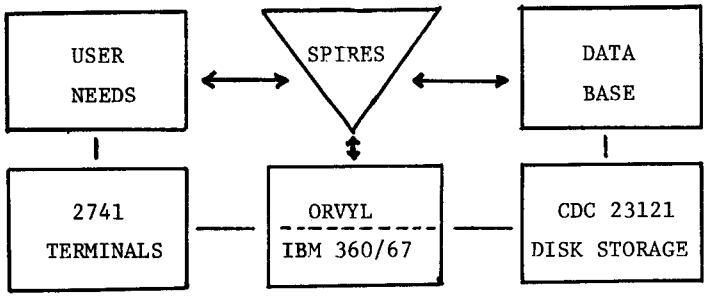
LANGUAGE DECISIONS MADE WHILE DESIGNING AN INTERACTIVE INFORMATION RETRIEVAL SYSTEM*

Thomas H. Martin and Richard L. Guertin
Stanford University

Introduction

The intent of this paper is to discuss language-related decisions made during the design of the Stanford Public Information REtrieval System (SPIRES), and to express personal opinions derived from that design experience. SPIRES II (1, 2, 3) has become a generalized interactive information storage and retrieval system. After selecting a data base, an authorized on-line user can 1) iteratively retrieve and display records from the data base, 2) insert, delete, or update records, and 3) revise the definition for the data base or enter the definition for a new data base. Conceptually, the language interface, via the computer, brings the user and data base into contact.



In 1967, when the prototype system (SPIRES I) was being designed (4, 5), the programmers thought there was just one programming language decision to make, and that their choice of PL/1 (with recourse to assembly language) settled the matter. By 1969, when the operational system (SPIRES II) was being designed, we recognized the need for at least three language decisions--a translater-writer-system for the user interface, a data definition language for the data base interface, and a programming language for the computer interface. We will discuss these three decisions below, in each case describing the functional needs, the reasons for rejecting a proposed solution, and the features of the adopted solution that weigh in its favor. In conclusion, we will draw parallels to natural language communication and summarize the SPIRES design philosophy.

The user interface

The SPIRES II target population was the Stanford academic community. We expected most users to be students, research personnel, and secretaries. Since the system was to run on the Computer Center's Campus Facility, we felt that most users would have access to IBM 2741 terminals and be conversant with Stanford's WYLBUR text editor system (6). Rather than create an entirely

*This work was supported in part by National Science Foundation Grant GN830.

new data base editing language, we chose to have SPIRES recognize all WYLBUR commands. Our overriding concern was to develop an efficient and flexible system--one that fit nicely into an existing community and could easily be changed in response to user feedback (7).

During the design of SPIRES I, the need for some formal description of the user interface was recognized, and limited parsing tables were incorporated. Unfortunately, they were an afterthought, and much of the user interface language remained buried in the PL/1 code. With SPIRES II, we sought to gather the entire user interface language together and to build the rest of the system around it. Since McKeeman, Horning, and Wortman (8) had recently developed a compiler writing language, we examined it in light of the SPIRES requirements. There are a number of reasons we decided not to use their bottom-up parsing scheme. Bottom-up parsers look for a limited class of primitive tokens, and use tables to determine whether or not arrangements of tokens are legal. When a legal combination is recognized, semantic routines are called. While it may be true of computational languages that a limited class of tokens can be arranged in many ways, it is not true of interactive information retrieval languages. With retrieval languages, an immense variety of tokens fit together in very limited ways. Each command begins with a distinctive word and is followed by a few parameter values. Therefore the most efficient parsing scheme is top-down--having recognized the command name, look for the appropriate parameter values. With bottom-up parsing, it is hard to vary the scanning rules as the context varies. In some instances we wanted hyphens to be treated like letters (e.g., school year 1972-73) and in others to be treated like special characters (e.g., 1972-73 = 1899). Finally, with bottom-up parsing, there is no way to leave gaps for filling in at run time. We wanted the data base definition to form a part of the user interface language--FIND BILL JONES should mean one thing when attached to a purchase order data base having an element named BILL, and another thing when attached to a bibliographic data base having a default search element named AUTHOR.

We chose to write the command language in a modified form of BNF, to be used in conjunction with a top-down parser (9). The right part of a BNF production is treated like a list of calls upon scan routines, semantic routines, and other productions. While the modified BNF is similar to list-processing languages, tests for determining whether or not to proceed along a list are made after returning from a call rather than before executing it. The modifications to the BNF allow for calls to be optional or required; singular or multiple (loop until failure is reported); ordinary (pass control along the right part if the call is successful, to the next right part if not successful) or lookahead (abandon the entire production if the call is successful, pass control to the next right part if not successful). Since calls upon semantics can be interspersed with syntax and the semantics can interrogate the data base definition, parsing of input can be influenced by the data base definition. Not only can languages be written that are compact and flexible, but the languages can be tested out before most of their semantic routines have been written. We have found that by writing the SPIRES user interface language in modified BNF, changes to the language can easily be viewed from the perspective of the total language, and can be incorporated without upsetting other parts of the system.

