Intelligent Information Systems, ?, 1–21 (?)

Token-Templates and Logic Programs for Intelligent Web Search

BERND THOMAS bthomas@uni-koblenz.de
Universität Koblenz, Institut für Informatik, Rheinau 1, D-56075 Koblenz, Germany

Received ?; Revised ?, ?

Editor: Zbigniew Ras

Abstract. We present a general framework for the information extraction from web pages based on a special wrapper language, called *token-templates*. By using *token-templates* in conjunction with logic programs we are able to reason about web page contents, search and collect facts and derive new facts from various web pages. We give a formal definition for the semantics of logic programs extended by *token-templates* and define a general answer-complete calculus for these extended programs. These methods and techniques are used to build intelligent mediators and web information systems.

Keywords: information extraction; template based wrappers; mediators; logic programming; theory reasoning; deductive web databases; softbots; logic robots

1. Introduction

In the last few years it became appearant that there is an increasing need for more intelligent World-Wide-Web information systems. The existing information systems are mainly document search engines, e.g. *Alta Vista, Yahoo, Webcrawler*, based on indexing techniques and therefore only provide the web user a list of document references and not a set of facts he is really searching for. These systems overwhelm the user with hundreds of web page candidates. The exhausting and highly inconvenient work to check these candidates and to extract relevant information manually is left to the user. The problem gets even worse if the user has to take comparisons between the contents of web pages or if he wants to follow some web links on one of the candidate web pages that seem to be very promising. Then he has to manage the candidate pages and has to keep track of the promising links he has observed.

To build intelligent web information systems we assume the WWW and its web pages to be a large relational database, whose data and relations can be made available by the definition and application of special extraction descriptions (*token-templates*) to its web pages. A library of such descriptions may then offer various generic ways to retrieve facts from one or more web pages. One basic problem we are confronted with is to provide means to access and extract the information offered on arbitrary web pages, this task is well known as the process of *information extraction* (IE). The general task of IE is to locate specific pieces of text in a natural language document, in this context web pages. In the last few years many techniques have been developed to solve this problem [1, 6, 10, 11, 15, 21, 30], where *wrappers* and *mediators* fulfill the general process to retrieve and integrate information from heterogeneous data sources into one information system.

We focus our work on a special class of wrappers, which extract information from web pages and map it into a relational representation. This is of fundamental interest because it offers a wide variety of possible integrations into various fields, like relational databases, spreadsheet applications or logic programs. We call this information extraction process *fact-retrieval*, due to logic programming the extracted information is represented by ground atoms. In this article we present a general framework for the fact retrieval based on our special wrapper language, called *token-templates*. Our general aim was to develop a description language for the IE from semi-structured documents, like web pages are. This language incorporates the concepts of *feature structures* [25], *regular expressions*, *unification*, *recursion* and *code calls*, to define templates for the extraction of facts from web pages.

How does this contributes to *logic programming*? The key idea of using logic programs for intelligent web browsing is as follows: Normally the user is guided by his own domain specific knowledge when searching the web, manually extracting information and comparing the found facts. It is very obvious that these user processes involve inference mechanisms like reasoning about the contents of web pages, deducing relations between web pages and using domain specific background knowledge.Therefore he uses deduction, based on a set of rules, e.g. which pages to visit and how to extract facts. We use logic programs in conjunction with token-templates to reason about the contents of web pages, to search and collect relevant facts and to derive new facts from various web pages. The logic programming paradigm allows us to model a background knowledge to guide the web search and the application of the extraction templates. Furthermore the extracted facts in union with additional program clauses correspond to the concept of deductive databases and therefore provide the possibility to derive new facts from several web pages. In the context of *wrappers* and *mediators* [30], *token-templates* are used to construct special wrappers to retrieve facts from web pages. Logic programs offer a powerful basis to construct mediators, they normalize the retrieved information, reason about it and depending on the search task to fulfill, deduce facts or initialize new sub searching processes. By merging token-templates and logic programs we gain a mighty inference mechanism that allows us to search the web with deductive methods. We emphasize the well defined theoretical background for this integration, which is given by *theory reasoning* [2] [26] in logical calculi, whereas *token-templates* are interpreted as theories.

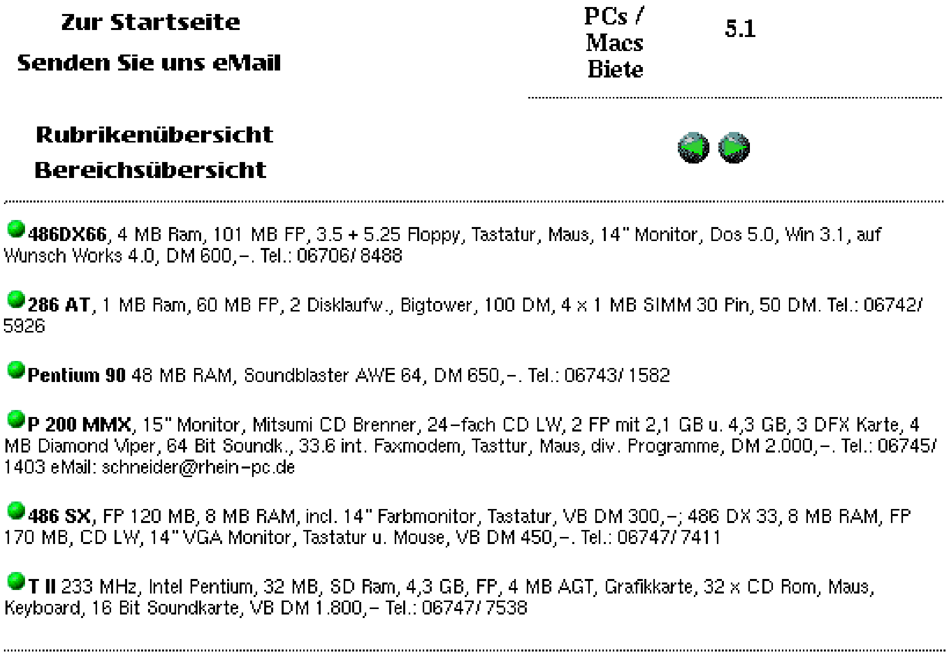
This article is organized as follows: in Section 2 we describe the language of *token-templates* for the fact-retrieval from web pages. In Section 3 the integration of *token-templates* into logic programs and the underlying $\mathcal{T}_T$–$Calculus$ is explained. Section 4 describes how logic programming techniques can be used to enhance the fact-retrieval process with deductive techniques. A practical application of our developed methods, a *LogicRobot* to search private advertisements, is briefly presented in Section 5. Related approaches and conclusions are given in Section 6.

2. A Wrapper Language for Web Documents

In this section we describe our information extraction language, the *token-templates*. We assume the reader to be familiar with the concepts of *feature structures* [25] and *unification* [13].

2.1. *The Fact Retrieval*

We split the process of fact-retrieval into several steps, whereas the first step is the preprocessing of the web page to be analyzed. We transform a web page as shown in Figure 1 into a list of tokens which we will explain in detail in section 2.2. In our existing system (Section 5) this is done by the lexical analyzer *FLEX* [17] (by the definition of a *FLEX* grammar to build tokens in extended term notation). We want to emphasize that we are not bound to a special lexical analyzer generator tool like *FLEX*, any arbitrary tool can be used as long as it meets the definition of a token.



