THE DESIGN OF THE KERNEL ARCHITECTURE FOR THE EUROTRA* SOFTWARE

R.L. Johnson**, U.M.I.S.T., P.O. Box 88, Manchester M60 1QD, U.K.
S. Krauwer, Rijksuniversiteit, Trans 14, 3512 JK Utrecht, Holland
M.A. Rosner, ISSCO, University of Geneva, 1211 Geneve 4, Switzerland
G.B. Varile, Commission of the European Communities, P.O. Box 1907, Luxembourg

ABSTRACT

Starting from the assumption that machine translation (MT) should be based on theoretically sound grounds, we argue that, given the state of the art, the only viable solution for the designer of software tools for MT, is to provide the linguists building the MT system with a generator of highly specialized, problem oriented systems. We propose that such theory sensitive systems be generated automatically by supplying a set of definitions to a kernel software, of which we give an informal description in the paper. We give a formal functional definition of its architecture and briefly explain how a prototype system was built.

I. INTRODUCTION

A. Specialized vs generic software tools for MT

Developing the software for a specific task or class of tasks requires that one knows the structure of the tasks involved. In the case of Machine Translation (MT) this structure is not a priori known. Yet it has been envisaged in the planning of the Eurotra project that the software development takes place before a general MT theory is present. This approach has both advantages and disadvantages. It is an advantage that the presence of a software framework will provide a formal language for expressing the MT theory, either explicitly or implicitly. On the other hand this places a heavy responsibility on the shoulders of the software designers, since they will have to provide a language without knowing what this language will have to express.

There are several ways open to the software designer. One would be to create a framework that is sufficiently general to accomodate any theory. This is not very attractive, not only because this could trivially be achieved by selecting any existing programming language, but also because this would not be of any help for the people doing the linguistic work. Another, equally unattractive alternative would be to produce a very specific and specialized formalism and offer this to the linguistic community. Unfortunately there is no way to decide in a sensible way in which directions this formalism should be specialized, and hence it would be a mere accident if the device would turn out to be adequate. What is worse, the user of the formalism would spend a considerable amount of his time trying to overcome its deficiencies.

In other words, the difficulties that face the designer of such a software system is that it is the user of the system, in our case the linguist, who knows the structure of the problem domain, but is very often unable to articulate it until the language for the transfer of domain knowledge has been established. Although the provision of such a language gives the user the ability to express himself, it normally comes after fundamental decisions regarding the meaning of the language have been frozen into the system architecture. At this point, it is too late to do anything about it: the architecture will embody a certain theoretical commitment which delimits both what can be said to the system, and how the system can handle what it is told. This problem is particularly severe when there is not one user, but several, each of whom may have a different approach to the problem that in their own terms is the best one.

This requires a considerable amount of flexibility to be built into the system, not only within a specific instance of the system, but as well across instances, since it is to be expected that during the construction phase of an MT system, a wide variety of theories will be tried (and rejected) as possible candidates.

* We are grateful to the Commission of the European Communities for continuing support for the Eurotra Machine Translation project and for permission to publish this paper; and also to our colleagues in Eurotra for many interesting and stimulating discussions.

** order not significant

In order to best suit these apparently conflicting requirements we have taken the following design decisions :

1. On the one hand, the software to be designed will be oriented towards a class of abstract systems (see below) rather than one specific system. This class should be so restricted that the decisions to be taken during the linguistic development of the end user system have direct relevance to the linguistic problem domain, while powerful enough to accommodate a variety of linguistic strategies.

2. On the other hand, just specifying a class of systems would be insufficient, given our expectation that the highly experimental nature of the linguistic development phase will give rise to a vast number of experimental instantiations of the system, which should not lead to continuously creating completely new versions of the system. What is needed is a coherent set of software tools that enable the system developers to adapt the system to changes with a minimal amount of effort, i.e. a system generator.

Thus, we reject the view that the architecture should achieve this flexibility by simply evading theoretical commitment. Instead it should be capable of displaying a whole range of highly specialized behaviours, and therefore be capable of a high degree of internal reconfiguration according to externally supplied specifications. In other words we aim at a system which is theory sensitive.

In our philosophy the reconfiguration of the system should be achieved by supplying the new specifications to the system rather than to a team in charge of redesigning the system whenever new needs for the user arise. Therefore the part of the system that is visible to the linguistic user will be a system generator, rather than an instance of an MT system.

