of nodes N_T, N_R, N_Y, N_G, which represent the attributes of the table, $N_E = \textit{Movies}$.

Thus, if the reader is not interested in looking at movies, these nodes will disappear.

Example 6.4 Suppose we had a database containing the movie table from Example 6.3 and added another table containing information about museums. This table is shown in Table 6.2. Let *Mo* and *Mu* represent the movie and museum tables respectively.

Now we would also like to list the location and showtimes of the movies in Table 6.1. Therefore we create a table with theater information and another table which lists the showtimes for those theaters (see Tables 6.3 and 6.4). Let *Sh* and *Th* represent the showtime and theater tables respectively.

Name	Location	Type
American Museum of the Moving Image	Queens	Film
American Museum of Natural History	Upper West Side	Science
The Liberty Science Center	New Jersey	Science
Metropolitan Museum of Art	Upper East Side	Art
Museum of Modern Art	Midtown	Art

TABLE 6.2: A table with museum data.

Name	Location
American Cinemas	Midtown
Mega SuperPlex	Upper West Side
Super MegaPlex	Upper East Side

TABLE 6.3: A table with theater information.

Theater Name	Movie Title	Time
American Cinema	Attack of the Killer Tomatoes	12:00am
Mega Superplex	Beastmaster	6:20pm
Mega Superplex	Beastmaster	9:20pm
Super Megaplex	Commando	4:00pm
Super Megaplex	Commando	6:00pm

TABLE 6.4: A table with showtimes.

We would like to integrate this information in some way. For example, a reader might be interested in all of the museums and theaters nearby, or he might be interested in the showtimes for all films in the genre of Horror, or some combination of multiple constraints.

We can start by creating a node asking the reader which type of entertainment they are interested in, in this case either movies or museums, or maybe even both. We call that node N_E. We then create attribute nodes for each table. The precondition of the nodes representing movie attributes would be $N_E = \textit{movies} \vee N_E = \textit{both}$ and museum attributes $N_E = \textit{museums} \vee N_E = \textit{both}$. We could then create result nodes with query formulas as discussed in earlier examples.

Once the reader has expressed an interest in movies, he might also be interested in theaters and showtimes. Therefore the preconditions for the attribute nodes for these two tables will also be $N_E = \textit{movies} \vee N_E = \textit{both}$.

The other thing a reader might like to do is query the showtimes based on his answers about movies and theaters. We create a query node whose answer field is the following formula:

$$\pi_{\text{Mo.Title, Th.Name, Sh.Time}}\sigma_F(\mathit{Mo} \bowtie \mathit{Th} \bowtie \mathit{Sh})$$

where

$$\begin{aligned} F = \quad & \text{Mo.Title} \in N_T \ \wedge \ \text{Mo.Genre} \in N_G \wedge \cdots \\ & \wedge \text{Th.Name} \in N_N \ \wedge \ \text{Th.Location} \in N_L \ \wedge \cdots \end{aligned}$$

where $N_T, N_N, N_L, \ldots$ represent the appropriate attributes.

This query restricts the showtimes to those movies or theaters that the reader has requested. For example, if the reader had answered N_L (Theater Location) with *Upper East Side*, and left the rest blank, the result would be:

Theater Name	Movie Title	Time
Super Megaplex	Commando	4:00pm
Super Megaplex	Commando	6:00pm

The reason is that Super Megaplex is the only theater on the Upper East Side.

Another useful feature we can add to this Thinksheet is to link the location of both theaters and museums to a single node (in our normal breakdown, there would be two nodes representing theater location and museum location). This is useful because then a reader can look for both museums and movies that are at the same location without redundantly answering two nodes.

A typical interaction with the reader might run something like this. The reader only wants to go somewhere close by, so he specifies his location—e.g. Upper West Side—in the location node. He is not sure whether he wants to see a movie or go to a museum, so he answers node N_E with *both*. If he does go to a movie, he is only interested in fantasy movies, so he answers the node representing movie genre with

Fantasy. He can then look at result nodes for showtimes and for museums and get the relevant information. In this example, he would find that the American Museum of Natural History is the only museum and *Beastmaster* is the only fantasy film nearby.

6.2 Related Work

Our work with Thinksheet and metadata focuses on the problem of finding the right data set, as opposed to integrating data from various sources [CH96, Maz97].

Typical applications for metadata include annotating scientific data sets [CJ96, DCMG97]. These annotations include such information as the quality of the data, the types of instruments used to obtain the data, etc., in addition to information useful for finding particular data sets (e.g. location, time of measurements, and so forth).

The metadata framework described here most closely resembles the framework described in [GST98]. In this paper, Galhardas, et al, propose a metadata framework based on first-order logic. Data and queries are represented as logic sentences. This framework can easily be modeled in Thinksheet. The advantage of using Thinksheet is that it is easier for a reader to issue queries by answering questions rather than by constructing logic sentences.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The world is full of complex documents. Countless hours are spent reading and writing such documents. Those who give in to frustration normally consult experts, such as lawyers, tax consultants and so on.

Our thesis presented Thinksheet, which is a tool for easing the task of reading and writing complex documents. We have presented the Thinksheet model, the interfaces to the system, and discussed the implementation and algorithms behind it.

Thinksheet helps readers of complex documents by tailoring the document based on the reader's answers to questions. Doing this filters out the irrelevant information and provides the text tailored to the reader. It helps writers of complex documents by allowing them to specify conditions under which each portion of the document is relevant.

While Thinksheet makes reading and writing complex documents easier, we believe that there is always room for improvement. The next section discusses future work.

7.2 Future Work

7.2.1 User Interfaces

This subsection discusses ways the user interface to Thinksheet can be enhanced. It first discusses possible enhancements to the reader interface and then to the writer interface.

Reading a Thinksheet

The spreadsheet interface may not be the best reader interface for all applications of Thinksheet. The interface has particular problems when the number of cells becomes quite large, in which case the reader has a difficult time finding the right cell. We could either develop tools to deal with this difficulty or try another paradigm.

In general, we believe that there is no one best interface for all the applications that Thinksheet covers, hence the motivation for creating Tcl interface to Thinksheet. Tcl is an interpreted language with access to graphical user interface functions, which allows the creation of custom interfaces for specific applications. We would like to see further exploration of different kinds of interfaces for Thinksheet, especially for large Thinksheets.

Writing a Thinksheet

Even with Thinksheet's help, writing a complex document is a difficult and sometimes time consuming task.

Preconditions are based on boolean logic, which is known to be inherently difficult for people to understand [WS94]. However, this work generally deals with using boolean logic to query the data (i.e. as a reader application). Here, preconditions as boolean logic formulas form part of the data itself. More study needs to be done to see if their are any hidden issues when boolean logic is used as data.

Our experience with users who write Thinksheets show there are some tools that could make their job easier. Particularly useful would be a set of tools to manipulate and query the dependencies formed from the preconditions of nodes. For example, writers might like to search and replace all occurrences of the term B0 $>$ 5000 with B0 $>$ 10000 because of some change to B0. Also, writers might like to see the path of dependencies between two nodes n_1 and n_2 (i.e. all paths in a Thinksheet graph starting with node n_1 and ending with n_2).

7.2.2 Querying a Thinksheet

Sometimes we would like to form *queries* over a Thinksheet without being restricted to simply answering the questions. For example, suppose a prosecutor was developing a trial strategy much like the example shown in Chapter 3. The prosecutor would be very interested in knowing if there were ever any cases where the defendant would be found not guilty. If the prosecutor had a node in his Thinksheet representing *Not Guilty* he might like a list of all possible question/answer sessions that led to that result. Doing this by hand would be tedious. Instead we would like a query language that would allow us to specify such constraints.

Such a query language is described in [Tan98]. The query language allows queries of the type "Enumerate the answers to questions that would make these two nodes *true*," or "Give the list of *true* nodes under the condition that the answer to this node is greater than 10,000 and the answer to this other node is less than 5,000."

Such a language would be useful for both the reader and the writer. Readers can now query a Thinksheet without being restricted to the path laid out by the writer. Writers can check the consistency of their Thinksheets.

Unfortunately, if a Thinksheet contains general boolean precondition and answer formulas, answering these queries is intractable. However, for a useful subclass

of formulas, [Tan98] has found efficient algorithms . Future work includes refining the algorithms and implementing the query system.

7.2.3 New Applications

While this thesis has focused on using Thinksheets for complex documents such as laws and billing plans, other types of complex documents exist. For example, a website can be considered a complex document. We could use Thinksheet to tailor a website for a particular user, bringing to mind such applications as web purchasing (e.g. displaying warm clothes to those people who say they are from Alaska) and customer service (e.g. to help a person with technical information about a particular computer hard drive).

As we enter the information age, the word *document* does not have to mean text alone. Typically, an electronic document now contains a variety of multimedia, such images, video and sound. For example, one could imagine using Thinksheet to tailor video (e.g. suppose a person only wants to see the action sequences from one of the movies listed in Chapter 6). The video could be broken up into chunks and tagged with preconditions, much like a textual document.

In general, Thinksheet can be used for complex *information*. Chapter 6 has already discussed how Thinksheet can also be used for metadata. There has also been work on using Thinksheet for workflow management [Sim98]. Thinksheet is useful as long as the information shares the characteristics discussed in Chapter 1:

1. It is large and structurally complex.

2. Only a small portion applies to a particular information worker.

Appendix A

Proofs of Optimizations in Propagation

This appendix gives the proofs for the checks listed in Section 4.2. Let f be the formula being checked, and let p be a parent node of the formula. In each case, assume that p is the only parent that has changed its effective value. Let T^{old} and T^{new} be the old and new Thinksheets. Then if $e(f, T^{\text{new}}) = e(f, T^{\text{old}})$, we say that "the effective value of f has not changed."

Proof of Simple Check 1

Theorem A.1 *If* $e(p, T^{old}) =$ possible *and* $e(p, T^{new}) \neq$ possible *and* f *is simple and* $e(f, T^{old}) =$ true *or* false *then* $e(f, T^{new}) = e(f, T^{old})$.

Proof We will give our proof by induction of the formation of the precondition formulas.

1. **Base case:** Formula f is a single atom—a predicate of the form $p(t_1, t_2, \ldots, t_n)$, where
   $t_1, t_2, \ldots, t_n$ are terms.

   Originally the parent's effective value was *possible* and the atom's value may have been either *true* or *false*. However, it can't have been *true* (because by Definition 2.4 we take the minimum of truth values when evaluating atoms containing truth values), so it must have been *false*.

   There must be at least one other truth value in the phrase that is *false* (if not, then the atom's value would be *possible*). Therefore changing the value of the parent from *possible* to something else does not have an effect on the atom's value.

2. **Induction:** Let f be a formula constructed by either $f_1 \wedge f_2$ or $f_1 \vee f_2$ or $\neg f_1$, where f_1 and f_2 are formulas. The old effective value of f was either *true* or *false*.

   Suppose first that p is in only one of f_1 or f_2. Without loss of generality, assume that f_1 contains p. If $e(f_1, T^{\text{old}}) =$ *true* or *false* then by the induction hypothesis,

f_1 does not change its value. Since nothing has changed in f_2 its value does not change either, so the effective value f does not change.

However, if $e(f_1, T^{\text{old}}) = \textit{possible}$ then f_1 might change its value. For the case of $f = f_1 \wedge f_2$ and $e(f_1 \wedge f_2, T^{\text{old}}) = \textit{false}$, then $e(f_2, T^{\text{old}})$ must be *false*. Since f_2 hasn't changed the effective value of the whole formula stays the same. Note that $e(f_1 \wedge f_2, T^{\text{old}})$ can not be *true* because we assume that f_1 evaluates to *possible*.