Data base interface

We suspected that a community as heterogeneous as the Stanford academic community would want to store, update, and access data bases containing widely varying types of records (i.e., bibliographic citations, survey data, full text documents, time-dependent medical records, and personnel data). A data base definition language had to allow for fixed and variable length elements, singular and multiply occurring elements, as well as elements consisting of groupings of lower level elements. We had to find a formalism in which to express recognition, validation, conversion, file inversion, and display rules for widely varying elements without either the user or staff having to program each new rule from

scratch.

During the design of SPIRES I, the need for some formal description of the data base interface language was recognized and limited tables were developed. Standard lists of data elements, index types, and exclusion words were maintained. The file manager was expected to go down the lists checking the appropriate boxes. The staff found itself constantly embroiled in revising the lists and writing code to handle special cases. For example, a user might want element B to occur in a record whenever element A did not occur, and to provide a default value for B in case the person entering the record did not. At the time we were writing the data base definition language (DBDL) there was no obvious candidate to consider first. Today we surely would consider the CODASYL data definition language (10). Since we were already using modified BNF for the user interface, we also attempted to use it for the DBDL. There were a number of reasons why it turned out to be inappropriate. A data base definition is predominantly semantic. Element names need to be unique, the links binding together elements from different files need to have similar names, parameter values need to be given values only when they deviate from default presumptions, and parameters do not need to be entered in a specific order. We found that we could almost do without syntax: within the hierarchy of a definition almost all information could be represented as parameter names followed by parameter values.

An exception to this generalization involved the processing rules for recognizing, validating, and transforming values. Many different "actions" had to be strung together in a specific order. Even in this case the syntax was not important--slashes could separate the actions and each action could be written as a code number, then a flag indicating what to do in case test conditions were not met, and finally values for up to three parameters. Once again we found the semantic aspects predominating.

The solution we adopted was to write the DBDL as if it were a data base definition itself. Information in a definition is presented in outline form with parameter names preceding parameter values. DBDL processing rules are used to make sure required parameters are given values, that values which should match values from a predefined list in fact match values from the list, and that numeric values fall within the proper limits. Since the processing of definitions is usually more complex than the processing of typical records, we use a special definition compiler to transform a validated definition into what we call the characteristics of the data base. The compiler is driven by processing rules stored in the DBDL. The decision to implement the DBDL in this manner led to many unforeseen consequences. The entire security structure for SPIRES easily fell into place. User account numbers became keys pointing to data base characteristics. Any restrictions limiting access to a file could be stored with the account numbers. Another index could contain names used for accessing data bases accompanied by text describing the contents of the data base. The major disadvantage of this approach is that definition compiling became a two-step process--first entering the definition, and then having the definition inverted into the system access files.

It should be apparent from the preceding discussion that the processing rules are the bridge connecting the data base interface to the user interface. When an element name is recognized during searching, a semantic routine calls upon the search processing rule for the element. For example, FIND (assume TOPIC) EARTHQUAKES IN CALIFORNIA is broken down by TOPIC's processing rule as follows: first call upon an action to change all commas, periods, and semicolons and colons into spaces, next call upon an action to break the value into multiple values using spaces as delimiters, then call upon an action to remove final s's, next call upon an action to remove common words from the list of values, and finally call upon an action to insert logical ANDs between the remaining values. Thus the request becomes FIND TOPIC EARTHQUAKE AND TOPIC CALIFORNIA. We have developed about two hundred different actions, with no action having more than three parameters (the final parameter may have multiple values), and feel that this set

provides us with all the flexibility needed for most definitions.

Computer interface

Since SPIRES has to share the IBM 360/67 with almost all other academic applications at Stanford, the code in core at any time must occupy the minimum amount of space possible. ORVYL (11), the Campus Facility time-sharing monitor, makes a sharp distinction between the read-only pages of subprocessor code and the read-write pages of user data. The techniques we developed for handling the user and data base interface languages made it essential that we have more flexible data structures than found in most programming languages. In particular, the programming language had to provide for superimposing structure upon data regardless of storage location.

We naturally looked to IBM's PL/1, since it had been used to implement SPIRES I. After much consideration we decided to use another language. Not only is it hard in PL/1 to separate code from data storage, but too often compiled code takes up excessive space and is slow to execute. Our interface languages required heavy use of pointers, yet too frequently we encountered restrictions limiting use of pointers. One thing we did not figure out how to encode in PL/1 was the technique of branching into semantic routines via semantic routine numbers. PL/1 also required use of OS Supervisor Calls which ORVYL does not allow.

