JAUNT: A Constraint Solver for Disjunctive Feature Structures

Hiroshi Maruyama
IBM Research, Tokyo Research Laboratory
maruyama@trl.vnet.ibm.com

June 14, 1991

Abstract

Some of the modern linguistic theories, such as LFG and unification-based grammars with disjunctions, require an NP-complete process for their analysis. Solving such hard problems efficiently is crucial to the realization of a practical system based on these theories. We have developed a constraint solver named JAUNT for solving various NP-complete problems encountered in the natural language processing field. This paper describes the design and the implementation of JAUNT. JAUNT is not only a constraint solver but a general-purpose programming language whose principal data structure is feature structures. The constraint-satisfaction algorithms employed in JAUNT, namely generalized constraint propagation and forward checking, are also described.

1 Introduction

One of the goals of the grammatical theories for natural language processing is a high descriptive power for describing various language phenomena naturally. Generally speaking, however, a high descriptive power spoils the computational efficiency of sentence analysis. For example, it is known that parsing LFG[7], parsing disjunctive unification grammar (e.g. [8]), and parsing Constraint Dependency Grammar[12] are all NP-complete problems, for which there are no known polynomial algorithms[1, 5, 11]. These NP-complete problems can be formalized as a constraint-satisfaction problem over a finite domain, and therefore, efficient algorithms for solving such problems should be employed for natural language systems based on these grammatical theories.

On the other hand, constraint-satisfaction problems over a finite domain, sometimes called *consistent-labeling problem*, have been well studied in the areas of image understanding and graph theory. *Constraint propagation*[18, 10] and *forward checking* are the examples of efficient algorithms which have been developed in these research areas.

We have developed a constraint solver named JAUNT for solving the combinatorial tasks in natural language processing, using the latest techniques of constraint satisfaction problems. JAUNT is also a general-purpose programming language for manipulating feature structures, the most commonly used data structure for natural language processing. This paper describes the design and the implementation of JAUNT. We discuss the relationships between various natural language processing tasks and constraint satisfaction problems in the next section. The design of JAUNT is given in section 3, and the constraint satisfaction algorithms adapted in JAUNT is explained in section 4. Section 5 describes the implementation details. JAUNT is currently used in two machine translation systems. Their usage are given in section 6. Section 7 concludes this paper.

2 Natural language processing and constraint satisfaction

Let $X_1, X_2, ..., X_n$ be *variables* whose value should be selected from a finite *domain* $A_i (1 \leq i \leq n)$. *Constraint* R is a predicate which determines the legal combinations of the values of the variables. Constraint satisfaction problem on a finite domain (hereafter called *consistent-labeling problem*) is a task to find the existence of a value assignment to the variables $X_1, X2, ..., X_n$[14]. For example, giving different colors to adjacent vertices of a graph is a consistent-labeling problem. It is known that solving a consistent-labeling problem is NP-complete, for which there are no known efficient (i.e. polynomial) algorithms. Let us look at some of the NP-complete problems in natural language processing in the rest of this section.

In Lexical Function Grammar[7], applications of grammar rules are controlled by the unification between F-structures. Assume that every word in an input sentence has multiple meanings. You can impose agreement constraints between the syntactic features of different words by means of unification. If you consider a word as a variable and its lexical meaning as its value, parsing LFG can be formalized as a consistent-labeling problem. Following this formalization, Berwick[1] proved that parsing LFG is an NP-complete problem.

Feature structure is one of the most popular data structure used in unification-based grammars. There are attempts to increase the descriptive power by incorporating disjunctions to feature structures (e.g. [6, 8]). Whether unification of two disjunctive feature structures succeeds or not depends on the existence of the value combinations of the disjunctions. Therefore, if we consider the disjunctions as variables, unification of disjunctive feature structure can be regarded as a consistent-labeling problem. In fact, Johnson[5] showed that unifying two disjunctive feature structures can be formalized as a satisfaction problem of the quantification-free first order logic with equality, and as a result, that it is an NP-complete problem.

Constraint Dependency Grammar[11, 12] is a grammatical formalism based on constraints between modifier-modifiee relationships. The variables are, in this case, the *modifiee* slots of each word whose value is one of the other word in the input sentence. The grammatical rules in Constraint Dependency Grammar are given as constraints between their values. Because cross-serial dependency sentences such as *respectively sentences* in English can be generated, the weak generative capacity of Constraint Dependency Grammar is strictly greater than Context Free Grammar[12], but the computational complexity of its analysis is NP-complete.