B. Computational Paradigm for MT Software

The computational paradigm we have chosen for the systems to be generated is the one of expert systems because the design of software for an MT system of the scope of Eurotra has much in common with the design of a very large expert system. In both cases successful operation relies as much on the ease with which the specialist knowledge of experts in the problem domain can be communicated to and used by the system as on the programming skill of the software designers and implementers. Typically, the designers of expert systems accommodate the need to incorporate large amounts of specialist knowledge in a flexible way by attempting to build into the system design a separation between knowledge of a domain and the way in which that knowledge is applied. The characteristic architecture of an expert system is in the form of a Production System (PS) (cf Davis & King 1977).

A programming scheme is conventionally pictured as having two aspects ("Algorithms + Data = Programs") -- (cf Wirth 1976); a production system has three : a data base, a set of rules (sometimes called 'productions' -- hence the name), and an interpreter. Input to the computation is the initial state of the data base. Rules consist, explicitly or implicitly, of two parts : a pattern and an action. Computation proceeds by progressive modifications to the data base as the interpreter searches the data base and attempts to match patterns in rules and apply the corresponding actions in the event of a successful match. The process halts either when the interpreter attempts to apply a halting action or when no more rules can be applied.

This kind of organisation is clearly attractive for knowledge-based computations. The data base can be set up to model objects in the problem domain. The rules represent small, modular items of knowledge, whose syntax can be adjusted to reflect formalisms with which expert users are familiar. And the interpreter embodies a general principle about the appropriate way to apply the expert knowledge coded into the rules. Given an appropriate problem domain, a good expert system design can make it appear as if the statement of expert knowledge is entirely declarative -- the ideal situation from the user's point of view.

A major aim in designing Eurotra has been to adapt the essential declarative spirit of production systems to the requirements of a system for large scale machine translation. The reason for adapting the architecture of classical expert systems to our special needs was that the simple production system scheme is likely to be inadequate for our purposes.

In fact, the success of a classical PS model in a given domain requires that a number of assumptions be satisfied, namely:

1. that the knowledge required can be appropriately expressed in the form of production rules;

2. that there exists a single, uniform principle for applying that knowledge;

3. finally, that the principle of application is compatible with the natural expression of such knowledge by an expert user.

In machine translation, the domain of knowledge with which we are primarily concerned is that of language. With respect to assumption (1), we think automatically of rewrite rules as being an obvious way of expressing linguistic knowledge. Some caution is necessary, however.

First of all, rewrite rules take on a number of different forms and interpretations depending on the linguistic theory with which they are associated. In the simplest case, they are merely criteria of the well-formedness of strings, and a collection of such rules is simply equivalent to a recognition device. Usually, however, they are also understood as describing pieces of tree structure, although in some cases -- phrase structure rules in particular -- no tree structure may be explicitly mentioned in the rule: a set of such rules then corresponds to some kind of transducer rather than a simple accepting automaton.

The point is that rules which look the same may mean different things according to what is implicit in the formalism. When such rules are used to drive a computation, everything which is implicit becomes the responsibility of the interpreter. This has two consequences :

a. if there are different interpretations of rules according to the task which they are supposed to perform, then we need different interpreters to interpret them, which is contrary to assumption (2); an obvious case is the same set of phrase structure rules used to drive a builder of phrase structure trees given a string as input, and to drive an integrity checker given a set of possibly well-formed trees;

b. alternatively, in some cases, information which is implicit for one type of interpreter may need to be made explicit for another, causing violation of assumption (3); an obvious case here is the fact that a phrase structure analyser can be written in terms of transductions on trees for a general rewrite interpreter, but at considerable cost in clarity and security.

Secondly, it is not evident that 'rules', in either the pattern-action or the rewrite sense, are necessarily the most appropriate representation for all linguistic description. Examples where other styles of expression may well be more fitting are in the description of morphological paradigms for highly inflected languages or the formulation of judgements of relative semantic or pragmatic acceptability.

The organisational complexity of Eurotra also poses problems for software design. Quite separate strategies for analysis and synthesis will be developed independently by language groups working in their own countries, although the results of this decentralised and distributed development will ultimately have to be combinable together into one integrated translation system. What is more, new languages or sublanguages may be added at any time, requiring new strategies and modes of description.

Finally, the Eurotra software is intended not only as the basis for a single, large MT system, but as a general purpose facility for researchers in MT and computational linguistics in general.