For the case of $f = f_1 \vee f_2$ and $e(f_1 \vee f_2, T^{\text{old}}) = \textit{true}$, then $e(f_2, T^{\text{old}})$ must be *true* and therefore changing f_1 will not change the new effective value. Note again that $e(f_1 \vee f_2, T^{\text{old}})$ can not be *false*.

Finally f_1 can not be *possible* in the case of $f = \neg f_1$ because that would violate our condition.

Now suppose that p is in both f_1 and f_2. We look at both case separately.

(a) $f = f_1 \wedge f_2$.
If both f_1 and f_2 are either *true* or *false*, then the induction hypothesis holds and neither will change value. However, it might be the case that one is *possible* and the other *false* (the other can not be *true* because then f would be *possible*, which violates our assumption). Without loss of generality, assume that $e(f_1, T^{\text{old}})$ is *possible* and $e(f_2, T^{\text{old}})$ is *false*. Then, by the induction hypothesis, $e(f_2, T^{\text{new}}) = e(f_2, T^{\text{old}})$ will not change. Therefore $e(f, T^{\text{new}})$ will remain *false* no matter what the value of $e(f_1, T^{\text{new}})$ is.

(b) $f = f_1 \vee f_2$.
If both $e(f_1, T^{\text{old}})$ and $e(f_2, T^{\text{old}})$ are either *true* or *false*, then the induction hypothesis holds and neither will change value. However, it might be the case that one is *possible* and the other *true* (the other can not be *false* because then $e(f, T^{\text{old}})$ would be *possible*, which violates our assumption). Without loss of generality, assume that $e(f_1, T^{\text{old}})$ is *possible* and $e(f_2, T^{\text{old}})$ is *true*. By the induction hypothesis, f_2 will not change. Therefore $e(f, T^{\text{new}})$ will be *true* no matter what the value of $e(f_1, T^{\text{new}})$ is. ■

Proof of Simple Check 2

To prove Simple Check 2, we first give the following lemma.

Lemma A.1 *For a simple formula f, if $e(p, T^{new}) =$ possible then $e(f, T^{new}) =$ possible or $e(f, T^{new}) = e(f, T^{old})$.*

Proof We give our proof by induction on the formula.

1. **Base case:** f is an atom.

$e(f_1, T^{\text{old}})$	$e(f_2, T^{\text{old}})$	$e(f_1 \wedge f_2, T^{\text{old}})$	$e(f_1 \wedge f_2, T^{\text{new}})$
true	*true*	*true*	*possible*
true	*possible*	*possible*	*possible*
true	*false*	*false*	*false*
false	*true*	*possible*	*possible*
false	*possible*	*possible*	*possible*
false	*false*	*false*	*false*

TABLE A.1: Case by case analysis of the proof of Lemma A.1 for $f_1 \wedge f_2$. In each of these cases $e(f_1, T^{\text{new}}) = \textit{possible}$ and $e(f_2, T^{\text{new}}) = e(f_2, T^{\text{old}})$. We leave out the cases that $e(f_1, T^{\text{old}}) = \textit{possible}$ because then $e(f_1, T^{\text{new}}) = e(f_1, T^{\text{old}})$ and then $e(f_1 \wedge f_2, T^{\text{new}}) = e(f_1 \wedge f_2, T^{\text{old}})$.

By Definition 2.4, an atom's effective value is the minimum of any truth values it contains. If the $e(f, T^{\text{old}}) = \textit{true}$, then it can't contain any *false* truth values. Therefore, is $e(p, T^{\text{new}}) = \textit{possible}$, then $e(f, T^{\text{new}}) = \textit{possible}$.

If the $e(f, T^{\text{old}}) = \textit{false}$, the atom either contained at least one *false* truth value, or no truth values whatsoever. If it contained a *false* value, then $e(f, T^{\text{new}}) = \textit{false}$. If not, $e(f, T^{\text{new}}) = \textit{possible}$.

2. **Induction:** Let f be a formula constructed by either $f_1 \wedge f_2$ or $f_1 \vee f_2$ or $\neg f_1$, where f_1 and f_2 are formulas.

   Suppose p is in only one of f_1 or f_2. Without loss of generality, assume that the parent is contained in f_1. Then $e(f_2, T^{\text{new}}) = e(f_2, T^{\text{old}})$. By the induction hypothesis, either $e(f_1, T^{\text{new}}) = \textit{possible}$, or $e(f_1, T^{\text{new}}) = e(f_1, T^{\text{old}})$. In the latter case, then $e(f, T^{\text{new}}) = e(f, T^{\text{old}})$.

   However, if the new effective value of f_1 is *possible*, then the effective value of f might change. We show that it might either change to *possible*, or keep its original value, by listing each possible case for f in the following tables:

   (a) $f = f_1 \wedge f_2$. The cases are listed in Table A.1.

   (b) $f = f_1 \vee f_2$. The cases are listed in Table A.2.

   (c) $f = \neg f_1$. If $e(f_1, T^{\text{new}}) = \textit{possible}$, then $e(\neg f_1, T^{\text{new}}) = e(f, T^{\text{new}}) = \textit{possible}$.

   Now suppose that p is in both f_1 and f_2. By the induction hypothesis, the new effective values for f_1 and f_2 will either be the same as the old effective values or become *possible*. If both remain the same, then $e(f, T^{\text{new}}) = e(f, T^{\text{old}})$. If both become *possible*, then $e(f, T^{\text{new}}) = \textit{possible}$, since:

$$e(\textit{possible} \wedge \textit{possible}, T^{\text{new}}) = e(\textit{possible} \vee \textit{possible}) = \textit{possible}$$

$e(f_1, T^{\text{old}})$	$e(f_2, T^{\text{old}})$	$e(f_1 \vee f_2, T^{\text{old}})$	$e(f_1 \vee f_2, T^{\text{new}})$
true	*true*	*true*	*true*
true	*possible*	*true*	*possible*
true	*false*	*true*	*possible*
false	*true*	*true*	*true*
false	*possible*	*possible*	*possible*
false	*false*	*false*	*possible*

TABLE A.2: Case by case analysis of the proof of Lemma A.1 for $f_1 \vee f_2$. In each of these cases $e(f_1, T^{\text{new}}) = \textit{possible}$ and $e(f_2, T^{\text{new}}) = e(f_2, T^{\text{old}})$. We leave out the cases that $e(f_1, T^{\text{old}}) = \textit{possible}$ because then $e(f_1, T^{\text{new}}) = e(f_1, T^{\text{old}})$ and then $e(f_1 \vee f_2, T^{\text{new}}) = e(f_1 \vee f_2, T^{\text{old}})$.

If only one of them becomes *possible*, but the others new effective value equals its old effective value, then we have the same result as our previous analysis where we assumed that p is in only one of f_1 or f_2. ■

We now prove the second check.

Theorem A.2 *If* $e(p, T^{new}) =$ possible *and* f *is a simple formula and* $e(f, T^{old}) =$ possible *then* $e(f, T^{new}) = e(f, T^{old})$.

Proof We will give our proof by induction of the formation of the precondition formulas.

1. **Base case:** A single atom.

   If $e(f, T^{\text{old}}) = \textit{possible}$, and parent $e(p, T^{\text{new}})$ becomes *possible*, then (because we have a single atom), $e(f, T^{\text{new}}) = \textit{possible}$.

2. **Induction:** Let f be a formula constructed by either $f_1 \wedge f_2$ or $f_1 \vee f_2$ or $\neg f_1$, where f_1 and f_2 are formulas and $e(f, T^{\text{old}}) = \textit{possible}$.

   If $f = \neg f_1$, then because $e(f, T^{\text{old}}) = \textit{possible}$, it also must be the case that $e(f_1, T^{\text{old}}) = \textit{possible}$. By the induction hypothesis $e(f_1, T^{\text{new}}) = \textit{possible}$, so therefore $e(f, T^{\text{new}}) = e(f, T^{\text{old}})$.

   This leaves the cases where $f = f_1 \wedge f_2$ or $f = f_1 \vee f_2$. Suppose that p is in only one of f_1 or f_2. Without loss of generality, assume that f_1 contains p. Therefore $e(f_2, T^{\text{new}}) = e(f_2, T^{\text{old}})$.

   By Lemma A.1, either $e(f_1, T^{\text{new}}) = e(f_1, T^{\text{old}})$ or $e(f_1, T^{\text{new}}) = \textit{possible}$. In the former case, then $e(f, T^{\text{new}}) = e(f, T^{\text{old}})$.

   If $e(f_1, T^{\text{new}}) = \textit{possible}$ and $e(f_1, T^{\text{new}}) \neq e(f_1, T^{\text{old}})$ then we have the following cases:

(a) $f = f_1 \wedge f_2$.
Since $e(f_1 \wedge f_2, T^{\text{old}}) = \textit{possible}$ the old effective values of one or both of f_1 and f_2 must be *possible*. Since we are assuming that the effective value of f_1 has changed, then $e(f_2, T^{\text{old}}) = \textit{possible}$. Then, if $e(f_1, T^{\text{new}}) = \textit{possible}$, then

$$e(f_1 \wedge f_2, T^{\text{new}}) = e(\textit{possible} \wedge \textit{possible}, T^{\text{new}}) = \textit{possible}$$

Therefore $e(f, T^{\text{new}}) = e(f, T^{\text{old}})$.

(b) $f = f_1 \vee f_2$.
Similarly, since $e(f_1 \vee f_2, T^{\text{old}}) = \textit{possible}$, the old effective values of one or both of f_1 and f_2 must be *possible*. Since we are assuming that the effective value of f_1 has changed, then $e(f_2, T^{\text{old}}) = \textit{possible}$. Then, if $e(f_1, T^{\text{new}}) = \textit{possible}$, then

$$e(f_1 \vee f_2, T^{\text{new}}) = e(\textit{possible} \vee \textit{possible}, T^{\text{new}}) = \textit{possible}$$

Therefore $e(f, T^{\text{new}}) = e(f, T^{\text{old}})$.

Suppose that p is in both f_1 and f_2. Since $e(f, T^{\text{old}}) = \textit{possible}$ at least one of f_1 or f_2 must have an effective value of *possible*. Without loss of generality, assume that it is f_1. By Lemma A.1, $e(f_1, T^{\text{new}}) = \textit{possible}$ or $e(f_1, T^{\text{new}}) = e(f_1, T^{\text{old}})$. In this case both hold (because $e(f_1, T^{\text{old}}) = \textit{possible}$). Since, $e(f_1, T^{\text{new}}) = e(f_1, T^{\text{old}})$, we can use the same analysis as if p was a parent for only one of the formulas. ■

Proof of Positive Simple Check 1

Theorem A.3 *If* $e(p, T^{old}) \neq$ false *and* $e(p, T^{new}) =$ false *and* f *is positive simple and* $e(f, T^{old}) =$ false *then* $e(f, T^{new}) = e(f, T^{old})$.

Proof We give the proof by induction on the formation of the precondition formula.

1. **Base case:** A single atom

   When the parent becomes *false*, the atom becomes *false*. Since the atom is already *false*, we can leave it unchanged.

2. **Induction:** Let f be a formula constructed by either $f_1 \wedge f2$ or $f_1 \vee f_2$.