We have seen that various tasks in natural language analysis can be formalized as a consistent-labeling problem. The rest of the paper describes the design and implementation of a constraint solver that we have developed for solving such problems efficiently.

```
$board:= {
  [c={%1,2,3,4,5,6,7,8%},r=1],
  [c={%1,2,3,4,5,6,7,8%},r=2],
  [c={%1,2,3,4,5,6,7,8%},r=3],
  [c={%1,2,3,4,5,6,7,8%},r=4],
  [c={%1,2,3,4,5,6,7,8%},r=5],
  [c={%1,2,3,4,5,6,7,8%},r=6],
  [c={%1,2,3,4,5,6,7,8%},r=7],
  [c={%1,2,3,4,5,6,7,8%},r=8]
};

for X,Y in $board begin
  addc
    X?r != Y?r
        =>
    X?c != X?c  &  abs(X?r-Y?r) != abs(X?c-Y?c);
end;

find $board begin
  call print $result;
end;
```

Figure 1: 8-queen program

3 Design of JAUNT

3.1 Program example

First we give a small example of a JAUNT program. Figure 1 is an 8-queen program written in JAUNT. In the figure, `[...]` is a feature structure, $\{\ldots\}$ is a list, and $\{\%\ldots\%\}$ is a disjunction.

The first statement of the program is an assignment statement: a list of length eight is assigned to the global variable `$board`. Each element of the list is a feature structure having two attributes `c` (column) and `r` (row). The value of the attribute c is a disjunction whose value should be selected from the set of elements 1,2,..,3.

The second statement is an iteration. For every combination of elements X, Y of the list $board, the body from begin to end is executed.

There is only one statement, the addc statement, in the body of the loop. addc (*add constraint*) applies a new constraint to the disjunctions. In this case, since x and y are bound to a feature structure representing the position of a queen, the constraint can be read as: *If the rows of the queens are different (*X?r != Y?r*), their columns are different (*X?c != Y?c*) and the differences of their columns and their rows are different (*abs(X?r-Y?r) != abs(X?c-Y?c)*).* The question mark (?) is an operator for accessing a component of a feature structure.

The last statement (find ...) actually initiates a search to obtain the solutions. find searches every value combination of the disjunctions in the given argument that satisfies all the constraints simultaneously, binds it to the global variables $result, and executes the body.

3.2 Design principle

The design principles of JAUNT as a programming language are as follows:

1. Disjunctive feature structure as a basic data type
2. Full power of conventional procedural language
3. Meta-inference capability

We discuss these points in the rest of this subsection.

Disjunctive feature structure as a basic data type

As a general-purpose programming language, a constraint solver is desirable to be combined with an existing programming paradigm. For this, it is important that variables and constraints among them should be naturally represented in the programming paradigm. Notice that *variables* here are the variables in the sense of constraint satisfaction problems, and should be clearly distinguished from the variables in the conventional programming language such as Pascal whose *variables* allow destructive assignments. To avoid confusion, we call a variable in the sense of constraint satisfaction a *choice point* in the rest of the paper. A choice point represents exactly one value to be selected, although it may be not explicitly specified during the program execution. Therefore, in contrast to variables of conventional languages, the value of a choice point is never changed although its domain may be narrowed down as the execution proceeds. In this respect, *logical variables* in logic programming is similar to choice points in constraint satisfaction problems, and hence, the *Constraint Logic Programming* languages such as [2] are natural integration between constraint satisfaction and an existing programming paradigm.

A logical variable may or may not represent a choice point, but a disjunction in a feature structure represents a choice point more explicitly. Therefore, it is quite natural to build a constraint solver by regarding disjunctions as choice points. Unlike constraint logic programming, choice points in JAUNT may appear in an input data. Besides, JAUNT programmers do not need to know where and how many choice points are there in the given task. For example, when X is bound to

```
[ head="word1",
  agreement={%p1,p2%},
  subject=[ head="word2",
            agreement={%p2,p3%}
]],
```

the statement

```
addc X?agreement == X?subject?agreement;
```

imposes a constraint between the agreement features of "word1" and "word2" where when X is bound to

```
[ head="word1",
  agreement={%p1,p2%},
  subject={% [head="word2", agreement=p2],
             [head="word3", agreement=p3]
           %}
]],
```

it is a constraint between the ambiguity of the subject slot and the choice of the agreement feature of "word1". In this regard, JAUNT is contrasted to constraint solvers such as CHIP[2] and ALICE[9] in which choice points and their domains are explicitly defined in a program text.