We eventually decided to use the programming language PL360 developed by Niklaus Wirth (12). While the original PL360 was not intended for very large programs, by the time we needed it, it had been extended to handle them. PL360 is an ALGOL-like language that uses a bottom-up parser to translate statements directly into IBM 360 machine instructions. The programmer has complete control over index registers. Pages of code (csects) and pages of data (dsects) are treated as distinct units referred to by different index registers. Conceptually a procedure contains one csect, any number of dsects, and any number of other procedures. If the main csect is the top of the hierarchy, a level n csect can refer to procedures and dsects at level n+1 or at any higher level. Procedures and dsects can easily be moved around, and new levels can be added either at the top or at the bottom of the hierarchy. The net result of shifting code around is that heavily used general-purpose routines eventually shift to the top levels, leaving special-purpose routines in the lower levels. Thus PL360 in a virtual memory environment leads to minimum shifting of pages from drum to core.

The branching from the modified BNF to semantics is easily handled by first checking the range the number falls into, branching to a lower-level csect, and then using an address list to translate the number into an address within the csect. We have found that coding in PL360 has the advantages of efficient execution and rapid compilation, as well as the legibility and encodability of ALGOL or PL/1. One drawback of PL360 is that interfacing with programs written in FORTRAN IV or PL/1 is tricky (but rarely necessary). Overall, we have found that our decision to use PL360 made SPIRES an economically viable system.

Overview

In retrospect, we feel that it was proper to take three different approaches to the language decision. The resulting system has a unity of design because each system component was developed with the other components in mind. The command language permits semantics to be mixed with syntax, the data base definition language permits processing rules to be intermixed with declaratory information, and the programming language permits data to be kept distinct from code.

While the user interface language is not "natural", it has much in common with more natural interaction languages. Woods (13) notes that natural language translation requires that calls upon semantics be intermixed with syntax. He discovered a small set of semantic actions which process values extracted by the syntax. Winograd (14) discovered parsing of natural

language requires that some aspects of the data be encoded as procedures so that the context of discourse can influence syntactic recognition. While our notations differ, each of us has discovered that human communicators use what they know about each other, the subject being discussed, the environment, and the rules of the language to reduce complexity. Ambiguity is of little importance because either party can ask the other for clarification. Good interactive system design, Corbató (15) suggests, should seek to pare away possibilities until the essential components of the system are revealed in their simplicity. By keeping the functional needs of the Stanford community in mind, by splitting the problem into three subproblems, by considering alternate solutions to the problems, and by designing from the top down, we have sought to achieve good design.

REFERENCES

1. SPIRES Staff. Requirements for SPIRES II. Stanford University, Stanford, California, April 1971. (ED 048 747)
2. Parker, Edwin B. SPIRES 1970-71 Annual Report. Stanford University, Stanford, California, 1971.
3. SPIRES Staff. SPIRES User's Manual. Stanford Computation Center Campus Facility, Stanford, California, October 1972.
4. Parker, Edwin B. SPIRES 1967 Annual Report. Stanford University, Stanford, California, December 1967. (ED 617 294)
5. Parker, Edwin B. SPIRES 1968 Annual Report. Stanford University, Stanford, California, January 1968. (PB 184 960)
6. Fajman, R. and Borgelt, J. "WYLBUR: An Interactive Text Editing and Remote Job Entry System," CACM, 16:5 (May 1973), pp. 314-322.
7. Martin, Thomas H.; Parker, Edwin B. "Designing for user acceptance of an interactive bibliographic search facility." In Walker, Donald E., ed. Interactive Bibliographic Search: The User/Computer Interface. AFIPS Press, Montvale, New Jersey, 1971, pp. 42-52.
8. McKeeman, W.M.; Horning, J.J.: Wortman, D.B. A Compiler Generator. Prentice Hall, Englewood Cliffs, New Jersey, 1970.
9. Martin, Thomas H. "Action Controlled Translation: A New Approach to BNF." In SPIRES Staff, General Design Document for SPIRES II, Stanford University, August 1971, pp. A-134 through A-152.
10. CODASYL Data Base Task Group. April 1971 Report. ACM, New York, April 1971.
11. Fajman, R. and Borgelt, J. ORVYL User's Guide. Campus Facility, Stanford University Computer Center, Stanford, California, 1971.
12. Wirth, Niklaus. "PL360, A Programming Language for the 360 Computers," JACM, 15:1 (January 1968), pp. 37-74.
13. Woods, W.A. "Transition Network Grammars for Natural Language Analysis," CACM, 13:10 (October 1970), pp. 591-606.
14. Winograd, T. Procedures as a Representation for Data in a Computer Program for Understanding Natural Language, M.I.T., MAC-84, 1971.
15. Corbató, F.J.; Saltzer, J.H.; Clingen, C.T. "Multics--The First Seven Years." In AFIPS Spring Joint Computer Conference 1972 Proceedings. Vol. 40, pp. 571-583, 1972.