   Suppose p is in only one of either f_1 or f_2. Without loss of generality, assume that f_1 contains p. Therefore $e(f_2, T^{\text{new}}) = e(f_2, T^{\text{old}})$. If $e(f_1, T^{\text{old}}) = \textit{false}$ then by the induction hypothesis $e(f_1, T^{\text{new}}) = \textit{false}$, so therefore $e(f, T^{\text{new}}) = e(f, T^{\text{old}})$. In t case of $f = f_1 \vee f_2$, this must be the case since both f_1 and f_2 must be *false* in order for the whole formula to be *false*.

   In the case of $f = f_1 \wedge f_2$, $e(f_1, T^{\text{old}})$ may not be *false*, but then $e(f_2, T^{\text{old}})$ must be, so the entire formula still does not change value.

Now suppose f is both f_1 and f_2. In order for the old effective value of f to be *false*, the old effective value of one or both of f_1 or f_2 must be *false*. Assume $e(f_1, T^{\text{old}}) = \mathit{false}$. Then by the induction hypothesis, $e(f_1, T^{\text{new}}) = \mathit{false}$. Since the effective value of f_1 has not changed, we can use our analysis for the case where p is in only one of either f_1 or f_2. ■

Proof of Positive Simple Check 2

Theorem A.4 *If* $e(p, T^{old}) = \text{false}$ *and* $e(p, T^{new}) \neq \text{false}$ *and and* f *is positive simple and* $e(f, T^{old}) = \text{true}$ *then* $e(f, T^{old}) = e(f, newT)$.

Proof We give the proof by induction on the formation of the precondition formula.

1. **Base case:** A single atom

   If the parent $e(p, T^{\text{old}})$ is *false*, then $e(f, T^{\text{old}})$ can not be *true* (because the effective value of the atom is the minimum of its truth values), so it holds true vacuously.

2. **Induction:** Let f be a formula constructed by either $f_1 \wedge f2$ or $f_1 \vee f_2$.

   Suppose p is in only one of either f_1 or f_2. Without loss of generality, assume the p is in f_1. Therefore $e(f_2, T^{\text{new}}) = e(f_2, T^{\text{old}})$. If $e(f_1, T^{\text{old}}) = \mathit{true}$ then by the induction hypothesis, $e(f_1, T^{\text{new}}) = e(f_1, T^{\text{old}})$, and therefore $e(f, T^{\text{new}}) = e(f, T^{\text{old}})$.

   In the case of $f = f_1 \wedge f_2$, $e(f_1, T^{\text{old}})$ must be *true*, since both parts of the conjunction must be *true*.

   In the case of $f = f_1 \vee f_2$, $e(f_1, T^{\text{old}})$ may not be *true*, but then $e(f_2, T^{\text{old}})$ must be, so the entire formula still does not change value. ■

Appendix B

The Thinksheet Language

This appendix serves as a reference to the Thinksheet language.[1] It does not attempt to give a rigorous definition of the language. Rather it is meant as a user reference, with explanations of the various language constructs, plus many examples.

The Thinksheet language is relatively simple, but it is still more complex than the first order logic model given in Chapter 2. The complete language includes user-defined functions, local variables, and the ability to execute shell commands. The results from the simplified model, however, are more or less the same.

General Rules for the Language

1. Spaces, tabs and carriage returns are considered whitespace.
2. Two consecutive slash characters `//` are used to comment out text. The rest of the line following `//` is ignored.
3. The precondition and answer fields of nodes consist of statements and/or expressions.
4. Each statement and expression should end with a semi-colon. The exception is the last line; here it is not necessary. (Therefore a one-statement precondition need not have any semicolons.)
5. The return value of the last non-empty statement or expression of the precondition field is the value that is assigned to the precondition of the node; similarly for the answer.
6. The parser returns a value of *true* for an empty precondition field and *possible* for an empty answer field.

[1] This appendix is based on a earlier document written by Minna Cha. Much of the implementation of the Thinksheet language was done by Dao-i Lin.

The Available Types

1. Integers, floating point numbers (including negative numbers)

   Examples:

   - `7`
   - `4.5`
   - `-2`

2. Strings

   - Strings must be enclosed within double quotes.

   Examples:

   - `"yikes"`

3. Logic Values

   - There are three logic values in the Thinksheet language, corresponding to *true*, *false*, and *possible*. They are written as:
     - `ATRUE`
     - `AFALSE`
     - `APOSSIBLE`

4. Discrete sets

   - Members of a set must be either all integers or all strings.
   - Members are separated by commas.

   Examples:

   - `[1,3,5,10]`
   - `["abc","ick"]`
   - `["avs",5,6]`—not valid

5. Ranges

   - Ranges are inclusive (they include the endpoints).
   - They can have both an upper and lower bound or just one with `..` used to signify negative or positive infinity.
   - Ranges can be floating point numbers or strings.

Examples:

- `1..5`—all floating point numbers between 1 and 5, inclusive
- `4.5..`—all floating point numbers great than or equal to 4.5.
- `"b".."ca"`—all strings alphabetically between `"b"` and `"ca"` including `"b"` and `"ca"`.
- `.."be"`—all strings alphabetically before `"be"` including `"be"`.
- `4.."c"`—not valid

Variables

The Thinksheet language has three different types of variables. Nodes are one type of variable and are global to the Thinksheet. The other two types are local to a statement block. They are local variables, and local arrays.

1. Node identifiers

   A node identifier is the column and row number of the node in the spreadsheet. They have the following properties:

   - They are used in expressions to refer to the node's effective value.
   - They are case insensitive.
   - The value of a node identifier is the effective value of the node that's being referred to.
   - A node's effective value is based on its precondition and answer (see Chapter 2.)

   Examples:

   - `B10`

2. Local variables

   - These are local to a given statement block—i.e. a node's precondition or answer field, or inside of a function definition.
   - They must start with an underscore.
   - `:=` is used to assign values to local variables.
   - the left side of `:=` is the local variable being assigned the value while the right may be any expression.
   - The return value of the assignment statement is the value of the expression.

   Examples:

- `_ten   :=10;`
- `_fooval:="Foo";`
- `_b5val :=b5;`

Note that the assignment operator `:=` is used only to assign values to local variables and not to nodes. A node can only be assigned a value through its answer and precondition fields. The value of each field is determined by the return value of the last statement or expression of the field. The value (or effective value) of a node is based on the values of the two fields.

3. Arrays

   First we declare a variable as an array using the `array` declaration.

   ```
   array _arrayname[bound1, bound2, ...];
   ```

   Like local variables, arrays are local to a statement block.

   Example:

   `array _X [3,4,5];`

   This creates a variable `_X` which is a 3 dimensional array, with bounds 3 by 4 by 5.

   You can use an array as a variable.

   Examples:

   - `_X[0,1,2] := 3;`
   - `_X[1,2,4] := "string";`
   - `_Y := _X[0,1,2] + 1;`
   - `_X[A0,2,_Y] := 1;`

Expressions

Here we discuss the different types of expressions in the Thinksheet language.

Except for the predicates *or*, *and*, *not*, any operand that has one or more truth values as its argument has the minimum of the truth values as its result (as defined in Chapter 2).

1. Arithmetic operators:

   `+ - * / ^`

   addition, subtraction, multiplication, division, exponent.

2. Arithmetic operands:

   integers/floating points (`-6.7`), singleton numerical sets (`[5]`), logical values (`ATRUE`), and any variables which evaluate to any of the above.

   Examples:

   - `[4] * 5;`—20
   - `6 * AFALSE;`—AFALSE
   - `6 + b6;`—is okay as long as `b6` evaluates to a number or a singleton numerical set.

3. Relational operators:

   - `X > Y` is *true* when $\exists x \in$ `X` and $y \in$ `Y` such that $x > y$.
   - `X >= Y` is *true* when $\exists x \in$ `X` and $y \in$ `Y` such that $x \geq y$.
   - `X < Y` is *true* when $\exists x \in$ `X` and $y \in$ `Y` such that $x < y$.
   - `X <= Y` is *true* when $\exists x \in$ `X` and $y \in$ `Y` such that $x \leq y$.
   - `X == Y` (or `X = Y`) is *true* when $\exists x \in$ `X` and $y \in$ `Y` such that $x = y$.
   - `X != Y` (or `X ! Y`) is *true* when $\exists x \in$ `X` and $y \in$ `Y` such that $x \neq y$.

   If `X` or `Y` are singletons, then they are treated as singleton sets.

4. Relational operands:

   integers and floating points (`-6.7`), strings (`"hello"`), numerical sets (`[5,6,7]`), string sets (`["be","me","sigh"]`), numerical ranges (`6..9.7`), string ranges (`"lulu".."zzz"`), logical values (`APOSSIBLE`) and any variables which evaluate to any of the above.

   Examples:

   - `10.6 > 4`—*true*
   - `1.6..3 > 2..4`—*true*
   - `[1,2,3] > 2`—*true*
   - `[1,2,3] > [1,2]`—*true*
   - `[1,2,3] > 1..2`—*true*
   - `"abc".. > ["fg"]`—*true*

5. Set containment operators:

   - `X [==] Y` (or `X [=] Y`) is *true* when `X` and `Y` are identical as sets.
   - `X [!=] Y` (or `X [!] Y`) is *true* when `X` and `Y` are not identical.

- `X [>] Y` is *true* when `X` contains `Y` and `X [!=] Y`.
- `X [>=] Y` is *true* when `X` contains `Y`.
- `X [<] Y` is *true* when `Y` contains `X` and `X [!=] Y`.
- `X [<=] Y` is *true* when `Y` contains `X`.

6. Set containment operands:

   integers and floating points (`-6.7`), strings (`"hello"`), numerical sets (`[5,6,7]`), string sets (`["be","me","sigh"]`), numerical ranges (`6..9.7`), string ranges (`"lulu".."zzz"`), logical values (`APOSSIBLE`), any variables which evaluate to any of the above.

   In a containment expression, at least one of the operands must evaluate to a set. The other operand may be any of the above as long as numerical-valued operands are not mixed with string-valued operands.

   Examples:

   - `[1,2,3] [>=] 1..3`—*false*
   - `[1,2,3] [<=] 1..3`—*true*
   - `[1,2,3] [<] 1..3`—*true*
   - `[1,2,3] [>] [1,2,3]`—*false*
   - `[1,2,3] [>=] [1,2,3]`—*true*
   - `[1,2,3] [>] 2`—*true*
   - `.."xyz" [>] ["a","b"]`—*true*
   - `2 [>=] 2`—not valid.

7. Logic operators:

   - not: `~`
   - and: `&`, `&&`, `and`
   - or: `|`, `||`, `or`

8. Logic operands:

   logical values (`AFALSE`, etc.), and any variables which evaluate to logical values.

   Examples:

   - `(b5>3 || a2<2) & b2=4`
   - `~(b7=6) and c4=[6,9,2]`

Note that `a2 != 3` and `~(a2 = 3)` are slightly different. In the case that the effective value of `a2` is *false*, then `a2 != 3` would evaluate to *false*, while `~(a2 = 3)` would evaluate to *true*.

