UKernel: A Unification Kernel*

Benjamin Han, Alon Lavie
{benhdj | alavie}@cs.cmu.edu

Language Technologies Institute
Carnegie Mellon University

August 13, 2004

*This paper describes UKernel version 1.92.

Contents

List of Figures

Listings

List of Tables

1 Introduction

This paper describes *UKernel* (pronounced as You-Kernel), a module providing unification services. The purpose of having a stand-alone module providing unification services is for greater modularity - UKernel can be either integrated with a natural language parser or generator which adopts a unification-based grammar formalism [3]. A suitable front-end is then needed to parse the actual grammar rules into the inernal UKernel data structures.

UKernel is essentially a re-implementation of the unification part of GenKit [5] in C++[1]. Implemented in LISP in 1988 by Tomita and Nyberg, GenKit is a natural language sentence generator using a unification-based grammar formalism. One of its notable characteristics is that for simplicity and efficiency, GenKit implemented *pseudo-unification* rather than full-unification [4]. UKernel inherits the use of pseudo-unification and provides an efficient re-implements of almost all of the functionalities of GenKit. In addition, UKernel implements a more flexible lexicon interface, corrects several implementation inaccuracies in GenKit (such as the interactions between complex and atomic values and feature structures), and adds several extentions. See Appendix A for a list of differences between GenKit and UKernel.

As a package UKernel also provides an interactive shell, *UShell*, for experimenting most of the supported feature structure operations. This is useful for beginners to learn different operators, as well as helpful for system/grammar developers to pinpoint their problems.

As mentioned UKernel is implemented in C++, with heavy use of STL (Standard Template Library). The implementation depends on another independent library *Toolbox*, which provides the code for creating and manipulating generic tree and context-free grammar structures. The direct manipulation of strings is restricted in the `symbol` module so internationalization is more straightforward[2]. The equation block interface is also made extension-friendly in that the future developers can add new functions by simply providing the implementations of the functions and defining their arguments interfaces (as is done for the lexical-related functions in the current implementation). Interested readers can refer to Appendix B for more details.

In the rest of the paper a top-down approach is adopted for decomposing the various aspects of UKernel, and efforts have been made in each section to point to the relevant other parts so a reader can quickly find what she needs and only reads the part. The paper is organized as follows. First as an overview Section 2 gives a toy grammar and lexicon. Section 3 to 7 then dissect each part of a grammar/lexicon (in particular the operators are described in Section 7). In practice grammars and lexicons are usually written as modules, and Section 8 describes how to load multiple grammars/lexicons and its implications. Finally Section 9 lists a few suggestions for writing an efficient grammar. Appendix A gives a detailed account of the differences between UKernel and the 1988 GenKit, and is recommended for the users of the 1988 GenKit. Appendix B, which is recommended to system developers who need to incorporate UKernel in their own natural language systems, describes the implementation in more depth and gives a list of future works. Appendix C describes UShell, the interactive shell for UKernel, and Appendix D shows the execution trace of the toy grammar presented in Section 2.

[1]This is not the only C++ re-implementation one can find. In fact a similar core resides in a commercial system KANTOO[1], which is a C++ implementation of an interlingua-based machine translation system, to meet the needs for both language analysis and generation. A detailed comparison between the two systems is not possible as of writing.

To the authors' knowledge UKernel is the only *open source* C++ re-implementation of the unification core in GenKit to date.

[2]Internationalization is listed as one of the future works in Appendix B.2.

2 A Toy Grammar and Lexicon

The functionalities of UKernel are best illustrated by simply showing how a toy grammar and lexicon works. Before doing so however it is important to note that UKernel is agnostic to the direction of language processing - the grammar can be either for language *analysis* (parsing) or *generation*. In either case UKernel is only called to execute a unification *equation block* (see below). A rule is then successfully fired if executing its equation block returns `T` (true), and the subsequent actions are outside the responsibility of UKernel. A second point to make is that UKernel follows the grammar notations used in the 1988 GenKit, and it is up to a higher system (front-end) to parse a grammar/lexicon file into the data structures UKernel requires (these structures include *grammars*, *lexicons, equation blocks*, *equations*, *feature structures, values* and *symbols*; see Appendix B). Note internally UKernel is *case-sensitive*, however a system developer may decide to convert the grammars entirely to lower/uppercase, thus in effect to make the system case-insensitive.

A toy grammar for sentence generation is shown in Listing 1[3]. Note that the execution of a generation grammar starts from the rule with `<START>` as its *left-hand-side* (see line 1 in Listing 1)[4]. Each rule consists of a *c-structure rule* (constituent structure rule) and an *equation block*. A c-structure rule is used to build smaller/larger constituents when the equation block is successfully executed. The language processing direction, i.e., analysis/generation, is indicated by the orientation of the arrow in a c-structure rule. The equation block in a grammar rule is an *AND equation block* which contains a set of equations or equation blocks that are executed sequentially until one of them fails. More details are described in the subsequent sections.

```
1  (<START> ==> (<S>)
2    ((X1 = X0)))
3
4  (<S> ==> (<NP> <VP>)
5    ((X1 == (X0 SUBJ))
6     ((X2 PERS) = (X1 PERS))
7     ((X2 NUM) = (X1 NUM))
8     (X2 = X0)))
9
10 (<N> --> (<BREED> D O G)
11   (((X0 PRED) = DOG)
12    ((X0 NUM) = SG)
13    (X1 = (X0 BREED))))
14
15 (<BREED> ==> (%)
16   ((*CASE* X0
17         (PITBULL (((X1 VALUE) <= BIG_)))
18         (CHIHUAHUA (((X1 VALUE) <= SMALL_))))))
19
20 (<N> --> (%)
21   ((X1 == (X0 PRED))
22    (X0 = (GET-LEX-FS X1 '((CAT N)) :LEX-ID 'ENG))
23    ((X1 VALUE) <= (X0 ROOT))))
24
25 (<V> --> (B I T E S)
26   (((X0 PRED) = BITE)
27    ((X0 TENSE) = PRES)
28    ((X0 NUM) = SG)
29    ((X0 PERS) = 3)))
30
31 (<P> --> (W I T H)
```

[3]In this paper UKernel is mostly described in the context of *generation*. However whenever appropriate the considerations for *analysis* will also be discussed.

[4]For analysis the execution of a grammar ends at the `<START>` rule.

```
  (((X0 PRED) = WITH)))

(<NP> ==> (<DET> <N>)
  (((X0 FIN) = *DEFINED*)
   ((X1 FIN) == (X0 FIN))
   ((X1 NUM) <= (X0 NUM))
   (X2 = X0)))

(<NP> ==> (<N>)
  (((X0 FIN) = *UNDEFINED*)
   (X1 = X0)))

(<VP> ==> (SUDDENLY <V> <NP>)
  (((X0 INST) = *UNDEFINED*)
   (X3 == (X0 OBJ))
   (X2 = X0)))

(<VP> ==> (WITHOUT_A_BLINK <V> <NP> <PP'>)
  ((X4 == (X0 INST))
   (X3 == (X0 OBJ))
   (X2 = X0)))

(<VP> ==> (% <V> <NP> % <PP>)
  ((X5 == (X0 INST))
   (X3 == (X0 OBJ))
   (X2 = X0)
   ((X1 VALUE) <= ALL_OF_A_SUDDEN)
   (*EOR* (((X1 VALUE) <= *SHOULD_NOT_GENERATE_THIS*)
           ((X3 COAT_COLOR) =t RED)
           ((X4 VALUE) <= IN_RED))
          (((X3 COAT_COLOR) =t WHITE)
           ((X4 VALUE) <= IN_WHITE)))))

(<DET> --> (T H E)
  (((X0 FIN) = +)))

(<DET> --> (A)
  (((X0 FIN) = -)
   ((X0 NUM) = SG)))

(<PP'> ==> (<P'> <NP>)
  (((X1 PRED) <= WITH)
   (X2 = X0)))

(<PP> ==> (<P> <NP>)
  (((X1 PRED) <= WITH)
   (X2 = X0)))

(<P'> --> (B Y)
  (((X0 PRED) = BY)))
```

Listing 1: A toy generation grammar.

With the toy lexicon (with LEX-ID=ENG) shown in Listing 2 and the sample input feature structure given in Listing 3, running a simple recursive-descent generator utilizing UKernel produces the output "`A SMALL_DOG ALL_OF_A_SUDDEN BITES THE GIRL IN_WHITE WITH THE TEETH`"[5]. The execution trace of the generator can be found in Appendix D, which can be very helpful for understanding the inner workings of UKernel.

```
(A ((CAT D) (ROOT A) (FIN -)))
(THE ((CAT D) (ROOT THE) (FIN +)))
(DOG ((CAT N) (ROOT DOG) (NUM SG)))
(GIRL ((CAT N) (ROOT GIRL) (NUM SG)))
(TOOTH ((CAT N) (ROOT TEETH) (NUM PL)))
(BITE ((CAT V) (ROOT BITES) (NUM SG) (PERS 3) (TENSE PRES)))
(WITH ((CAT P) (ROOT WITH)))
```

Listing 2: A toy lexicon.

```
X0: ((SUBJ
      (*OR*
       ((PRED DOG)
        (FIN -)
        (PERS 3)
        (NUM SG)
        (BREED CHIHUAHUA))
       ((PRED DOG)
        (FIN -)
        (PERS 3)
        (NUM SG)
        (BREED PITBULL))))
     (OBJ
      ((PRED GIRL)
       (FIN +)
       (PERS 3)
       (NUM SG)
       (COAT_COLOR WHITE)))
     (INST
      ((PRED TOOTH)
       (FIN +)
       (PERS 3)
       (NUM PL)))
     (PRED BITE)
     (TENSE PRES))
```

Listing 3: An input feature structure.

In the rest of this paper we shall use the following notations for brevity.

[5]The output is produced by *Generator*, a simple recursive-descent generator we implemented based on UKernel. The execution trace in Appendix D was captured from Generator version 1.36, running under the default *fork* mode, in which the generator tries out each *continuous* disjunct in an `*OR*` input feature structure (see Section 6.1), and stops at the first successful generation (note the "Fork Call" in Appendix D). The other execution mode *plain* treats an `*OR*` input feature structure as a whole in running the equations.

By *continuous disjuncts* we mean all of the `*OR*` sub-feature structures whose enclosing feature structures are all `*OR*` feature structures.

- LHS: Left-hand-side.
- RHS: Right-hand-side.
- T: Terminals; e.g., `D`.
- NT: Non-terminals; e.g., `<NP>`.
- FS: Feature Structures (Section 6.1).
- VAR: Variables denoting FS; e.g., `X0`, `X1`, `X2`, etc (Section 5).
- QFS: Quoted Feature Structures; e.g., `'((CAT N))`.
- PATH: A sequence of feature names; e.g., `SUBJ PERS`. Paths can be *penetrating* (Section xx).
- FSPATH: To uniquely identify a feature; could be either VAR or (VAR PATH) pair; e.g., `X0`, `(X0 SUBJ PERS)`.
- V: Values; e.g., `SG`, `(*OR* SG PL)`, `(*NOT* SG)` (Section 6.1).
- TV: Truth Value; e.g., `T`, `NIL`. To distinguish between T (terminals) and `T` (true), we use typewritter typeface for the latter. In all of the contexts it should be self-evident.
- ARG: A function argument; could be a QFS, FSPATH, V, or TV.

Figure 1: Some notations used in this paper.

3 Grammar Rules

A single grammar rule has the form shown in Fig. 2[6]. The difference in notations between a T and an NT is that the latter is surrounded by a pair of angle brackets. The arrow indicates the direction of processing: a left-to-right arrow means generation (thus the LHS is broken down into the RHS) while a right-to-left arrow is for analysis (thus the RHS are combined into the LHS). Following the c-structure rule is an AND equation block[7]: a set of equations/equation blocks surrounded by a pair of parentheses. All equations/equation blocks inside the AND equation block must be executed successfully in the order of their appearance for the rule to succeed. In this section we shall focus on the c-structure rule while leaving the details of equations/equation blocks to Section 5.

```
<grammar_rule> ::= “(” <c_structure_rule> <AND_equation_block> “)”
<c_structure_rule> ::= <rule_LHS> <arrow> “(” <rule_RHS> “)”
<rule_LHS> ::= <NT>
<rule_RHS> ::= <literal> [<rule_RHS>]
<literal>  ::= <T> | <NT> | “%”
<arrow>    ::= --> | ==> | <-- | <==
```

Figure 2: A grammar rule.

Case is significant for all Ts and NTs. In case of Ts the generated string from a T will have the same case; e.g., in the toy grammar and lexicon presented in Section 2 had we changed all Ts and all `ROOT` values into lowercase we would have had the output “`a small_dog all_of_a_sudden bites the girl in_white with the teeth`”[8].

The wildcard symbol % is treated as a T, and can be freely mixed with Ts and NTs (see line 15, 20 and 54 in Listing 1). In generation its content will be initialized via executing the equation block, while in analysis its content is used to initialize a feature in the RHS FS (see Section 5.1).

The meanings of differently oriented arrows are already discussed. For each oriented arrow one could have a single arrow or a double arrow; e.g., for generation they are `-->` and `==>`. The only difference between the two is that for generation the latter inserts a space character between the surface forms generated from each individual RHS literal[9], while the former does not. For example, in Listing 1 when the rule on line 25 is successfully executed, the generated string will be “`BITES`” with no space between the letters; but for the rule on line 34 it generates “`THE GIRL`” with a space between the two words. The rules with single arrows are also known as the *character-level* rules.

[6] In this paper we use Backus-Naur Form (BNF) [2] to describe various grammar constructs. In particular “::=” means “is defined as”, ‘|’ means ‘or’, the angle brackets “<>” are used to surround category names, the square brackets “[]” are used to surround optional parts, and a pair of double quotes are used to surround terminals with single characters. However we do not attempt to bore the readers with a complete BNF description, since some of them can be straightforwardly described in English, e.g., those found in Fig. 1.

[7] The other types of equation blocks are: `*OR*`, `*EOR*`, and `*CASE*` blocks. See Section 5 for details.

[8] Again we note that the front-end which uses UKernel can make the case insignificant if the application requires it.

[9] For parsing a double arrow consumes the space characters between the RHS literals rather than inserting them.

4 Lexicons

4.1 The Basics

A lexicon consists of a list of entries, with each entry representing a word[10]. A lexical entry has the form shown in Fig 3. The definition is essentially an FS, and its only requirement is that you *must* provide a feature `CAT` to indicate the category of the entry. In other words, there are two mandatory pieces of information for each entry: the name (a string) and the feature `CAT`. An example lexicon can be found in Listing 2.

```
<lex_entry> ::= "(" <name> "(" <definition> ")" ")"
```

Figure 3: A lexical entry.

The category of a lexical entry can be simply its grammatical function (nouns, verbs, prepositions, etc.), or its semantic category, or anything one can think of to reasonably partition the lexicon. The rest of the definition usually, but not necessarily, has a `ROOT` feature to indicate its word root, and some other features for further morphological processing (e.g., number, person, gender, tense, etc.).

