John F. Sowa

Interactive Language Implementation System

The Interactive Language Implementation System (ILIS) is a tool for implementing language processors. It is fast enough for conventional compilers and general enough for processing natural languages. ILIS is built around a language for writing grammars. Unlike most compiler-compilers, the language includes a full range of semantic operators that reduce or eliminate the need for invoking other programming languages during a translation. ILIS is also highly interactive: It has facilities for tracing a parse and for adding or deleting grammar rules dynamically. This paper describes the features of ILIS and its use in several different projects.

Processing languages with ILIS

The Interactive Language Implementation System (ILIS) is a tool for *rapid prototyping.* Using it, a programmer can quickly implement a language or interface and test its human factors with actual users. If some feature proves hard to understand or awkward to use, the designer can change the syntax in a few minutes and test it again.

ILIS can support any language for which a grammar can be written: a command facility, a macro processor, an application driver, a help facility, a conversion aid, a checker and formatter for programming standards, or a dialog handler for computer-aided instruction. As a parser, ILIS is equivalent to an augmented transition net [1]; for computation, its semantic operators are as general as a Turing machine. ILIS can also be used for teaching grammar theory; it is simple enough that students can begin to use it after a one-hour introduction.

Natural languages like English are more difficult to handle than programming languages. Compilers use syntax alone in parsing programming languages. But when people use language, they take advantage of their background knowledge about the subject matter. Unrestricted natural language requires a richer semantic, pragmatic, and deductive component than ILIS supports [2]. Yet restricted subsets of natural language have been implemented in ILIS: an English query language for a database system and a Japanese front-end to some English-oriented systems. By restricting the vocabulary and range of topics, these systems make the language more manageable.

This paper describes the use of ILIS in several projects. It then proceeds with a systematic description of the features of ILIS and their use in language processing. Finally, it discusses the design considerations that led to the ILIS forms and their implications for efficiency and generality.

Applications of ILIS

The original version of ILIS was designed as a language handler for invoking application programs. As it evolved, new features and operators were added to make it a general language processor. Within IBM, ILIS has been used as an experimental tool for implementing several prototype systems:

- *Natural language parsing* The ILIS parser normally runs top-down with backtracking. It also manipulates stacks and registers that make it similar to an augmented transition net, but with a notation of grammar rules instead of networks. Since it is implemented in optimized PL/I, it runs faster than most parsers implemented in LISP. With the interactive editing and debugging tools of ILIS, Handel [3] wrote a grammar for an interesting subset of English in about a month.
- *Microcode assembler* The SWARD project at the IBM Systems Research Institute involved designing a software-

directed machine architecture [4]. Hocker [5] used ILIS to write an assembler that generated the special microcode used in SWARD. The entire assembler was written in ILIS except for a final PL/I routine that mapped ILIS character strings into bit strings.

- *RENDEZVOUS query project* At the IBM Research Laboratory at San Jose, California, Codd et al. [6] designed an Engish query facility for a relational database. Their initial implementation used a pattern matcher written in APL that was too slow for practical use. Since their pattern rules were nearly a subset of the ILIS facilities, the designers were able to map them into ILIS. As a result, the CPU time for parsing a typical query was reduced from 10 seconds to 0.16 second; the generation time for a series of responses was reduced from 37 seconds to 0.26 second. Although most ILIS parsers run top-down, RENDEZVOUS used the associative pattern matching of ILIS to implement a bottom-up parser.
- *Japanese front-ends* At the IBM Tokyo Scientific Center, ILIS was used to support Japanese-style inputs and outputs to English-oriented systems, including STAIRS/VS, PASCAL, and PSL/PSA. Since ILIS imposes no restrictions on the character set, it accepts Japanese-style input in kanji characters, translates it into an English-style command, and passes it to the back-end system as though it had been typed directly in that form. When the system generates a response, ILIS intercepts it, translates it into Japanese, and sends it to the output handler in kanji form. Of these efforts, the most complete was the JISDOS project in which a front end to PSL/PSA was developed [7]. In these projects, ILIS served as a more general translator than a simple table of synonyms: It had to rearrange the word order, recognize Japanese particles that had no equivalents in English, and generate correct syntactic forms in the target languages. Because of the limited domain of discourse, however, the conversion was far easier than full translation of unrestricted natural language.
- *General syntax checker* To check a file while it is being edited, Lawrence Margolis wrote an interface to ILIS from one of the standard editors. When the user hits a program function key, the checker notes what type of file is being edited, searches for a table of grammar rules for that type, invokes ILIS to parse the file, and displays error messages in the lower half of the screen keyed to line numbers in the upper half. Then the user can correct the errors and recheck the file without leaving the editing environment. Margolis mapped a grammar for Pascal into ILIS form in less than twenty minutes; writing error messages and inserting them in the rules at the appropriate points took about two days. More complex languages take more time: Writing a syntax checker for Ada took two weeks; an analyzer for COBOL took about a month. Once a syntax checker has been written, other features can easily be added: a formatter for standardizing the indentation; a preprocessor for expanding synonyms and macros; or a programming aid that generates code skeletons for procedure headers, loops, and other coding sequences.
- *Teaching aid* Since 1976, ILIS has been used for teaching courses on compiler design and natural language processing. Feedback from the students helped to refine the system and make it more general, more flexible, and easier to use. After a short course (18 hours in the classroom), students were able to implement simple language processors: a compiler from a subset of BASIC to System/370 assembler, a compiler from a PL/I-like language into assembler for a microprocessor, and a syntax checker for the SQL query language. Other students implemented various games and novelties, including a guide to Japanese restaurants in New York and an analyzer for barbershop quartet harmony.