These extra considerations impose requirements of complexity, modularity, extensibility and transparency not commonly expected of today's expert systems.

The conclusion we have drawn from these and similar observations is that the inflexible, monolithic nature of a simple PS is far too rigid to accommodate the variety of diverse tasks involved in machine translation. The problem, however, is one of size and complexity, rather than of the basic spirit of production systems.

The above considerations have led us to adopt the principle of a controlled production system, that is a PS enhanced with a control language (Georgeff 1982). The elements of the vocabulary of a control language are names of PSs, and the well-formed strings of the language define just those sequences of PS applications which are allowed. The user supplies a control 'grammar', which, in a concise and perspicuous way, specifies the class of allowable application sequences. Our proposal for Eurotra supports an enhanced context free control language, in which names of user-defined processes act as non-terminal symbols. Since the language is context free, process definitions may refer recursively to other processes, as well as to grammars, whose names are the terminal symbols of the control language.

A grammar specifies a primitive task to be performed. Like a production system, it consists of a collection of declarative statements about the data for the task, plus details of the interpretation scheme used to apply the declarative information to the data base. Again, as in a production system, it is important that the information in the declarative part should be homogeneous, and that there should be a single method of application for the whole grammar. We depart somewhat from conventional productions system philosophy in that our commitment is to declarative expression rather than to production rules.

The device of using a control language to define the organisation of a collection of grammars provides the user with a powerful tool for simulating the procedural knowledge inherent in constructing and testing strategies, without departing radically from an essentially declarative framework.

An important feature of our design methodology is the commitment to interaction with potential users in order to delineate the class of tasks which users themselves feel to be necessary. In this way, we aim to avoid the error which has often been made in the past, of presenting users with a fixed set of selected generally on computational grounds, which they, the users, must adjust to their own requirements as best they can.

II OVERVIEW OF THE SYSTEM GENERATOR

The task of our users is to design the problem oriented machine here called "eurotra". Our contribution to this task is to provide them with a machine[1] in terms of which they can express their perception of solutions to the problem (bearing in mind also that we may need to accommodate in the future not only modifications to users's perception of the solution but also to their perception of the problem itself).

It is clearly unreasonable to expect users to express themselves directly in terms of some computer, especially given the characteristics of the conventional von Neumann computers which we can expect to be available in the immediate future. The normal strategy, which we adopt is to design a problem-oriented language which then becomes the users' interface to a special purpose virtual machine, mediated by a compiler which transforms solutions expressed in the problem-oriented language into programs which can be run directly on the appropriate computer. Functionally, can express this in the following way :

eurotra = compiler : usd[2]

where usd stands for "user solution definition". We can depict the architecture graphically as :

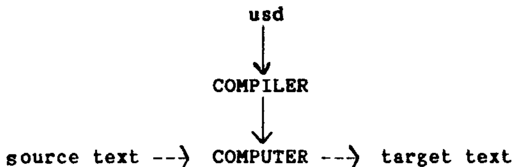

Fig. 1

In Symbols :

(compiler : usd) : text -> text

The picture above is clearly still an oversimplification. In the first place, such a compiler would be enormously difficult to write and maintain, given the tremendous complexity of the possible solution space of Machine Translation problems which the compiler is intended to represent. Secondly, especially in the light of our observation above that the users' view of the problem space itself may change, it would be very unwise to invest enormous effort in the construction of a very complex compiler which may turn out in the end to be constructed to accept solutions to the wrong class of problems.

Following well-established software engineering practice, we can compensate for this difficulty by using a compiler generator to generate appropriate compilers rather than building a complete new compiler from scratch. Apart from making the actual process of compiler construction more rapid, we observe that use of a compiler generator has important beneficial side effects. Firstly, it enables us to concentrate on the central issue of language design rather than secondary questions of compiler implementation. Secondly, if we choose a well-designed compiler generator, it turns out that the description of the user language which is input to the generator may be very close to an abstract specification of the language, and hence in an importance sense a description of the potential of the user machine.

----

1 We use the term "computer" to refer to a physical object implemented in hardware, while "machine" is the object with which a programmer communicates. The essence of the task of designing software tools is to transform a computer into a machine which corresponds as closely as possible to the terms of the problem domain of the user for whom the tools are written.

2 For the remainder of this section we shall use the notation

x : y -> z

with the informal meaning of "application of x to y yields result z", or "execution of x with input y gives output z".