APPENDIX

Samples of a modified BNF command language grammar, a PL360 procedure, and a file definition are presented on the following three pages. If the search request

FIND TITLE CALIF# EARTHQUAKE

were parsed using the sample grammar, semantic processes would be called in the following sequence:

<1> FIND <60> TITLE <62> CALIF# <73> EARTHQUAKE <73> <71> <4> <61> <79>.

A Sample Command Language Grammar (written in Modified BNF).

```
|COMMAND LANGUAGE| ::= (0,MASTER LANGUAGE) <LOGOFF>
<MASTER LANGUAGE> ::= <SEARCH>
                      |LOGOFF|
                      <EXTRA COMMANDS>
<LOGOFF> ::= <1> LOG(OFF) (SP) <4>
<OFF> ::= OFF
<EXTRA COMMANDS> ::= <1> SELECT <SP> FILE (SP) <4> <50>
<SEARCH> ::= <1> FIND <SP> <60> (SRCH COMMAND) <79>
<SRCH COMMAND> ::= (SRCH-1) <4> <61>
<SRCH-1> ::= ( (SP) <63> (2,SRCH-1)
             (SRCH-MNEMONIC) <SRCH-VALUE> (2,SRCH-2)
<SRCH-2> ::= ) (SP) <66> (2,SRCH-2)
             <LOGICAL OP> (2,SRCH-1)
<SRCH-MNEMONIC> ::= <CHARACTERS> <62> (SP)
<SRCH-VALUE> ::= <VALUE> (0,VALUE) <71>
<VALUE> ::= |TERMINATOR|
            <CHARACTERS> <73> (SP)
<TERMINATOR> ::= )
                 <LOGICAL OP>
<LOGICAL OP> ::= AND <SP> <67> (NOT)
                 OR <SP> <68>
<NOT> ::= NOT <SP> <69>
(CHARACTERS) ::= 0,1,0,40,()
(SP) ::= 0,1,1,40
```

Explanations for Selected Terms in Right Parts of the Sample Productions.

FIND	Required call on the character string scanner for FIND.
<60>	Required call on semantic process 60.
<CHARACTERS>	Required call on production CHARACTERS.
(SP)	Optional call on production SP.
(0,VALUE)	Repeat call on production VALUE until failure.
(2,SRCH-2)	Transfer of control to production SRCH-2.
|TERMINATOR|	Lookahead call on production TERMINATOR. If TERMINATOR succeeds then VALUE fails.
0,1,0,40,()	Scan for (0=no maximum) 1 or more characters not (0=not) like blanks (hexidecimal 40) or parentheses.

The semantic processes would carry out the following operations:

```
<1>   Read a user's command.
<60>  Initialize for a "FIND" command, assign core, etc.
<62>  Look up the value parsed by <SRCH-MNEMONIC>
       in the file's search characteristics.  If the
       value does not match an access name and no
       default access name is in effect, the semantic
       process reports failure to the Parser.
<73>  Save the parsed value for later processing.
<71>  Apply Search Processing Rules (if any) to the values.
       Store the final values in the search command stack.
       Logical operators are also stored in the stack.
<4>   Insure that all the command has been parsed.
<61>  Perform the required search based on the information
       saved in the command stack.
<79>  Clean up, release core assigned by <60>.
```