Comparisons with other systems

ILIS belongs to the general class of translator writing systems. It also has a lot in common with macro processors and pattern matching languages like SNOBOL. Basically, a macro processor is a system for replacing one string of symbols with another. Such systems are useful for adding synonyms and abbreviations to a language, but they usually have a limited scope. Besides replacing strings, ILIS can do the following:

- When given the grammar of an input language, ILIS can do a complete parse.
- While parsing, it can call external programs to do I/O, perform computations, or run some other system.
- It can dynamically create new rules and add them to its internal tables.
- It is general enough to support parsers for English and other natural languages.

Like the ILIS parser, the SNOBOL pattern matcher is a top-down parser with backtracking, but its pattern matching is limited to character strings. ILIS, however, matches lists of words, each with its own string of indicator bits. The difference is partly a matter of efficiency: ILIS can run faster because it has fewer units to process. But the most important difference is the structure of the data. Each ILIS word is a unit with associated information, such as indicator bits that specify the syntax and a type field that specifies the semantic category.

ILIS, like many translator writing systems, separates the parsing phase into an initial scan that chops the string into a list of words and a secondary stage that applies grammar rules to the list. For a survey of the field, see [8]. But ILIS has the following combination of features that is rarely found in other translator writing systems:

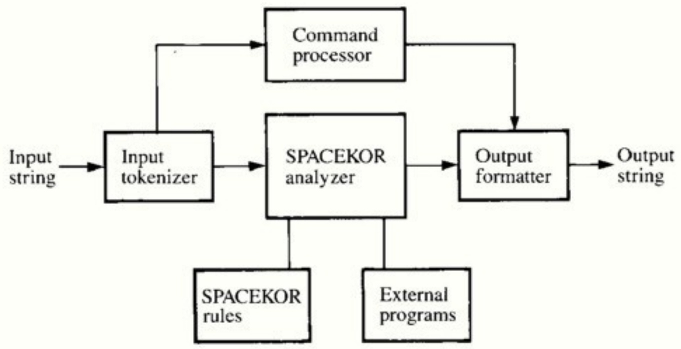

Figure 1 ILIS system components.

- An interactive mode that permits new rules to be defined and used immediately,
- A simple syntax for rules that allows rules to treat other rules as data and dynamically update the workspace,
- A trace mode for displaying every operator as it is executed,
- The option of using top-down, bottom-up, or mixed parsing strategies for any source language,
- The efficiency of deterministic parsing for programming languages and the power of an augmented transition net for natural languages,
- Techniques for handling context-sensitive and transformational grammars by testing and setting indicators, registers, and stacks,
- The option of having multiple sets of grammar rules for different languages and sublanguages and switching from one to the other automatically,
- An output formatter that formats output according to the conventions of the target language.

System components

At the heart of the ILIS system is the rule interpreter called SPACEKOR. This name is an acronym for the first eight operators used to define the syntax and the semantics of a language. When a user enters an input statement, SPACEKOR rules analyze it, manipulate stacks and registers to generate a translation, and invoke external procedures to perform additional services. Figure 1 shows how the SPACEKOR analyzer is related to the other system components.

The six components illustrated in Fig. 1 make up the ILIS system for processing languages and driving application programs:

1. The *input tokenizer* takes a character string and breaks it down into a list of separate words or tokens.
2. A table of *SPACEKOR rules* defines the grammar of an input language and the associated semantic operations.
3. The *SPACEKOR analyzer* uses the rules to parse the input language and execute the semantic operators.
4. *External programs* do computations and perform services that are more convenient to write in a conventional programming language, such as PL/I.
5. A *command processor* interprets system commands for loading and saving workspaces, for displaying information about the system, and for entering, editing, and debugging SPACEKOR rules.
6. An *output formatter* types a list at the terminal or writes it to a file under the control of format commands, such as *indent* and *tab*.

Of these components, four are a basic part of ILIS itself: input tokenizer, SPACEKOR analyzer, output formatter, and command processor. These need not change from one application to the next. The table of SPACEKOR rules and the external programs, however, are tailored for each application. The SPACEKOR rules define the input language, and the programs are called by the SPACEKOR analyzer to do the actual processing.

The external programs may include simple functions like TIME or DATE as well as large application packages. For the Japanese front-ends mentioned earlier, systems as large as STAIRS/VS or PSL/PSA run as external programs under ILIS. Since ILIS is implemented in PL/I, it can directly call other PL/I programs. Applications written in COBOL, FORTRAN, or PASCAL may be called from an interface routine written in PL/I. An interface to the IBM CMS Operating System commands or EXEC routines is provided by the procedure SYSTEM. Although ILIS is designed as an integrated system, it is completely modular, and various pieces can be incorporated separately in other packages. For a particular application, a programmer might use the complete system while debugging and testing SPACEKOR rules. For the finished product, however, everything but the SPACEKOR analyzer and the final set of rules could be stripped out of ILIS and linked with the other package.

Data formats

ILIS has only one primitive data type, the *word*, which has seven subfields: a pointer NEXT, which points to the next word in a list or to *nil* if it is the last word; two integers LINE# and COL#, which show the original position of the word in the input stream; an integer TYPE, which represents the type of concept or data associated with the word; an integer LEN, which specifies the word length; a character string LETTERS, which represents the printed form of the word; and a bit string INDICATORS, which may be set or tested by the INDICATOR operator. In the current implementation, a word is a block of storage defined by the following PL/I declaration:

```
DECLARE 1 WORD BASED,
  2 NEXT POINTER,                /* To next word      */
  2 LINE# FIXED BINARY(15),      /* Line number       */
  2 COL# FIXED BINARY(15),       /* Column number     */
  2 TYPE FIXED BINARY(15),       /* Concept type      */
  2 LEN FIXED BINARY(15),        /* Word length       */
  2 LETTERS CHARACTER(12),       /* Printable form    */
  2 INDICATORS BIT(64);          /* Indicator bits    */
```

The LETTERS field is of length 12, but words may be up to 255 characters long. If a word is longer than 12, it is continued in as many additional blocks as needed. The implementation details, however, are completely invisible to the ILIS programmer. The operators test and set various fields, but the programmer never needs to know the internal formats.