Since it is possible to load in multiple lexicons, one must specify a *lexicon ID* (`LEX-ID`) for each of them in order for UKernel to uniquely identify each one of them (even when there is only one lexicon loaded; see Section 8.2 for the discussions of loading multiple lexicons). Candidates for a lexicon ID include the language of the lexicon (`ENGLISH`, `GERMAN`, etc.), the domain of the lexicon (`MEDICAL`, `TRAVEL`, etc.), or anything appropriate. When lexical lookup is engaged it is then possible to give a particular lexicon ID to search, or search all of the lexicons when no lexicon ID is specified. The example lexicon in Listing 2 has a lexicon ID `ENG`.

Internally the lexical entries are indexed using the entry name as the primary key, and the entry category as the secondary key - that is why they are mandatory. It is not unusual to have entries that have the same names and categories. In a more general scenario it is possible when doing lexical lookup multiple entries match the criteria, in which case one can choose between obtaining only the first match, or receiving a selected set of the matches merged into a complex *OR* FS (see Section 6.1 for the descriptions of complex FSs).

4.2 Lexical Lookup Functions: `GET-LEX` and `GET-LEX-FS`

We now introduce the lexical lookup functions: `GET-LEX` and `GET-LEX-FS` - they are used on the RHS of an equation (see Section 5 for the details of equations). Fig. 4 gives their forms.

```
<lex_lookup> ::= "(" <function> <ARG> <ARG>
                     [:LEX-ID <ARG>] [:CHECK <ARG>] [:AMBIGUITY <ARG>] ")"
<function>   ::= GET-LEX | GET-LEX-FS
```

Figure 4: Lexical lookup functions: `GET-LEX` and `GET-LEX-FS`.

Both forms return the matching lexical entries (if any) as an FS. The possible FS returned depends on whether a lookup is successful, or depends on the cause of a failure. This is detailed as follows:

- If a lookup is successful:
  - If there is only one matching entry, or only a single entry is requested (by ignoring `:AMBIGUITY` keyword argument or set it to `NIL`, see below), the returning FS contains the matching definition FS with added features `SEM-VALUE` and `LEX-ID`.
  - If there are multiple matching entries and the user requests them all (by setting `:AMBIGUITY` keyword argument to `T`), the returning FS is a complex `*OR*` FS merging the definition FSs from all matching entries, with added features `SEM-VALUE` and `LEX-ID` *per* definition FS.
- If a lookup fails:

[10] Or a phrase, or whatever surface units one can think of.

– If the reason is that no such entry exists in the lexicons, an FS with *only* one feature `SEM-VALUE` is returned.
– If it is because of the errors in processing one of the arguments (more on the arguments below in this section), an *empty* FS is returned.

In all of the cases above whenever `SEM-VALUE` is set its value is set to the 1st ARG of the function call, and whenever `LEX-ID` is set its value is set to the ID of the lexicon in which the entry is found.

It is important to note when a lexical lookup fails, it is still possible for the host equation to succeed, depending on the operator and the LHS of the equation.

The difference between the two lookup functions is that `GET-LEX` looks for entries which have their names matched with the 1st ARG and their categories matched with the 2nd ARG, while `GET-LEX-FS` finds entries which have their names matched with the 1st ARG, but their definitions matched with the 2nd ARG (using implicitly operator `=t'`, with the 2nd ARG as the RHS, see Section 7). For example, in Listing 1 on line 22, the lexicon with ID "`ENG`" is searched to find an entry with the same name as that stored in the VAR `X1`, and with a unifiable definition with the QFS `'((CAT N))`. The same lexical lookup can be realized as shown in Fig. 5 using `GET-LEX`[11]. When `X1` is `GIRL`, both forms unifies `X0` with the augmented FS: `((SEM-VALUE GIRL) (CAT N) (ROOT GIRL) (NUM SG) (LEX-ID ENG))`.

```
(X0 = (GET-LEX X1 'N :LEX-ID 'ENG))
```

Figure 5: Rewriting line 22 in Listing 1 using `GET-LEX`.

More precise restrictions on the 1st ARG and the 2nd ARG of `GET-LEX` and `GET-LEX-FS` are given below. Note if any of the restrictions is violated a lookup will return an *empty* FS (but the host equation might still succeed).

- For both: The 1st ARG must be either an FSPATH evaluated to an atomic V, or a quoted atomic V itself (see Section 6.1 for the descriptions of atomic/complex Vs).
- For `GET-LEX`: The 2nd ARG must be either an FSPATH evaluated to an atomic V, or a quoted atomic V itself.
- For `GET-LEX-FS`: The 2nd ARG must be an FSPATH or a QFS, and in the former case the PATH must exist.

The rest of the arguments of both `GET-LEX` and `GET-LEX-FS` are all *optional*, and their relative ordering is insignificant[12]. They share the same restrictions and semantics described as follows. Again if any of the restrictions is violated the entire host equation fails.

- `:LEX-ID` ARG: It must be either an FSPATH evaluated to an atomic V, or a quoted atomic V itself. If it is present the functions only search in the lexicon with the corresponding ID. Otherwise all lexicons will be searched.
- `:CHECK` ARG: It must be either an FSPATH or a QFS, and in the former case the PATH must exist. The ARG forms a *filter FS*, which is used to filter out the disqualified matching entries. In the case where an FSPATH is given, the filter FS is obtained by evaluating the ARG. For every feature in a given filter FS, if a matching entry has in its definition FS the same feature but with a non-unifiable value, than the entry is filtered (removed). Note it is okay for a definition FS and a filter FS to have different features - filtering only depends on the common features between the two. Finally if no entry is left after filtering, an FS with only one feature `SEM-VALUE`, which gets its value from the 1st ARG (as usual), is returned.
- `:AMBIGUITY` ARG: It must be a TV. If it is present and the value is `T` (the opposite is `NIL`), all matching entries will be returned via a complex `*OR*` FS, otherwise only the 1st matching entry is returned (based on the ordering of the entries in the lexicon file). If it is used with a `:CHECK` ARG, the filtering will take place *after* all matching entries are found. If there is only one entry left, the returning FS will be an atom instead of a complex `*OR*` FS.

[11]This implies there's a speed difference between the two lookup functions. See Section 9 for general suggestions on writing an efficient grammar.

[12]LISP programmers will recognize these as keyword arguments, thus by definition they are optional and the ordering is insignificant.

Several examples using various arguments to search in the toy lexicon given in Listing 2 are shown below (see Section 5.3 and 7 for the descriptions of equations and assignment operator <=).

```
(X0 <= (GET-LEX 'DOG 'N :LEX-ID 'ENG))
 result: X0: ((SEM-VALUE DOG) (CAT N) (ROOT DOG) (NUM SG) (LEX-ID ENG))
(X0 <= (GET-LEX 'DOG 'N))
 result: X0: ((SEM-VALUE DOG) (CAT N) (ROOT DOG) (NUM SG) (LEX-ID ENG))
(X0 <= (GET-LEX 'CAT 'N))
 result: X0: ((SEM-VALUE CAT))
(X0 <= (GET-LEX 'DOG 'N :AMBIGUITY T))
 result: X0: (*OR*
              ((SEM-VALUE DOG) (CAT N) (ROOT DOG) (NUM SG) (LEX-ID ENG))
              ((SEM-VALUE DOG) (CAT N) (ROOT DOGS) (NUM PL) (LEX-ID ENG)))
(X0 <= (GET-LEX 'DOG 'N :AMBIGUITY T :CHECK '((NUM PL))))
 result: X0: ((SEM-VALUE DOG) (CAT N) (ROOT DOG) (NUM PL) (LEX-ID ENG))
```

Listing 4: Examples of lexical lookups with various arguments; note `X0` is assumed to be empty before the execution of each equation

5 Equations and Equation Blocks

This section describes the second component of a grammar rule: an equation block and the equations within.

5.1 Variables: the Interface Between Rules and Equations

Before diving into the details of equations/equation blocks we shall introduce the use of VARs (variables) as a bridge between a c-structure rule and its accompanied equation block. A VAR has the form given in Fig. 6, and is used to denote the FS of a literal in the c-structure rule: `X0` is the FS for the LHS of the c-structure rule, and X1 ... Xn denotes the FS of the 1st ... n-th literal of the RHS. By manipulating these variables we are then able to generate the RHS literals of a c-structure rule for generation, or obtain the LHS NT for analysis. Use of an out-of-range VAR is *not* allowed, as it may cause unexpected system behavior (for system developers see Appendix B.1.4 for VAR addressing, and Appendix B.1.7 for automatic resizing of `FSRegisters`). Plenty of examples of manipulating VARs can be found in Listing 1. In particular, the equation on line 5 assigns the value of feature `SUBJ` of the FS of `<S>` (`X0`) to the FS of `<NP>` (`X1`), and removes the feature `SUBJ` from the FS of `<S>` after the assignment (see Section 7 for the descriptions of various operators).

```
<VAR> ::= X<digits>
```

Figure 6: A VAR (variable).

At this point it should be noted that for generation, any change to `X0` in a rule is only visible within the rule, while for analysis any change to the RHS VARs (`X1` and beyond) is only visible to the host rule. For example, line 5 in Listing 1 changes `X0` by removing its `SUBJ` feature; however had we had another rule with `<S>` as the LHS, when executing the rule after the rule on line 4 (which of course implies that the latter fails), `X0` would have had feature `SUBJ` still.

The wildcard "`%`" in the RHS of an equation, like the other RHS literals, has its own corresponding VAR (hence its own FS). The twist here is that the feature `VALUE` of the VAR is defined to correspond to the surface content of the wildcard. For example, after executing line 58 in Listing 1, the leftmost wildcard of the rule corresponds to the surface token "`ALL_OF_A_SUDDEN`"[13].

5.2 Equation Blocks

Equations and equation blocks play a central role in UKernel. As mentioned in Section 3 each grammar rule has its equation block, an AND equation block, which consists of a list of equations/equation blocks. The form of an AND equation block is shown in Fig. 7. For a grammar rule to succeed, each equation and equation block in the AND block must be executed successfully, in the order of their appearance.[14]

```
<AND_equation_block> ::= “(” <equation_list> “)”
<equation_list> ::= (<equation> | <equation_block>) [<equation_list>]
```

Figure 7: An AND equation block.

The three other equation blocks are shown in Fig. 8. All four of them can be arbitrarily embedded inside one another, with only restriction that the top-level block must be an AND block. The semantics is described below.

- `*OR*` equation block: *All* equations and equation blocks inside an `*OR*` block are executed in the order of their appearance, but *independently from each other*; i.e., for the same VAR used by two equations/equation blocks, the change to it in one is not reflected in the other[15]. The execution is successful *iff* (if and only if) at

[13] In analysis it is the other way around: first a wildcard gets its matched surface content, then the feature `VALUE` of the wildcard's VAR should be initialized with the surface content implicitly.

[14] In the case of failure the equations after the point of the failure will not be executed. This is mainly to save time - since if a rule fails every FS should maintain its original content.

[15] Internally it is done by executing on a local copy of the relevant VARs, and later multiple local copies of same VARs are merged into a complex `*OR*` FS.

```
<equation_block> ::= <AND_equation_block> | <OR_equation_block>
                     <EOR_equation_block> | <CASE_equation_block>
<OR_equation_block> ::= "(" *OR* <equation_list> ")"
<EOR_equation_block> ::= "(" *EOR* <equation_list> ")"
<CASE_equation_block> ::= "(" *CASE* <FSPATH> <val_blocks> ")"
<val_blocks> ::= "(" <V> <equation_block> ")" [<val_blocks>]
```

Figure 8: Three other different equation blocks.

least one equation or equation block is executed successfully, and for each VAR the final content is a complex `*OR*` FS merging all of the same VARs appearing in the successfully executed equation/equation blocks (an example is given later in this section). If such merged FS has only one daughter FS, it will be reduced to an atomic FS.

- `*EOR*` equation block: This is similar to an `*OR*` block except that the execution stops at the point where the first successful equation/equation block is found (so the rest of the block is not executed), and for each VAR the final content is an atomic FS instead of a complex `*OR*` FS (since no merging is necessary)[16]. An example `*EOR*` block can be found starting from line 59 in Listing 1.
- `*CASE*` equation block[17]: A `*CASE*` block has a leading FSPATH which must be evaluated to an atomic V, and a list of value blocks. Each value block consists of a value and an equation block. When executing a `*CASE*` block, first a non-filtering test (using the operator `=t'`, see Section 7) between the atomic V from the FSPATH and the value in each value block is done, in the order of their appearance. Upon seeing the first value block whose value is successfully tested, its equation block is executed, and its result is returned as the result of executing the entire `*CASE*` block. An example `*CASE*` block can be found starting from line 16 in Listing 1: the feature `VALUE` of `X1` is initialized based on the feature `BREED` of `X0`: if it is a pitbull then we have `BIG_`, or if it is a chihuahua we get `SMALL_`.

An example of `*OR*` equation block is shown in Listing 5, which can be successfully executed on `X0` given in Listing 6, and the resulting VARs are shown in Listing 7. However when executing the `*OR*` equation block in Listing 8 on the same `X0`, the equation block fails due to type mismatch (values and FSs cannot be operated on in the same equation). The reader might need to read Section 7 to understand the examples.

```
(*OR* (((X0 SUBJ NUM) =c SG)
       ((X1 NUM) = SG)
       ((X2 NUM) = (X1 NUM)))
      (((X0 SUBJ NUM) =c PL)
       ((X1 NUM) = PL)
       ((X2 NUM) = (X1 NUM))))
```

Listing 5: An `*OR*` equation block.

```
((SUBJ
  ((PRED YOU)
   (PERS 2)
   (NUM (*OR* SG PL))))
 (PRED RUN))
```

Listing 6: The VAR `X0`.

[16] This is similar to the concept of a *short-circuited or* found in most contemporary programming languages.

[17] The basic idea is similar to a `switch` statement in C/C++. Note one can always rewrite a `*CASE*` block using an `*EOR*` block, but not vice versa. There is also a performance difference between the two. See Section 9 for details.

```
X0: ((SUBJ
      ((PRED YOU)
       (PERS 2)
       (NUM (*OR* SG PL))))
     (PRED RUN))
X1: (*OR*
     ((NUM SG))
     ((NUM PL)))
X2: (*OR*
     ((NUM SG))
     ((NUM PL)))
```

Listing 7: The resulting VARs of executing the `*OR*` block in Listing 5 on `X0` in Listing 6.

```
(*OR* ((((X0 SUBJ NUM) =c SG)
       ((X1 NUM) = SG)
       (X2 = FOO))
      (((X0 SUBJ NUM) =c PL)
       ((X1 NUM) = PL)
       ((X2 FOO) = BAR)))
```

Listing 8: Another `*OR*` equation block.