Zur Startseite

Senden Sie uns eMail

PCs / Macs Biete **5.1**

Rubrikenübersicht

Bereichsübersicht

● **486DX66**, 4 MB Ram, 101 MB FP, 3.5 + 5.25 Floppy, Tastatur, Maus, 14" Monitor, Dos 5.0, Win 3.1, auf Wunsch Works 4.0, DM 600,–. Tel.: 06706/ 8488

● **286 AT**, 1 MB Ram, 60 MB FP, 2 Disklaufw., Bigtower, 100 DM, 4 x 1 MB SIMM 30 Pin, 50 DM. Tel.: 06742/ 5926

● **Pentium 90** 48 MB RAM, Soundblaster AWE 64, DM 650,–. Tel.: 06743/ 1582

● **P 200 MMX**, 15" Monitor, Mitsumi CD Brenner, 24–fach CD LW, 2 FP mit 2,1 GB u. 4,3 GB, 3 DFX Karte, 4 MB Diamond Viper, 64 Bit Soundk., 33.6 int. Faxmodem, Tasttur, Maus, div. Programme, DM 2.000,–. Tel.: 06745/ 1403 eMail: schneider@rhein–pc.de

● **486 SX,** FP 120 MB, 8 MB RAM, incl. 14" Farbmonitor, Tastatur, VB DM 300,–; 486 DX 33, 8 MB RAM, FP 170 MB, CD LW, 14" VGA Monitor, Tastatur u. Mouse, VB DM 450,–. Tel.: 06747/ 7411

● **T II** 233 MHz, Intel Pentium, 32 MB, SD Ram, 4,3 GB, FP, 4 MB AGT, Grafikkarte, 32 x CD Rom, Maus, Keyboard, 16 Bit Soundkarte, VB DM 1.800,– Tel.: 06747/ 7538

Figure 1. An advertisement web page

This methods allows us to apply our techniques not only to *HTML*-documents, but also to any kind of semi-structured text documents. Because we can construct arbitrary tokens, our wrapper language is very flexible to be used in different contexts.

After the source code transformation of the web document, the matching and extraction process takes place. Extraction templates built from tokens and special operators are applied to the tokenized documents. According to the successful matching of these extraction templates the relevant information is extracted by means of unification techniques and mapped into a relational representation. We will now explain the basic element of our web wrapper language, the *token*.

2.2. *The Token*

A token describes a grouping of symbols in a document. For example the text `<b>Pentium 90</b>` may be written as the list of tokens:

```
[token(type=html,tag=b), token(type=word,txt='Pentium'),
token(type=whitespace,val=blank), token(type=int,val=90),
token(type=html_end,tag=b)]
```

We call a feature structure (*simple* and acyclic feature structure) a *token*, if and only if it has a feature named `type` and no feature value that consists of another feature. A feature value may consist of any constant or variable. We write variables in capital letters and constants quoted if they start with a capital letter. Furthermore, we choose a term notation for feature structures (token), that is different from that proposed by Carpenter in [4].

We do not code the features to a fixed argument position, instead we extend the arguments of the annotated term, by the notation `Feature = Value` (extended term notation),
this offers us more flexibility in the handling of features. Figure 2 shows the graph notation of a token and our extended term notation of it. In the following we denote a token in extended term notation, simply token.

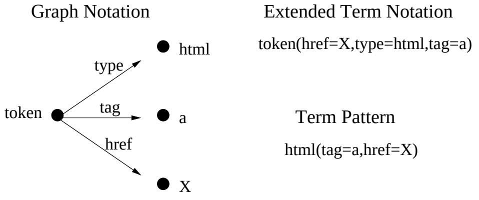


Figure 2. Token notations

2.3. *Token Matching*

In the following let us assume, that an arbitrary web page transformed into a token list is given. The key idea is now to recognize a token or a token sequence in this token list. Therefore we need techniques to match a token description with a token. For feature structures a special unification, the *feature unification* was defined in [23]. For our purposes we need a modified version of this unification, the *token-unification*.

Definition 1 (Token-Unification). Let $T_1 = token(f_1 = v_1, f_2 = v_2, \ldots, f_j = v_j)$ $j \geq 1$ and $T_2 = token(f'_1 = v'_1, f'_2 = v'_2, \ldots, f'_k = v'_k)$ $k \geq 1$ be tokens.
Let A_1 and A_2 be the feature-value sets of the tokens T_1 and T_2 that is $A_1 = \bigcup_{i=1}^{j} \{(f_i, v_i)\}$ and $A_2 = \bigcup_{i=1}^{k} \{(f'_i, v'_i)\}$
Further let $F = \{f \mid (f, v) \in A_1\}$ be the set of features T_1 and T_2 have in common. The terms T'_1 and T'_2 are defined as follows: $T'_1 := token(f_1 = v_1, \ldots, f_n = v_n)$ and $T'_2 := token(f_1 = v'_1, \ldots, f_n = v'_n)$ with $f_i \in F \wedge (f_i, v_i) \in A_1 \wedge (f_i, v'_i) \in A_2 \wedge 1 \leq i \leq n \wedge n = |F|$.

The tokens T_1 and T_2 are token-unifiable *iff* the following two conditions hold:

(1) $\{f_1, \ldots, f_j\} \subseteq \{f'_1, \ldots, f'_k\}$

(2) T_1' is unifiable with T_2'. The most general unifier (mgU) σ of T_1 and T_2 is the mgU of T_1' and T_2' wrt. the usual definition [16, 13]

If (1) or (2) does not hold, we call T_1 and T_2 not token-unifiable, written $T_1 \not\sqcup T_2$. We write $T_1 \sqcup_\sigma T_2$ *iff* T_1 is token-unifiable with T_2 and σ is the mgU of the unification from T_1' with T_2'.

The motivation for this directed unification [1] is to interpret the *left* token to be a pattern to match the *right* token. This allows us to set up feature constraints in a easy way, by simply adding a feature to the *left* token. On the other hand we can match a whole class of tokens, if we decrease the feature set of the *left* token to consist only of the $type$ feature.

EXAMPLE:

- *token(type=word)* $\sqcup_\sigma$ *token(type=word, txt='Pentium')* with $\sigma = []$
- *token(type=word, int=X)* $\not\sqcup_\sigma$ *token(type=word,txt='Pentium')*
- *token(tag=X,type=html,href=Y)* $\sqcup_\sigma$ *token(type=html, tag=a, href='http://www.bmw.de')* with $\sigma = [X/a, Y/'http://www.bmw.de']$

□

For ease of notation we introduce an alternative notation t^k for a token of type k (the feature type has value k), that is given by k or $k(f_1 = v_1, f_2 = v_2, \ldots, f_n = v_n)$ with f_i and v_i are the features and values of t^k where n is the number of features of t^k. We call this notation *term-pattern* and define a transformation $\mathcal{V}$ on *term-patterns* such that $\mathcal{V}$ transforms the *term-pattern* into the corresponding token. For example $\mathcal{V}(word(len = 5, text = X))$ gives us the token $token(type = word, len = 5, text = X)$. This transformation exchanges the functor of the *term-pattern*, from type k to $token$ and adds the argument $type = k$ to the arguments. Now we can define the basic match operation on a *term-pattern* and a token:

Definition 2 (Term-Match). Let t^k be a term-pattern and T a token in extended term notation. The term-pattern t^k matches the token T, $t^k \sqsubset_\sigma T$, *iff* $\mathcal{V}(t^k)$ is token-unifiable with T. The term-match is defined as follows:

$$t^k \sqsubset_\sigma T \Longleftrightarrow \mathcal{V}(t^k) \sqcup_\sigma T \textit{ where } \sigma \textit{ is mgU.}$$

EXAMPLE: For the demonstration of the term-match operation, consider the three examples mentioned above and the following modification:

- *word* $\sqsubset_\sigma$ *token(type=word, txt='Pentium')* with $\sigma = []$
- *word(int=X)* $\not\sqsubset_\sigma$ *token(type=word,txt='Pentium')*
- *html(href=Y)* $\sqsubset_\sigma$ *token(type=html, tag=a, href='http://www.bmw.de')* with $\sigma = [Y/'http://www.bmw.de']$

□

2.4. *Token Pattern*

If we interpret a token to be a special representation of text pieces, the definition of term-matching allows us to recognize certain pieces of text and extract them by the process of unification. This means the found substitution σ contains our extracted information.

But yet we are not able to match sequences of tokens in a tokenized web page. Therefore we define the syntax of *token-pattern*, which will build our language to define templates for the information extraction from web documents.

The language of *token-patterns* is built on a similar concept as regular expressions are. The difference is, that the various iteration operators are defined on tokens. Beside these basic operators we define $greedy$ and $moderate$ operators. These two operator classes determine the enumeration order of matches. *Greedy* Operators are: $?, +$ and $*$. *Moderate* Operators are: $!, -$ and $\#$. For example a *token-pattern* like $*word$, matches zero or arbitrary many tokens of type word, but the first match will try to match as many tokens of type word as possible ($greedy$). Whereas the pattern $\#word$ will try in its first attempt to match as less tokens as possible ($moderate$). This makes sense if a pattern is just a part of a larger conjunction of patterns. Another advantage is given by the use of unification, which in fact allows us with the later described concept of recursive *token-templates* to recognize context sensitive languages.