The only lists that the SPACEKOR analyzer processes are lists of words. A rule is a list of words, some of which may be operator strings. For efficiency, the SPACEKOR compiler converts rules to an internal form for pattern matching, but a rule in its external form can be constructed and manipulated by the standard list handling techniques. The empty list *nil* can be considered as either a list of no words or a rule with no operators. The SPACEKOR stacks and registers are treated as lists of lists of words. No deeper nesting of lists is possible.

SPACEKOR operators

The SPACEKOR operators fall into three major categories: *pattern-matching operators* that match something in the input stream; *action operators* that do some computation; and *control operators* that define a loop, a list of options, or an invocation of some other rule. Some of the operators have several variants to make a total of 50 different operators or variants. The appendix to this paper lists all of them; this section briefly illustrates a few of them.

An example of a pattern-matching operator is $A*2. The symbol $ is the escape character that starts every string of operators. The letter A names the ANY operator, and *2 is a qualifier that specifies a length of exactly two words. During pattern matching, the operator $A*2 would match any list of two words. The KEEP operator $K is an example of an action operator. It takes a list that has just been matched and keeps it on a stack for further processing. The sequence

```
$A*2 $K
```

would match two words from the input stream and keep them on the stack. An example of a control operator is $O for OPTION. The sequence

```
$O CAT | DOG | FLOWER O$
```

matches any one of the three words "CAT," "DOG," or "FLOWER." The vertical bar separates options, and the symbol O$ terminates the option list. Any word that is not part of an operator string is assumed to be a literal that is matched to the input stream.

Certain combinations of operators occur so frequently that they become standard idioms. The string $AEK, which is equivalent to the sequence $A $E $K, starts with the ANY operator $A to match any list of zero or more words, matches the END of the input stream with $E, and KEEPS the list on the stack with $K. The SPACEKOR programmer can just think of $AEK as a primitive operator that takes the remainder of the input stream and puts it on the stack. Another example is the string $RS in the sequence,

```
$AK DON'T $RS DO NOT S$
```

First, $A matches any list up to but not including the word "DON'T," and $K keeps it on the stack. Then the word "DON'T" is matched, and the REJECT operator $R throws it away. The STACK operator $S then places the list "DO NOT" on the stack; the symbol S$ terminates the list started by $S. To the programmer, the combination $RS may be considered as a substitution operator that throws away one list and replaces it with another.

SPACEKOR rules

A SPACEKOR rule is any sequence of SPACEKOR operators and literal words. It may be invoked in either of two ways: *associatively*, when the pattern part of the rule matches something in the input stream, or *explicitly*, when it is called from some other rule by a MATCH or PERFORM operator. The following rule would normally be invoked associatively:

```
RULE (TIME): $A TIME $A ? $C=TIME.
```

The keyword "RULE" indicates the start of a rule. In parentheses is the *index word* "TIME," which will cause this rule to be invoked if a matching word occurs in the input stream. Since the ANY operator $A matches any list of words, the pattern part matches inputs like "What time is it?" or just "time?" When the pattern part successfully matches the input stream, the CALL operator $C=TIME calls the external program named TIME to get the current time of day.

Before a rule can be called explicitly, it must have a name. The following rule is named S; it uses the MATCH operator $M to call two other rules, named NP and VP:

```
RULE S: $M=NP $M=VP.
```

Words in parentheses after the keyword "RULE" are index words to the rule. An alphanumeric word that is not in parentheses is the name of the rule. This rule named S may be part of a grammar for recognizing English sentences. It first calls a noun phrase rule NP to match the beginning of a sentence. Then it calls a verb phrase rule VP to match the remainder.

A rule can have both a name and a list of index words. The following rule can be called explicitly by the MATCH operator $M=LOGOFF or the PERFORM operator $P=LOGOFF. It can also be invoked associatively when any of its four index words (BYE GOODBYE LOGOFF LOGOUT) appears in the input stream:

```
RULE LOGOFF (BYE GOODBYE LOGOFF
        LOGOUT): $S CP LOGOFF S$ $C=SYSTEM.
```

Since the pattern part of this rule is empty, it will succeed no matter how it is invoked. The STACK operator $S puts the list "CP LOGOFF" on the stack. Then the CALL operator $C passes that list to the external program named SYSTEM. Under IBM's VM/370, the effect is to terminate the session.

Representing grammar rules

Context-free grammar rules map directly to SPACEKOR rules. Terminal symbols in the grammar map to literals, and nonterminal symbols map to the MATCH operator $M. The following grammar rule defines an expression EXPR as a TERM followed by zero or more repetitions of the terminal symbol "+" followed by another TERM:

EXPR → TERM ["+" TERM]. . .

In this notation, square brackets represent an optional constituent; three dots, one or more repetitions; and brackets followed by three dots, zero or more repetitions. In SPACEKOR, the LOOP operator $L executes a string of operators zero or more times as long as the pattern matching succeeds. The above grammar rule becomes the SPACEKOR rule,

```
RULE EXPR:  $M=TERM $L + $M=TERM L$.
```

This rule can do the parsing, but it has no semantics. To specify some action to be taken or result to be generated, action operators are inserted among the pattern matching operators.