5.3 Equations

An equation takes the form shown in Fig. 9. Depending on the different operator used (there are eleven of them: =, =c, =c', =i, =i', =t, =t', <=, ==, > and <), different restrictions on the RHS of an equation apply. The general semantics of an equation is to apply the operator on both the LHS and the RHS of an equation. We shall leave the details of the different operators and the implied restrictions on the RHS to Section 6.

```
<equation> ::= <equation_pseudo_unification> |
               <equation_constraint> | <equation_constraint_nofilter> |
               <equation_test> | <equation_test_nofilter> |
               <equation_isomorphism> | <equation_isomorphism_nofilter> |
               <equation_assignment> | <equation_removal_assignment> |
               <equation_push> | <equation_pop>
<equation_pseudo_unification> ::= “(” <equ_LHS> “=” <equ_RHS_pseudo_unification> “)”
<equation_constraint> ::= “(” <equ_LHS> =c <equ_RHS_constraint> “)”
<equation_constraint_nofilter> ::= “(” <equ_LHS> =c’ <equ_RHS_constraint> “)”
<equation_test> ::= “(” <equ_LHS> =t <equ_RHS_test> “)”
<equation_test_nofilter> ::= “(” <equ_LHS> =t’ <equ_RHS_test> “)”
<equation_isomorphism> ::= “(” <equ_LHS> =i <equ_RHS_isomorphism> “)”
<equation_isomorphism_nofilter> ::= “(” <equ_LHS> =i’ <equ_RHS_isomorphism> “)”
<equation_assignment> ::= “(” <equ_LHS> <= <equ_RHS_assignment> “)”
<equation_removal_assignment> ::= “(” <equ_LHS> == <equ_RHS_removal_assignment> “)”
<equation_push> ::= “(” <equ_LHS> “>” <equ_RHS_push> “)”
<equation_pop> ::= “(” <equ_LHS> “<” <equ_RHS_pop> “)”
<equ_LHS>  ::= <FSPATH>
```

Figure 9: An equation. For the forms of all RHS see Section 7.

6 Feature Structures, Values and Paths

This section dives into an individual equation to describe feature structures, values and paths.

6.1 Feature Structures and Values

Feature Structures (FSs) and values (V) are the "information carriers" between the constituents in a unification-based grammar. An FS can either be *atomic* or *complex* (Fig. 10). An atomic FS consists of a list of features. Each feature can either have a V or an FS[18]. A complex FS, on the other hand, has a type keyword (`*OR*` or `*MULTIPLE*`), and a list of at least two FSs. For a `*MULTIPLE*` FS, the semantics is that the conjunction of all embedded FSs represents the entire FS, while for an `*OR*` FS only one of the embedded FSs can represent the entire FS (ambiguity packing). The ordering of the embedded FSs is significant in a `*MULTIPLE*` FS, but it is not in an `*OR*` FS. Listing 3 gives an example of an atomic FS, and Listing 7 shows two `*OR*` FSs (`X1` and `X2`).

```
<FS> ::= <atomic_FS> | <complex_FS>
<atomic_FS> ::= "(" <feature> [<feature>] ")"
<feature> ::= "(" <feature_name> (<V> | <FS>) ")"
<complex_FS> ::= "(" <complex_FS_type> <FS> <one_or_more_FSs> ")"
<complex_FS_type> ::= *OR* | *MULTIPLE*
<one_or_more_FSs> ::= <FS> [<one_or_more_FSs>]
```

Figure 10: Feature Structures. Note a complex FS must have at least two embedded FSs (which can be a complex FS as well).

A V can be an atomic, a complex, or a *constraint* V (Fig. 11). For the first two, an atomic V is simply a string token, while a complex V consists of a type keyword (`*OR*`, `*NOT*` or `*MULTIPLE*`) and a list of at least two non-constraint Vs. For a `*MULTIPLE*` V, the semantics is that the conjunction of all embedded Vs represents the entire V, while for an `*OR*` V only one of the embedded Vs can represent the entire V (ambiguity packing). For a `*NOT*` V it denotes the negation of the disjunction of all of the embedded Vs. The ordering of the embedded Vs is significant in a `*MULTIPLE*` V, but it is not in an `*OR*` or `*NOT*` V. An example `*OR*` V can be found in Listing 6.

```
<V> ::= <atomic_V> | <complex_V> | <constraint_V>
<atomic_V> ::= <token>
<complex_V> ::= "(" <complex_V_type> (<atomic_V> | <complex_V>) <one_or_more_Vs> ")"
<complex_V_type> ::= *NOT* | *OR* | *MULTIPLE*
<one_or_more_Vs> ::= (<atomic_V> | <complex_V>) [<one_or_more_Vs>]
```

Figure 11: Values. A token is just a string.

Constraint Vs take the form given in Fig. 12. As will be covered in Section 7 they can only be used with the pseudo-unification operator `=`, and are all used to test the property of the LHS of an equation, except for `*REMOVE*`, which removes a feature from the LHS. An example of a constraint V can be found on line 35 in Listing 1.

```
<constraint_V> ::= *DEFINED* | *UNDEFINED* | *REMOVE* |
                   *NUMBER* | *INTEGER* | *POSITIVE* |
                   *NOT-NUMBER* | *NOT-INTEGER* | *NOT-POSITIVE*
```

Figure 12: Constraint Vs.

The interactions between various atomic/complex FSs and between different atomic/complex Vs can be fairly tricky. Section 7.2 and 7.3 are devoted to these topics.

[18]Thus it is possible to have complex FSs embedded inside an atomic FS.

6.2 Paths

As briefly described in Fig. 1, a PATH is just a sequence of feature names. When combined with a VAR to become an FSPATH, it uniquely identifies a component FS or V inside the FS denoted by VAR. For example, if the FS shown in Listing 3 is denoted by `X0`, `(X0 SUBJ)` is evaluated to the atomic FS underneath feature `SUBJ`, and `(X0 SUBJ FIN)` is evaluated to value `-`.

For complex FSs, it is possible to have a PATH "penetrating" into them. For example, `(X1 NUM)` is a penetrating FSPATH in Listing 7, since by walking the PATH the embedded FSs underneath the type keyword (`*OR*`) are reached. A more complex example is shown in Listing 9 below, where `(X0 A B C)` is a penetrating FSPATH.

```
X0: ((A
      (*OR*
       ((B
         (*MULTIPLE*
          ((C E))
          ((C E))
          ((C Y))))
        (C Y))
       ((B
         (*OR*
          ((C E))
          ((D Y))
          ((C Y)))))
       ((B
         ((C E)
          (D Y))))
       ((C Y)))))
```

Listing 9: A complex FS for which `(X0 A B C)` is a penetrating FSPATH.

The semantics of a penetrating FSPATH is described below. In an equation if the LHS VAR is a complex FS, given a penetrating FSPATH for the LHS, the specified operator is applied on each visited daughter FS directly under the type keyword `*OR*`/`*MULTIPLE*`, together with the RHS of the equation. In the case where the type keyword is `*OR*`, the execution of the equation is successful *iff* at least one of such operator applications is successful, and for `*MULTIPLE*` the execution is considered successful *iff* all of such operator applications succeed. With a destructive operator (`=`, `=c`, `=t`, `=i`, `<=`, `==`, `>` and `<`; see Section 7 for the descriptions of these operators) the execution also removes each failed daughter FS of the type keyword `*OR*`/`*MULTIPLE*` to form the resulting FS of the LHS VAR. Note that this description is *recursive*, in that we may have an `*OR*` FS embedded inside a `*MULTIPLE*` FS, or vice versa, for arbitrarily many times and in arbitrary places. Listing 10 below gives the resulting `X0` after executing the equation `(X0 A B C) = Y` on `X0`, which is given in Listing 9. Note in the example the first daughter FS of `*MULTIPLE*` fails, which causes the removal of the FS `(B (*MULTIPLE* ((C E)) ((C Y))))`. For the same reason the third daughter FS `(B ((C E) (D Y))))` is also removed. The first daughter FS of the `*OR*` FS in the middle is removed because of the unification failure, but since the unifications of the other two siblings are successful, the (modified) FS `((B (*OR* ((D Y) (C Y)) ((C Y)))))` stays. The last daughter FS `((C Y))` is successfully unified, and the (modified) FS `((C Y) (B ((C Y)))))))` also stays.

```
X0: ((A
      (*OR*
       ((B
         (*OR*
          ((D Y)
           (C Y))
          ((C Y)))))
       ((C Y)
```

```
    (B
     ((C Y)))))))
```

Listing 10: The resulting `X0` of executing `(X0 A B C) = Y` on `X0` given in Listing 9.

During the execution of a destructive operator with a penetrating FSPATH on the LHS, the *isomorphic* daughter FSs/Vs are removed to leave only one (see Section 7 for the descriptions of isomorphic operator `=i`). This is illustrated in Listing 11, where the equation `(X0 A B) <= Y` is executed over the FS given in Listing 9. The operation would have resulted in two copies of FS `((B Y) (C Y))`, and two copies of FS `((B Y))`, but the removal of isomorphic FSs leaves only one copy for each of them.

```
X0: ((A
      (*OR*
       ((B Y)
        (C Y))
       ((B Y)))))
```

Listing 11: The resulting `X0` of executing `(X0 A B) <= Y` on `X0` given in Listing 9.

A penetrating FSPATH can also be used on the RHS of an equation. For such RHSs a *collection* process is engaged to assemble a new FS/V by collecting all targeted features. The new FS/V is then used in the execution of the host equation. Similar to an LHS with a penetrating FSPATH, the collection process succeeds only if it is successful on at least one daughter FS under the type keyword `*OR*`, or it is successful on all daughter FSs under the type keyword `*MULTIPLE*`. The assembled FS could be degenerated into a V, if all targeted features contain a V instead of an FS. If a type mismatch occurs, i.e., some of the targeted features contain Vs and the others contain FSs, the collection process fails. Note the failure of the collection process does not necessarily imply the failure of the host equation - it depends on the particular operator used[19] (see Section 7 for details). Listing 12 shows several examples of assembled new FS/V from the FS `X0` given in Listing 9.

```
RHS FSPATH (X0 A B C) is collected as a V:
(*OR* (*MULTIPLE* E E Y) (*OR* E Y) E)

RHS FSPATH (X0 A B) is collected as an FS:
X0: (*OR*
     (*MULTIPLE*
      ((C E))
      ((C E))
      ((C Y)))
     (*OR*
      ((C E))
      ((D Y))
      ((C Y)))
     ((C E)
      (D Y)))

RHS FSPATH (X0 A) is collected as an FS:
X0: (*OR*
     ((B
       (*MULTIPLE*
        ((C E))
        ((C E))
        ((C Y))))
      (C Y))
```

[19]For example, the pseudo-unification operator = returns `T` even when the collection process fails.

```
((B
  (*OR*
   ((C E))
   ((D Y))
   ((C Y)))))
((B
  ((C E)
   (D Y))))
((C Y)))
```

Listing 12: Assembled new FS/V from `X0` given in Listing 9

7 Operators

There are eleven different operators available in UKernel[20]. Section 7.1 first describes their syntax and semantics, Section 7.2 and 7.3 then discuss the interactions between atomic/complex FSs and Vs. and this section is devoted to their syntax and semantics.

7.1 Syntax and Semantics

First some notes applying to almost all operators:

1. Each operator imposes restrictions on the possible RHS, which will be described below (see Fig. 9 for the forms of equations with various operators, and Section 4.2 for the lexical lookup functions).
2. For all operators except for assignment `<=`, when applying them if there is a type mismatch between the targets led by LHS and the RHS (one is a V and the other is an FS), the operations fail and return `NIL`.
3. For all operators it is possible to use a penetrating FSPATH on both LHS and RHS - in that case the operator is applied on all targeted features (see Section 6.2 for details on penetrating paths).
4. In applying each operator UKernel automatically removes any *isomorphic* component FS/V (see the descriptions of isomorphic operator `=i` in this section), and reduces a complex FS/V to atomic FS/V if possible in order to get a compact FS/V[21].

Table 1 gives a summary of the semantics of the operators. A note about the column headed with "Destructive?": a *left-destructive* operator (labeled as 'Left' in the table) has power to modify its LHS, while a *destructive* operator (labeled with 'Both') has power to modify both LHS and RHS. However, whether the modification actually occurs depends on the run-time context. On the other hand, non-destructive operators (labeled with 'No') will *never* alter either of the invovled FSs.

We now describe the syntax and semantics of each operator.

- **Pseudo-unification** operator (`=`, left-destructive): The possible forms of the RHS of a pseudo-unification equation is given in Fig. 13. Note this is the only operator with which you can use a constraint V on the RHS. The semantics of the operator is described below. An example pseudo-unification equation can be found on line 6 in Listing 1.

  1. If the RHS is a constraint value:
     (a) If the RHS is `*DEFINED*/*UNDEFINED*`, return `T` *iff* the LHS is/is not defined.
     (b) If the RHS is `*NUMBER*/*INTEGER*/*POSITIVE*`, return `T` *iff* the LHS is a real number/an integer/a positive number.
     (c) If the RHS is `*NOT-NUMBER*/*NOT-INTEGER*/*NOT-POSITIVE*`, return `T` *iff* the LHS is *not* a real number/an integer/a positive number.
     (d) If the RHS is `*REMOVE*`, undefine the LHS if it is defined and *always* return `T`.
  2. If the LHS is defined, pseudo-unify the RHS (if any) with it. Return `T` *iff* the unification succeeds.
  3. If the LHS is not defined, create it with the RHS (if any) and return `T`.
  4. If the RHS is not defined (or in the case of a penetrating FSPATH, when the collection process fails), do nothing and return `T`.
  5. When LHS is a penetrating path into an `*OR*` FS, all of the branches that do not pass the pseudo-unification are pruned. If no branch is left return `NIL`, otherwise return `T`.
  6. Whenever `NIL` is returned nothing is modified.
  7. This operator is symmetrical when the RHS is a FSPATH, *iff* no `*MULTIPLE*` FS/V is involved, because unifying two `*MULTIPLE*` FSs/Vs is pushing the RHS to the front of the LHS, which is an ordering dependent operation.

Op	**Name**	**Destructive?**	**Return value when RHS is undefined**	**Semantics**
`=`	Pseudo-unification	Left	`T`	Pseudo-unify LHS and RHS, or constraining LHS with the constraint value specified on RHS
`=c`	Constraint	Left	`T`	Test isomorphism between LHS and RHS, up to the unifiability of values; remove failed branches in an `*OR*` LHS; `=c` implies `=t` (not vice versa)
`=c'`	Constraint, no filtering	No	`T`	Non-filtering version of `=c`
`=t`	Test	Left	`T`	Test unifiability between LHS and RHS; remove failed branches in an `*OR*` LHS; `=t` implies `=` (not vice versa)
`=t'`	Test, no filtering	No	`T`	Non-filtering version of `=t`
`=i`	Isomorphism	Left	`T`	Test isomorphism between LHS and RHS; remove failed branches in an `*OR*` LHS; `=i` implies `=c` (not vice versa)
`=i'`	Isomorphism, no filtering	No	`T`	Non-filtering version of `=i`
`<=`	Assignment	Left	`T`	Assign RHS to LHS
`==`	Removal assignment	Both	`NIL`	Assign RHS to LHS and remove RHS
`>`	Push	Left	`T`	Push RHS into LHS
`<`	Pop	Both	`NIL`	Pop RHS into LHS

Table 1: Summary of the semantics of all operators

```
<equ_RHS_pseudo_unification> ::= <V> | <FSPATH> | <QFS> | <lex_lookup>
```

Figure 13: Possible RHS of a pseudo-unification equation.

- **Constraint** operator (`=c`, left-destructive): The possible forms of the RHS of a constraint equation is given in Fig. 14. The detailed semantics of the operator is described below.