Unification is a common method for expressing constraints on disjunctive feature structures in terms of *equality* relationships (e.g. [8]). In JAUNT, constraints are expressed in a form of logical formula which allows more powerful conditions. As Johnson[5] pointed out, any constraint expressed by unification can be restated by a first-order logic formula with equality. In addition, JAUNT allows inequality predicates (i.e. $<$, $\leq$, etc.) for integers and user-defined predicates. Using the inequality predicates, constraints on word positions can be expressed easily. For example, let X?pos be the position of the word X in a sentence and X?mod be the position of the modifiee of the word X. Then, a constraint stating *projectivity*, that is, no modification links do not cross over each other, is written as follows.

```
addc X?pos < Y?pos  &  Y?pos < X?mod  =>
     X?pos <= Y?mod  &  Y?mod <= X?mod;
addc X?mod < Y?pos  &  Y?pos < X?pos  =>
     X?mod <= Y?mod  &  Y?mod <= X?pos;
```

This kind of constraints is very difficult to represent by unification.

Flexibility of conventional procedural language

If you want to apply the no-cross-over constraint described above to every combination of the words in the input sentence, there must be a means to iterate the execution of the addc statements. In general, the applicability of constraints depends on the situation, so mechanisms to control the application are necessary. In addition, user-defined predicates may require extra-logical programming features such as destructive assignments. Moreover, it is convenient if constructing and modifying feature structures can be done within a JAUNT program.

To fulfill these requirements, we have designed JAUNT as a general purpose procedural language as well, incorporating such constructs as destructive assignment, iteration, conditional execution, and function definition and invocation. To improve the productivity of the program development, JAUNT also provides separate-compilation and macro definition/expansion.

```
define existP(X,F)
   local Dau;
begin
   if F in X?synflag then return 1;
   else
     for Dau in X?daus
       if existP(Dau,F)==1 then return 1;
   return 0;
end;
```

Figure 2: User-defined function

Since input/output of structural data (i.e. feature structure) and storage management are integrated parts of JAUNT, programmers do not have to worry about those low-level programming elements.

As an example of user-defined function, we show function `existP(X,F)` which returns 1 when these exists feature `F` in the tree rooted `X` in figure 2.

Mechanism for meta-inference

A consistent-labeling problem may or may not have a solution. If it has one, it is most probable that there are multiple solutions. In fact, if given constraints are not 'tight' enough to narrow down a few, if not one, solutions, the problem may have exponentially many solutions. The same situation is common in natural language processing. Strict grammars cause analysis failures for grammatical sentences, on the other hand, loose grammars produces combinatorial number of parse trees for certain type of sentences. To avoid such situation, grammatical constraints should be dynamically added and removed depending on the size of the solution space. In other words, a constraint solver should be provided with means to watch the inference process of its own and change the strategy depending on the observation [1].

To support the meta-inference capability, JAUNT provides

1. a mechanism to detect inconsistencies between constraints, and

[1]These features of programming language are called *reflection*.

2. functions to save and restore the state of constraint-satisfaction process.

In JAUNT, the state of constraint-satisfaction process is defined as the set of all choice points and the constraint matrices (described later) between them. Other status such as global and local variables, the program counter, and the control stack are not saved, so applications of constraints can be undone without disturbing the control flow.

Meta-inference is sometimes performed in an external module. JAUNT has inter-process communication primitives based on UNIX sockets. Using these meta-inference capabilities, an independent inference process using external knowledge can monitor and intervene a JAUNT program. If it detects an inconsistency, it instructs the JAUNT program to go back to the previous inference state and try another set of constraints; if it finds that the solution space is not small enough, it may give new constraints from its own knowledge source. By separating the meta-inference module from the object-level JAUNT program, modularity of knowledge can be achieved.

Examples of practical usage of the meta-inference capability of JAUNT are described in section 5.

4 Constraint-satisfaction algorithm

Two algorithms are used in JAUNT for constraint satisfaction. One is the *constraint propagation* algorithm[18, 14] which is activated at the time when a new constraint is added by the `addc` statements. The constraint propagation algorithm runs in a polynomial time, and it eliminates locally inconsistent values from the choice points and propagates the result to the neighboring constraints. The constraint propagation algorithm usually reduces the size of the search space significantly, but in general, it does not solve the entire problem by itself.