After the introduction of a compiler generator the picture of our architecture looks like this (uld stands for "user language defintion"; CG stands for "compiler generator"):

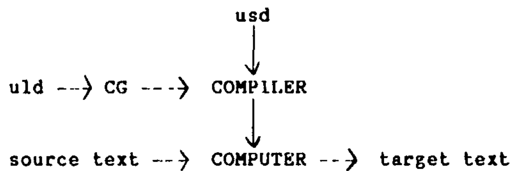

Fig. 2

In symbols :

((CG : uld) : usd) : text --> text

For many software engineering projects this might be an entirely adequate architecture to support the design of problem oriented systems. In our case, however, an architecture of this kind only offers a partial resolution of the two important issues already raised above : incomplete knowledge of the problem domain, and complexity of the semantics of any possible solution space. The use of a compiler generator certainly helps us to separate the problem of defining a good user language from that of implementing it. It also gives us the very important insight that the use of generators as design tools means that in optimal cases input to the generator and formal specification of the machine to be generated may be very close or even identical. However, we really only are addressing the question of finding an appropriate syntax in which users can formulate solutions in some problem domain; the issue of defining the semantics underlying that syntax, of stating formally what a particular solution means is still open.

We can perhaps make the point more explicitly by considering the conventional decomposition of a compiler into a parser and a code generator (cf, for example, Richards and Whitby-Strevens 1979). The function of the parser is to transform a text of a programming language into a formal object such as a parse tree which is syntactically uniform and easy to describe; this object is then transformed by the code generator into a semantically equivalent text in the language of the target machine. Within this approach, it is possible to contemplate an organisation which, in large measure, separates the manipulation of the syntax of a language from computation of its meaning. Since the syntactic manipulation of programming languages is by now well understood, we can take advantage of this separation to arrive at formal definitions of language syntax which can be used directly to generate the syntactic component of a compiler. The process of automatically computing the meaning of a program is, unfortunately much more obscure.

Our task is rendered doubly difficult by the fact that there is no obvious relation between the kind of user program we can expect to have to treat and the string of von Neumann instructions which even the most advanced semantically oriented compiler generator is likely to be tuned to produce.

We can gain some insight into a way round this difficulty by considering strategies like the one described for BCPL (Richards and Whitby-Strevens, cit). In this two-stage compiler, the input program is first translated into the language of a pseudo-machine, known as O-code. The implementer then has the choice of implementing an O-code machine directly as an interpreter or of writing a second stage compiler which translates an O-code program into an equivalent program which is runnable directly on target machine. This technique, which is relatively well established, is normally used as a means of constructing easily portable compilers, since only the second-stage intermediate code to target code translation need be changed, a job which is rendered much easier by the fact that the input language to the translation is invariant over all compilers in the family.

Clearly we cannot adopt this model directly, since O-code in order to be optimally portable is designed as the language of a generic stack-oriented von Neumann machine, and we have made the point repeatedly that von Neumann architectures are not the appropriate point of reference for the semantics of MT definitions. However, we can also see the same organisation in a different light, namely as a device for allowing us to build a compiler for languages whose semantics are not necessarily fully determined, or at least subject to change and redefinition at short notice. In other words, we want to be able to construct compilers which can compile code for a class of machines, so as to concentrate attention on finding the most appropriate member of the class for the task in hand.

We now have a system architecture in which user solutions are translated into a syntactically simple but semantically rather empty intermediate language rather than the native code of a real computer. We want to be able easily to change the behaviour of the associated virtual machine, preferably by adding or changing external definitions of its functions. We choose to represent this machine as an interpreter for a functional language; there are many reasons for this choice, in particular we observe here that such machines are characterised by a very simple evaluator which can even accept external redefinitions of itself and apply

them dynamically, if necessary; they typically have a very simple syntax - normally composed only of atoms and tuples - which is simple for a compiler to generate; and the function definitions have, in programming terms, a very tractable semantics which we can exploit in identifying an instance of an experimental implementation with a formal system definition.