As the input language is parsed, values generated by one rule are left on the SPACEKOR stack to be processed by other rules. When a rule is invoked by the MATCH operator, values passed to it are called *inherited attributes*; values returned by a rule are called *synthesized attributes*. In an interpreter for arithmetic expressions, the synthesized values are the numbers that result from the computation. When the TERM rule leaves a synthesized number on the stack, the EXPR rule that called it uses the external program DYADIC to add it to the previous result:

```
RULE EXPR: $M=TERM $L + $K $M=TERM $S*3
           $C=DYADIC L$.
```

Inside the loop, the literal "+" matches a plus sign, and $K keeps it on the stack. The following $M=TERM operator evaluates another term and leaves the resulting number on the stack. Then the last three lists contain a number, the word "+", and another number. The STACK operator $S*3 concatenates the three lists of one word each into a single list of three words. That list is then passed to DYADIC by the CALL operator, and DYADIC returns the sum as result. That number remains on the stack, and each iteration of the loop adds another number to it until all occurrences of "+" followed by a term have been processed.

SPACEKOR rules permit any regular expression on the right-hand side of a rule. The next grammar rule defines a noun phrase NP as an optional determiner DET, an optional string of adjectives ADJ, a required NOUN, and an optional prepositional phrase PP, participial phrase PARTP, or relative clause RELC:

NP → [DET] [ADJ] . . . NOUN [PP|PARTP|RELC]

In SPACEKOR notation, that rule becomes

```
RULE NP: $O $M=DET|O$ $L $M=ADJ L$
         $M=NOUN
         $O $M=PP|$M=PARTP|$M=RELC|O$.
```

To represent an optional constituent, the OPTION operator $O permits an empty option represented as just a blank.

Context-sensitive conditions are handled by operations on registers, stacks, and symbol tables. The simplest context-sensitive language is the one consisting of an arbitrary number of A's followed by exactly the same number of B's and the same number of C's. That language can be parsed in several different ways, each of which illustrates an interesting SPACEKOR technique.

The first method is to use a counter on the stack. The following rule named AB calls itself recursively to match pairs of A and B. For each pair, it calls the external program ADD1 to increment a number left at the end of the stack:

```
RULE AB:   A $O $M=AB|O$ B $C=ADD1.
```

This rule is called by the following rule, whose index word is "A." After it initializes the stack to zero with $S 0 S$, it calls the AB rule:

```
RULE (A):   $S 0 S$ $M=AB
            $L C $C=SUB1 L$ $EP 0 P$.
```

The $L loop matches a letter C, calls SUB1 to decrement the counter, and repeats. Then $E checks for the end of input, and the PERFORM operator $P 0 P$ matches 0 to the counter on the stack.

Another method for parsing the ABC language is to build a list of C's for each AB pair and then compare that list with the remaining input. The following version of the AB rule builds the first list of C's on the stack:

```
RULE AB:   A $O $M=AB|O$ B $S C S$ $S*2.
```

The STACK operator $S C S$ puts one C on the stack for every pair of A and B; the concatenate variant $S*2 concatenates the new C on the stack to the previous list of C's. The following rule is triggered by the index word "A." It initializes the stack to *nil* with $S S$ before it calls the AB rule:

```
RULE (A):   $S S$ $M=AB $RAEKH%C $V=C.
```

The REJECT operator $R causes the list of A's and B's matched by the AB rule to be ignored in further processing. Then $A matches any list up to the end $E and keeps it on the stack with $K. At this point, the last two lists on the stack should contain strings of C's. To check that they are equal, $H%C HOLDS the last list in a register named C and pops the stack. Finally, $V=C VERIFIES that the last list on the stack is now equal to the list in register C.

To illustrate the ease of creating and using rules dynamically, the next rule shows another method of matching the two strings of C's. It first creates a special rule that matches exactly the correct number of C's:

```
RULE (A): $S RULE C: S$ $M=AB $S . S$ $S*3
            $C=MATCH $M=C $E.
```

This rule creates a list of the form

```
RULE C: C C C C C C C C,
```

where the list of C's to be matched was left on the stack by the call to AB with $M=AB. Then $C=MATCH invokes the SPACEKOR analyzer recursively to enter this list as a new rule in the current workspace. Finally, $M=C calls that rule to match the remaining list of C's in the input stream, and the operator $E tests for the end of input. All three of these techniques parse the ABC language in linear time. The rule that uses a counter is slightly faster, but the other two techniques can be generalized to more complex languages.

Production system

Production systems are frequently used for computations in artificial intelligence [9]. Each rule in a production system has a pattern part and an action part; if the pattern matches something in working storage, the action is performed. When SPACEKOR rules are triggered associatively, they can be used as a production system. The input stream is the working data, and the action operators put their changes on the stack. Then the rescan variant of the PERFORM operator $P=* concatenates the modified list on the stack with anything left in the input stream and treats the result as new input.

As with all production systems, ILIS must have some convention for determining which rule to try when multiple patterns match the same data. Consider the next two rules:

```
Rule (TIME): $A CPU TIME $A ? $C=CPUTIME.
Rule (?): $A ? $S WHY DO YOU ASK? S$.
```

The first rule will match a sentence like "How much CPU time have I used?" and then call the procedure CPUTIME; the second rule will match any sentence containing "?" and generate the response "WHY DO YOU ASK?". Since both rules are capable of matching the same sentence, the SPACEKOR analyzer must adopt some strategy for resolving the conflict. To determine which rule to invoke, it starts at the beginning of the input stream, looks up each word in its index, and takes the following actions:

- If the word is not in the index, then the SPACEKOR analyzer skips it and goes on to the next word.
- If the word is in the index, the analyzer invokes the associated rule.
- If the rule matches, the analyzer sets a flag to indicate success and returns whatever is left on the stack as the result.
- If the rule fails to match, the analyzer checks whether there is another rule with the same index word and invokes it; when two rules have the same index word, the more recently defined one is tried first.
- If all rules associated with the index word fail, the analyzer takes the next word of input as a possible index.
- If the end of the input is reached before any rule matches, the analyzer returns the input stream unchanged and sets a flag to indicate failure.