In Figure 2.4 we give an informal definition [2] for the semantics of *token-patterns*. Assume that a tokenized document D is given. A match of a *token-pattern* p on D, $p \rhd D$, returns a set of triples (MS, RS, σ), where MS is the matched token sequence, RS is the rest sequence of D and σ is the mgU of the token unifications applied during the matching process.

We emphasize, that we compute all matches and do not stop after we have found one successful match, though this can be achieved by the use of the *once* operator.

EXAMPLE: Let us have a closer look at the source code of the advertisement web page (D) shown in Figure 4 and the corresponding token-pattern (p) given in 5. This *token-pattern* extracts the item name of the offered object (*Item*) and the description (*Description*) of the item. For this small example our set of matches consists of a set with two tuples, where we will leave out the matched sequence MS and the rest sequence RS, because we are only interested in the substitutions σ of each match:

$p \rhd D = \{[Item/'$*286 AT*$'$, $Description/'$*1 MB Ram, 60 MB FP, 2 Disklaufw., Bigtower,*
100 DM, 4 x 1 MB SIMM 30 Pin, 50 DM. Tel.: 06742/ 5926’],
$[Item/'$*Pentium 90*$'$, $Description/'$*48 MB RAM, Soundblaster AWE 64, DM 650,-.*
Tel.: 06743/ 1582$']\}$

□

2.5. *Token-Templates*

A *token-template* defines a relation between a tokenized document, extraction variables and a *token-pattern*. Extraction variables are those variables used in a *token-pattern*, which are of interest due to their instantiation wrt. to the substitutions obtained from a successful match.

pattern	semantics
p	If $p \sqsubset_\sigma D(1)$ holds, then the matched sequence is the list containing exactly one element $D(1)$ and RS is D without the first element. $D(n)$ denotes the n-th element of the sequence D.
$?p_1$	Matches the pattern p_1 once or never; first the match of p_1 then the empty match sequence is enumerated.
$!p_1$	Matches the pattern p_1 once or never; first the empty match sequence then the match of p_1 is enumerated.
$+p_1$	Matches the pattern p_1 arbitrarily often but at least once; uses a decreasing enumeration order of the matches according to their length, starting with the longest possible match.
$-p_1$	Matches pattern p_1 arbitrarily often but at least once; uses an increasing enumeration order of the matches according to their length, starting with the shortest possible match.
$*p_1$	Matches the pattern p_1 arbitrarily often; uses a decreasing enumeration order of the matches according to their length, starting with the longest possible match.
$\#p_1$	Matches the pattern p_1 arbitrarily often; uses a increasing enumeration order of the matches according to their length, starting with the shortest possible match.
$not(t_1, \ldots, t_n)$	The not operator matches exactly one token t in D, if no $t_i \in \{t_1, \ldots, t_n\}$ exists, such that $t_i \sqsubset_\sigma t$ holds. The token $t_1 \ldots t_n$ are excluded from the match.
$times(n, t)$	Matches exactly n tokens t.
any	Matches an arbitrary token.
$once(p_1)$	The $once$ operator 'cuts' the set of matched tokens by p_1 down to the first match of p_1; useful if we are interested only in the first match and not in all alternative matches defined by p_1.
$X = p_1$	Unification of X and the matched sequence of p_1; only successful if p_1 is successful and if MS of p_1 is unifiable with X.
$p_1\, and\, p_2$	Only if p_1 and p_2 both match successfully, this pattern succeeds. The matched sequence of p_1 and p_2, is the concatenation of MS of p_1 and MS of p_2.
$p_1\, or\, p_2$	p succeeds if one of the pattern $p1$ or p_2 is successfully matched. The matched sequence of p is either the matched sequence of p_1 or p_2. The and operator has higher priority than the or operator, (e.g. $a\ and\ b\ or\ c \equiv (a\ and\ b)\ or\ c$)

Extended Token Patterns

$[t_1, \ldots, t_n]$	Execute token-template t_1 to t_n (see Section 2.5.1)
$\{c_1, \ldots, c_n\}$	Call *foreign* functions c_1 to c_n (see Section 2.5.1)

Figure 3. Language of Token-Patterns

```
<P><IMG SRC=img/bmp_priv.gif><B>286 AT</B>,
1 MB Ram, 60 MB FP, 2 Disklaufw., Bigtower, 100 DM, 4 x 1 MB SIMM
30 Pin, 50 DM. Tel.: 06742/ 5926
<P><IMG SRC=img/bmp_priv.gif><B>Pentium 90</B> 48 MB RAM,
Soundblaster  AWE 64, DM 650,-. Tel.: 06743/ 1582
```

Figure 4. HTML source code of an online advertisement

$$\begin{array}{l} \#any\ and\ html(tag = img)\ and\ html(tag = b) \\ and\ Item = *not(html\_end(tag = b))\ and\ html\_end(tag = b) \\ and\ once(Description = *not(html(tag = p))) \end{array}$$

Figure 5. Token-pattern for advertisement information extraction

Extraction variables *hold* the extracted information obtained by the matching process of the *token-pattern* on the tokenized document.

Definition 3 (Token-Template). Let p be a token-pattern, D an arbitrary tokenized document and $v_1, \ldots, v_n \in \mathcal{V}ar(p)$ variables in p. For $v_1, \ldots, v_n$ we write $\vec{v}$ and $\vec{v}_\sigma$ for applying the substitution σ to $\vec{v}$. A token-template r is defined as follows:

$$r(D, \vec{v}, p) := \{(D, \vec{v}_\sigma, p_\sigma) | (MS, RS, \sigma) \in p \rhd D\}$$

Template definitions are written as: $\underline{template}\ r(D, v_1, \ldots, v_n) := p$
r is called the template name, $\vec{v}_\sigma$ is the extraction tupel and $v_1, \ldots, v_n$ are called extraction variables.

EXAMPLE: Consider the case where we want to extract all links from a web page. Therefore we define the following *token-template*:

$$\begin{array}{l} \underline{template}\ link(D, Link, Desc) := \#any \\ \qquad\qquad and\ html(tag = a, href = Link)\ and\ Desc = +not(html\_end(tag = a)) \end{array}$$

The first sub pattern $\#any$ will ignore all tokens as long as the next token is of type *html* and meets the required features $tag = a$ and $href$. After the following subexpression matched and a substitution σ is found, $Link$ and $Desc$ hold the extracted information. Now further alternative matches are checked, for example the $\#any$ expression reads up more tokens until the rest expression of the template matches again. □

2.5.1. Extending Token-Templates

To be able to match more sophisticated syntactic structures, we extend the *token-templates* with the three major concepts of *Template Alternatives, Code Calls and Recursion*

Template Alternatives: To gain more readability for template definitions we enhanced the use of the or operator. Instead of using the or operator in a *token-pattern* like in the template $\underline{template}\ t(D,\vec{v}) := \vec{v} = (p_1\ or\ p_2)$ we can alternatively define two templates:

$$\begin{aligned}
&\underline{template}\ t_1(D,\vec{v_1}) := \vec{v_1} = p_1\\
&\underline{template}\ t_2(D,\vec{v_2}) := \vec{v_2} = p_2
\end{aligned}$$

where the templates t_1 and t_2 have the same name. In fact this does not influence the calculation of the extraction tuples, because we can easily construct this set by the union of t_1 and t_2.

Code Calls: A very powerful extension of the *token-pattern* language is the integration of function/procedure calls within the matching process. We named this extension, *code calls*. A code call may be any arbitrary boolean calculation procedure that can be invoked with instantiated *token-pattern* variables or unbound variables that will be instantiated by the calculation procedure. The following example demonstrates the use of a code call to an database interface function db, that will check if the extracted $Name$ can be found in the database. On success it will return $true$ and instantiates $Birth$ to the birthday of the person, otherwise the match fails. In this example we will leave out the *token-pattern* for the recognition of the other extraction variables and will simply name them p_1 to p_3 :

$$\begin{aligned}
&\underline{template}\ person(D, Name, Str, Tel, Birth) := \#any\ and\\
&\qquad Name = p_1\ and\ \{db(Name, Birth)\}\ and\ Str = p_2\ and\ Tel = p_3
\end{aligned}$$

Especially the use of logic programs as code to be called during the matching process, can guide the information extraction with additional deduced knowledge. For example, this can be achieved by a given background theory and facts extracted by preceeding sub-patterns.