A Sample Search Procedure (written in PL360).

```
GLOBAL PROCEDURE SEARCH (R14);
BEGIN  COMMENT -- PROCEDURES 'DOSEARCH', 'BADPARENS',
            'INITSEARCH', AND 'ABORT' ARE NOT SHOWN --;

 DUMMY BASE R8;  COMMENT -- SEARCH DSECT --;
    SHORT INTEGER PARENDEPTH, MNEMONICTYPE;
    ARRAY 30 INTEGER COMMANDSTACK;
    BYTE OPERATOR, RETURNSWITCH;
    BYTE FINDFLAG, ERRORFLAG;
 CLOSE BASE;
 EQUATE ERRORCODE SYN 100,  OROP SYN 1,
        ANDOP SYN 2, ANDNOTOP SYN 3;

COMMENT -- SAMPLE SEMANTIC BRANCHING TECHNIQUE --;
   R7 := R7 - 60 SHLL 2;
   R8 := DSECTADDRESS;
   BALR(R3,R0);  BRANCH(B3(R7+4));
      GOTO SEM60;
      GOTO SEM61;
      GOTO SEM62;

COMMENT -- SAMPLE SEMANTIC PROCESSES --;

SEM63: COMMENT -- <63> UPDATE PARENTHESIS DEPTH --;
   R2 := 1 + PARENDEPTH;  PARENDEPTH := R2;  GOTO EXIT;
SEM62: COMMENT -- <62> CHECK <CHARACTERS> AGAINST THE FILE'S
       SEARCH CHARACTERISTICS TO DETERMINE IF IT IS
       A VALID SEARCH MNEMONIC --;  GOTO EXIT;
SEM61: COMMENT -- <61> PROCESS PARSED SEARCH COMMAND --;
   IF ZERO ¬= PARENDEPTH THEN BADPARENS;
   IF OPERATOR = 0 AND ZERO = MNEMONICTYPE
      AND ¬ERRORFLAG THEN DOSEARCH ELSE
   BEGIN  R1 := ERRORCODE;  ABORT;
   END;  GOTO EXIT;
SEM60: COMMENT -- <60> SETUP FOR FIND COMMAND --;
   INITSEARCH;  SET(FINDFLAG);
   OPERATOR := OROP;
EXIT: R13 := R13 -- #01000040;  LM(R1,R0,R13);
END;
```

When semantic process <71> calls upon the search processing rule for TITLE, the following operations would be carried out:

A45, Break the value into multiple values using blanks as delimeters. ANDs are inserted between values.

A14,# If a value ends with a pound sign, it is removed and a switch is set indicating that all pointers for access records beginning with this stem should be ORed together.

A Sample File Definition (with Explanatory Notes in the Right Column).

```
FILE  X123.SAMPLE;                  The name of the FILE.
RECORD  BOOK;                       The main RECORDs of the FILE
  KEY  BOOK-NUMBER;                   are keyed on BOOK number
  ELEM  TITLE;                        with a TITLE (which is restricted
    OCC  1;                           to occur only once),
  ELEM  AUTHOR;                       and multiple AUTHORs.
RECORD  AUTHOR;                     The AUTHOR index records are
  KEY  LAST-NAME;                     keyed on the AUTHOR's last name.
  ELEM  NAMES;
    TYPE  STR;
STRUCTURE  NAMES;                   The AUTHOR's first name is the key
  KEY  FIRST-NAME;                    of a multiply occurring structure
  ELEM  POINTER;                      containing POINTERs back to the
    TYPE  PTR;                        BOOK records.
RECORD  TITLE;                      The TITLE index records are
  KEY  WORD;                          keyed on individual TITLE words
  ELEM  POINTER;                      with POINTERs back to the
    TYPE  PTR;                        BOOK records.

LINKAGE  BOOK;                      The BOOK records are "linked" to
  REFERENCE  POINTER;                 the indexes by the POINTERs.
    NAME  BOOK;
      PASSPROC  A170;
  ACCESS  AUTHOR;                   The AUTHOR records are used to
    NAME  AUTHOR, A;                  access the BOOKs when a search
      PASSER  AUTHOR;                 mnemonic refers to AUTHOR or A.
      PASSPROC  A166/ A38;          The file inversion and search
      SRCPROC  A38/ A14,#;            processing rules allow
    IN  NAMES;                        truncated surname searching.
      NAME  NULL;                   The author's first names are handled
        PASSPROC  A165;               by the structure portion of the
    RETRIEVAL  POINTER;               AUTHOR index records.
  ACCESS  TITLE;                    The TITLE records are used to
    NAME  TITLE, T;                   access the BOOKs when a search
      PASSER  TITLE;                  mnemonic refers to TITLE or T.
      PASSPROC  A166/ A45,;         The PASSPROC and SRCPROC rules
      SRCPROC  A45,/ A14,#;           allow truncated (word stem)
    RETRIEVAL  POINTER;               searching.
SUBFILE  BOOKS;                     The BOOK records constitute the
  GOAL  BOOK;                         primary (or GOAL) records,
  ACCT  PUBF;                         and are publicly accessible.
```

QUESTIONS

Harry Lotis:

Were there quite a bit of similarities that caused you to reach a decision to break the data base into three parts? I am just wondering how you actually separated it?

Guertin:

What I said was that the system is broken into three parts, not the data base. What I mean by that is we have a data base definition in which we declare how these data elements are to interact and be handled. The code for handling that is inside the SPIRES system in these 150 semantic processes which can be combined together, depending on the particular data base. The first split is between the code and the data base. The second split is between the code and the user through the command language. The command language services the user with the information in the data base, and the programming language in between only accomplishes the work. The translation between what the user wants and what the file contains is done by the BNF on one side and the file definition on the other. The result is a three phrase structure: command language, semantic processes and file definition.