9. Conditional operator:

   (logical exp) ? exp1 : exp2 : exp3

   - The `(logical expression)` may be relational, containment, or boolean or any variable evaluating to a boolean; `exp1`, `exp2` and `exp3` may be any expression.
   - The logical expression is enclosed in parentheses followed by a question mark followed by three expressions separated by colons.
   - When the value of the logical expression is:
     - *true* then `exp1` is returned
     - *possible* then `exp2` is returned
     - *false* then `exp3` is returned

   Examples:

   - `(a4<0) ? 0 : -1 : a4`
   - `(b4="yuck") ? [1,2,3] : APOSSIBLE : [10,12];`
     Only valid if `b4`'s answer is not a numerical value since clearly a string value is expected.
   - `(5<6 & 6<3) ? "weird" : "weirder" : answer(b4);`
   - `(60<0) ? AFALSE : APOSSIBLE : ATRUE;`
     true
   - `((4<0) ? 2 : 20 : 200 ) * 40;`
     8000
   - `((b4=0) ? 1 : 0 : -1) >  0;`
     true if the effective value of `b4` is 0 or *true*.
     false if the effective value of `b4` is not 0, or if it is *possible* or *false*.

Built-in Functions

Thinksheet has several built-in functions available. We list them in terms of category: date functions, logical tests, string functions and math functions.

Date Functions

- `thisyear()`—returns current year.
- `thismonth()`—returns current month in range 1 to 12.
- `thisdayofmonth()`—returns current date in range 1 to 31.
- `thisdayofweek()`—returns current day of week in range 1 to 7, with 1 being Sunday.

  These functions have the following properties:

- They take no arguments.
- They return a numerical value in the ranges specified.

  Example:

```
thisyear() + thismonth()/100;
```

  For July 1998, this will give the result 1998.07.

Logical Tests

- `true(<logical exp>)`—returns *true* if the value of the argument is *true*, *false* otherwise
- `possible(<logical exp>)`—returns *true* if the value of the argument is *possible*, otherwise *false*.
- `false(<logical exp>)`—returns *true* if the value of the argument is *false*, otherwise *false*.

  `<logical exp>` must evaluate to a logical constant.
  These functions will return *true* or *false* (note they never return *possible*).
  Examples:

- `AFALSE=AFALSE`—*false*
- `APOSSIBLE=APOSSIBLE`—*possible*
- `AFALSE!=ATRUE`—*false*
- `false(AFALSE)`—*true*
- `false(7000<0)`—*true*

- `true(6)`—*false*, if the argument had been a node whose value was 6, it would be *false*
- `true(false(AFALSE))`—*true*
- `possible(possible(b5))`—*false*

Pure Precondition and Pure Answer

- `precondition(nodeid)`—returns the value of the precondition field of the node referred to by the argument.
- `answer(nodeid)`—returns the value of the answer field (not necessarily the effective value) of the node referred to by the argument.
- `answered(nodeid)`—returns *true* if `nodeid` has a *true* precondition and has an answer, *false* otherwise.

The functions have the following properties:

- They take as arguments node identifiers.
- The return values of the functions depend is the value of the particular field (*not* the effective value of the node).
- Note that `answer()` will return *possible* if the answer field of the node is empty, but will not necessarily return *possible* or *false* if the precondition() of the node is *possible* or *false*.

Examples:

- `answer(b3) + 10`
- `possible(precondition(b5)) || possible(answer(b5))`
- `~(precondition(b7))`

A typical use for `answered()` is within a Smartfield. For example:
`%if% answered(c3) %then% .... %endif%`

String functions

Thinksheet has a function for formating numbers into strings and for concatenating expressions into a single string.

- `string(<exp>, <total length>, <decimal length>)`

  The expression `<exp>` must evaluate to a number (either a floating point or an integer). `<total length>` specifies the entire length of the number, while `<decimal length>` specifies the number of digits after the decimal point.

  Example:

  - `string(a4, 10, 2)`
    This returns the string representing the value of `a4` with two numbers past decimal point and total length 10.

- `concatenate(arg1, arg2, ... argN)`

  This function returns a string resulting from the concatenation of arguments `arg1` though `argN`. The arguments may be any expressions.

  Example:

  - `concatenate("Hello, ", B4, ", your tax is ", C3 * 7)`

Math Functions

Thinksheet has the following math functions available:

- `sin(x)`—returns $\sin x$.
- `cos(x)`—returns $\cos x$.
- `atan(x)`—returns $\arctan x$.
- `log(x)`—returns $\ln x$.
- `log10(x)`—returns $\log_{10} x$.
- `log2(x)`—returns $\log_2 x$.
- `exp(x)`—returns e^x.
- `sqrt(x)`—returns $\sqrt{x}$.
- `int(x)`—returns x with the truncation of any fractional part.
- `abs(x)`—returns the absolute value of x.

Statements

1. `if` statement:

   - ```
     if (logical exp)
         statement1;
     ```
   - ```
     if (logical exp) {
         st1; st2; ...;
     };
     ```
   - ```
     if (logical exp)
         statement1
     else
         statement2;
     ```
   - ```
     if (logical exp) {
         st1; st2; ...;
     } else {
         st3; st4; ...;
     };
     ```

   Note the final semicolon.

   - `logical exp` must evaluate to a boolean.
   - If there is more than one statement/expression in either the `if` or `else` part, then semicolons must separate each statement and/or expression and also the set must be enclosed in curly brackets.
   - If the logical expression evaluates to *true*, then the first (set of) statement(s) and/or expression(s) is executed; if the logical expression evaluates to *false* or *possible*, then the second (set of) statement(s) and/or expression(s) is executed.
   - The return value is the last statement or expression in the `if` part or `else` part, depending on which (set of) statement(s) and/or expression(s) is executed.
   - The `if` statement may be used without the `else` part in which case, if the logical expression evaluates to *true*, then the return value is the last statement or expression of the `if` part; otherwise *false* or *possible* will be returned depending on the value of the logical expression.

   Examples:

   - `if (a2 >4) a2 else 4;`
   - `if (b5=4) a1=4 else (a2=0 & b2=0);`
     This would be used where a logical value is expected.

2. `while` statement:

   - `while (logical exp) statement;`
   - ```
     while (logical exp) {
         st1; st2; ...;
     };
     ```

   Note the final semicolon.

   - the logical expressions may be relational, containment, or boolean; the statements may also be expressions.
   - if there is more than one statement/expression in the `while` statement, then semicolons must separate each and the set must be enclosed in curly brackets.
   - the logical expression is evaluated and if it evaluates to *true* then the (set of) statement(s) and/or expression(s) is executed; then the logical expression is evaluated again; looping until the logical expression evaluates to *false*.
   - the return value will be *false* since the last operation for the `while` statement is the logical test and it must be *false* for the `while` statement to stop.

   Examples:

   - `_x := 2; while (_x <= 20) _x := _x^2; _x;`

User-Defined Functions

To define a function in the Thinksheet language, use the following syntax.

```
func _name()
{
    st1; st2; ...;
    return value;
};
```

To call a function, use the following syntax:
`_name(arg1, arg2, ...);`

- Like variables, the name of the function must start with an underscore.
- To define a function, the keyword `func` is followed by the function's name followed by a set of parentheses, followed by the statements/expressions which are separated by semicolons and enclosed in curly brackets if there is more than one.
```

- To call a function to be executed, the function name is followed by the arguments in order, separated by commas, enclosed in parentheses.
- Unlike variables, functions are not local; they are defined for the whole Thinksheet.
- Functions may take arguments; they are referred to in the body of the function as $1, $2, ...
- The `return` statement returns from a function. It consists of the keyword `return` followed by the value that is to be returned by the function.

  Examples:

- Definition:

```
func _ack()
{
    if ($1=0)
                return $2+1;
    if ($2=0)
                return _ack ($1-1,1);
    return _ack ($1-1,_ack($1,$2-1);
};
```

  Call:

```
_ack (0,0);
```

- Definition:

```
func _add() return $1+10;
```

  Call:

```
_add(2);
```

Appendix C

Smartfield Directives

This appendix lists the available processing directives for Smartfields. For an introduction to Smartfields, see Section 3.3.2.

Smartfield directives are bracketed by the percent symbol (%). A directive is fully delimited, meaning it has a begin symbol and an end symbol. All directives take the form of `%<directive>% ... %end<directive>%`. The available directives are listed below in alphabetical order.

Several directives *concatenate* their arguments into a single string. Arguments are separated by spaces, except for quoted arguments, which are delimited by the start and end quotes. If the arguments to a directive are `arg1`, `arg2` through `argN`, then the *concatenation* of these arguments is equivalent to the result of:

```
concatenate(arg1, arg2, ..., argN)
```

where `concatenate` is the command in the Thinksheet language which concatenates its arguments into a single string.

The end result of concatenation is that unquoted arguments are interpreted in the Thinksheet language and replaced with their values. Quoted arguments are not interpreted. For example, suppose the effective value of node B1 were 3. Then the concatenation of the following:

```
"Testing 1, 2, " B1*2
```

would result in

```
Testing 1, 2, 6
```

Note that there can be no spaces in the expression `B1*2`. Otherwise the single expression will be treated as multiple arguments.

The following section lists the available directives. The section after that demonstrates how to use one of the directives to connect to a database. The last section

demonstrates how to use Smartfields to modify a running Thinksheet in response to a reader's answers.

Listing of Directives

- `%calc% <formula> %endcalc%`

  This directive allows calculations to be inserted into the static text. These calculations are written in the Thinksheet language described in Appendix B. For example:

  ```
  Your taxes are %calc% A0 * .3 %endcalc%.
  ```

  If the effective value of A0 were 60,000, the result of processing this text would be

  ```
  Your taxes are 18000
  ```

- `%comment% <text> %endcomment%`

  All `<text>` in between the comment directive is ignored.

- `%globalpreamble% <preamblename> %endglobalpreamble%`

  A particular Thinksheet may have several predefined preambles associated with it (see `%preamble%` for an explanation of what a preamble is). Each predefined preamble has its own name. The `%globalpreamble%` directive takes the code associated with `<preamblename>` and processes it like a normal preamble.

- `%if% <formula> %then% <text> %endif%`

  If the formula represented by `<formula>` is *true* or *possible*, then insert `<text>` into the resultant text of the Smartfield. The expression in `<formula>` is written in the Thinksheet language.

- `%importfile% <arg1> <arg2> ... <argN> %endimportfile%`

  Concatenate `<arg1>`, `<arg2>`, through `<argN>` into a single string. This string is taken as a filename representing a report file. Thinksheet merges the report with the current Thinksheet. The report file is in the format specified in Section 3.3.4. See `%importfile%` and `%postcondition%` below for an example of the use of this directive.

- `%include% <arg1> <arg2> ... <argN> %endinclude%`

  Concatenate `<arg1>`, `<arg2>`, through `<argN>` into a single string. This string is taken as a filename. The text in this file is appended into the Smartfield directly. The included text is not processed.

Example:

```
%include% "/tmp/file" A0 %endinclude%
```

If the current effective value of A0 is 3, then this will include the text from the file `/tmp/file3`.

- `%insertcontents%` `<node>(<arg1>, <arg2>, ..., <argN>)` `%endinsertcontents%`

  Inserts the Smartfield of node `<node>` passing arguments `<arg1>` through `<argN>`, *only* if the effective value of `<node>` is not *false*. The node's Smartfield is processed and then inserted into the resultant text.

- `%outputcontents%` `<filename>, <nodeid1>, <nodeid2>, ... <nodeidn>` `%endoutputcontents%`

  Destroys and replaces the file named `<filename>` with the processed contents fields of the list of node ids.

- `%postcondition%` `...` `%endpostcondition%`

  This directive is available only when the Smartfield is the question field for a node. When a reader answers the question, all of the text between the directive is processed as a Smartfield. This is useful for executing actions based on the reader's answer, for example, via the `%system%` directive (see below). See the section on `%importfile%` and `%postcondition%` below for an example of how to use this directive.