Template Calls & Recursive Templates: To recognize hierarchical syntactic structures in text documents (e.g. tables embedded in tables) it is obvious to use recursive techniques. Quite often the same sub-pattern has to be used in a template definition, therefore we extended the *token-pattern* by template calls. A template call may be interpreted as an inclusion of the *token-pattern* associated with the template to be called. For example the first example template matches a HTML table row existing of 3 columns, where the first two are text columns and the third contains price information. The terms set in squared brackets function as template calls. Repeated application of this pattern, caused by the sub pattern $\#any$, gives us all table entries. The second template demonstrates a recursive template call, that matches correct groupings of parenthesises.For a more detailed description of the *token-template* language the reader is referred to [27].

$$\begin{aligned}
&\underline{template}\ table\_row(D, Medium, Label, Price) := (\#any\\
&\qquad and\ html(tag = tr)\\
&\qquad and\ [text\_col(Medium), text\_col(Label), price\_col(Price)]\\
&\qquad and\ html\_end(tag = tr))
\end{aligned}$$

$$\begin{aligned}
&\underline{template}\ correct\_paren(D, A) := \ A = (paren\_open\ and\\
&\qquad (*word\ or\ *\ word\ and\ [correct\_paren(\_)]\ and\ *\ word\ )\\
&\qquad and\ paren\_close\ )
\end{aligned}$$

3. Logic Programs and Token-Templates

In this section we will explain how *token-templates* can be merged with logic programs (LP's). The basic idea of the integration of *token-templates* and LP's is to extend a logic program with a set of *token-templates* (extended LP's), that are interpreted as special program clauses. The resulting logic program can then answer queries about the contents of one or more web documents. Intuitively *token-templates* provide a set of facts to be used in logic programs. Extended LP's offer the possibility to derive new facts based on the extracted facts from the WWW. From the implementational point of view these *token-template* predicates may be logical programs or modules that implement the downloading of web pages and the token matching. From the theoretical point of view we consider these template sets to be axiomatizations of a theory, where the calculation of the theory (the facts) are performed by a background reasoner.

In the following we refer to *normal logic programs* when we talk about logic programs. In Section 3.1 we describe how *token-templates* are interpreted in the context of first order predicate logic. The extension of a calculus with *template theories*, which will lead to the $\mathcal{T}_T$–$Calculus$, is defined in Section 3.2. Section 4.1 and 4.2 will give some small examples for the use of *token-templates* in logic programs.

We assume the reader to be familiar with the fields of *logic programming* [16] and *theory reasoning* [2] [26].

3.1. Template Theories

In the context of first order predicate logic (PL1) we interpret a set of *token-templates* to be an axiomatization of a theory. A *token-template theory* $\mathcal{T}_T$ is the set of all template ground atoms, that we obtain by applying all templates in T. For example, consider the template set $\{t(D,v,p)\}$. Assume p to be an arbitrary *token-pattern* and v an extraction tupel. A *template theory* for T is given by $\mathcal{T}_{\{t(D,v,p)\}} := \{t(D,v',P')|(D,v',p') \in t(D,v,p)\}$. This interpretation of *token-templates* associates a set of ground unit clauses with a given set of *token-templates*. The formal definition is as follows:

Definition 4 ($Template\!-\!Theory$, $\mathcal{T}_T\!-\!Interpretation$, $\mathcal{T}_T\!-\!Model$). Let T be a set of token-templates: $T = \{t_1(D,\vec{v}_1,p_1), t_2(D,\vec{v}_2,p_2),\ldots,t_n(D,\vec{v}_n,p_n)\}$. A *token-template theory* $\mathcal{T}_T$ for T is defined as follows:

$$G(t(D,\vec{v},p)) = \{t(D,\vec{v}',p')|D \textit{ is a ground term and } (D,\vec{v}',p') \in t(D,\vec{v},p)\}$$

$$\mathcal{T}_T = \bigcup_{i=1}^{n} G(t_i(D,\vec{v}_i,p_i)) \ \textit{ with } t_i(D,\vec{v}_i,p_i) \in T$$

Let P be a normal logic program with signature Σ, such that Σ is also a signature for $\mathcal{T}_T$:

- A $\Sigma\!-\!Interpretation$ $\mathcal{I}$ is a $\mathcal{T}_T$–$\Sigma\!-\!Interpretation$ *iff* $\mathcal{I} \models \mathcal{T}_T$.
- A Herbrand $\mathcal{T}_T$–Σ–$Interpretation$ is a $\mathcal{T}_T$–Σ–$Interpretation$, that is also a herbrand interpretation.
- A $\mathcal{T}_T$–$\Sigma\!-\!Interpretation$ $\mathcal{I}$ is a $\mathcal{T}_T\!-\!Model$ for P *iff* $\mathcal{I} \models P$.

- Let X be a clause, wrt. Σ. X is a logic $\mathcal{T}_T$–$Consequence$ from P, $P \models_{\mathcal{T}_T} X$, *iff* for all $\mathcal{T}_T$–$\Sigma-Interpretation$ $\mathcal{I} : \mathcal{I} \models P$ follows $\mathcal{I} \models X$.

EXAMPLE: Consider a *token-template advertise* with the *token-pattern* given in Figure 5 and the extraction variables *Item* and *Description* for the example web source code shown in Figure 1. The corresponding *template theory* for the template *advertise* is the set:

{ $advertise$('*286 AT*',' *1 MB Ram, 60 MB FP, 2 Disklaufw., Bigtower, 100 DM,*
4 x 1 MB SIMM 30 Pin, 50 DM. Tel.: 06742/ 5926'),
$advertise$('*Pentium 90*',' *48 MB RAM, Soundblaster AWE 64, DM 650,-.*
Tel.: 06743/ 1582') }

□

3.2. The $\mathcal{T}_T$–Calculus

So far we have shown when a formula is a logical consequence from a logic program and a *template theory*. This does not state how to calculate or check if a formula is a logical consequence from an *extended logic program*. Therefore we have to define a calculus for *extended logic programs*. But instead of defining a particular calculus we show that any sound and answer-complete calculus for normal logic programs can serve as calculus for extended logic programs.

Definition 5 ($\mathcal{T}_T$–$Derivation$). Let $\mathcal{K}$ be a sound and answer-complete calculus for normal logic programs and $\vdash$ is the derivation defined by $\mathcal{K}$. Let P be a normal logic program, T a set of *token-templates* and $\mathcal{T}_T$ the template theory for T. A query $\exists\, Q$ with calculated substitution σ is $\mathcal{T}_T$–$derivable$ from P, $P \vdash_{\mathcal{T}_T} Q\sigma$, *iff* $Q\sigma$ is derivable from $P \cup \mathcal{T}_T$, $P \cup \mathcal{T}_T \vdash Q\sigma$. A calculus $\mathcal{K}$, with $\mathcal{T}_T$–$Derivation$ is called $\mathcal{T}_T$–$Calculus$.