Robert A. Landau:

Can you give us some insights into the power of the system in terms of the number of simultaneous ports available and the size of the data base in millions of characters?

Guertin:

The size of the data base differs depending on which one you are using. We have complete structural depth capability, i.e., a record can contain a data element, which is a structure which contains data elements, and some of those can be structures which contain data elements. What was that first question again?

Robert A. Landau:

You didn't quite answer the second. I am interested in knowing the number of simultaneous users that can access the system and the largest possible data base that the system can handle. I would not consider 100,000 records, for instance, a very large data base.

Guertin:

SPIRES is a twin of another system called BALLOTS. The BALLOTS and SPIRES are separate processors that share the same computer. SPIRES has an average of about 40 sessions per day, ranging in length from 15 minute to a half hour. We have tried loading SPIRES with 50 terminals, and there doesn't seem to be any degradation other than the normal degradation you would expect from having 50 people trying to access the same data base simultaneously, i.e., disc tie-ups, channel interference. With the paging system such as the IBM 370 uses, the load is really not that high since SPIRES is contained in 25 pages of 4,000 bytes each, to which is appended approximately three pages of user specific information.

Martin Dillon: Is there a HELP for data definition?

Guertin: That is being worked on right now.

Martin Dillon:

Suppose you do not want to go to the trouble of doing your own file definition, and you want to use one of the 75 that already exists? Is that possible?

Guertin:

Oh yes, if the file owner wishes to make his file public or semi-public, he needs to change only one thing in the file definition.

Thomas Martin:

I would like to add that while it takes only about four hours to define a file, it certainly should be understood that people are never quite satisfied with the first crack at file definition and it is possible to revise things quite a bit before one needs to scratch all that he has done previously. Also, in terms of the file structure that we use, Knuth would call it an n-way binary tree. It has the property of never forcing you to give up and rebuild the entire structure from scratch.

Richard E. Nance:

In terms of the BNF definition of the command language, how much input came from the user community in defining the command language?

Guertin:

A SPIRES I system existed prior to the current system, and the experience with that taught us what was needed. At least so we thought. After we had begun our simple set of commands for SPIRES II, we found that the user community was not satisfied with it. We began to effect changes in the command language based on the user needs, and we periodically review the command language by monitoring the user's session at the terminal (if he so desires, since he is informed that our monitoring is being done). We do not render existing commands inoperable, but we seek to modify them to develop alternate paths.

Martin:

Let me also respond to that. I think this comes out most clearly when you implement a command and find that it just doesn't work the way you thought it should. We had such an example in the update area Actually, we have done this on a number of commands.

Martin Dillon:

How many people know what PL 360 is? (The answer is over half of the audience.) Good, I have always wondered why computer companies didn't come out with more languages of the PL 360 type instead of assembly languages. It seems to me that what assembler is to octal, PL 360 is to assembler. I just don't understand why it hasn't been accepted as a realized advance over assemblers.

Guertin:

We wanted to use PL 360 because of its portability, at least among IBM 360 installations. We found that it took at least 10 times the amount of time in assembler as it did with PL 360. Also with the PL 360, we did not have to operate under OS.

Gerard Salton:

I have a question, since you brought up PL 360, and I don't feel that you really rationalize this in the paper. In fact, there seems to be a contradiction somehow, for you say, when you discuss the design of SPIRES II:

> We examined a compiler writing language in light of the SPIRES requirements...unfortunately that language is a bottom-up parser and bottom-up parsers have certain well known problems.

And then a few pages later you go on to say,

> We use PL 360, which is an ALGOL-like language with the bottom-up parser, that has statements corresponding directly to IBM 360 machine instructions.

And my question is either you like PL 360 or you don't, which is it? Surely the reason you use PL 360 is not because of the parser but for something else.

Martin:

The problem is that we were thinking about two different things there. Bottom-up parsing does not work well at all in the interactive command language, but it works beautifully in the implementation of the semantics.

Guertin:

Recall that the system exists in three phases: the command language, the semantics, and the file definition. The bottom-up parser does not work well at all for the command language, but it does work quite well for a compiler and similar uses, such as our semantic processes.

Nagib A. Badre:

My impression is that you can do anything with bottom-up parsing that you can do with top-down parsing.

Thomas L. Martin:

It's the same sort of business if you are thinking of top-down parsing like you are thinking of transition networks. We were using the augmented transition networks. In the work that Woods has done, there is quite a difference between transition networks. What we couldn't do with bottom-up parsing was this blend of spreading out the information about the data base between the command language, the semantic processes and the file definition. The top-down parsing approach allowed us to blend these things in a very nice way.

Guertin:

Top-down gave us the capability to place calls among semantic processes wherever we desired; conversely, bottom-up parsing says it limits you to doing semantic processing only after you have finished production.