  1. If the LHS is defined, test the isomorphism between LHS and RHS, up to the unifiability of values.
     (a) See the description of the isomorphism operator `=i` for the definition of isomorphism between two FSs (below). Note this operator is less strict than `=i` since for values we test their *unifiability* (using `=t`), not isomorphism - so essentially `=c` is a `=i` at the FS level and `=t` (non-destructive *with respect to values*) at the V level.
     (b) For a *penetrating* LHS on an `*OR*` FS, the operation amounts to the following: reach a component on the LHS FS according to the specified path and test the isomorphism between the RHS and *that* component. The operation succeeds *iff* at least one such test succeeds; i.e., the thing gets tested must be the thing defined. For example, if `X0` is the FS `(*OR* (A B) (C D))` , the equation `(X0 A) =c E` would fail, although both `(X0 A) = *defined*` and `(X0 A) = E` succeed.
     (c) In the case above, all branches that do not pass the test will be pruned (filtered). Namely, the operation succeeds *iff* there exists at least one branch after the operation.
  2. If the LHS is not defined, return `NIL`.
  3. If the RHS is not defined (or in the case of a penetrating FSPATH, when the collection process fails), do nothing and return `T`.

[20] The reader can experiment with each different operator *interactively* by running UShell. See Appendix C for details.
[21] `*NOT*` values are not reducible.

4. Whenever `NIL` is returned nothing is modified.
5. This operator is symmetrical when the RHS is an FSPATH and no filtering occurs.

```
<equ_RHS_constraint> ::= <atomic_V> | <complex_V> | <FSPATH> | <QFS> | <lex_lookup>
```

Figure 14: Possible RHS of a constraint equation.

- **Non-filtering constraint** operator (`=c'`, non-destructive): This is essentially the constraint operator `=c`, with only one difference: for a penetrating LHS on an `*OR*` FS, no filtering of the failed branches will take place - hence it's a non-destructive and symmetrical operator.

- **Test** operator (`=t`, left-destructive): The possible forms of the RHS of a testing equation is given in Fig. 15. An example testing equation can be found on line 60 in Listing 1. The detailed semantics of the operator is described below.

  1. If the LHS is defined, test the unifiability between LHS and RHS (but not really unifying them).
     (a) This is less strict than `=c` (see above), since isomorphism implies unifiability, but not vice versa. However it is stricter than `=`, since `=t` requires the LHS to be defined.
     (b) Similar to `=c`, for a *penetrating* LHS on an `*OR*` FS, the operation amounts to the following: reach a component on the LHS FS according to the specified path and test the unifiability between the RHS and *that* component. The operation succeeds *iff* at least one such test succeeds; i.e., the thing gets tested must be the thing defined.
     (c) Similar to `=c`, in the case above, all branches that do not pass the test will be pruned (filtered). Namely, the operation succeeds *iff* there exists at least one branch after the operation.
  2. If the LHS is not defined, return `NIL`.
  3. If the RHS is not defined (or in the case of a penetrating FSPATH, when the collection process fails), do nothing and return `T`.
  4. Whenever `NIL` is returned nothing is modified.
  5. This operator is symmetrical when the RHS is an FSPATH and no filtering occurs.

```
<equ_RHS_test> ::= <atomic_V> | <complex_V> | <FSPATH> | <QFS> | <lex_lookup>
```

Figure 15: Possible RHS of a testing equation.

- **Non-filtering test** operator (`=t'`, non-destructive): This is essentially the test operator `=t`, with only one difference: for a penetrating LHS on an `*OR*` FS, no filtering of the failed branches will take place - hence it's a non-destructive and symmetrical operator.

- **Isomorphism** operator (`=i`, left-destructive): The possible forms of the RHS of an isomorphism equation is given in Fig. 16. Before introducing the semantics we shall define the isomorphism between two FSs/Vs. Two FSs/Vs are *isomorphic iff* they have exactly identical structures. Recall from Section 6.1 that for `*MULTIPLE*` FSs/Vs the ordering of daughter FSs/Vs is significant, but for the other types of FSs/Vs the ordering of daughters is insignificant. Several examples of isomorphic/non-isomorphic FSs/Vs are shown in Listing 13. At all times UKernel attempts to remove isomorphic component FSs/Vs inside any FS/V.

  The semantics of the isomorphism operator is described below.

  1. If the LHS is defined, test the isomorphism between LHS and RHS (but not really unifying them).

(a) This is stricter than `=c` (see above); actually this is the strictest among `=`, `=c`, `=t` and `=i`, since `X0 =i X1` implies `X0 =c X1`, `X0 =t X1`, and `X0 = X1`, but not vice versa.
(b) Similar to `=c`, for a *penetrating* LHS on an `*OR*` FS, the operation amounts to the following: reach a component on the LHS FS according to the specified path and test the isomorphism between the RHS and *that* component. The operation succeeds *iff* at least one such test succeeds; i.e., the thing gets tested must be the thing defined.
(c) Similar to `=c`, in the case above, all branches that do not pass the test will be pruned (filtered). Namely, the operation succeeds *iff* there exists at least one branch after the operation.

2. If the LHS is not defined, return `NIL`.
3. If the RHS is not defined (or in the case of a penetrating FSPATH, when the collection process fails), do nothing and return `T`.
4. Whenever `NIL` is returned nothing is modified.
5. This operator is symmetrical when the RHS is an FSPATH and no filtering occurs.

```
<equ_RHS_isomorphism> ::= <atomic_V> | <complex_V> | <FSPATH> | <QFS> | <lex_lookup>
```

Figure 16: Possible RHS of an isomorphism equation.

```
Vs:

0 vs. 0: isomorphic
0 vs. 1: non-isomorphic
(*OR* 0 1 2) vs. (*OR* 2 0 1): isomorphic
(*OR* 0 1) vs. (*OR* 0 1 2): non-isomorphic
(*NOT* 0 1 2) vs. (*NOT* 2 0 1): isomorphic
(*NOT* 0 1) vs. (*NOT* 0 1 2): non-isomorphic
(*MULTIPLE* 0 1 2) vs. (*MULTIPLE* 0 2 1): non-isomorphic
(*OR* 0 (*MULTIPLE* 0 1) 2) vs. (*OR* (*MULTIPLE* 0 1) 0 2): isomorphic
(*OR* 0 1 2) vs. (*NOT* 3): non-isomorphic (but unifiable)

FSs:

((A B)) vs. ((A C)): non-isomorphic
((A B)) vs. ((A B)): isomorphic
(*OR* ((A B)) ((A C))) vs. (*OR* ((A C)) ((A B))): isomorphic
(*OR* ((A B)) ((A C))) vs. (*OR* ((A B)) ((A C)) ((A D))): non-isomorphic
(*MULTIPLE* ((A B)) ((A C))) vs. (*MULTIPLE* ((A C)) ((A B))): non-isomorphic
(*OR* ((A B)) (*MULTIPLE* ((A C)) ((A D)))) vs.
(*OR* (*MULTIPLE* ((A C)) ((A D))) ((A B))): isomorphic
(*OR* ((A B)) ((A C))) vs. (*MULTIPLE* ((A (*NOT* D))) ((A (*NOT* E)))):
non-isomorphic (but unifiable)
```

Listing 13: Examples of isomorphic/non-isomorphic FSs/Vs.

- **Non-filtering isomorphism** operator (`=i'`, non-destructive): This is essentially the test operator `=i`, with only one difference: for a penetrating LHS on an `*OR*` FS, no filtering of the failed branches will take place - hence it's a non-destructive and symmetrical operator.
- **Assignment** operator (`<=`, left-destructive): The possible forms of the RHS of an assignment equation is given in Fig. 17. The semantics of the operator is described below. An example assignment equation can be found on line 23 in Listing 1.

1. If the LHS is defined, remove it and assign the RHS (if any) to it.
2. If the LHS is not defined, create it with the RHS (if any).
3. If the RHS is not defined (or in the case of a penetrating FSPATH, when the collection process fails), undefine the LHS if it is defined.
4. *Always* return `T`.
5. This operator is asymmetrical.

```
<equ_RHS_assignment> ::= <atomic_V> | <complex_V> | <FSPATH> | <QFS> | <lex_lookup>
```

Figure 17: Possible RHS of an assignment equation.

- **Removal assignment** operator (==, destructive): The possible forms of the RHS of a removal-assignment equation is given in Fig. 18. The semantics of the operator is described below. An example removal-assignment equation can be found at line 5 in Listing 1.

  1. If the LHS is defined, pseudo-unify the RHS (if any) with it. Undefine the RHS and return `T` *iff* the unification succeeds.
  2. If the LHS is not defined, create it with the RHS (if any). Undefine the RHS and return `T`.
  3. If the RHS is not defined (or in the case of a penetrating FSPATH, when the collection process fails), return `NIL`.
  4. Whenever `NIL` is returned nothing is modified.
  5. This operator is asymmetrical.

```
<equ_RHS_removal_assignment> ::= <FSPATH>
```

Figure 18: Possible RHS of a removal-assignment equation.

- **Push** operator (>, left-destructive): The possible forms of the RHS of a push equation is given in Fig. 19. The semantics of the operator is described below:

  1. If the LHS is defined, Push the RHS (if any) to the front of the LHS.
  2. If the LHS is not defined, create it with the RHS (if any).
  3. If the RHS is not defined (or in the case of a penetrating FSPATH, when the collection process fails), do nothing and return `T`.
  4. If the LHS is an atom FS/V it will be changed to a `*MULTIPLE*` FS/V.
  5. Whenever `NIL` is returned nothing is modified.
  6. This operator is asymmetrical.

```
<equ_RHS_push> ::= <FSPATH> | <QFS> | <lex_lookup>
```

Figure 19: Possible RHS of a push equation.

- **Pop** operator (<, destructive): The possible forms of the RHS of a pop equation is given in Fig. 20. The semantics of the operator is described below:

1. If the LHS is defined, obtain the 1st (leftmost) element from the RHS (if any) and pseudo-unify the LHS with it. Remove that element and return `T`, *iff* the unification succeeds.
2. If the LHS is not defined, obtain the 1st element from the RHS (if any) and create the LHS with it. Remove that element and return `T`.
3. If the RHS is not defined (or in the case of a penetrating FSPATH, when the collection process fails), return `NIL`.
4. When removing the 1st element from the RHS, if the RHS is a `*MULTIPLE*` FS/V simply remove the 1st element from it; if it is not undefine the RHS. After the removal if the RHS becomes a `*MULTIPLE*` FS/V with single element, reduce it to an atomic FS/V.
5. Whenever `NIL` is returned nothing is modified.
6. This operator is asymmetrical.

```
<equ_RHS_pop> ::= <FSPATH>
```

Figure 20: Possible RHS of a pop equation.

7.2 Interactions Between Atomic/Complex Feature Structures

This sub-section describes how *pseudo-unifications* (operator `=`) work between atomic/complex FSs. The reason why among all of the operators only `=` is considered here is because it is the core operation behind many other seemingly different operators.

In the following descriptions the LHS is modified *iff* the unification is successful.

1. Atom vs. atom: This can be best described by the algorithm shown below.

```
For each RHS feature
  if the same LHS feature can be found
    if both are Vs
      return T iff the two values unify with each other
    else if both are FSs
      return T iff the two FSs unify with each other
    else return NIL
  else copy the RHS feature to the LHS and return T
```

Listing 14: Algorithm for pseudo-unifying two atomic FSs.

2. Atom vs. `*OR*`: Same with 4 except it is the other way around.
3. Atom vs. `*MULTIPLE*`: Same with 7 except it is the other way around.
4. `*OR*` vs. atom: Unify each daughter of the LHS with the RHS, and the result is the collection of all successfully unified FSs, without the isomorphic ones. Return `T` *iff* the result is non-empty. If the result is empty (returning `NIL`) the LHS is *not* modified. An example is shown below.

```
X1: (*OR*
      (*MULTIPLE* ((D E)) ((B C)))
      ((A
        (*MULTIPLE* ((D E)) ((B C))))
       (F
        ((G
```

```
          ((D E))))))
      ((D E)))

X2: ((D E))
----------------------------------------
* X1 = X2
result = T
X1: (*OR*
     (*MULTIPLE* ((D E) (B C))
                 ((D E)))
     ((D E)
      (A
       (*MULTIPLE* ((D E)) ((B C))))
      (F
       ((G
         ((D E))))))
     ((D E)))
```

Listing 15: An example of pseudo-unifying an `*OR*` FS with an atomic FS.

5. `*OR*` vs. `*OR*`: Unify each daughter of the LHS with each daughter of the RHS, and the result is the collection of all successfully unified FSs, without the isomorphic ones. Return `T` *iff* the result is non-empty. If the result is empty (returning `NIL`) the LHS is *not* modified. An example is shown below.

```
X1: (*OR*
     (*MULTIPLE* ((D E)) ((B C)))
     ((A
       (*MULTIPLE* ((D E)) ((B C))))
      (F
       ((G
         ((D E))))))
     ((D E)))
X2: (*OR*
     ((A
       (*MULTIPLE* ((D E)) ((B C))))
      (F
       ((G
         ((D E))))))
     ((D E)))
----------------------------------------
* X1 = X2
result = T
X1: (*OR*
      (*MULTIPLE* ((B C)
                   (A (*MULTIPLE*
                          ((D E))
                          ((B C))))
                   (F
                    ((G
                      ((D E))))))
                  ((D E)
                   (A (*MULTIPLE*
                       ((D E))
                       ((B C))))
```

```
                    (F
                     ((G
                       ((D E)))))))
      (*MULTIPLE* ((B C) (D E))
                  ((D E)))
      ((A (*MULTIPLE* ((B C))
                      ((B C)
                       (D E))
                      ((D E)
                       (B C))
                      ((D E))))
       (F
        ((G
          ((D E))))))
      ((A (*MULTIPLE* ((D E)) ((B C))))
       (F
        ((G
          ((D E)))))
       (D E))
      ((D E)))
```

Listing 16: An example of pseudo-unifying two `*OR*` FSs.

6. `*OR*` vs. `*MULTIPLE*`: Same with 3, but substituting the atom with the `*OR*` FS.
7. `*MULTIPLE*` vs. atom: Return `T` *iff* all of the daughters of the LHS is unified with the RHS, from left to right, in which case each unified daughter is pushed to the front of the new LHS (so the ordering of the daughters is reversed). An example is shown below.

```
X1: (*MULTIPLE*
     ((D E))
     ((B C)))
X2: ((D E))
----------------
* X1 = X2
result = T
X1: (*MULTIPLE*
     ((D E)
      (B C))
     ((D E)))
```

Listing 17: An example of pseudo-unifying a `*MULTIPLE*` FS with an atomic FS.