With the addition of the interpreter simulating the abstract machine, our informal picture now looks like this :

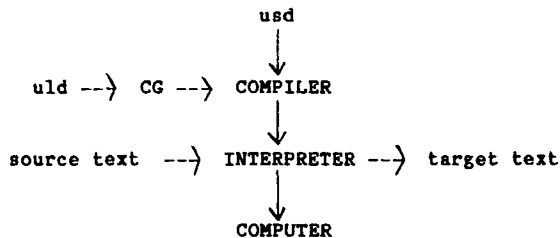

Fig. 3

or in symbols :

(INTERPRETER : ((CG:uld) : usd)) : text --> text

We now turn to the kind of definitions which we shall want to introduce into this system. We decompose the function of the machine notionally into control functions and data manipulation functions (this decomposition is important because of the great importance of pattern-directed computations in MT). Informally, in deference to the internal organisation of more conventional machines, we sometimes refer to the functionality of these two parts with the terms CPU and MMU, respectively. What we want to do is to make the "empty" kernel machine into a complete and effective computing device by the addition of a set of definitions which :

- allow the kernel interpreter to distinguish between control operations and data operations in an input language construct;
- define the complete set of control operations;
- define the domain of legal data configurations and operations on them.

With these additions, the complete architecture has the form :

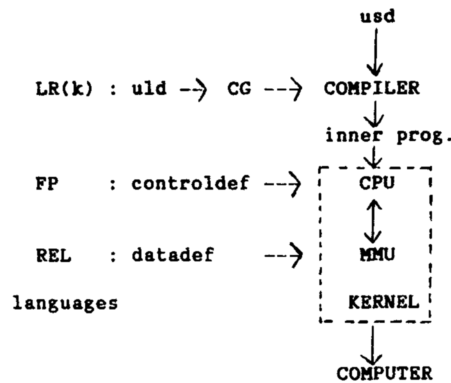

Fig. 4

or symbolically, writing "addef" for the name of the function which adds definitions:

(((addef : controldef,datadef) : KERNEL)
: ((CG : uld) : usd)) : text --> text

Capitalized symbols denote components which are part of the system generator, while lower case symbols denote definitions to generate a system instance.

An alternative way of describing Fig 4. is to see the system generator as consisting of a set of generators (languages and programs). The languages of the generator are :

a. an LR(k) language for defining the user language syntax (cf Knuth 1965);

b. a functional programming (FP) language for defining the semantics of the user supplied control (for FP cf Backus 1978);

c. a relational language (REL) for defining the semantics of user defined pattern descriptions;

d. the definition of the inner program syntax (see APPENDIX).

The programme of the system, which, supplied with the appropriate definitions, will generate system instances, are :

e. a compiler-compiler defined functionally by a. and d. in such a way that for each token of user language syntax definition and each token of user program expressed in this syntax it will generate a unique token of inner program.

f. a CPU, which is essentially an FP system, to be complemented with the definitions of point b. The CPU is responsible for interpreting the scheduling (control) parts of

the user program. It can pass control to the MMU at defined points.

g. a MMU to be complemented with the definitions of point c. The MMU is responsible for manipulating the data upon request of the CPU.

Given the above scheme, a token of a problem oriented system for processing user programs is obtained by supplying the definition of :

- the user language syntax;
- the semantics of the control descriptions;
- the semantics of the data pattern descriptions;
- the expansion of certain nonterminal symbols of the inner program syntax.

Note that a primitive (rule-)execution scheme (i.e. a grammar), is obtained recursively in the same way, modulo the modification necessary given the different meaning of the control definition.

III. FORMAL DEFINITION OF THE SYSTEM GENERATOR'S ARCHITECTURE

This section presupposes some knowledge of FP and FFP (cf. Backus cit, Williams 1982). Readers unfamiliar with these formalisms may skip this section.

We now give a formal definition of the generator's architecture by functionally defining a monitor M for the machine depicted in Fig. 4. We will do so by defining M as an FFP functional (i.e. higher order function) (cf. Backus cit, Williams cit).

An FP system has a set of functions which is fully determined by a set of primitive functions, a set of functional forms, and a set of definitions.

The main difference between FP systems and FFP systems is that in the latter objects (e.g. sequences) are used to <u>represent</u> functions, which has as a consequence that in FFP one can create new functionals. The monitor M is just the definition of one such functional.

Sequences in FFP represent functionals in the following way : there is a representation function ρ (which belongs to the representation system of FFP, not to FFP itself) which associates objects and the functions they represent.