When the SPACEKOR analyzer gets the input "What's the CPU time?" it reaches the index word "CPU" before it reaches the index word "?". Therefore, the first rule is invoked to give the CPU time. If the input stream were "How fast is the CPU?" then the rule with index word "CPU" would be invoked first. But since that rule does not match the input, it causes a failure, and the SPACEKOR analyzer goes on to the index word "?". The rule with that index word then succeeds and generates the output "WHY DO YOU ASK?".

Stacks and registers

For working storage, the SPACEKOR stack is a list of lists that is manipulated by the action operators. As a rule is processed, the stack works in a LIFO order: The last list put on the stack is the first one used. The CALL operator $C passes the last list to external programs and replaces it with the result they return. When a SPACEKOR rule r is called by the MATCH operator $M=r, the lists left on the stack are available to operators in r. The lack of explicit parameters for SPACEKOR rules makes calling and backtracking more efficient since the amount of saving and restoring is minimized.

All operators except PERFORM continue to use the same stack. The PERFORM operator $P saves the current stack and creates a new stack that is initially empty; when PERFORM finishes, the contents of the new stack replace the last list of the previous stack. A recursive call on the SPACEKOR analyzer by the operator $C=MATCH also creates a new stack.

Some variants of the STACK operator manipulate lists that are deeper in the stack. The operator $S*n concatenates the last n lists into a single list; the head of each list is concatenated to the end of the preceding list. The operator $S%n interchanges the last list with the nth list from the end, and $S¬n deletes the nth list.

During processing, the stack operates in a LIFO fashion. At the end of processing, however, all lists are concatenated into a single list in a FIFO fashion—the oldest one at the head. This convention is good for most language processing. As rules parse the input stream from left to right, they build up the translated results on the stack. The most frequent processing takes place at the end of the stack, but the result should be read from the beginning.

The stack may be compared to human short-term memory. For most processing, it rarely has more than seven lists, although each list may be arbitrarily long. For global long-term memory, there is a set of named registers. Each register contains a list of words, which like the stack can be processed in either a LIFO or FIFO fashion with the GET and HOLD operators. The HOLD operator $H transfers the last list on the stack to a register, and GET $G transfers a register to the end of the stack. For dictionaries and symbol tables, the call $C=DEFINE saves a list indexed by its first word, and $C=LOOKUP retrieves a previous list.

Backtracking

The SPACEKOR analyzer does automatic backtracking: Whenever an operator or word match fails, the analyzer backs up to a previous option, called a *choice point.* For a deterministic grammar, backtracking is not necessary during the parsing phase, but it may be needed when generating diagnostics for a syntax error. Even programming languages like PL/I may require backtracking to handle forms that cannot be parsed with an LL(1) grammar. Typical problems arise in PL/I when the compiler cannot tell whether an identifier is a keyword or the name of a variable:

```
IF (X+Y) = 15.7 THEN GO TO HOME;
IF (X+Y) = 15.7;
```

In the first statement, the keyword IF introduces a conditional. The second statement is an assignment to the vector IF indexed by X+Y. The parser, however, cannot tell how to interpret the first word "IF" until it reaches the ninth word "THEN" or ";".

A potential source of inefficiency in backtracking is the need to undo changes that had been made to the environment by operators that lay along a failing path. To minimize the effort lost, SPACEKOR does not do a perfect restoration: It throws away new additions to the stack by operators along a failing path, but it does not undo all side effects or changes to lists that were put on the stack before the previous choice point. In most cases, this minimal restoration is sufficient. In other cases, the programmer can postpone nonrestorable operators until the critical pattern matches have been completed.

When syntax analysis has reached a certain stage, the programmer may decide to commit the analyzer to the current path and eliminate all possibility of backup to pending, untaken options. To do that, the UNOPTION operator $U clears out the list of choice points. Most systems that do backtracking have a similar facility for stopping the backtracking: SNOBOL has the FENCE operator, PROLOG has the CUT operator, and MICROPLANNER has FINALIZE.

Types and indicator bits

Dictionaries represent two kinds of information about words: syntactic features that specify parts of speech, and semantic types that specify the associated concepts. Programming languages make a similar distinction between the syntactic categories like operator or variable, and the semantic categories called data types. Some systems represent the syntactic features with binary options, such as +SING +TRNS +VERB for a singular, transitive verb [10]. Semantic grammars [11] represent a hierarchy of concept types, such as BEAGLE<DOG, DOG<ANIMAL, and ANIMAL<ENTITY. Heidorn's NLP system [12] supports both a string of *indicator bits* for the syntactic features and a hierarchy of types for the semantics. ILIS follows NLP in maintaining two fields with each word: a 64-bit field of indicators for the syntax and a 16-bit integer for the semantic type.

The first four indicator bits are set by the tokenizer. They specify whether the word is purely alphabetic, alphanumeric, integer, real, punctuation, quoted string, or compound symbol like := or **. The other 60 bits can be named, set, and tested by the INDICATOR operator $I. For parsing English, the programmer may have named bit 17 VERB and bit 21 TRNS. Then the following operator tests whether the current word is a transitive verb:

$I+ VERB TRNS I$

The qualifier + after $I requires that all named bits must be 1; the other bits are ignored. The symbol I$ terminates the list of indicators. For words in the dictionary, the call $C=LOOKUP replaces a word with a definition that has set the appropriate bits. Since a word may have more than one part of speech, a word like "MOVE" would have bits for noun, verb, transitive, and intransitive.