8. `*MULTIPLE*` vs. `*OR*`: Same with 6 except it is the other way around.
9. `*MULTIPLE*` vs. `*MULTIPLE*`: For each LHS daughter, from left to right, unifies it with each of the RHS daughter, from left to right. Return `T` *iff* all of the unifications succeed, in which case each unified daughter is pushed to the front of the new LHS, so the resulting `*MULTIPLE*` is a concatenation of the unified RHS daughters in the reversed order. Note that this operation is asymmetrical. Two examples are shown below.

```
X1: (*MULTIPLE* ((D E)) ((B C)))
X2: ((A
      (*MULTIPLE* ((D E)) ((B C))))
     (F
```

```
    ((G
      ((D E))))))
----------------------------------------
* X1 = (X2 A)
result = T
X1: (*MULTIPLE* ((B C))
                ((B C) (D E))
                ((D E) (B C))
                ((D E)))
----------------------------------------
* (X2 A) = X1
result = T
X2: ((A
      (*MULTIPLE* ((B C))
                  ((B C) (D E))
                  ((D E) (B C))
                  ((D E))))
     (F
      ((G
        ((D E))))))
```

Listing 18: Examples of pseudo-unifying two *MULTIPLE* FSs.

7.3 Interactions Between Atomic/Complex Values

This sub-section describes how *pseudo-unifications* (operator =) work between atomic/complex Vs. The reason why among all of the operators only = is considered here is because it is the core operation behind many other seemingly different operators.

In the following descriptions the LHS is modified *iff* the unification is successful.

1. Atom vs. atom: Return T *iff* the LHS and the RHS are the same string tokens. Nothing is modified.
2. Atom vs. *NOT*: Return T *iff* the LHS is *not* unified with any daughter of the RHS. Nothing is modified.
3. Atom vs. *OR*: Return T *iff* the LHS is unified with at least one daughter of the RHS. Nothing is modified.
4. Atom vs. *MULTIPLE*: Return T *iff* the LHS is unified with all daughters of the RHS, from left to right, in which case each unified daughter is pushed to the front of the new LHS. An example is shown below.

```
1 = (*MULTIPLE* 1 1)
  result = T, result value = (*MULTIPLE* 1 1)
```

Listing 19: An example of pseudo-unifying an atomic V with a *MULTIPLE* V.

5. *NOT* vs. atom: Same with 2 except that (a). it is the other way around; and (b). if the unification is successful the new LHS is the RHS atom.
6. *NOT* vs. *NOT*: *Always* return T in pseudo-unification ("=") The LHS will be a union of both the LHS and the RHS - this is a "shallow union" in that all daughters of both sides are thrown into the new LHS, and the isomorphic ones are pruned (so if the LHS has daughter (*OR* 1 0 2) and the RHS has daughter (*OR* 0 2 1) only one of them is kept since they are isomorphic; but if the RHS has daughter (*OR* 0 (*OR* 2 1)) which is *not* isomorphic to the LHS daughter, both will be kept, even though the RHS is actually a redundant version of the isomorphic FS (*OR* 0 2 1)). Two examples are shown below.

```
(*NOT* 1 2 3) = (*NOT* 1 4 5 0)
  result = T, result value = (*NOT* 1 2 3 4 5 0)
(*NOT* (*OR* 0 1) 2 3) = (*NOT* 1 4 5 0)
  result = T, result value = (*NOT* (*OR* 0 1) 2 3 1 4 5 0)
```

Listing 20: Examples of pseudo-unifying two `*NOT*` Vs.

7. `*NOT*` vs. `*OR*`: For each daughter of the RHS unify it with the LHS, and remove it from the RHS if the unification fails. Return `NIL` *iff* every unification fails. If the unification is successful the new LHS is the filtered RHS.
8. `*NOT*` vs. `*MULTIPLE*`: Same with 4. If the unification is successful, the new LHS is a `*MULTIPLE*` which has daughters in the reversed order with respect to the original RHS. An example is shown below.

```
(*NOT* 1 2 3) = (*MULTIPLE* 4 5 0)
  result = T, result value = (*MULTIPLE* 0 5 4)
```

Listing 21: An example of pseudo-unifying a `*NOT*` V with a `*MULTIPLE*` V.

9. `*OR*` vs. atom: Same with 3 except that (a). it is the other way around; and (b). if the unification is successful, the new LHS is the RHS atom.
10. `*OR*` vs. `*NOT*`: Same with 7 except it is the other way around.
11. `*OR*` vs. `*OR*`: Unify each daughter of the LHS with each daughter of the RHS, and the result is the collection of all successfully unified Vs without the isomorphic ones. Return `T` *iff* the result is non-empty. If the result is empty (returning `NIL`) the LHS is *not* modified. Several examples are shown below.

```
(*OR* 1 2 3) = (*OR* 1 4 5 0)
  result = T, result value = 1
(*OR* (*OR* 0 1) 1 3) = (*OR* 1 4 5 0)
  result = T, result value = (*OR* 1 0)
(*OR* (*NOT* 1 2 3) 4) = (*OR* 1 2 5 6)
  result = T, result value = (*OR* 5 6)
```

Listing 22: Examples of pseudo-unifying two `*OR*` Vs.

12. `*OR*` vs. `*MULTIPLE*`: Same with 8, but substituting the `*NOT*` value with the `*OR*` value.
13. `*MULTIPLE*` vs. atom: Same with 4 except it is the other way around.
14. `*MULTIPLE*` vs. `*NOT*`: Same with 8 except it is the other way around.
15. `*MULTIPLE*` vs. `*OR*`: Same with 8 except it is the other way around.
16. `*MULTIPLE*` vs. `*MULTIPLE*`: For each LHS daughter, from left to right, unifies it with each of the RHS daughter, from left to right. Return `T` *iff* all of the unifications succeed, in which case each unified daughter is pushed to the front of the new LHS, so the resulting `*MULTIPLE*` is a concatenation of the unified RHS daughters in the reversed order. Note that this operation is asymmetrical. Two examples are shown below.

```
(*MULTIPLE* 1 2 3) = (*MULTIPLE* (*NOT* 4) (*NOT* 5 6))
  result = T, result value = (*MULTIPLE* 3 3 2 2 1 1)
(*MULTIPLE* (*NOT* 4) (*NOT* 5 6)) = (*MULTIPLE* 1 2 3)
  result = T, result value = (*MULTIPLE* 3 2 1 3 2 1)
```

Listing 23: Examples of pseudo-unifying two `*MULTIPLE*` Vs.

8 Loading Multiple Grammars and Lexicons

In a real-world application grammars and lexicons are usually prepared as multiple modules. Each module can be made to correspond to a specific language (e.g., English vs. German), a domain model (e.g., travel domain vs. medical domain), etc. Only when it is necessary is a module loaded into UKernel. In general modularity promotes code reuse and provides a cleaner design of a natural language system. In this section we present the details of loading multiple grammars and lexicons.

8.1 Multiple Grammars

UKernel handles multiple grammars in a very simplistic way - there is no namespace introduced to individual grammar module, and all grammar modules, when loaded, are *visible* to UKernel. The ordering of rule appearance is mostly preserved: rules with same LHS or RHS are ordered based on their appearance in grammar files, but across different LHSs/RHSs the ordering of appearance is lost. It is totally up to a particular higher system implementation to decide whether to honor such ordering. For example, in the recursive-descent Generator we implemented, for a same LHS the system attempts to generate each of the different RHSs based on this ordering - so a rule which appears later than the others with the same LHS may be effectively *shadowed*. In systems such as this, as a general principle therefore the user should load in the most general grammar module first, and then load in the increasingly more specific modules[22].

8.2 Multiple Lexicons

Unlike the rather simple mechanism of loading multiple grammars, UKernel can load in multiple lexicons with an arbitrary loading ordering. Each lexicon needs to be assigned with a lexicon ID (Section 4.1), and lexical lookups can be targeted to a specific lexicon by setting the `:LEX-ID` argument (Section 4.2), or can be made to search through all lexicons by omitting the argument.

[22] There is no reason why a more sophisticated grammar module mechanism, including a properly designed namespace scheme, cannot be introduced to UKernel. The decision to stick with the simplistic approach is mainly for the backward compatibility with the 1988 GenKit. However we do not rule out the possibility of introducing a more flexible approach in the future (see Appendix B).

9 Suggestions for Writing an Efficient Grammar

Due to the ways UKernel is implemented, there are situations where theoretically equivalent grammar constructs have vastly different performances. Grammar developers are therefore strongly recommended to follow the suggestions given in this section should performance become an important consideration to them.

1. Efficiency comparison of the various operators (from the least to the most computationally intensive ones): Note this only gives a rough idea about the relative costs of various operators - it is not a conclusive benchmark comparison between them.

   “`=`” (with constraint Vs except for `*REMOVE*`) $\prec$ “`= *REMOVE*`” $\prec$ [“`<=`”, “`>`”] $\prec$ “`=i'`” $\prec$ “`=i`” $\prec$ “`=c'`” $\prec$ “`=c`” $\prec$ “`=t'`” $\prec$ “`=t`” $\prec$ “`=`” (pseudo-unifications) $\prec$ [“`==`”, “`<`”]
   (the operators grouped by “[]” have roughly the same cost)

2. Other things being equal, operations involving *destructive* operators are more expensive than those with *non-destructive* operators.

3. Other things being equal, operations involving *complex* FSs/Vs are more expensive than those with only *atomic* FSs/Vs. In particular the operations involving `*OR*` FSs/Vs are more expensive than the ones involving `*MULTIPLE*` because the formers require isomorphism tests to ensure there are no duplicates.

4. Other things being equal, operations involving *penetrating* FSPATHs are more expensive than those which do not.

5. Other things being equal, operations having RHS *penetrating* FSPATHs are more expensive than those which do not. This is because a collection process must be engaged to assemble a new FS/V from the RHS FSPATH (see Section 6.2).

6. Efficiency comparison of the two lexical lookup functions (from the least to the most computationally intensive ones):

   `GET-LEX` $\prec$ `GET-LEX-FS`

   This is because `GET-LEX-FS` needs to do FS tests (non-destructive pseudo-unifications) to identify the matching entries. If you can write equivalent equations using either one of the two, choose `GET-LEX`.

7. For `GET-LEX` and `GET-LEX-FS`: using `:LEX-ID` argument can speed up the search, while using `:CHECK` and `:AMBIGUITY` slows things down (even more true if both are used).

8. Efficiency comparison of `*CASE*`, `*EOR*` and `*OR*` equation blocks (from the least to the most computationally intensive ones):

   `*CASE*` $\prec$ `*EOR*` $\prec$ `*OR*`

   If you can write equivalent equations using any one of the three, choose `*CASE*`. `*OR*` is particularly expensive because isomorphism tests are engaged to remove the duplicate daughter FSs.

9. When writing equations, if possible try to put the less computationally intensive equations in the front - this way if any of them fails the more expensive ones will never get executed.

A Differences between the 1988 GenKit and UKernel

This section documents the differences between the 1988 GenKit and UKernel, in the hope that it will help GenKit users to migrate more easily. The differences are listed below.

1. Full support of complex values: The 1988 GenKit does not have full support of complex values (or the implementation is not clear, or inadequate).

   (a) UKernel allows unlimited embeddings of different value types within one another.

   (b) UKernel automatically compacts values so that they do not contain isomorphic components. The following listing shows several value compaction examples. Note that value compaction does not attempt to roll recursive embedding of the same type of Vs into one, e.g. it does not compact `(*or* 0 (*or* 1 2))` to `(*or* 0 1 2)`.

```
(*OR* 0 0 1) is (*OR* 0 1)
(*OR* 0 0 0) is 0
(*NOT* 0 0 1) is (*NOT* 0 1)
(*NOT* 0 0 0) is (*NOT* 0)
(*MULTIPLE* 0 0 1) is not compactable
(*OR* (*OR* 0 1) (*OR* 1 0)) is (*OR* 0 1)
(*OR* (*MULTIPLE* 0 1) (*MULTIPLE* 1 0)) is not compactable
(*OR* (*MULTIPLE* 0 (*OR* 0 1)) (*MULTIPLE* 0 (*OR* 1 0)))
  is (*MULTIPLE* 0 (*OR* 0 1))
(*OR* (*MULTIPLE* 0 (*OR* 0 1 (*MULTIPLE* 0 1)))
      (*MULTIPLE* 0 (*OR* 1 0 (*MULTIPLE* 1 0)))) is not compactable
```

Listing 24: Examples of value compaction.

   (c) All operations are well defined with respect to complex values (Section 7.3).

      i. `*NOT*` `=/=c` `*NOT*` behaves differently from the one in the 1988 GenKit (the manual is semi-accurate in describing the actual behaviors). In the old implementation the behavior is that it is `T` *iff* the RHS is a superset of the LHS (could be the same set).

      ii. The description of `*MULTIPLE*` `=/=c` `*MULTIPLE*` in the 1988 GenKit manual is entirely inaccurate. UKernel models after the old implementation itself.

2. Full support of complex FSs: Complex FSs are not documented in the 1988 GenKit, hence it is not clear whether they are fully supported (i.e., well-defined for all possible operators).

   (a) UKernel allows unlimited embeddings of different FS types within one another.

   (b) All operations are well defined with respect to complex FSs (Section 7.2).

3. Full support of penetrating FSPATHs (Section 6.2): UKernel allows the use of penetrating paths in both LHS and RHS. All operators are well-defined with respect to penetrating FSPATHs.

4. Constraint equations: The following type of constraint equations was not documented in the 1988 GenKit:

   (FSPATH `=c` FSPATH)

   For example: `((X0 SUBJ) =c (X1 OBJ))`

5. The new operators `=t` (*test* operator) and `=i` (*isomorphism* operator) (Section 7): UKernel provides these two destructive operators for the purpose of testing the unifiability/isomorphism of two FSs.

6. The new operators `=i'`, `=c'`, and `=t'`: they are the non-filtering versions of `=i`, `=c`, and `=t`, respectively, and should be faster than the filtering ones.

7. Lexical lookups (Section 4.2):
   (a) The use of the keyword arguments are new (`:LEX-ID`, `:CHECK`, and `:AMBIGUITY`).
   (b) Added `GET-LEX-FS`.
   (c) All arguments, when appropriate, can be either quoted V/FS, or a FSPATH.
8. The following constraint values are new: `*NUMBER*`, `*INTEGER*`, `*POSITIVE*`, `*NOT-NUMBER*`, `*NOT-INTEGER*` and `*NOT-POSITIVE*`(Section 6.1).
9. The starting rule (the one with `<START>` as the LHS) is mandatory in UKernel. However it can be overridden during run-time - the specification of a starting rule only serves as a default choice.

B Implementation Notes and Future Works

In this section we describe some of the implementation details of UKernel, and suggest the future works. The section is recommended to system developers, especially the front-end developers who would like to incorporate UKernel as part of their natural language systems[23].

UKernel is implemented in standard C++, with extensive use of the Standard Template Library (STL)[24]. The implementation relies on *Toolbox*, another independent library we implemented, which provides code of some basic facilities, such as token strings/dictionaries, indented printing, generic tree and context-free grammar structures, etc. At the time of this writing UKernel is known to run on both Linux and Windows NT/2000/XP[25].

In the rest of the section we first present the details of the various code modules, together with the relevant application programming interface (API) functions for the developers in Section B.1, and then a list of future works is given in Section B.2.

B.1 Code Modules and the APIs

In UKernel the code is organized around the major data structures it implements: *symbols*, *values*, *feature structures*, *equations*, *equation blocks*, *lexicons* and *grammars* (in the order of increasing complexity). For a front-end which parses grammars and lexicons, it must initialize each of these data structures based on the input files. The following sub-sections present the details together with some coding examples for each module.