The other algorithm used in JAUNT is the *forward checking* algorithm which is triggered by the execution of the `find` statement. It is essentially a back-track algorithm, but it prunes unpromising branches whenever temporal choices are made, which significantly reduces the size of the remaining search space.

This section describes the constraint propagation algorithm used in JAUNT in detail. The readers are referred to [4] for the forward checking algorithm.

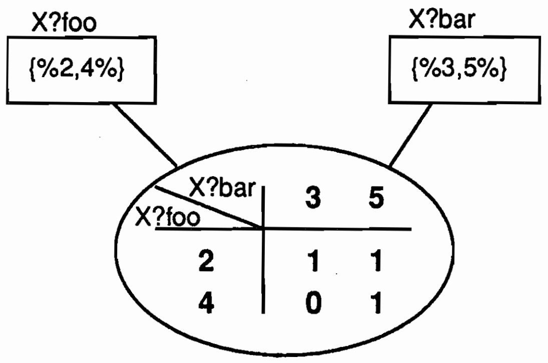

Figure 3: Constraint matrix

4.1 Internal representation of constraints

Before describing the algorithm in detail, let us explain the internal representation of the constrains. In a compiled module of a JAUNT program, a choice point is represented by a data structure called *CP*. A CP maintains a list of possible values (i.e. *domain*) at the time of the program execution. When a new constraint is added by an *addc* statement, the constraint is represented internally as a *constraint matrix*. For example, assume that

```
X := [foo={%2,4%}, bar={%3,5%}];
```

is executed. `X?foo` and `X?bar` are represented internally as CPs whose domain size is two. Then, when

```
addc X?foo < X?bar;
```

is executed, a new two-dimensional constraint matrix is created between the two CPs, as shown in figure 3.

Each dimension of the constraint matrix corresponds to a CP. The elements indicate whether the particular combination of the CP values is legal (1) or illegal (0). For example, `X?foo` = 2 and `X?bar` = 3 satisfies the constraint and hence, the corresponding element in the matrix is 1.

If another `addc` statement

```
addc X?foo * X?bar < 20;
```

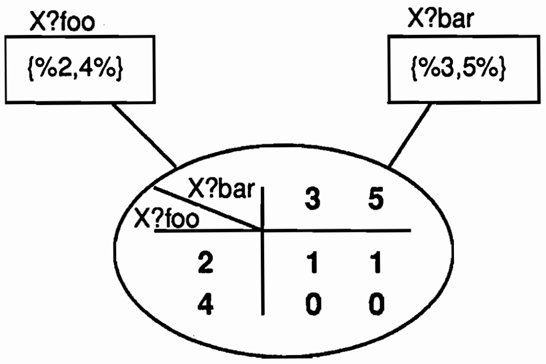

Figure 4: After `X?foo * X?bar < 20` added

is executed after that, the value combination of X?foo $= 4$ and X?bar $= 5$ does not satisfy the constraint, and the corresponding element in the matrix is changed to 0, yielding the matrix shown in figure 4.

Suppose that the execution of an addc statement referring to n different CPs $X_1, X_2, ..., X_n$ reveals that the value combination $< x_1, x_2, ..., x_n >$ is illegal. JAUNT first locates an n-dimensional constraint matrix connected to $X_1, X_2, ..., X_n$, and set its element corresponding to the value combination $< x_1, x_2, ..., x_n >$ to 0. If there is no such constraint matrix, JAUNT creates a new one whose elements are all 1, and sets the element of $< x_1, x_2, ..., x_n >$ to 0.

The size of a constraint matrix is the product the domain size of each connecting CP. Therefore, programmers should be careful not to execute an addc statement referring to many CPs when the domains of the CPs are not small enough. Such constraints should be applied as late as possible to improve the efficiency of a JAUNT program.

4.2 Constraint propagation

The basic idea of constraint propagation is to remove locally inconsistent values from the choice points and to reduce their domain size before backtracing search is performed.

In the example above, let us consider the row of X?foo $= 4$ in the constraint matrix. When X?foo $= 4$, the elements of the matrix is zero whatever

$$\begin{array}{c|ccccc}
X \setminus Y & & & & \ldots & \\
\hline
. & . & . & . & \ldots & . \\
. & . & . & . & \ldots & . \\
x_i & 0 & 0 & 0 & \ldots & 0 \\
. & . & . & . & \ldots & . \\
. & . & . & . & \ldots & .
\end{array}$$

Figure 5: Removing x_i

value `X?bar` takes. This means that there are no solutions with `X?foo` $= 4$ and therefore, this value can be safely removed from the domain of the CP `X?foo`.