Most programming languages require identifiers to be alphabetic or alphanumeric strings starting with a letter. The following rule matches any identifier:

RULE IDENT: $I− 1 2 3 I$.

Indicator bits that have not been named can be specified by their positions. This operator matches any word whose first three bits are zero. Since the tokenizer sets the first four bits to 0000 for purely alphabetic strings and 0001 for alphanumeric strings starting with a letter, this combination matches any valid identifier.

Matching a number that is either integer or real requires checking for two different patterns. The bit pattern is 1000 for integer and 1001 for real numbers. The following rule matches either:

RULE NUMBER: $O $I= 1 I$|$I=1 4 I$ O$.

The first option tests whether the first bit is 1 and all other bits are 0; the second option tests bits 1 and 4. Any combination of 0 and 1 bits can be tested by $I+ and $I− on the same word. Since every pattern match automatically advances the cursor to the next word, the YET AGAIN operator $Y must be used to move the cursor back for another test. The following rule could also be used to match numbers:

RULE NUMBER: $I+ 1 I$ $YI− 2 3 I$.

The operator $I+ tests for a 1 in the first position. Then the $I− operator tests whether bits two and three are 0. Without the intervening $Y, the operator $I+ would test the first word of input, and $I− would test the second word.

The type field is intended for user-defined types, as in Pascal or Ada, and for conceptual categories in natural languages. The programmer can specify a set of *type labels* with their position in the type hierarchy, such as DOG<ANIMAL or SHIP<MOBILE_ENTITY. ILIS then assigns a 16-bit integer to each label. The programmer does not refer to the internal integer, but to the symbolic label: The operator $T<SHIP tests for any subtype of SHIP, and $T<YACHT tests only for yachts.

Top-down vs bottom-up parsing

For the past 20 years, arguments over top-down vs bottom-up parsing have continued in the fields of compiler design and natural language processing. ILIS supports both approaches. Handel [3] used ILIS in a top-down mode for his English parser, and in RENDEZVOUS [6] it was used as a bottom-up production system. In the top-down parser, Handel required all words to be defined before parsing began; if a word was not in the dictionary, his system would ask the user for its part of speech. RENDEZVOUS was more flexible. It used associative indexing to find rules for appropriate semantic patterns. When it found an approximate match, it would fill in the slots of the pattern and ignore input words that did not fit. Since approximate matching is error-prone, RENDEZVOUS mapped its interpretation back into an English sentence and asked the user for confirmation. After a short dialog, the user and the system would converge on a complete and correct formulation of a database query.

Although Handel's parser was top-down, he did use associative indexing for idioms and abbreviations. The following rule, for example, matches input containing the word "DON'T," replaces it with "DO NOT," and uses the $P=* operator to PERFORM a rescan:

RULE (DON'T): $AK DON'T $RS DO NOT S$ $P=*.

After making all the substitutions with rules of this form, the system switched to a top-down mode for the actual parse. For programming languages, associative indexing would be useful for synonyms and macro expansions. After the macro stage, the actual parsing could be done in a top-down mode.

For programming languages, ILIS runs best in top-down mode with an LL(1) grammar [13]. For efficiency, the SPACEKOR analyzer looks ahead one word whenever it finds a list of options. If the correct choice is found [always possible with an LL(1) grammar] it takes that option without backtracking. It thereby achieves the speed of a purely deterministic parse, but with the option of backtracking to handle any context-free language.

Some translator writing systems use an LR parser generator, which supports a deterministic, bottom-up parse [14–16]. Although LR parsers can run about as efficiently as LL parsers, they cannot be generalized to a wider class of languages. Furthermore, LR parser generators cannot be as interactive as ILIS because they must process all the rules of a grammar at once in order to create their parsing tables. With ILIS, however, any rule can be added, deleted, or modified without affecting any other rules in the workspace.

Input and output formats

A system that handles all possible languages must be able to format those languages according to their standard conventions. For input, the ILIS tokenizer supports several options that can be changed by the SET command:

- Beginning and ending delimiters for comments and quoted strings,
- Distinctions between upper- and lower-case letters,
- Compound operators like := and ** that are tokenized as single words,
- Escape character for command strings,
- Special characters defined as alphabetic (especially important for non-English languages),
- Period or comma for the decimal point in real numbers,
- End of each input stream signaled by the end of line, special punctuation such as period or semicolon, or the end of an entire file.

For each word, the tokenizer sets the first four bits of the indicator field to show the token category, such as purely alphabetic, alphanumeric, integer, real, quoted string, punctuation, command string, compound operator, or other. One of the set options allows the tokenizer to be turned off completely so that SPACEKOR rules can parse everything. This option is the most general, but it is less efficient than allowing the tokenizer to group the input into tokens.

For output, the formatter converts a linked list of words into lines. It normally inserts a blank between words, but it suppresses the blank before a word with indicator bits for punctuation. The formatter also recognizes *formatting commands* embedded in the list. The command $B, for example, breaks to a new line, $W=n sets the line width to n, $F=n tabs forward to column n, $F+n tabs forward n spaces beyond the current position, and $Z suppresses blanks until the next formatting command. With the formatting commands, a *prettyprinter* can parse a language and insert line breaks and indentation to standardize the formatting.

The $X command for exact formatting positions each word according to its own line and column number. The input tokenizer sets the line and column number for each word as it maps the input string into a list. Then those numbers, which are carried through all the parsing and transformations, are used by the formatter to recreate a line that appears identical to the original. In SPACEKOR rules, the JUNCTURE operator $J can test or set the line and column numbers or copy them to the stack. Then a syntax checker can print an erroneous statement exactly as it appears in the input and generate a diagnostic message keyed to the position where the error was found.