B.1.1 Symbols

A symbol is a "numeralized" string token - in initializing it a string token is converted internally into a unique number ID so comparing two symbols would amount to comparing two numbers, with better efficiency. A symbol can also be initialized by one of the *special code* (see Fig. 21) - these special symbols are used as the type keyword of complex FSs/Vs (Section 6.1), as constraint values (Section 6.1), or as operators (Section 7). Several examples below illustrate how to initialize, compare and print out symbols.

```
#include "symbol.hpp"
#include <iostream>
//...

Symbol symA("apple");     // initialize symA with "apple"
Symbol symB(symA);        // initialize symB with symA
Symbol symC, symD;
Symbol symE(Symbol::MULTIPLE);   // head for *MULTIPLE* FSs
Symbol symF,symG;

symC.set("banana");       // initialize symC with "banana"
symD=symC;                // assign symC to symD

symF.setSpecial(Symbol::DEFINED);   // constraint value
symG.setSpecial(Symbol::OP_PUSH);   // operator

if (symA==symB)           // comparing symbols
  cout<<symA<<" is equal to "<<symB<<endl;    // this prints out string tokens

if (symA!=symC)
  cout<<symA<<" is not equal to "<<symC<<endl;
```

[23]However nothing can replace the pleasure of peeking into the source code. Interested readers can obtain the source code via `http://www.cs.cmu.edu/~benhdj`.

[24]More specifically, UKernel has been implemented early on using GNU C++ version 3.0-3.2 on a Linux machine, and later using GNU C++ version 3.3 on a Macintosh running Mac OS X 10.3.4.

[25]With the help of Cygwin development suite, which can be freely downloaded from `http://sources.redhat.com/cygwin/`.

```
cout<<symE<<endl;          // this prints out "*MULTIPLE*"
cout<<symF<<endl;          // this prints out "*DEFINED*"
cout<<symG<<endl;          // this prints out ">"
//...
```

Listing 25: Code excerpt demonstrating the use of `Symbol`.

```
enum SymCode{
NUMBER=-25,INTEGER,POSITIVE,NOT_NUMBER,NOT_INTEGER,NOT_POSITIVE,
DEFINED,UNDEFINED,REMOVE,MULTIPLE,OR,NOT,
OP_PSEUDO_UNIFY,OP_CONSTRAIN,OP_CONSTRAIN_NOFILTER,OP_TEST,OP_TEST_NOFILTER,
OP_ISO,OP_ISO_NOFILTER,OP_ASSIGN,OP_REMOVE_ASSIGN,OP_PUSH,OP_POP,BOX,UNKNOWN};
```

Figure 21: Special code for Symbol (defined in `symbol.hpp`).

Several global symbols are already defined for use, as shown in Table 2.

Symbol name	**Initialized with**	**Remarks**
`symNot`	`Symbol::NOT`	Head of `*NOT*` value (Section 6.1).
`symOr`	`Symbol::OR`	Head of `*OR*` FS/V (Section 6.1).
`symMultiple`	`Symbol::MULTIPLE`	Head of `*MULTIPLE*` FS/V (Section 6.1).
`symNull`	“`NIL`”	For a TV.
`symWildcard`	“`%`”	Wildcard (Section 3).
`symValue`	“`VALUE`”	Pre-defined feature name for wildcard VAR (Section 5.1).
`symSemValue`	“`SEM-VALUE`”	One of the added features in the matching lexical entries (Section 4.2).
`symCat`	“`CAT`”	Mendatory feature of a lexical definition (Section 4.1).
`symID`	“`LEX-ID`”	One of the added features in the matching lexical entries (Section 4.2).
`symGetLex`	“`GET-LEX`”	Lexical lookup function name (Section 4.2).
`symGetLexFS`	“`GET-LEX-FS`”	Lexical lookup function name (Section 4.2).
`symLexID`	“`:LEX-ID`”	Keyword arguement for the lexical lookup functions (Section 4.2).
`symCheck`	“`:CHECK`”	Keyword arguement for the lexical lookup functions (Section 4.2).
`symAmbiguity`	“`:AMBIGUITY`”	Keyword arguement for the lexical lookup functions (Section 4.2).

Table 2: Pre-defined global symbols (defined in `symbol.cpp`).

Note that the symbol modules are the only part of the implementation which deals directly with string tokens. This makes the future internationalization of UKernel easier.

B.1.2 Values

A value (V) is essentially a tree of symbols. For complex Vs the root node must carry one of the pre-defined type keyword symbols (`symNot`, `symOr`, or `symMultiple`, see Table 2). A user must construct the value tree node-by-node, by using the *value iterators*, the tree method `insert()` and `begin()`, the tree iterator method `end()`, and the value method `compact()`, as shown in the following listing.

```
#include "value.hpp"
#include <iostream>
//...

Value v1,v2,v3;
Value::Iterator iter;
```

```
// v1 is now constructed as an atomic V "0"
v1.insert(v1.begin(),Symbol("0")); // v1.begin() points to v1's root
cout<<v1<<endl;                    // this prints out "0"

// v2 is now constructed as a constraint V "*REMOVE*"
v2.insert(v2.begin(),Symbol(Symbol::REMOVE));
cout<<v2<<endl;                    // this prints out "*REMOVE*"

// v3 is now constructed as (*OR* 0 0 1)
iter=v3.insert(v3.begin(),symOr);
v3.insert(iter.end(),Symbol("0")); // insert "0" before the last daughter
v3.insert(iter.end(),Symbol("0"));   // ... of the root
v3.insert(iter.end(),Symbol("1"));
v3.compact();                      // compact the value
cout<<v3<<endl;                    // this prints out "(*OR* 0 1)"

// clear all values - maybe for another round of initializations?
v1.clear();
v2.clear();
v3.clear();
//...
```

Listing 26: Code excerpt for initializing values.

The purposes of various methods should be clear in the examples. In particular it is recommended to call `compact()` for every newly initialized value as it is potentially helpful to optimize the performance (see Appendix A for more on value compaction). Also when initializing a recursive complex V, more than one iterator might be necessary for constructing the tree correctly, and using a stack to keep track of the iterators is a good idea.

The values module is internally implemented based on template `Tree` from library Toolbox.

B.1.3 Feature Structures

Similar to a V, a feature structure (FS) is essentially a tree of Vs. However to construct an FS a user does not need to construct the tree node-by-node as for constructing a V. The FS method `assign()` can be called with a PATH and a V to put the V to the right place, as shown in the following examples.

```
#include "fStruc.hpp"
#include <iostream>
//...

FStruc X0;
Path path;
Value v;

// initialize X0 to be
// ((SUBJ
//    ((PRED YOU)
//     (PERS 2)
//     (NUM (*OR* SG PL))))
//   (PRED RUN))

// set the name of the FS
// if we omit this the default name is "NIL"
X0.setName(Symbol("X0"));
```

```
// (X0 SUBJ PRED) <= YOU
path.clear();
path.push_back(Symbol("SUBJ"));
path.push_back(Symbol("PRED"));
v.clear();
v.insert(v.begin(),Symbol("YOU"));
v.compact();
X0.assign(path,v);

// (X0 SUBJ PERS) <= 2
path.clear();
path.push_back(Symbol("SUBJ"));
path.push_back(Symbol("PERS"));
v.clear();
v.insert(v.begin(),Symbol("2"));
v.compact();
X0.assign(path,v);

// (X0 SUBJ NUM) <= (*OR* SG PL)
path.clear();
path.push_back(Symbol("SUBJ"));
path.push_back(Symbol("NUM"));
v.clear();
vIter=v.insert(v.begin(),symOr);
v.insert(vIter.end(),Symbol("SG"));
v.insert(vIter.end(),Symbol("PL"));
v.compact();
X0.assign(path,v);

// (X0 PRED) <= RUN
path.clear();
path.push_back(Symbol("PRED"));
v.clear();
v.insert(v.begin(),Symbol("RUN"));
v.compact();
X0.assign(path,v);

// this prints out the FS with proper indentation (default)
cout<<X0<<endl;

// this prints out the FS without indentation (everything is on one line)
FStruc::indentPrint=false;
cout<<X0<<endl;

// maybe for another round of initialization?
X0.clear();
//...
```

Listing 27: Code excerpt for initializing feature structures.

In an analysis grammar file the only places where initializing FSs is necessary are for parsing QFSs. In parsing a lexicon file, however, the definition of every lexical entry is an FS.

The FS module is internally implemented based on template `Tree` from library Toolbox.

B.1.4 Equations

An equation is actually a degraded *main equation block* (or interchangeably, an AND equation block), in that it contains nothing but a single equation. There are four types of equations, and the initialization of each of them is illustrated below.

```
#include "eBlock.hpp"
#include "grammar.hpp"  // for type 4 equations
#include "fStruc.hpp"   // for type 5 equations
//...

EBlockMain *equPtr1,*equPtr2,*equPtr3,*equPtr4,*equPtr5;
Path lPath,rPath;
Value v,*vPtr;
EFArgs req,opt;   // required and optional arguements
FSPath *fsPathPtr;
Symbol symCatN("N"),symPred("PRED");
Grammar g(lexicons,Symbol("START"));   // an empty grammar
FStruc x0;

// Equation type 1 (the RHS is an FSPath)
// equPtr1 points to the equation "X1 == (X0 SUBJ)"
lPath.clear();
rPath.clear();
rPath.push_back(Symbol("SUBJ"));
equPtr1=new EBlockMain(1,lPath,0,rPath,Symbol::OP_REMOVE_ASSIGN);

// Equation type 2 (the RHS is a V)
// equPtr2 points to the equation "(X0 PERS) <= 2"
lPath.clear();
lPath.push_back(Symbol("PERS"));
v.clear();
v.insert(v.begin(),Symbol("2"));
equPtr2=new EBlockMain(0,lPath,v,Symbol::OP_ASSIGN);

// Equation type 3 (the RHS is a constraint V)
// equPtr3 points to the equation "(X0 NUM) = *DEFINED*"
lPath.clear();
lPath.push_back(Symbol("NUM"));
equPtr3=new EBlockMain(0,lPath,Symbol::DEFINED);

// Equation type 4 (the RHS is an extension function)
// equPtr4 points to the equation
//   "X1 <= (GET-LEX (X0 PRED) 'N :AMBIGUITY T :LEX-ID 'ENG)"
req.clear();                          // set up the required arguements
fsPathPtr=&req.newFSPath();
fsPathPtr->fsIdx=0;
fsPathPtr->path.push_back(symPred);
vPtr=&req.newValue();
vPtr->insert(vPtr->begin(),symCatN);
opt.clear();                          // set up the optional (keyword) arguements
opt.newBool(symAmbiguity,true);
vPtr=&opt.newValue(symLexID);
vPtr->insert(vPtr->begin(),Symbol("ENG"));
```

```
49 lPath.clear();
50 equPtr4=new EBlockMain(1,lPath,symGetLex,req,opt,Symbol::OP_ASSIGN,g));
51
52 // Equation type 5 (the RHS is a feature structure)
53 // equPtr5 points to the equation
54 // X0 <= '((SUBJ
55 //           ((PRED YOU)
56 //            (PERS 2)
57 //            (NUM (*OR* SG PL))))
58 //          (PRED RUN))
59 // ASSUMING feature structure x0 is already initialized to become the quoted
60 // feature structure
61
62 // ... set up x0 here ...
63 lPath.clear()
64 equPtr5=new EBlockMain(0,lPath,x0,Symbol::OP_ASSIGN);
65
66 // free memory
67 delete equPtr1;
68 delete equPtr2;
69 delete equPtr3;
70 delete equPtr4;
71 delete equPtr5;
72
73 //...
```

Listing 28: Code excerpt for initializing equations.

Initializing equations of type 1 to 3 is fairly straightforward. The only thing to note is the use of the index to address a VAR (FS); e.g. on line 20 of Listing 28 `X0` is denoted by the integer argument 0 and `X1` is denoted by the integer argument 1. These numbers refer to the element FSs in the `FSRegisters` of the class `Grammar` (Section B.1.7).

Initializing equations of type 4, which are equations using *extension functions* on the RHS, requires a little more explanation. An extension function consists of a function name, a list of required arguments, a list of optional keyword arguments, and an actual function implementation. There are already two built-in extension functions available: `GET-LEX` and `GET-LEX-FS` for lexical lookups (Section 4.2). To use them as the RHS of type 4 equations, we need to first set up the required and the optional arguments as shown between line 39 to 50 in Listing 28. There are two sets of `new*()` methods for doing this, and the set without specifying names of arguments should be used for the required arguments (since they are nameless), and the other set should be used for the optional arguments. The initialization ordering *is* significant for the required arguments, but it *is not* for the optional ones (since they have names). Fig. 22 gives a glimpse at the arguments interface.

(new in version 1.91) Setting up equations of type 5 is almost identical to initializing equations of type 3 - the only difference is that the RHS is a feature structure instead of a value. In terms of grammar formalism, the RHS is a quoted FS. The code on line 64 assumes that feature structure `x0` is already initalized before this point (see B.1.3 for how to set up a feature strucutureq).

We shall only say a few words on defining new extension functions - the interested readers should definitely refer to the code `eFunc.hpp` and `eFunc.cpp` for the details. Basically the process involves preparing a function implementation, and hooking it up with the rest of the system. More specifically the prototype (signature) of an extension function needs to be inserted into an internal `_EFTable` so that the system will be able to recognize a particular function call and find the right function implementation to execute.

The memory allocated for the equations does not need to be freed when the equations are added into an equation block, as it is taken care of by the destructor of the host equation block (see next sub-section).

```
//...
class EFArgs: protected vector<EFArg>
{
  //...
  Value &newValue ();
  FStruc &newFS ();
  FSPath &newFSPath ();
  void newBool (bool t);

  Value &newValue (const Symbol &name);
  FStruc &newFS (const Symbol &name);
  FSPath &newFSPath (const Symbol &name);
  void newBool (const Symbol &name, bool t);
  //...
};
```

Figure 22: Arguments interface for extension functions (declared in `eFArgs.hpp`).