The association between objects and functions is given by the following rule (metacomposition) :

```
(ρ <x1,...,xn>) : y =
(ρ x1) :<<x1,...,xn> ,y>
```

The formal definition of the overall architecture of the system is obtained by the following FFP definition of its monitor M :

```
ρ <M, uld, cd, dd > : usd = (ρM):<<ρM, uld, cd,
                                   dd >, usd> with :

M ≡ apply·[capply,1·[apply1·
    [apply2·[ yapply,2·1] , 2],
     apply·[↓(3·1), 'CD] ,
     apply·[↓(4·1),'DD]]]
```

where :

M is the name of the system monitor
uld is the user language definition in BNF
cd is the control definition (controldef in Fig 4.)
dd is the data definition (datadef in Fig 4.)
usd is the user solution definition

The meaning of the definition is as follows :

M is defined to be the application of capply to the internal programe ip

```
apply : <capply, ip >
```

capply is the semantic definition of the machine's CPU (see below).

ip is obtained in the following way :

```
apply1 : <apply2 : <yapply,uld >, usd >
```

Where apply2 : <yapply, uld> yields the COMPILER which is then applied to the usd.

For a definition of apply1, apply2, yapply see the section on the implementation.

```
apply· [↓(3·1), 'CD ]
```

and

```
apply· [↓(4·1), 'DD ]
```

just add definitions to the control, resp. data definition stores of the CPU and the MMU respectively.

↓ is the 'store' functional of FFP.

A. <u>Semantic Definition of the CPU</u>

As mentioned earlier, the bare CPU consists essentially of the semantic definition of an FP-type application mechanism, the set of primitive functions and functionals being the ones defined in standard FP.

The application mechanism of the CPU is called capply, and its definition is as follows :

$$
\begin{aligned}
\mu(x) \equiv\ & x \in A \rightarrow x; \\
& x = \langle x1,\ldots,xn\rangle \rightarrow \langle \mu x1,\ldots,\mu xn \rangle; \\
& x = (y{:}z) \rightarrow \\
& \quad (y \in A\ \&\ (\uparrow{:}DD) = T \rightarrow mapply{:}\langle y,z\rangle; \\
& \quad\ y \in A\ \&\ (\uparrow{:}CD) = \# \rightarrow \mu((\rho y)\ (\mu z)); \\
& \quad\ y \in A\ \&\ (\uparrow{:}CD) = w \rightarrow \mu(w{:}z); \\
& \quad\ y = \langle y1,\ldots,yn\rangle \rightarrow \mu(y1{:}\langle y,z\rangle); \\
& \quad \mu(\mu y{:}z));
\end{aligned}
$$

being the FFP semantic function defining the meaning of objects and expressions (which belongs to the descriptive system of FFP, not to FFP itself, (cf Backus cit)).

The functionality of μ is

$$\mu : \text{Expression} \rightarrow \text{Object}$$

that is, μ associates to each FFP expression an object which is its meaning. It is defined in the following way :

x is an object $\rightarrow \mu x = x$
$e = \langle e1,\ldots,en\rangle$ is an expression $\rightarrow$
$\mu e = \langle \mu e1,\ldots,\mu en\rangle$
if x,y are objects $\rightarrow \mu(x{:}y) = \mu(\rho x{:}y)$

where ρx is the function represented by the object x.

$\uparrow$ is the FFP functional 'fetch'
DD is the definition store of the MMU
CD is the definition store of the CPU
is the result of an unsuccesful search
mapply is the apply mechanism of the MMU

The execution of a primitive (i.e. a grammar) represents a recursive call to the monitor M, modulo the different function of the control interpreter (the CPU).

For the rest, as far as the user language definition is concerned things remain unchanged (remember that if appropriate,the language for expressing knowledge inside a grammar as well as the data structure can be redefined for different primitives).

The recursive call of M is caused by capply whose definition has to be augmented by inserting after line 6 of the definition given above the following condition:

$$y = applyprim \rightarrow \langle M,uld,cd,dd \rangle : x$$

where x is the specification of the primitive (e.g. the rule set).

IV. EXPERIMENTAL IMPLEMENTATION

An experimental implementation of the architecture described above has to accomodate two distinct aims. First, it must reflect the proposed functionality, which is to say, roughly, that the parts out of which it is made correspond in content, in function and interrelationship to those laid down in the design. Second, it must, when supplied with a set of definitions, generate a system instance that is both correct, and sufficiently robust to be released into the user community to serve as an experimental tool.

The entire implementation runs under, and is partly defined in terms of the Unix* operating system. The main reason for this choice is that from the start, Unix has been conceived as a functional architecture. What the user sees is externally defined, being the result of applying the Unix kernel to a shell program. Furthermore, the standard shell,or csh,itself provides us with a language which can both describe and construct a complex system, essentially by having the vocabulary and the constructs to express the decomposition of the whole into more primitive parts. We shall see some examples of this below.