Sample translation

A feature that distinguishes ILIS from most parsing systems is the ability to include action operators in the grammar rules. To illustrate the technique, consider the problem of translating assignment statements from a programming language into Polish notation. The first step is to write a grammar for the language. The following four rules define a statement STMT as an identifier IDENT followed by an assignment operator ":=", an expression EXPR, and a semicolon. An EXPR is a TERM followed by zero or more repetitions of "+" or "−" and another TERM. A TERM is a factor FAC followed by zero or more repetitions of "*" or "/" and another FAC. Finally, a FAC is an identifier, a number, or another expression in parentheses:

```
STMT  →  IDENT ":=" EXPR ";"
EXPR  →  TERM [("+"|"−") TERM] . . .
TERM  →  FAC [("*"|"/") FAC] . . .
FAC   →  IDENT|NUMBER|"(" EXPR ")"
```

These rules have an embedded recursion of EXPR inside of the rule for FAC, but they have no left recursions. The next step is to convert them to SPACEKOR rules:

```
RULE STMT (:=): $M=IDENT := $M=EXPR ; .
RULE EXPR: $M=TERM $L $O +|− O$
$M=TERM L$.
RULE TERM: $M=FAC $L $O *|/ O$ $M=FAC L$.
RULE FAC: $O $M=IDENT|$M=NUMBER|
               ( $M=EXPR ) O$.
```

The first rule can be called by an operator $M=STMT or $P=STMT, or it can be triggered associatively by the occurrence of ":=" in the input stream. The rules for IDENT and NUMBER were defined in earlier examples.

As the SPACEKOR rules recognize each type of phrase, the translated output can be built on the stack by the action operators. Words are transferred from the input to the stack with $K, subphrases are concatenated to form larger phrases with $S*n, and the interchanges for reverse Polish notation are done with $S%n. Following is a complete set of SPACEKOR rules for translating simple assignment statements into reverse Polish form:

```
RULE STMT (:=): $M=IDENT := $K $M=EXPR
        $S%2 ; $R.
RULE EXPR: $M=TERM $L $O +|− O$ $K
        $M=TERM $S%2 $S*3 L$.
RULE TERM: $M=FAC $L $O *|/ O$ $K $M=FAC
        $S%2  $S*3 L$.
RULE FAC: $O $M=IDENT|$M=NUMBER|( $R
        $M=EXPR ) $R O$.
RULE IDENT: $I− 1 2 3 I$ $K.
RULE NUMBER: $I+ 1 I$ $YI− 2 3 I$ $K.
```

If the user typed the following statement at the terminal,

```
XYZ := A*B + (C−D)*E;
```

these rules would generate the following output in reverse Polish form:

```
XYZ A B * C D − E * + :=
```

Design considerations

The original version of ILIS was designed as a pattern-directed mechanism for invoking PL/I procedures. It was intended to be fast and simple, and the programmer was expected to use PL/I for general computation. The original eight SPACEKOR operators did not include $M, which is essential for top-down parsing, but they could be used as a production system that was general enough to simulate a Turing machine. The SPACEKOR operators turned out to be so flexible that programmers chose to use them in preference to invoking external programs. When the additional operators were added, ILIS developed into a complete translation system. Margolis implemented a translator from a subset of Ada into Pascal without invoking any external programs other than those provided in the basic library.

As ILIS evolved, the original goal of simplicity was kept. If some feature, like arithmetic computation or array handling, could easily be done in PL/I, it was not added to ILIS. Some features, like input and output formatting, could be done in PL/I, but since they depended heavily on the internal formats of ILIS data, a complete set was added to ILIS. To keep the parser efficient, some implementation considerations affected the definition of the operators:

- All dynamic storage is maintained in fixed, 32-byte blocks. This includes the internal forms for words, the activation records for SPACEKOR rules, and the backtracking information for choice points.
- The SPACEKOR analyzer can always determine when a list can be returned to the free list. As a result, no garbage collection phase is needed.
- Backtracking does not do a perfect restoration of the environment: It just restores some pointers and erases new lists that may have accumulated on the stack. The programmer can freely add new data to the stack, but must postpone operators that change older data until the parsing has reached a known state.
- Since the arguments and results for SPACEKOR rules are passed on the stack, the MATCH operator requires a minimum of status saving. The SPACEKOR analyzer automatically optimizes tail recursions: If a MATCH operator occurs at the end of a rule or at the end of an option that occurs before the end of a rule, it is executed by a simple GO-TO instruction.

Acknowledgments

All the users of ILIS, especially the hundreds of students who did their homework with it, have helped to improve the system with their suggestions for enhancements and reports of difficulties. Two of the users have made extensive additions to the implementation: James Rhyne, who was working on the RENDEZVOUS project, made a number of extensions including a new method for indexing the rules; and Lawrence Margolis, at the IBM Systems Research Institute, has used ILIS in several projects and has implemented all the extensions and modifications since 1981. I would also like to thank George Heidorn for numerous comments on the content and presentation of this paper.

Appendix: Summary of SPACEKOR operators

Following is a list of all SPACEKOR operators. A literal string s that appears in a rule has exactly the same effect as the operator $A=s. In this summary, stack[n] represents the nth list on the stack counting from the end; stack[1] is the last list.