Unidentified Questioner:

Could you amplify on how you do the processing between the command language and the rest of the system?

Guertin:

If you look at the appendix to our paper, you will see that the command language is shown as a BNF language. Imbedded in that are angled brackets with numbers. The numbers are indications of semantic calls, i.e., calls on the semantic processors for work to be done. A PL 360 program is used to drive down the BNF in order to parse the user's query. A semantics routine is then called. A branch is made to the proper semantics processor from this routine based on what the parsing of the user's command has produced. Information gleaned from the data base and placed in core is then examined to see whether the semantics will succeed or fail. The semantics routine will then return control to the parser indicating whether it has been able to succeed in its task or something is wrong. Failure causes the attempt to try alternate routes in the BNF. This process can be repeated several times until finally a complete indication of task completion is given by the semantics routine, and at that time a search is effected.

Robert Gaskill:

I am quite interested in the tools that are used in the implementation systems such as SPIRES. I believe you mentioned that it took the two of you something like two years. You further indicated that the basic reasons were that you used a high level language, PL 360, and then you gave a couple of others. I submit, at least for your comment, that maybe you left out one of the most important reasons, and that is that there were only two of you.

Martin:

There were also some additional contributing factors and they were the availability of all the other things at Stanford. These included an efficient, capable text editor, the various on-line debugging techniques, etc. Also, to be honest, the second person is actually a third person who is not here. There were other people on the staff who contributed also, but perhaps 90% of the work was done by these two people.

Guertin:

By the way, I have broken down for you the division by language of the code for SPIRES II. Note that there are 2,000 bytes of assembly language, 4,000 bytes in BNF, and the remainder, 86,000 bytes, in PL 360.

Bill Carlson:

What do you do to help users keep bad data out of their data bases? Are people expected to edit a sequential file before they enter it into your system?

Martin:

Once again in terms of our input processing rules, the user can construct his added records at a terminal and then we can submit those to all the validation tests the user has specified before entering them in his data base. Acceptance of the record actually means its entrance into what we call an overnight queue. The record is actually entered at this point, but the updating of the inverted files is done only at night, so that the inverted file version of the record will be available only after the updating at night.

Guertin:

This process of deferred updating means that we can do the updating for a series of records at one time rather than having to do each one individually.

Bill Carlson:

And the semantic structures for this editing and validation are contained within your 150. Still, you do not find that you need to assist users in doing this?

Martin:

I think it's fair to say that he who is defining his file had better go through an individual who really understands the file definition process. It is not a simple process.

Leo Bellew:

I am still confused as to the processing between the semantic routines and the data base. Could you clarify this?

Martin:

(The following is a severely edited version of Martin's reply.) The user's query stimulates a series of calls upon semantic processes that eventually terminate with a reference to the file definition. Within the file definition are contained those terms that are present in an inverted file. This process then returns to the semantic routine with the information as to whether the terms constructed in the parsing of the query are contained in the inverted file. If there, the result is a success; if not, a failure.

Guertin:

There is a multi-level here in that what you see in the paper is coded PL 360. One of the semantic processes that can be called contains rules for search procedures. These rules may differ depending on particular data bases. All the rules are also coded in PL 360, but the user does not see that from the BNF level. Note that the data base contains not only the data but also the rules as to how the data can be used and how different data elements are related. Does this answer the question?

Unidentified Questioner:

The question is why are you doing that? Are you actually going out to the data base while you are doing all those things?

Martin:

The answer is "No".

Unidentified Questioner:

(This question related to the file definition usage in the SPIRES system. Unfortunately the questioner did not use a microphone, so it was impossible to determine the exact wording of his question.)

Martin:

When a file is selected, brought into the user's area is a portion of the file that is related to file definition. This portion of the file definition includes both needed information for searching and for file updating. These tables created by the file definition portion are then called upon by the semantic processing routines. Imbedded in these tables are the rules to which we have referred several times.

Guertin:

When you have completed the parsing, a search routine physically accesses the data base. Consequently, the data base is physically accessed only when the parsing is completed, but file definition information, stored in core, is accessed during the parsing process.

Bernard Plagman:

How does your system handle the relations between more than one data base, and can the system establish work areas for treating subfiles within two or more data bases?

Martin:

We have not done that. It can be done only by working with one data base and storing the results as a temporary file, then working with the second data base and storing its results as a temporary file before trying to bring the two together in any sense.

Bernard Plagman:

Then the responsibility would be with the user to keep account of what is happening in his work area?

Martin:

That is correct.

Unidentified Questioner:

Does the appendix to the paper show actually how the file definition procedures work?

Martin:

Yes. That is the intent of the appendix. In an attempt at clarification, think of both the BNF and these processing rules and as strings of calls on small programs resulting in an interpretive type system.