Another reason for the choice of Unix is the availability of suitable, ready-made software that has turned out to be sufficient, in large measure, to construct a respectable first approximation to the system. Finally, the decentralised nature of our project demands that experimental implementations should be maximally distributable over a potentially large number of different hardware configurations. At present, Unix is the only practical choice.

A. System Components

The system consists of 4 main parts, these being :

a. A user language compiler generator.
b. A control definition generator.
c. A kernel CPU.
d. A data definition generator.

These modules, together with a user language description, a control description, and a data description, are sufficient to specify an instance of the system.

1. User Language Compiler Generator

YACC

After reviewing a number of compiler-compilers, it was decided to use YACC

----

* UNIX is a trademark of the Bell Laboratories

(Johnson 1975). Quite apart from its availability under Unix, YACC accepts an LALR(1) grammar, a development of LR(k) grammars (Knuth cit; Aho & Johnson (1974). LALR parsers (Look Ahead LR) give considerably smaller parsing tables than canonical LR tables. The reader is referred to Aho & Ullman (1977) which gives details of how to derive LALR parsing tables from LR ones.

LEX

LEX (Lesk 1975) generates lexical analysers, and is designed to be used in conjunction with YACC. LEX accepts a specification of lexical rules in the form of regular expressions. Arbitrary actions may be performed when certain strings are recognised, although in our case, the value of the token recognised is passed, and an entry in the symbol table created.

2. Control Generator

A user programe presupposes, and an inner program contains a number of control constructs for organising the scheduling of processes, and the performance of complex database manipulations. The meaning that these constructs shall have is determined by the definitions present in the control store of the kernel.

The language in which we have chosen to define such constructs is FP (Backus cit). It follows that the generator must provide compilations of these defintions in the language of the kernel machine. The implementation of the control generator is an adaptation of Baden's (1982) FP interpreter. This is a stand-alone program that essentially translates FP definitions into kernel language ones.

3. Kernel CPU

We are currently using the Unix Lisp interpreter (Foderaro & Sklower 1982) to stand in for FFP, although an efficient interpreter for the latter is under development. Notice that an FFP (or Lisp) system is necessary to implement the applicative schema described in section III, since these systems have the power to describe their own evaluation mechanisms; FP itself does not.

4. Data Definition Generator

Unfortunately, we know of no language as suitable for the description of data as FP for the description of control. The reason is that at this moment, we are insufficiently confident of the basic character of data in this domain to make any definitive claims about the nature of an ideal data description language.

We have therefore chosen to express data definitions in the precise, but over general terms of first order logic, which are then embedded with very little syntactic transformation into the database of a standard Prolog implementation (Pereira & Byrd 1982). The augmented interpreter then constitutes the MMU referred to above. The data definition for the current experiment presents the user with a database consisting of an ordered collection of trees, over which he may define arbitrary transductions.

The CPU and MMU run in parallel, and communicate with each other through a pair of Unix pipelines using a defined protocol that minimises the quantity of information passed. A small bootstrap program initialises the MMU and sets up the pipelines.

B. Constructing the System

The decomposition of a system instance into parts can be largely described within the shell language. Figure 5. below summarises the organisation using the convention that a module preceded by a colon is constructed by executing the shell commands on the next line. The runnable version of figure 4. (that contains rather more odd symbols) conforms to the input requirements of the Unix 'make' program.

```
targettext :
    ((cpu < bootstratp) < eurotra) < sourcetext
   > targettext                        /*capply*/

eurotra :
    compiler < usd > eurotra           /*apply 1*/

COMPILER :
    yacc < uld | cc > compiler         /*apply 2*/

controldef :
    fpcomp < cd > controldef

MMU :
    echo 'save(mmu)' | prolog  dd

CPU :
    echop '(dumplisp cpu)' | lisp < controldef
```

Fig. 5

V. CONCLUSION

We have argued for the need of theory-specific software for computational linguistics.

In cases where, as in MT, such a theory is not available from the beginning of a project, but rather, is expected as a result of it, we have argued for the need of a problem-oriented system generator.