In general, when a particular row or column (or plane or hyper plane, if the dimension is greater than two) contains all zero elements, corresponding value x_i of CP X can never participate in a solution (see figure 5). Therefore, x_i can be thrown away from the domain of X. Whenever a constraint matrix is updated, JAUNT searches for a hyper plane with all zero elements and removes the corresponding value from its domain. This may update other constraint matrices connected to the CP, and it may cause values in other CPs to be eliminated. Thus, updates are propagated over the network of constraints until the entire network reaches a stable state.

For every hyper plane in a constraint matrix, JAUNT keeps the current number of ones on that plane, called *support* (see figure 6). When a certain element of a constraint matrix appears to be inconsistent as a result of evaluating an `addc` statement, the corresponding support in each dimension is decremented. When a value in a CP is removed by constraint propagation, the corresponding hyper plane of every constraint matrix connected to the CP is removed, and the result is reflected to all the support values in the matrix. This algorithm is a natural extension of Mohr and Henderson's arc-consistency algorithm[15] to allow n-ary constraints.

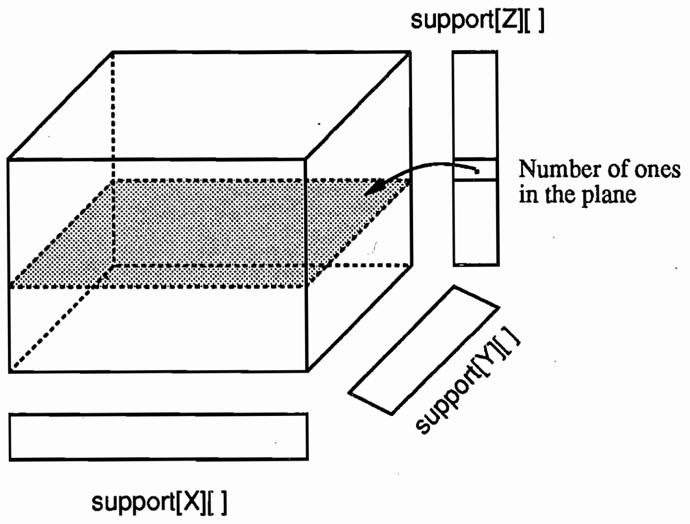

Figure 6: Support

5 Implementation

JAUNT is realized as a preprocessor to an object-oriented language called COB[17] running on UNIX. A JAUNT program is first translated into a COB program by the translator, then compiled into an object machine code by the COB compiler. The compiled object modules are linked with other COB and C libraries when necessary to form an executable code. The JAUNT preprocessor is also written in COB, and its code size is about 8,000 lines. The compiled 8-queen program shown in figure 1 generates all the solutions in 2.4 seconds on an i80386-based personal computer. In this section, we discuss the details of the compilation rules for constraints. Other program constructs such as assignment and iteration are translated into COB virtually one-to-one.

Let us consider the following statement.

```
addc X?foo < X?bar;
```

This fragment of code is translated into the following COB program.

```
{
    class Generator g = new@Generator();
    GSTART(fun_builtin_dot(lvar_X,const_28),v000) {
      GSTART(fun_builtin_dot(lvar_X,const_32),v001) {
        if (!(jc_lt(v000,v001))) g->noGood();
      } GEND;
    } GEND;
  }
```

`GSTART(X,Y)` is a macro and is expanded to a loop, which first evaluates `X` and if it is a CP, binds each value of its domain to the temporary variable `Y` and executes the body (if the value of `X` is not a CP, the body is executed only once with `X` = `Y`). The above example is, therefore, expanded to a nested loop for the values of `X?foo` and `X?bar`. The variable `g` is an object of the class `Generator` which keeps track of the temporal bindings of the referred CPs. When `g` receives a message `nogood()`, `g` locates the appropriate constraint matrix (or creates one if there is none), sets the element corresponding to the temporal value combination to 0, and activates the constraint propagation. `fun_builin_dot()` and `jc_lt()` are functions for the operations `?` and `<`, respectively. `const_28` and `const_32` are initialized to `foo` and `var` at the beginning of the program.