THEOREM 1 (SOUNDNESS) *Let $\mathcal{K}$ be a $\mathcal{T}_T$–$Calculus$ and $\vdash_{\mathcal{T}_T}$ the derivation relation defined by $\mathcal{K}$. Let T be a set of token-templates, $\mathcal{T}_T$ the template-theory for T and Q a query for a normal program P. Further let σ be a substitution calculated by $\vdash_{\mathcal{T}_T}$. Then $\forall\, Q : P \vdash_{\mathcal{T}_T} Q\sigma \Rightarrow P \models_{\mathcal{T}_T} \forall\, Q_\sigma$*

THEOREM 2 (COMPLETENESS) *Let $\mathcal{K}$ be $\mathcal{T}_T$–$Calculus$ and $\vdash_{\mathcal{T}_T}$ the derivation relation defined by $\mathcal{K}$. Let T be a set of token-templates, $\mathcal{T}_T$ the template-theory for T and Q a query for a normal program P. Let θ be a correct answer for Q, σ a calculated answer and γ a substitution.*
Then $\forall\, \theta : P \models_{\mathcal{T}_T} Q\theta \Rightarrow P \vdash_{\mathcal{T}_T} Q\sigma$ where $\theta = \sigma\gamma$

Soundness and Completeness: Let $\mathcal{K}$ be a $\mathcal{T}_T$–$Calculus$ and $\vdash_{\mathcal{T}_T}$ the derivation relation defined by $\mathcal{K}$. Let T be a set of templates , $\mathcal{T}_T$ the template theory of T and Q a query on the normal program P. Let σ be a calculated substitution for Q such that:

$$\begin{array}{lll} & P \vdash_{\mathcal{T}_T} Q\sigma & \\ \Leftrightarrow & P \cup \mathcal{T}_T \vdash Q\sigma & \textit{def. 5} \end{array}$$

$$\begin{array}{ll}\Leftrightarrow & P \cup \mathcal{T}_T \models \forall\, Q\sigma \quad \textit{sound/completness of } \vdash \\ \Leftrightarrow & P \models_{\mathcal{T}_T} \forall\, Q\sigma\end{array}$$

To prove: $P \cup \mathcal{T}_T \models \forall\, Q\sigma \Leftrightarrow P \models_{\mathcal{T}_T} \forall\, Q\sigma$

$'' \Rightarrow''$:

$$\begin{array}{lll} & P \cup \mathcal{T}_T \models \forall\, Q\sigma & \\ \Leftrightarrow & \forall\, \mathcal{I} : \mathcal{I} \models P \cup \mathcal{T}_T \Rightarrow \mathcal{I} \models \forall\, Q\sigma & \textit{def. logical consequence} \\ \Leftrightarrow & \forall\, \mathcal{I} : (\mathcal{I} \models P \textit{ and } \mathcal{I} \models \mathcal{T}_T) \Rightarrow \mathcal{I} \models \forall\, Q\sigma & \textit{def. interpretation}\end{array}$$

$'' \Leftarrow''$:

$$\begin{array}{lll} & P \models_{\mathcal{T}_T} \forall\, Q\sigma & \\ \Leftrightarrow & \forall\, \mathcal{T}_T\text{–}Interpret.\, \mathcal{I} : (\mathcal{I} \models P \Rightarrow \mathcal{I} \models \forall\, Q\sigma) & \textit{def. } \mathcal{T}_T\text{–}Consequence \\ \Leftrightarrow & \forall \mathcal{I} : \mathcal{I} \models \mathcal{T}_T \Rightarrow (\mathcal{I} \models P \Rightarrow \mathcal{I} \models \forall\, Q\sigma) & \textit{def. } \mathcal{T}_T\text{–}Interpretation\end{array}$$

$$\begin{array}{l}\forall\, \mathcal{I} : (\mathcal{I} \models P \textit{ and } \mathcal{I} \models \mathcal{T}_T) \Rightarrow \mathcal{I} \models \forall\, Q\sigma \Leftrightarrow \\ \qquad \forall\, \mathcal{I} : \mathcal{I} \models \mathcal{T}_T \Rightarrow (\mathcal{I} \models P \Rightarrow \mathcal{I} \models \forall\, Q\sigma)\end{array}$$

$$\begin{array}{l}\forall\, \mathcal{I} : (\mathcal{I} \models P \textit{ and } \mathcal{I} \models \mathcal{T}_T) \Rightarrow \mathcal{I} \models \forall\, Q\sigma \Leftrightarrow \\ \qquad \forall\, \mathcal{I} : (\mathcal{I} \models \mathcal{T}_T \textit{ and } \mathcal{I} \models P) \Rightarrow \mathcal{I} \models \forall\, Q\sigma\end{array}$$

■

EXAMPLE: In Figure 6 an example $\mathcal{T}_T$–$Derivation$ based on the SLD-Calculus [12] is shown. The calculation of the *template theory* is done by a theory box [2], this may be any arbitrary calculation procedure, that implements the techniques needed for *token-templates*. Furthermore this theory box has to decide if a template predicate, like *institute('http://www.uni-koblenz.de',Z,P)* can be satisfied by the calculated theory. Let us have a closer look at the logic program *P* given in Figure 6:

b(uni,'http://www.uni-koblenz.de') A given web page containing some information about a university.

a(X,Z) ← b(X,Y), institute(Y,Z,P) An institution X has an department Z if there exists an web page Y describing X and we are able to extract department names Z from Y.

With this given logic program and the template definition T we can find a SLD–$\mathcal{T}_T$–$Derivation$, assumed our template theory is not empty. What this example shows is: modeling knowledge about web pages by logic programs and combining this with token-templates allows us to query web pages. □

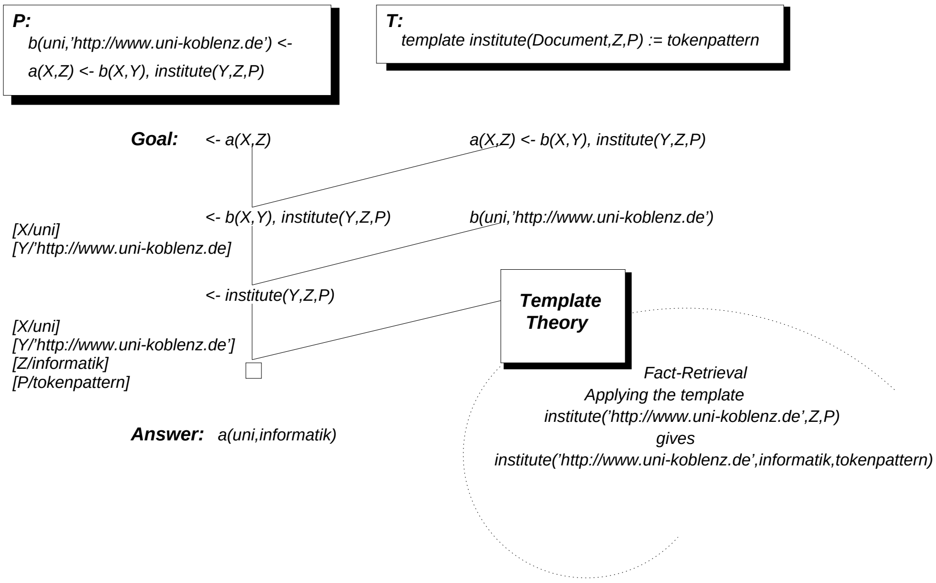

P:
b(uni,'http://www.uni-koblenz.de') <-
a(X,Z) <- b(X,Y), institute(Y,Z,P)
T:
template institute(Document,Z,P) := tokenpattern
Goal:
<- a(X,Z)
a(X,Z) <- b(X,Y), institute(Y,Z,P)
<- b(X,Y), institute(Y,Z,P)
b(uni,'http://www.uni-koblenz.de')
[X/uni]
[Y/'http://www.uni-koblenz.de]
<- institute(Y,Z,P)
Template Theory
[X/uni]
[Y/'http://www.uni-koblenz.de']
[Z/informatik]
[P/tokenpattern]
Fact-Retrieval
Applying the template
institute('http://www.uni-koblenz.de',Z,P)
gives
institute('http://www.uni-koblenz.de',informatik,tokenpattern)
Answer:
a(uni,informatik)

Figure 6. $SLD - \mathcal{T}_T\text{–}Derivation$

4. Deductive Techniques for Intelligent Web Search

Logic programming and deduction in general offer a wide variety to guide the web search and fact-retrieval process with intelligent methods and inference processes. This section describes some of these techniques.

4.1. Deductive Web Databases

Assume we know two web pages of shoe suppliers, whose product descriptions we want to use as facts in a deductive database. Additionally we are interested in some information about the producer of the product, his address and telephone number that can be retrieved from an additional web page. Therefore we define two *token-templates*, $price\_list$ and $address$. To simplify notation we leave out the exact *token-pattern* definitions instead we write p_1 and p_2 .

The following small deductive database allows us to ask for articles and to derive new facts that provide us with information about the product and the producer. We achieve this by the two rules *article* and *product*, which extract the articles offered at the web pages and will derive new facts about the article and the producer.