We have proposed a solution by which, starting from the notion of a compiler generator driven by an external definition, one arrives at a way of building runnable, problem-oriented systems which are almost entirely externally defined. In our view, this approach has the advantage, for a domain where the class of problems to be solved is underdetermined, that the semantics of the underlying machine can be redefined rapidly in a clean and elegant way. By a careful choice of definition languages, we can use the definitions simultaneously as input to a generator for experimental prototype implementations and as the central part of a formal specification of a particular application-oriented machine.

VI REFERENCES

Aho, A.V & Johnson, S.C. (1974) - LR parsing. Computing Surveys 6 : 2

Aho, A.V. & Ullman, J.D. (1977) - Principles of Compiler Design. Addison-Wesley.

Backus, J (1978) - Can programming be liberated from the von Neumann style? Comm. ACM 21 : 8.

Baden, S. (1982) - Berkeley FP User's Manual, rev 4.1. Department of Computer Science, University of California, Berkeley.

Davis, R. & King, J.J. (1977) - An overview of production systems, in : Elcock, E.W. & Michie, D. (eds) - Machine Intelligence B: Machine representation of knowledge, Ellis Horwood.

Foderaro J.K. & Skowler K. (1982). The Franz Lisp Manual. University of California.

Georgeff, M.P. (1982) - Procedural control in production systems. Artificial Intelligence 18 : 2.

Johnson, S.C. (1975) - Yacc : Yet another Compiler-Compiler, Computing Science Technical Report No. 32, Bell Laboratories, NJ

Knuth, D.E. (1965) - On the translation of languages from left to right. Information and Control 8:6.

Lesk, M.E. (1975) - Lex : a Lexical Analyzer Generator, Computing Science Technical Report No. 39, Bell Laboratories, NJ.

Pereira & Byrd (1982) - C-Prolog, Ed CAAD, Department of Architecture, University of Edinburgh.

Richards, M & Whitby-Strevens, C. (1979) - BCPL: The language and its compiler, Cambridge University Press.

Williams, (1982) - Notes on the FP functional style of programming, in: Darlington, J., Henderson, P. and Turner, D.A. (eds), Functional programming and its applications, CUP.

Wirth, N. (1976) - Algorithms + Data Structures = Programs, Prentice Hall, Englewood Cliffs, New Jersey.

VII. APPENDIX

Below we give a BNF definition of the inner program syntax. Capitalized symbols denote non-terminal symbols, lower case symbols denote terminals.

```
PROG        ::= ⟨QUINT+⟩
QUINT       ::= ⟨NAME EXPECTN FOCUS BODY
                 GOALL⟩
NAME        ::=  IDENTIFIER
IDENTIFIER  ::=  **
EXPECTN     ::=  PAT | nil
FOCUS       ::=  VARPAIR
VARPAIR     ::= ⟨ARG  ARG⟩
VAR         ::=  VARID
VARID       ::=  **
BODY        ::= ⟨nonprim  CEXP⟩|⟨prim  PRIMSP⟩
CEXP        ::=  COMPLEX | SIMPLEX
COMPLEX     ::= ⟨CONTRLTYP  CEXP+⟩
SIMPLEX     ::=  NAME
CONTRLTYP   ::=  serial | parallel | iterate
PRIMSP      ::= ⟨RULE+⟩
RULE        ::= ⟨PAT  PAT⟩
GOALL       ::= ⟨PAT*
PAT         ::= ⟨SYMBTAB  ASSERT⟩
SYMBTAB     ::=  ARGL
ARGL        ::= ⟨ARG+⟩
ASSERT      ::= ⟨& ASSRT  ASSRT⟩|
                ⟨v ASSRT  ASSRT⟩|⟨~ASSRT⟩
ASSRT       ::=  SIMPLASSRT | ASSERT
SIMPLASSRT  ::= ⟨RELNAM  TERML⟩
RELNAM      ::=  > | < | = | ≠ |
                 IDENTIFIER |
                 prec | dom | prefix |
                 suffix | infix
TERML       ::= ⟨TERM+⟩
TERM        ::=  ARG | ⟨FUNC  TERML⟩
ARG         ::= ⟨TYP  VAR⟩ | LITERAL  null
LITERAL     ::=  **
FUNC        ::=  IDENTIFIER | length
TYP         ::=  node | tree | chain | bound
```

For each instance of the system, there is an instance of the inner program syntax which differs from the bare inner program syntax in that certain symbols are expanded differently depending on other definitions supplied to the system.

** trivial expansions omitted here.:= PAT*