B.1.5 Equation Blocks

There are four types of equation blocks, `EBlockMain`, `EBlockCase`, `EBlockOr` and `EBlockEor`. To initialize an equation block a user essentially creates sub-blocks and add them one-by-one to the host block. This is illustrated in the following listing.

```
#include "eBlock.hpp"
#include <iostream>
//...

EBlockMain mainEB,*mainEBPtr;
EBlockOr orEB;
Path lPath,rPath;
Value v;

//========== Initialize a main (AND) equation block ==========
mainEB.clearBlocks();

// X1 == (X0 SUBJ)
lPath.clear();
rPath.clear();
rPath.push_back(Symbol("SUBJ"));
mainEB.addBlock(new EBlockMain(1,lPath,0,rPath,Symbol::OP_REMOVE_ASSIGN));

// X2 <= X0
lPath.clear();
rPath.clear();
mainEB.addBlock(new EBlockMain(2,lPath,0,rPath,Symbol::OP_ASSIGN));

// (X2 PERS) = (X1 PERS)
lPath.clear();
lPath.push_back(Symbol("PERS"));
mainEB.addBlock(new EBlockMain(2,lPath,1,lPath,Symbol::OP_PSEUDO_UNIFY));

// (X2 NUM) = (X1 NUM)
```

```
lPath.clear();
lPath.push_back(Symbol("NUM"));
mainEB.addBlock(new EBlockMain(2,lPath,1,lPath,Symbol::OP_PSEUDO_UNIFY));

// this prints out the block as
// ((X1 == (X0 SUBJ))
//   (X2 <= X0)
//   ((X2 PERS) = (X1 PERS))
//   ((X2 NUM) = (X1 NUM)))
cout<<mainEB<<endl;

//========== Initialize an OR equation block ==========
orEB.clearBlocks();

// the 1st block: (((X0 SUBJ NUM) =c SG)
//                 ((X1 NUM) = SG)
//                 ((X2 NUM) = (X1 NUM)))
mainEBPtr=new EBlockMain();
lPath.clear();
rPath.clear();
lPath.push_back(Symbol("SUBJ"));
lPath.push_back(Symbol("NUM"));
v.clear();
v.insert(v.begin(),Symbol("SG"));
mainEBPtr->addBlock(new EBlockMain(0,lPath,v,Symbol::OP_EQUAL));
lPath.clear();
rPath.clear();
lPath.push_back(Symbol("NUM"));
v.clear();
v.insert(v.begin(),Symbol("SG"));
mainEBPtr->addBlock(new EBlockMain(1,lPath,v,Symbol::OP_PSEUDO_UNIFY));
lPath.clear();
rPath.clear();
lPath.push_back(Symbol("NUM"));
rPath.push_back(Symbol("NUM"));
mainEBPtr->addBlock(new EBlockMain(2,lPath,1,rPath,Symbol::OP_PSEUDO_UNIFY));
orEB.addBlock(mainEBPtr);

// the 2nd block: (((X0 SUBJ NUM) =c PL)
//                 ((X1 NUM) = PL)
//                 ((X2 NUM) = (X1 NUM)))
mainEBPtr=new EBlockMain();
lPath.clear();
rPath.clear();
lPath.push_back(Symbol("SUBJ"));
lPath.push_back(Symbol("NUM"));
v.clear();
v.insert(v.begin(),Symbol("PL"));
mainEBPtr->addBlock(new EBlockMain(0,lPath,v,Symbol::OP_EQUAL));
lPath.clear();
rPath.clear();
lPath.push_back(Symbol("NUM"));
v.clear();
v.insert(v.begin(),Symbol("PL"));
```

```
mainEBPtr->addBlock(new EBlockMain(1,lPath,v,Symbol::OP_PSEUDO_UNIFY));
lPath.clear();
rPath.clear();
lPath.push_back(Symbol("NUM"));
rPath.push_back(Symbol("NUM"));
mainEBPtr->addBlock(new EBlockMain(2,lPath,1,rPath,Symbol::OP_PSEUDO_UNIFY));
orEB.addBlock(mainEBPtr);

// this prints out the block as
// (*OR* (((X0 SUBJ NUM) =c SG)
//         ((X1 NUM) = SG)
//         ((X2 NUM) = (X1 NUM)))
//        (((X0 SUBJ NUM) =c PL)
//         ((X1 NUM) = PL)
//         ((X2 NUM) = (X1 NUM))))
cout<<orEB<<endl;

//========== Initialize a CASE equation block==========
lPath.clear();
lPath.push_back(Symbol("BREED"));
EBlockCase caseEB(0,lPath);

// the 1st block: (PITBULL (((X1 VALUE) <= BIG_)))
mainEBPtr=new EBlockMain();
rPath.clear(); lPath.clear();
lPath.push_back(symValue);
v.clear();
v.insert(v.begin(),Symbol("BIG_"));
mainEBPtr->addBlock(new EBlockMain(1,lPath,v,Symbol::OP_ASSIGN));
v.clear();
v.insert(v.begin(),Symbol("PITBULL"));
ebCasePtr->addCaseBlock(v,mainEBPtr);

// the 2nd block: (CHIHUAHUA (((X1 VALUE) <= SMALL_)))
mainEBPtr=new EBlockMain();
rPath.clear(); lPath.clear();
lPath.push_back(symValue);
v.clear();
v.insert(v.begin(),Symbol("SMALL_"));
mainEBPtr->addBlock(new EBlockMain(1,lPath,v,Symbol::OP_ASSIGN));
v.clear();
v.insert(v.begin(),Symbol("CHIHUAHUA"));
ebCasePtr->addCaseBlock(v,mainEBPtr);

// this prints out the block as
// (*CASE* (X0 BREED)
//      (PITBULL (((X1 VALUE) <= BIG_)))
//      (CHIHUAHUA (((X1 VALUE) <= SMALL_))))
cout<<caseEB<<endl;

//...
```

Listing 29: Code excerpt for initializing equation blocks.

When an equation block is destructed (via its destructor), the memory associated with its embedded equations/equation blocks will be freed automatically.

B.1.6 Lexicons

Lexicons are containers of individual lexical entries. A user can obtain a `Lexicon` object by invoking the method `Lexicons::addLexicon()` with an appropriate lexicon ID (Section 8.2), then add in individual entries one-by-one. This is shown in the following listing.

```
#include "lexicon.hpp"
#include <iostream>
//...

Lexicons lexicons;
Path path;
FStruc *fsPtr;
Value v;
Symbol symCatN("N"),symRoot("ROOT"),symClass("CLASS");

Lexicon &engLex=lexicons.addLexicon(Symbol("ENG"));  // create an English lexicon

// add in entry (DOG ((CAT N) (ROOT DOG) (CLASS *ANIMAL*)))
fsPtr=&engLex.addLex(Symbol("DOG"),symCatN);
path.clear();
path.push_back(symRoot);
v.clear();
v.insert(v.begin(),Symbol("DOG"));
fsPtr->assign(path,v);
path.clear();
path.push_back(symClass);
v.clear();
v.insert(v.begin(),Symbol("*ANIMAL*"));
fsPtr->assign(path,v);

Lexicon &gerLex=lexicons.addLexicon(Symbol("GER"));  // create a German lexicon

// add in entry (DOG ((CAT N) (ROOT Hund)))
fsPtr=&gerLex.addLex(Symbol("DOG"),symCatN);
path.clear();
path.push_back(symRoot);
v.clear();
v.insert(v.begin(),Symbol("Hund"));
fsPtr->assign(path,v);
path.clear();
path.push_back(symClass);
v.clear();
v.insert(v.begin(),Symbol("*ANIMAL*"));
fsPtr->assign(path,v);

// this prints out the entire lexicons: both the English one and the German one
cout<<lexicons<<endl;

// or you can print out an individual lexicon by finding it first
cout<<lexicons.findLexicon(Symbol("ENG"))<<endl;
```

```
//...
```

Listing 30: Code excerpt for initializing lexicons.

B.1.7 Grammars

(Revised for version 1.92) A `Grammar` essentially consists of a list of grammatical rules. There are two kinds of grammars in UKernel: grammars for *generation* and grammars for *analysis*. Both kinds of grammars feature the *same* set of APIs, and whether one kind is used instead of the other is determined at compile-time, depending on whether `UK_GEN` (for generation grammars) or `UK_ANALYSIS` (for analysis grammars) is defined (one of the two must be defined). This can be done with GNU C++ compiler via the command-line option `-D` (the default Makefile uses `-DUK_GEN`).

Both kinds of grammars are indexed by keys: for generation grammars the keys are LHSs of the rules (`Symbols`), and for analysis grammars the keys are RHSs of the rules (class `RHS`, essentially a list of `RLiterals`, is defined in `_cfgBase.hpp in Toolbox`) - this is due to the difference in the nature of a generation task and an analysis (parsing) task. Regardless of the difference in keys, the rules with the same key are ordered based on their appearances in the grammar file (so an earlier rule will be placed before any later ones; see Section 8.1). The body of a rule (`RuleBody`) then contains a main equation block (`EBlockMain`, and AND block; see B.1.5). When initializing a grammar, the `RuleBody` of a new rule is returned by calling `Grammar::addRule()` method with the desired LHS and RHS, and then the equation block inside the `RuleBody` is initialized. This is illustrated in the following listing.

```
#include "grammar.hpp"
#include <iostream>
#include <stdarg.h>
//...

Lexicons lexicons;

// ... set up lexicons ...

// a convenience function to create an FS path
void makePath (Path &path, char *first , ...)
{
  va_list argp;
  char *ptr;

  path.clear();
  if (first==NULL) return;

  path.push_back(Symbol(first));

  for (va_start(argp, first);(ptr=va_arg(argp,char*))!=NULL;
        path.push_back(Symbol(ptr)));
  va_end(argp);
}


// set up a grammar with starting LHS "S"
// whether it is for generation or for analysis is determined by
// whether UK_GEN or UK_ANALYSIS is #defined.
Grammar g(lexicons,Symbol("S"));
Grammar::RHS rhs;
```

```
const Grammar::RuleIterator ri,riFirst,riLast;

RuleBody *rbPtr;
Path lPath,rPath;
Value v;

// ******** Rule: <S> ==> (<NP> <VP>) ********************************
rhs.clear();
rhs.push_back(RLiteral(Symbol("NP")));      // for NTs no boolean is passed
rhs.push_back(RLiteral(Symbol("VP")));
rbPtr=&g.addRule(Symbol("S"),rhs);          // for non-char-level rules no
                                            // final boolean argument is needed

//--------- Equ: X1 == (X0 SUBJ)
makePath(lPath,NULL);
makePath(rPath,"SUBJ",NULL);
rbPtr->eBlock.addBlock(new EBlockMain(1,lPath,0,rPath,Symbol::OP_REMOVE_ASSIGN));

//--------- Equ: (X2 PERS) = (X1 PERS)
makePath(lPath,"PERS",NULL);
makePath(rPath,"PERS",NULL);
rbPtr->eBlock.addBlock(new EBlockMain(2,lPath,1,rPath,Symbol::OP_PSEUDO_UNIFY));

//--------- Equ: (X2 NUM) = (X1 NUM)
makePath(lPath,"NUM",NULL);
makePath(rPath,"NUM",NULL);
rbPtr->eBlock.addBlock(new EBlockMain(2,lPath,1,rPath,Symbol::OP_PSEUDO_UNIFY));

//--------- Equ: X2 = X0
makePath(lPath,NULL);
makePath(rPath,NULL);
rbPtr->eBlock.addBlock(new EBlockMain(2,lPath,0,rPath,Symbol::OP_PSEUDO_UNIFY));


// ******** Rule: <DET> --> (T H E) ********************************
rhs.clear();
rhs.push_back(RLiteral(Symbol("T"),false));   // note an extra "false" for terminals
rhs.push_back(RLiteral(Symbol("H"),false));   //  (default is "true": non-terminals)
rhs.push_back(RLiteral(Symbol("E"),false));
rbPtr=&g.addRule(Symbol("DET"),rhs,true);     // "true" signifies a char-level rule
                                              //  (default is "false")

//--------- Equ: (X0 FIN) = +
makePath(lPath,"FIN",NULL);
makePath(rPath,NULL);
v.clear();
v.insert(v.begin(),Symbol("+"));
rbPtr->eBlock.addBlock(new EBlockMain(0,lPath,v,Symbol::OP_PSEUDO_UNIFY));

// print out # of rules
cout<<"* # of rules = "<<g.size()<<endl;

// print out # of unique LHS literals
cout<<"* # of unique LHS literals = "<<g.lhsSize()<<endl;
```

```
// this prints out the entire grammar
cout<<g<<endl;

// if we want we can also change the starting LHS
// but be careful: the specified LHS must exist in the grammar
g.setStart(Symbol("DET"));

// find the rules with the LHS "<NP>"
// findRule() is a method of _CFGBase (see _cfgBase.hpp)
if (g.findRule(Symbol("NP"),riFirst,riLast)) {
  // yes we found some rules

  // print out rach rule found in the form "LHS -> RHS"
  // and print out the equation block of the rule
  for (ri=riFirst;;++ri) {
    cout<<gi.readLHS()<<" -> "<<ri->readRHS()<<endl;
    cout<<ri->body.eBlock<<endl;
    if (ri==riLast) break;
  }
}

//...
```

Listing 31: Code excerpt for initializing a grammar.

Note the code assumes you have already initialized a lexicon (see B.1.6), which is then passed into the Grammar constructor to build an empty grammar. Also note that for a *character-level* rule (Section 3) when calling `Grammar::addRule()` one needs to set the boolean argument to `true`. In the RHS for a terminal an extra boolean argument `false` must be passed into the constructor of `RLiteral` to indicate that it is not an NT. For more sample code, readers are referred to the source files `graTstSub.cpp` and `graTst.cpp` in UKernel, and `cfgTst.cpp` in Toolbox[26] .

As mentioned in Section B.1.4 about specifying a VAR using an integer (e.g., `0` for `X0`), `Grammar` has its own `FSRegisters` - an array of FSs to accommodate all VARs. The size of the registers will be automatically adjusted os that it is guaranteed to be big enough (as long as there is no attempt to access an out-of-range VAR[27]).

The grammar module is internally implemented by instantiating either template `CFGGen` (generaion, in `cfgGen.hpp`) or `CFGAnalysis` (analysis, in `cfgAnalysis.hpp`) from library Toolbox. Both templates inherit from the base template `_CFGBase` (in `_cfgBase.hpp`), which defines the interactions between an index and a list of rules, abstract enough to accomodate grammars of both directions. Finally, a special note about the mechanism of rule removal in the current implementation: since removing rules requires additional code which brings run-time overhead if it is not needed, the functionality by default is turned off by *not* defining the macro `WITH_REMOVE_RULE` (see the `Makefile` of Toolbox). Therefore to switch from a built with rule removal functionality built-in to a built without (or vice versa), one needs to undefine/define the macro in the `Makefile` of Toolbox and then recompile.

B.2 Future Works

The possible extensions to the current UKernel are listed in the order of decreasing priority.

1. Internationalization: This will allow grammar developers to write grammars in various natural languages encodings, e.g., Chinese, Japanese, etc. Since direct string manipulations are restricted in the Symbol modules, this should be fairly straightforward.

[26]This one demonstrates how you define your own context-free grammar from the ground up, which is basically how class `Grammar` is defined in UKernel.

[27]Otherwise the system might crash.

2. Integrating a morphology package: This would make processing with morphologically rich languages possible. PC-Kimmo is one possibility.

3. Introducing more sophisticated grammar module scheme: As discussed in Section 8.1 UKernel adopts a rather inflexible way to handle multiple grammar modules. A more sophisticated one with a proper design of the namespaces could further facilitate the development of modular grammars.

4. Embedding a scripting language: This will allow grammar developers to directly embed code in their grammars. This is probably the only advantage the 1988 GenKit still has over UKernel (since GenKit was implemented in LISP). For example the entire lexical lookup functions `GET-LEX` and `GET-LEX-FS` can be straightforwardly added in GenKit using the native LISP language (at the expense of system performance). With code embedding grammar developers can tailor the system as they see fit without the hassle of rebuilding the entire system.