$A	Match ANY input list of zero or more words.
$A*n	Match ANY input list of exactly n words.
$A=s	Match ANY word exactly equal to string s.
$A:s	Match ANY word with initial letters equal to string s.
$C=p	CALL external procedure p.
$E	Succeed if END of input list has been reached.
$G=r	GET a copy of list from register r to stack[1].
$G%r	GET list from register r to stack[1] & clear r.
$G:r	GET only first word from register r to stack[1].
$H=r	HOLD a copy of stack[1] list in register r.
$H%r	HOLD stack[1] list in register r & pop stack.
$H:r	HOLD stack[1] concatenated with r in r & pop stack.
$H*r	HOLD r concatenated with stack[1] in r & pop stack.
$I= list I$	Check if INDICATORS in list are 1 & all others are 0.
$I+ list I$	Check if INDICATORS in list are 1.
$I− list I$	Check if INDICATORS in list are 0.
$I: list I$	Set INDICATORS of word in stack[1] to list.
$I; list I$	Merge INDICATORS of word in stack[1] with list.
$J	Transfer JUNCTURE line and column position to stack[1].
$J= n1 n2 J$	Check JUNCTURE at line and column position n1 n2.

$J< n1 n2 J$	Check JUNCTURE before line and column position n1 n2.
$J> n1 n2 J$	Check JUNCTURE after line and column position n1 n2.
$J: n1 n2 J$	Set JUNCTURE to line and column positions n1 n2.
$K	KEEP a copy of current input sublist in stack[1].
$L r L$	LOOP repeatedly until first failure in rule r.
$M=n	MATCH rule named n to input.
$N	Succeed if next test does NOT succeed.
$O opt \| opt \| . . . O$	List of OPTIONS tried from left to right.
$P rule P$	PERFORM rule with stack[1] as input.
$P=n	PERFORM rule named n with stack[1] as input.
$P=*	Concatenate stack to input and PERFORM some rule.
$Q	QUIT executing current rule & execute rule named QUIT.
$R	REJECT current input sublist & start new sublist.
$R=*	REJECT current input sublist & return it to free list.
$S list S$	Put the enclosed list of words on STACK.
$S*n	Concatenate STACK lists n, n − 1, . . . , to stack[1].
$S=n	Copy the nth list on STACK to stack[1].
$S=*	Copy the entire STACK to stack[1].
$S¬n	Delete the nth list on STACK.
$S%n	Interchange the nth STACK list with stack[1].
$T=t	Check if TYPE of word is identical to t.
$T<t	Check if TYPE of word is subtype of t.
$T>t	Check if TYPE of word is supertype of t.
$T:t	Set TYPE of first word in stack[1] to t.
$U	UNOPTION choice points for the current rule.
$U=*	UNOPTION choice points for all options not taken.
$V=r	VERIFY if stack[1] list equals list in register r.
$V:r	VERIFY if first word in stack[1] and register r are equal.
$V<r	VERIFY if stack[1] list is sublist of list in register r.
$Y	Move input cursor back for YET another match.

References

1. W. A. Woods, "Transition Network Grammars for Natural Language Analysis," *Commun. ACM* **13,** 591–606 (1970).
2. J. F. Sowa, *Conceptual Structures: Information Processing in Mind and Machine*, Addison-Wesley Publishing Co., Reading, MA, 1984.
3. J. L. Handel, "An English Parser," *Technical Report TR 73-005*, IBM Systems Research Institute, New York, 1980.
4. G. J. Meyers, *Advances in Computer Architecture*, John Wiley & Sons, Inc., New York, 1982.
5. D. G. Hocker, "AMAS: A Microcode Assembler for the SWARD Processor," *Technical Report TR 73-018*, IBM Systems Research Institute, New York, 1981.
6. E. F. Codd, R. S. Arnold, J.-M. Cadiou, C. L. Chang, and N. Roussopoulos, "RENDEZVOUS Version I: An Experimental English-Language Query Formulation System for Casual Users of Relational Data Bases," *Research Report RJ-2144*, IBM Research Division, San Jose, CA, 1978.
7. J. Murai, N. Saito, N. Doi, M. Morohashi, and T. Fujisaki, "Requirement Specification Description in Japanese Language, JISDOS," *Proceedings of 6th International Conference on Software Engineering*, IEEE Computer Society Press, New York, 1982, pp. 127–136.
8. *Compiler Construction*, F. L. Bauer and J. Eickel, Eds., Springer-Verlag New York, Inc., New York, 1976.
9. R. Davis and J. King, "An Overview of Production Systems," in *Machine Intelligence*, Vol. 8, E. W. Elcock and D. Michie, Eds., John Wiley & Sons, Inc., New York, 1977, pp. 300–332.
10. F. B. Thompson and B. H. Thompson, "Practical Natural Language Processing: The REL System as Prototype," in *Advances in Computers*, Vol. 13, M. Rubinoff and M. C. Yovits, Eds., Academic Press, Inc., New York, 1975, pp. 109–168.
11. J. S. Brown and R. R. Burton, "Multiple Representations of Knowledge for Tutorial Reasoning," in *Representation and Understanding*, D. G. Bobrow and A. Collins, Eds., Academic Press, Inc., New York, 1975, pp. 311–349.
12. G. E. Heidorn, "Natural Language Inputs to a Simulation Programming System," *Report NPS-55HD72101A*, Naval Postgraduate School, Monterey, CA, 1972.
13. M. Griffiths, "LL(1) Grammars and Analyzers," in *Compiler Construction*, F. L. Bauer and J. Eickel, Eds., Springer-Verlag New York, Inc., New York, 1976, pp. 57–84.
14. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley Publishing Co., Reading, MA, 1977.
15. J. Cohen and M. S. Roth, "Analyses of Deterministic Parsing Algorithms," *Commun. ACM* **21,** 448–458 (1978).
16. J. J. Horning, "LR Grammars and Analyzers," in *Compiler Construction*, F. L. Bauer and J. Eickel, Eds., Springer-Verlag New York, Inc., New York, 1976, pp. 85–108.

Received August 25, 1983; revised September 15, 1983