$\underline{template}\ price\_list(Document, Article, Price, ProducerUrl) = p_1$
$\underline{template}\ address(Document, Producer, Str, Tel, City) = p_2$

$web\_page('ABC-Schuhe', http://www.abcshoes.com/offer.html) \leftarrow$
$web\_page('Schuhland', http://www.shland.com/angebot.html) \leftarrow$

$article(Supplier, Article, Price, ProducerUrl) \leftarrow$
$\quad web\_page(Supplier, Document),$
$\quad price\_list(Document, Article, Price, ProducerUrl, Pattern)$

$product(Supplier, Producer, Article, Price, Str, Tel, City) \leftarrow$
$\quad article(Supplier, Article, Price, ProducerUrl),$
$\quad address(ProducerUrl, Supplier, Str, Tel, City, Pattern)$

EXAMPLE: Here are some example queries to demonstrate the use of the deductive web database:

Select all products with article name "$Doc\ Martens$" that cost less than 100:
$\leftarrow Article = \text{"}Doc\ Martens\text{"},$
$\quad product(Supplier, Article, Producer, Price, Str, Tel, City), Price < 100$

Select all products offered at least by two suppliers:
$\leftarrow product(Supplier1, Producer, Article, Price1, Str1, Tel1, City1),$
$\quad product(Supplier2, Producer, Article, Price2, Str2, Tel2, City2),$
$\quad Supplier1 \backslash= Supplier2$ $\Box$

4.2. *Optimizing Web Retrieval*

The following example shows how a query optimization technique proposed by Levy [19] can be implemented and used in *extended logic programs*. To avoid the fetching of senseless web pages and starting a *fact-retrieval* process we know for certain to fail, Levy suggests the use of source descriptions. For the fact retrieval from the WWW this might offer a great speed up, because due to the network load the fetching of web documents is often very time intensive. In the context of *extended logic programs*, we can easily apply these methods, by the definition of rules, whose body literals define constraints on the head arguments expressing our knowledge about the content of the web pages. The following example illustrates these methods:

$offer('http://www.autos.com/biete.html', Price, Country) \leftarrow$
$\quad Price > 20000, Price < 40000, Country = 'Deutschland'$
$offer('http://www.cars.com/sells.html', Price, Country) \leftarrow$
$\quad Price > 40000, Price < 60000, Country = 'USA'$
$product(Car, Price, Country) \leftarrow$
$\quad offer(WebPage, Price, Country),$
$\quad template\_cars(WebPage, Price, Country)$

Assuming we are interested in american cars that costs 50000 dollars, the query *product(C,50000,'USA')* will retrieve the according offers. Because of the additional constraints on the price and the country given in the body of the rule `offer`, the irrelevant web page with german car offers is left out. By simple methods, provided by the logic programming paradigm for free, we are able to guide the search and fact retrieval in the world wide web based on knowledge representation techniques [3] and we are able to speed up the search for relevant information.

4.3. Conceptual Reasoning

Many information systems lack of the ability to use a conceptual background knowledge to include related topics of interest to their search. Consider the case that a user is interested in computer systems that cost less than 1000 DM. It is obvious that the system should know the common computer types and descriptions (e.g. IBM, Toshiba, Compaq, Pentium, Notebook, Laptop) and how they are conceptually related to each other. Such knowledge will assist a system in performing a successful search. One way to represent such knowledge is by concept description languages or in general by means of knowledge representation techniques. In the last few years it showed up that logic is a well suited analytic tool to represent and reason about represented knowledge. Many formalisms have been implemented using logic programming systems, for example *PROLOG*.

For example a simple relation is_a can be used to represent conceptual hierarchies to guide the search for information. Consider the following small knowledge base:

$$\begin{array}{rcl}
notebook('ThinkPad') & \leftarrow & \\
notebook('Satellite') & \leftarrow & \\
 & & \\
is\_a(notebook, computer) & \leftarrow & \\
is\_a(desktop, computer) & \leftarrow & \\
is\_a(X, notebook) & \leftarrow & notebook(X) \\
 & & \\
relevant(Q, Z) & \leftarrow & is\_a(Z, Q) \\
relevant(Q, Z) & \leftarrow & is\_a(Y, Q), relevant(Y, Z)
\end{array}$$

Assume our general query to search for computers less than 100 DM is split into a sub query like $\leftarrow relevant(computers, X)$ to our small example knowledge base. The query computes a set of answers:

$$\theta = \{[X/notebook], [X/desktop], [X/'ThinkPad'], [X/'Satellite']\}.$$

This additional inferred query information θ can be used in two different ways:

1. The derived conceptual information θ is used to search for new web pages, e.g. by querying standard search engines with elements of θ as search keywords. On the returned candidate pages further extended logic programs can be applied to extract facts.

2. The information extraction process itself is enhanced with the derived information θ by reducing the constraints on special token features in the token-templates to be applied. Consider the case where only a single keyword Q is used with a token pattern to constrain the matching process e.g. *word(txt=Q)*. We can reduce this constrainedness by constructing a more general (disjunctive) pattern by adding simple term-patterns, whereas their feature values consist of the deduced knowledge θ, e.g. *word(txt=Q) or word(txt=*θ_1*) or* ... *or word(txt=*θ_n*)* with $\theta_i \in \theta$. That means we include to the search all sub concepts (e.g. $computer \rightarrow notebook \rightarrow \ldots$) or instances (e.g. $'ThinkPad'$) of the query concept.

5. The LogicRobot

This section will give a short overview of an application we implemented using the described methods and techniques. A brief explanation of a domain dependent search engine for an online advertisement newspaper is given.

5.1. The Problem

Very often web pages are organized by a chain of links the user has to follow to finally reach the page he is interested in. Or the information the user wants to retrieve is split into many pages. In both cases the user has to visit many pages to finally reach the intended page or to collect data from them. To do this manually is a very exhausting and time consuming work and furthermore it is very difficult for the user to take comparisons between the information offered on the various web pages. Therefore an automatic tool to follow all links, to collect the data and to provide the possibility to compare the retrieved information is needed to free the user from this annoying work.

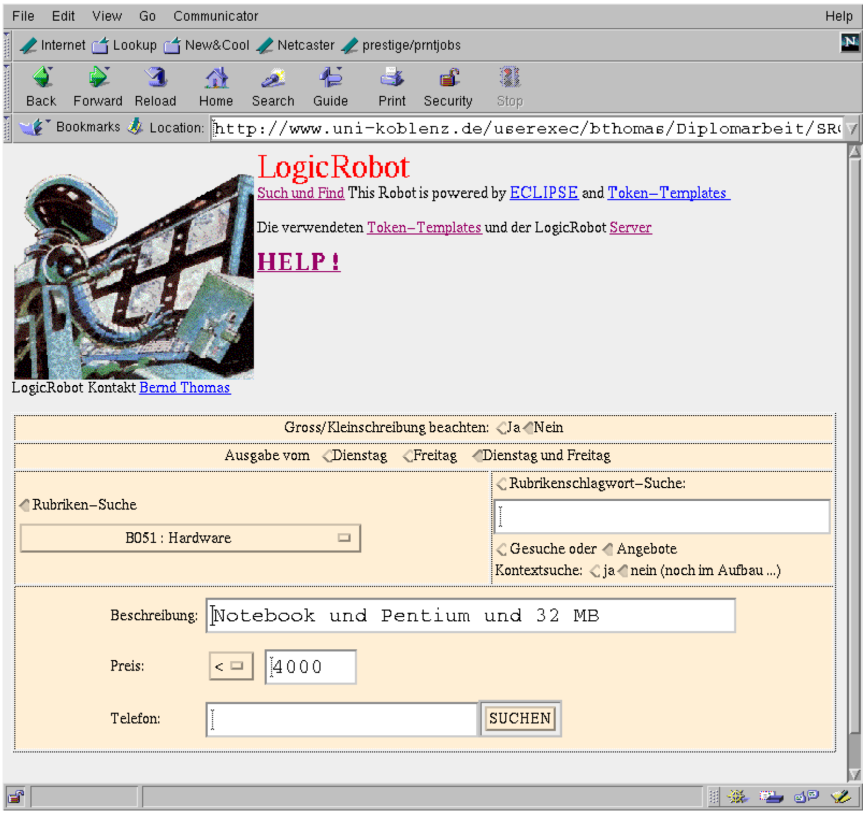
We call web information systems based on *logic programs* and *token-templates LogicRobots*. Similar to physical robots they navigate autonoumasly through their environment, the web. According to their ability to analyze and reason on web page contents and the incorporation of knowledge bases they are able to percept their environment, namely what's on a web page. Due to the underlying logic program and the used AI techniques, e.g. knowledge representation, default reasoning etc., they act by collecting facts or follow up more promising links. The problem we focused on was to build a *LogicRobot* for a web vendor offering private advertisements. Some of the offered columns forces the user to follow about 80 to 100 links to see all advertisements, which is of course not very user-friendly. A more elegant way would be to offer a web form where the user can specify the column or columns to be searched either by entering a specific name or a keyword for a column name, a description of the item he is searching for, a price constraint like less, greater or equal to and finally a pattern of a telephone number to restrict the geographical area to be searched. Figure 7 depicts the three main templates used for extracting information about advertisements similar to those shown in Figure 4. The *LogicRobot* web interface for this special task is presented in Figure 8 and a sample result page is given in Figure 9.

```
template price(Price) := once( # any and ? (word(value='VB') or word(value='FP'))
                  and word(value='DM') and ( (int(value=P) and {cfloat(P,Price)})
                  or (float(value=P) and {price_notation(P,Price)})))

template telefon(T) := # any and word(value='Tel') and ? punct(value='.')
                  and ? punct(value=':') and once(T = ( ? op(value='+') and +int
                  and ? (punct or op) and +int)))

template product_description(Article,Desc) := # any and html(tag=img) and html(tag=b)
                  and Article = * not(html_end(tag=b)) and html_end(tag=b)
                  and once(Desc = * not(html(tag=p)))
```