- `%preamble%` `<program>` `%endpreamble%`

  The preamble defines variables to be used later. Inside of the preamble any code in the Thinksheet language may be written. This code is executed when the Smartfield is processed and any local variables that are defined by the code may be used elsewhere in the Smartfield.

  Example:

```
%preamble%
_foo := a2 *5;
%endpreamble%

This is an example of a Smartfield with a preamble.  The value
of _foo is %calc% _foo %endcalc%
```

  If the effective value of A2 was 2, then the result of processing this Smartfield would be:

```
This is an example of a Smartfield with a preamble.  The value
of _foo is 10.
```

- %servercommand% <arg1> <arg2> ... <argN> %endservercommand%

  This directive concatenates <arg1> through <argN> into a single string. The result of this concatenation is passed as a command to the server. The server's output is included in the resultant text. See the next section for a full example.

- %system% <arg1> <arg2> ... <argN> %endsystem%

  This directive concatenates <arg1> through <argN> into a single string. The resultant concatenation is used as a command that is executed in the default shell environment. This directive does not have an effect on the resultant text.

  Example:

  ```
  %system%
  "netscape http://www." B3 ".com &"
  %endsystem%
  ```

- %text% <arg1> <arg2> ... <argN> %endtext%

  This command concatenates its arguments into a single string and adds the result to the resultant text.

  Example:

  ```
  %text% _person " owes " b3 " in taxes." %endtext%
  ```

Querying Databases with the %servercommand% Directive

This section explains the use of servers in Thinksheet and in Thinksheet's Smartfields. Since this directive is often used in conjunction with databases, our running example will use an SQL interpreter as the server.

Our example will use an SQL interpreter that filters out *possible* and *false* values. We will call this interpreter ThinkQL. When ThinkQL encounters a comparison with *possible*, it will consider that expression to be true. When it encounters a comparison with *false*, it will consider that expression to be false. Comparisons between *possible* or *false* occur because the effective values of nodes may be either *possible* or *false*. For example, a query to ThinkQL might be:

```
SELECT name FROM movies WHERE genre = A0 or rating = A1;
```

If the effective value of A0 is *possible* and the effective value of A1 is *false*, then this is equivalent to

```
SELECT name FROM movies WHERE genre = Possible or rating = False;
```

Therefore, the expression `genre = Possible` will evaluate to true for every tuple, and the expression `rating = False` will evaluate to false. In this example, this would reduce to `True or False`, which is true for every row in the table. Therefore, all of the names would be selected.

ThinkQL currently parses only a small subset of the SQL language. It handles `SELECT` statements in which the `WHERE` clause contains boolean expressions, and the atoms of the boolean expressions are simple comparisons between attributes and values.

Connecting the Server to Thinksheet

A server has three fields that are used to initialize it.

1. Executable Path

   This is the shell command that will run the process we want to communicate with. Thinksheet will use the search path to execute the command. In our example, we would type `thinkql` (the command to start ThinkQL) assuming thinkql is in the search path, `<pathname>/thinkql` if not.

   Examples:

   ```
   thinkql
   ```

   ```
   /usr/thinksheet/bin/thinkql
   ```

2. Initialization

   This is the initialization string that is sent over to the interpreter after it is invoked. This string can be used to define procedures, set defaults etc. For example, the initialization string for ThinkQL could be a command to connect to a certain database:

   ```
   connect movie_db;
   ```

3. End Output Command

   Thinksheet communicates to the server through UNIX pipes [Ste90]. When Thinksheet reads the results of what the server prints to its standard out (which gets sent through the pipe), it has no way of knowing when the server is done. Consequently, Thinksheet may freeze trying to read from an empty pipe. To remedy this, Thinksheet and the server follow a simple protocol. When the server is done sending Thinksheet information, it prints out a period (`.`) on a line by itself. This way Thinksheet knows that it can stop reading from the pipe and continue processing. In ThinkQL, we would do it this way:

   ```
   print ".";
   ```

   This line will automatically be placed after every command that Thinksheet invokes, so the user doesn't have to worry about placing the dots.

Sending Queries to ThinkQL

Currently, communication to the server is possible only through Smartfields using the `%servercommand%` directive. Suppose our Smartfield contained this text:

```
The server says:
%servercommand%
"SELECT DISTINCT name FROM movies;"
%endservercommand%
```

This sends the query to the server. The result might be:

```
The server says:
Attack of the Killer Tomatoes
Beastmaster
Commando
The Road Warrior
Yor, the Hunter from the Future
```

When Thinksheet sees a server command it sends the string that is in the directive to the Server and then also sends the End Output Command. In the above example, the string Thinksheet would send would be:

```
SELECT DISTINCT name FROM movies;
print ".";
```

The ThinkQL process would print the movie names, followed by a period (.) on a separate line. Thinksheet reads from that process's standard out until it sees the period on a line by itself. After seeing this, it knows that the list of movie names is done and inserts that result into the Smartfield.

Dynamic Queries

The arguments to the `%servercommand%` directive are concatenated together into a single string and then sent to the server. The unquoted arguments are interpreted in the Thinksheet language. This allows the reader of a Thinksheet to adjust the queries dynamically. For example, one query might be:

```
%servercommand%
"SELECT DISTINCT name "
"FROM movies "
"WHERE genre = " B3 ";"
%endservercommand%
```

Thus if reader's answer to node B3 was "Action", the query that will be sent to the server would be:

```
SELECT DISTINCT name
FROM movies
WHERE genre = 'Action';
```

Mutual Restriction

This section shows how we might implement mutual restriction, as described in Chapter 6. The idea of mutual restriction is that the answers of a particular node restricts the choices of other nodes. We can implement this in Thinksheet using a Smartfield for the nodes' questions. For example, suppose a database table about movies had three fields, *name*, *genre* and *rating*. We create three nodes in Thinksheet, A1, B1, and C1, each node representing one of the respective attributes.

If node A1 represents the *name* attribute, then the Smartfield for its question might be:

```
What is the name of the movie you are interested in?
%servercommand%
"SELECT DISTINCT name "
"FROM movies "
"WHERE genre IN " B1 " "
"AND rating IN " C1 ";"
%endservercommand%
```

The answer choices presented to the reader of this Thinksheet will be the result of the query sent through the `%servercommand%`. For example, if both B1 and C1 have not been answered yet, the effective values for these nodes will be *possible*. Then the query will retrieve all of the names from the database.

Note the comparisons in this `WHERE` clause use the keyword `IN`. Since the effective values of nodes B1 and C1 may be a set of values (e.g. the effective value of B1 may be {Action, Fantasy}), we must use the keyword `IN` with the attribute comparison. This ANSI SQL predicate tests to see if the value of the attribute in a given row is any of the values in the given set [Gru93]. Thus the expression

```
genre IN ('Action', 'Fantasy')
```

is true if *genre* is either "Action" or "Fantasy" for a given row.

If node B1 represents *genre*, the Smartfield for its question would be:

```
What is the name of the movie you are interested in?
%servercommand%
"SELECT DISTINCT genre "
"FROM movies "
"WHERE name IN " A1 " "
"AND rating IN " C1 ";"
```

```
%endservercommand%
```

This query restricts the choices for *genre* based on the effective values of A1 and C1. Node C1 would have a Smartfield similar to nodes A1 and B1.

Using %servercommand% with MySQL

Currently, ThinkQL works only with the database system MySQL [MyS]. MySQL is a relational database system that works under UNIX and Microsoft Windows. It currently does not implement the whole ANSI SQL standard, but still contains a rich subset. This section gives a brief overview of how to create a database and load files into tables so that they may be used by ThinkQL. Readers should refer to the MySQL documentation for more information (see [MyS] for information about downloading the MySQL system and its documentation).

To create a database in MySQL, we run the command:

```
mysqladmin create <database>
```

This will create a database named `<database>`. We may now add tables through the MySQL interpreter. To run the interpreter, one simply enters the command `mysql`. Inside the interpreter, to create a table we use the `CREATE TABLE ...` command. For example, if we wanted to create a table called `movies` which had three fields for `name`, `genre` and `rating`, we would enter the following command:

```
CREATE TABLE movie (name VARCHAR(80), genre VARCHAR(10),
rating VARCHAR(10));
```

Having created the table, we now want to load it with information. We could do this one entry at a time using the `INSERT` command, or we can create a delimited file and then use the `LOAD` command. If we use the one-entry-at-a time approach, the command to insert an entry would look something like this:

```
INSERT INTO movies VALUES ('Beastmaster', 'Fantasy', 'PG');
```

The `LOAD` command has many options for loading a file into a table that are too extensive to explain here. However, it uses the simple defaults that attribute values are delimited by tab characters and rows delimited by newlines. After creating such a file, we could load it into the table `movies` by entering the following command:

```
LOAD DATA INFILE '/tmp/movies.txt' INTO TABLE movies;
```

Once we have entered the data into the database, it can now be accessed by ThinkQL. ThinkQL must be compiled with information about the location of the MySQL libraries and C header files. It uses this information to create a new mini-interpreter that accesses the MySQL database. See the documentation associated with the source code for more information.

Using %importfile% and %postcondition% to Modify a Thinksheet

The %importfile% directive and the %postcondition% directive may be used together to modify a currently running Thinksheet in response to a reader's question. For example, suppose we had a Thinksheet about Social Security benefits. One factor in calculating benefits is the annual income of the individual for all of the years he has worked [DMT94]. The Social Security Thinksheet might have nodes representing each year of income. Manually entering this income information would be tedious. If instead the information were stored in a database, we would prefer if the Thinksheet automatically entered the income information once the reader entered his name.

The combination of %importfile% and %postcondition% directives allow us to do this. Suppose the node asking for the reader's name was in cell A1. The question for that node would be a Smartfield that would look something like the following:

```
What is your name?
%postcondition%
%system%
"dbaccess " A1 " > /tmp/income.rpt"
%endsystem%
%importfile%
"/tmp/income.rpt"
%endimportfile%
%endpostcondition"
```

In this example, we use the %system% directive and the fictional dbaccess program to access the database. The dbaccess program must create a report file of the format described in Section 3.3.4. This report file will fill in the nodes representing annual income with the appropriate values. For example, suppose B1 and B2 were two nodes representing two particular years of income. The dbaccess program would find the income for the person named in A1 and add those two nodes to the report file. Thus, for example, part of the resultant report file could contain the following:

```
%node% B1
%answer%
10000
%endanswer%
%endnode%
%node% B2
%answer%
20000
%endanswer%
%endnode%
```

Here the income for the reader was 10,000 and 20,000 for those two years.

The `%importfile%` directive then merges that report file with the currently running Thinksheet. Because the `%system%` and `%importfile%` directives are inside of a `%postcondition%` directive, they are processed and executed *after* the reader has answered the question. Thus the reference to node A1 in the `%system%` directive would correctly contain the reader's name.

Appendix D

The Tcl Interface

This appendix lists the commands added to Tcl to allow access to Thinksheets (see Section 3.6). Readers interested in learning more about Tcl and Tk are referred to [Wel97].

The next section lists the new Tcl commands. The section after that presents an example interface developed with these added commands.