Notice that one execution of an `addc` statement may affect several different constraint matrices. For example, if

```
X := [agreement={p1,p2%},
      modifiee={% [agreement={%p2,p3%}],
                  [agreement={%p1,p3%}]
                %}
     ],
```

evaluation the statement

```
addc X?agreement==X?modifiee?agfreement;
```

affects two different three-dimensional constraint matrices. This is the reason why `g` locates the constraint matrix *after* it receives the `nogood()` message.

Since the number of CPs in a constraint has a large impact both on the size of the constraint matrices and the evaluation time of the constraint, a complex constraint is first transformed into a conjunctive normal form and each subformula is treated as a separate constraint. For example, the constraint

```
addc α => ( β & γ );
```

is decomposed into the following two statements:

```
addc not(α) | β;
addc not(α) | γ;
```

Although duplicate evaluation of *alpha* may be done, each subformula contains less CPs and is executed more efficiently in general.

6 Application

JAUNT is currently used in the Japanese-to-English machine translation system JETS[13] and the English-to-Japanese machine translation system Shalt2.

The system structure of the source language analysis parts of JETS is shown in figure 7. The morphological analyzer analyzes an input sentence based on a type-3 grammar and creates a feature structure representing a sequence of phrases, which contains disjunctions (choice points) for lexical ambiguities and attachment ambiguities (figure 8). The syntactic analysis program written in JAUNT applies grammatical constraints based on Constraint Dependency Grammar to these choice points and sends the result to a user-interface running on a separate machine. The ambiguous choice points (i.e. those with domain size> 1) are highlighted on the screen, and the end user can select an appropriate value for each of them. This information is sent back to the JAUNT program through the inter-process communication channel and applied as new constraints. Thus, in JETS, the end user acts as an external knowledge source to guide the inference process of the program.

Figure 9 shows the structure of the source language analysis part of Shalt2. In Shalt2, most of the syntactic analysis task is done by an augmented-context-free grammar called PEG. In general, however, since PEG does not determine the attachment of prepositional phrases, the output of PEG involves disjunctions. The constraint program applies grammatical and semantic constraints to it, and if there remains ambiguities, the external case-based inference generates appropriate constraint for resolving the remaining ambiguity[16]. Thus, clear separation of the grammatical and semantic knowledge from the case-based knowledge is achieved.

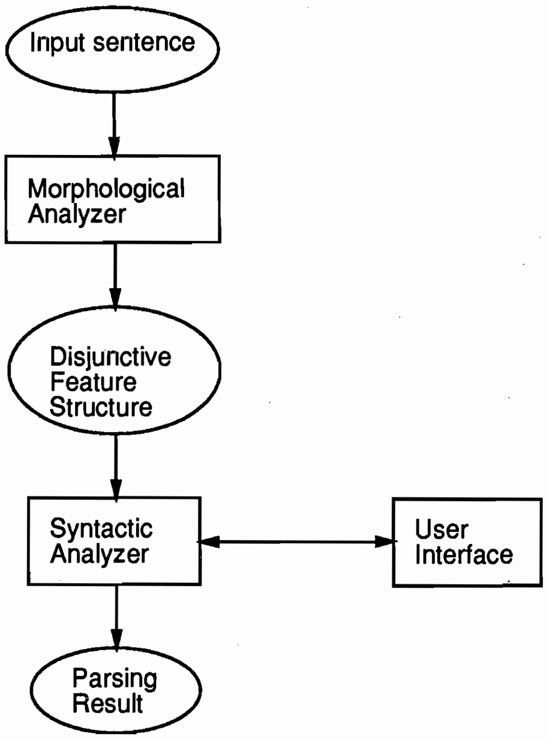

Figure 7: Analysis part of JETS

```
{ [word_id=0,
   string="ANATA",
   modifiee={%1,2,3,4%},
   lex={% [part_of_speech=pronoun, sf={hum}],
          [part_of_speech=noun, sf={loc}]
   %},
    .
    .
}
```