Figure 7. Templates used for telefon number, price and advertised product extraction



http://www.uni-koblenz.de/userexec/bthomas/Diplomarbeit/SR(

LogicRobot

Such und Find This Robot is powered by ECLIPSE and Token–Templates

Die verwendeten Token–Templates und der LogicRobot Server

HELP !

LogicRobot Kontakt Bernd Thomas

Gross/Kleinschreibung beachten: ◇Ja ◆Nein

Ausgabe vom ◇Dienstag ◇Freitag ◆Dienstag und Freitag

◆Rubriken–Suche

B051 : Hardware

◇Rubrikenschlagwort–Suche:

◇Gesuche oder ◆Angebote

Kontextsuche: ◇ja ◆nein (noch im Aufbau ...)

Beschreibung: Notebook und Pentium und 32 MB

Preis: < 4000

Telefon:

SUCHEN

Figure 8. The LogicRobot web interface

5.2. Implementational Notes

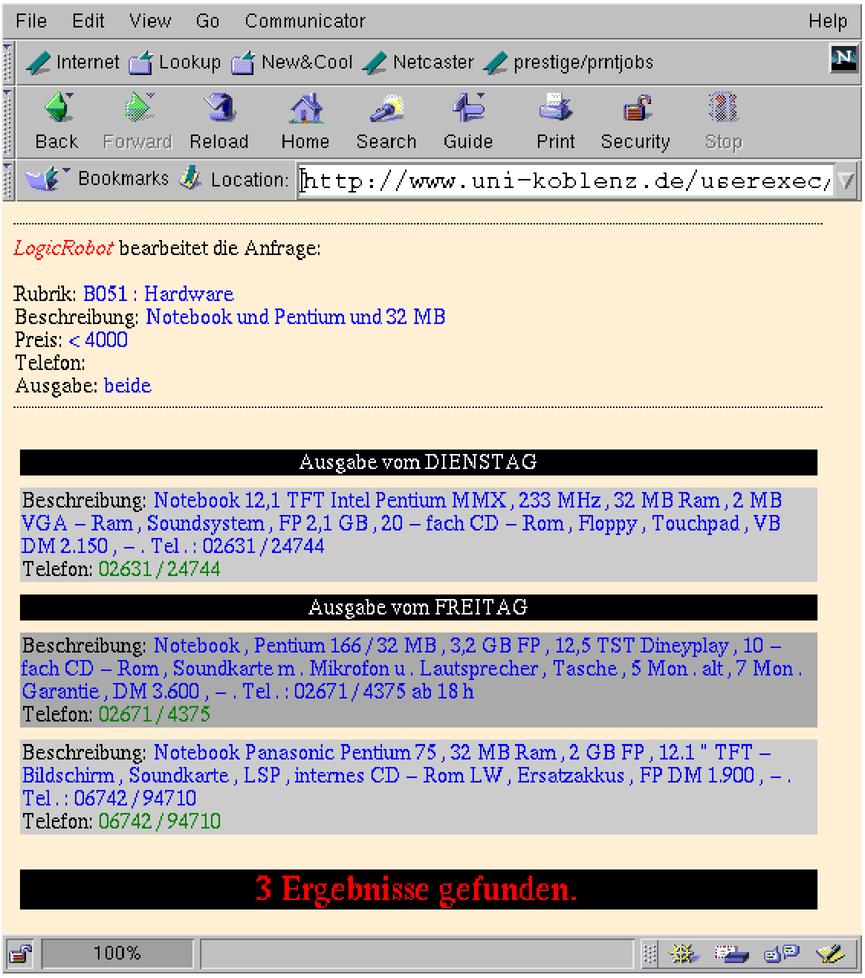
The *LogicRobot* for the search of advertisements is based on the logic programming library *TXW3* [28] that implements the techniques presented in this paper using *ECLIPSE*-Prolog [8]. *ECLIPSE* supports modularized logic programming, so we modularized the architecture of the *LogicRobot* into two main modules, the first module containing all needed token-template definitions and the second the prolog program implementing the appropriate template calls, the evaluation of the price constraints and further control operations. This prolog module is executed by the *CGI* mechanism and communicates with the local *http daemon* via *stdin/stdout* ports. So there is no additional server programming or network software needed to setup a search engine based on *extended logic programs*.

By only 5 template definitions and approx. 200 lines of prolog code we implemented this *LogicRobot*. The tests we carried out with our application are very promising. For example the query answering time, which contains fetching, tokenizing, extracting and comparing is beneath 2 minutes for 100 web pages of advertisements (Figure 1) . We think this is a very promising way for a domain specific search tool, that can be easily extended by AI methods, that offers a flexible and fast configurability by means of declarative definitions (e.g. using *PROLOG*) and most important this concept of *LogicRobots* can be applied to various information domains on the World Wide Web.

6. Related Work and Conclusion

We presented the *token-template* language for the IE from semistructured documents, especially from web pages. We showed how our wrapper language can be merged with logic programs and gave a formal definition for the extension of an arbitrary answer complete first order logical calculus with *template theories*. In conjunction with the area of logic programming and deductive databases we can use these wrapper techniques to obtain inferences or new deductively derived facts based on information extracted from the WWW. Furthermore these methods can be used to build intelligent web information systems, like *LogicRobots*, that gain from the closely related areas like deductive databases, knowledge representation or logic programming based AI methods. We also showed how already developed query optimization techniques (Section 4.2), can easily be integrated into our approach. Our methods have been successfully integrated and used in the heterogeneous information system *GLUE* [20] to access web data and integrate it into analytical and reasoning processes among heterogeneous data sources (e.g. relational databases, spreadsheets, etc.). In addition to our theoretical work we also implemented a logic programming library *TXW3* that provides the language of *token-templates* and various other logic modules to program *LogicRobots* for the WWW.