Listing of New Commands

- `sheetGraph <path>`

  This command opens the Thinksheet located in `<path>` and creates a new Tcl command whose name is `graph<n>` where `<n>` is an increasing number starting from 0. The return value of `sheetGraph` is the string `graph<n>` (hereby referred to simply as `graph`). The `graph` command may be used to invoke various operations on the Thinksheet. It has the following general form:

  `graph option ?arg arg ...?`

  `Option` and `arg`s determine the exact behavior the command. The following commands are available:

  - `graph keyword <pattern> <nodeid1> <nodeid2> ...`
    This command searches for `<pattern>` in the text field of the node id's listed in the arguments. It creates the file `links.keyword.html` which contains links to the text fields of the nodes that contain the keyword. The return value of this command is a list containing

    {`error <errorVal> <nodeid1> <hits1> <nodeid2> <hits2> ...` }

    where the literal string `error` is followed by an error value (0 means no error) and where `nodeid1` is the node's id and `hits1` is the number of pattern matches for `nodeid1`. One can use the `array set` command to easily make use of this information.

Example:

```
set result [eval [$graph keyword foo [$graph nodeList]]]
array set hits $result
if $hits(error) {
    ...
} else {
    foreach node [array names hits] {
                ...
    }
}
```

– `graph mergReport <filename>`

  This command merges the report from `<filename>` into the current sheet. The contents of `<filename>` must be in the Report Format described in Section 3.3.4. If it is not, then the command throws a Tcl error. The return value of this command is the empty string.

– `graph name`

  The command returns the name of the Thinksheet (the filename or directory used to load the Thinksheet).

– `graph nodeAnswer <id>`

  This command returns the formula string of the answer field of the node whose id is `<id>`.

– `graph nodeAnswerValue <id>`

  This command returns the value of the answer of node `<id>`.

– `graph nodeAnswerChildren <id>`

  This command returns the list of nodes whose answer fields depend on node `id`.

– `graph nodeAnswerChoices <id>`

  This command returns the list of valid answer choices for node `<id>`. It returns an empty list if there are no choices for this node.

– `graph nodeAnswerParents <id>`

  This command returns the list of parents for the answer field of node `<id>` (i.e. the list of nodes referenced in the answer formula of that node).

– `graph nodeChoicePositions <id>`

  This command returns the list of choices currently selected by the user from the answer choice list for node `<id>` (see `nodeAnswerChoices`). (e.g. if choices 1, 2, and 3 were selected, it would return {`1 2 3`}). Note, this does not have to correspond to the actual answer of the node, rather it pertains to the position of the the selected answers from the answer choice list.

- `graph nodeClearAnswers`

  This command clears all of the reader-settable answers of the Thinksheet.

- `graph nodeContents <id>`

  This command returns the text of the contents field of node `<id>` in its originally (unprocessed) form. The contents field is also known as the text field of the node. The term "contents" is used because future extensions to Thinksheet may store something other than text in this field.

- `graph nodeContentsFile <id>`

  This command returns the file name associated with the contents field for node `<id>`.

- `graph nodeContentsModified <id>`

  This command returns whether the contents have been modified since the last call to `nodeFilteredContents` for node `<id>`.

- `graph nodeDelete <idlist>`

  This command deletes the nodes specified in `<idlist>`. If it can not remove some of the nodes (because of dependencies), it throws an error and does not modify the Thinksheet (it does not remove *any* of the nodes in this case).

- `graph nodeFilteredContents <id> ?import?`

  This command returns the text of the contents field for node `<id>` after it has been processed as a Smartfield. The optional argument `import` is a boolean that tells whether to execute any `%importfile%`'s in the Smartfield (assumes true if not given).

- `graph nodeFilteredContentsFile <id> ?import?`

  This command returns the associated file name where the processed text of the text field for node `<id>` is stored, also processing the text at the same time. The optional argument `import` is a boolean that tells whether to execute any `%importfile%`'s in the Smartfield (assumes true if not given).

- `graph nodeFilteredQuestion <id> ?import?`

  This command returns the processed text of the question field for node `<id>` after it has been processed as a Smartfield. The optional argument `import` is a boolean that tells whether to execute any `%importfile%`'s in the Smartfield (assumes true if not given). This command returns only the question for the node (everything up to and including the first question mark (`?`)). To retrieve the list of answer choices for this node use `nodeAnswerChoices`.

- `graph nodeFilteredQuestionFile <id> ?import?`

  This command returns the associated file name where processed question field for node `<id>` is stored, processing the field at the same time. The optional argument `import` is a boolean that tells whether to execute an `%importfile%`'s in the Smartfield (assumes true if not given).

- `graph nodeHasContents <id>`
  This command returns 1 if node `<id>` has text in its contents field, otherwise 0.
- `graph nodeHasQuestion <id>`
  This command returns 1 if node `<id>` has text in its question field, otherwise 0.
- `graph nodeList`
  This command returns a list of node ids belonging to that graph. Node ids may be considered unique specifiers but actually follow a particular format prescribed by their placement on the grid in the spreadsheet interface. The formula for calculating a node id is to take its row and column in the spreadsheet interface and use the following formula : `row * 1000 + column`.
- `graph nodePreconditionValue <id>`
  This command returns the value of the precondition field of node `<id>`
- `graph nodePreconditionChildren <id>`
  This command returns the list of of nodes whose precondition field depends on node `<id>`.
- `graph nodePreconditionParents <id>`
  This command returns the list of nodes which the precondition field of node `<id>` depends on.
- `graph nodeQuestion <id>`
  This command returns the original unprocessed text of the question field of node `<id>`.
- `graph nodeQuestionFile <id>`
  This command returns the associated file name of the question of node `<id>`
- `graph nodeQuestionModified <id>`
  This command returns whether the question has been modified since the last call to `nodeFilteredQuestion <id>`.
- `graph nodeQuestionAnswerType <id>`
  This command returns the answer type of the question field of node `<id>` (either `text`, `numeric` or `any`)
- `graph nodeQuestionSelectionType <id>`
  This command returns the selection type of the question field of the node `<id>` (either `singleton`, `multiple` or `range`).
- `graph nodeSetAnswer <id> <answer>`
  This command sets the node `<id>`'s answer to `<answer>`
- `graph nodeStatus <id>`
  This command returns a string describing the status of node `<id>`.

- `graph nodeTitle <id>`
  This command returns the title field of node `<id>`.
- `graph nodeToposort <nodeid1> <nodeid2> ...`
  This command returns a list which is the topological sort of the node arguments.
- `graph report`
  This command returns a string representing the Thinksheet in the Report Format described in Section 3.3.4.
- `graph save ?savedir?`
  This command saves the Thinksheet to the current directory, or to `savedir` if specified.
- `graph tmpName`
  This command returns the name of the temporary directory for the graph data.

- `cell graph nodeId initProc updateProc deleteProc`

  This command creates a "cell" for the node labeled `nodeId` in `graph`. A cell is a basically a trigger that gets executed whenever the status (i.e. effective value) of the node changes. Three procedure names are passed to the `cell` command. They are:

  - `<initProc>`: This is the name of a Tcl procedure for initializing the cell. It must be called manually (see below).
  - `<updateProc>`: This is the name of a Tcl procedure that gets called when the status of the node changes. It may also be manually invoked (see below).
  - `<deleteProc>`: This is the name of a Tcl procedure that gets called when the cell or corresponding node is deleted.

  The `cell` command creates a new Tcl command whose name is `cell<n>` where `<n>` is an increasing number starting from 0. The return value of `cell` is the string `cell<n>`. The `cell<n>` command may be used to manually call the initialize or the update procedure. It may be called in two possible ways:

  - `cell<n> initialize`
    This calls the initialization procedure defined by `initProc`.
  - `cell<n> updateProc`
    This calls the update procedure defined by `updateProc`.

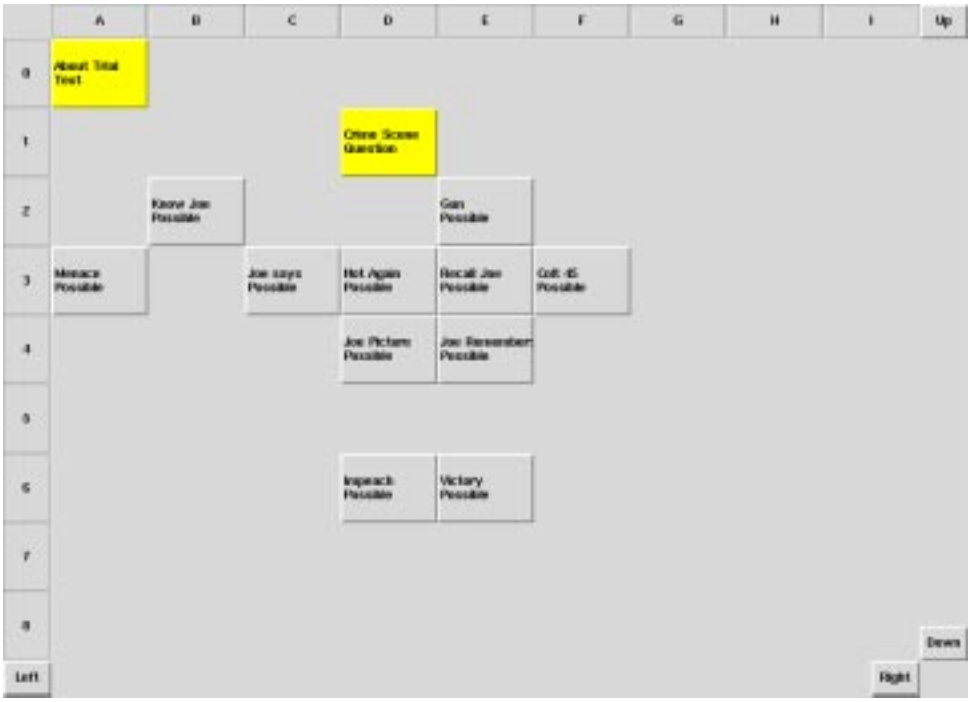


FIGURE D.1: Screen shot of a Tcl/Tk interface.

An Example Interface in Tcl/Tk

This section uses the added Tcl commands to build a simple reader-mode only interface using Tcl/Tk. Since this only meant as a tutorial, we will tend to choose simplicity over efficiency and/or some user-interface components (e.g. menus, etc). Some knowledge of Tcl and Tk may be helpful, although the tutorial will contain explanations of the code's action.

The interface we will build will be a simplified version of the spreadsheet interface described in Chapter 3. A screenshot of the interface is shown in Figure D.1. The overall flow and architecture of this example is as follows:

1. We create the graphical widgets, in this case a grid of buttons and labels.
2. We load the Thinksheet and associate cells with each of the nodes.
3. We create a refresh procedure which refreshes the entire screen. The refresh procedure reconfigures all the labels and buttons, and associates buttons to nodes. It is called when the reader scrolls the spreadsheet grid and at initialization time.
4. When a reader clicks on a button, the button invokes a command to ask the reader a question or present some text.
5. When a reader answers a question, the effective values and status of multiple nodes might change. Each node notifies its cell to update; the cell then calls the appropriate Tcl procedure. This procedure will reconfigure the appropriate button, if the node is on-screen. Otherwise, if the node is off-screen, it does nothing.

The Tcl/Tk Code for the Interface

Figure D.2 shows the Tcl code for opening the Thinksheet and creating the grid of buttons. This is the initial code that is called when the program is first run. The first line of this code merely configures the height and width of the main window. The loop creates a nine by nine grid of buttons (the number nine allows for easy computation of the placement of labels and buttons).