Guertin:

The whole system, too, is recursive in itself. That is, the BNF analyzer has BNF that drives it, and you can analyze the BNF to get a new BNF analyzer. The PL 360 compiler is written in PL 360 so that you can change the compiler in its own language. And the file definitions are written as a file which is a file definitions file. Consequently, the whole system is totally recursive and can be boot-strapped at any point. The system is portable in every extent except that small part which is written in the assembly language. This represents the interface with the operating system and consequently would differ among implementations.

Bill Malthouse:

What is the interface between PL 360 and IBM supplied macros? For example, when advances are made in the IBM macros, can you get at these easily through PL 360?

Guertin:

The decision was made not to use any macros in PL 360. This decision was prompted by the desire to remain independent of any operating system.

Bill Malthouse:

Then PL 360 provides in itself some sort of macro capability?

Guertin:

It will. The University of Alberta has just developed a macro capability in PL 360. I received a tape last week, but have not had the opportunity to examine it. Let me say that I do not think that the absence of the macro capability inhibited our efforts at all.

Martin:

Perhaps an addendum is necessary at this point, since we should note that Dick Guertin had complete control of the PL 360 compiler during our efforts. If he found something in the compiler that he didn't like, he simply changed the compiler.

Willard Draisin:

Perhaps you should mention some of the weaknesses in the system, for I think that they would be as useful to us as the strengths that have been emphasized.

Martin:

I could provide some of these. There is such a great deal of flexibility in the data definition capability that you get widely differently appearing files. On the other hand, all of the messages are at the system level so that it becomes quite difficult for a file manager to communicate with his users about problems specific to a particular file.

Guertin:

We need file associated assistance and diagnostic routines. These HELP and error diagnostics should be particular to a data base.

Martin:

Because there is so much flexibility in the data definition area, a user has to rely on one or two individuals who can keep many of the details in mind because they are constantly creating files for other users. A user cannot at present define a file from scratch.

Guertin:

This is something that we hope to remedy with our automatic file definition compiler.

Steven R. Roman:

Could you comment on restrictions on the length of the record or the size of the file?

Guertin:

There are no limitations on the size of the data elements. We have fixed length area of records, and we have required record areas whose entries may be multiple or single and may be fixed or variable in length. Then there are optional elements that may or may not occur. Two bytes are used to process any data element that is a fixed length or fixed occurring. As we have said before, a data element may be a structure, and a structure may have fixed or variable occurring or length characteristics. You can go 10 deep in structures, so that there is little overhead required by the system. The last statistics I saw on the MARC data base indicated that 80% of the file was being used for actual data.

Martin:

I don't believe we have answers to some of these questions. Under ORVYL there can be only nine data sets so that we could have at one time only seven indexes or inverted files. We circumvent this restriction by defining what we call a combined index for the type of element that occurs rarely but you still wish to use in searching. In that case we sort of create a mini-index that is a record in and of itself. There is a separate record in that combined index for every type of data element that you will be treating in that way. The size of the record is no longer limited by the size of the data set, and as a practical matter the size of the file in limited by the number of disks we have at Stanford.

Leo Bellew:

How difficult would it be to enter a foreign file, perhaps contained on 200 tapes?

Martin:

We would begin by writing a BNF description of what the file looked like. As a practical matter, we have not had any gigantic files, in the neighborhood of millions of records, built on the system at this time. I am sure that files of this size would cause problems in the way storage is maintained in the current Stanford system.

Alan Beals:

I understand from your discussion that you enter the data record in an updating context at the time that the data record is defined, but the indexes are not changed until a later point in time. Since the only way one can get at the data record is through the indexes, why bother to add it at all at that time?

Guertin:

The reason we do not do on-line updating is that, for example, changing the title of a record might

mean unindexing a lot of different index terms and indexing on a lot of new ones. Perhaps 17 or 20 new index records could be affected by changing this particular data record. This produces a wider window of time during which, if the system fails, we will have an incomplete updating process. Also, we found that a user in updating a record may make several changes in a short period of time, for example, a day. Thus we shall defer updating until the record is, in the user's opinion, of the form that he wishes. Also, we make a comparison of the updated record with the original and change only those records that have been altered.

Martin:

The stimulus for this procedure with regard to updating was the desire for a very high system reliability by the BALLOTS people. They felt that high reliability was a prime characteristic.

Guertin:

By the way, the "overnight" updating process can be done interactively on-line by those users who wish to run that risk.

Alan Beals:

There seems to me that in some instances and in some applications a very gross error could be made by referencing a data element whose key value has been changed but that change has not been made. For example, consider the case in a bank where an account balance although changed has not been noted.

Martin:

Yes, I agree there are applications where our procedures are not suitable. We have just not been dealing with that dynamic an environment.