Several web information systems have been developed in the last few years. One class of applications called *Softbots*, which are domain specific automated search tools for the WWW, searching autonomously for relevant web pages and user requested informations, are similar to our concept of a *LogicRobot*. But such existing systems like *Ahoy!* [24] or *Shopbot* [7] use either tailored extraction techniques (*Ahoy!*) that are very domain specific or their extraction techniques are based on highly restrictive assumptions about the syntactical structure of a web page (*Shopbot*). Both systems do not follow the concept of a general


File Edit View Go Communicator Help
Internet Lookup New&Cool Netcaster prestige/prntjobs
Back Forward Reload Home Search Guide Print Security Stop
Bookmarks Location: http://www.uni-koblenz.de/userexec/
100%

LogicRobot bearbeitet die Anfrage:

Rubrik: B051 : Hardware
Beschreibung: Notebook und Pentium und 32 MB
Preis: < 4000
Telefon:
Ausgabe: beide

Ausgabe vom DIENSTAG

Beschreibung: Notebook 12,1 TFT Intel Pentium MMX , 233 MHz , 32 MB Ram , 2 MB VGA – Ram , Soundsystem , FP 2,1 GB , 20 – fach CD – Rom , Floppy , Touchpad , VB DM 2.150 , – . Tel . : 02631 / 24744
Telefon: 02631 / 24744

Ausgabe vom FREITAG

Beschreibung: Notebook , Pentium 166 / 32 MB , 3,2 GB FP , 12,5 TST Dineyplay , 10 – fach CD – Rom , Soundkarte m . Mikrofon u . Lautsprecher , Tasche , 5 Mon . alt , 7 Mon . Garantie , DM 3.600 , – . Tel . : 02671 / 4375 ab 18 h
Telefon: 02671 / 4375

Beschreibung: Notebook Panasonic Pentium 75 , 32 MB Ram , 2 GB FP , 12.1 " TFT – Bildschirm , Soundkarte , LSP , internes CD – Rom LW , Ersatzakkus , FP DM 1.900 , – . Tel . : 06742 / 94710
Telefon: 06742 / 94710

3 Ergebnisse gefunden.

Figure 9. Query result page

purpose extraction language like *token-templates* are. *Token-templates* are applicable to any kind of semistructured text documents, and hence not restricted to a specific domain.

Systems like *IM* [18] or *W3QS* [14] also provide means to query web information sources. Though Levy et. al. also choose a relational data model to reason about data, and show several techniques for source descriptions or constructing query plans, they leave the problem of information extraction undiscussed in their work. We showed solutions for both the extraction of facts and reasoning by extended logic programs. The *W3QS* system uses a special web query language similar to the relational database query language *SQL*. *W3QS* uses enhanced standard SQL commands, e.g. by additional external unix program calls or HTML related commands. Though an additional construction kit for information extraction processes is given, this seems to be focused only on the detection of hyper links and their descriptions. The concept of database views for web pages is also introduced, but no information about recursive views is provided, whereas extended logic programs offer these abilities.

Heterogeneous information systems, like *DISCO* [29], *GLUE* [20], *HERMES* [22], *Infomaster* [9] or *TSIMMIS* [5] all use special mediator techniques to access web information sources among other data sources. These systems use their own mediator model (language) to interface with the special data source wrappers. The system *HERMES* for example is based on a declarative logical mediator language and therefore is similar to our approach using extended logic programs as mediators and *token-templates* as special wrapper language. The advantage of our presented approach is simply, that the above named systems except *TSIMMIS* and *GLUE* do not incorporate a general purpose wrapper language for text documents. Additionally work on the expressive power of the mediator languages and the used wrapper techniques of the other systems is of interest.

Different from the template based extraction languages described in [11] and [6] or the underlying language used in the wrapper construction tool by Gruser et. al. [10], *token-templates* incorporate the mighty concepts of recursion and *code calls*. These concepts allow the recognition and extraction of arbitrary hierarchical syntactic structures and extends the matching process by additional control procedures invoked by code calls. Especially logic programs used as *code calls* can guide the extraction process with a manifold of AI methods in general.

Notes

[1] In the sense that the feature set of the *left* token must be a subset of the feature set of the *right* token.

[2] see [27] for a detailed formal definition.

References

1. N. Ashish and C. Knoblock. Wrapper generation for semistructured internet sources. In *Proceedings of the Workshop on Management of Semi-structured Data*, 1997.
2. Peter Baumgartner. *Theory Reasoning in Connection Calculi and the Linearizing Completion Approach*. PhD thesis, University Koblenz, 1997.
3. Gerhard Brewka, editor. *Principles of Knowledge Representation*. CSLI Publications, 1996.
4. Bob Carpenter. Typed Feature Structures: an Extension of First-order Terms. In *Proceedings of the International Symposium on Logic Programming*, 1991. San Diego.
5. S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, Y. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *Proceedings of IPSJ*, 1994. Japan.
6. Mary Elaine Califf and Raymond J. Mooney. Relational Learning of Pattern-Match Rules for Information Extraction. In *Working Papers of the ACL-97 Workshop in Natural Language Learning*, 1997.
7. R. Doorenbos, O. Etzioni, and D. Weld. A scalable comparison-shopping agent for the world-wide web. In *Proceedings of Autonomous Agents*, pages 39–48. New York: Association of Computing Machinery, 1997.
8. International Computers Limited and IC-Parc, http://www.ecrc.de/eclipse/. *ECLiPSe User Manual / Extensions User Manual – Release 3.6*, 1997. Two volumes.
9. M. R. Genesereth, A. M. Keller, and O. Duschka. Infomaster: An Information Integration System. *Proceedings of ACM SIGMOD Conference*, May 1997.
10. J. Gruser, L. Raschid, M. Vidal, and L. Bright. A wrapper generation toolkit to specify and construct wrappers for web accesible data. Technical report, UMIACS, University of Maryland, 1998.

11. J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. Extracting semistructured information from the web. In *In Proceedings of the Workshop on Management of Semistructured Data*, May 1997.
12. R. Kowalski and D. Kuehner. Linear Resolution with Selection Function. *Artificial Intelligence*, (2):227–260, 1971.
13. Kevin Knight. Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, March 1989.
14. David Konopnicki and Oded Shmueli. W3QS: A query system for the world-wide web. In *Proceedings of VLDB'95*, 1995.
15. Nicholas Kushmerick, Daniel S. Weld, and Robert Doorenbos. Wrapper Induction for Information Extraction. In Martha E. Pollack, editor, *Fifteenth International Joint Conference on Artificial Intelligence*, volume 1, pages 729–735, August 1997. Japan.
16. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2 edition, 1987.
17. John Levine, Tony Mason, and Doug Brown. *Lex & Yacc*. O'Reilly and Associates., 1990.
18. Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Query-Answering Algorithms for Information Angents. In *Proceedings of the 13th National Conference on Artificial Intelligence*, 1996. Portland, Oregon, USA.
19. Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *Proceedings of the 22nd VLDB Conference*, 1996. Mumbai(Bombay), India.
20. Gerd Neugebauer and Dorothea Schäfer. GLUE - Opening the World to Theorem Provers. In *Proceedings of Logic Programming and Non-Monotonic Reasoning 97*, July 1997. Dagstuhl, Germany.
21. Louiqa Raschid, Maria Esther Vidal, and Jean-Robert Gruser. A flexible meta-wrapper interface for autonomous distributed information sources. Technical Report AR_309, University of Maryland, Institute for Advanced Computer Studies- Dept. of Computer Sience, March 1997.
22. V.S. Subrahmanian, Sibel Adali, Anne Brink, Ross Emery, James J. Lu, Adil Rajput, Timothy J. Rogers, Robert Ross, and Charles Ward. HERMES: A Heterogeneous Reasoning and Mediator System, 1996. http://www.cs.umd.edu//projects/hermes/overview/paper/index.html.
23. Stuart M. Shieber. *An Introduction to Unification-Based Approaches to Grammar*. CSLI, Leland Stanford Junior University, 1986. CSLI Lecture Notes 4.
24. Jonathan Shakes, Marc Langheinrich, and Oren Etzioni. Dynamic reference sifting: A case study in the homepage domain. In *Proceedings of the Sixth International World Wide Web Conference*, pages 189–200, 1997.
25. Gert Smolka and Ralf Treinen. Records for Logic Programming. *Journal of Logic Programming*, 18:229–258, 1994.
26. M.E. Stickel. Automated Deduction by Theory Resolution. *Journal of Automated Reasoning*, 1:333–355, 1985.
27. Bernd Thomas. Token-Templates und Logisches Programmieren im World-Wide-Web. Master's thesis, University of Koblenz-Landau, Abteilung Landau, Institut für Informatik, March 1998.
28. Bernd Thomas. The txw3-module, March 1998. http://www.uni-koblenz.de/∼bthomas/TXW3.html.
29. Anthony Tomasic, Louiqa Raschid, and Patrick Valduriez. Scaling Heterogeneous Databases and the Design of Disco. Technical Report 2704, INRIA, Institut National De Recherche En Informatique Et En Automatique, Novembre 1995.
30. G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, pages 38–49, March 1992.