Each of the button ids start with the letter "b" followed by an identification number. This id is computed the same way as node ids are computed—as 1000 times the row of the button plus one times the column of the button. These button identifiers will be used later on to compute which node is to be associated with which button.

At the corners of the screen we place buttons labeled `Up`, `Down`, `Left` and `Right` (see Figure D.3). We will use these buttons to scroll our grid (a real application would use Tk scrollbars but those are too complicated to describe for this tutorial). These buttons modify the variables `startRow` and `startCol` which hold the row and column of the top left cell. For example, clicking on the button `Right` will increment the variable

`startCol` and then execute the function `refresh` (described later). The effect of this is that the spreadsheet will scroll one column to the right.

Finally in Figure D.4 we load a Thinksheet with the `sheetGraph` command. The command `tk_getOpenFile` provides a graphical file selector, allowing the user to choose the Thinksheet. We then create a cell for each node, which runs the `cellUpdate` procedure for updates and the `doNothing` procedure otherwise. The procedure `doNothing` is, as the name implies, an empty procedure. The `cellUpdate` procedure is described below. We save the association between cell and node in the `cellInfo` array.

Finally, we call the `refresh` procedure. The definition for `refresh` is given in Figure D.5. The `refresh` procedure must configure the cells and the buttons, based on the values of `startRow`, `startCol` and the Thinksheet (stored in `graph`). The result of `refresh` is two things. First, the labels and the buttons will be configured correctly. Secondly the array `buttonInfo` will store a mapping between nodes and their buttons. This variable will be used by the `cellUpdate` procedure.

The code for `cellUpdate` is given in Figure D.6. This code is called whenever the associated node's status changes, or when `refresh` manually calls this procedure. The procedure first checks to see if there is a button associated with the node by checking the `buttonInfo` array (the node may be off screen, and therefore there is nothing to update). If there is button, the procedure reconfigures the color, label and various other features of the button. It also reconfigures the command to be executed when the button is clicked. This command runs the procedure `buttonClicked`. The code for this command is given in Figure D.7.

In a real graphical user interface, `buttonClicked` would open up either a listbox, an entry widget, or a text widget to allow the reader to answer a question or read text. Implementing these features is beyond the scope of the tutorial. To give a flavor of what the procedure should do, the code given in Figure D.7 interacts with the terminal.

The procedure `buttonClicked` takes two arguments, the graph representing the Thinksheet and the node id. These two arguments are set when the button is configured by the `cellUpdate` procedure. The `buttonClicked` procedure checks to see if the node has a question. If it does, it presents the question and any answer choices it may have using `nodeFilteredQuestion` and `nodeAnswerChoices`. It then reads the terminal for the answer and sets the answer using `nodeSetAnswer`.

If the node has no question, the command assumes that it should present the text of the node, which it does using `nodeFilteredContents`.

```
. configure -height 400 -width 700
for {set i 0} {$i < 9} {incr i} {
    label .c$i -relief groove
    label .r$i -relief groove
    place .c$i -relwidth 0.1 -relheight 0.05 \
        -relx [expr $i / 10.0 + 0.05] -rely 0.0
    place .r$i -relwidth 0.05 -relheight 0.1 \
        -relx 0.0 -rely [expr $i / 10.0 + 0.05]
    for {set j 0} {$j < 9} {incr j} {
        set id [expr $i * 1000 + $j]
        set button [button .b$id]
        place $button -relwidth 0.1 -relheight 0.1 \
            -relx [expr $j / 10.0 + 0.05] \
            -rely [expr $i / 10.0 + 0.05]
    }
}
```

FIGURE D.2: Tcl/Tk code to create a grid of buttons for the spreadsheet interface.

```
set startRow 0
set startCol 0
button .up -text "Up" \
    -command [list if {$startRow} {incr startRow -1; refresh}]
button .down -text "Down" \
    -command {incr startRow; refresh}
button .left -text "Left" \
    -command [list if {$startCol} {incr startCol -1; refresh}]
button .right -text "Right" \
    -command  {incr startCol; refresh}
place .up -relwidth 0.05 -relheight 0.05 -relx .95 -rely 0.0
place .down -relwidth 0.05 -relheight 0.05 -relx .95 -rely .90
place .left -relwidth 0.05 -relheight 0.05 -relx 0 -rely 0.95
place .right -relwidth 0.05 -relheight 0.05 -relx .90 -rely .95
```

FIGURE D.3: Tcl/Tk code to create the buttons for scrolling.

```
set graph [sheetGraph [tk_getOpenFile]]
wm title . [$graph name]
foreach n [$graph nodeList] {
    set cellInfo($graph,$n) \
        [cell $graph $n doNothing cellUpdate doNothing]
}
refresh
```

FIGURE D.4: Tcl/Tk code to load the Thinksheet and create a cell for each node.

```
proc refresh {} {
    global startRow startCol buttonInfo graph cellInfo
    set nodeList [$graph nodeList]
    catch {unset buttonInfo}
    for {set i 0} {$i < 9} {incr i} {
        .c$i configure -text [colLabel [expr {$i + $startCol}]]
        .r$i configure -text [expr {$i + $startRow}]
        for {set j 0} {$j < 9} {incr j} {
            set id [expr {$i * 1000 + $j}]
            .b$id configure -text "" -relief flat -state disabled \
                    -bg [. cget -bg]
            set pnode [expr {($i + $startRow) * 1000 + $j + $startCol}]
            if {[lsearch $nodeList $pnode] != -1} {
                set buttonInfo($graph,$pnode) .b$id
                $cellInfo($graph,$pnode) update
            }
        }
    }
}
```

FIGURE D.5: The definition of the `refresh` procedure.

```
proc cellUpdate {i g n} {
    global buttonInfo
    if ![catch {set button $buttonInfo($g,$n)}] {
        set color [. cget -background]
        if ![string compare [$g nodeStatus $n] "False"] {
            $button configure -relief flat -state disabled -text "" \
                -bg $color
            return
        }
        if {![string compare [$g nodeStatus $n] "Question"] \
                || ![string compare [$g nodeStatus $n] "Text"]} {
            set color yellow
        }
        set title "[$g nodeTitle $n]\n[$g nodeStatus $n]"
        $button configure -relief raised -state normal -text $title \
            -anchor w -padx 1 -pady 1 -bg $color -justify left \
            -command [list buttonClicked $g $n]
    }
}
```

FIGURE D.6: The definition of the `cellUpdate` procedure.

```
proc buttonClicked {g n} {
    if [$g nodeHasQuestion $n] {
        puts [$g nodeFilteredQuestion $n]
        foreach c [$g nodeAnswerChoices $n] {
            puts $c
        }
        $g nodeSetAnswer $n [gets stdin]
    } else {
        puts [$g nodeFilteredContents $n]
    }
}
```

FIGURE D.7: The definition of the `buttonClicked` procedure.

Appendix E

The Thinksheet API

This appendix is a reference to the Application Programming Interface (API) for the Thinksheet core library. It is ordered by the various objects and the methods available for each of them.

Thinksheet uses three general classes to store sets, maps and strings. They are:

- `template <class T> Set`

  An object of this class stores a set of objects of type `T`.

- `template <class Key, class Val> Map`

  An object of this class maps objects of type `Key` to objects of type `Val`.

- `CString`

  An object of this class represents a string of characters.

Objects of these three types are often used as parameters and return values in Thinksheet functions.

Listing of Objects and Their Functions

The `Node` class

An object of this class represents a node in the Thinksheet graph. It has the following functions:

- `int userSetAnswer (const char *formula);`

  This function sets the answer field of the node to the formula specified by `formula`. Returns 1 if successful, 0 otherwise (e.g. if there was a syntax error in the formula).

- `int userSetPrecondition (const char *formula);`

  This function sets the precondition field of the node to the formula specified by `formula`. Returns 1 if successful, 0 otherwise (e.g. if there was a syntax error in the formula).

- `const Value& effectiveValue() const;`

  This function retrieves the effective value of the node.

- `SmartString* title();`

  Returns a pointer to the Smartfield representing the title field of the node.

- `Question* question();`

  Returns a pointer to the Smartfield representing the question field of the node.

- `SmartFile* contents();`

  Returns a pointer to the Smartfield representing the contents field of the node. The contents field is also known as the text field.

- `const Set<Node*>& node_ans_parents();`

  Returns the answer parents of the node.

- `const Set<Node*>& node_pre_parents();`

  Returns the precondition parents of the node.

- `const Set<Node*>& node_ans_children();`

  Returns the set of nodes whose answer fields depend on this node.

- `const Set<Node*>& node_pre_children();`

  Returns the set of nodes whose precondition fields depend on this node.

- `int status() const;`

  Returns the *status* of the node. The status may be one of the following:

  - `APOSSIBLE`

    The node's effective value is *possible*.

  - `AFALSE`

    The node's effective value is *false*.

  - `ADERIVED`

    The node is an uncomputed formula (i.e. the node's answer field is a formula that depends on other nodes, and the effective value of that formula is *possible*).

- `AFRONTIER`
  The node's precondition is *true* and it has an unanswered question.
- `ACONTENTS`
  The node's precondition is *true* and it has a non-empty contents field but no question.
- `AEFFECTIVEVALUE`
  The node's effective value is neither *false* nor *possible*, and the node does not have contents.

- `CString statusString() const;`

  This returns a string representation of the node's status. The string that is returned depends on the current status of the node:

  - `APOSSIBLE`
    Returns the string `Possible`.
  - `AFALSE`
    Returns the string `False`.
  - `ADERIVED`
    Returns the string `Derived`.
  - `AFRONTIER`
    Returns the string `Question`.
  - `ACONTENTS`
    Returns the string `Text`.
  - `AEFFECTIVEVALUE`
    Returns the effective value of the node in string form.

- `int HasContents() const`

  Returns 1 if the node has a non-empty contents field, 0 otherwise.

- `int HasQuestion() const`

  Returns 1 if the node has a non-empty question field, 0 otherwise.

- `CString getTitleText() const;`

  Returns the original (unprocessed) text of the title Smartfield.

- `CString getAnswerText() const;`

  Returns the text of the formula for the answer field.

- `CString getPreconditionText () const;`

  Returns the text of the formula for the precondition field.

- `CString getQuestionText() const;`

  Returns the original (unprocessed) text of the question Smartfield.

- `CString getFilteredQuestionText();`

  Returns the processed text of the question Smartfield, but does not execute any `%importfile%` commands.

- `CString getFilteredQuestionTextAndClick();`

  Returns the processed text of the question Smartfield and also executes any `%importfile%` commands.

- `CString getContentsText() const;`

  Returns the original unprocessed text of the contents Smartfield.

- `CString getFilteredContentsText();`

  Returns the processed text of the contents Smartfield, but does not execute any `%importfile%` commands.

- `CString getFilteredContentsTextAndClick();`

  Returns the processed text of the contents Smartfield and also executes any `%importfile%` commands.

- `int putQuestionText(const char *question);`

  Sets the text of the question Smartfield to `question`. Returns 1 is successful, 0 otherwise.

- `int putContentsText(const char *contents);`

  Sets the text of the contents Smartfield to `contents`. Returns 1 if successful, 0 otherwise.

- `const Map<int, CString>& getAnswerChoices();`

  Returns the set of answer choices available for the question field, if any. For example, if the question is:

  ```
  Where you on the corner of Warner and Tampa on April 25, 1998?
  1. Yes
  2. No
  ```

  It will return the choices `1. Yes` and `2. No`. The type `Map<int, CString>` maps the choices to their position in the choice list (e.g. the choice `1. Yes` would be in the first position, and `2. No` would be in the second.). The ordering depends on the choices position in the question field, not on the labeling of the choice itself. For example, if the question had been instead:

```
Where you on the corner of Warner and Tampa on April 25, 1998?
2. No
1. Yes
```

Then the choice `2. No` would be in the first position.