Figure 8: Input feature structure

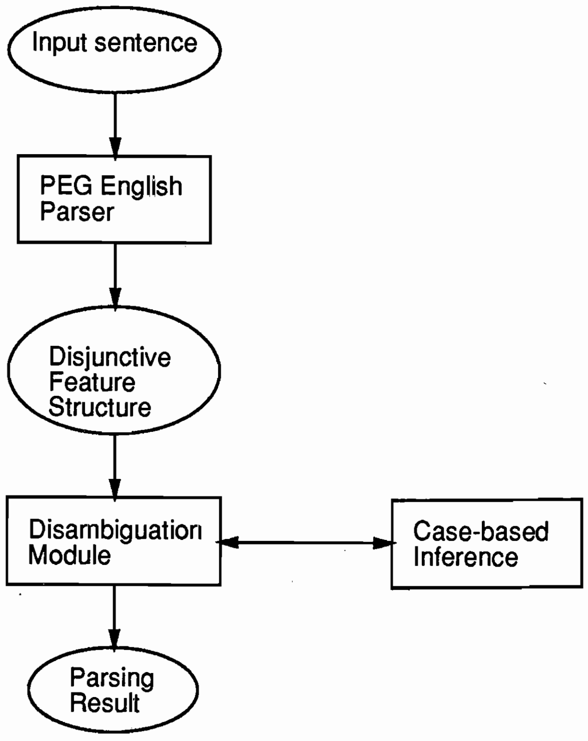

Figure 9: Analysis part of Shalt2

7 Conclusion

We have described the design and the implementation of a constraint solver whose principal data structure is disjunctive feature structures. It solves constraint-satisfaction problems on a finite domain efficiently by using the generalized constraint-propagation algorithm and the forward checking algorithm. Mechanisms for meta-inference are also provided so that different type of knowledge can be effectively combined to solve a given problem. We are planning to incorporate preferences between multiple solutions and to allow not only choice points but also certain types of constraints to be specified in the input data as well.

References

[1] Berwick, R. C., 1982, "Computational Complexity and Lexical-Functional Grammar," *American J. of Computational Linguistics,* Vol. 8, No. 3-4.

[2] Dincbus, et. al., 1988, "The Constraint Logic Programming Language CHIP," *Proc. of the International Conference on 5th Generation Computer Systems.*

[3] Haralick, M. and Gordon, L. E., 1980, "Increasing Tree Search Efficiency for Constraint Satisfaction Problems," *Artificial Intelligence,* Vol. 14.

[4] Hentenryck, P. V., 1989, *Constraint Satisfaction in Logic Programming,* MIT Press.

[5] Johnson, M., 1990, "Expressing Disjunctive and Negative Features with First-Order Logic," *Proc. of ACL Annual Meeting '90.*

[6] Karttunen, L., 1984, "Features and Values," *Proc. of COLING '84,* Stanford, CA.

[7] Kasper, R. T. and Bresnan, J., 1981, "Lexical-Functional Grammar: A Formal System for Grammatical Representation," in: J. Bresnan, ed. *The Mental Representation of Grammatical Relations,* MIT Press.

[8] Kaplan, R. T., 1987, "A Unification Method for Disjunctive Feature Descriptions," *Proc. of ACL Annual Meeting '87.*

[9] Lauriere, J. L., 1978, "A Language and a Program for Stating and Solving Combinatorial Problems," *Artificial Intelligence,* Vol. 10.

[10] Mackworth, A. K., 1977, "Consistency in Networks of Relation," *Artificial Intelligence,* Vol. 8.

[11] Maruyama, H., 1990, "Structural Disambiguation with Constraint Propagation," *Proc. of ACL Annual Meeting '90.*

[12] Maruyama, H., 1991, "Constraint Dependency Grammar and its Weak Generative Capacity," *Advances in Software Science and Technology,* to appear.

[13] Maruyama, H., Watanabe, H., and Ogino, S., 1990, "An Interactive Japanese Parser for Machine Translation," *Proc. of COLING '90,* Helsinki.

[14] Montanari, U., 1974, "Networks of Constraints: Fundamental Properties and Applications to Picture Processing," *Information Science,* Vol. 7.

[15] Mohr, R. and Henderson, T., 1986, "Arc and Path Consistency Revisited," *Artificial Intelligence,* Vol. 28.

[16] Nagao, K., 1990, "Constraints and Preferences: Integrating Grammatical and Semantic Knowledge for Structural Disambiguation," *Proc. of Pacific Rim International Conference on Artificial Intelligence,* Nagoya.

[17] Onodera, T., Kuse, K., and Kamimura, T., 1990, "Increasing Safety and Modularity of C-Based Objects," *Proc. of 3rd International conference TOOLS 3,* Sydney.

[18] Waltz, D. L., 1975, "Understanding Line Drawings of Scenes with Shadows," in: Winston, P. H. ed.: *The Psychology of computer vision,* McGraw-Hill.