5. Supporting full-unifications: With full-unifications UKernel can be more useful for the more theoretical pursuits (such as the research on bidirectional grammars). One possibility is to sport a complete Lexical Functional Grammar (LFG) system on it.

6. Augmenting UShell: Add in the missing support of equation blocks and lexical lookup equations so users can try out all functionality UKernel offers in an interactive environment.

7. Optimization: For example, change the use of `map` to hash tables.

C Using UShell

UShell is an interactive shell in which a user can type in most of the UKernel-supported equations and immediately sees the results. This is useful for beginners to learn different operators, as well as helpful for system/grammar developers to pinpoint their problems. To run UShell, simply type

```
./ush
```

The startup screen is shown in Fig. 23. The prompt in UShell is ">", and there are two types of commands available: the shell commands (with a prefix "`!`") and the kernel commands. In particular the shell command "`!help`" prints out an overview help screen, as shown in Fig. 24. In this section we shall briefly introduce some of the commands - the user is encouraged to take advantage of the help facility (note the shell command "`!help`" followed by a command name gives the more detailed instructions on the given command).

```
UShell - an interactive shell for the Unification Kernel (UKernel) v1.92
(c) 2004 Benjamin Han <benhdj@cs.cmu.edu>
Type "!help" to list the commands available.

* Default FS registers size is 20
>
```

Figure 23: Startup screen of UShell.

Internally UShell has an array of VARs (or interchangeably, FSs) - by default these are `X0` to `X19`, hence the message on the startup screen indicating that the default size of the FS registers is 20. The size of the registers can be adjusted via executing the shell command "`!size`". Attempts to access out-of-range VARs will result in an error message. The content of all registers, or individual registers, can be printed out using the shell command "`!show`".

To quit UShell, type "`!quit`", "`!exit`", or "`!bye`".

Several limitations exist in the current version of UShell: it does not support equation blocks, nor does it support the lexical lookup equations. Future versions might add in the full support of all possible equations.

```
> !help
* Complete list of UShell commands:
  (FS=Feature Structure; v=Value; LHS=Left-hand Side; RHS=Right-hand Side)

----------- Shell commands (case-insensitive; with a prefix '!') -----------
* help [cmd/operator]: description of commands/operators.
* quit | exit | bye | ctrl-c | ctrl-d: Exit UShell.
* show [fs1] [fs2]...: show the content of feature structure(s).
* indent [on|off]: turn on/off or show indentation printing status.
* size [n]: set the size of the FS registers to 'n'.

-------------- Kernel commands (case-sensitive) --------------
* FS | (FS path) = v | FS | (FS path): pseudo-unification.
* FS | (FS path) =c v | FS | (FS path): constraint test.
* FS | (FS path) =c' v | FS | (FS path): constraint test (no filtering).
* FS | (FS path) =t v | FS | (FS path): equivalence test.
* FS | (FS path) =t' v | FS | (FS path): equivalence test (no filtering).
* FS | (FS path) =i v | FS | (FS path): isomorphism test.
* FS | (FS path) =i' v | FS | (FS path): isomorphism test (no filtering).
* FS | (FS path) <= v | FS | (FS path): assign RHS to LHS.
* FS | (FS path) == FS | (FS path): removal assign RHS to LHS.
* FS | (FS path) > FS | (FS path): push RHS to LHS.
* FS | (FS path) < FS | (FS path): pop the 1st element of RHS to LHS.
* FS | (FS path) = *DEFINED*: test if LHS is defined.
* FS | (FS path) = *UNDEFINED*: test if LHS is undefined.
* FS | (FS path) = *REMOVE*: remove LHS.
* FS | (FS path) = *NUMBER*: test if LHS is a real number.
* FS | (FS path) = *INTEGER*: test if LHS is an integer.
* FS | (FS path) = *POSITIVE*: test if LHS is a positive real number.
* FS | (FS path) = *NOT-NUMBER*: test if LHS is not a real number.
* FS | (FS path) = *NOT-INTEGER*: test if LHS is not an integer.
* FS | (FS path) = *NOT-POSITIVE*: test if LHS is not a positive real number.

* A value can be either atomic or complex:
  Atomic: e.g. apple, banana, 12, etc.
  Complex: e.g. (*NOT* apple banana), (*OR* apple banana),
           (*MULTIPLE* apple, banana), or any possible embeddings.

* An FS must be named as X0, X1, ... etc - the name is used to refer
  to the corresponding FS register. You can increase the size of the
  FS registers by using shell command 'size'

* A path is a list of feature names delimited by space.

* Implementation note: so far UShell doesn't support equation blocks
  (multiple equations grouped by *OR*, *EOR*, *CASE*, or a pair of
  parentheses - in the case it's an implicit *AND*), and all lexical
  lookup equations.
>
```

Figure 24: Overview help screen in UShell.

D Execution Trace of the Toy Grammar

By default the generator runs in the "fork" mode: in this mode if an *ambiguous* feature structure is encountered, i.e., an `*OR*` feature structure such as `(*OR* A B)`, then each branch of the structure will be used in turn in the generation process (e.g., first `A` and then `B`, etc.), and the process stops at the first successful generation. This is illustrated in the following execution trace, which is obtained by running the generator using the grammar shown in Listing 1, and the input feature structure given in Listing 3.

```
* Running the generator with tracing (verbosity=showRule; modeFork):
[1] Call '<S> ==> (<NP> <VP>)' : Succeeded.
  [2] Fork Call '<NP> ==> (<DET> <N>)' : Succeeded.
    [3] Call '<DET> --> (T H E)' : Failed.
    [3] Call '<DET> --> (A)' : Succeeded.
      [4] Generate a token 'A' without a following space.
    [3] Call '<N> --> (<BREED> D O G)' : Succeeded.
      [4] Call '<BREED> ==> (%)' : Succeeded.
        [5] Generate a token 'SMALL_' without a following space.
      [4] Generate a token 'D' without a following space.
      [4] Generate a token 'O' without a following space.
      [4] Generate a token 'G' without a following space.
  [2] Call '<VP> ==> (SUDDENLY <V> <NP>)' : Failed.
  [2] Call '<VP> ==> (WITHOUT_A_BLINK <V> <NP> <PP'>)' : Succeeded.
    [3] Generate a token 'WITHOUT_A_BLINK' with a following space.
    [3] Call '<V> --> (B I T E S)' : Succeeded.
      [4] Generate a token 'B' without a following space.
      [4] Generate a token 'I' without a following space.
      [4] Generate a token 'T' without a following space.
      [4] Generate a token 'E' without a following space.
      [4] Generate a token 'S' without a following space.
    [3] Call '<NP> ==> (<DET> <N>)' : Succeeded.
      [4] Call '<DET> --> (T H E)' : Succeeded.
        [5] Generate a token 'T' without a following space.
        [5] Generate a token 'H' without a following space.
        [5] Generate a token 'E' without a following space.
      [4] Call '<N> --> (<BREED> D O G)' : Failed.
      [4] Call '<N> --> (%)' : Succeeded.
        [5] Generate a token 'GIRL' without a following space.
    [3] Call '<PP'> ==> (<P'> <NP>)' : Succeeded.
      [4] Call '<P'> --> (B Y)' : Failed.
    [3] Backtracked to '<PP'> ==> (<P'> <NP>)'.
  [2] Backtracked to '<VP> ==> (WITHOUT_A_BLINK <V> <NP> <PP'>)'.
  [2] Call '<VP> ==> (% <V> <NP> % <PP>)' : Succeeded.
    [3] Generate a token 'ALL_OF_A_SUDDEN' with a following space.
    [3] Call '<V> --> (B I T E S)' : Succeeded.
      [4] Generate a token 'B' without a following space.
      [4] Generate a token 'I' without a following space.
      [4] Generate a token 'T' without a following space.
      [4] Generate a token 'E' without a following space.
      [4] Generate a token 'S' without a following space.
    [3] Call '<NP> ==> (<DET> <N>)' : Succeeded.
      [4] Call '<DET> --> (T H E)' : Succeeded.
        [5] Generate a token 'T' without a following space.
        [5] Generate a token 'H' without a following space.
        [5] Generate a token 'E' without a following space.
```

```
      [4] Call '<N> --> (<BREED> D O G)' : Failed.
      [4] Call '<N> --> (%)' : Succeeded.
        [5] Generate a token 'GIRL' without a following space.
    [3] Generate a token 'IN_WHITE' with a following space.
    [3] Call '<PP> ==> (<P> <NP>)' : Succeeded.
      [4] Call '<P> --> (W I T H)' : Succeeded.
        [5] Generate a token 'W' without a following space.
        [5] Generate a token 'I' without a following space.
        [5] Generate a token 'T' without a following space.
        [5] Generate a token 'H' without a following space.
      [4] Call '<NP> ==> (<DET> <N>)' : Succeeded.
        [5] Call '<DET> --> (T H E)' : Succeeded.
          [6] Generate a token 'T' without a following space.
          [6] Generate a token 'H' without a following space.
          [6] Generate a token 'E' without a following space.
        [5] Call '<N> --> (<BREED> D O G)' : Failed.
        [5] Call '<N> --> (%)' : Succeeded.
          [6] Generate a token 'TEETH' without a following space.
-> TRUE: "A SMALL_DOG ALL_OF_A_SUDDEN BITES THE GIRL IN_WHITE WITH THE TEETH".
```

Listing 32: Execution trace of the generator, in the "fork" (default) mode.

An alternative mode of the generator is the “plain” mode: in this mode an entire ambiguous feature structure is used in the generation process. Below is the execution trace obtained by running the generator on the same toy grammar and the input feature structure:

```
* Running the generator with tracing (verbosity=showRule; modePlain):
[1] Call '<S> ==> (<NP> <VP>)' : Succeeded.
  [2] Call '<NP> ==> (<DET> <N>)' : Succeeded.
    [3] Call '<DET> --> (T H E)' : Failed.
    [3] Call '<DET> --> (A)' : Succeeded.
      [4] Generate a token 'A' without a following space.
    [3] Call '<N> --> (<BREED> D O G)' : Succeeded.
      [4] Call '<BREED> ==> (%)' : Succeeded.
        [5] Generate a token 'BIG_' without a following space.
      [4] Generate a token 'D' without a following space.
      [4] Generate a token 'O' without a following space.
      [4] Generate a token 'G' without a following space.
  [2] Call '<VP> ==> (SUDDENLY <V> <NP>)' : Failed.
  [2] Call '<VP> ==> (WITHOUT_A_BLINK <V> <NP> <PP'>)' : Succeeded.
    [3] Generate a token 'WITHOUT_A_BLINK' with a following space.
    [3] Call '<V> --> (B I T E S)' : Succeeded.
      [4] Generate a token 'B' without a following space.
      [4] Generate a token 'I' without a following space.
      [4] Generate a token 'T' without a following space.
      [4] Generate a token 'E' without a following space.
      [4] Generate a token 'S' without a following space.
    [3] Call '<NP> ==> (<DET> <N>)' : Succeeded.
      [4] Call '<DET> --> (T H E)' : Succeeded.
        [5] Generate a token 'T' without a following space.
        [5] Generate a token 'H' without a following space.
        [5] Generate a token 'E' without a following space.
      [4] Call '<N> --> (<BREED> D O G)' : Failed.
```

```
      [4] Call '<N> --> (%)' : Succeeded.
        [5] Generate a token 'GIRL' without a following space.
    [3] Call '<PP'> ==> (<P'> <NP>)' : Succeeded.
      [4] Call '<P'> --> (B Y)' : Failed.
    [3] Backtracked to '<PP'> ==> (<P'> <NP>)'.
  [2] Backtracked to '<VP> ==> (WITHOUT_A_BLINK <V> <NP> <PP'>)'.
  [2] Call '<VP> ==> (% <V> <NP> % <PP>)' : Succeeded.
    [3] Generate a token 'ALL_OF_A_SUDDEN' with a following space.
    [3] Call '<V> --> (B I T E S)' : Succeeded.
      [4] Generate a token 'B' without a following space.
      [4] Generate a token 'I' without a following space.
      [4] Generate a token 'T' without a following space.
      [4] Generate a token 'E' without a following space.
      [4] Generate a token 'S' without a following space.
    [3] Call '<NP> ==> (<DET> <N>)' : Succeeded.
      [4] Call '<DET> --> (T H E)' : Succeeded.
        [5] Generate a token 'T' without a following space.
        [5] Generate a token 'H' without a following space.
        [5] Generate a token 'E' without a following space.
      [4] Call '<N> --> (<BREED> D O G)' : Failed.
      [4] Call '<N> --> (%)' : Succeeded.
        [5] Generate a token 'GIRL' without a following space.
    [3] Generate a token 'IN_WHITE' with a following space.
    [3] Call '<PP> ==> (<P> <NP>)' : Succeeded.
      [4] Call '<P> --> (W I T H)' : Succeeded.
        [5] Generate a token 'W' without a following space.
        [5] Generate a token 'I' without a following space.
        [5] Generate a token 'T' without a following space.
        [5] Generate a token 'H' without a following space.
      [4] Call '<NP> ==> (<DET> <N>)' : Succeeded.
        [5] Call '<DET> --> (T H E)' : Succeeded.
          [6] Generate a token 'T' without a following space.
          [6] Generate a token 'H' without a following space.
          [6] Generate a token 'E' without a following space.
        [5] Call '<N> --> (<BREED> D O G)' : Failed.
        [5] Call '<N> --> (%)' : Succeeded.
          [6] Generate a token 'TEETH' without a following space.
-> TRUE: "A BIG_DOG ALL_OF_A_SUDDEN BITES THE GIRL IN_WHITE WITH THE TEETH".
```

Listing 33: Execution trace of the generator, in the "plain" mode.

Note there is a difference in the generation. In the fork mode the generation is "`A SMALL_DOG ALL_OF_A_SUDDEN BITES THE GIRL IN_WHITE WITH THE TEETH`", whereas in the plain mode the generation is "`A BIG_DOG ALL_OF_A_SUDDEN BITES THE GIRL IN_WHITE WITH THE TEETH`". This is caused by the ambiguous feature structure fed to the rule on line 15 in the Listing 1: `(*OR* CHIHUAHUA PITBULL)`. In the fork mode since `CHIHUAHUA` is tested first, in the `*CASE*` block the string "`SMALL_`" is chosen. Whereas in the plain mode the entire ambiguous structure is used, therefore the first choice in the `*CASE*` block wins, which produces string "`BIG_`".

References

[1] T. Mitamura and E. H. Nyberg. The KANTOO Machine Translation Environment. In *Proceedings of AMTA 2000*, 2000.

[2] P. Naur. Revised Report on the Algorithmic Language ALGOL 60. *Communications of the ACM*, 3(5):299–314, May 1960.

[3] S. M. Shieber. *An Introduction to Unification-based Approaches to Grammar.* CSLI, Stanford, 1988.

[4] M. Tomita and K. Knight. Pseudo-unification and Full-unification. Technical Report CMU-CMT-88-MEMO, Language Technologies Institute, Carnegie Mellon University, 30 November 1987.

[5] M. Tomita and E. H. Nyberg. Generation Kit and Transformation Kit Version 3.2 User's Manual. Technical Report CMU-CMT-88-MEMO, Language Technologies Institute, Carnegie Mellon University, 26 October 1988.