- `static int propagate (const Map< Node*, Set<toposortMarks> >& startNodes);`

  This function initiates propagation as described in Chapter 4. It is normally called by other functions in the Thinksheet core, such as `userSetAnswer` and `userSetPrecondition`, and not the outside interface.

  The parameter passed to `propagate` is a map from node pointers to marks. The possible marks are:

  - `SETANSWER`

    This marks tells the propagation engine that the answer formula for the node has changed (i.e. the formula string has changed) and it must be re-evaluated.

  - `INITANSWER`

    This mark tells the propagation engine that the answer formula should be initialized to the empty string (this avoids some checks that `SETANSWER` performs, so is slightly faster).

  - `SETPRECONDITION`

    This mark tells the propagation engine that the precondition formula has changed (i.e. the formula string has changed) and it must be re-evaluated.

  - `REEVAL_ANSWER`

    This mark tells the propagation engine that the answer formula needs to be re-evaluated (however, the formula string is still the same).

  - `REEVAL_PRECONDITION`

    This marks tells the propagation engine that the precondition formula needs to be re-evaluated (however the formula string is still the same).

  The parameter that is passed to the propagation engine is the map which associates nodes to a set of marks (thus we may do things like have the marks `SETANSWER` and `SETPRECONDITION` for the same node). For example, `userSetAnswer` will add the current node to the map and associate it with a single mark `SETANSWER`, and then call `propagate` with this map as the parameter.

The `SheetGraph` class

An object of this class represents an entire Thinksheet. It has the following functions:

- `void operator+= (Node *node);`

  This function adds the node pointed to by `node` to the Thinksheet.

- `void operator-= (Node *node);`

  This function removes the node pointed to by `node` from the Thinksheet.

- `Node *operator (int id);`

  This function returns a pointer to the node whose identifier is `id`. If no such node exists, it returns `NULL`.

- `Set<Node*> nodeList() const;`

  Returns the set of nodes in the Thinksheet.

- `CString name();`

  Returns the name of the Thinksheet (usually the file or directory name from where the Thinksheet was loaded).

- ```
  int merge(const SheetMap &sheetMap,
            const Set<CString> &functionDefs,
            const Set<CString> &preambleDefs,
            const Map<CString, CString> &serverDefs);
  ```

  The intention of this command is to merge a report file with the SheetGraph. In order to use this command, the user must first read the report file with a `ReadReportFormat` object. Here is an example of how to do this:

  ```
  ReadReportFormat reader (filename);
  SheetMap sheetmap;
  Set<CString> functions;
  Set<CString> preambles;
  Map<CString, CString> server;

  reader.ReadSheetMap (sheetmap, functions, preambles, server);
  if (!graph->merge(sheetmap, functions, preambles, server)) {
      // report the error ...
  }
  ```

The `SmartField` class

The class `SmartField` is an abstract base class for `SmartString` and `SmartFile`. The `SmartField` class abstracts out the notion of how the text of the Smartfield is stored. Therefore, any functions dealing the storage and retrieval of the text are declared abstract. `SmartString` and `SmartFile` implement these functions—the former stores the text in memory, while the latter stores the text in a file. The functions available to an object of the `SmartField` class are listed below:
  ```

- `virtual int isempty() const = 0;`

  Returns 1 if the Smartfield is empty, 0 if not. This is an abstract function in `SmartField`.

- `virtual int setField (const CString&)=0;`

  Sets the text of the Smartfield, returns 1 if successful, 0 if not. This is an abstract function in `SmartField`.

- `virtual CString filtered();`

  Returns the processed text of the Smartfield, but does not execute any `%importfile%` commands.

- `virtual CString filteredAndClick();`

  Returns the processed text of the Smartfield, and executes any `%importfile%` commands.

- `virtual CString original() const = 0;`

  Returns the original unprocessed text of the Smartfield. This is an abstract function in `SmartField`.

- `virtual void clear() = 0;`

  Clears the Smartfield. This is an abstract function in `SmartField`.

- `void parentChange();`

  This notifies the Smartfield that one of its parent nodes has changed its effective value (so the text must be re-processed on the next call to `filtered` or `filteredAndClick`).

- `void modify();`

  This notifies the Smartfield that it has been modified (so the text must be re-processed on the next call to `filtered` or `filteredAndClick`). It is normally not called (`setField` usually takes care of this).

- `int isDifferent();`

  Returns 1 if something about the Smartfield has changed since the last time `filtered` or `filteredAndClick` has been called (either a parent has changed or the text itself has been modified).

- `int recalculateOnClick();`

  Returns 1 if the Smartfield must re-process itself every time the user "clicks" on it, 0 otherwise. A "click" means that the user wants to retrieve the information and look at it (i.e. the processed text was not asked for by some system call like

keyword search). In such case, if the Smartfield contains directives like `%system%` and `%importfile%`, it must be reprocessed every time, even if none of its parents have changed or the text hasn't been modified.

- `const Map<int,CString>& importFiles();`

  Returns the list of filenames from all of the `%importfile%` directives in the Smartfield.

- `const Map<int,SmartField*> &postconditions();`

  Returns the list of Smartfields given in all of the `%postcondition%` directives in the Smartfield.

Bibliography

[AL97] José Abracos and Gabriel Pereira Lopes. Statistical methods for retrieving most significant paragraphs in newspaper articles. In *Proceedings of the ACL/EACL Workshop on Intelligent Scalable Text Summarization*, Madrid, Spain, 1997.

[BE97] Regina Barzilay and Michael Elhadad. Using lexical chains for text summarization. In *Proceedings of the ACL/EACL Workshop on Intelligent Scalable Text Summarization*, pages 10–17, Madrid, Spain, 1997.

[BH94] Benjamin Bederson and James D. Hollan. Pad++: A zooming graphical interface for exploring alternate interface physics. In *Proceedings of the ACM Symposium on User Interface Software and Tecnology*, pages 17–26, 1994.

[CH96] Waiman Cheung and Cheng Hsu. The model-assisted global query system for multiple databases in distributed enterprises. *ACM Transactions on Information Systems*, 14(4):421–470, October 1996.

[CJ96] Shawn Callahan and David Johnson. Dataset publishing - a means to motivate metadata entry. In *First IEEE Metadata Conference*, 1996. http://www.llnl.gov/liv_comp/metadata/events/ieee-md.4-96.html.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[DCMG97] Marilyn Drewry, Helen Conover, Susan McCoy, and Sara J. Graves. Metadata: Quality vs. quantity. In *Second IEEE Metadata Conference*, NOAA Auditorium, Silver Spring, Maryland, 1997. http://www.llnl.gov/liv_comp/metadata/md97.html.

[DMT94] Dale R. Detlefs, Robert J. Myers, and J. Robert Teanor. *1995 Mercer Guide to Social Security*. William H. Mercer, Inc., 1994.

[End72] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, Inc., 1972.

[Fur86] George Furnas. Generalized fisheye views. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 16–23, 1986.

[Gri97] Nancy Griffeth, September 1997. Memorandum.

[Gru93] Martin Gruber. *SQL Instant Reference.* SYBEX Inc., 2021 Challenger Drive, Alameda, CA 94501, 1993.

[GST98] Helena Galhardas, Eric Simon, and Anthony Tomasic. A framework for classifying scientific metadata. In *AAAI98 Workshop on AI and Information Integration*, July 1998.

[Hel91] Dan Heller. *Motif Programming Manual.*, volume 6 of *The Definitive Guides to the X Windows System.* O'Reilly and Assoc., 1991.

[HRWL84] F. Hayes-Roth, D. Waterman, and D. Lenat. *Building Expert Systems.* Addison-Wesley, 1984.

[Ign91] James Ignizio. *Introduction to Expert Systems: The Development and Implementation of Rule-Based Expert Systems.* McGraw Hill, 1991.

[Imm97] Immigration Moratorium Act of 1997, January 1997. Bill H.R. 347 of the United States House of Representatives.

[Int] The Internet Movie Database. http://www.imdb.com/.

[Jac90] Peter Jackson. *Introduction to Expert Systems.* Addison-Wesley, second edition, 1990.

[KP84] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*, chapter 8, pages 233–287. Prentice-Hall, 1984.

[KR96] Judith L. Klavans and Philip Resnik, editors. *The Balancing Act: Combining Symbolic and Statistical Approaches to Language.* The MIT Press, Cambridge, Massachusetts, 1996.

[Maz97] Subhasish Mazumdar. Organizing metadata using datalog rules. In *Second IEEE Metadata Conference*, NOAA Auditorium, Silver Spring, Maryland, 1997. http://www.llnl.gov/liv_comp/metadata/md97.html.

[Mer94] Dennis Merritt. Deceptive user interfaces impede AI. *AI Expert Magazine*, pages 19–24, August 1994.

[MR95] Kathleen McKeown and Dragomir R. Radev. Generating summaries of multiple news articles. In *Proceedings of the Eighteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 74–82, 1995.

[MyS] MySQL. T.c.X. DataKonsult AB. http://www.tcx.se/.

[PF93] Ken Perlin and David Fox. Pad: An alternative approach to the computer interface. In *Proceedings of the ACM SIGGRAPH Conference*, pages 57–64, 1993.

[PYLS96] Peter Piatko, Roman Yangarber, Daoi Lin, and Dennis Shasha. Thinksheet: A tool for tailoring complex documents. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, page 546, Montreal, Quebec, Canada, 4–6 June 1996.

[SB92] Manjit Sarkar and Marc H. Brown. Graphical fisheye views of graphs. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 83–91, 1992.

[Shn89] Ben Shneiderman. Reflections on authoring, editing and managing hypertext. In Edward Barrett, editor, *The Society of Text*, pages 115–131. MIT Press, Cambridge, MA, 1989.

[Sim98] June 1998. Personal communication with Eric Simon.

[SSTR93] M. Sarkar, S. Snibbe, O. Tversky, and S. Reiss. Stretching the rubber sheet: A metaphor for viewing large layouts on small screens. In *Proceedings of the ACM Symposium on User Interface Software and Tecnology*, pages 81–91, 1993.

[Ste90] W. Richard Stevens. *UNIX Network Programming*, chapter 3, pages 102–109. Prentice-Hall, 1990.

[Tan98] David Tanzer. Precondition theories and a query algorithm for thinksheet. Thesis proposal, June 1998.

[TM97] Simone Teufel and Marc Moens. Sentence extraction as a classification task. In *Proceedings of the ACL/EACL Workshop on Intelligent Scalable Text Summarization*, Madrid, Spain, 1997.

[Ull88] Jeffery D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[Wel97] Brent B. Welch. *Practical Programming in Tcl and Tk*. Prentice-Hall, second edition, 1997.

[Wor] World Wide Web Consortium. W3C's HTML Homepage. http://www.w3.org/MarkUp/.

[WS94] C. Williamson and B. Shneiderman. The dynamic homefinder: evaluating dynamic queries in a real-estate information exploration system. In B. Shneiderman, editor, *Sparks of Innovation in Human-Computer Interaction*, pages 295–307. Ablex Publishing Corp, 1994.

[You92] Douglas A. Young. *Object-oriented programming with C++ and OSF/Motif.* Prentice-Hall, 1992.